Computing Distributed Representations for Polysemous Words

Master Thesis
Software Systems Engineering

Haiqing Wang
Matriculation number 340863

July 6, 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

Mathematical Symbols and Acronyms

C The given corpus containing the sentences/documents of words. 2, 8–11, 13, 20

c The size of a context $Context^{n-1}(w_t)$, i.e. the number of words before and after w_t. 24

$Context^{n-1}(w_t)$ The context of a word w_t in the sentence S_i may be defined as the subsequence of the words $Context^{n-1}(w) = (w_{\max(t-n+1,1)}, \ldots, w_{t-1})$, and $n-1$ is the number of words before w_t. 10, 20

$Context(w_t)$ The context of a word w_t in the sentence S_i may be defined as the subsequence of the words $Contex(w_t) = (w_{i,\max(t-c,0)}, \ldots, w_{i,t-1}, w_{i,t+1}, \ldots, w_{i,\min(t+c,L_i)})$. 24

D The vocabulary, i.e. the set of N different words w in the corpus C. 2, 7, 9–13, 20

d The length of the embedding vector $v(w) \in \Re^d$, e.g. $d = 100$. 2, 10

K The number of negative samples randomly generated for a word w_t. 23

L_i The number of words in the i-th sentence of the corpus C, $S_i = (w_{i,1}, w_{i,2}, \ldots, w_{i,L_i})$. 2

M The number of sentences S_i in the corpus, $C = (S_1, \ldots, S_M)$. 23

N The number of different words w in the corpus C, usually $N \geq 100.000$. 2, 9, 11–13, 20

N_w The number of different senses of a words w in the corpus C. 23

$phrase(w_t)$ A phrase with the center word w_t from the sentence S_i may be defined as the subsequence of the words $phrase(w_t) = (w_{t-c}, \ldots, w_{t-1}, w_t, w_{t+1}, \ldots, w_{t+c})$. 11, 12

S_i The i-th sentence of the corpus C, $S_i = (w_{i,1}, w_{i,2}, \ldots, w_{i,L_i})$. 2

$U_{w,s}$ The d-dimensional output embedding $U_{w,s} \in \Re^d$ corresponding to the sense $s \in \{1, \ldots, N_w\}$ of word $w \in D$. 23

$v(w)$ The d-dimensional embedding $v(w) \in \Re^d$ corresponding to a word $w \in D$. 2

$V_{w,s}$ The d-dimensional input embedding $V_{w,s} \in \Re^d$ corresponding to the sense $s \in \{1, \ldots, N_w\}$ of word $w \in D$. 23

$w_{i,j}$ The j-th word $w_{i,j} \in D$ in sentence S_i. 23

List of Abbreviations and Symbols

C The given corpus containing the sentences/documents of words. 2, 8–11, 13, 20

c The size of a context $Context^{n-1}(w_t)$, i.e. the number of words before and after w_t. 24

$Context^{n-1}(w_t)$ The context of a word w_t in the sentence S_i may be defined as the subsequence of the words $Context^{n-1}(w) = (w_{\max(t-n+1,1)}, \ldots, w_{t-1})$, and $n-1$ is the number of words before w_t. 10, 20

$Context(w_t)$ The context of a word w_t in the sentence S_i may be defined as the subsequence of the words $Contex(w_t) = (w_{i,\max(t-c,0)}, \ldots, w_{i,t-1}, w_{i,t+1}, \ldots, w_{i,\min(t+c,L_i)})$. 24

D The vocabulary, i.e. the set of N different words w in the corpus C. 2, 7, 9–13, 20

d The length of the embedding vector $v(w) \in \Re^d$, e.g. $d = 100$. 2, 10

K The number of negative samples randomly generated for a word w_t. 23

L_i The number of words in the i-th sentence of the corpus C, $S_i = (w_{i,1}, w_{i,2}, \ldots, w_{i,L_i})$. 2

M The number of sentences S_i in the corpus, $C = (S_1, \ldots, S_M)$. 23

N The number of different words w in the corpus C, usually $N \geq 100.000$. 2, 9, 11–13, 20

N_w The number of different senses of a words w in the corpus C. 23

$phrase(w_t)$ A phrase with the center word w_t from the sentence S_i may be defined as the subsequence of the words $phrase(w_t) = (w_{t-c}, \ldots, w_{t-1}, w_t, w_{t+1}, \ldots, w_{t+c})$. 11, 12

S_i The i-th sentence of the corpus C, $S_i = (w_{i,1}, w_{i,2}, \ldots, w_{i,L_i})$. 2

$U_{w,s}$ The d-dimensional output embedding $U_{w,s} \in \Re^d$ corresponding to the sense $s \in \{1, \ldots, N_w\}$ of word $w \in D$. 23

$v(w)$ The d-dimensional embedding $v(w) \in \Re^d$ corresponding to a word $w \in D$. 2

$V_{w,s}$ The d-dimensional input embedding $V_{w,s} \in \Re^d$ corresponding to the sense $s \in \{1, \ldots, N_w\}$ of word $w \in D$. 23

$w_{i,j}$ The j-th word $w_{i,j} \in D$ in sentence S_i. 23

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. organization of the thesis

1.1 Background

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.

Consider a corpus C of interest containing documents and sentences. Traditional text mining analyses use the vector space representation [Salton et al., 1975], where a word w

is represented by a sparse vector of the size N of the vocabulary D (usually $N \geq 100.000$), 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 w as a real vector of d (e.g. $d = 100$) real numbers as proposed by [Collobert and Weston, 2008] and [Mikolov et al., 2013]. Generally, we call such a vector $v(w) \in \Re^k$ 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]. Consider a sentence (or document) S_i in the corpus C consisting of L_i words $S_i = (w_{i,1}, w_{i,2}, \ldots, w_{i,L_i})$. Then the context of a word w_t may be defined the words in the neighborhood of w_t in the sentence. [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.

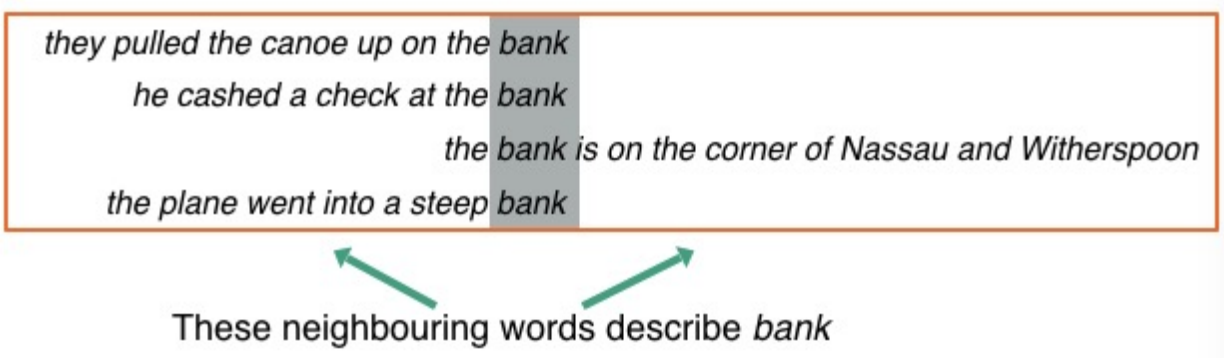

Figure 1.1: Neigboring words defining the specific sense of "bank".

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

[1] COMMENT: please more details on this

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 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.

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.

[2] COMMENT: I do not understand that

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 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.

1.2 Project

sss

1.3 Goal

5

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. [6] Our main work will focus on

[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

[4] COMMENT: Please reformulate sentence

[5] COMMENT: about how to solve the problem

[6] 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.

the extension of Skip-gram model [Mikolov et al., 2013] in connection to the approach of [Neelakantan et al., 2015a] because these models are easy to use, very efficient and convenient to train. [7] 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.
[8]

1.4 Organization

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 embeddings. 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.
[9]

[7] 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.

[8] COMMENT: Improvement: better speed and better similarity

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

Chapter 2

Background

2.1 Word Embedding

Recently machine learning algorithms are used often in many NLP tasks, but the machine can not directly understand human language, so the first thing is to transform the language to the mathematical form like word vectors, that is using digital vectors to represent words in natural languages. The above process is called word embedding.

One of the easiest word embedding is using one-hot representation, which is to use a long vector to represent a word. 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 D. 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 embedding is Distributed Representation, it was firstly proposed by Williams and Hinton [1986], 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 vector can be regarded as 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.

There are many different models can be used to estimate the word vector, including the famous LSA (Latent Semantic Analysis) and LDA (Latent Dirichlet Allocation). In addition, the neural network algorithm based language model is a very common method and becomes more and more popular. In these neural network based models, the mainly goal is to generate a language model, in the meantime, they get the word vectors. In fact, in most cases, the word vector and the language model are bundled together. After the training, we can get both. The most classical paper in this aspect is the Neural Probabilistic Language Model from [Bengio et al., 2003], followed by a series of related research, including SENNA from [Collobert et al., 2011] and word2vec from from [Mikolov et al., 2013].

Word embedding actually is very useful, for example, Ronan Collobert's team makes use of the word vectors trained 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.

2.2 Neural Probabilistic Language Model

Statistical Language Model

Statistical language model is widely used in speech recognition, machine translation, word segmentation, POS tagging and information retrieval.

Statistical language model is a Probability Model to calculate the probability of an sentence or word sequence, which is always built on a corpus C build up by several sentences. Most of statistic language models build models to maximize objective function like

$$\prod_{w \in C} p(w|Context(w)). \tag{2.1}$$

where $w \in C$ means w is in some sentence from corpus C, $Context(w)$ represents the context of w, that is the set of words surrounding w, which can be a complete sentence or just word sequence.

Neural Network

General speaking, a neural network defines a mapping between some input and output, and it usually contains a input layer, an output layer and several hidden layers between the input layer and the output layer. Each layer has some nodes meaning that it can contains a vector with some dimension. From one layer to another layer, there is a map function made up by an active function with a weight matrix and and an offset vector. Specifically a vector multiplies a weight matrix and then pluses an offset vector, after that an active function such as $tanh$ function shown as Figure 2.1, where $tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$, will apply on that vector to normalize each element between 0 to 1. To be noted that the last map function from the last hidden layer to output layer is special, it usually has no active function and will use some other normalization function to achieve different goals.

Let's start with the simplest neural network with three layers (only one hidden layer) shown as Figure 2.2. We can find that input layer has two nodes, that is the input should be a vector with dimension 3. Hidden layer has four nodes, which means the input vector will be mapped to another vector with dimension 4. And then it will be mapped again to the output vector with dimension 2. Specifically, the input is a vector x (with dimension 3). H (with size 3×4) and U (with size 4×2) are respectively the weight matrix between input layer and hidden layer and the weight matrix between hidden layer and output layer, p (with size 4) and q (with size 2) are the offset vectors of respectively the hidden layer

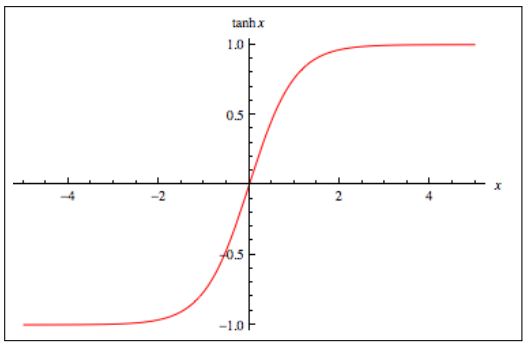

Figure 2.1: *tanh* function

and the output layer. The active function is the *tanh* function and the output vector is y (with dimension 2). So we can have the following formula

$$y = q + Utanh(p + Hx) \tag{2.2}$$

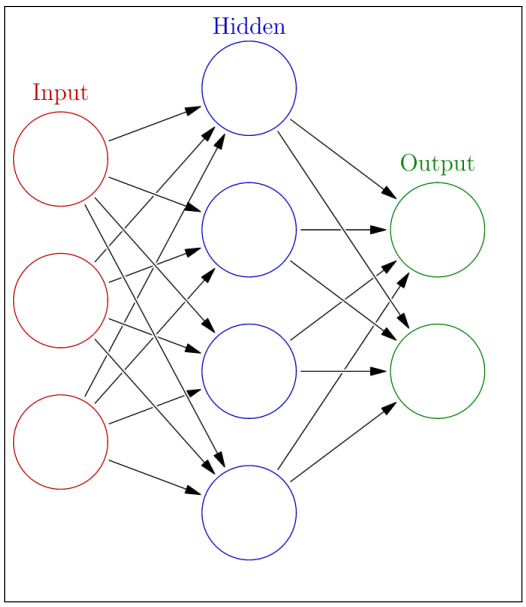

Figure 2.2: An example of neural network with three layers

Neural Probabilistic Language Model

Bengio et al. [2003] introduce a neural probabilistic language model shown as Figure 2.3, which uses neural network and is based on the language model. Such language model also uses Formula 2.1 as the objective function and build a neural network on this language model.

C is the given corpus containing the sentences/documents of words. N is the number of different words in the corpus C. D is the vocabulary (the set of N different words in

the corpus C). Considering a word w_t in some sentence $S = (w_1, w_2, \ldots, w_{T-1}, w_T)$ from corpus C, where t is some position of S and T is the length of S, define $Context^{n-1}(w_t) = (w_{\max(t-n+1,1)}, \ldots, w_{t-1})$, and $n-1$ is the number of words before w_t. Firstly for each word w in dictionary D, there is a look-up table C [1] mapping the word w to vector $C(w)$. The vector size is d. The input layer is a long vector catenated by $n-1$ word vectors. The matrix C in the Figure is used to map the word index to the word vector. So the input vector is x with dimension $(n-1)d$, and the output vector is y with the dimension $|D|$, where D is vocabulary and $|D|$ is the the of vocabulary. And they also use $tanh$ function for the active function in hidden layer. H (with size $(n-1)d \times m$) and U (with size $m \times |D|$) are respectively the weight matrix between input layer and hidden layer and the weight matrix between hidden layer and output layer, p (with size m)and q (with size D) are the offset vectors of respectively the hidden layer and the output layer. Additionally, they introduce another weight matrix between the input layer and output layer W (with size $(n-1)d \times |D|$). So The mapping function from input to output is

$$y = q + Wx + Utanh(p + Hx) \tag{2.3}$$

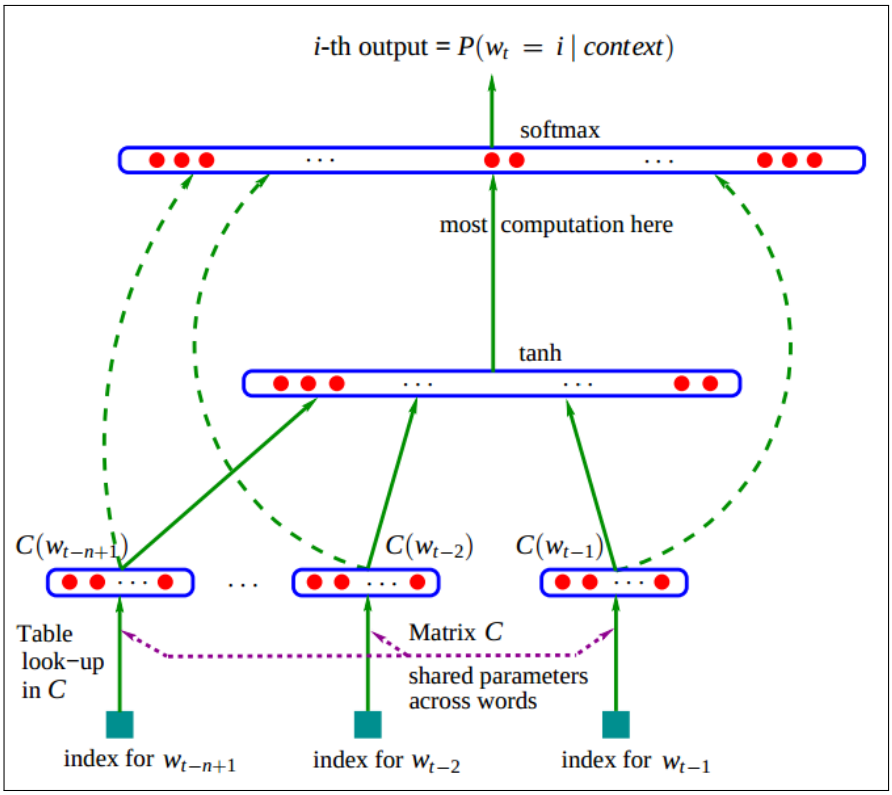

Figure 2.3: The neural network structure from [Bengio et al., 2003]

[1] COMMENT: here should be another letter different from C

Softmax Function

The softmax function, is a generalization of the logistic function that "squashes" a K-dimensional vector z of arbitrary real values to a K-dimensional vector $\sigma(z)$ of real values in the range $(0, 1)$ that add up to 1. [From wikipedia] The function is given by

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

From above, we know the output y is a vector with the length of $|D|$ and can not represent probabilities. Because it is a language model, it needs to model the probability as $p(w_t|Context^{n-1}(w_t))$. Actually, they uses the softmax function to do normalization. After normalization, the final result is a value between 0 to 1, which can represent a probability. Using x_{w_t} to represent the input vector connected by word vectors from $Context(w_t)$ and y_{w_t} to represent the output vector mapped from the neural network. From Formula 2.3 we have

$$y_{w_t} = b + Wx_{w_t} + Utanh(d + Hx_{w_t})$$

and $p(w_t|Context^{n-1}(w_t))$ can be expressed as

$$p(w_t|Context^{n-1}(w_t)) = \frac{e^{y_{w_t,i_{w_t}}}}{\sum_{i=1}^{|D|} e^{y_{w_t,i}}}, \tag{2.4}$$

where i_{w_t} represents the index of w_t in the dictionary D, $y_{w_t,i}$ means the i-th element in the vector y_{w_t}. Note that the denominator contains a term $e^{y_{w_t,i}}$ for every word in the vocabulary. The goal is also to maximize the function shown as 2.1. In the beginning, all word vectors are initialized randomly. And after maximizing the objective function, they can get the meaningful word vectors.

2.3 The model of Collobert and Weston

The main purpose from Collobert and Weston [2008] originally is not to build a language model, but to use the word vectors from their model 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. They do not use the Formula 2.1 as other language models used. Their idea is to optimize a score function on phrase so that if the phrase is reasonable or makes sense, the score would be positive, otherwise the score would be negative.

C is the given corpus containing the sentences/documents of words. N is the number of different words in the corpus C. D is the vocabulary (the set of N different words in the corpus C). Consider a word w_t in some sentence $S = (w_1, w_2, \ldots, w_{T-1}, w_T)$ from corpus C, where t is some position of S and T is the length of S. Define $phrase(w_t) =$

$(w_{t-c}, \ldots, w_{t-1}, w_t, w_{t+1}, \ldots, w_{t+c})$, and c is the number of words before and after w_t. Note that $phrase(w_t)$ contains w_t. Replace the center word w_t in $phrase(w_t)$ to anther random word w' and use $phrase(w_t)'$ to represent it. Specifically, $phrase(w_t)' = (w_{\max(t-c)}, \ldots, w_{t-1}, w', w_{t+1}, \ldots, w_{t+c})$, where w' is selected randomly from vocabulary D but different with w_t. Each word is represented by a vector with dimension d . For phrase $phrase(w_t)$, connect these $2c+1$ vectors to be a long vector x_{w_t} with dimension $d \times (2c+1)$. The input of f is the vector x_{w_t} with dimension $d \times (2c+1)$. And the output is a real number (positive or negative). Use x'_{w_t} to represent the vector connected by $2c+1$ word vectors from $phrase(w_t)'$.

They also use a neural network to build their model. And the neural network structure is similar as the network structure from [Bengio et al., 2003]. The same thing is connecting several word vectors together to get a long vector as the input. The difference is that the output layer has only one node representing the score, rather than Bengio's N nodes, where N is the size of dictionary D. Note that Bengio's model uses another softmax funtion to get the probability value in order to represent the Formula 2.4. Doing so greatly reduced the computational complexity.

Based on the above description, the model use f to represent its neural network, the input is a phrase vector, the output can be an arbitrary real number. Note that there is no active function in the output layer. The objective of this model is that for every phrase $phrase(w_t)$ from corpus, $f(x_{w_t})$ should be always positive and $f(x'_{w_t})$ should be always negative. Specifically the model gives an objective function to be minimized as following

$$\sum_{w \in C} max\{0, 1 - f(x_w), f(x'_w)\} \tag{2.5}$$

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).

2.4 Word2Vec

The main purpose of Word2Vec is to accelerate the training process and simplify the model. The models from previous sections really needs much time to train especially on softmax function calculation. So the greatest contribution of Word2Vec is to introduce two approximation probability calculation methods to replace the softmax function.

Word2Vec actually contains two different models: the CBOW model (Continuous Bag-of-Words Model), which also uses function 2.1 as the objective function like, and the Skip-gram model(Continuous Skip-gram model), which uses another objective function as following

$$\prod_{w \in C} p(Context(w)|w), \tag{2.6}$$

C is the given corpus containing the sentences/documents of words. N is the number of different words in the corpus C. D is the vocabulary (the set of N different words in the corpus C). Considering a word w_t in some sentence $S = (w_1, w_2, \ldots, w_{T-1}, w_T)$ from corpus C, where t is some position of S and T is the length of S, they define $Context(w) = (w_{,\max(t-c,1)}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{,\min(t+c,T)})$, and c is the number of words before and after w_t in the $Context(w)$. The Figure 2.4 shows the structures of CBOW model and Skip-gram model when $c = 2$.

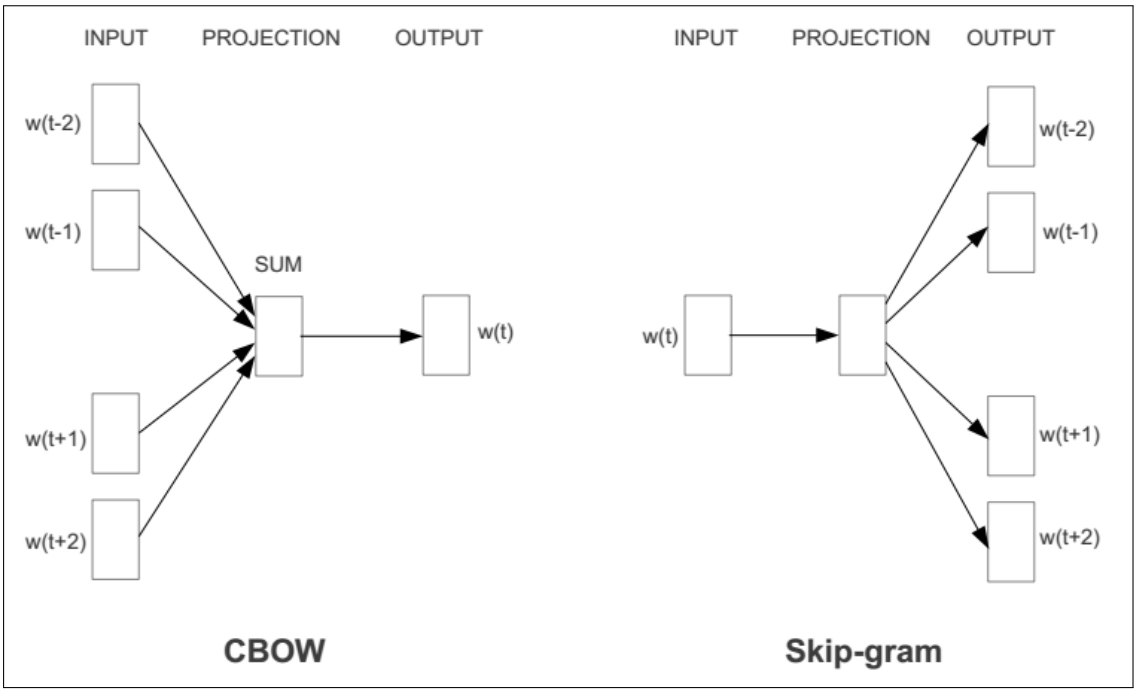

Figure 2.4: word2vec

In both neural networks from CBOW and Skip-gram model, the input is a vector with length d, for CBOW the input is the context vector calculated by average vector of word in the context, for skip-gram model the input is the word vector of w_t. And the output is the vector of length N, N is the size of the vocabulary. The difference is that their prediction strategies are different. CBOW is to use context to predict the center word. And Skip-gram model is to use the center word to predict several words in the context.

Skip-gram model with negative sampling

Next we will focus on Skip-gram model with negative sampling to optimize the objective function as Formula 2.5. Let V and U 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, $V_w \in \Re^d$ means the input embedding vectors from word w. Similarly $U_w \in \Re^d$ is the output embedding of word w where $w \in D$, $1 \leq s \leq N_w$. The number of negative samples is K. And $P(w)$ is the smoothed unigram distribution which is used to generate negative samples. Specifically, $P(w) = \frac{count(w)^{\frac{3}{4}}}{(\sum_{i=1}^{M} L_i)^{\frac{3}{4}}}$ $(w \in D)$, 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. The objective function of skip-gram model with negative sampling can be defined specifically as

$$G = \frac{1}{M}\sum_{i=1}^{M}\frac{1}{L_i}\sum_{t=1}^{L_i}\sum_{\substack{-c \leq j \leq c \\ j \neq 0 \\ 1 \leq j+t \leq L_i}}\left\{\log\, p(w_{i,t+j}|w_{i,t}) + \sum_{k=1}^{K}\mathbb{E}_{z_k \sim P(w)}\log\,[1 - p(z_k|w_{i,t})]\right\} \tag{2.7}$$

where $p(w'|w) = \sigma({U_{w'}}^{\mathrm{T}} V_w)$ and $\sigma(x) = \frac{1}{1+\mathrm{e}^{-x}}$.

$p(w_{i,t+j}|w_{i,t})$ is the probability of using center word $w_{i,t}$ to predict one surrounding word $w_{i,t+j}$, which needs to be **maximized**. $z_1,\ldots,z_K$ are the negative sample words to replace word $w_{i,t+j}$, and $p(z_k|w_{i,t})$ $(1 \leq k \leq K)$ is the probability of using center word $w_{i,t}$ to predict one negative sample word z_k, which needs to be **minimized**. Equivalently, the whole objective function needs to be **maximized**.

Take $(w_{i,t}, w_{i,t+j})$ as a training sample, and define **loss function** $loss$ for each sample

$$\begin{aligned} &loss(w_{i,t}, w_{i,t+j}) \\ &= -\log\, p(w_{i,t+j}|w_{i,t}) - \sum_{k=1}^{K}\mathbb{E}_{z_k \sim P(w)}\log\,[1 - p(z_k|w_{i,t})] \end{aligned}$$

Here the loss is defined as the negative log probability.

And the loss function of whole corpus is

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

To maximize the objective function is equivalently to minimize the loss function. So the objective of learning algorithm is

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

Use

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

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.) And the number of epochs is T. So the total iterations is $N * T$.

Use stochastic gradient descent:

- Initialize $\{V, U\}$
- For $N * T$ Iterations:
  - For each training sample $(w_{i,t}, w_{i,t+j})$
    - Generate negative sample words to replace $w_{i,t+j}$: $(w_1, \ldots, z_k)$
    - Calculate the gradient $\Delta = -\nabla_{\{V,U\}} loss(w_{i,t}, w_{i,t+j})$
    - Δ is only made up by $\{\Delta_{V_{w_{i,t}}}, \Delta_{U_{w_{i,t+j}}}, [\Delta_{U_{w_1}}, \ldots, \Delta_{U_{z_k}}]\}$
    - 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)

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

$$\Delta_{V_{w_{i,t}}} = -\frac{\partial loss(w_{i,t}, w_{i,t+j})}{\partial V_{w_{i,t}}} = [1 - \log\,\sigma(U_{w_{i,t+j}}^{\mathrm{T}} V_{w_{i,t}})]U_{w_{i,t+j}} + \sum_{i=1}^{k}[-\log\,\sigma(U_{z_k}^{\mathrm{T}} V_{w_{i,t}}))]U_{z_k}$$

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

$$\Delta_{U_{z_k}} = -\frac{\partial loss(w_{i,t}, w_{i,t+j})}{\partial U_{z_k}} = [-\log\,\sigma(U_{z_k}^{\mathrm{T}} V_{w_{i,t}}))]V_{w_{i,t}}, 1 \le k \le K$$

Chapter 3

Related Works

3.1 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. For each center word, he took 10 nearest context words and calculated the

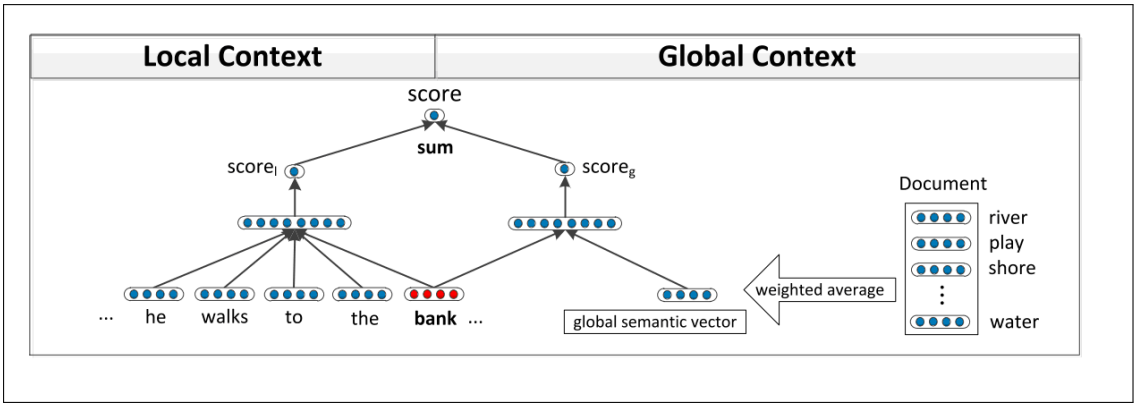

Figure 3.1: The network structure from [?]

Table 3.1: 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

weighted average of the embeddings of these 10 word vectors (idf weights) as the context vector. Huang used all context vectors to do a k-means clustering. He relabel each word based on the clustering results (different classes of the same words would be considered as different words to process)[1]. Finally he re-trained the word vectors. Table 3.1 gives some examples from his model's results.
[2]

3.2 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

[1] COMMENT: How was this exactly done?
[2] COMMENT: The paper does not give the details of clustering

Table 3.2: 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

sense in the corpus. For example, considering $bank_1$ in the sense of "side of the river" and $bank_2$ 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 3.2 lists some results from this model.

3.3 A Method to Determine the Number of Senses

Neelakantan et al. [2015b] comes up with two different models, the first one is the MSSG (Multi-Sense Skip-gram) model, in which the number of senses for each word is fixed and decided manually. The second one is NP-MSSG (Non-Parametric MSSG) which is based on the MSSG model , the number of senses for each word in this model is not fixed and can be decided dynamically by the model itself.

MSSG

C is the given corpus containing the sentences/documents of words. N is the number of different words in the corpus C. D is the vocabulary (the set of N different words in the corpus C). Considering a word w_t in some sentence $S = (w_1, w_2, \ldots, w_{T-1}, w_T)$ from corpus C, where t is some position of S and T is the length of S, define $Context(w) = (w_{,\max(t-c,1)}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{,\min(t+c,T)})$, and c is the number of words before and after w_t in the $Context(w)$.

In the MSSG model, each word has K senses (K is set advance manually.[3] Like huang's model, MSSG model uses the clustering, but its clustering strategy is different. Assuming each word has a context vector, which is summed up by all word vectors in the context. Huang's model do clustering on all context vectors (calculated by the weighted average of word vectors) from the corpus. While for MSSG model, they do clustering based on each word, that is each word has its own context clusters. Another thing is that MSSG only records the information (vector) of context cluster center for each word. For example, word w has K clusters , it has only K context vectors. And in the initialization, these K context vectors are set randomly. When there is a new context of word w, the model will firstly check which cluster this context should belong to and then use this new context vector to update the vector of selected context cluster center.

Specifically, each word has a global vector, K sense vectors and K context cluster center vectors. All of them are initialized randomly. For word w_t, its global vector is $v_g(w_t)$ and its context vectors are $v_g(w_{t-c}), \ldots, v_g(w_{t-1}), v_g(w_{t+1}), \ldots, v_g(w_{t+c})$. $v_{context}(c_t)$ is the average context vector calculated by the average of these $2c$ global word vectors. And w_t's sense vectors are $v_s(w_t, k)(k = 1, 2, \ldots, K)$ and the context cluster center vectors are $\mu(w_t, k)(k = 1, 2, \ldots, K)$. The architecture of the MSSG is like Figure 3.2 when $c = 2$. It uses the following formula to select the best cluster center:

$$s_t = \arg \max_{k=1,2,\ldots,K} sim(\mu(w_t, k), v_{context}(c_t))$$

[3] COMMENT: here should not use K

where s_t is index of selected cluster center and sim is the cosine similarity function. The selected context cluster center vector $\mu(w_t, s_t)$ will be updated with current context vector $v_{context}(c_t)$. And then the model selects $v_(w, s_t)$ as current word's sense. The rest thing is similar as skip-gram model: use this sense vector to predict global word vectors in the context and then update these global word vectors and this sense vector.

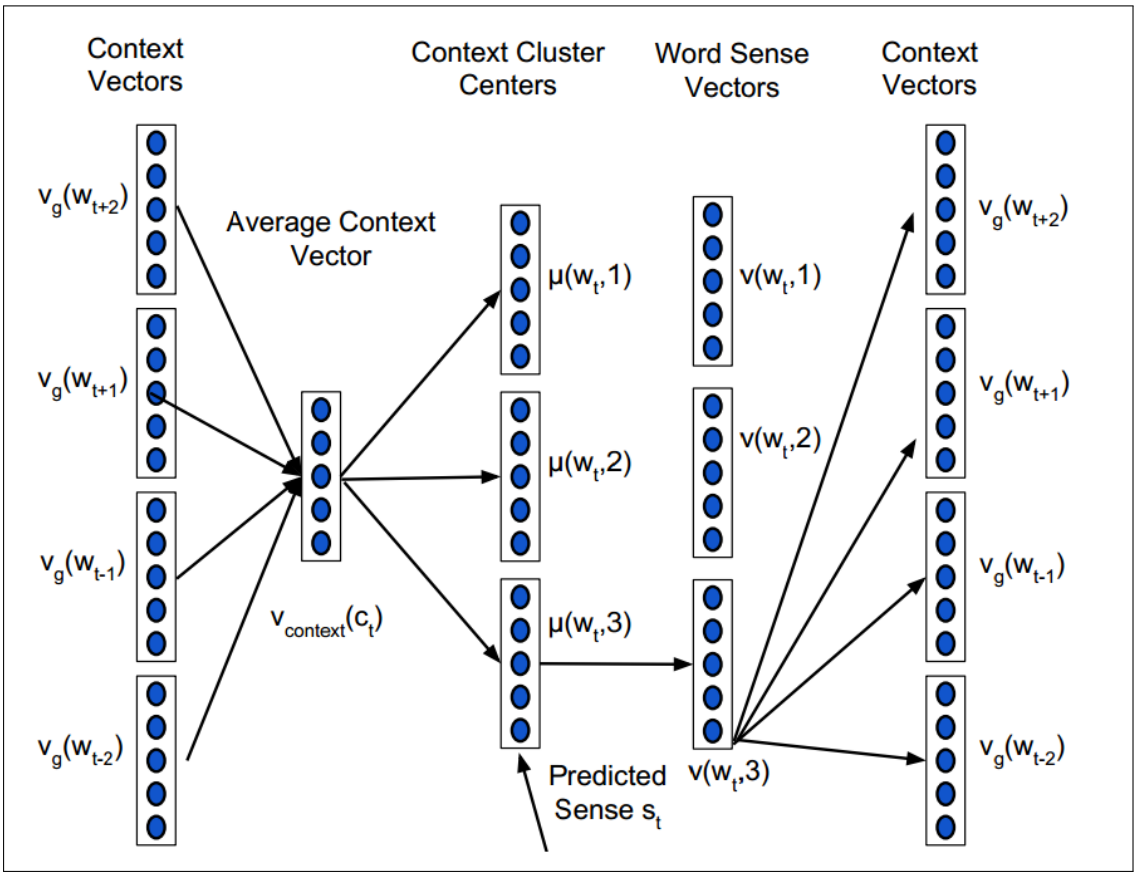

Figure 3.2: Architecture of MSSG model with window size $R_t = 2$ and $K = 3$

NP-MSSG

Unlike MSSG, in NP-MSSG model, the number of senses for each word is unknown and is learned during training. In the beginning, each word only has a global vector and does not have sense vectors and context clusters. When meeting a new context, for each word, the model will decide whether to create a new sense vector and a context cluster. When a word meets the first context, that is the first occurrence of this word in the corpus, the model creates the first sense vector and the first context cluster for this word. After that, when there is a new context, the model will calculate all similarities between this word and current all context clusters, if the biggest similarity value is smaller than some given hyper-parameter λ, that is the new context is different from all current context clusters,

the model will create a new context cluster and also a new sense vector as the beginning step. Otherwise, it will select the context with the biggest similarity value and do the same thing as what MSSG model do . Specifically, the selecting step can be described as the following formula :

$$s_t = \begin{cases} k(w_t) + 1, & \text{if } \max\limits_{k=1,2,...,k(w_t)} \{sim(\mu(w_t, k), v_{context}(c_t))\} < \lambda \\ k_{max}, & otherwise \end{cases}$$

where $k(w_t)$ is the number of senses (context clusters) of word w_t, $u(w_t, k)$ is the cluster center of k^{th} cluster of word w_t and $k_{max} = \arg\max_{k=1,2,...,k(w_t)} sim(\mu(w_t, k), v_{context}(c_t))$.

Chapter 4

Solution

In this section we present a model for the automatic generation of embeddings for the different senses of words. 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 more senses. As described above Huang et al. [2012] cluster the embeddings of word contexts to label word senses and once assigned, these senses can not be changed. Our model is different. We do not assign senses to words in a preparatory step, instead we just initialize each word with random senses and they can be adjusted afterwards. We also follow the idea from EM-Algorithm based method [Tian et al., 2014], 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.

4.1 Definition

C is the corpus containing M sentences, like $(S_1, S_2, \ldots, S_M)$, and each sentence is made up by several words like $S_i = (w_{i,1}, w_{i,2}, \ldots, w_{i,L_i})$ where L_i is the length of sentence S_i. We use $w_{i,j} \in D$ to represent the word token from the vocabulary D in the position j of sentence S_i. We assume that each word $w \in D$ in each sentence has $N_w \geq 1$ senses. We use the lookup function h to assign senses to words in a sentence, specifically $h_{i,j}$ is the sense index of word $w_{i,j}$ ($1 \leq h_{i,j} \leq N_{w_{i,j}}$).

Similar to Mikolov et al. [2013] we use two different embeddings for the input and the output of the network. Let 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, $V_{w,s} \in \Re^d$ means the input embedding vectors from sense s of word w. Similarly $U_{w,s} \in \Re^d$ is the ouput embedding of word w where $w \in D$, $1 \leq s \leq N_w$. Following the Skip-gram model with negative sampling, K. The context of a

word w_t in the sentence S_i may be defined as the subsequence of the words $Context(w_t) = (w_{i,\max(t-c,0)}, \ldots, w_{i,t-1}, w_{i,t+1}, \ldots, w_{i,\min(t+c,L_i)})$, where c is the size of context. And $P(w)$ is the smoothed unigram distribution which is used to generate negative samples. Specifically, $P(w) = \frac{count(w)^{\frac{3}{4}}}{(\sum_{i=1}^{M} L_i)^{\frac{3}{4}}}$ $(w \in D)$, 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.

4.2 Objective Function

Based on the skip-gram model with negative sampling. We still use same neural network structure to optimize the probability of using the center word to predict all words in the context. The difference is that, such probability is not about word prediction, instead it is about sense prediction.

[1] [2]

$$G = \frac{1}{M}\sum_{i=1}^{M}\frac{1}{L_i}\sum_{t=1}^{L_i}\sum_{\substack{-c \le j \le c \\ j \neq 0 \\ 1 \le j+t \le L_i}} \left(\log p\Big[(w_{i,t+j}, h_{i,t+j})|(w_{i,t}, h_{i,t})\Big] + \sum_{k=1}^{K} \mathbb{E}_{z_k \sim P_n(w)} \log \Big\{1 - p\Big[[z_k, R(N_{z_k})]|(w_{i,t}, h_{i,t})\Big]\Big\}\right) \tag{4.1}$$

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**. $[z_1, R(N_{z_1})],\ldots,[(z_K, R(N_{z_K})]$ 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**.

[1] COMMENT: What is this objective function for? Why is it defined?
[2] COMMENT: What is $(w_{i,t+j}, h_{i,t+j})$

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

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

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

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

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

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:

[3] COMMENT: Is this for sense assignment?

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

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

[4]

4.3 Algorithm 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.

[4] COMMENT: You could use the algorithmic package

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** ▷ Loop over sentences.
  repeat
    for t:= 1 TO L_i **do** ▷ Loop over words.
      $h_{i,t} = \max_{1 \leq s \leq N_{w_{i,t}}} f_{i,t}(s)$
    end for
  until no $h_{i,t}$ changed
end for

Learn:

for i:= 1 TO M **do** ▷ Loop over sentences.
  for t:= 1 TO L_i **do** ▷ Loop over words.
    for FOR j:= $-c$ TO c **do**
    end for
  end for
end for

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

$\Delta = -\nabla_\theta loss\big((w_{i,t}, h_{i,t}), (w_{i,t+j}, h_{i,t+j})\big)$

Δ 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

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

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

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

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

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

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

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 5

Implementation

For the implementation of our algorithm, we use the distributed framework Apache Spark[1]. 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.

5.1 Introduction of Spark

Spark was developed by Zaharia et al. [2010] and has many useful features for the parallel execution of programs. As the basic datastructure it has the RDD (Resilient Distributed Dataset). The RDD is a special data structure containing the items of a dataset, e.g. sentences or documents. Spark automatically distributes these items over a cluster of compute nodes and manages its physical storage.

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 fetches part of data and deals with it, and then the driver collects data from all executors.

The Spark operations can be called from different programming languages, e.g. Python, Scala, Java, and R. For this thesis we use Scala to control the execution of Spark and to define its operations.

Firstly, Spark reads text file from file system (e.g. from the unix file system or from HDFS, the Hadoop file system) and creates an RDD. An RDD usually is located in the RAM storage of the different executors, but it may also stored on (persisted to) disks of the executors. Spark operations follow the functional programming approach. There are

[1] http://spark.apache.org/

two types of operations on RDDs: *Transformation operations* and *action operations*. A transformation operation transforms a RDD to another RDD. Examples of transformation operations are map (apply a function to all RDD elements), filter (select a subset of RDD elements by some criterion), or sort (sort the RDD items by some key). Note that an RDD is not mutable, i.e. cannot be changed. If its element are changed a new RDD is created.

Generally after some transformation operations, people use action operations to gain some useful information from the RDD. Examples of action operations are count (count the number of items), reduce (apply a single summation-like function), aggregate (apply several summation-like functions), and collect (convert an RDD to a local array).

5.2 Implementation

We use $syn0$ to represent the input embedding V and $syn1$ to represent the output embedding U. $syn0$ and $syn1$ are defined as broadcast variables, which are only readable and can not be changed by executors. When they are changed in a training step copies are returned as a new RDD.

Data preparing We use the same corpus as other papers used, a snapshot of Wikipedia at April, 2010 created by **?**, 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). And then we calculate the frequency of word count. For example, there are 300 words which appear 10 times in the corpus. So the frequency of count 10 is 300. From this we can calculate the accumulated frequency. That is, if the accumulated frequency of count 200 is 100000, there would be 100000 words whose count is at least 200. This accumulated frequency can be used to select a vocabulary D with the desired number of entries, which all appear more frequent than $minCount$ times in the corpus. If the count of a word is smaller than $minCount$ we remove it from corpus, so it won't be in the vocabulary.

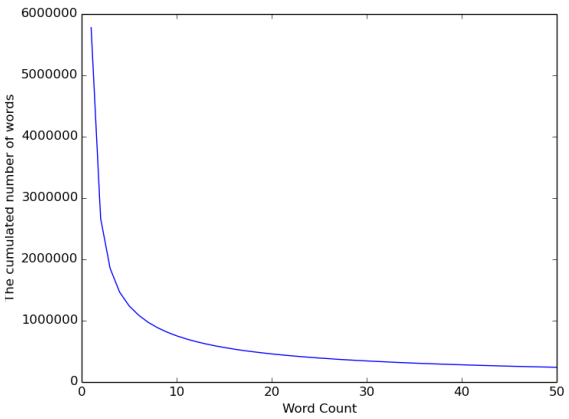

Figure 5.1: 1to51

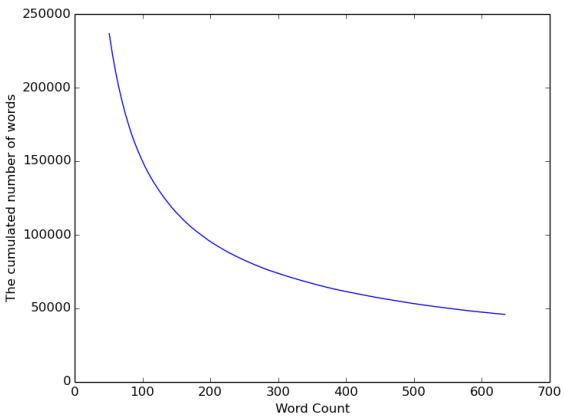

Figure 5.2: 51to637

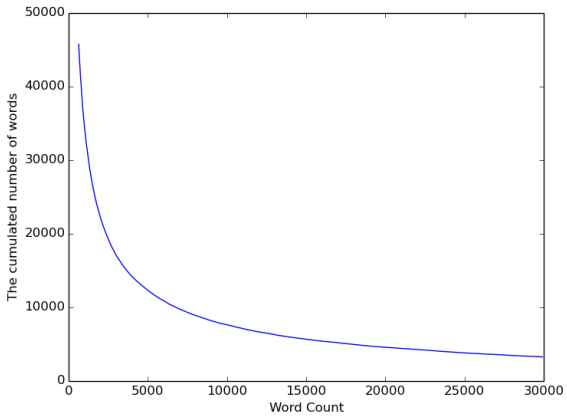

Figure 5.3: 637to31140

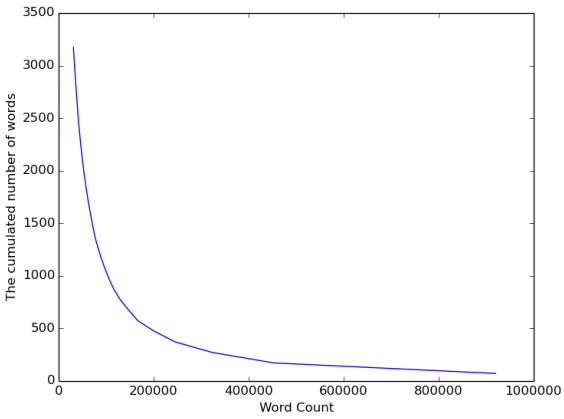

Figure 5.4: 31140torest

Computing Environment

Our program is running on a single machine with 32 cores. For some experiments, we use all cores as executers. We also tried some experiments on a compute cluster of several machines, 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 the corpus into a training set and a validation set. The training set has 99% of the data and validation set has only 1% of the data. We use the validation set to monitor our training process if it is converging. If the training algorithm converges, 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 the validation set after 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 to the disk. That is, after each calculation of the loss of validation set, we will store our results.

Note that, because we want to use the validation set to monitor our training, the validation set and training set should not be overlapping. And another import thing is that, the negative samples of validation set should always be fixed to reduce variance. The assignment step for the word senses of the validation set is almost the 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 are not changed. But for each iteration of sense assignment for sentences in the training set, the negative sampling are new.

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

Learn Step
In the training, we also use a map transformation. Instead of transforming sentences to sentences, we transform the original sentence RDD into the two-element collection of the updated $syn0$ and updated $syn1$. We broadcast these variables to the local $syn0$ and $syn1$ in each executor, so that each executor has its own copy of $syn0$ and $syn1$ and can update them independently. So each executor has copies of $syn0$ and $syn1$. And then we use *treeAggregate* to collect all such vectors together from different executors (cpu cores). In the aggregation operation, different $syn0$'s vectors add up together, and different $syn0$'s vectors add up together. Finally, by dividing by the the number of partitions, we get one global $syn0$ and one global $syn1$ in the driver. For now, we set them as new $syn0$ and $syn1$, which will be broadcasted again in the next iteration.

Normalization
After getting the new global $syn0$ and $syn1$, 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 their Euclidean length is bigger than 4. If that is the case we just normalize them to the new embeddings with length of 4.

Chapter 6

Evaluation

[1]

In the following chapter we use three different methods to evaluate our results. First we compute the nearest neighbors of different word senses. Then we use the t-SNE approach to project the embedding vectors to two dimensions and visualize semantic similarity. Finally we perform the WordSim-353 task proposed by Finkelstein et al. [2001] and the Contextual Word Similarity (SCWS) task from Huang et al. [2012].

The WordSim-353 dataset is made up by 353 pairs of words followed by similarity scores from 10 different people and an average similarity score. The SCWS Dataset has 2003 words pairs with their context respectively, which also contains 10 scores from 10 different people and an average similarity score. The task is to reproduce these similarity scores.

For the WordSim-353 dataset, we use the $maxSim$ function to calculate the similarity score of two words $w, \tilde{w} \in D$ from our model as following

$$maxSim(w, \tilde{w}) = \max_{1 \leq i \leq N_w, 1 \leq j \leq N_{\tilde{w}}} \cos(V_{w,i}, V_{\tilde{w},j}) \tag{6.1}$$

where $\cos(x, y)$ denotes the cosine similarity score of vectors x and y, N_w means the number of senses for word w, and $V_{w,i}$ represents the i-th sense input embedding vector of word w.

Cosine similarity is a measure of similarity between two vectors of an inner product space that measures the cosine of the angle between them. [From wikipedia] Specifically, given two vectors a and b with the save dimension d, the cosine similarity of them is

$$cos(a, b) = \frac{\sum_{i=1}^{d} a_i b_i}{\sqrt{\sum_{i=1}^{d} {a_i}^2} \sqrt{\sum_{i=1}^{d} {b_i}^2}}$$

[1] COMMENT: In the previous chapter you forgot to mention that the derivatives were checked by the empirical derivative computed by the finite difference approaximation.

Table 6.1: Definition of Hyper-Parameters of the Experiments

Fixed Parameters	
numRDD=20	The number of RDD to split training data set.
windowSize=5	The window size
numNegative=10	The number of negative samples
Variable Parameters	
id	The id number of the experiment.
c1	Minimal count for the inclusion of a word the in vocabulary D
vec	Vector size for each embedding vector
cm	Count thresholds for different numbers of senses
lr	The learning rate at the beginning of the experiment.
gm	The reduction factor of the learning rate for each iteration
syn1	true if each word has only one output embedding vector
init	The id number of experiment in step 1 (used to initialize embedding vectors)

The SCWS task is similar as the **Assign** operation. We use a 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.

6.1 Results for different Hyper-Parameters

[2] [3] [4]

Different hyper-parameters can generate different loss values on the validation set and require different computation time and memory. We tried many different parameters and found that the number of negative samples, the window size are not the typical factors to affect the final results. From the experiememnts we choose $c = 5$, the size of the $Context(w_t)$, i.e. the number of words before and after w_t. [7] The number K of negative samples randomly generated for a word was set to 10.[8] [9] And we also found that it is better

[2] COMMENT: Table captions are always above the tabular. The placement should alwas be tb

[3] COMMENT: Vector size for each embedding vector was d in the glossary. Please change

[4] COMMENT: What is The id number of experiment in step 1

[7] COMMENT: $windowSize = 5$

[8] COMMENT: $negNegative = 10$

[9] COMMENT: In the previous chapter you did not mention that the training set was split into $numRDD$ different RDDs, which were persisted to disk to allow execution of the training in RAM. You have to do this.

Table 6.2: Definition of Evaluation Scores

t1	The average Training time of each iteration [5] (excluding the parameter collection in the driver).
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 parameter collection)
t4	Total training time
loss	The best loss of the validation set
SCWS	The Spearman's rank correlations on the SCWS dataset. [6]
word353	The Spearman's rank correlations on the word353 dataset

Table 6.3: 8 Different Experiments in Step 1

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

to choose $numRDDs = 20$, which can balance the learning time and collection time for parameters. So in the following analysis, we do not change these three hyper-parameters shown as Table **??** and only focus on hyper-parameters shown in Table 6.1. And we mainly use the time, the loss and the score of similarity task shown as Table 6.2 to compare these hyper-parameters.

Note that we need two steps to train sense embedding vectors. In Step 1 the number of all word senses is set to one and the word embedding vectors are trained as in the usual word2vec approach. In Step 2 the program will use the result from Step 1 to do initialization of senses vectors (adding a tiny noise) and then train the sense embedding vectors. Finally, we decide to list only 11 experiments on Step 2 shown as Table 6.4, which are based on 8 experiments on Step 1 shown as Table 6.3.

In the following, we build 5 comparison groups based these 11 experiments to check how these hyper-parameters affect the final validation loss, the convergence speed, training time and similarity task scores.

Table 6.4: 11 Different Experiments in Step 2

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 6.5: Different Vector Size Comparison

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

Different sizes of embedding vectors
From the comparison in Table 6.5, it is clear that the vector size of the embeddings (vec) 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.

Different Min Count
We can find from Table 6.6 that the size of dictionary is not the important factor influencing loss or performance of similarity tasks. A higher mincount remove some words from the vocabulary which are 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 enters the training of frequent words only in a small amount. So they won't affect the final embedding vectors of frequent words.

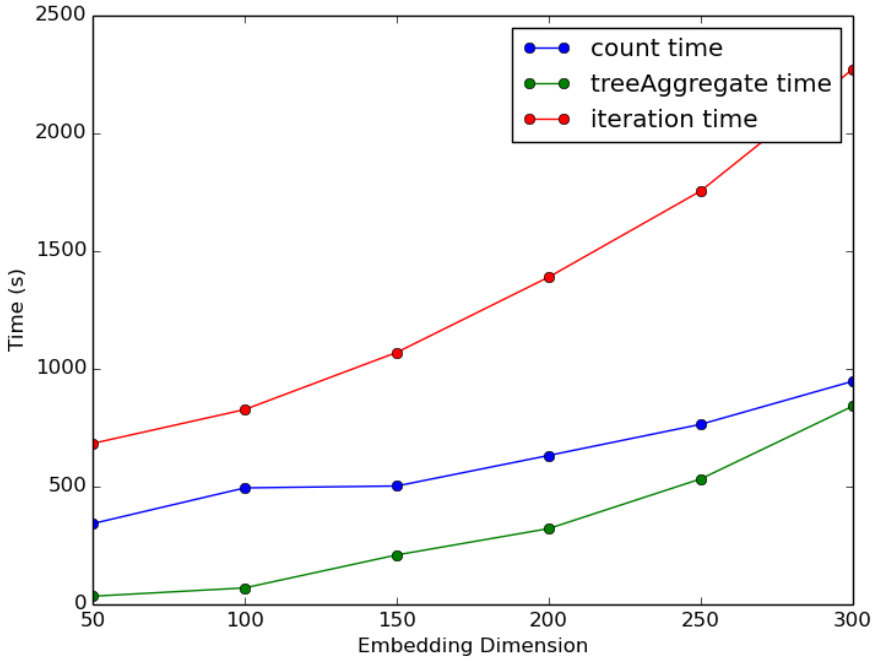

Figure 6.1: Shows the effect of varying embedding dimensionality of our Model on the Time

Table 6.6: 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

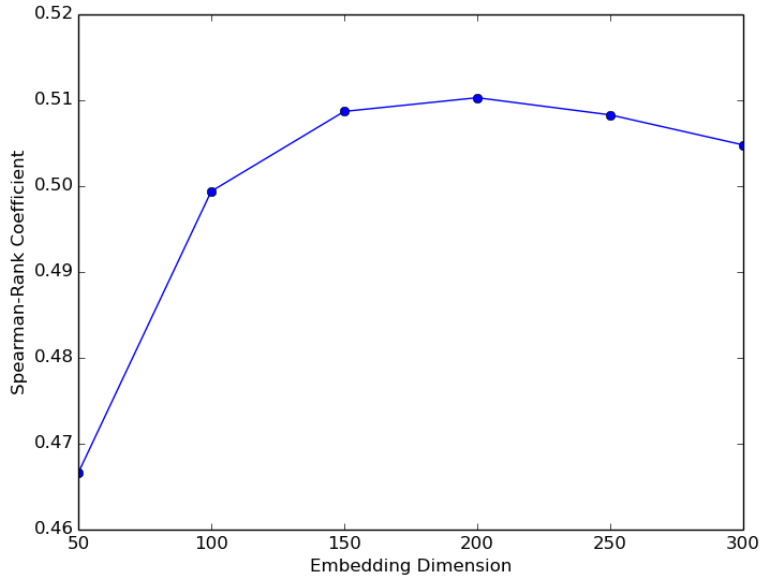

Figure 6.2: Shows the effect of varying embedding dimensionality of our Model on the SCWS task

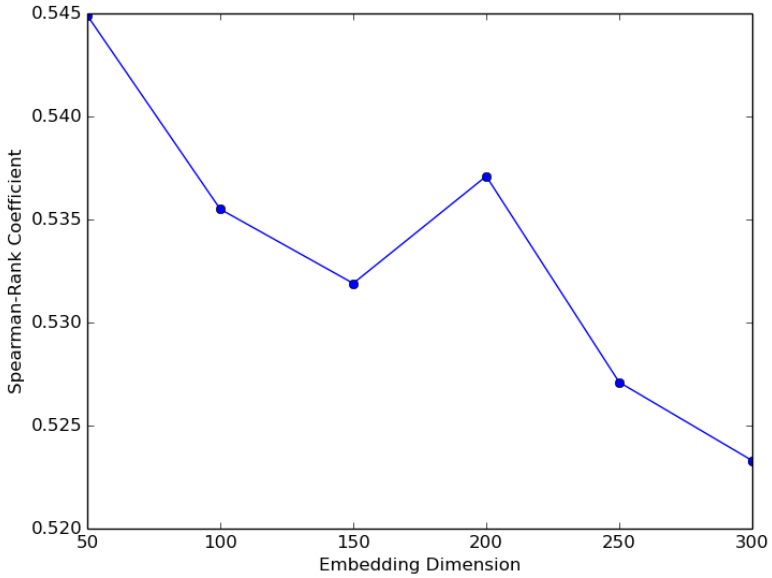

Figure 6.3: Shows the effect of varying embedding dimensionality of our Model on the word353 task

Table 6.7: 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

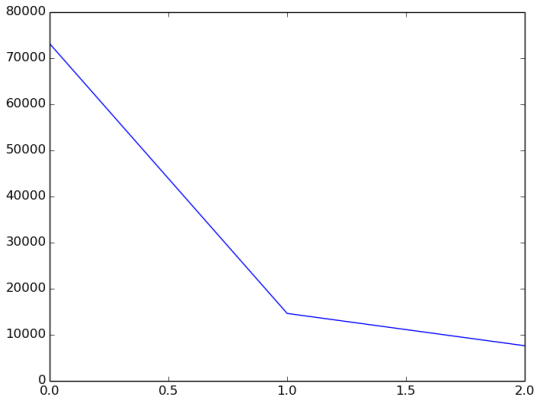

Figure 6.4: 2000to10000

Different Sense Count Comparison
From Table 6.7, we can know the sense count is not the important factor to affect the final loss. And their similarity task scores are also similar.

[73168,14636,7630] [73168,21225,1041] [85650,2154, 7630]

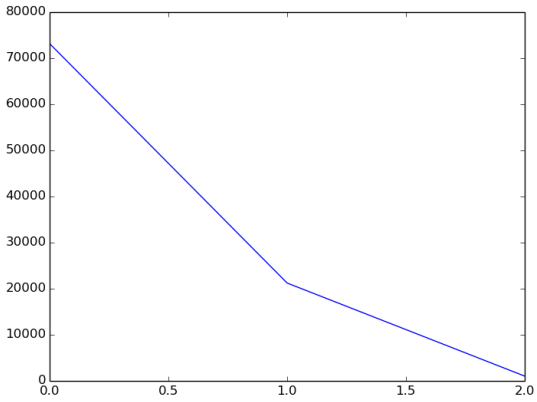

Figure 6.5: 2000to100000

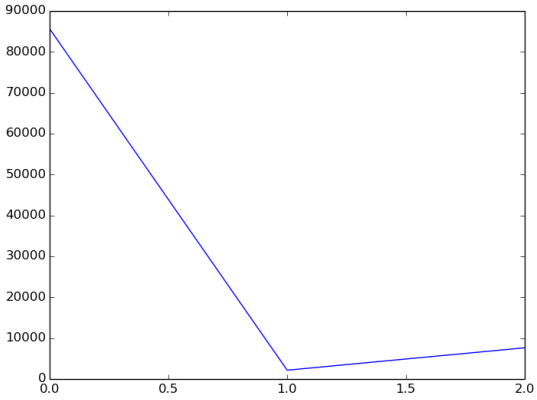

Figure 6.6: 7000to10000

Table 6.8: Different Learning Rate and Gamma Comparison

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	..	..

Different Learning Rate and Gamma
Table 6.8 shows that ...

Table 6.9: Comparison of the different number of output senses Syn1

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 6.10: Nearest words 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

Different Number of Output Senses Syn1
From Table 6.9, 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. After inspection the closest neighbors of senses the reason got clear. Say there are two words, e.g. "bank" and "money" with multiple senses. Then if money_1 was a close neighbor of bank_0 then it turned out that money_0 was a close neighbor of bank_1. Hence the closest senses were simply permuted, and the senses were not really meaningful. Hence we concluded that there should be only one output sense for each word. This will avoid this effect.

6.1.1 Comparison to prior analyses

Here you have to compare to the results similarity analysis results of Mikolov et al. [2013] and Huang et al. [2012].[10]

[11]

[10] COMMENT: todo!!!!

[11] COMMENT: A possible final verdict: Due to the little time left no thorough experiments could be performed. Spark parameters have to be tuned to distribute the calculations over multiple computing machines. The number of iterations as well as the schedule for reducing the learning rate has to be explored. If there is more time then the results should get much better.

Table 6.11: Sense Similarity Matrix of *apple*

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

Table 6.12: Nearest Words 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

6.2 Case Analysis

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

Firstly we give the result for the word *apple*, where different sense are quite nicely separated. Table 6.11 shows the sense similarity matrix of *apple*. The similarity value is the cosine similarity between two embedding vectors. Table 6.12 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 the computer company. The next are some sentence examples including the word *apple* in Table 6.13. These are the sentences containing the assigned word senses from the last iteration of training. To make it clear, we only display the sense label of the *apple*, although the other words also have multiple senses.

To visualize semantic neighborhoods we selected 100 nearest words for each sense of *apple* and use t-SNE algorithm [Maaten and Hinton, 2008] to project the embedding vectors into two dimensions. And then we only displayed 70% of words randomly to make visualization better, which is shown in Figure 6.7. And we use another table (Table ..) to show the comparison of with other two models (huang and em).

Table 6.13: Sentence Examples 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

Figure 6.7: Nearest words from *apple*

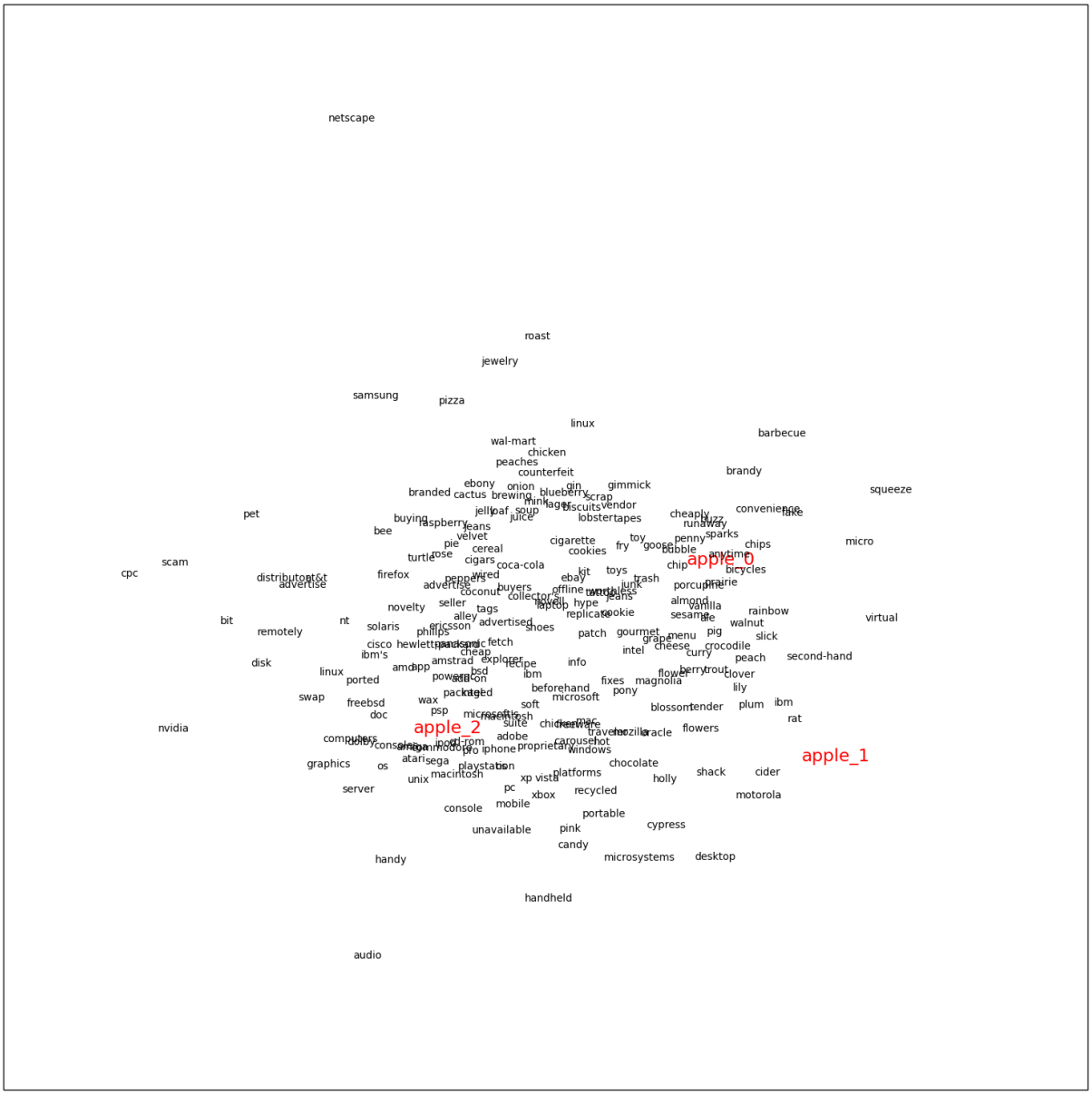

Table 6.14: Nearest words from *fox* , *net* , *rock* , *run* and *plant*

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

Next, we select other 5 words *fox* , *net* , *rock* , and *plant*, and list nearest words to each of their 3 senses in Table 6.14. Each line contains the nearest words for one of the senses. This table nicely illustrates the different meanings of words:

- fox: Sense 1 and 2 cover different movies and film directors while sense 3 is close to tv networks.
- net: Sense 1 is related to communication networks, sense 2 to profits and earnings and sense 3 to actions
- rock: Sense 1 and sense 2 is related to music while sense 3 to stone.
- run: Sense 1 is related to election campains, sense 2 expresses the movement and sense 3 to public transport.
- plant: Sense 1 is close to biologic plants and small animals, sense 2 is related to flowers and sense 3 to factories.

In table 6.15 we show one example sentence for each sense. 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: .

Finally, for each sense of each word (*apple*, *fox*,*net*,*rock* and *plant*), we select only the 20 nearest words, and combine them together to do another t-SNE embeddingof two dimensions. The the result is shown in Figure 6.8.

Table 6.15: Sentence Examples of 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

From these visualization, we can say our model is able to extract meaningful sense vectors which may be used for subsequent analyses. There is, however, room for improvement.

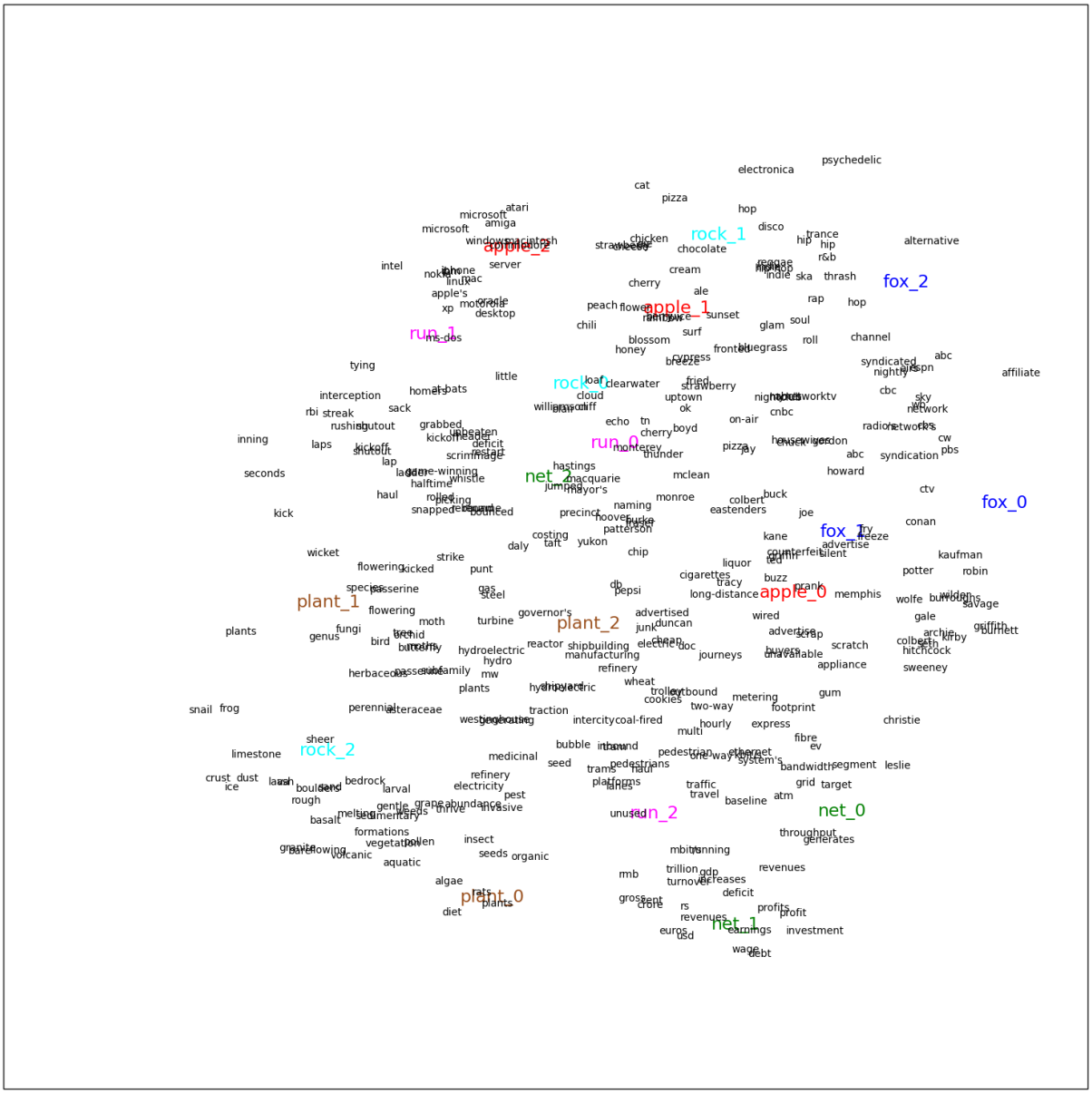

Figure 6.8: Nearest words from *apple*, ,*fox*, ,*net*, ,*rock*, , *run* and *plant*

Chapter 7

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

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *journal of machine learning research*, 3(Feb):1137–1155.

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

Finkelstein, L., Gabrilovich, E., Matias, Y., Rivlin, E., Solan, Z., Wolfman, G., and Ruppin, E. (2001). Placing search in context: The concept revisited. In *Proceedings of the 10th international conference on World Wide Web*, pages 406–414. ACM.

Gutmann, M. U. and Hyvärinen, A. (2012). Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. *Journal of Machine Learning Research*, 13(Feb):307–361.

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.

Maaten, L. v. d. and Hinton, G. (2008). Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605.

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. (2015a). Efficient non-parametric estimation of multiple embeddings per word in vector space. arXiv preprint arXiv:1504.06654.

Neelakantan, A., Shankar, J., Passos, A., and McCallum, A. (2015b). 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.

Williams, D. and Hinton, G. (1986). Learning representations by back-propagating errors. *Nature*, 323:533–536.

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*.