Master Thesis: Computing Distributed Representations for Polysemous Words

Haiqing Wang
Matriculation number: 340863

Advisors:
Dr. Gerhard Paaß
Dr. Jörg Kindermann

July 2, 2016

Contents

Chapter 1

Introduction

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

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

1.1 Distributed word representations

? Machine learning approaches for natural language processing have to represent the words of a language in a way such that Machine Learning modules may process them. This is especially important for text mining, where data mining modules analyze text corpora.

Traditional text mining analyses use the vector space representation [Salton et al., 1975], where a word is represented by a sparse vector of the size of the vocabulary (usually more than 100.000), where all values are 0 except the entry for the actual word. This representation is also called *One-hot representation*. This sparse representation, however, has no information on the semantic similarity of words.

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

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

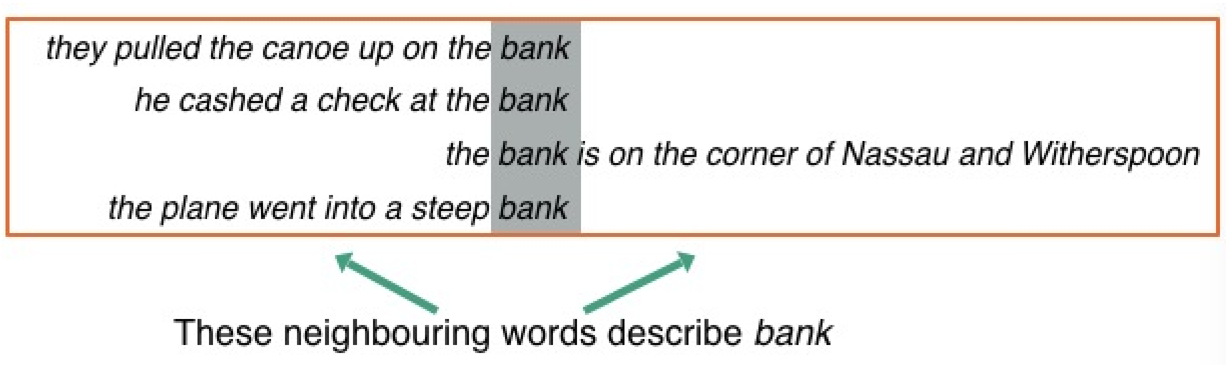
they pulled the canoe up on the bank
he cashed a check at the bank
the bank is on the corner of Nassau and Witherspoon
the plane went into a steep bank
These neighbouring words describe bank

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

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

Recently, artificial neural networks became very popular to generate lower-dimensional word embeddings. Prominent algorithms are *Senna* [Collobert and Weston, 2008], *Word2vec*

[1] COMMENT: please more details on this
[2] COMMENT: I do not understand that

[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 Polysemy

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

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

Further examples of polysemy are the words "book", "milk" or "crane". WordNet [Fellbaum, 1998] and other lexical resources show, that most common words have 3 to 10 different meanings. Obviously each of these meanings should be represented by a separate embedding vector, otherwise the embedding will no longer represent the underlying sense. This in addition will harm the performance of subsequent text mining analyses. Therefore we need methods to learn embeddings for senses rather than words.

Sense embeddings are a refinement of word embeddings. For example, "bank" can appear either together with "money", "account", "check" or in the context of "river", "water", "canoe". And the embeddings of "money", "account", "check" will be quite different from the embeddings of "river", "water", "canoe". Consider the following two sentences

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

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

[3]

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

In the last years a number of approaches to derive sense embeddings have been presented. Huang et al. [2012] used the clustering of precomputed one-sense word embeddings and their neighborhood embeddings to define the different word senses. The resulting word senses are fixed to the corresponding word neighborhoods and their values are trained until convergence. A similar approach is described by Chen et al. [2014]. Instead of a single embedding each word is represented by a number of different sense embeddings. During each iteration of the supervised training of Senna or Word2vec for each position of the word the best fitting embedding is selected according the fitness criterion[4]. Subsequently only this embedding is trained using back-propagation. Note that during training a word may be assigned to different senses thus reflecting the training process. A related approach was proposed by Tian et al. [2014].

It turned out that the resulting embeddings get better with the size of the training corpus and an increase of the dimension of the embedding vectors. This usually requires a parallel environment for the execution of the training of the embeddings. Recently *Apache Spark* [Zaharia et al., 2010] has been presented, an opensource cluster computing framework. Spark provides the facility to utilize entire clusters with implicit data parallelism and fault-tolerance against resource problems, e.g. memory shortage. The currently available sense embedding approaches are not ready to use compute clusters, e.g. by Apache Spark.

1.3 Goal and Organization of the Thesis

The main aim of this thesis is to derive expressive word representations for different senses in an efficient way. We will investigate sense assignment models which will extend known word embedding (one sense) approaches. Our goal is to implement such a method on a compute cluster using Apache Spark to be able to process larger training corpora and employ higher-dimensional sense embedding vectors. [5] Our main work will focus on the extension of Skip-gram model [Mikolov et al., 2013] in connection to the approach of [Neelakantan et al., 2015] because these models are easy to use, very efficient and convenient to train. [6] When using the Spark big data framework, we want to gain some experience and get feedback about the advantages and disadvantages of the new techniques.

[7] In the next chapter, we will introduce relative word embedding methods and sense embedding methods. We start with the neural language model and explain the early models of word embeddings. And then we focus on the word2vec Mikolov et al. [2013] especially about Skip-gram model. There will be many mathematical details including gradient

[4]COMMENT: Please reformulate sentence

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

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

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

calculation. After that, we will introduce two famous sense embedding models based the above word embedding works. The chapter 3 is our model description for sense embeddings. We use the spark framework to implement our model. The chapter 4 will introduce our implementation and show the experiment we did including parameter comparison and word senses visualization. At last chapter conclusion, we will analysis the advantages and disadvantages about our methods including model and implementation and give some ideas about how we can improve it in the future and what else we can do.

Chapter 2

Background and Related Works

2.1 Neural Probabilistic Language Model

This section will introduce a neural probabilistic language model from [Bengio]. Such model use a very important tool—Word Embedding. So what is the word embedding? General speaking, for any word w in the dictionary D, one can specify a fixed length of real-valued vector $v(w) \in \mathbb{R}^m$, $v(w)$ called the word embedding of w, and m is the length of word embedding. A further understanding about the word embeddings will be explained in the next section.

Since it is a neural probabilistic language model, it is obvious to use an neural network. Figure [neural4] shows the structure of the neural network, it include 4 layers: **Input** layer, **Projection** layer, **Hidden** layer and the Output layer. W and U are respectively the weight matrix between projection layer and hidden layer and the weight matrix between hidden layer and output layer, $\mathbf{p}$ and $\mathbf{q}$ are the offset vectors of respectively the hidden layer and the output layer.

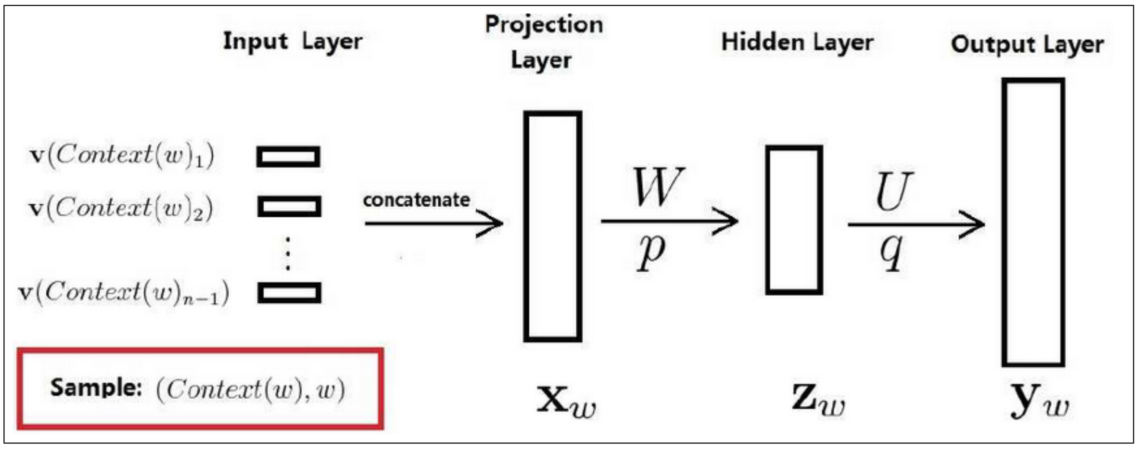

Figure 2.1: neural4[web1]

When talking about the above neural network, generally we consider it as a three-layer structure as following Figure [neural3]. But this thesis still use the structure of Figure [neural4]. On the one hand it is easy to describe, on the other hand it is more convenient to do comparison with the network structure in word2vec.

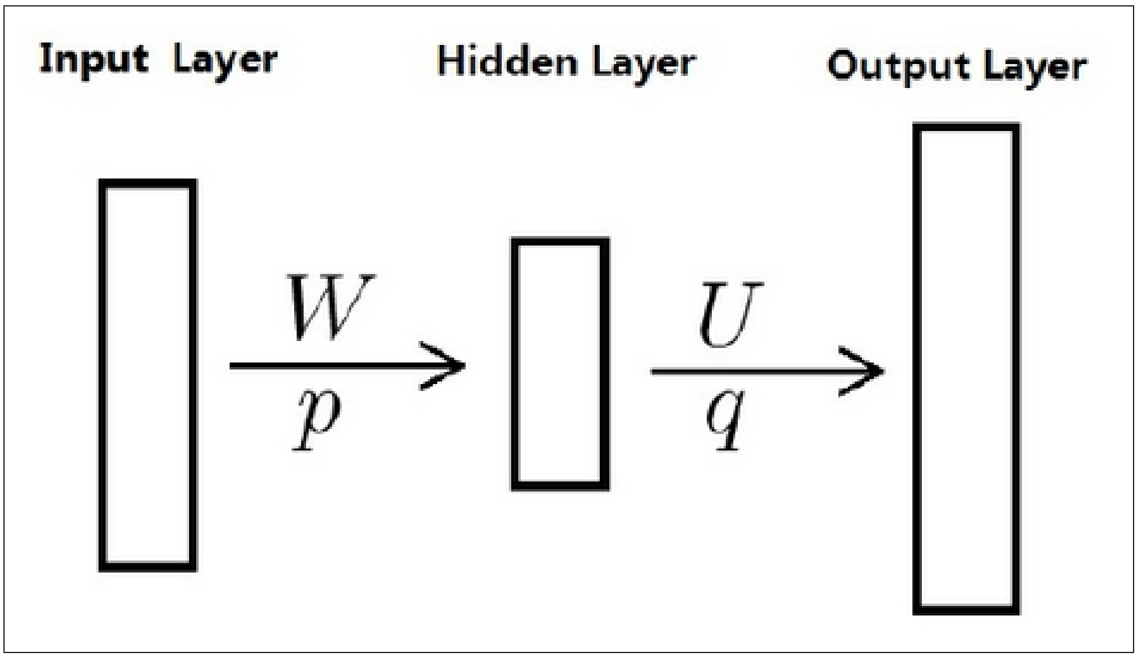

Figure 2.2: neural3[web1]

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

For any word w in corpus C , assuming $Context(w)$ takes its front $n-1$ words (similar to the n-gram), this binary pair $(Context(w), w)$ is a a training sample. So how is the sample $(Context(w), w)$ involved in computing through the neural network? Note that once word corpus C and vector length m is given, the scale of projection layer and the scale of output layer are determined. The former is $(n-1)m$, the latter is $N = |D|$, that is, the size of vocabulary. The size of the hidden layer n_h is the adjustable parameter which can be specified by the user.

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

$$\begin{cases} \mathbf{z_w} = \tanh(Wx_w + \mathbf{p}), \\ \mathbf{y_w} = Uz_w + \mathbf{q} \end{cases}$$

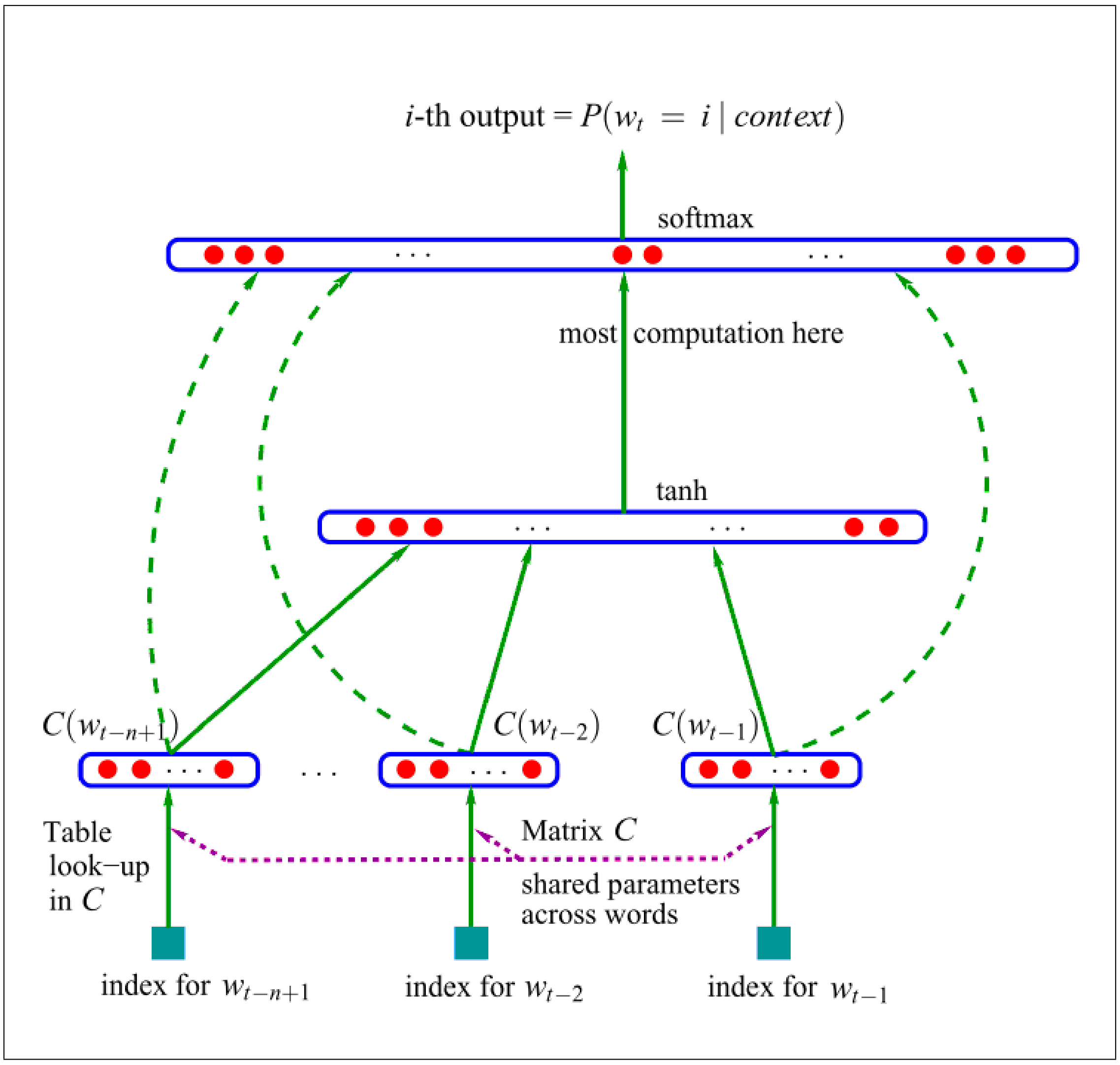

Figure 2.3: bengio[bengio]

where tanh is the Hyperbolic Tangent Function, used as the Active Function in the hidden layer. In the above formula, tanh acting on the vector represents acting each component of the vector. How about the number of first few words of a given sentence is less than $n-1$? Usually, we can artificially add some filler vectors, and they will also be involved in the training process.

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

$$p(w|Context(w)) = \frac{e^{y_{w,i_w}}}{\sum_{i=1}^{N} e^{y_{w,i}}}, \tag{2.1}$$

where i_w represents the index of w in the dictionary D.

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

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

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

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

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

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

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

1. Similarity between words can be be reflected in the word vectors.

   If in an (English) corpus, $S_1 =$ "A dog is running in the room" appears 10000 times, and $S_2 =$ "A cat is running in the room" only appears once. According to the n-gram model, $p(S_1)$ will certainly be much greater than $p(S_2)$. Note that the only difference between S_1 and S_2 is the "dog" and "cat" , but this two words play the same role either semantically or grammatically, so $p(S_1)$ and $p(S_2)$ should be very close.

   However, the probabilities $p(S_1)$ and $p(S_2)$ calculated by the neural network are approximately equal. The reason is that:

(a) In the neural network probabilistic language model, there is an assumption that the “similar ” words should have similar vectors

(b) The probability function on the word vectors is smooth, that is there is only very small influence for the probability when word vector change a little.

As a result, for the following sentences

A dog is running in the room

A cat is running in the room

The cat is running in a room

A dog is walking in a bedroom

The dog was walking in the room

...

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

2. Models based on the word vector have smoothing already (from (), we can know $p(w|Context(w)) \in (0,1)$ can not be 0), no longer need to carry the additional processing like n-gram model.

Finally, let's look back and think about what kind of role the word vector plays in the neural probability model. When training, it is just the auxiliary parameter used to construct the objective function; after the training, it seems just a by-product of the language model. However, this by-product can not be underestimated, the next section will be further elaborate its usefulness.

2.2 Understanding of the Word Embedding

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

One of the easiest word vector is one-hot representation, which is to use a long vector to represent a word, the vector's length is N, the size of dictionary D. It only has one component which is 1, and the other components are all 0s. The position of 1 corresponds to the index of the word in dictionary. But this word vector representation has some disadvantages, such as troubled by the huge dimensionality, especially when it is applied to deep learning scenes; another thing, it can not describe the similarity between words very well. Another word vector is Distributed Representation, it was firstly proposed by Hinton in 1986[], which can overcome the above drawbacks from one-hot representation.

The basic idea is to train the particular language to map each word into a short vector of fixed length (here “short” is respected to “long” in one-hot representation). All of these vectors constitute a vector space, and each can be regarded as a a point in the vector space. After introducing the “distance”in this space , it is possible to judge the similarity between words (morphology and syntax) according to the distance. Actually, word2vec uses this Distributed Representation for word vector.

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

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

A good word vector is valuable, for example, Ronan Collobert’s team makes use of the word vector from software package SENNA[SENNA] to do POS, CHK, NER and other tasks, and achieves good results. Google’s Tomas Mikolov team has developed an automatic generation technology for dictionary and glossary, which is able to convert one language into another language. The relation collection between words in each language, that is “language space” , can be characterized as a set of vectors in the mathematical sense. As long as the mapping and translation of a vector space to another vector space are realized, language translation can be realized. This technique has very good performance for translation between English and Spanish, with the accuracy rate up to 90%.

2.3 C&W’s Model

C&W’s original main purpose is not to generate a good word vectors, or even do not want to train the language model, but to use this word vectors to complete several tasks from natural language processing, such as speech tagging, named entity recognition, phrase recognition, semantic role labeling, and so on. Due to the different purpose, their training method is also different and special. They do not use language model’s idea like optimizing the probability $P(w_t|w_1, w_2, ..., w_{t-1})$, but directly use the score $f(w_{t-n+1}, ..., w_{t-1}, w_t)$ to determine if the sentence is reasonable and normal; low score illustrates the sentence is not reasonable; if you put a few words randomly together, it would be certainly a negative score. The score is just about high or low, not business with probabilities.

With the above assumption, C&W used the pair-wise method to train the word vectors. Specifically, it is to minimize the following objective function.

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

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

The structure of f is almost save as the network structure from Bengio 2003 []. The same thing is connecting n word vectors together to get a long vector and passing through one layer (a matrix multiplication) to get the hidden layer. The difference is that C&W's output layer has only one node representing the score, rather than Bengio's $|V|$ nodes. Doing so greatly reduced the computational complexity. Of course, C&W does not want to make a real language model, but just use some idea from the language model to assist them to complete other tasks in NLP.

Specifically, they give two different neural network structures window approach and sentence approach, shown as figure [SENNA]. Window approach is a feedforward neural network including a linear layer, HardTanh layer. Its input is the the vector concatenated by all all word vectors within the current word window including itself. Window approach is able to deal with most of natural language processing tasks, but has very poor performance on SRL taks. There, they proposed sentence approach to solve such problem. It is convolutional neural network structure. Apart from the linear layer and HardTanh layer, it has another convolutional layer and Max layer.

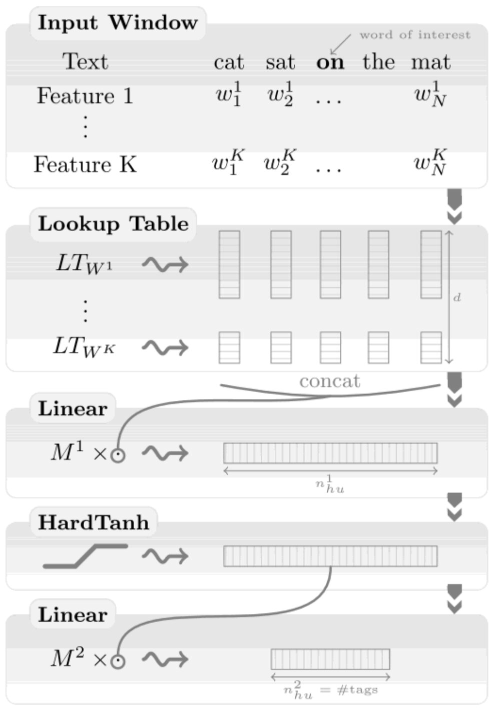

Figure 2.4: window approach[senna]

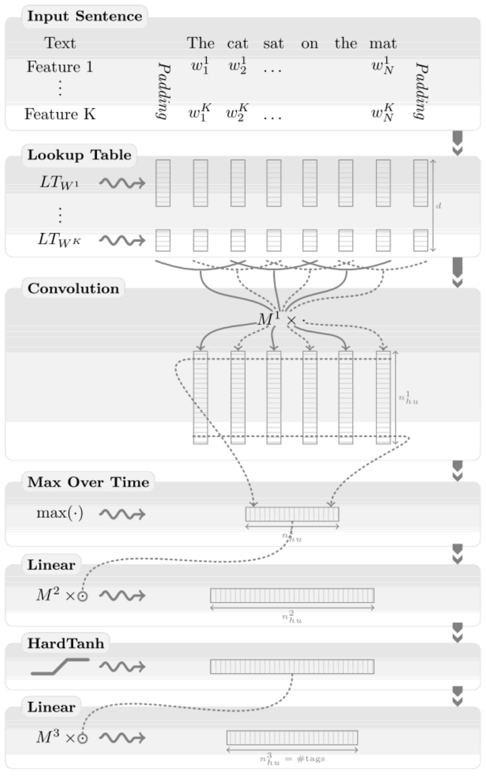

Figure 2.5: sentence approach[senna]

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

2.4 Word2Vec

This section will introduce two important model in word2vec: CBOW model (Continuous Bag-of-Words Model) and Skip-gram model (Continuous Skip-gram Model).

From the figure, two models both include three layers: **Input Layer**, **Projection Layer**, **Output Layer**. The former is to predict the current word w_t giving its context w_{t-2},w_{t-1},w_{t+1},w_{t+2}

With the foregoing preparation, this section describes word2vec officially used in two important models –CBOW model (Coutinuous Bag-of-Words Model) and Skip-gram model (Continuous Skip-gram Model). About two models, author Tomas Mikolov in [] shows the schematic diagram shown in Figures 8 and 9. Be seen by the two models contain three layers: **Input layer**, **projection layer** and **output layer**. The former is known in the current word w_t context w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2} premise predictive current word w_t (see Figure 8); and the latter on the contrary, it is known in the current word w_t premise predict its context w_{t-2}, w_{t-1} , w_{t+1}, w_{t+2} (see Figure 9). For two CBOW and Skip-gram model, word2vec given two frameworks, which are based on Hierarchical Softmax and Negative Sampling to design. This section describes the Hierarchical Softmax CBOW and Skip-gram model. In the previous section, we mentioned that the objective function neural network based language model is generally taken as follows **log-likelihood function**

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

The key is the conditional probability function $p(w|Context(w))$ configuration, text [] in this model is given a construction method function (see (3.6) formula). For the objective function Hierarchical Softmax CBOW word2vec model based on optimized also the form (4.1); and for the objective function based on Hierarchical Softmax of Skip-gram model, the optimization of the form

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

Therefore, the discussion process, we should focus on the $p(w|Context(w))$ or $p(Context(w)|w)$ on the structure, realize that this is very important because it allows us to targeted, distractions, and will not fall into some of the tedious details were to go. Next, we will focus on the Skip-gram model with negative sampling and explain some mathematical details, because our model is based that.

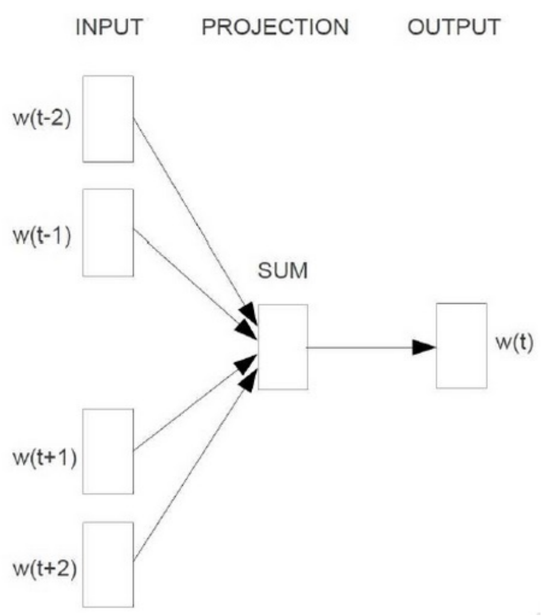

Figure 2.6: CBOW model

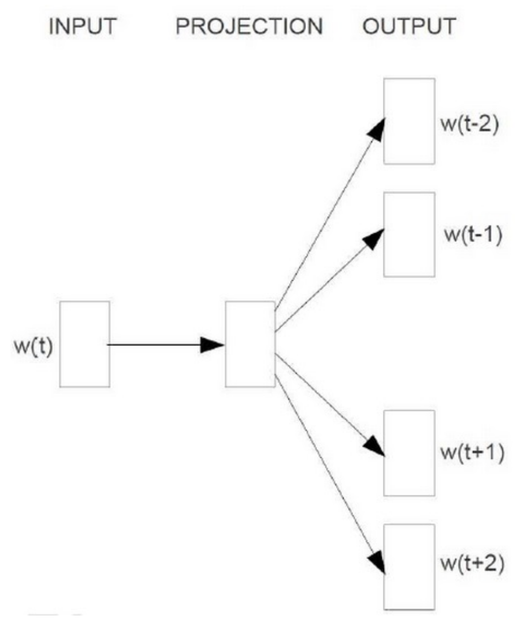

Figure 2.7: Skip-Gram model

2.4.1 Skip-gram model with Hierarchical Softmax

This section describes word2vec another model – Skip-gram model, since the derivation and CBOW similar, and therefore will inherit the measure introduced mark.

Figure 12 shows the network structure of Skip-gram model, with network structure CBOW model, it also includes three layers: an input layer, a projection layer and output layer. The following sample $(w, Context(w))$, for example, three layers are described briefly.

1. **input layer**: the center of the current sample containing only the word w word vector $\mathbf{v}(w) \in \mathbb{R}^m$.
2. **projection layer**: This projection is identical to $\mathbf{v}(w)$ projection to $\mathbf{v}(w)$. Therefore, **this projection layer is actually superfluous** reason here mainly to facilitate retention projection layer and network structure CBOW models do comparison.
3. **Output layer**: and CBOW model, the output layer is also a lesson Huffman tree.

gradient calculation

For Skip-gram model, it is known that the current word w, need to predict its context $Context(w)$ of the words, the objective function should therefore form (4.2), and the key is the conditional probability function $p(Context(w)|w)$ configuration, in the Skip-gram model which is defined as

$$p(Context(w)|w) = \prod_{u \in Context(w)}^{p(u|w} ,$$

In the above formula $p(u|w)$ in accordance with section describes the Hierarchical Softmax thought, similar to (4.3) written as

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

among them

$$p(d_j^u|\mathbf{v}(w), \theta_{j-1}^u) = [\theta(\mathbf{v}(w)^{\mathrm{T}}\theta_{j-1}^u)]^{1-d_j^u} \cdot [1 - \theta(\mathbf{v}(w)^{\mathrm{T}}\theta_{j-1}^u)]^{1-d_j^u} \tag{2.2}$$

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

$$\mathcal{L} = \sum_{w \in \mathcal{C}} \log \prod_{u \in Context(w)} \prod_{j=2}^{l^u} \{[\theta(\mathbf{v}(w)^{\mathrm{T}}\theta_{j-1}^u)]^{1-d_j^u} \cdot [1 - \theta(\mathbf{v}(w)^{\mathrm{T}}\theta_{j-1}^u)]^{\ d_j^u}\}$$

$$= \sum_{w \in \mathcal{C}} \sum_{u \in Context(w)} \sum_{j=2}^{l^u} \{(1 - d_j^u) \cdot \log[\theta(\mathbf{v}(w)^{\mathrm{T}}\theta_{j-1}^u)] + d_j^u \cdot \log[1 - \theta(\mathbf{v}(w)^{\mathrm{T}}\theta_{j-1}^u)]\}. \tag{2.3}$$

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

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

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

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

2.4.2 Skip-gram with Negative Sampling

Negative Sampling (NEG) is proposed by Tomas Mikolov et al.[word2vec] , which is the simplified version of NCE(Noise Contrastive Estimation), the purpose is to improve the training and the quality of word vectors. Comparison with Hierarchical Softmax, NEG do not use the Huffman tree. Instead, it use Random Negative Sampling, which can improve the performance much.

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

The objective function is:

$$G = \prod_{w\in\mathcal{C}}\prod_{u\in Context(w)} g(u), \tag{2.4}$$

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

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

where $NEG(u)$ represents the negative samples generated by u, the conditional probability

$$p(z|w) = \begin{cases} \sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z), & L^u(z)=1; \\ 1-\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z), & L^u(z)=0; \end{cases}$$

where

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

It can also be written as one expression

$$p(z|w) = [\sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)]^{L^u(z)} \cdot [1 - \sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)]^{1-L^u(z)} \tag{2.5}$$

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

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

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

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

And next, let's use **Stochastic gradient ascent method** to optimize it. The point is to calculate two kinds of gradient. Let's consider the gradient θ^z firstly.

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

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

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

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

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

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

$$\begin{aligned}
\mathbf{v}(w) &:= \mathbf{v}(w) + \eta \sum_{z\in\{u\}\cup NEG\{u\}} \frac{\partial \mathcal{L}(w, u, z)}{\partial \mathbf{v}(w)} \\
&= \mathbf{v}(w) + \eta \sum_{z\in\{u\}\cup NEG\{u\}} [L^u(z) - \sigma(\mathbf{v}(w)^{\mathrm{T}}\theta^z)]\theta^z.
\end{aligned}$$

2.5 Huang's Model

Eric H. Huang's work[] is based on the model from C&W []. The goal of his working is about trying to make the word vectors with richer semantic information than other models. He 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 think C&W's work uses only "local context". In the process of training vectors, C&W used only 10 words as the context for each word, counting the center word itself, there are totally 11 words' information. These local information can not fully exploit the semantic information of the center word. Huang used C&W's 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.

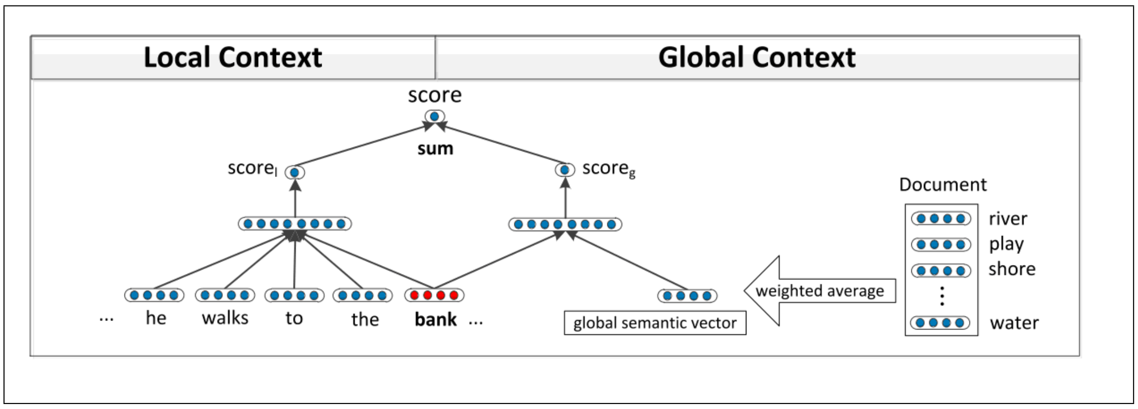

Figure 2.8: huang[huang]

The second contribution of this paper is to represent polysemy using multiple word vectors. Bengio 2003 [] also mentioned that this would be an very important issue, but he was still looking for a solution, now Huang gives an idea. For each center word, he took

10 nearest context words and calculated the weighted average of these 10 word vectors (idf weights) as the context vector. Huang used all context vectors to do k-means clustering, and relabel each word based on the clustering results (different classes of the same words would be considered as different words to process), and finally re-trained the word vectors. The following gives some examples from his model's results.

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

2.6 EM-Algorithm based method

There is a very famous method based on the EM-algorithm from []. This method is the extension of 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 can represent if a sense is used frequent in the corpus. For example, considering $bank_1$ and $bank_2$, $bank_1$ represents the side of the river with the smaller probability and $bank_2$ means the institute about money with the higher probability. We can say in the corpus, in most sentences of the corpus the word "bank" means the institute about money and in other fewer cases it means the 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. The following are some results from this model.

Most Similar Words

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

Chapter 3

Model

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

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

3.1 Definition

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

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

3.2 Introduction

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).

3.3 Objective Function

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

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

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

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

Learn.

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

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

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

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

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

Here the loss is defined as the negative log probability.

And the loss function of whole corpus is

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

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

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

Use

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

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

Use stochastic gradient descent:

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

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

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.

3.4 Algorithm Description

Initialization:

$$h_{i,j} = R(N_{w_{i,j}}),\ 1 \le i \le M,\ 1 \le j \le L_i$$
$$V_{w,s} = \Big[\underbrace{\frac{R() - 0.5}{d}, \ldots, \frac{R() - 0.5}{d}}_{d}\Big]^{\mathrm{T}},\ w \in D,\ 1 \le s \le N_w$$
$$U_{w,s} = \Big[\underbrace{0, \ldots, 0}_{d}\Big]^{\mathrm{T}},\ w \in D,\ 1 \le s \le N_w$$

where $R(x)$ means generating a random number (integer) from 1 to x, and $R()$ is to generate a random number (real) from 0.0 to 1.0.

Assign:

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

Learn:

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

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

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

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

            END
        END
    END
END
```

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

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

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

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

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

Iterating between **Assign** and **Learn** till the convergence of the value of G makes the whole algorithm complete.

Chapter 4

Experiment

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

4.1 Spark

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

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

operation is mainly about map and filter, which is very similar as the operations in any functional programming. And the action operation is mainly about aggregate, reduce, count, and collect. RDD can be operated only by transformation operation and stored logically in every executor. Data in RDD can not be changed. Spark use some operations to Each transformation operation will create a new RDD and would not change the DATA in the original RDD.

4.2 Implementation

Parameters
We use $syn0$ to represent the input embedding V and $syn1$ to represent the output embedding U. $syn0$ and $syn1$ are set to be as the "global" value, which the executors can not change them directly. Remember, executors

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

Learn Step
In the training, we also use map transformation. Instead of transforming sentences to sentences, we transform the original sentence RDD into the collection of updated $syn0$ and updated $syn1$. Yes, we update $syn0$ and $syn1$ in this process, because we need to train our parameters. But we do not really change the global $syn0$ and $syn1$. Instead, we copy the global $syn0$ and $syn1$ (broadcast value) to the local $syn0$ and $syn1$ in each executor, so that each executor has its own $syn0$ and $syn1$ and update them independently. And then we use the average of them as the new global $syn0$ and $syn1$.

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

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

them to the new embeddings with length of 4.

4.3 Experiment

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

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

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

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

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

4.3.1 Parameters Comparison

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

We did four different experiments for the first step, based on that we did another then 9 experiments for the second step. In the following, we will fix some parameters and build 5 comparison groups based these 9 experiments to check how different parameters affect the final validation loss, the convergence speed, training time and similarity results.

Group 1 (different vec) c1=200, cm=2000_10000, lr=0.1, gm=0.9, syn1=true

id	vec	t1	t2	iter	t3	t4	loss	SCWS	word353	init
3	50	342.91	34.6	35	683.29	23915.12	0.2458	0.4666	0.5449	(2)
1	100	494.7	70.1	35	827.30	28955.58	0.2446	0.4994	0.5355	(1)
10	300	947.79	842	35	2272.9	79550.23	0.2437	0.5048	0.5233	(1)

Group 2 (different c1) vec=50, cm=2000_10000, lr=0.1, gm=0.9, syn1=true

id	c1	t1	t2	iter	t3	t4	loss	SCWS	word353	init
3	200	342.9	34.6	35	683.29	23915.12	0.2458	0.4666	0.5449	(2)
8	20	849.0	342.7	35	1838.14	64335.04	0.2457	..	..	(3)

Group 3 (different cm) c1=20, vec=50, lr=0.1, gm=0.9, syn1=true

id	cm	t1	t2	iter	t3	t4	loss	SCWS	word353	init
8	2000_10000	849	343	35	1838	64335	0.2457	..	..	(3)
5	2000_100000	798	338	35	1712	59912	0.2465	0.443	0.498	(3)
6	7000_10000	808	340	35	1740	60909	0.2462	0.4351	0.506	(3)

Group 4 (different lr and gm)

c1=20, vec=50, cm=2000_10000, syn1=true

id	lr	gm	t1	t2	iter	t3	t4	loss	SCWS	word353	init
8	0.1	0.9	849	343	35	1838	64335	0.246	..	..	(3)
9	0.01	0.95	797	370	40	1851	74032	0.267	..	..	(4)

c1=20, vec=50, cm=2000_10000, syn1=false

id	lr	gm	t1	t2	iter	t3	t4	loss	SCWS	word353	init
4	0.1	0.9	1192	365	45	2866	128949	0.207	0.422	0.480	(3)
2	0.01	0.95	993	338	35	2491	87198	0.229	0.216	0.151	(4)

Group 5 (different syn1) c1=20, vec=50, cm=2000_10000, lr=0.1, gm=0.9

id	syn1	t1	t2	iter	t3	t4	loss	SCWS	word353	init
8	true	849	343	35	1838	64335	0.2457	..	..	(3)
4	false	1192	365	45	2866	128949	0.2069	0.4224	0.4802	(3)

Nearest words

word	id=8, syn1=true	id=4, syn1=false
apple	cheap, junk, scrap chocolate, chicken, cherry macintosh, linux, ibm	kodak, marketed, nokia portable, mgm, toy marketed, chip, portable
bank	corporation, banking, banking deposit, stake, creditors banks, side, edge	trade, trust, venture trust, corporation, trade banks, border, banks
cell	imaging, plasma, neural lab, coffin, inadvertently cells, nucleus, membrane	dna, brain, stem cells, dna, proteins dna, cells, plasma

4.3.2 Case Analysis

In the following, we will select only one experiment's result to do visualization and nearest words. The selection is based on the final loss and similarity task, specifically it is experiment 8 from above.

apple

sense matrix

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

nearest words

$apple_0$:	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

examples

- 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 in order to make the mash soft
- the cultivar is not to be confused with the dutch rubens $apple_1$ which is a cross of cox's orange pippin and reinette
- the rome beauty $apple_1$ was developed by joel gillette and his son alanson in rome township near proctorville in
- a list of all $apple_2$ internal and external drives in chronological order of introduction
- in may the game was made available for the $apple_2$ iphone os mobile platform

visualization

fox

sense matrix

	fox_0	fox_1	fox_2
fox_0	1.000000	0.798174	0.772341
fox_1	0.798174	1.000000	0.813265
fox_2	0.772341	0.813265	1.000000

nearest words

fox_0	archie	potter	wolfe	hitchcock	conan	burnett	savage
fox_1	buck	housewives	colbert	eastenders	howard	kane	freeze
fox_2	abc	sky	syndicated	cw	network's	ctv	pbs

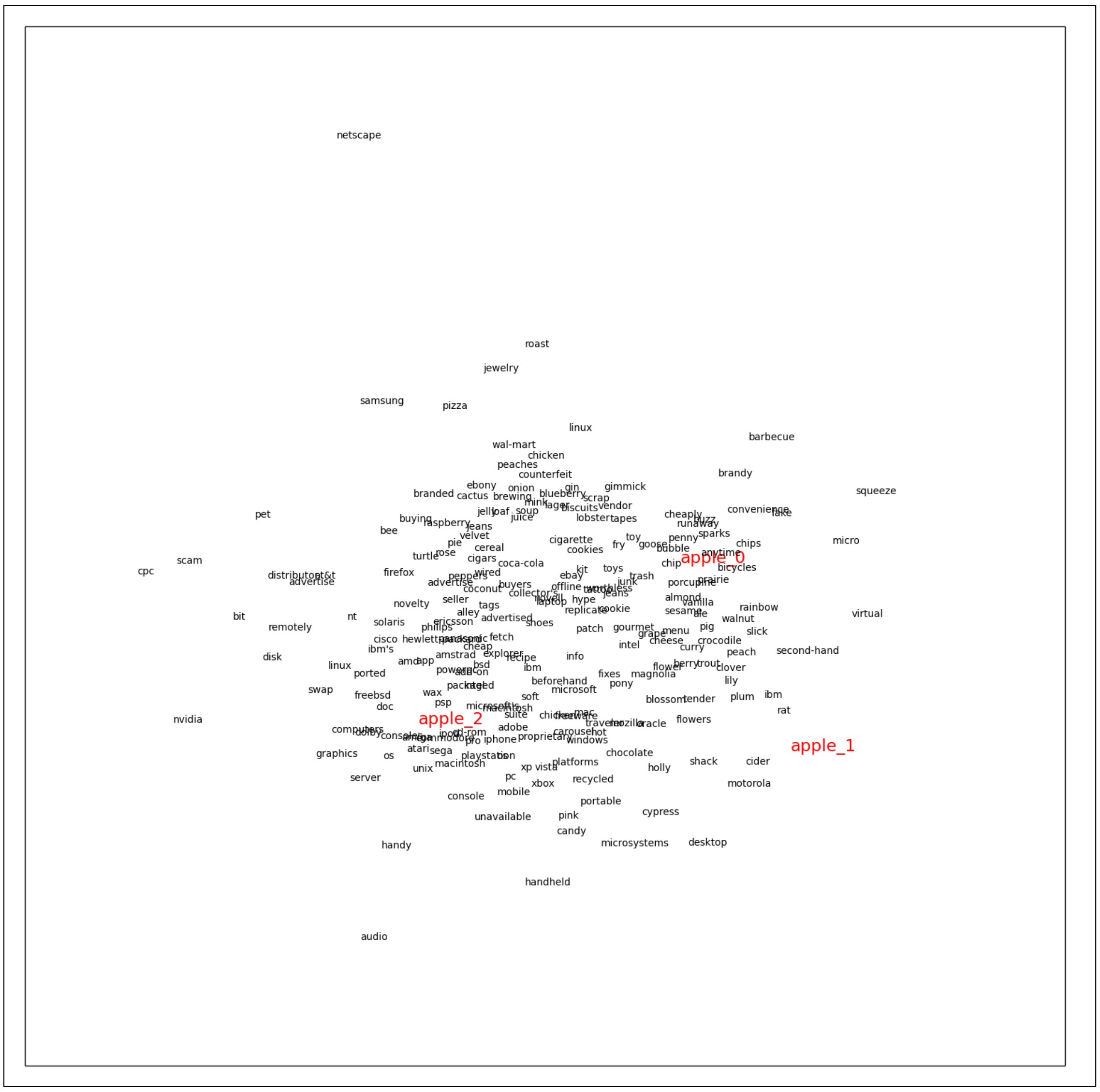

Figure 4.1: apple

examples

- ryan fox is an american rowing athlete with a very decorated list of achievements with the us rowing team
- junior aspirin records is run by nathaniel mellors dan fox andy cooke and ashley marlowe
- he can box like a fox he's as dumb as an ox
- aka the quick brown fox jumps over the lazy pig philippines english title long title

- the grand final was replayed on fox sports australia and the fox footy channel
- following the september attacks fox news reported that the west had recruited al-madani post

visualization

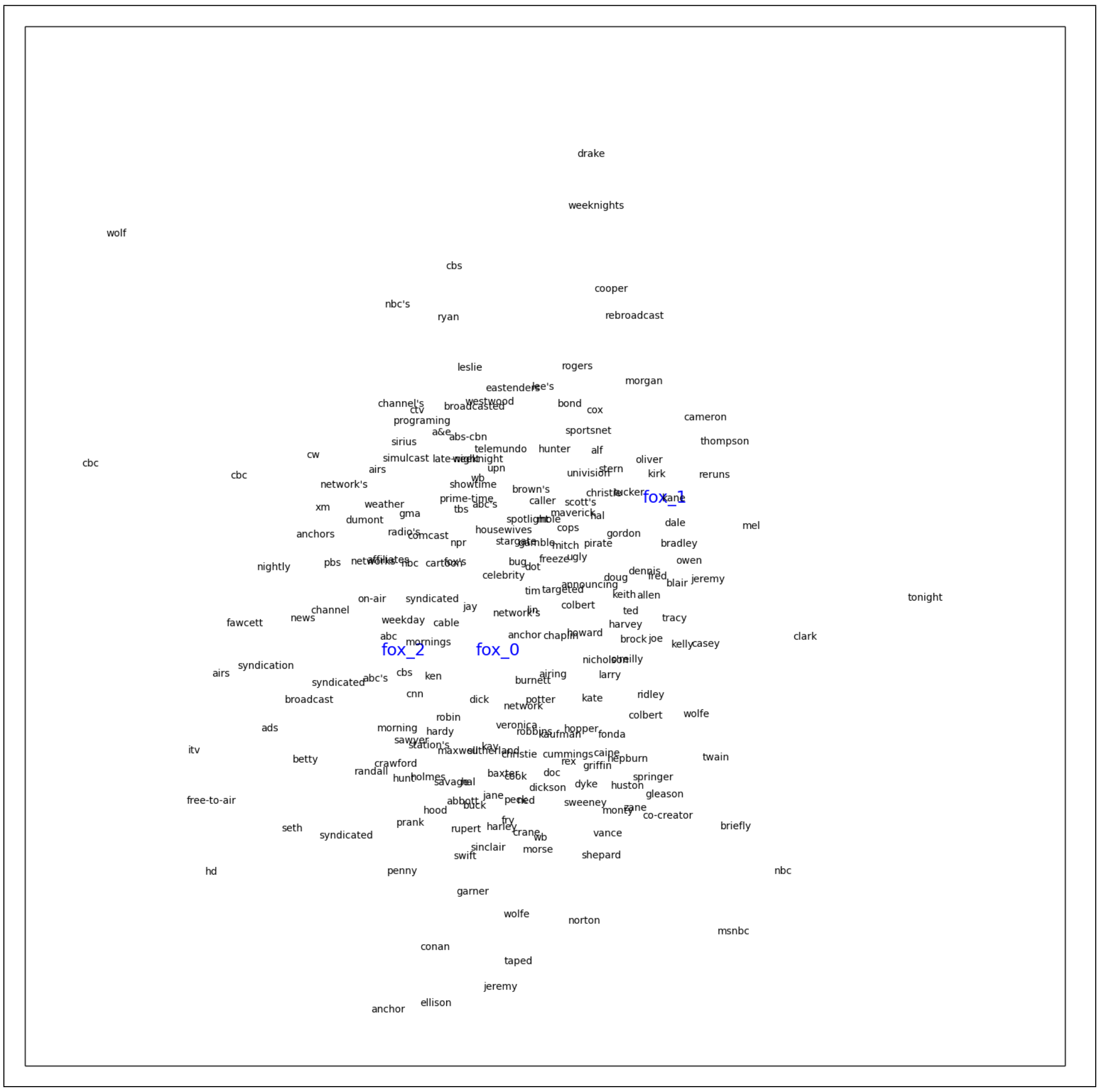

Figure 4.2: fox

net

sense matrix

	net_0	net_1	net_2
net_0	1.000000	0.851080	0.768358
net_1	0.851080	1.000000	0.779539
net_2	0.768358	0.779539	1.000000

nearest words

net_0:	generates	atm	footprint	target	kbit/s	throughput	metering
net_1:	trillion	rs	earnings	turnover	gross	euros	profit
net_2:	jumped	rolled	rebound	ladder	deficit	snapped	whistle

examples

- it took a much higher dose c.f mg/kg for the dat occupancy to approach the same as the net_0 and sert i.e saturation
- net_0 supports several disk image formats partitioning schemes and windows file systems
- in mr cook was on the forbes with a net_1 worth of billion
- an independent audit of children incorporated shows the total liabilities and net_1 assets as for the year
- on april a new storm worm was released onto the net_2 with april subject titles
- nothin but net_2 freefall feet into a net below story tower

visualization

rock

sense matrix

	$rock_0$	$rock_1$	$rock_2$
$rock_0$	1.000000	0.761824	0.734151
$rock_1$	0.761824	1.000000	0.679431
$rock_2$	0.734151	0.679431	1.000000

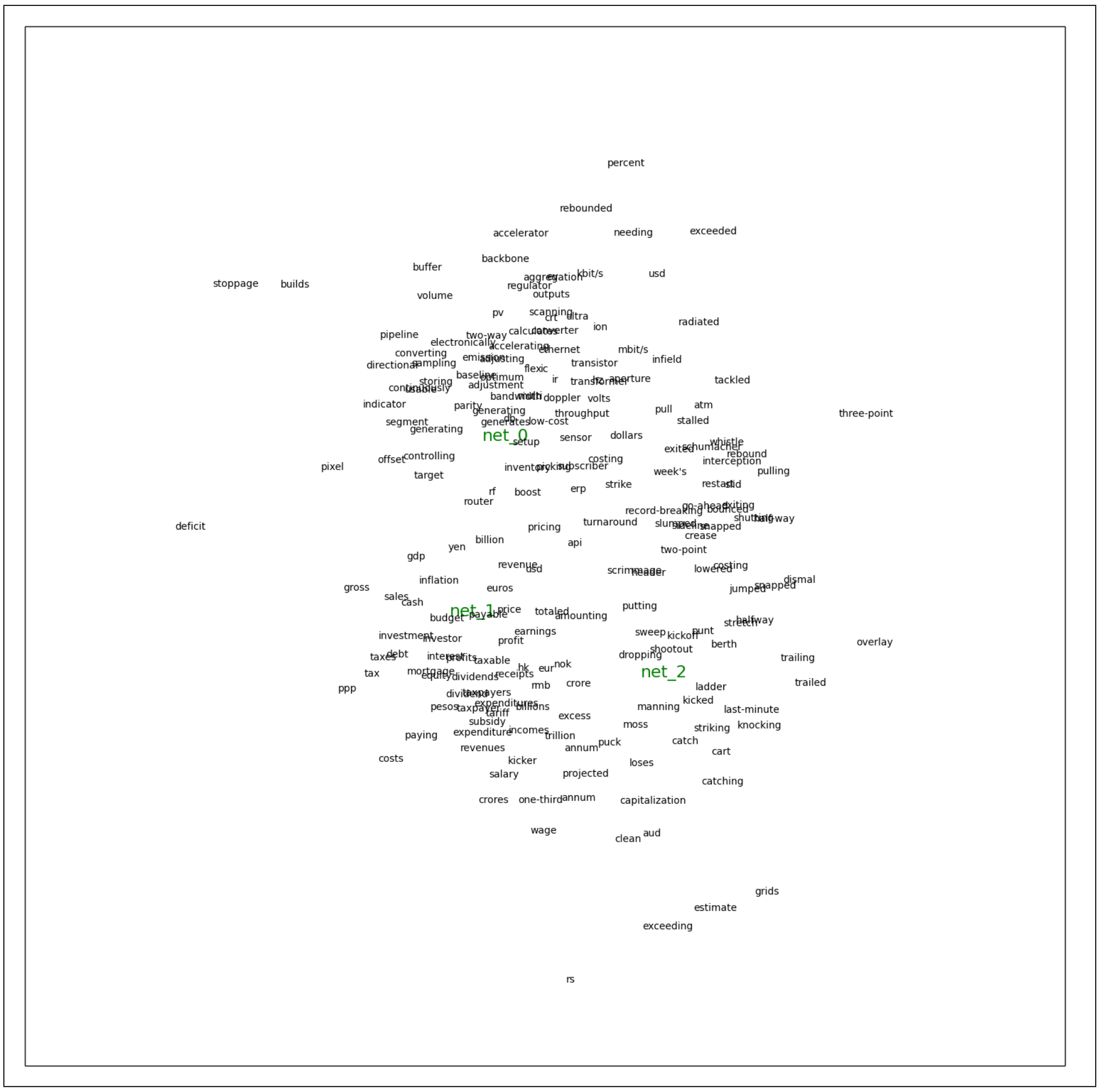

Figure 4.3: net

nearest words

$rock_0$:	echo	surf	memphis	strawberry	clearwater	cliff	sunset
$rock_1$:	r&b	hip	roll	indie	ska	indie	hop
$rock_2$:	formations	crust	melting	lava	boulders	granite	dust

examples

- zero nine is a finnish hard $rock_0$ band formed in kuusamo in

- west $rock_0$ is a small unincorporated community in pine county minnesota united states
- matt ellis b december is a folk $rock_1$ genre singer-songwriter who became an indie identity of critical acclaim in his hometown of sydney australia
- drawing shapes was the first ep by british $rock_1$ band morning runner that they released with parlophone it was first released may see in british music
- cabo de natural park is characterised by volcanic $rock_2$ formations
- in may rolling stone named against me best punk band on a list entitled the plus people places and things ruling the $rock_2$ roll universe

visualization

run

sense matrix

	run_0	run_1	run_2
run_0	1.000000	0.593599	0.698333
run_1	0.593599	1.000000	0.621636
run_2	0.698333	0.621636	1.000000

nearest words

run_0:	blair	taft	fraser	monroe	precinct	mayor's	governor's
run_1:	streak	rushing	tying	shutout	inning	wicket	kickoff
run_2:	running	tram	travel	express	trams	inbound	long-distance

examples

- as of the school year pleasant run_0 elementary had a total of students african-american hispanic and white
- in july dean announced that she intends to run_0 for mayor again in the november election
- the movie was a flop at the box office bringing in a total of in a limited run_1 at the theater

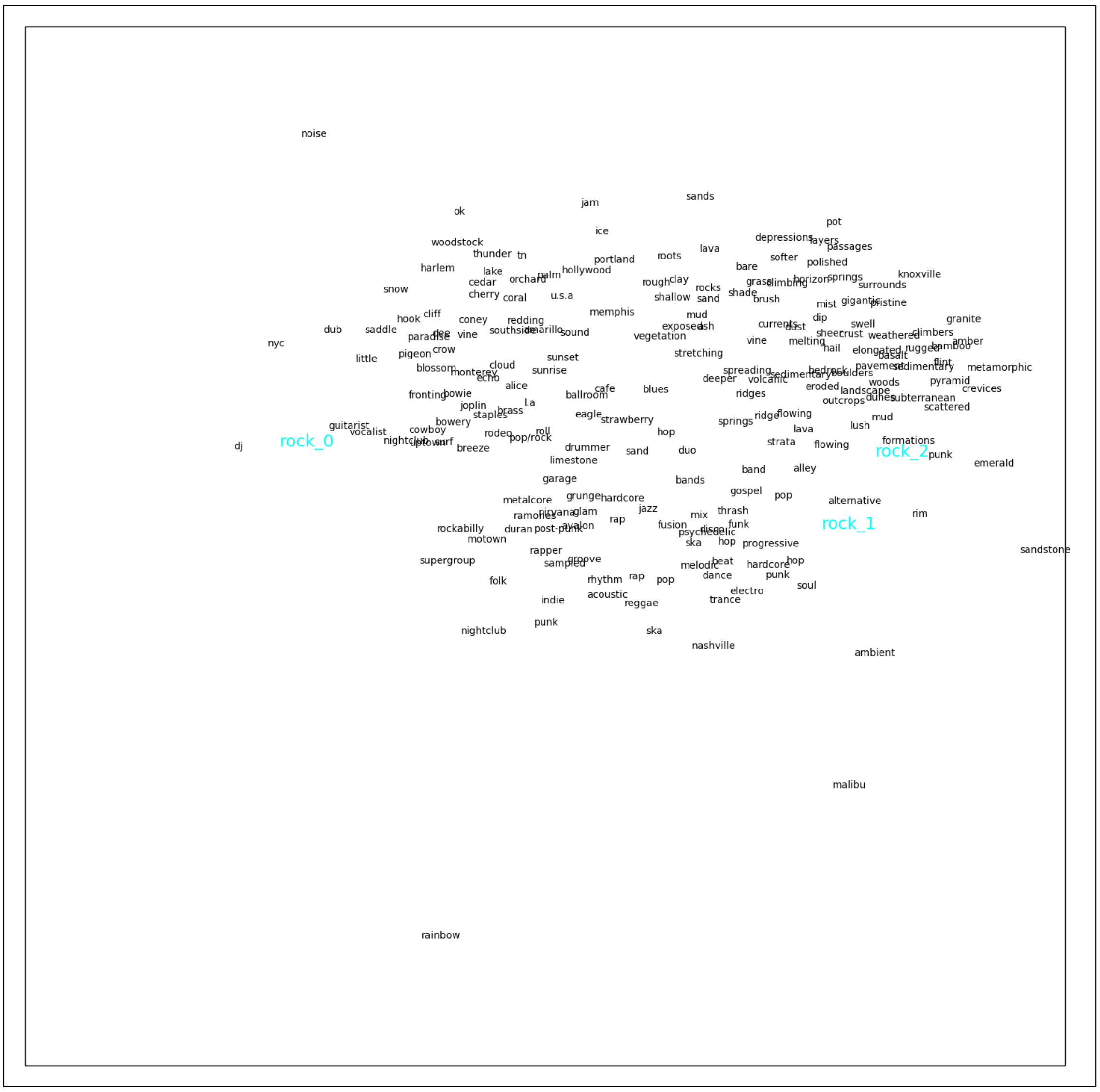

Figure 4.4: rock

- we just couldn't run_1 the ball coach tyrone willingham said
- for example the beanshell language runtime will run_2 a program contained in the file
- the terminal is run_2 by british rail freight company ews

visualization

plant

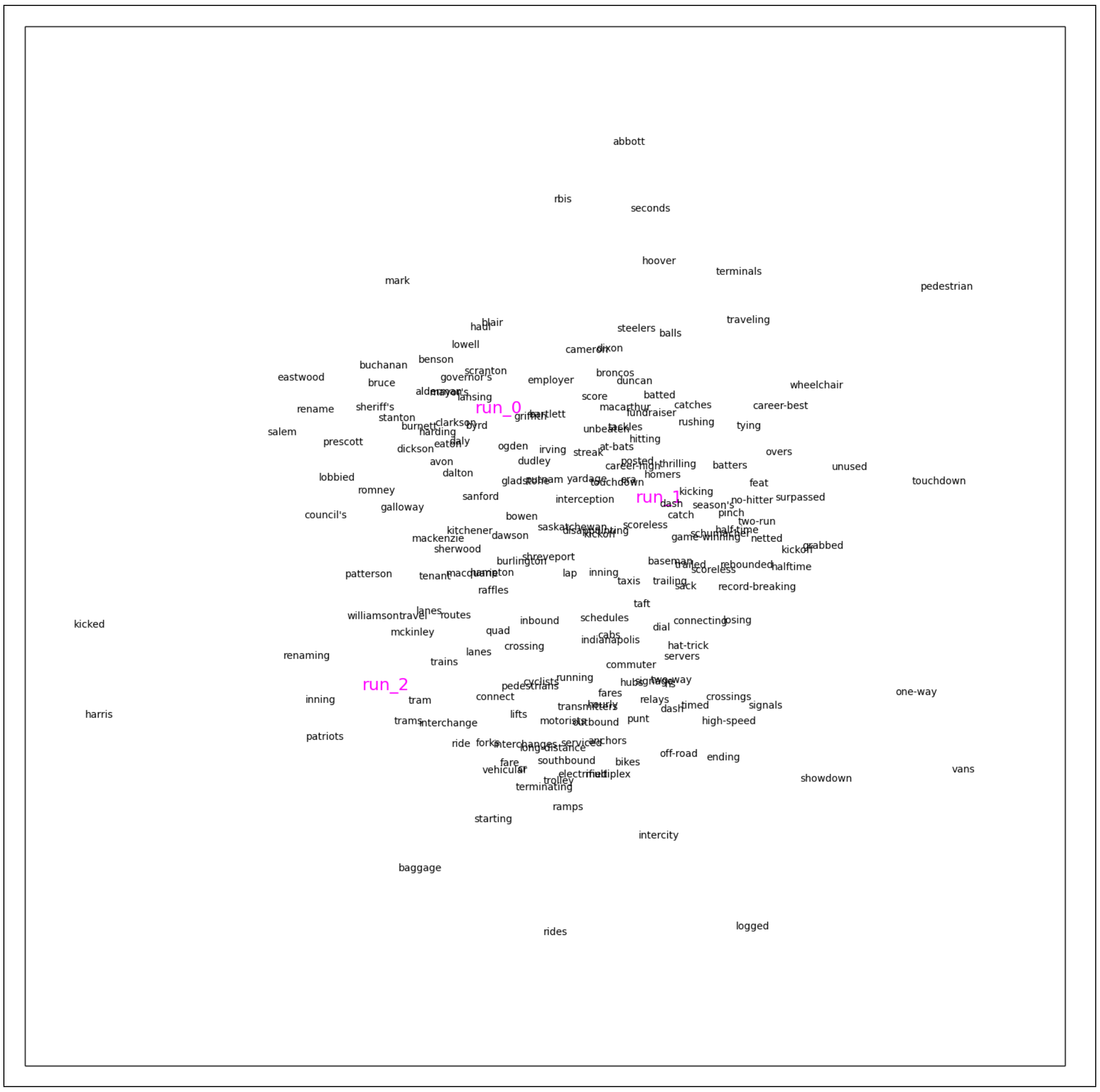

Figure 4.5: run

sense matrix

	$plant_0$	$plant_1$	$plant_2$
$plant_0$	1.000000	0.764150	0.631629
$plant_1$	0.764150	1.000000	0.539350
$plant_2$	0.631629	0.539350	1.000000

nearest words

$plant_0$:	plants	insect	seeds	seed	pollen	aquatic	organic
$plant_1$:	flowering	orchid	genus	bird	species	plants	butterfly
$plant_2$:	electricity	steel	refinery	refinery	manufacturing	gas	turbine

visualization

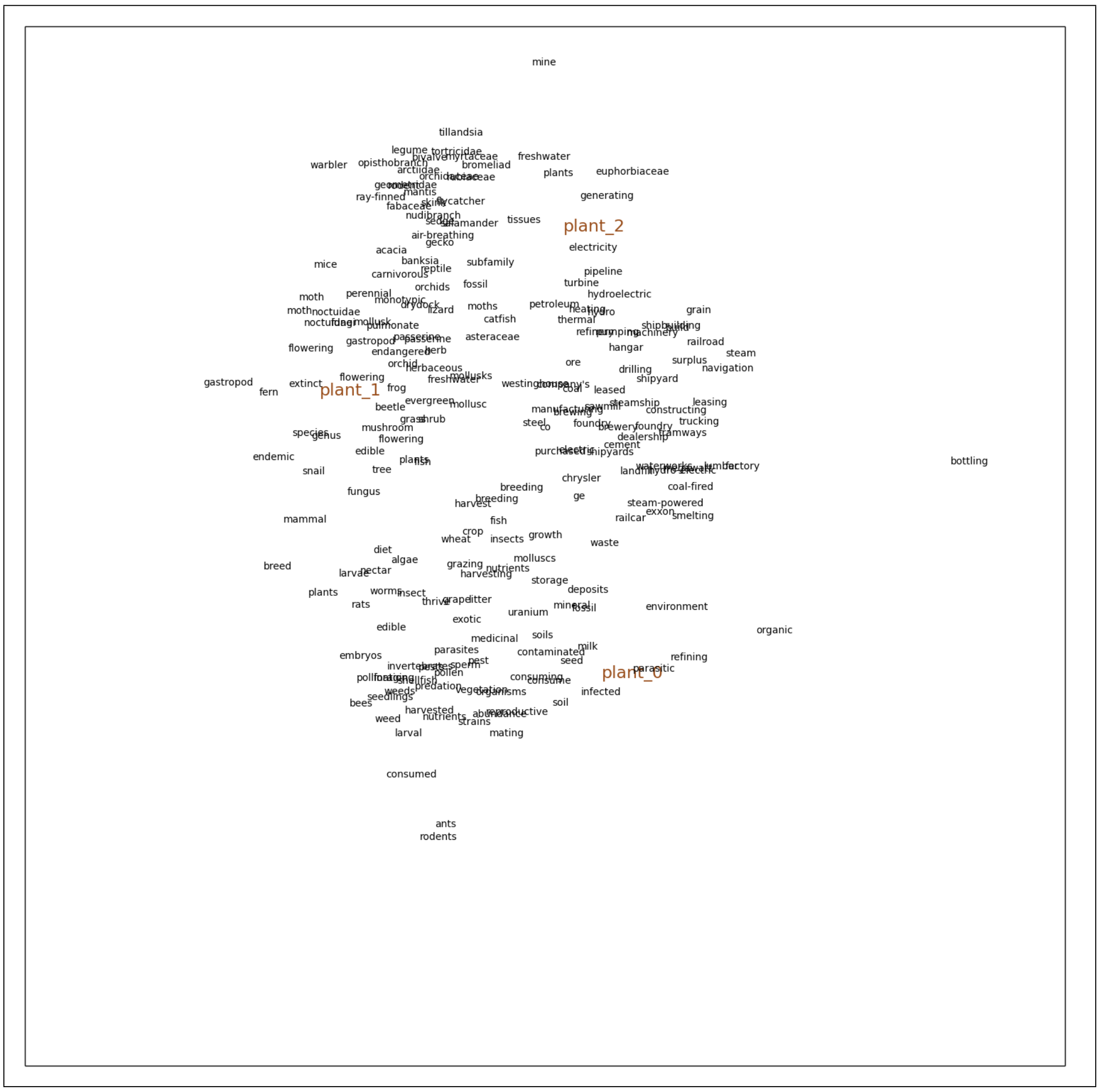

Figure 4.6: plant

Combining 5 group words

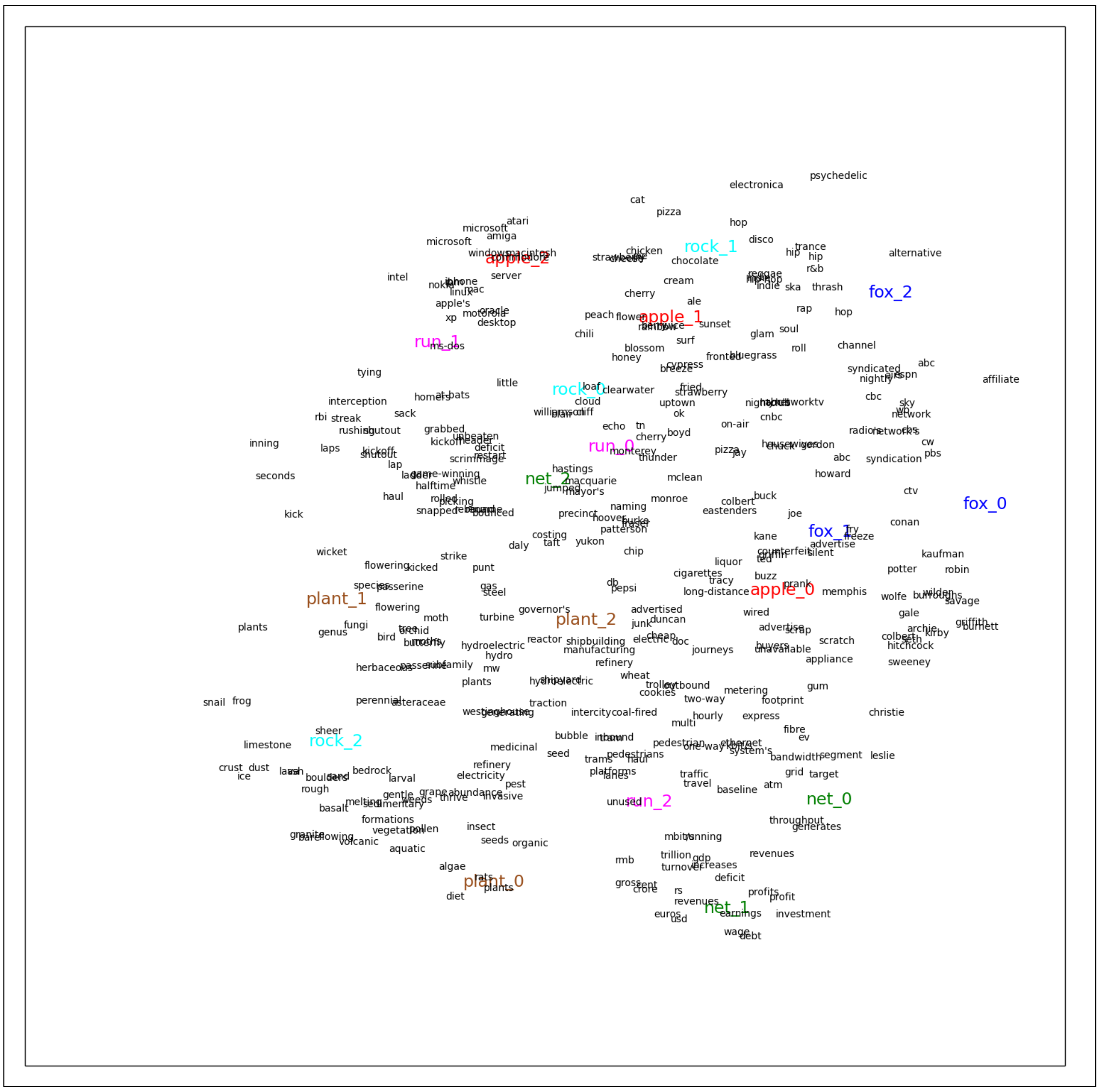

Figure 4.7: all words

Chapter 5

Conclusion

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

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

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

Appendix A

Appendix

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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