Word2Vec and Sentiment Analysis Project Report

Shihan Ran - 15307130424

***Abstract*—This project is aimed at using word2vec models for sentiment analysis, which can be separated as two subtasks: implementing word2vec model(Skip-gram in this task) to train my own word vectors, and use the average of all the word vectors in each sentence as its feature to train a classifier(e.g. softmax regression) with gradient descent method.**

During this project, alone with implementing the already well-framed code block, I've spent much time improving my code's efficiency and comparing different implementation methods. Talking about the sentiment analysis, to achieve higher accuracy, I've tried different combinations with Context size C, word vector's dimension dimVectors and REGULARIZATION. In terms of training and testing models, the Development Set has been divided into Training Set and Dev-Test Set. Finally, the best accuracy for dev set was achieved at 29.79% for parameters as C=9, dim=10, reg=10^{-7}.

I. Introduction

REPRESENTATION of words as continuous vectors has a long history. It was later shown that the word vectors can be used to significantly improve and simplify many natural language processing applications.

In this project, we focus on training our own word vector, and using it in the sentiment analysis of Stanford Sentiment Treebank(SST) dataset, to predict which sentiment categories a sentence should be assigned.

For the following parts of the report, we will begin with the introduction of word2vec model firstly, explaining the principles of both skip-gram[1] algorithms and negative sampling[2] algorithms. Then we discuss some improvements we made in our code for higher efficiency by presenting the design and implementation of word2vec model. Finally, we analyze the testing results and draw a conclusion.

II. Word2Vec Models

A. Skip-gram model

The training objective of the Skip-gram model is to find word representations that are useful for predicting the surrounding words in a sentence or a document. More formally, given a sequence of training words $w_1, w_2, w_3, \cdots, w_T$, the objective of the Skip-gram model is to maximize the average log probability

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\le j\le c, j\ne 0}\log p(w_{t+j} \mid w_t)$$

The basic Skip-gram formulation defines $p(w_{t+j} \mid w_t)$ using the softmax function:

$$p(w_o \mid w_I) = \frac{\exp({v'}_{w_o}^{T} v_{w_I})}{\sum_{w=1}^{W}\exp({v'}_{w}^{T} v_{w_I})}$$

We specified skip-gram by the following components from Table I.

TABLE I
THE SKIP-GRAM'S COMPONENTS

c	the size of the training context
t	index of the centered word
W	the number of words in the vocabulary
$v_w, {v'}_w$	input and output vector representations of w

B. Negative Sampling

The formulation of Skip-gram is impractical because the cost of computing $\log p(w_o \mid w_I)$ is proportional to W, which is often large(10^5-10^7 terms).

However, the Skip-gram model is only concerned with learning high-quality vector representations, so we are free to simplify the process as long as the vector representations retain their quality. We define Negative sampling (NEG) by the objective

$$\log(\sigma(u_o^T v_c)) + \sum_{K=1}^{K}\log(\sigma(-u_k^T v_c))$$

Assume that K negative samples (words) are drawn, and they are $1, \cdots, K$ respectively for simplicity of notation($o \notin 1, ..., K$). For a given word, o, we denote its output vector as u_o.

III. Stanford Sentiment Treebank dataset

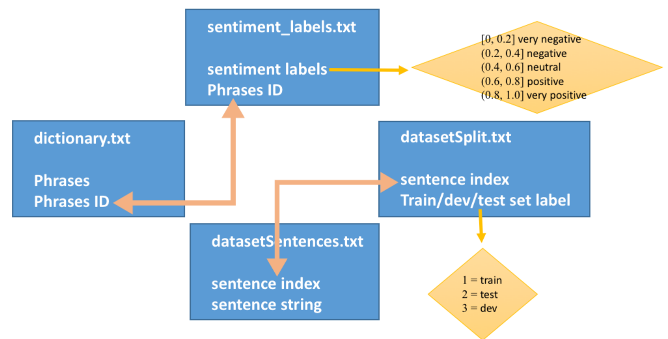

Fig. 1. The structure of dataset.

The structure of this dataset and relation between entities can be concluded as the Figure 1. Although we don't need to implemented the testing process ourselves, it may be helpful if we have a full understanding of the project's structure.

As always, read and understand code before using it!

IV. Implementation of word2vec

We implemented word2vec model using our calculation of cost and gradient. However I've found that it is crucial to optimize my code for higher efficiency. Hence, for higher efficiency, during this process, I've looked through many tutorials and tried several methods for each function.

A. *normalizeRows*

Method1 Follow the formulation of normalization, intuitively I write a function using "for" loop.

```
1: for i in range(len(x)) do
2:     x[i, :] = x[i, :]/np.sqrt(np.sum(x[i, :]*x[i, :]))
3: end for
4: return x
```

Algorithm 1: normalizeRows using "for" loop

Method2 After searching online and I found this can be achieved by a package!

```
1: return sklearn.preprocessing.normalize(x, norm='l2')
```

Algorithm 2: normalizeRows using package

AMAZING! After using method2, the average time of normalizeRows has decreased from 0.000484943389893s to 0.000205039978027s, which means it has saved us about **60%** time!

B. *softmaxCostAndGradient*

For the implementation of deriving expected gradient:

$$\frac{\partial}{\partial u_w} J_{softmax-CE} = \frac{\exp(u_w^T v_c) v_c}{\sum_{i=1}^{V} \exp(u_i^T v_c)} (w \neq o)$$

I've chosen two methods to achieve it. One is a naive implementation using "for" loop, another is the use of *numpy.outer*.

Method1 Follow the formulation of softmax's cost and gradient, I write a function using "for" loop.

```
1: grad = np.zeros([p.shape[0], v_c.shape[0]])
2: for i in range(p.shape[0]) do
3:     grad[i, :] = v_c * p[i]
4: end for
5: grad[target, :] -= v_c
```

Algorithm 3: softmaxCostAndGradient using "for" loop

Notice! Pay extra attention to the fact that the vectors are stored by row.

Method2 After doing some math calculating and searching online and I found this can be achieved by numpy itself! This operating is called outer product.

AMAZING! After using method2, the average time of softmaxCostAndGradient has decreased from 6.01028216279e-05s to 4.65352400599e-05s, which means it has saved us about **25%** time!

```
1: grad = np.outer(p, v_c)
2: grad[target, :] -= v_c
```

Algorithm 4: softmaxCostAndGradient using numpy

C. *negSamplingCostAndGradient*

For the implementation of deriving expected gradient:

$$\frac{\partial}{\partial v_c} J(\theta) = (\sigma(u_o^T v_c) - 1)\, u_o + \sum_{K=1}^{K} (1 - \sigma(-u_k^T v_c))\, u_k$$

$$\frac{\partial}{\partial u_k} J(\theta) = (\sigma(u_o^T v_c) - 1)\, v_c$$

$$\frac{\partial}{\partial u_o} J(\theta) = [(\sigma(u_o^T v_c) - 1) + 1]\, v_c$$

I've tried four methods to achieve it.

Method1 Follow the formulation of cost and gradient, I write a function using "for" loop.

```
1: indices = [dataset.sampleTokenIdx() for k in range(K)]
2: u_o = outputVectors[target, :]
3: v_c = predicted
4: sigma1 = sigmoid(np.dot(u_o, v_c))
5: cost = -np.log(sigma1)
6: gradPred = u_o * (sigma1 - 1)
7: grad = np.zeros(outputVectors.shape)
8:
9: for i in range(K) do
10:     u_k = outputVectors[indices[i], :]
11:     sigma2 = sigmoid(-np.dot(u_k, v_c))
12:     cost = cost - np.log(sigma2)
13:     gradPred = gradPred + u_k * (1 - sigma2)
14:     grad[indices[i]] += v_c * (1 - sigma2)
15:     grad[i, :] = v_c * p[i]
16: end for
17: grad[target, :] += v_c * (sigma1 - 1)
```

Algorithm 5: A slow and inefficient method using "for" loop

Method2 I tried to use some matrix operating to replace "for" loop. Hence, I replace $\sum$ by *np.sum*, replace * by *np.dot* and broadcasting.

```
1: u_k = outputVectors[indices, :]
2: sigma1 = sigmoid(np.dot(u_o, v_c))
3: sigma2 = sigmoid(-np.dot(u_k, v_c))
4: cost=-np.log(sigma1) - np.sum(np.log(sigma2))
5: gradPred = u_o * (sigma1 - 1) + np.dot((1 - sigma2).T,
   u_k)
6: grad = np.zeros(outputVectors.shape)
7:
8: for i in range(K) do
9:     grad[indices[i], :] += v_c * (1 - sigma2)[i]
10: end for
```

Algorithm 6: Use matrix operating to replace "for" loop

Method3 I tried to use *np.outer* to replace a loop!

```
1: temp = np.outer(1 - sigma2, v_c)
2: for i in range(K) do
3:     grad[indices[i], :] += temp[i]
4: end for
```

Algorithm 7: Use numpy package

Method4 I tried to only use matrix operations to avoid "for" loop but failed :(

The reason why it can't work, I think it is because there are replicate values in indices, hence we can't use index to read them out and update them all together at the same time.

```
1: grad[indices, :] += np.tile(v_c, [len(sigma2), 1]) * (1 - sigma2)[:, None]
```

Algorithm 8: Use only matrix operations

AMAZING! After using these methods, the average time of negSamplingCostAndGradient has decreased much. Method2 has saved us about **10%** time, and method3 has saved us about **30%** time!

Actually, if you want to make this project more efficient, instead of only using packages or changing "for" loop into matrix operations, noticing that many operations and functions are paralleled, you can also adjust the whole project structure **from single thread into multi thread**, which can be achieved by python's package **multiprocessing**. And this will **definitely tremendously** save your time!

V. Result Analysis

A. *Different C for fixed dimVectors*

TABLE II
PERFORMANCE OF DIFFERENT CONTEXT SIZE FOR DIMVECTORS=5

Best regularization	C	Train_acc	Dev_acc	Test_acc
10^{-7}	1	27.808989	26.793824	**25.791855**
10^{-6}	3	27.914326	27.611262	25.701357
10^{-6}	5	**28.242041**	**28.065395**	25.339367
10^{-4}	7	28.148408	27.429609	24.660633
10^{-7}	9	27.492978	27.611262	25.339367

TABLE III
PERFORMANCE OF DIFFERENT CONTEXT SIZE FOR DIMVECTORS=10

Best regularization	C	Train_acc	Dev_acc	Test_acc
10^{-6}	1	29.389045	29.336966	27.104072
10^{-5}	3	28.944288	28.701181	27.285068
10^{-6}	5	29.166667	28.792007	26.923077
10^{-5}	7	29.459270	29.064487	**27.873303**
10^{-7}	9	**29.588015**	**29.791099**	27.149321

TABLE IV
PERFORMANCE OF DIFFERENT CONTEXT SIZE FOR DIMVECTORS=20

Best regularization	C	Train_acc	Dev_acc	Test_acc
10^{-6}	1	29.330524	28.247048	26.877828
10^{-5}	3	**29.494382**	**27.974569**	26.515837
10^{-5}	5	28.885768	27.792916	**27.194570**
10^{-6}	7	29.225187	27.974569	26.968326
10^{-6}	9	29.435861	28.519528	27.149321

TABLE V
PERFORMANCE OF DIFFERENT CONTEXT SIZE FOR DIMVECTORS=30

Best regularization	C	Train_acc	Dev_acc	Test_acc
10^{-6}	1	28.593165	28.156222	26.339367
10^{-8}	3	**29.225187**	28.247048	27.058824
10^{-6}	5	29.166667	28.882834	**27.375566**
10^{-8}	7	29.096442	**29.155313**	27.285068
10^{-6}	9	29.073034	27.520436	26.832579

By fixing dimension and changing the value of context size, results can be concluded within the following Figure 2.

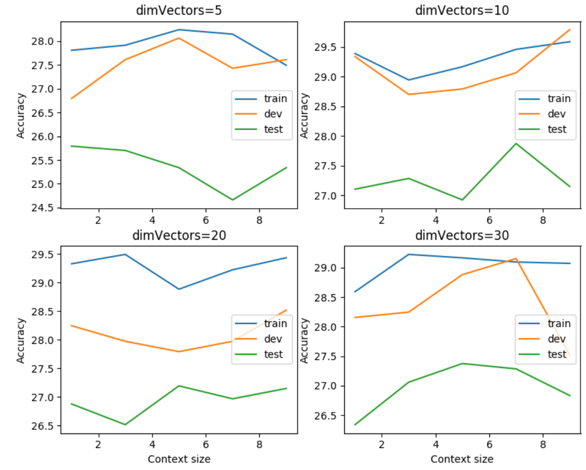

Fig. 2. Different C with fixed dimVectors

We can naturally draw the following conclusions from the above table and figure:

- Generally speaking, with **C** increasing, the performances of this model increases.
- However, the fact didn't exist: the higher C is, the better performance.

To explain this a little bit wired phenomena, I assume the reasons are as followings:

- When C becomes higher enough, there may exists two problems: convergence and overfitting.
- Sometimes, the accuracy just doesn't increase and becomes plain, because it reached its convergence.
- Other times, the accuracy suddenly decreases, this may due to the overfitting problem. As we can see, it achieved high accuracy on train set but relatively low accuracy on dev and test set.

B. *Different dimVectors for fixed C*

By fixing context size and changing dimension, results can be concluded within the following Figure 3.

We can draw conclusions from this figure:

- Generally speaking, the increase of **dimVectors** doesn't necessarily help increase the accuracy of dev set.
- And again, we can see through the plot that the increase of C doesn't necessarily means increase of accuracy.

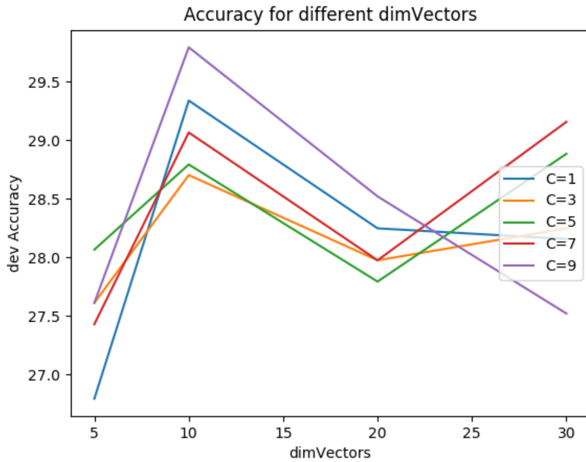

Fig. 3. Different dimVectors with fixed C

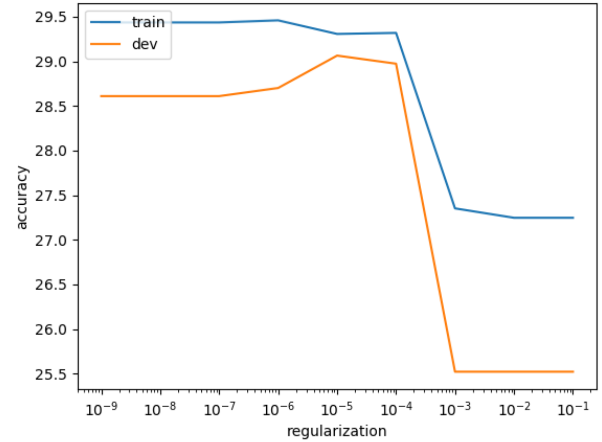

Fig. 4. Different regularization

To explain this phenomena, I assume the reasons are as the followings:

- Intuitively, we take it that larger dimension can help contain more information, because the learned vectors explicitly encode many linguistic regularities and patterns, which will lead to higher accuracy.
- However this project is based on a relatively small dataset, hence the larger dimension may also contain noise information which will decrease accuracy and bring problems like overfitting.

It seems like we can set **dimVectors** as either 10 or 30, and set C as 9, because these combination can attain high accuracy. Nevertheless, apart from considering accuracy, we should also take efficiency into consideration.

Don't forget that larger dimension and larger C are **at the expense of** larger training time!

TABLE VI
TRAINING TIME FOR FIXED **C=1**

	dim=5	dim=10	dim=20	dim=30
Training Time/h	**1.08**	2.73	5.04	**7.30**

TABLE VII
TRAINING TIME FOR FIXED **DIM=10**

	C=1	C=3	C=5	C=7	C=9
Training Time/h	2.73	2.99	3.17	3.21	3.59

C. *Different REGULARIZATION*

By changing regularization term, results can be concluded within the following Figure 4.

We can naturally draw the following conclusions from the above figure:

- Both large and small regularization value can lead to overfitting problem, which may occur as the huge difference between train accuracy and dev accuracy.
- Large regularization can also brings about problems like loss of accuracy.

VI. CONCLUSION

Finally, considering about both efficiency and accuracy, I choose combination of **C=9, dimVectors=10**.

The visualized word vector is like Figure 5.

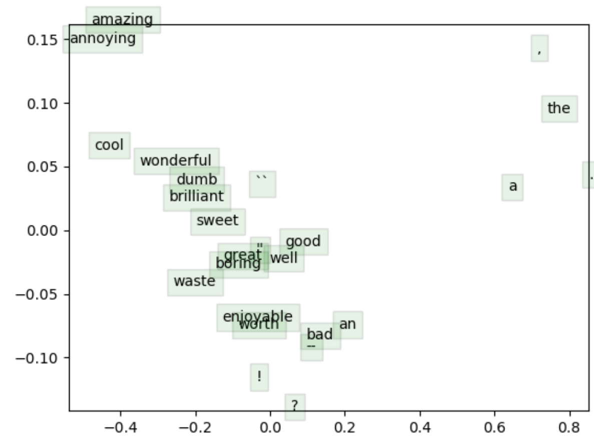

Fig. 5. Visualization of word vector

We can see it separates different kinds of words well, however it sometimes overlaps contradictory words like **great and boring**. This phenomena may due to the small size of training set and the similar usage for these two words. For example, I might say: "I felt this project is great!" and someone might say "I felt this project is boring!". Hence, with the lack of more corpus data, the word2vec model can't tell these words apart.

REFERENCES

[1] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. ICLR Workshop, 2013.

[2] Distributed Representations of Words and Phrases and their Compositionality