Text Similarity

Submitted by : Preetam Keshari Nahak
116CS0205

1. Introduction

Text similarity is defined as the technique in order to find out how close two text or documents. This similarity is measured considering two aspects i.e. syntactic and semantic structure of the texts. Text similarity method has various application in testing plagiarism, text categorization, topic detection etc.

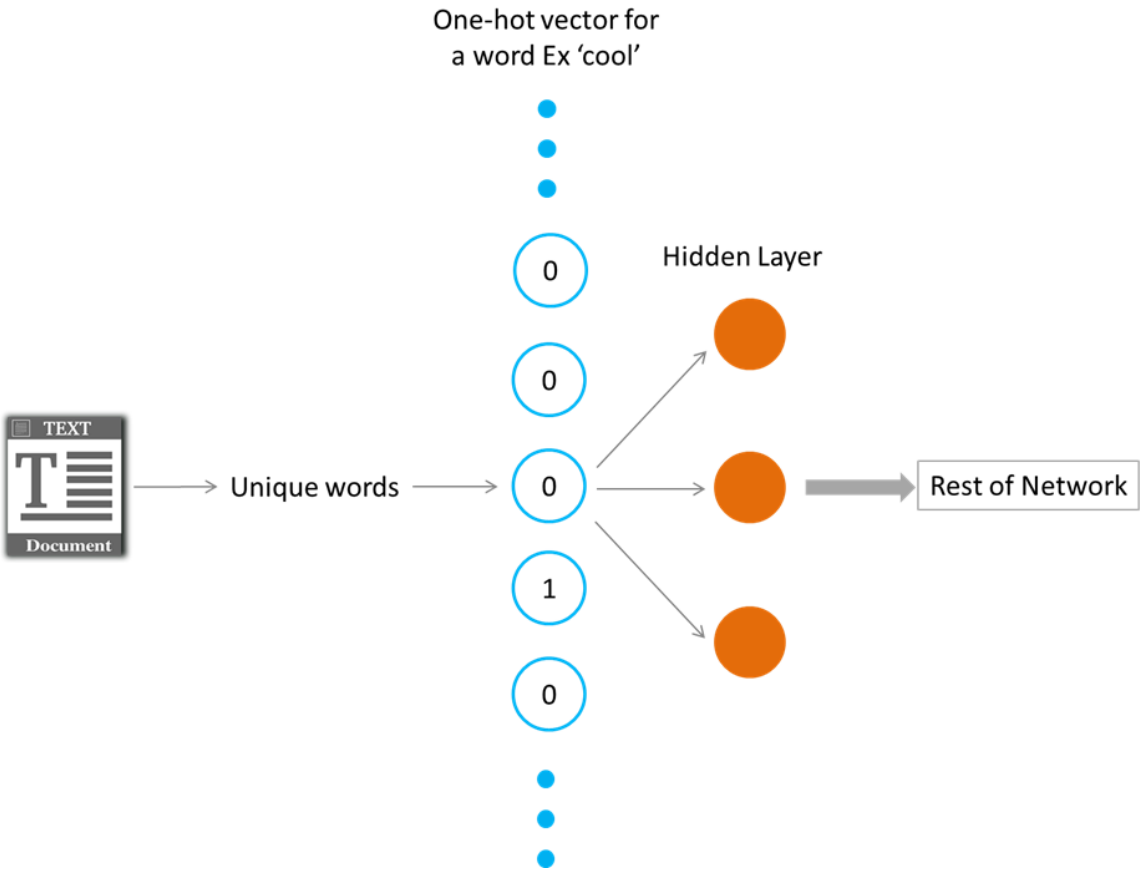
2. Theory

Basically the whole application consists of three main layers/modules. Word2vec, SIF and cosine-similarity module. Let's discuss the functionalities and usage of these one by one.

2.1 Word Embedding

Objective of word embedding is to represent words as vector in a desired dimension say D, which will preserve the similarities among words with similar context. There are two types of word2vec techniques COBW (Common bag of words model) and skip-gram model. We will follow the skip-gram model for word embedding purpose. The basic structure of skip-gram model in Fig 1.

From a set of text or documents, unique vocabulary set is created and each word is one-hot encoded. Then using a definite fixed window size say W, we find the leading W neighbor-hood words and following W neighbor-hood words of the word under consideration, say focused word. Then we prepare pairs of all possible neighbor-hood combination with the focused word. We use these neighboring words as the possible output data for a given input focused word. The we prepare our dataset accordingly. We then feed this processed data to a neural network model with one hidden layer and train the model. After training, the weights available for the hidden layer values will be our word embedding matrix of dimension vocabulary_size*D. When we multiply the one-hot encoded form of a word with the word embedding matrix, we get the vector representation of the word of dimension D.



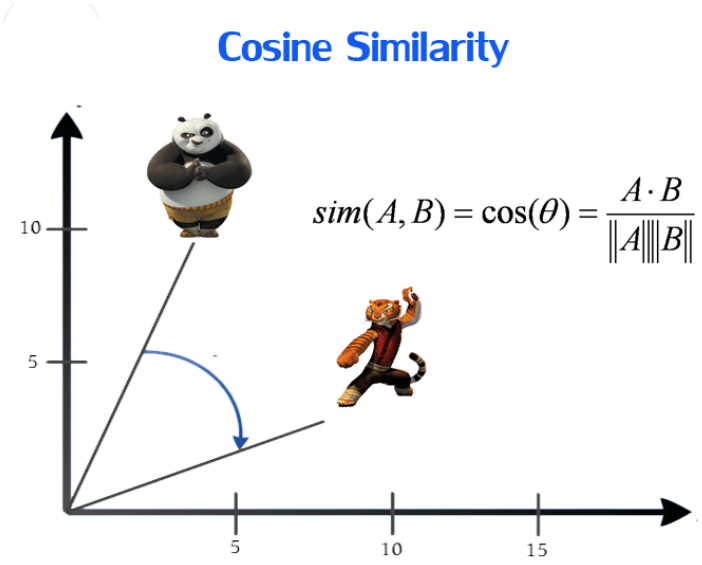
[Fig 1. Skip-gram word2vec method. src : Google]

2.2 Smooth Inverse Frequency

Next, we will discuss about our second module : SIF. SIF is also known as smooth inverse frequency. In order to represent a sentence as a vector in the D-dimensional space, instead of taking the average of all the word embeddings of all the words present in the sentence, we take weighted average of the word embeddings and the weights are determined by `a/(a + p(w))`, where `a` is a parameter that is typically set to `0.001` and `p(w)` is the estimated frequency of the word in a reference corpus.

2.3 Cosine Similarity

In the next step, we use cosine similarity to find similarity between two sentences. A figurative description is given below of the same.



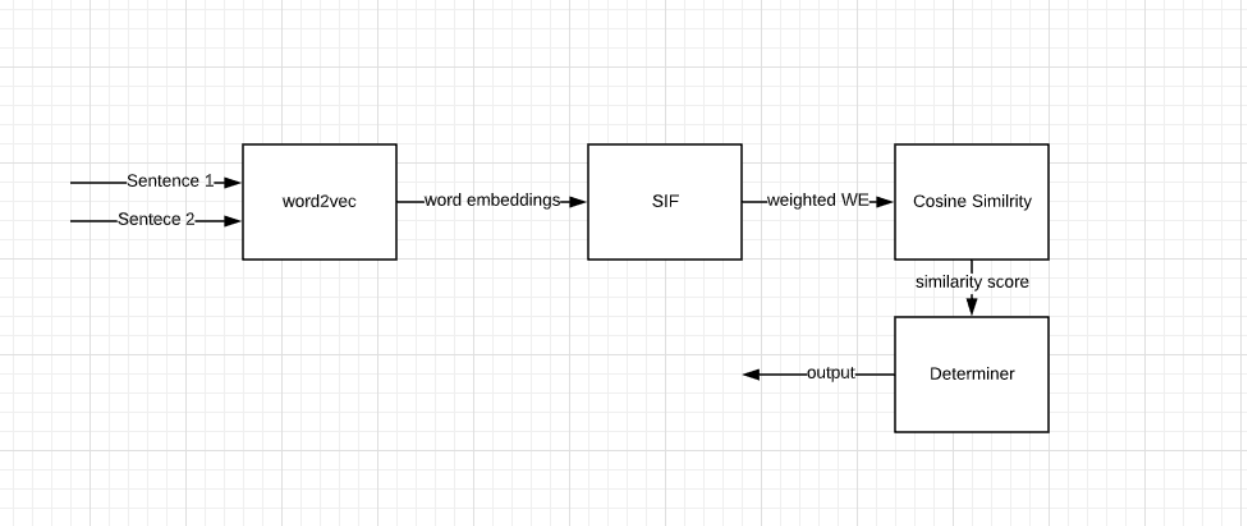
[Fig 2. Cosine similarity]

Then we compare the cosine similarity value with a threshold in the determiner and decide whether we should discard or accept the document.

In order to achieve these, we will first use the master material records to find our word embedding matrix and then we will act on the new material records as described above.

3. Methedology :

Different types of approaches can be adopted to solve the underlying problem. Here we will try to solve the problem based upon an efficient word embedding technique word2vec. The basic skeleton of this approach can be demonstrated as:



[Fig 3. Proposed Architecture]

Corpus we used :

'He is the king . The king is royal . She is the royal queen. King is great. His father was a great king also. His queen is beautiful. '

Preparing training data :

```
#Generating our training data
data = []
WINDOW_SIZE = 3
for sentence in sentences:
    for word_index, word in enumerate(sentence):
        for nb_word in sentence[max(word_index - WINDOW_SIZE, 0) : min(word_index + WINDOW_SIZE, len(sentence)) + 1] :
            if nb_word != word:
                data.append([word, nb_word])
print(data)
```

```
[['he', 'is'], ['he', 'the'], ['he', 'king'], ['is', 'he'], ['is', 'the'], ['is', 'king'], ['the', 'he'], ['the', 'is'], ['the', 'king'], ['king', 'he']
```

```
def convert_to_one_hot(word_index, vocab_size):
```

Building the model :

```
# making placeholders for x_train and y_train
x = tf.placeholder(tf.float32, shape=(None, vocab_size))
y_label = tf.placeholder(tf.float32, shape=(None, vocab_size))
```

```
EMBEDDING_DIM = 5 # can be anything => dimension of embedded vector
W1 = tf.Variable(tf.random_normal([vocab_size, EMBEDDING_DIM]))
b1 = tf.Variable(tf.random_normal([EMBEDDING_DIM])) #bias
hidden_layer = tf.add(tf.matmul(x,W1), b1) #hidden layer
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/op_def_library.py:263: colocate_with (from tensorflow.python.framework.
Instructions for updating:
Colocations handled automatically by placer.
```

```
W2 = tf.Variable(tf.random_normal([EMBEDDING_DIM, vocab_size]))
b2 = tf.Variable(tf.random_normal([vocab_size]))
prediction = tf.nn.softmax(tf.add( tf.matmul(hidden_layer, W2), b2)) #prediction/output layer
```

Training the model:

```
# train for n_iter iterations
for _ in range(n_iters):
    sess.run(train_step, feed_dict={x: X_train, y_label: Y_train})
    print('loss is : ', sess.run(cross_entropy_loss, feed_dict={x: X_train, y_label: Y_train}))
```

```
Streaming output truncated to the last 5000 lines.
loss is :  1.49126
loss is :  1.4912602
loss is :  1.49126
loss is :  1.4912602
loss is :  1.4912602
loss is :  1.49126
loss is :  1.49126
loss is :  1.49126
loss is :  1.4912602
loss is :  1.4912599
loss is :  1.49126
loss is :  1.49126
loss is :  1.49126
loss is :  1.4912602
loss is :  1.4912602
loss is :  1.4912599
loss is :  1.49126
loss is :  1.49126
```

Word embedding matrix:

```
#Our word2vec weight matrix
vect = sess.run(W1 + b1)
print(vect)
```

```
[[-0.64111066  2.2560225   1.6108928   0.7317032   1.4030876 ]
 [-1.1418805   4.1069965  -1.8773841   0.91014975 -0.44532585]
 [ 0.4302839   1.6512817   0.511131   -0.43024534 -3.0498624 ]
 [-0.46371967  3.6087728  -0.6171703   0.82512766  0.18847984]
 [ 0.74079883  0.26191056 -0.82312524 -2.5034394   0.04884979]
 [-1.5415593   1.3926307  -2.6432722  -1.6231148  -0.7741676 ]
 [ 0.986853   -0.9055383  -0.3849998   1.8600459   0.5618466 ]]
```

Verifying if two similar texts are close or not:

```
def euclidean_dist(vec1, vec2):
    return np.sqrt(np.sum((vec1-vec2)**2))

queen_vect = vect[ word2int['queen'] ]
she_vect = vect[ word2int['she'] ]

king_vect = vect[ word2int['king'] ]
he_vect = vect[ word2int['he'] ]

print(euclidean_dist(queen_vect, he_vect))
print(euclidean_dist(queen_vect, she_vect))
```

```
0.79876
0.34987
```

Here we used Euclidian distance two find distance between the word vectors to do a quick check and it turns out to show a good result. In this way we can

include SIF and then cosine similarity further to find closeness of two documents and prevent duplicate documents from storing.

4. Conclusion :

Here we demonstrated an efficient approach to solve the desired problem using deep neural networks. The model can be improved with more data and better model architecture and training.