Computing Distributed Representations for Polysemous Words

Master Thesis
Software Systems Engineering

Haiqing Wang
Matriculation number 340863

July 8, 2016

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

Advisors:
Dr. Gerhard Paaß
Dr. Jörg Kindermann

Statutory Declaration

I hereby certify that all work presented in this master thesis is my own, no other than the sources and aids referred to were used and that all parts which have been adopted either literally or in a general manner from other sources have been indicated accordingly.

Aachen, July 8, 2016

YOUR NAME

Acknowledgements

I would like to thank the Department of Computer Science 5 - Information Systems and Fraunhofer Institute IAIS for supporting my master thesis. I would like to express my great gratitude for the support of Dr. Gerhard Paaß and Dr. Jörg Kindermann. Their patient guidance helps me a lot during the research of this topic.

Abstract

Recently, machine learning especially deeplearning is very popular. Word vector is very important tool for many natural language processing when using machine larning algorithms. There are many methods ([Bengio et al., 2003],[Collobert and Weston, 2008] and [Mikolov et al., 2013]) to generate word vector, usually we call this process word embedding, and call such word vector distributed representation. Most of word embedding methods can not generate word vector based on word's context, that is similar words have similar vectors. But there are still problems when doing some tasks like decting word senses. Some polysemous words can represent different mearnings in different contexts. Acoddingly, each polysemous word should have several vector. Some models have been successively proposed ([Huang et al., 2012],[Tian et al., 2014] and [Neelakantan et al., 2015a]) to do sense embeddings to represent word senses. Our thesis investigates and improves current methods with multiple senses per word . Specifically we extend the bacis word embedding model, i.e. word2vec ([Mikolov et al., 2013]), to build a sense assignmen model. In short, each word can have several senses in each word, we use some score function to decide the best sense for each word, through a lot of unsupersived learning, our model adjust senses for each word in sentences and finally generate sense vectors. This thesis implements this model in Spark to be able to execute in parallel and trains sense vectors with Wikipedia corpus. We evaluate sense vectors by doing word similarity tasks using SCWS (Contextual word Similarityes) dataset from [Huang et al., 2012] and WordSim-353 dataset from [Finkelstein et al., 2001] .

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

. 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

1.1 Word Embedding

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. 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.

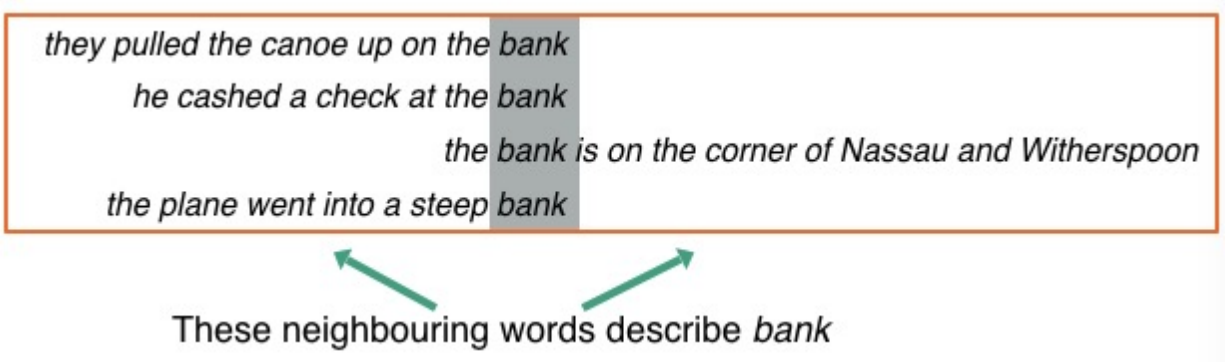

Figure 1.1: Neigboring words defining the specific sense of "bank".

Traditional word embedding methods first obtain the co-occurrence matrix and then perform dimension reduction suing singular value decomposition (SVD) [Deerwester et al., 1990].

Recently, artificial neural networks is very popular to generate 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.

1.2 Sense Embedding

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.

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.

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, for each position of the word, the best fitting embedding is selected according the fitness criterion. 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].

1.3 Goal

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.

Our goal is to investigate sense assignment models which will extend known word embedding (one sense) approaches and implement such method on a compute cluster using Apache Spark to be able to process larger training corpora and employ higher-dimensional sense embedding vectors. So that we can derive expressive word representations for different senses in an efficient way. And our main work will focus on the extension of Skip-gram model [Mikolov et al., 2013] in connection to the approach of [Neelakantan et al., 2015a] because these models are easy to use, very efficient and convenient to train.

1.4 Outline

The rest of the report is structured as the following. Chapter 2 introduces the background about word embedding and explain the mathematical details of one model (word2vec Mikolov et al. [2013]) that one must be acquainted with in order to understand the work presented. Chapter 3 presents some relevant literatures and several latest approaches about sense embedding. Chapter 4 describes the mathematical model of our work for sense embedding. Chapter 5 introduces Spark framework and discusses in detail the implementation of our model. Chapter 6 analyzes the effect of different parameters from the model and compares the result with other models. Chapter 7 represents concluding remarks to our work including the advantages and disadvantages and talks about what can be done further to improve our model.

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 embeddings is using a *one-hot representation*, which uses 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 $|D|$. Most important it can not describe the similarity between words well.

An alternative word embedding is the *Distributed Representation*. It was firstly proposed by Williams and Hinton [1986] and can overcome the above drawbacks from one-hot representation. The basic idea is to map each word into a short vector of fixed length (here "short" is with respect to "long" dimension $|D|$ of the 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 which can be used to estimate a Distributed Representation vector, including the famous LSA (Latent Semantic Analysis) [Deerwester et al., 1990] and the LDA (Latent Dirichlet Allocation) [Blei et al., 2003]. In addition, the neural network algorithm based language model is a very common method and becomes more and more popular. A language model has the task to predict the next word in a sentence from the words observed so far. In these neural network based models, the main goal is to generate a language model usually generating a word vectors as byproducts. In fact, in most cases, the word vector and the language model are bundled together. The first paper on this is the Neural Probabilistic Language Model from [Bengio et al., 2003], followed by

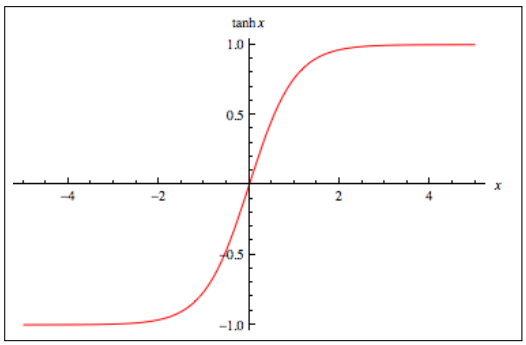

Figure 2.1: Graph of the *tanh* function

a series of related research, including SENNA from Collobert et al. [2011] and word2vec 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-of-speech tagging, chunking into phrases, named entity recognition and semantic role labeling, and achieves good results.

2.2 Neural Probabilistic Language Model

Neural Network

General speaking, a neural network defines a mapping between some input and output, and it usually contains an input layer, an output layer and one or more 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. Let $x(i-1)$ be the input vector to the i-th layer and $x(i)$ its output vector. From one layer to the next layer, there is a map function $f(x(i-1)) = x(i)$ made up by an *activation function* with a weight matrix and and an offset vector.

Specifically vector $x(i-1)$ is multiplied by a weight matrix H and then adds an offset vector q, after that an activation function such as *tanh* function shown as Figure 2.1. Hence we have

$$x(i) = f(x(i-1)) = \tanh(H * x(i-1)) + q \tag{2.1}$$

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 three nodes, that is the input should be a vector with dimension 3. The hidden layer has four nodes, which means the input vector will be mapped to a hidden vector with dimension 4. And then it will be mapped again to the output vector with dimension 2. Specifically, the input is a vector $x(0)$ (with dimension 3). H (with size 4×3) and U (with size 2×4) are respectively the weight matrix between input layer and hidden layer and the weight matrix between hidden layer and output layer,

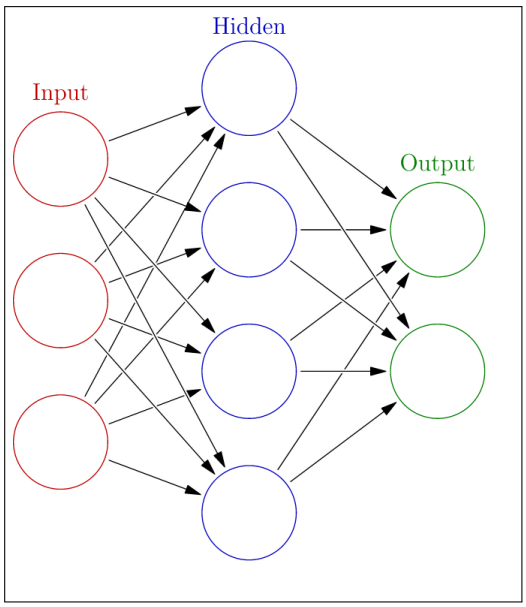

Figure 2.2: An example of neural network with three layers

p (with size 4) and q (with size 2) are the offset vectors of respectively the hidden layer and the output layer. The activation function of the hidden layer is the tanh function and the activation function of the output layer is the identity function. So we get the following formula

$$\begin{aligned} x(1) &= \tanh(H * x(0)) + p, \qquad x(2) = U * x(1) + q \\ x(2) &= U(\tanh(H * x(0)) + p) + q \end{aligned} \tag{2.2}$$

Statistical Language Model

Statistical language models are widely used in natural language processing, speech recognition, machine translation. A Statistical language model is a probability model to calculate the probability of a sentence or word sequence, and is always built on a corpus C containing sentences.

Let $C = (S_1, \ldots, S_M)$ be a corpus consisting of M sentences $S_i = (w_{i,1}, \ldots, w_{i,L_i})$, each of length L_i. Then a language model computes the probability $p(C)$ of the corpus and the

probabilities $p(S_i)$ of the sentences S_i according to the conditional probability definition

$$p(C) = \prod_{i=1}^{M} p(S_i) \tag{2.3}$$

$$p(S_i) = \prod_{t=1}^{L_i} p(w_t|w_{t-1}, \ldots, w_1)$$

$$\approx \prod_{t=1}^{L_i} p(w_t|Prev(w_t)) \tag{2.4}$$

where $Prev^{n-1}(w_t) = (w_{i,\max(t-n+1,1)}, \ldots, w_{i,t})$ is the subsequence of $n-1$ past words before w_t in sentence S_i. Note that the last equation is an approximation of the full conditional probability. The core part of a Statistic language model is to compute $p(w_t|Prev(w_t))$ by some model if $Prev(w_t)$ is given.

Neural Probabilistic Language Model

Bengio et al. [2003] introduce a neural probabilistic language model shown as Figure 2.3, which uses neural network to approximate $p(w_t|Prev(w_t))$ the language model (2.4).

C is the given corpus containing the sentences/documents of words from a dictionary D. Consider a word w_t in some sentence $S_i = (w_1, w_2, \ldots, w_{T-1}, w_{L_i})$ of corpus C, where t is some position in S and L_i is the length of S. Let $Prev^{n-1}(w_t) = (w_{\max(t-n+1,1)}, \ldots, w_{t-1})$ be the subsequence of words before w_t.

Firstly for each word w in dictionary D, there is a look-up table V mapping the word w to vector $V(w)$. Note that V is denoted as C in figure 2.3. The vector size of $V(w)$ is d. The input layer is a long vector concatenated by $n-1$ word vectors in $Prev^{n-1}(w_t)$. 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 size of vocabulary.

Bengio et al. [2003] also use tanh function for the activation function in the hidden layer. H (with size $m \times (n-1)d$) and U (with size $|D| \times m$) 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 $|D| \times (n-1)d$). So the mapping function from x to the output is

$$y = q + U * \tanh(p + H * x) + W * x \tag{2.5}$$

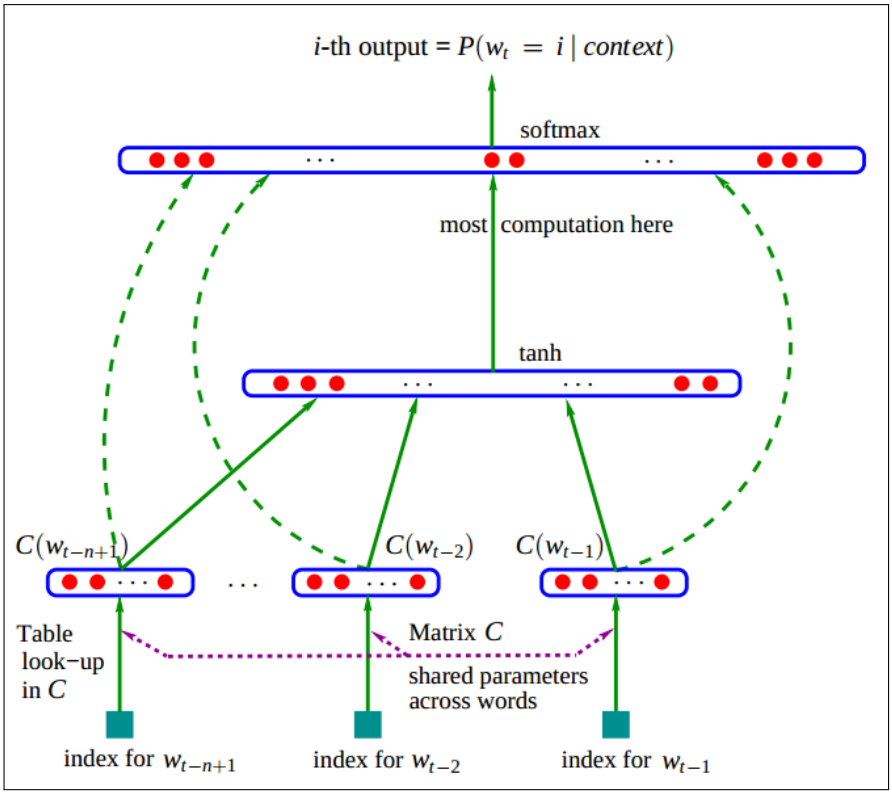

Figure 2.3: The neural network structure from [Bengio et al., 2003]

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 [1]. 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 $|D|$ with arbitrary values and can not represent probabilities. Because it is a language model, it needs to model the probability as $p(w_t|Prev^{n-1}(w_t))$. Actually, Bengio et al. [2003] use 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 2.5 we have

$$y_{w_t} = q + U * \tanh(p + H * x_{w_t}) + W * x_{w_t} \tag{2.6}$$

[1] https://en.wikipedia.org/wiki/Softmax_function

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

$$p(w_t|Prev^{n-1}(w_t)) = \frac{e^{y_{w_t,i_{w_t}}}}{\sum_{i=1}^{|D|} e^{y_{w_t,i}}}, \tag{2.7}$$

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 2.4. 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 of the approach of 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 2.4 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 with a vocabulary D of size N. 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. We define $phrase(w_t)'$ as $phrase(w_t)$ where the center word w_t is replaced by anther random word $w' \neq w_t$. Each word is represented by a vector of 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 $phrase(w_t)'$.

[Collobert and Weston, 2008] also use a neural network to build their model. But the neural network structure is different from the network structure in [Bengio et al., 2003]. Its 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 2.7. 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

$$\sum_{w \in C} max\{0, 1 - f(x_w), f(x'_w)\} \tag{2.8}$$

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.

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

$$\prod_{w \in C} p(Context(w)|w), \tag{2.9}$$

where $Context(w)$ are the words close to word w. Note that this criterion is related to but different from the language model criterion 2.4.

Again C is the given corpus with vocabulary D containing N different words. 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
$Context(w_t) = (w_{,\max(t-c,1)}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{,\min(t+c,T)})$, and c is the number of words before and after w_t in $Context(w_t)$. The Figure 2.4 shows the structures of CBOW model and Skip-gram model when $c = 2$.

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 words in the context, for the skip-gram model the input is the word vector of w_t. And the output is the vector of length N, (the size of the vocabulary). Obviously their prediction strategies are different. CBOW uses the context to predict the center word. And Skip-gram model uses the center word to predict the 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 (2.8). Let V and U represent 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

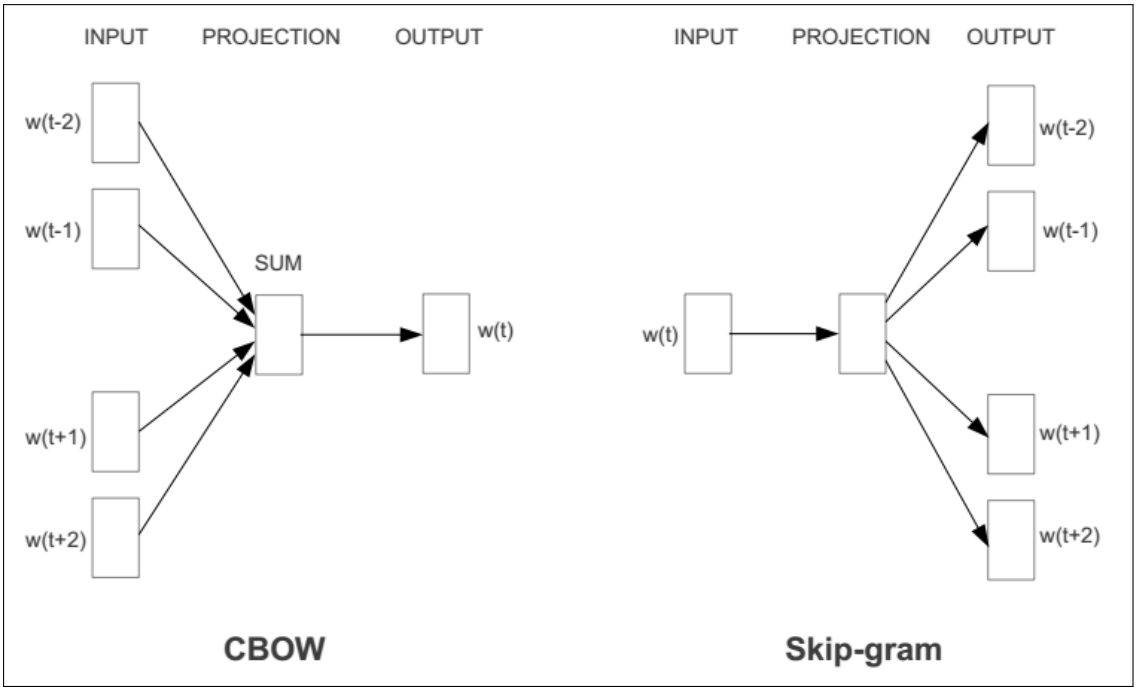

Figure 2.4: word2vec

samples is K. And $P(w)$ is the smoothed unigram distribution which is used to generate negative samples. Specifically for each $w \in D$

$$P(w) = \frac{count(w)^{\frac{3}{4}}}{(\sum_{i=1}^{M} L_i)^{\frac{3}{4}}}$$

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

$$G = \frac{1}{M} \sum_{i=1}^{M} \frac{1}{L_i} \sum_{t=1}^{L_i} \sum_{\substack{-c \leq j \leq c \\ j \neq 0 \\ 1 \leq j+t \leq L_i}} \left\{ \log\, p(w_{i,t+j}|w_{i,t}) + \sum_{k=1}^{K} \mathbb{E}_{z_k \sim P(w)} \log\, [1 - p(z_k|w_{i,t})] \right\} \tag{2.10}$$

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

$$loss(w_{i,t}, w_{i,t+j}) \quad = \quad -\log\, p(w_{i,t+j}|w_{i,t}) - \sum_{k=1}^{K} \mathbb{E}_{z_k \sim P(w)} \log\, [1 - p(z_k|w_{i,t})] \qquad (2.11)$$

Here the loss is defined as the negative log probability of $w_{i,t}$ given $w_{i,t+j}$.

And the loss function of whole corpus is

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

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

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

Use

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

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

Use stochastic gradient descent:[2]

- Initialize $\{V, U\}$
- For $It * T$ Iterations:
  - For each training sample $(w_{i,t}, w_{i,t+j})$
    - Generate negative sample words to replace $w_{i,t+j}$: $(w_1, \ldots, z_k)$

[2] https://en.wikipedia.org/wiki/Stochastic_gradient_descent

- 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)

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

$$\begin{aligned}
\Delta_{V_{w_{i,t}}} &= -\frac{\partial loss(w_{i,t}, w_{i,t+j})}{\partial V_{w_{i,t}}} = [1 - \log\, \sigma(U^{\mathrm{T}}_{w_{i,t+j}} V_{w_{i,t}})] U_{w_{i,t+j}} + \sum_{i=1}^{k} [-\log\, \sigma(U^{\mathrm{T}}_{z_k} V_{w_{i,t}}))] U_{z_k} \\
\Delta_{U_{w_{i,t+j}}} &= -\frac{\partial loss(w_{i,t}, w_{i,t+j})}{\partial U_{w_{i,t+j}}} = [1 - \log\, \sigma(U^{\mathrm{T}}_{w_{i,t+j}} V_{w_{i,t}})] V_{w_{i,t}} \\
\Delta_{U_{z_k}} &= -\frac{\partial loss(w_{i,t}, w_{i,t+j})}{\partial U_{z_k}} = [-\log\, \sigma(U^{\mathrm{T}}_{z_k} V_{w_{i,t}}))] V_{w_{i,t}}, 1 \leq k \leq K
\end{aligned}$$

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

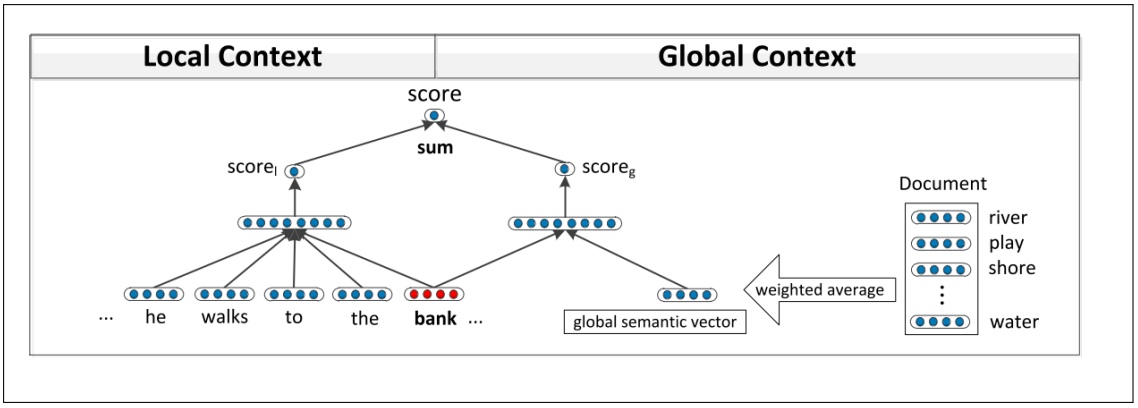

Figure 3.1: The network structure from [Huang et al., 2012]

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). Finally he re-trained the word vectors. Table 3.1 gives some examples from his model's results.

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 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

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

$bank_2$. 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 S senses (S is set advance manually. 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 S clusters , it has only S context vectors. And in the initialization, these S 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, S sense vectors and S 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 $v_s(w_t, s)(s = 1, 2, \ldots, S)$ and the context cluster center vectors are $\mu(w_t, s)(s = 1, 2, \ldots, S)$. The architecture of the MSSG is like Figure 3.2 when $c = 2$. It uses the following formula to select the best cluster center:

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

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.

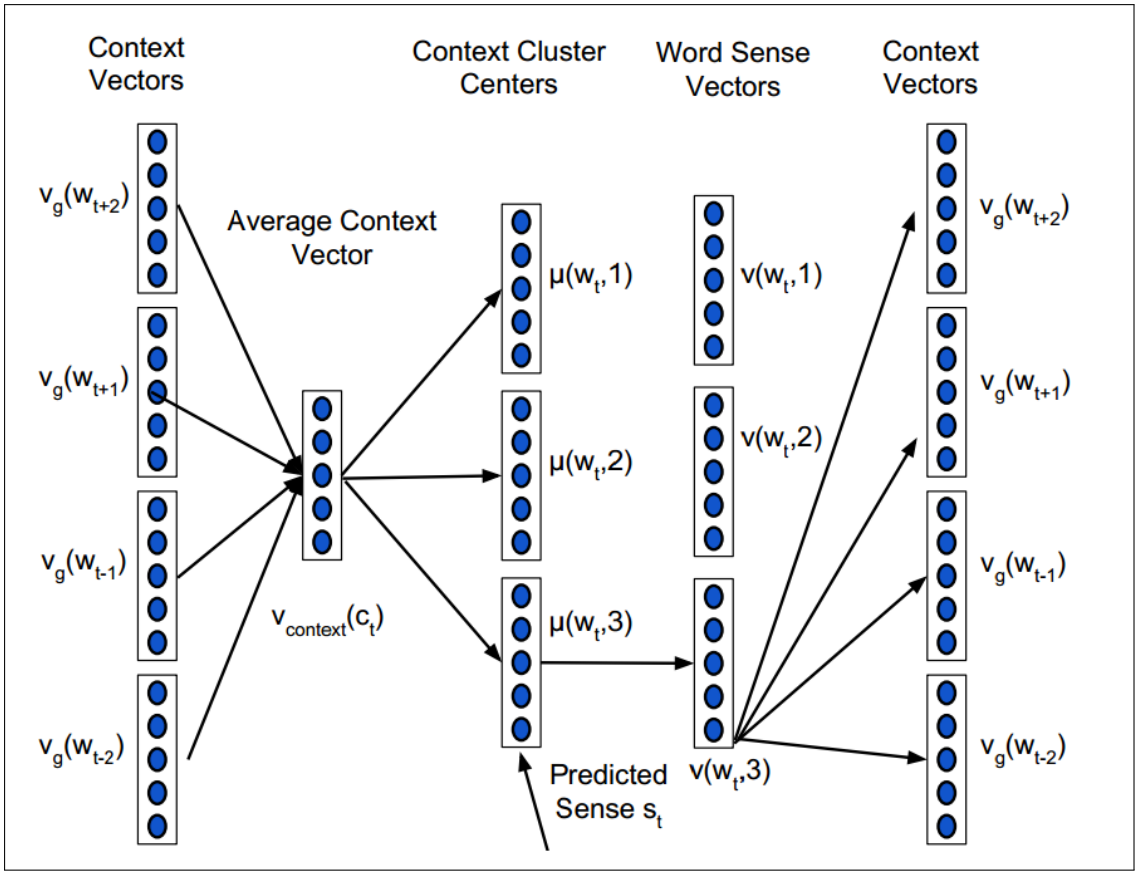

Figure 3.2: Architecture of MSSG model with window size $R_t = 2$ and $S = 3$

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 :

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

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

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. We use (w, s) to represent the word w's s-th sense, i.e. $(w_{i,t}, h_{i,t})$ represents the word $w_{i,t}$'s $h_{i,t}$-th sense, and $p((w_{i,t+j}, h_{i,t+j})|(w_{i,t}, h_{i,t}))$ represents the probability using $w_{i,t}$'s $h_{i,t}$-th sense to predict $w_{i,t+j}$'s $h_{i,t+j}$-th sense, where $w_{i,t}$ and $w_{i,t+j}$ are indexes of words in the position t and $t+j$ respectively from sentence S_i. And $h_{i,t}$ and $h_{i,t+j}$ represent their assigned sense indexes, which can be adjusted by model in the training. The above prediction probability is only for a pair of word with sense information, the goal of the model is to maximize every possible pairs of words which can use a probability computed by producing every prediction probabilities of word pairs to resent the prediction probability based on the whole corpus. The model's task is to adjust sense assignment and learn sense vectors in order to get the biggest prediction probability based on the whole corpus. Specifically, we use the following likelihood function to achieve above objective

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

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

In the **Assign**, 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) for word $w_{i,t}$. In the **Learn**, we take $[(w_{i,t}, h_{i,t}), (w_{i,t+j}, h_{i,t+j})]$ as a training sample and use the negative log probability as **loss function** *loss* for each sample

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

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

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 to get

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

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.

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.

Actually, these sense probabilities 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.

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

procedure 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**
        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 if
      end for
    end for
  end for
end procedure

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 machines. 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.

Gradient Checking

Firstly we set a very small dataset mutually and calculate empirical derivative computed by the finite difference approaximation derivatives and the derivative computed as our model shows from last chapter. The result shows the difference between these two derivative is very small. So our gradient calculation is correct.

Data preparing

We use the same corpus as other papers used, a snapshot of Wikipedia at April, 2010 created by Shaoul [2010], which has 990 million tokens. Firstly we count the all words in the corpus. We transform all words to lower capital and then generate our vocabulary (dictionary). 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. The following 4 figures (Figure 5.1,Figure 5.2,Figure 5.3 and Figure 5.4) show the relationship between accumulated frequency and word count. To make visualization more clear, we display it as four different figures with different ranges of word count. And with some experience from other papers ([Huang et al., 2012], [Tian et al., 2014] and [Neelakantan et al., 2015a]), in some of our experiments we set $minCount = 20$

and others have $minCount = 200$. Actually, when word count is 20, the accumulated frequency is 458142, that is the vocabulary size would be 458142; when word count is 200, the accumulated frequency is 95434, that is the vocabulary size would be 95434.

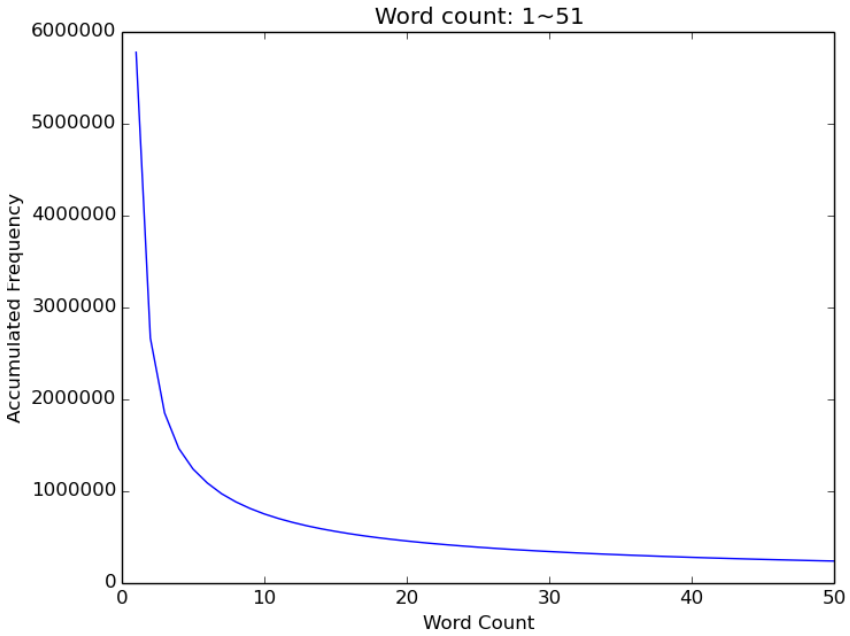

Figure 5.1: Shows the accumulated frequency of word count in range [1,51]

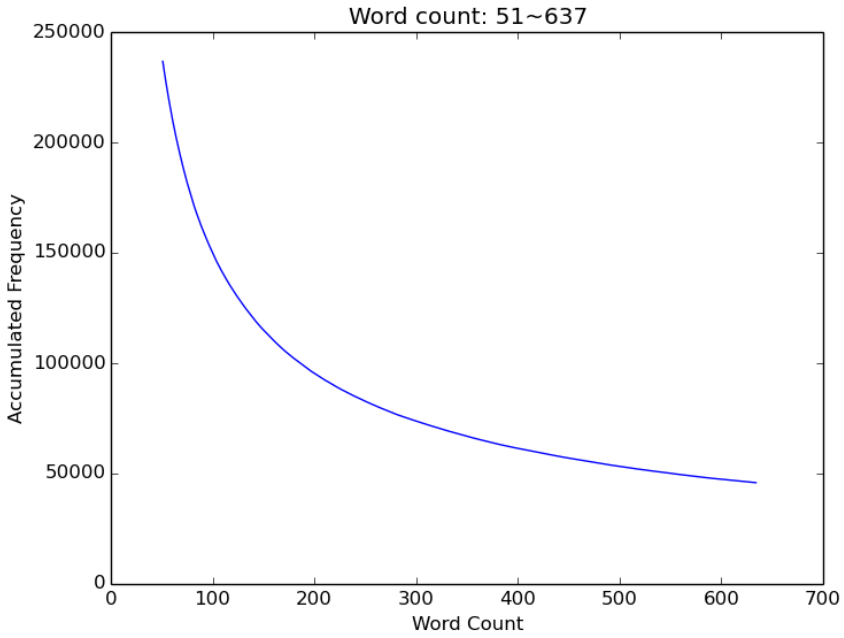

Figure 5.2: Shows the accumulated frequency of word count in range [51,637]

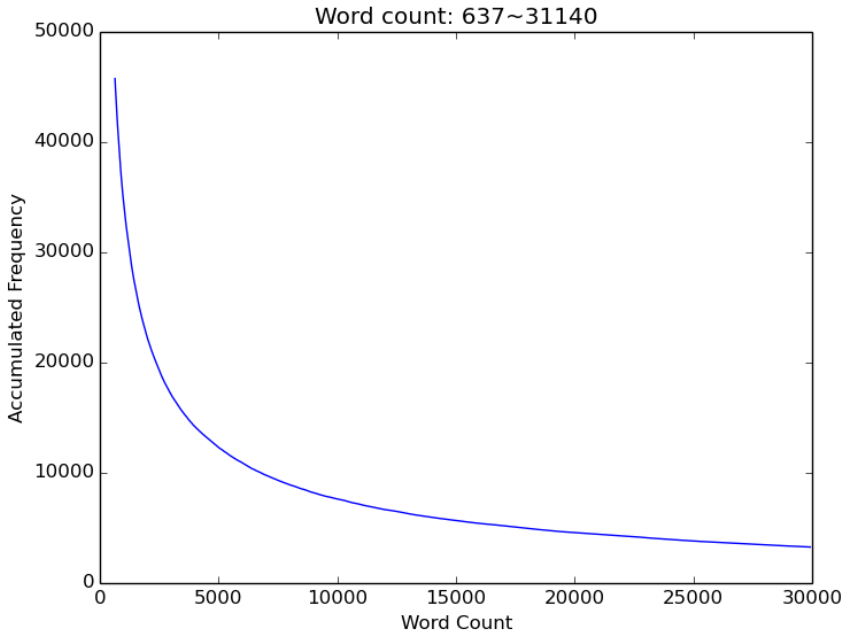

Figure 5.3: Shows the accumulated frequency of word count in range [637,31140]

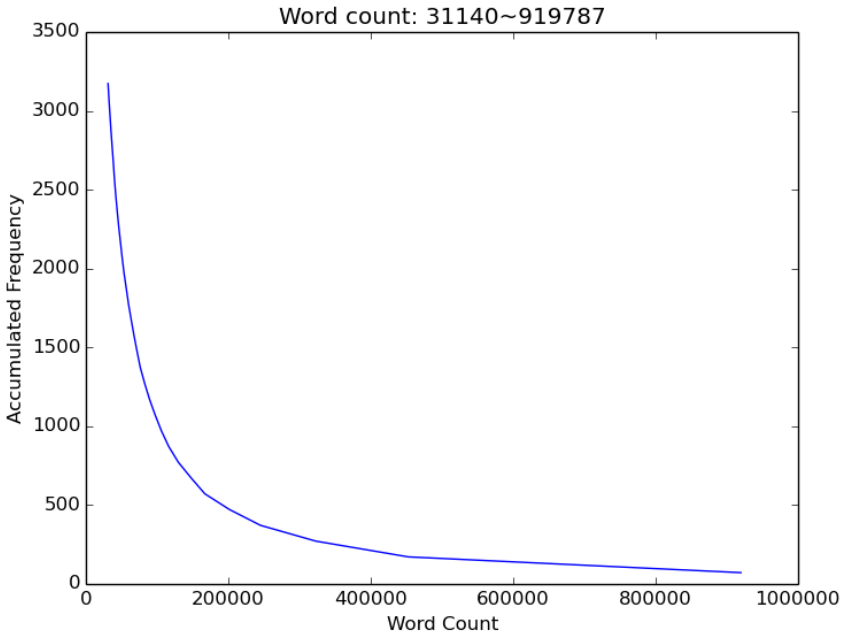

Figure 5.4: Shows the accumulated frequency of word count in range [637,919787]

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 the time of collecting parameters ($syn0$ and$syn1$) is too slow and actually too many executors actually is not really good for our program. After learning parameters parallel by stochastic gradient, the program collects all parameters and calculates the average, which reduces the learning effect and slow down the convergence especial when the number of executors is too many, although more executors can speed up training more or less.

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 gradually increases, which means the training is over-fitting. So we 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 store our results.

Note that, the validation set and training set should not be overlapping, because we use the validation set to monitor our training. 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.

Another thing is that, for each iteration, we do not use the whole training set to assign senses and learn parameters. Instead we split the training set into several parts and each time we only fetch one data part to do sense assignment (Assign Step)and parameters learning (Learn Step). Specifically, the training set was split into $numRDD$ different RDDs, which were persisted to disk to allow execution of the training in RAM. $numRDD$ is the number of RDD to split training dataset.

Learning Rate Reduction

At the beginning of experiment, the learning rate is set to lr. After each iteration, the learning rate will be reduced with a reduction factor gm. Specifically, using α to represent current learning rate and α' to represent the new learning rate, we have

$$\alpha' = \alpha * gm$$

Iteration

As described before, the while training process include several iterations. At the beginning the program fetches the first RDD and then for next each iteration, the it fetches one of other RDDs orderly. After several iterations, the program finishes processing all RDDs, and for the next iteration (if not stopping) it will fetches the first RDD again and repeat the above . Each iteration contains the sense assignment, adjusting the sense probabilities for each word based on the assigned senses, learning parameters , collecting parameters using *treeAggregate* and normalizing sense embeddings, and every operations are on training set. And in some iteration (not every), the program assigns senses on the validation set and stores the RDD data. In experiments we compare the time of each operation in an iteration, and find that comparing the time of learning parameters and the time of parameters collection, the time of other operations can be ignored. Define $t1$ as the average time of learning parameters in an iteration, $t2$ as the collecting parameters in an iteration, $t3$ as the average time of all operations in an iteration, $t4$ as the total training time and *iter* as the total number of iterations. And we will do some analysis on these time in next chapter.

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

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 $avgSim$ function to calculate the similarity of two words $w, \tilde{w} \in D$ from our model as following

$$avgSim(w, \tilde{w}) = \frac{1}{N_w} \frac{1}{N_{\tilde{w}}} \sum_{i=1}^{N_w} \sum_{j=1}^{N_{\tilde{w}}} \cos(V_{w,i}, V_{\tilde{w},j}) \tag{6.1}$$

where $\cos(x, y)$ denotes the cosine similarity 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 [1]. Specifically, given two vectors a and b with the save dimension d, the cosine similarity of them is

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

The SCWS task is similar as the **Assign** operation. In this task, we use $avgSim$ function as well and another function $localSim$. For two words $w, \tilde{w} \in D$

$$localSim(w, \tilde{w}) = cos(V_{w,k}, V_{\tilde{w},\tilde{k}})$$

[1] https://en.wikipedia.org/wiki/Cosine_similarity

where $k = \arg\max_i P(context_w|w,i)(1 \leq i \leq N_w)$ and $\tilde{k} = \arg\max_j P(context_{\tilde{w}}|\tilde{w},j)(1 \leq j \leq N_{\tilde{w}})$ and $P(context_w|w,i)$ is the probability that w takes the i^{th} sense given context $context_w$. Note that $context_w$ is the context information (several words before and after w) given by SCWS dataset.

Here we calculate the probability using the center word to predict the context words as above probability. But 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 (we will show the bad results when output embedding has multiple senses and explain the reason in the following section). So we just use the normal skip-gram model's prediction function to select the best center word's sense.

In order to evaluate our model, after getting the similarity score for each pair of words, we use Spearman's rank correlations ρ to calculate the correlation between the word similarities from our model and the given similarity values from SCWS and WordSim-353 datasets. The higher correlation represents the better result. The definition details can be referred in Wikipedia[2]. And in fact we always use this Spearman's rank correlations as the score of similarity task.

In the following analysis, for the hyper-parameters comparison we only use $localSim$ function to calculate the word similarity. And in the comparison with other models, we use both $localSim$ and $avgSim$ functions to get word similarity.

6.1 Results for different Hyper-Parameters

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 context size are not the typical factors to affect the final results. From the experiments we choose $c = 5$, the size of the $Context(w_t)$, i.e. the number of words before and after w_t. The number of negative samples K randomly generated for a word was set to 10.

And we also found that it is better to choose $numRDDs = 20$, which can balance the time of learning parameters and the time of collecting parameters. So in the following analysis, we do not change these three hyper-parameters only focus on other hyper-parameters, and Table 6.1 shows the hyper-parameters we need. 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 16 experiments on Step 2 shown as Table 6.4, which

[2]https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient

Table 6.1: Definition of Hyper-Parameters of the Experiments

Fixed Parameters	
$numRDD$=20	The number of RDD to split training data set.
c=5	The size of context
K=10	The number of negative samples
Variable Parameters	
id	The id number of the experiment.
d	Vector size for each embedding vector
$c1$	Minimal count for the inclusion of a word in vocabulary D
$c2$	Count thresholds for words with two senses i.e. the count of w is more than $c2$, w has at least two senses
$c3$	Count thresholds for words with three senses i.e. the count of w is more than $c3$, w has at least three senses
lr	The learning rate at the beginning of the experiment.
gm	The reduction factor of the learning rate for each iteration
$S1$	true if sense has only one output embedding vector

Table 6.2: Definition of Evaluation Scores

$t1$	The average time of learning parameters in one iteration
$t2$	The average time of collecting parameters using $treeAggregate$ in one iteration
$t3$	The average time of all operations in one iteration
$t4$	Total training time
$iter$	The number of total training iterations
$vLoss$	The best loss of the validation set
$SCWS$	The Spearman's rank correlations on the SCWS dataset.
$word353$	The Spearman's rank correlations on the WordSim-353 dataset

are based on 13 experiments on Step 1 shown as Table 6.3 and our experiments always use the same d, $c1$, lr and gm in two steps.

Another thing is that we calculate the loss of validation set every 5 iterations, so the the number of total training iterations would be multiple of 5.

In the following, we build 5 comparison groups based on these 16 experiments to check how these hyper-parameters affect the final validation loss, the convergence speed, training time and similarity task scores.

Different sizes of embedding vectors

Table 6.3: 13 Different Experiments in Step 1

id	*d*	*c*1	*lr*	*gm*
(1)	300	200	0.1	0.9
(2)	250	200	0.1	0.9
(3)	200	200	0.1	0.9
(4)	150	200	0.1	0.9
(5)	100	200	0.1	0.9
(6)	50	200	0.1	0.9
(7)	50	200	0.2	0.9
(8)	50	200	0.05	0.9
(9)	50	200	0.01	0.9
(10)	50	200	0.1	0.95
(11)	50	200	0.1	0.85
(12)	50	200	0.1	0.8
(13)	50	20	0.1	0.9

From the comparison in Table 6.5, we can find that their convergence speed is similar based on *iter* (the total number of iterations). Although *iter* of experiment 3 is bigger, it can not prove that when $d = 200$ (embedding vector size), the convergence speed is slowest. Because they are not so different based on the fact that *iter* is multiple of 5, and for each set of parameters we only did one experiment, where different initialization may affect the final results including *iter*. We think we can do more experiments for the same hyper-parameters in the future to make our results more reliable.

Figure 6.1 shows the relationship between time ($t1$,$t2$ and $t3$) and embedding dimension d. It is very clear that bigger embedding dimension requires more time to learn parameters and collect parameters, because bigger embedding dimension means more parameters to be dealt with. Figure 6.2 and Figure 6.3 show the effect of varying embedding dimensionality on the loss of validation set and the score of SCWS task respectively. When d becomes bigger, both *loss* and $SCWS$ firstly become better(smaller and bigger respectively) and then maintain same or gradually reduce. These results tell us it's better not to select embedding dimension too small. Because bigger embedding dimension can contain more information.

Figure 6.4 shows the varying embedding dimension on the score of WordSim-353 task. But we can not give a reasonable explanation for that. The possible reasons can be that the embedding dimension is not the key factor to affect this score and actually the difference is not so big comparing the score from other models, which we will introduce in the next section. We will do more experiments in the future to explain the above result.

Table 6.4: 16 Different Experiments in Step 2

id	d	$c1$	$c2$	$c3$	lr	gm	$S1$
1	300	200	2000	10000	0.1	0.9	true
2	250	200	2000	10000	0.1	0.9	true
3	200	200	2000	10000	0.1	0.9	true
4	150	200	2000	10000	0.1	0.9	true
5	100	200	2000	10000	0.1	0.9	true
6	50	200	2000	10000	0.1	0.9	true
7	50	200	2000	10000	0.2	0.9	true
8	50	200	2000	10000	0.05	0.9	true
9	50	200	2000	10000	0.01	0.9	true
10	50	200	2000	10000	0.1	0.95	true
11	50	200	2000	10000	0.1	0.85	true
12	50	200	2000	10000	0.1	0.8	true
13	50	20	2000	10000	0.1	0.9	true
14	50	20	2000	100000	0.1	0.9	true
15	50	20	7000	10000	0.1	0.9	true
16	50	20	2000	10000	0.1	0.9	false

Table 6.5: Different Vector Size Comparison

id	K	$t1$	$t2$	$t3$	$t4$	$iter$	$vLoss$	$SCWS$	$word353$
1	300	947.8	842	2272.9	79550	35	0.2437	0.5048	0.4823
2	250	764.7	533	1755.7	61450	35	0.2437	0.5083	0.4890
3	200	632.5	322	1389.9	55593	40	0.2436	0.5103	0.4921
4	150	502.7	210	1069.9	37448	35	0.2440	0.5048	0.4889
5	100	494.7	70.1	827.30	28956	35	0.2446	0.4994	0.4933
6	50	342.9	34.6	683.29	23915	35	0.2458	0.4666	0.4838

Different Min Count

We can find from Table 6.6 that the size of dictionary is not the important factor. A higher $c1$ (minimal count for the inclusion of a word in vocabulary D) 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. We think the above reason can also explain that their *iter* (the total number of training iteration) is same. As we know *iter* can imply the convergence speed. Removing infrequent words dose not influence the training of other

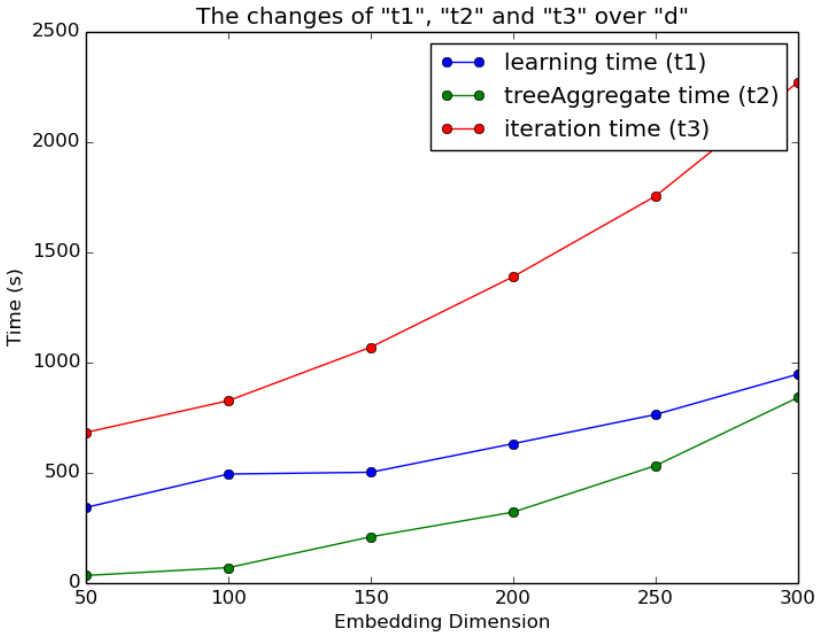

Figure 6.1: Shows the effect of varying embedding dimensionality of our model on the Time

Table 6.6: Different Min Count Comparison

id	$c1$	$t1$	$t2$	$t3$	$t4$	$iter$	$loss$	$SCWS$	$word353$
6	200	342.9	34.6	683.3	23915	35	0.2458	0.4666	0.4838
13	20	849.0	343	1838.1	64335	35	0.2457	0.4371	0.4293

words.

For the similarity tasks, experiment 6 has obviously better score on both datasets. Its $c1$ is bigger, accordingly its dictionary size is smaller, so that it focuses on those more frequent words and can obtain more meaningful information (some infrequent words may affect the final result). And the time ($t1$, $t2$, $t3$ and $t4$) from experiment 13 is much more, because it has much bigger size of dictionary (5 times of one in experiment 6). As we know from the last chapter, we can also check the Figure 5.1 and Figure 5.2: when $c1 = 20$, N (the size of dictionary D) is 458142; when $c2 = 200$, N is 95434.

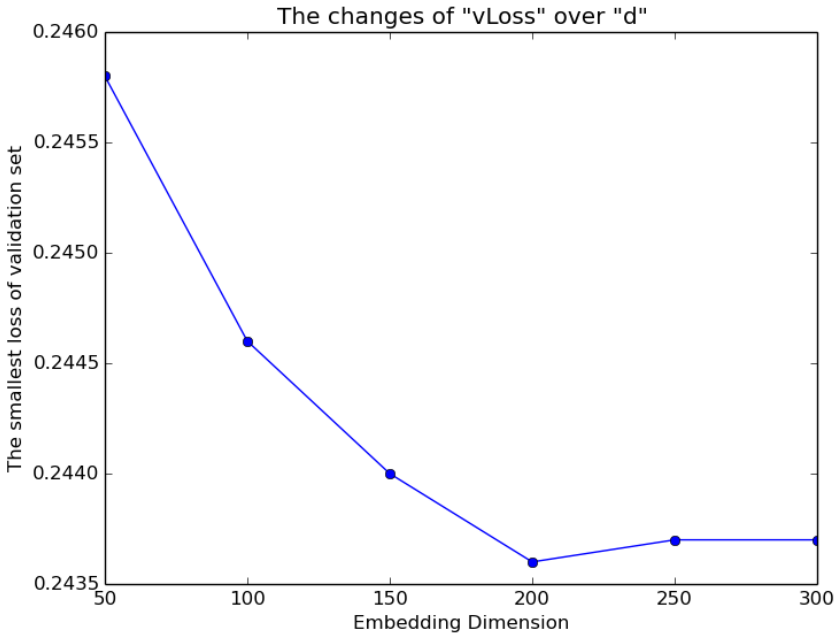

Figure 6.2: Shows the effect of varying embedding dimensionality of our model on the loss of validation set

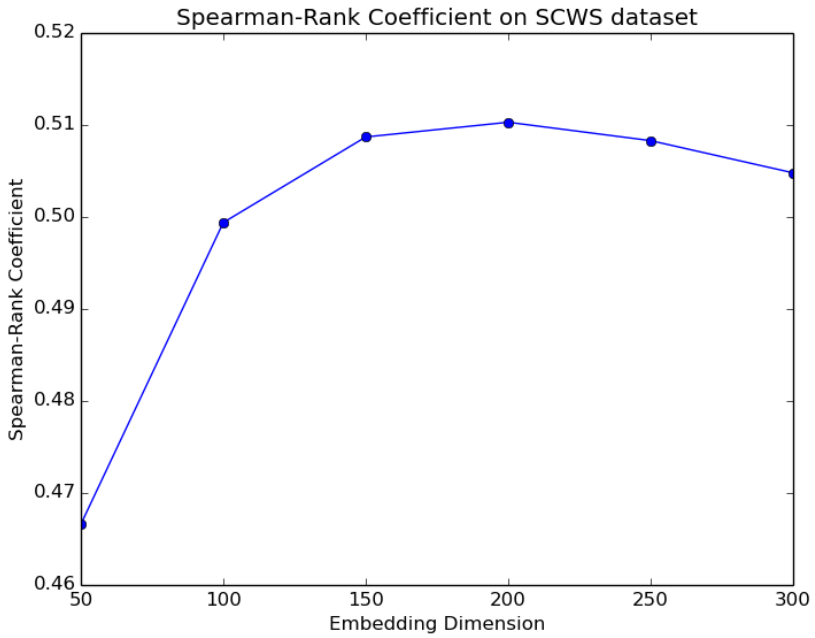

Figure 6.3: Shows the effect of varying embedding dimensionality of our model on the SCWS task

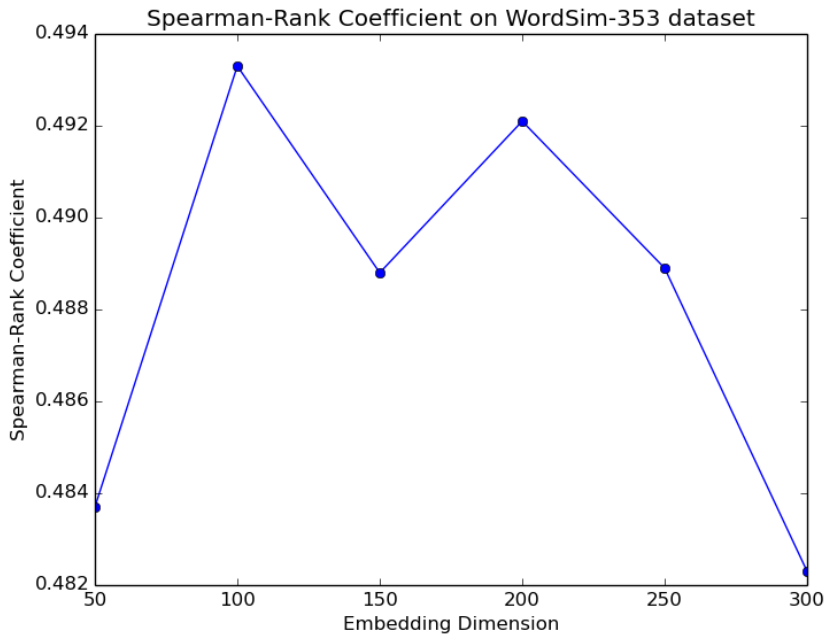

Figure 6.4: Shows the effect of varying embedding dimensionality of our model on the WordSim-353 task

Table 6.7: Different Sense Count Comparison

id	$c2$	$c3$	$t1$	$t2$	$t3$	$t4$	$iter$	$vLoss$	$SCWS$	$word353$
13	2000	10000	849	343	1838	64335	35	0.2457	0.4371	0.4293
14	2000	100000	798	338	1712	59912	35	0.2465	0.443	0.4375
15	7000	10000	808	340	1740	60909	35	0.2462	0.4351	0.4412

Different Sense Count Comparison
From Table 6.7, we can know the sense count is not the most important factor to affect the final loss. And their similarity task scores are also similar. Figure 6.5 shows the number of words for different number of senses per word. Comparing the running time of these three experiments, we can find that the time ($t1$, $t2$, $t3$ and $t4$) from experiment $id = 9$ are all less than the time from experiment $id = 7$, because they have the same number of words with one sense but experiment 9 has fewer words with sense 3. Similarly, the time of experiment 10 is also less than experiment 7, because they have the same number of words with three senses but experiment 10 has fewer words with two senses. Actually, more senses means more parameters, and the experiment with fewer parameters is faster. For the loss and similarity task scores, the difference is not so clear to analysis. We think we can do some experiments with more different number of senses and try more senses for each word in the future to find out how different number of senses influence the loss and similarity task scores.

Different Learning Rate and Gamma

Table 6.8 and Table 6.9 show that their time on one iteration ($t1$,$t2$ and $t3$) is almost same except the experiment 6 , which is very weird. The possible reasons about experiment 6 can be that we did experiment 6 earlier than other experiments and there may be some changes of hardware and environment which causes very different running time.

For the learning rate lr (the beginning learning rate) comparison, Figure 6.6, Figure 6.7 and Table 6.8 tell us bigger lr can get smaller $vLoss$ (the best loss of validation set) but requires more training iterations.

For the gamma gm (the reduction factor of learning rate) comparison, Figure 6.8, Figure 6.9 and Table 6.9 show that bigger gm can get smaller $vLoss$ (the best loss of validation set) but requires more training iterations.

In short, if we want to get smaller $vLoss$, we should increase both lr and gm, but from above figures (6.6, 6.7, 6.8 and 6.9), we can see that sometimes $vLoss$ reduces very few and $iter$ increases a lot. To balance the running time and the final loss, we select $lr = 0.1$ and $gm = 0.9$.

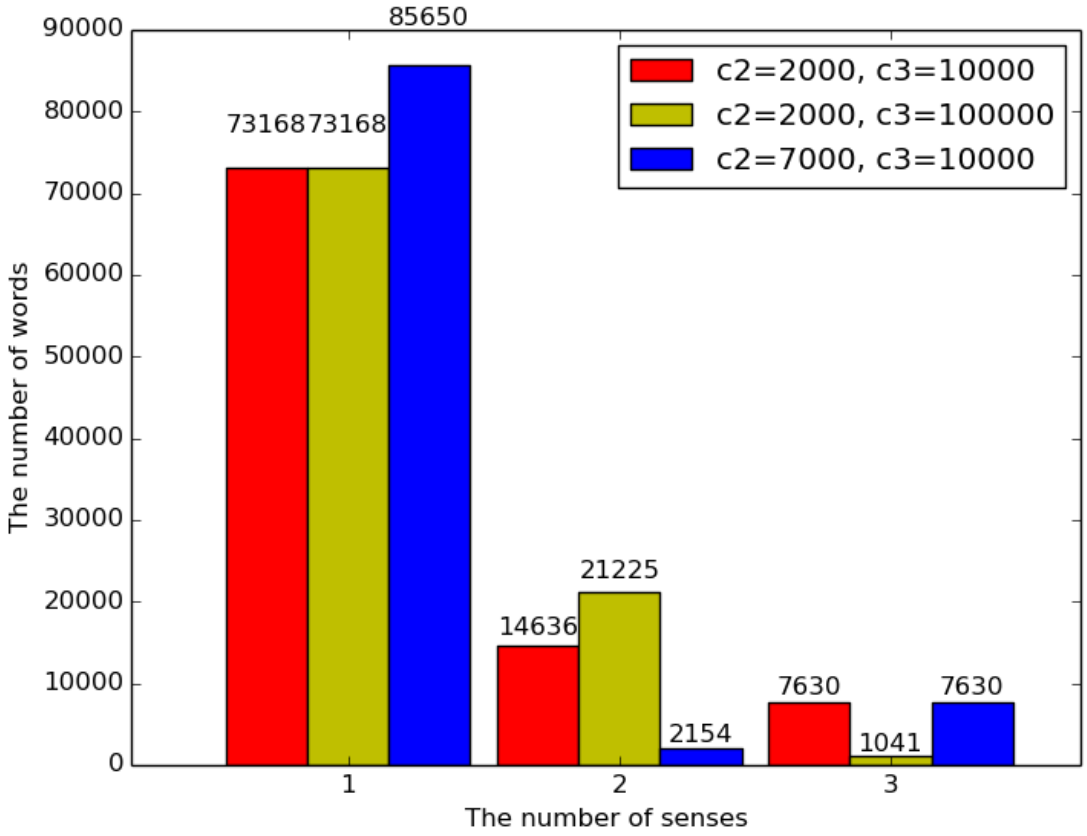

Figure 6.5: Shows the number of words with different number of senses from three experiments

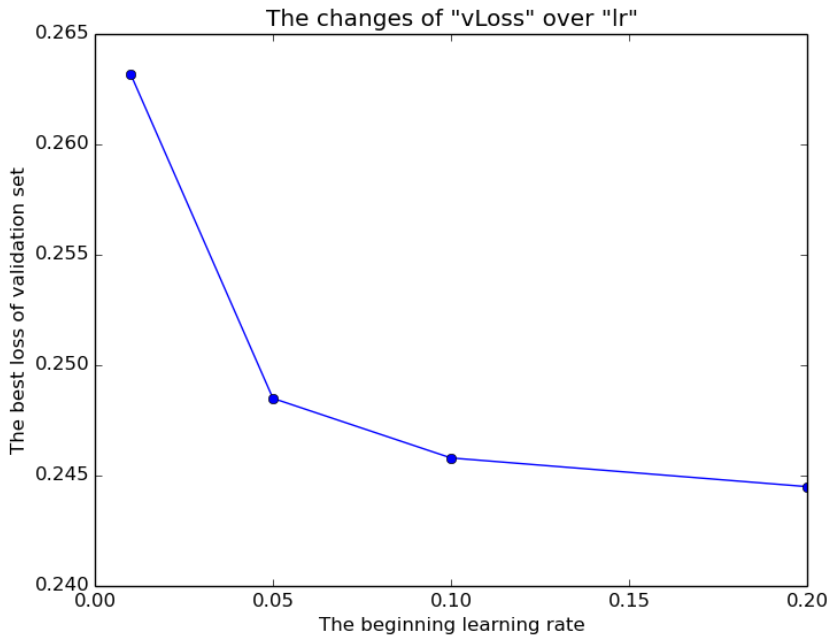

Figure 6.6: Shows the effect of varying beginning learning rate on the best loss of validation set

Table 6.8: Different Learning Rate Comparison

id	*lr*	*gm*	*t1*	*t2*	*t3*	*t4*	*iter*	*vLoss*
7	0.2	0.9	818.2	19.7	1416	63721	45	0.2445
6	0.1	0.9	342.9	34.6	683.3	23915	35	0.2458
8	0.05	0.9	789.1	18.4	1367	41013	30	0.2485
9	0.01	0.9	745.7	19.0	1381	34516	25	0.2632

Table 6.9: Different Gamma Comparison

id	*lr*	*gm*	*t1*	*t2*	*t3*	*t4*	*iter*	*vLoss*
10	0.1	0.95	854.0	18.5	1402	77110	55	0.2443
6	0.1	0.9	342.9	34.6	683.3	23915	35	0.2458
11	0.1	0.85	768.9	19.8	1413	42402	30	0.2476
12	0.1	0.8	850.0	19.0	1479	36985	25	0.2490

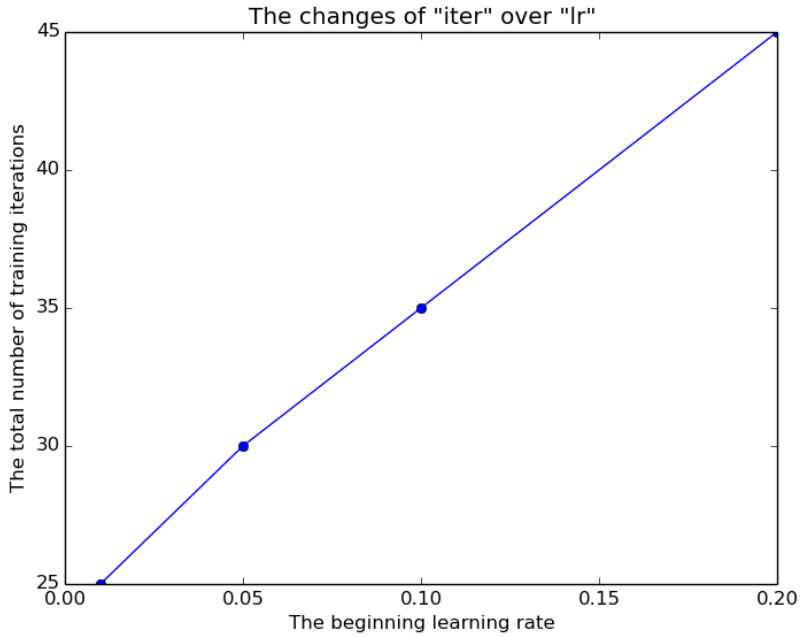

Figure 6.7: Shows the effect of varying beginning learning rate on the total number of training iterations

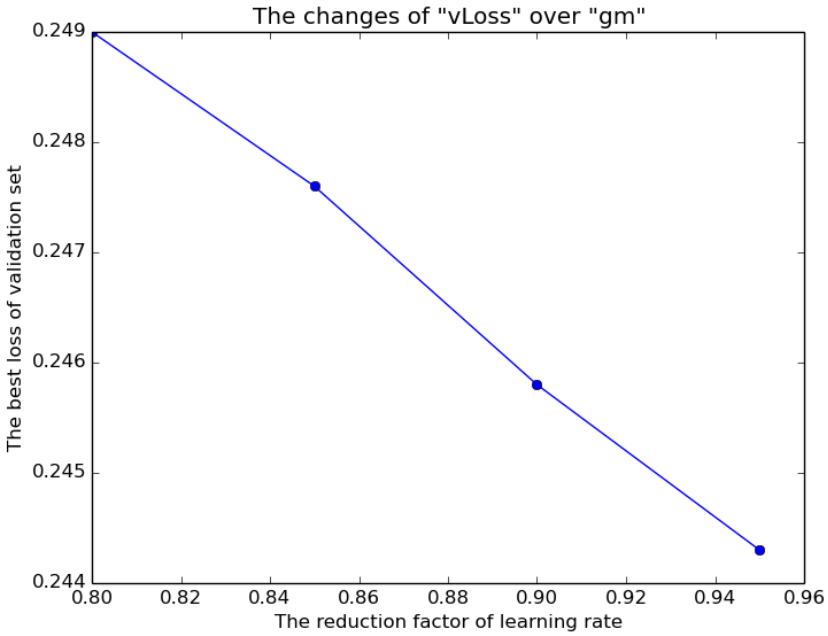

Figure 6.8: Shows the effect of reduction factor of the learning rate on the best loss of validation set

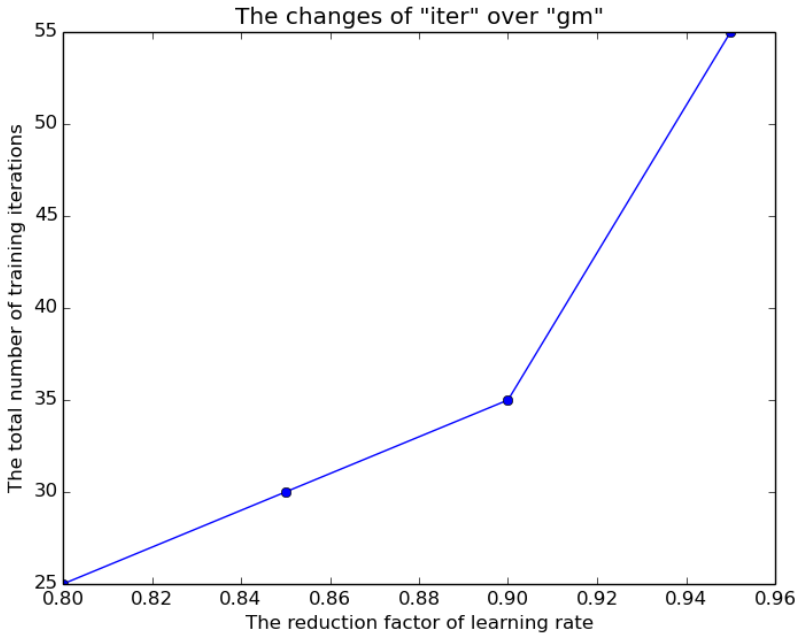

Figure 6.9: Shows the effect of reduction factor of the learning rate on the total number of training iterations

Table 6.10: Comparison of the different number of output senses

id	$S1$	$t1$	$t2$	$t3$	$t4$	$iter$	$vLoss$
13	true (one sense)	849	343	1838	64335	35	0.2457
16	false (multiple senses)	1192	365	2866	128949	45	0.2069

Different Number of Output Senses
From Table 6.10, we can find that the difference in this group is very obvious comparing with previous groups. But $t2$ (the average time of collecting parameters in one iteration) is almost same. Actually in our program to be able to change the number of output senses (one or multiple) easier, we do not change the data structure $syn1$ and let $syn1$ always have several embedding vectors for each word, if $S1 = true$ the program only process the first embedding vector for each word. So these two experiments have the same number of parameters. And the time of collecting parameters is only influenced by the number of parameters, that's why their $t2$ is very similar. Note that $t1$ (the average time of learning parameters in one iteration) of experiment 16 is a little bigger than one of experiment 13 and $t3$ (the average time of all operations in one iteration) of experiment 16 is much bigger than one of experiment 13. Multiple output embedding vectors for each word means more time to do sense assignment, specifically it requires more times of adjusting senses for each sentence to achieve stable, that's why $t3$ (including the time of sense assignment) is very different. In the process of learning parameters, no matter how many output senses, the the number of learning samples are same. So we can not tell the real reason about difference of $t1$. The possible reason can be from the program structure. We will analysis our program and do some testing experiments to find out the reason in the future.

The Table 6.10 also shows that the convergence speed of experiment 16 is slower (it has more training iterations) because it has several output senses and requires more iterations to adjust senses and learn embedding vectors. The $vLoss$ of experiment 16 is obviously smaller, because the fact of several output senses means more options for each center word to do prediction. It can make the final predicting probability based on the whole dataset bigger, which means smaller loss.

But we compare the nearest words for different senses of selected words from these two experiments in the Table 6.11. It is clear that if words can have multiple output embedding vectors (experiment 16), the nearest words of different senses for each word are similar, which can not achieve our goal. 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.

Table 6.11: Nearest words comparison

	id 13 , one sense output embedding	*id* 16, 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

6.1.1 Comparison to prior analyses

We fetch the results of similarity task scores from experiment 3 and experiment 6 to compare with other models in Table 6.12 and Table 6.13. Table 6.13 compares our model with Huang's model (Huang et al. [2012]), the model from [Collobert and Weston, 2008] (C&W), and the Skip-gram model [Mikolov et al., 2013], where C&W* is trained without stop words. Our result is very bad on WordSim-353 task. Our model may not be suitable for word similarity task without context information. Table 6.12 compares our model with Huang's model [Huang et al., 2012] , and the models from [Neelakantan et al., 2015a] (MSSG and NP-MSSG). The number after the model name is the embedding dimension, i.e. MSSG-50d means MSSG model with 50 embedding dimension.

Even that our score on SCWS is still not good. The possible reasons can be that our model do not remove the stop words, we do not use sub-sampling used word2vec (Mikolov et al. [2013]), our training is not enough and we uses too many executors (32 cores), where fewer executors may give us better results. Additionally, we only use $localSim$ and $avgSim$ for SCWS task and $avgSim$ for WordSim-353 task, which may not be suitable for our model. We will try other similarity functions. From Tabel 6.12, we can see NP-MSSG performs really good on SCWS task. We think we can also follow some idea from it and improve our model in the future so that the model can have dynamic number of senses.

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 13 from above.

Table 6.12: Experimental results in the SCWS task. The numbers are Spearmans correlation $\rho \times 100$

Model	avgSim	localSim
Our Model-50d	55.8	46.7
Our Model-300d	56.9	50.5
Huang et al-50d	62.8	26.1
MSSG-50d	64.2	49.17
MSSG-300d	67.2	57.26
NP-MSSG-50d	64.0	50.27
NP-MSSG-300d	67.3	59.80

Table 6.13: Results on the WordSim-353 dataset

Model	$\rho \times 100$
Our Model-50d	48.4
Our Model-300d	48.2
C&W*	49.8
C&W	55.3
Huang et al	64.2
Skip-gram-300d	70.4

Firstly we give the result for the word *apple*, where different sense are quite nicely separated. Table 6.14 shows the sense similarity matrix of *apple*. The similarity value is the cosine similarity between two embedding vectors. Table 6.15 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 6.16. 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.

Table 6.14: 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 6.15: 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

Table 6.16: 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

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 6.10.

Figure 6.10: Nearest words from *apple*

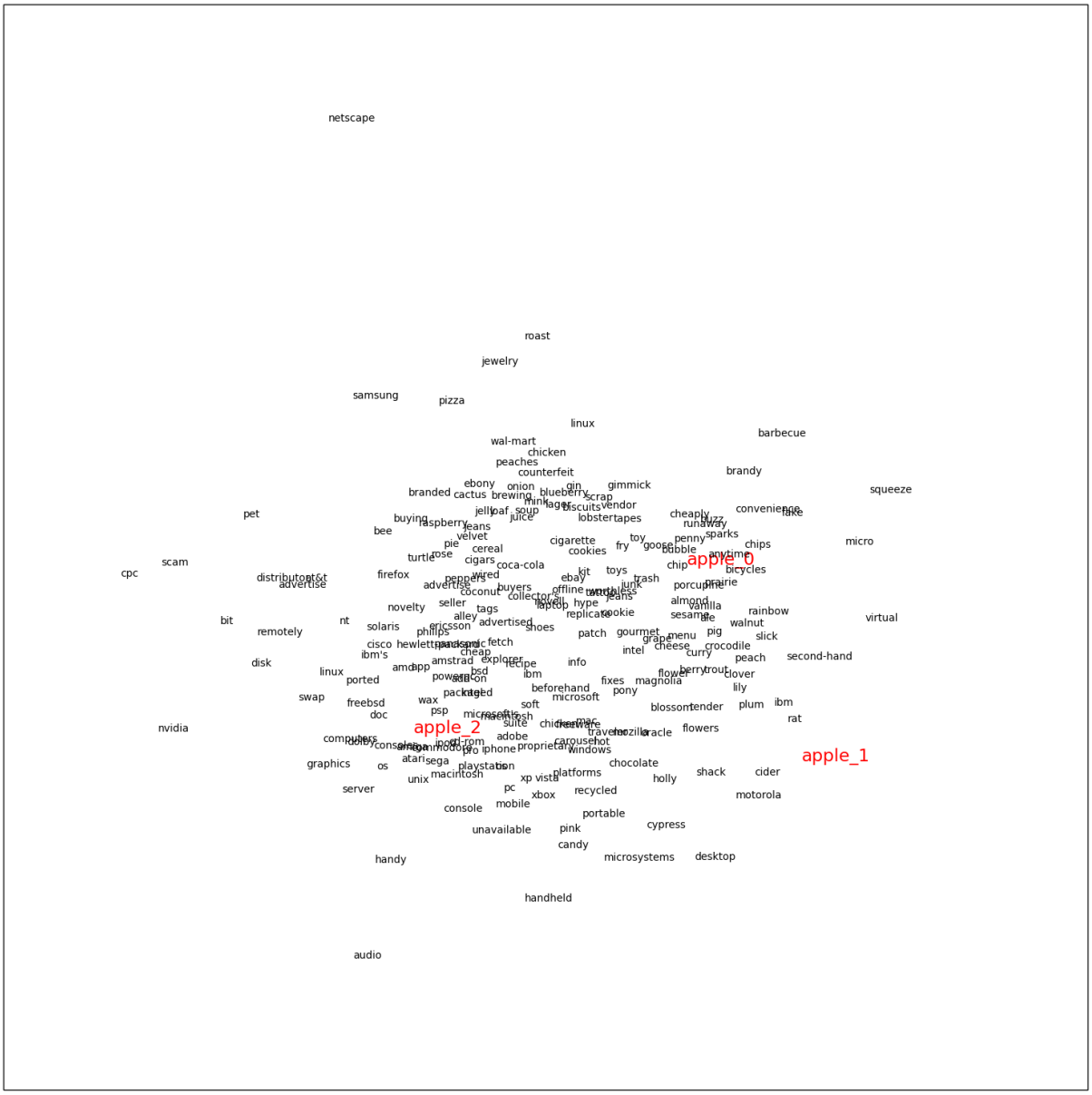

Table 6.17: Nearest words from *fox* , *net* , *rock* , *run* and *plant*

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 6.17. 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 6.18 we show one example sentence for each sense. The example sentences are also cut by ourself without affecting the meaning of the sentence.

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 6.11.

Table 6.18: 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.

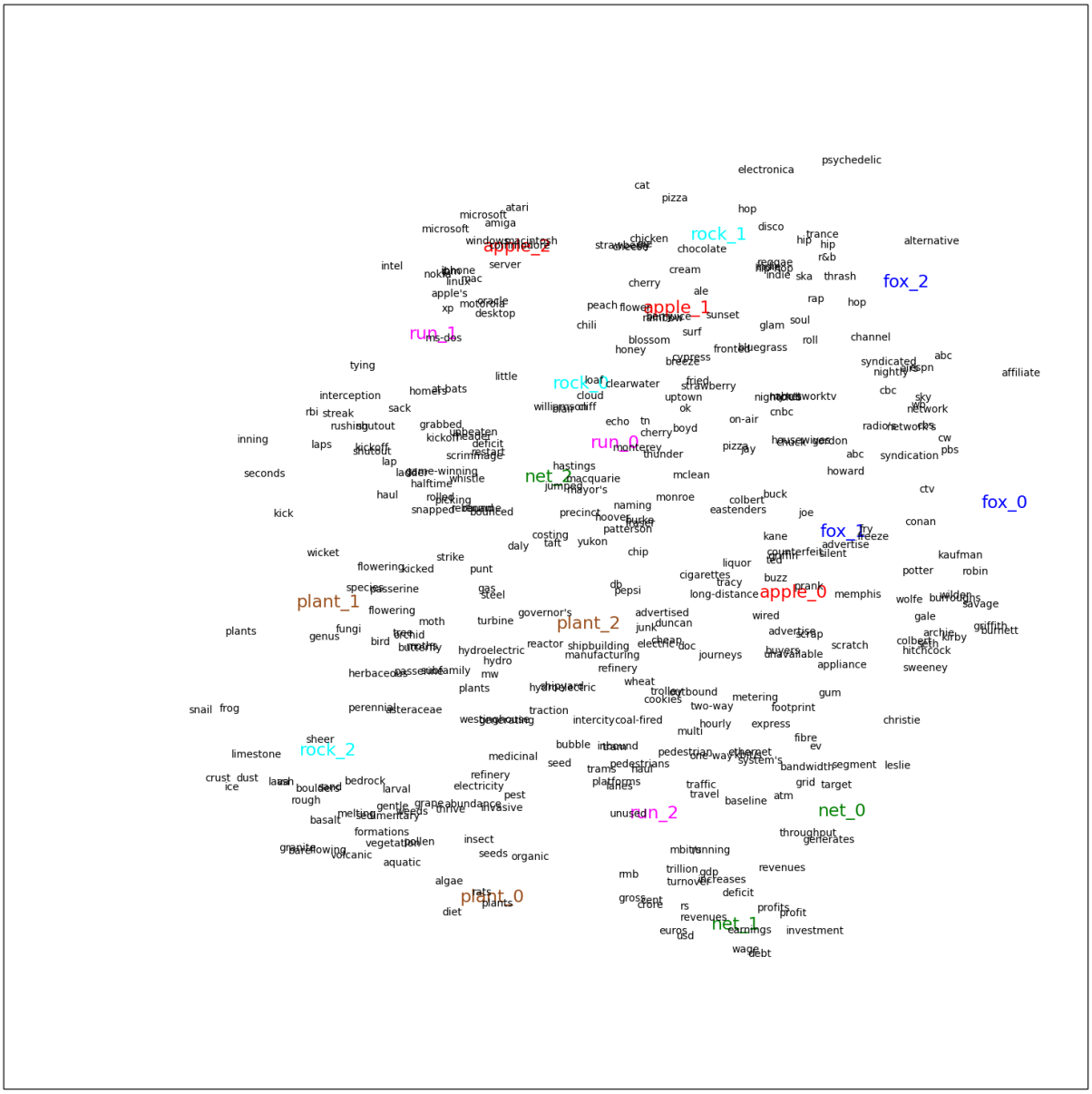

Figure 6.11: Nearest words from *apple*, *fox*, *net*, *rock*, *run* and *plant*

Chapter 7

Conclusion

To conclude this paper, In chapter 2, we introduced several word embedding methods including the details of gradient calculation in skip-gram model with negative sampling. In Chapter 3, we introduced three different sense embedding models. Based on these models, we described our mathematical model to generate sense embedding vectors in Chapter 4 and introduced the implementation using Spark in Chapter 5. After that we compared different experiments and analyzed different hyper-parameters with running time, loss of validation set and score of word similarity task. Originally our model assume that for each word both input embedding and output embedding have multiple senses. But the the experiment result told us, output would be better have only one sense. We displayed the nearest words of different senses from the same word. The result showed that our model can really derive expressive sense embeddings, which achieved our goal.

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 implementation is really efficient, that is also our goal.

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. And the max number of senses in our model is only three, we will more number of senses and try to extend our model so that it can decide the number of senses for each word as NP-MSSG ([Neelakantan et al., 2015a]). Further more, we think we can do more experiments for the same hyper-parameters in the future to make our results more reliable.

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.

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.

Shaoul, C. (2010). The westbury lab wikipedia corpus. *Edmonton, AB: University of Alberta*.

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*.