WORD EMBEDDINGS

A PREPRINT

Alperen Çakın
cakin.alperen@gmail.com

May 17, 2019

1 Introduction

Word embeddings are a newly discovered way of representing a word in a low-dimensional space. They provide a vectorial representation of words that bears any semantics or syntax. This paper reports algorithmic steps, abstractions and the results of the assignment that I have coded which consists of two tasks: Evaluating pre-trained word2vec vectors and document classification using trained doc2vec embeddings.

2 Part I: Evaluating Pre-trained Word2Vec Vectors

2.1 Loading Vectors

Pre-trained word vectors obtained from Google in a binary file format, is loaded using *.load()* method of the *KeyedVectors* class implemented in the *gensim* module. A *limit* parameter is passed to the *.load()* method causing it to ignore embeddings of less frequent words. This operation is performed because trying to store both normalized (explained in the next section) and raw vectors to the memory yields a *MemoryError* on a device with insufficient physical memory for this operation. (See the related link in the section 4)

2.2 L2 Normalization

L2 normalization can be defined as normalizing vectors by their lengths. This operation is shown in the equation below, for a vector v:

$$v_{normalized} = \frac{v}{||v||} \tag{1}$$

This operation is performed for each vector in the given dataset by using *.init_sims()* method of the *Word2VecKeyedVectors* class. The purpose is to calculate cosine similarities faster. This is explained in the section 2.4 throughly.

2.3 Calculating the Target Vector

Before calculating cosine similarities, a target vector needs to be calculated in order to execute the given analogy tests. For given words a, b, c and word vectors obtained from the given dataset v_a, v_b, v_c the target vector is calculated using the following formula:

$$v_{target} = v_b + v_c - v_a \tag{2}$$

This vector is also L2 normalized, allowing the software to calculate cosine similarities between this vector and all vectors in the dataset just by using dot product. This is illustrated in the equation below:

$$v_{target_{normalized}} = \frac{v_{target}}{||v_{target}||} \tag{3}$$

After calculating the target vector, the most similar vector in the dataset to this vector should give the expected word analogy. This can be demonstrated using the following example:

$$v_{king} - v_{man} + v_{woman} \xrightarrow{\text{most similar vector}} v_{queen} \tag{4}$$

2.4 Calculating Cosine Similarities

Cosine similarity for two given vectors v_1 and v_2 can be calculated using the following formula:

$$similarity_{v_1,v_2} = \frac{v_1 \cdot v_2}{||v_1||||v_2||} \tag{5}$$

The idea is, if we calculate L2 normalizations of all vectors in the dataset, calculating cosine similarity is just a matter of calculating dot product of any given two vectors. This is illustrated in the below figure:

$$\begin{aligned} similarity_{v_1,v_2} &= \frac{v_1 \cdot v_2}{||v_1||||v_2||} && (6) \\ &= \frac{v_1}{||v_1||} \cdot \frac{v_2}{||v_2||} && (7) \\ &= v_{1_{normalized}} \cdot v_{2_{normalized}} && (8) \end{aligned}$$

2.4.1 Getting a List of All Similarities

Word2VecKeyedVectors class stores loaded word vectors in the form of a matrix (or a list of vectors): *syn0* attribute of the class holds the raw vectors while *syn0norm* attribute holds L2 normalized vectors after calling the *.init_sims()* method.

$$syn0norm = \begin{bmatrix} v_{11} & v_{12} & v_{13} & v_{14} & \cdots & v_{1n} \\ v_{21} & v_{22} & v_{23} & v_{24} & \cdots & v_{2n} \\ v_{31} & v_{32} & v_{33} & v_{34} & \cdots & v_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ v_{m1} & v_{m2} & v_{m3} & v_{m4} & \cdots & v_{mn} \end{bmatrix}, v_{target} = \begin{bmatrix} v_{11} \\ v_{21} \\ v_{31} \\ \cdots \\ v_{m1} \end{bmatrix} \tag{9}$$

Considering this information, cosine similarities between the target vector and the all the other word vectors in dataset can be calculated in the form of a list by performing a dot product operation between the *syn0norm* and target vector.

$$syn0norm \cdot v_{target} = \begin{bmatrix} v_{11} & v_{12} & v_{13} & v_{14} & \cdots & v_{1n} \\ v_{21} & v_{22} & v_{23} & v_{24} & \cdots & v_{2n} \\ v_{31} & v_{32} & v_{33} & v_{34} & \cdots & v_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ v_{m1} & v_{m2} & v_{m3} & v_{m4} & \cdots & v_{mn} \end{bmatrix} \cdot \begin{bmatrix} v_{11} \\ v_{21} \\ v_{31} \\ \cdots \\ v_{m1} \end{bmatrix} = \begin{bmatrix} v_{similarity_{11}} \\ v_{similarity_{21}} \\ v_{similarity_{31}} \\ \cdots \\ v_{similarity_{m1}} \end{bmatrix} \tag{10}$$

This algorithm is inspired by inspecting the *.most_similar()* method of the *Word2VecKeyedVectors* class' source code. There are some differences between this algorithm and this class' way of calculating cosine similarities. However, they are similar as they use the same principle. (Getting cosine similarities using pre-calculated L2 normalized vectors).

2.5 Picking the Most Similar Word Vector

After creating a list of cosine similarities, a problem of picking the most similar vector arises. In *gensim*'s code, this problem is solved by sorting the whole list. However, sorting operation is expensive.

As an alternative to sorting, this problem can be converted into picking the k largest elements of this list. When word indices of the $k = 4$ largest elements of the similarity list are picked, it becomes possible to choose the most similar word vector. Word indices of the selected similarities are required, as those word indices will be used in *Word2VecKeyedVectors*'s *.index2word* mapping to retrieve words, both for filtering and output purposes.

k must be at least 4 because 3 input words are given as a parameter to each analogy test. Therefore we need at least 4 of the largest elements, as at most 3 of the input vectors may appear in the k largest list. If a value smaller than 4 were chosen, a scenerio explained in the below figure might occur for input words a, b, c:

$$largest\_three = [a_{index}, b_{index}, c_{index}] \tag{11}$$

In this scenerio, the method would fail to find the index of the most similar word vector as indices of the words a, b, c are filtered from the k largest elements. Thus, the method would end up with an empty list.

Picking the k largest elements problem is solved by using *numpy*'s *.argpartition()* method which works in linear time, clearly more efficient than the *numpy*'s *.argsort()* method used in *gensim*'s code.

After picking the 4 largest elements, the most similar word is chosen according to the pseudocode below:

Algorithm 1: Algorithm to pick the most similar word

```
1  $position\_max \leftarrow 0$
2  $value\_max \leftarrow -\infty$
3  for $i$ in $0, 1, 2, 3$ do
4      $current\_word\_index \leftarrow$ four_largest[i]
5      if $index2word[current\_word\_index]$ not in $input\_words$ then
6          if $sim[current\_word\_index] > value\_max$ then
7              $position\_max \leftarrow$ i
8              $value\_max \leftarrow$ sim[current_word_index]
9          end
10     end
11 end
12 $most\_similar\_word \leftarrow$ index2word[four_largest[position_max]]
```

2.6 Hardware Optimizations

Along with the software optimizations mentioned in the sections 2.4 and 2.5, some optimizations to use the hardware more efficiently are possible.

In this implementation, threads are used as a method of optimization. Using threads allows the software to use more than one CPU core at a time thus lowering the execution time. Built-in module *threading* is used in order to employ threads. Analogy tests are divided between threads using the *numpy*'s *.array_split()* method.

Another way to decrease the execution time would be using a GPU for matrix multiplication purposes while calculating cosine similarity. However, this method is not implemented in this assignment.

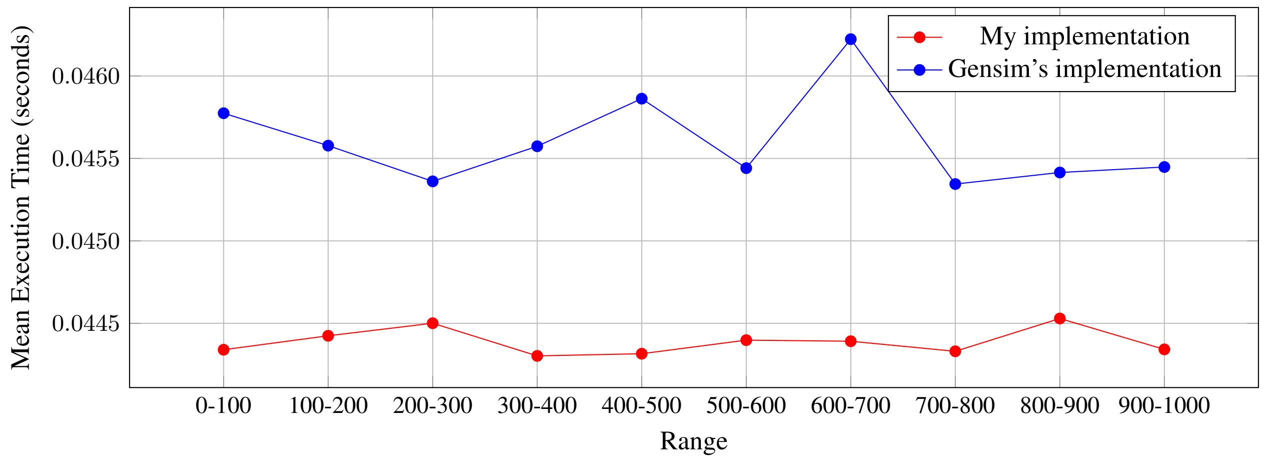
2.7 Calculating the Accuracy

After executing the given analogy tests, accuracy of the pre-trained model is calculated according to the below formula:

$$A(W) = \frac{\text{\# of correctly answered questions}}{\text{\# of questions attempted}} \tag{12}$$

2.8 Results

2.8.1 Running Time of the Most Similar Method



The line $\boxed{\text{Abuja Nigeria Accra Ghana}}$ from the analogy tests has been executed using both my implementation and the default *.most_similar()* method of the *Word2VecKeyedVectors* class for 1000 times. Mean of the execution times of ranges are shown in the above plot. As can be observed from the plot above, my implementation performed slightly better than the default *.most_similar()* implementation in *gensim*'s code. This test is conducted using a vector limit of 500000 and no threads. Total mean of the execution times without ranges are shown in the figure below:

$$Mean_{mine} = 0.04438748216629 \tag{13}$$
$$Mean_{gensim} = 0.045601971149445 \tag{14}$$

2.8.2 Accuracy and Total Running Time

All of the analogy tests are executed using a vector limit of 2000000 (because of the physical memory constraints of my device) and 4 threads. The following results are obtained without any exceptions indicating a test word is not in the loaded vectors:

Applied Method	Execution Time	Number of Correct Words	Accuracy
Gensim	38 minutes 27,206 seconds	9369	0.7153000458085204
My Implementation	38 minutes 15,688 seconds	9178	0.7007176668193618

Table 1: Accuracy and execution time of both methods

As you can see in the table above, my method performed slightly better in execution times. However, using my method, accuracy value is lower than the *gensim*'s default implementation. This is not caused by a bug in the software, this phenomenon occurs because *gensim* calculates the target vector in an unusual way.

The below figure represents the algorithm applied by the *gensim* to calculate the target vector:

$$v_{target} = v_{b_{normalized}} + v_{c_{normalized}} - v_{a_{normalized}} \tag{15}$$

$$v_{target_{mean}} = \frac{v_{target}}{3} \tag{16}$$

$$v_{target_{normalized}} = \frac{v_{target_{mean}}}{||v_{target_{mean}}||} \tag{17}$$

$$\tag{18}$$

Calculation of the target vector for my implementation, which is different than the algorithm above, is already explained in the section 2.3, so it is unnecessary to explain it again in this section. However, by looking at the above algorithm, it should be clear why accuracy results differ.

It is also good to note that, the above method uses less memory since it does not need to store raw vectors in the memory. Using the above algorithm *.init_sims()* method of the *Word2VecKeyedVectors* class can be called with the parameter *replace=True* causing *syn0* vectors to be garbage collected, thus requiring half the memory space compared to my implementation.

3 Part II: Training a Doc2Vec Model for Document Classification

3.1 Reading the Input File

Given dataset of movie plots has been read using the built-in *csv* module. After omitting the first line that describes columns, first 2000 movie genres are saved as the training set and the remaining movie genres are saved to be used as the test set while classifying plots. All of the input lines are saved in order to create vectors later.

3.2 Cleaning up the Movie Plots

All of the movie plots are cleaned up before training a model to create document vectors, using the following methods:

1. Plots are tokenized using *.word_tokenize()* method of the *nltk* module.
2. Stopwords are removed from the plot using a slightly customized list of stopwords. (See section 4.2 for details)
3. Punctuation and numbers are removed from the plots.

After this operation, plots are converted into *TaggedDocument*s in order to be used while creating a model.

3.3 Training a Model

A *Doc2Vec* model is instantiated using the following parameters:

- Vector size: 300
- Epochs: 50
- Window size: 5
- Minimum Frequency: 5
- Training Algorithm: Distributed bag of words

Each of those values are determined using trial and error in order to get a higher accuracy score.

After instantiation, the model is trained (by using *.build_vocab()* and *.train()* methods of the model) using the tagged documents created from given the movie plots and saved as a file (by using the *.save()* method of the model) for future usage.

3.4 Logistic Regression Classifier

A logistic regression classifier (of the *sklearn* module) is instantiated using the following parameters:

- solver: lbfgs
- multi_class: auto
- max_iter: 1000
- tol: 0.5

Those parameters are changed from default because of the warnings from the classifier except the *tol* parameter. Each of those values are determined using trial and error in order to get a higher accuracy score.

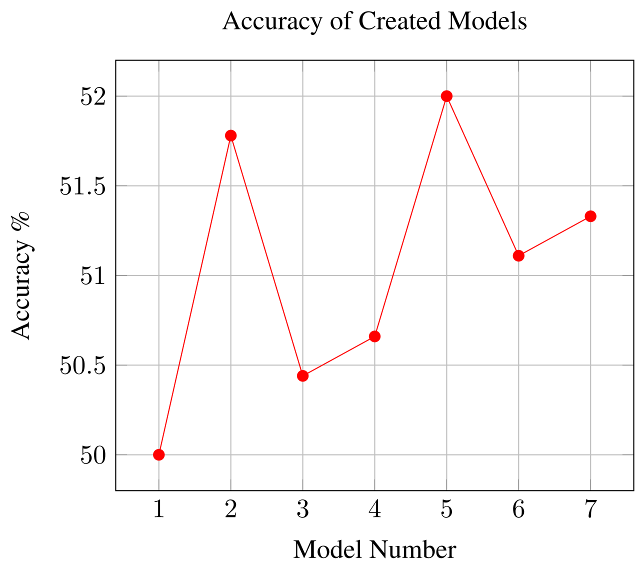
By using the movie plot vectors obtained from a previously trained model and correct categories of those plots, the logistic regression classifier is trained (by using the *.fit()* method of the classifier).

3.5 Calculating the Accuracy

After the training operation, categories of the test vectors are predicted (by using the *.predict()* method of the classifier) and accuracy of the model is calculated according to the below formula:

$$A(W) = \frac{\text{\# of correctly classified documents}}{\text{\# of test documents}} \tag{19}$$

3.6 Results



Using the formula mentioned in the above section, 7 of the created models have been sampled and evaluated. As can be seen from the above plot, their accuracy values vary. This is because *Doc2Vec* uses randomization while creating document vectors. In order to give a more valid result, mean of the obtained accuracy values has been calculated shown in the below figure.

$$Mean_{accuracies} = 51.05 \tag{20}$$

Apart from those samples, a maximum accuracy value of 54.46% is obtained by trying to create a model multiple times.

4 References

4.1 References for Part I

- Pre-trained vectors trained on part of Google News dataset
- Solution to memory error when using gensim module
- L2 normalization
- Gensim's source code for the .most_similar() method
- Argpartition method

4.2 References for Part II

- Doc2Vec documentation
- Logistic regression documentation
- Logistic regression demonstration
- List of english stopwords
- Cleaning up text tutorial