Word2Vec and Sentiment Analysis

Abstract--In this report, word2vec model was constructed for sentiment analysis. Our task is separated as two subtasks: firstly, we need to use word2vec model to train vectors and then use the average of word vectors as features to train a classifier of sentiment. To improve computation speed, negative sampling was tried to train word2vec model. Also, parameters such as context size, dimension of vectors were tested to achieve the best performance. Finally, taking training time into consideration, the best accuracy was achieved at 27.87%, for parameters as C=7, dim=10,regulation=10^{-4}.

I. Introduction

Word representation is indispensable in many NLP tasks. Discrete word representation has some problems, such as dimension explosion and missing nuance. Therefore, dense vectors are needed in many NLP problems. Dense word representation Word2Vec is a model directly learning low-dimensional word vectors. Based on neural network, there are two algorithms (Skip-grams and CBOW) and several training methods for this model.

In this report, we use Stanford Sentiment Treebank(SST) dataset to train word vectors. Skip-grams algorithm was complemented and negative sampling was focus. After that, features were extracted from the vectors to get a classifier to predict categories of sentiment.

For the following parts, we begin with introduction to the main models. Then implementation of models and measures to improve performance and efficiency will be discussed. Finally, we talk about results and draw a conclusion.

II. Models

A. Word2Vec Model

Word2Vec model uses neural network to train word vectors. While the CBOW algorithm predicts target word based on context words, the Skip-grams algorithm learns the representation of context words from the target word. Here, we mainly focus on the Skip-grams algorithm, which can be described as follows:

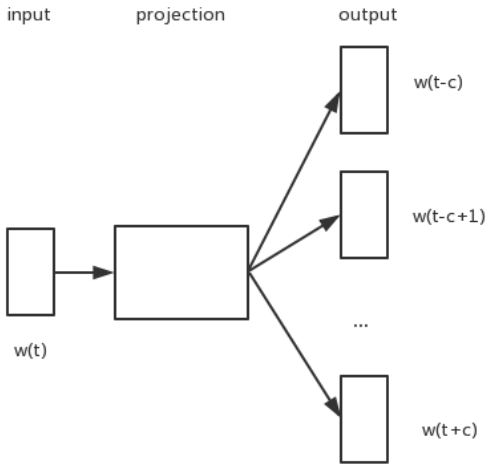


Fig 1. Skip-gram model

Formally, given a training word sequence $w_1, w_2, w_3 \dots w_T$, we need to maximize the log probability:

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

For computation of probability $p(w_o|w_I)$, this algorithms refers to cosine similarity and softmax function.

$$p(w_o|w_I) = \frac{\exp({u_{w_o}}^T v_{w_I})}{\sum_{w=1}^{W}\exp({u_w}^T v_{w_I})}$$

Obviously, size of the dictionary W greatly influence computation speed. It is a

time-consuming and impossible mission to compute similarity of all words and normalize them. To solve this problem, two optimization training methods have been provided: hierarchical softmax and negative sampling.

B. Negative Sampling

When training the neural network, every time a training sample is received, all neural unit weights are adjusted to make the neural network prediction more accurate. In other words, each training sample will adjust the parameters of all neural networks. This undoubtedly reduces training efficiency. Instead, negative sampling enables a training sample update only a small part of the weight parameters at a time, so as to reduce the amount of calculation in the gradient descent process.

Here, the probability of a word chosen as a negative sample was related to its frequency. More frequently a word appears, more likely it will be selected as a negative sample. Generally, empirical formula is used here:

$$\mathrm{p}(w_j) = \frac{f(w_j)^{3/4}}{\sum_{j=0}^{n} f(w_j)^{3/4}}$$

where $\mathrm{p}(w_j)$ is the probability w_j is selected as a negative sample, and $f(w_j)$ represents the frequency of w_j.

In this case, our objective is converted into:

$$J(\theta)' = log\sigma({u_O}^T v_I) + \sum_{k=1}^{K} log\sigma(-{u_k}^T v_I)$$

Where K means the size of negative sampling. It is noticeable that we regard the normal context combination as 1, and the abnormal context combination as 0. Then the problem is transformed into a binary problem. So softmax function of output layer can be replaced by sigmoid function σ.

III. Experiments

A. Implementation of Word2Vec

We implemented word2vec model using our calculation of cost and gradient. In order to improve code efficiency, some packages and functions are called here.

1. normalizeRows

In this part, we need to implement a function that normalizes each row of a matrix to have unit length. Both for loop and normalization function of sklearn package have been tried and finally **sklearn.preprocessing.normalize** function was used , which can save over 50% time.

2. softmaxCostAndGradient

Here, we need to implement the cost and gradients for one predicted word vector and one target word vector as a building block for word2vec models. The task is depicted as follows:

Tab 1. softmaxCostAndGradient description

Input	Output
Predicted word vector	cross entropy cost for the softmax word prediction
index of the target word	the gradient with respect to the predicted word vector
"output" vectors for all tokens	the gradient with respect to all the other word vectors

From the given formulas, word prediction is:

$$\widehat{y_0}=\mathrm{p(o|c)}=\frac{\exp(u_o^T v_c)}{\sum_{w=1}^{W}\exp(u_w^T v_c)}$$

And Softmax-CE loss function:

$$J_{softmax-CE} = CE(y, \hat{y})$$

To optimize our objective, we generate the derivative.

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

To get gradient, for loop is tried here:

```
for i in range(p.shape[0]):
    grad[i,:]=v_c*p[i]
grad[target,:]-=v_c
```

Also, this can be solved by function provided

by numpy: np.outer, with a row of code and fast computation speed.

3. negSamplingCostAndGradient

In this part, negative sampling technique is used and cost and gradients need to be computed. Input and output are as follows:

Tab 2. negSamplingCostAndGradient description

Input	Output
Predicted word vector	cross entropy cost for the softmax word prediction
index of the target word	the gradient with respect to the predicted word vector
"output" vectors for all tokens	the gradient with respect to all the other word vectors
dataset	

To get gradients, following formulas are provided:

$$J(\theta)' = log\sigma(u_O{}^T v_c) + \sum_{k=1}^{K} log\sigma(-u_k{}^T v_c)$$

$$\frac{\partial J(\theta)'}{\partial v_c} = (\sigma(u_O{}^T v_c) - 1)v_c + \sum_{k=1}^{K}(1 - \sigma(-u_k{}^T v_c))u_k$$

$$\frac{\partial J(\theta)'}{\partial u_k} = (\sigma(u_O{}^T v_c) - 1)v_c$$

$$\frac{\partial J(\theta)'}{\partial u_o} = [(\sigma(u_O{}^T v_c) - 1) + 1]v_c$$

In operation, numpy.outer is used again to calculate the gradients.

B. Parameter Adjustment

There are some parameters which can influence the training performance. Here, we take context size C, vector dimension dim and regulations into account. To find the most proper parameter values, we change one parameter and fix the rest and finally get results of different combinations.

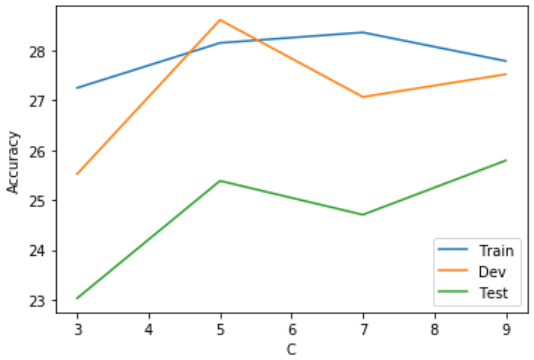
1. Different context size C

To find proper context size C, we fixed dim first. Here, we fix dim at 5,10,20 separately and results are below.

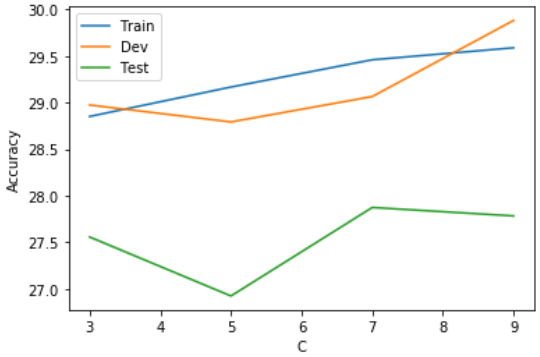
Tab 3. Result for different C and dimVec

dimVec	C	Best REGULATION	Train_acc	Dev_acc	Test_acc
5	3	10^{-6}	27.247191	25.522252	23.031674
	5	10^{-6}	28.148408	28.610354	25.384615
	7	10^{-4}	28.359082	27.066303	24.705882
	9	10^{-6}	27.785581	27.520436	**25.791855**
10	3	10^{-7}	28.850655	28.973660	27.556561
	5	10^{-8}	29.166667	28.792007	26.923077
	7	10^{-4}	29.459270	29.064487	**27.873303**
	9	10^{-7}	29.588015	29.881926	27.782805
20	3	10^{-5}	29.494382	27.974569	26.515837
	5	10^{-6}	29.190075	29.064487	**27.375566**
	7	10^{-6}	29.225187	27.974569	26.968326
	9	10^{-6}	29.435861	28.519528	27.149321

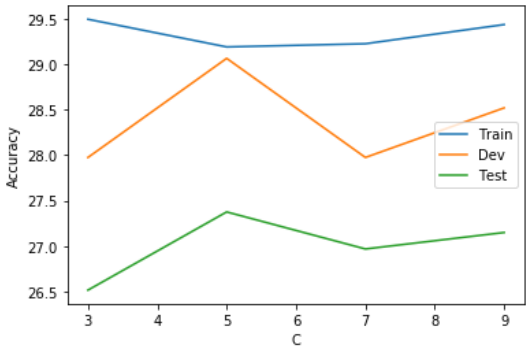
And we visualize these results by drawing line charts.



dim=5



dim=10



dim=20

Fig 2. different C with fixed dimVectors

Generally, performance of this model increases with context size C increases. But there are also some exceptions. This may be the result of overfitting, as we can see accuracy for train set and dev set achieved high but it is relatively low for test set.

2. Different vector dimensions

Control the context size C and we compare performance of models with different vector dimensions. From line chart below, we can get some conclusions:

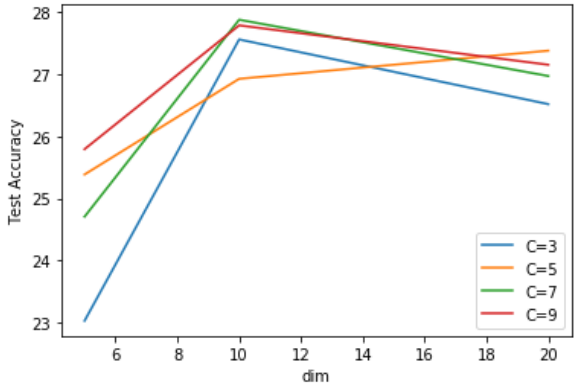


Fig 3. different dimVectors with fixed C

Firstly, simply increasing the dimVectors doesn't necessarily help improve the test accuracy. Also, we can see that larger context size C is not guarantee for better performance.

Usually, vectors of higher dimension can contain more information so that they are supposed to achieve higher accuracy. However, in this dataset, there may be noise influencing the performance so that we got results above.

3. Different REGULATION

When it comes to regulation, if regulation is too little, the model is easy to overfit; on the contrary, if large regularization is used, it can also bring about problems like loss of accuracy. From results above, we can figure that different combinations of C and dimVector have different proper regulations.

IV. Conclusion

Besides accuracy, we should also pay attention to time cost of our models. Larger context size C and larger dimension undoubtedly cost more training time. Take accuracy and training time into consideration, I choose combination of C=7,dimension=10. And the final visualized word vector is as follows:

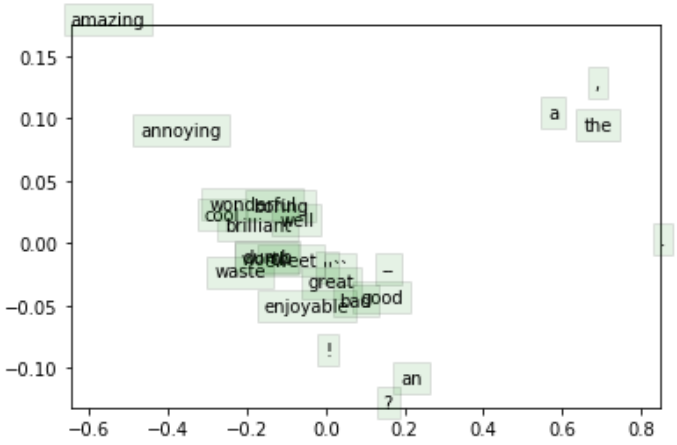


Fig. 4. Visualization of word vector

It works well on most words, as it separates different words. We can see words with similar meanings, for example, 'great', 'good' and 'enjoyable' gather together. However, it also has some problems, for it overlaps words like 'good' and 'bad'. That may be the result of insufficient corpus or noise. In the future, larger corpus data is needed to improve the word2vec model.