Learning and implementation of Word Embedding

Wang Xiaonan (Allen)

A0141870J

Introduction

This project aims to explore the concept of word embedding and implement several methods in real datasets. In the first part of the project, I will study the concept of word embedding and its advantages over traditional methods, discuss frequency based embedding and prediction based embedding, and evaluate the word2vec models in details, including the mechanisms of CBOW and Skip-Gram models, benefits of Hierarchical Softmax and negative sampling in practice. In the second half, I will implement the knowledge from word2vec to a dataset to demonstrate its benefits and challenges.

What is word embedding?

Word embedding is a technique in Natural Language Processing(NLP) that converts words in text documents into vectors of real numbers. The end product of word embedding is a vector space model in matrix structure to represent text documents elements as points in space which reveal their semantic and syntactic relationships."

The motivation of adopting word embedding is to improve the accuracy of language model as well as tackle the disadvantages of traditional methods. In Bengio's article on 2003, he mentioned the inadequacy of the traditional and popular n-gram model: "There are at least two characteristics in this approach[n-gram] which beg to be improved upon … First, it is not taking into account contexts farther than 1 or 2 words, second it is not taking into account the "similarity" between words."

While the first problem has been improved since 2003 with advances in computation capacities allowing n greater than 3, the second problem remains a crucial disadvantage of the n-gram model. Bengio gives an example to illustrate the problem: if we have observed the sentence "The cat is walking in the bedroom" in the training corpus, we should be able to generalize and predict the sentence "A dog was running in a room" since the two sentences are of the similar structure, and the word "dog" and "cat", "the" and "a", "room" and "bedroom" have similar semantic and grammatical roles. Since n-gram method only considers the possibility of a word occurring given the previous one or few words, it fails to capture the semantic meaning of the words or similarity between words. It is a setback because in reality we do not have a training corpus that captures every possible word combinations or sentences; hence word embedding is introduced as an attempt to study the relationship between words using vector representations.

Frequency Based Embedding

1. Count Vector

Count vector is a simple way to represent the frequency of word occurrences in the documents. This method requires us to sum the number of occurrences of each word in each document. The final matrix can be interpreted from word vectors (rows) and document vector (columns), where word vectors represent the number of occurrences of the words in different documents, and document vector shows the distribution of words in one document.

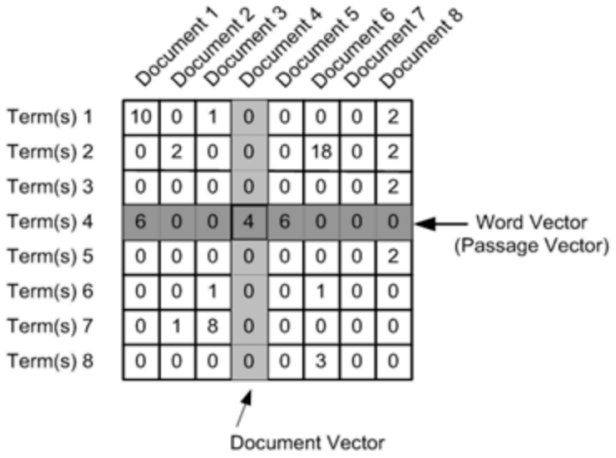

Fig 1: example of count vector

This method is straightforward in demonstrating the occurrences of words in different document. However, its effectiveness can be mitigated by the common words like “the” and “a” since they appear frequently in most documents.

2. TF-IDF Vector

To deal with the common word problem in Count Vector, TF-IDF Vector has been introduced to reduce the effect of common words in analysing the frequency of words. This method works by penalising common words that appear in most documents by assigning lower weights while assigning higher weights to words that are more specific to the documents.

TF-IDF Vector consists of two components as suggested by its name. For each word-document pair, we first calculate TF for by dividing the number of occurrence of this word by the total number of words in this document. Secondly we use IDF = log(N/n) to compute the weights of this word, where N stands for the number of documents and n is the number of documents that this word has appeared in. Finally we use the product of TF and IDF to represent the frequency of word occurrences.

3. Co-Occurrence Vector

Unlike the previous two methods that consider the occurrences of the word itself, Co-Occurrence Vector emphasises on the co-occurrence of one particular word with other words in the context window—with specified width and direction. For example, a 2 (around) context window will consider the four words around the main word.

A Co-Occurrence vector regarding one particular word is constructed by summing the number of occasions when this word co-occurs with each of the remaining words within the context window. The matrix comprising these vectors has a dimension V*V where V equals the number of words in total. To reduce computational difficulty, it is usually decomposed using techniques like PCA or SVD to reduce dimensions by choosing the top significant vectors/components. The Co-Occurrence method is useful because it preserves the semantic relationships between the words.

Prediction Based Embedding

Frequency based embedding models are inadequate because we usually need to make predictions about words. Similar to Co-Occurrence Vector, many prediction based embedding models are interested in exploring the semantic relationships between words to improve prediction accuracy. The distributional hypothesis suggests that words appear in the same contexts tend to be similar, and vice versa. In Sahlgren's article, he states "there is a correlation between distributional similarity and meaning similarity, which allows us to utilize the former in order to estimate the latter." Additionally, word-embedding models also need to avoid the curse of dimensionality. This problem can be introduced by multiplication of vectors leading to a sparse matrix, such as the process of one hot encoding. In 2003, Bengio introduced prediction based embedding by using neural network to construct a distributed representation of words, which "allows each training sentence to inform the model about an exponential number of semantically neighboring sentences". In this essay's setting, the vocabulary size is V, and the hidden layer size is N, and the training corpus consists of a sequence of T training words $w_1, w_2, w_3, \ldots, w_T$.

1. Bengio et al's model

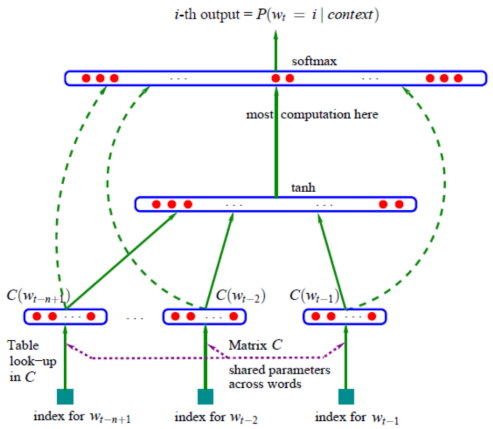

Fig 2: classic neural language model (Bengio et al., 2003)

This model consists of three crucial layers of neural networks:

1) Embedding Layer: a layer that generates word embeddings matrix (V×V) by multiplying an index vector (V) with a word embedding matrix (weights).
2) Intermediate Layer(s): one or more layers that are used to multiply with the embedding layer to obtain vector representation of the words.
3) Softmax Layer: the final layer that takes in output vectors and produces a probability distribution over words in V. The probability distribution is of the form:

$$p(w_t \mid w_{t-1}, \cdots, w_{t-n+1}) = \frac{\exp(h^\top v'_{w_t})}{\sum_{w_i \in V} \exp(h^\top v'_{w_i})}.$$

Where h is the specific row of the selected word from the embedding layer, and v'w is the column regarding the selected word from the intermediate layer. The objective function of the model is:

$$J_\theta = \frac{1}{T} \sum_{t=1}^{T} \log f(w_t, w_{t-1}, \cdots, w_{t-n+1}).$$

Which aims to maximise the probability of predicting each word given the previous n words. A significant challenge of the model is to mitigate the computational cost of computing the softmax layer. Since the cost is proportional to the number of words V, it takes a long time in practice since this number is usually in billions.

2. word2vec

The word2vec algorithm, originally created by Mikolov and his colleagues at Google, has built upon Bengio's framework and became a popular method in NLP. This algorithm consists of two methods: CBOW (Continuous Bag of words) and Skip-Gram model.

2.1 CBOW (One-word Context)

The CBOW model is able to predict the probability of a word given its surrounding words. The input of the CBOW model is a context consisting of one or a few words. We will first consider the case when there is only one word in the context.

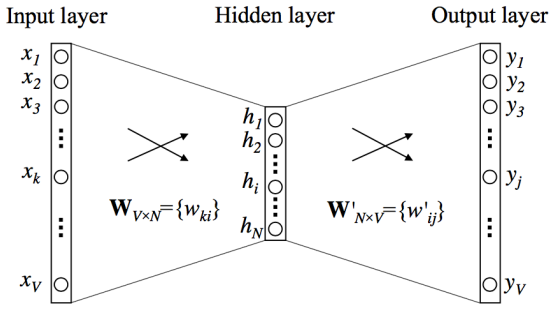

Fig 3: A simple CBOW model with only one word in the context

As shown above, the structure of the CBOW model consists of three layer: input layer, hidden layer and output layer. The input is a one-hot encoded vector, which means for a given word, only one out of V units, $\{x_1, \cdots, x_V\}$, is 1, while all other units are 0.

The weights between the input layer and the hidden layer can be represented by a V × N matrix W. Each row of W is the N-dimension vector representation v_w of the associated word of the input layer. The weights in W are randomly generated. N is chosen as a suitable dimension to represent the words. h refers to the matrix product of $W^T X$. Since the input is one-hot encoded, h simply equals the corresponding row in W.

From the hidden layer to the output layer, there is another N × V matrix W′. W′ multiples with hidden output h to obtain the output vector. This vector provides the score u_j for each word in V:

$$u_j = \mathbf{v}'^{T}_{w_j} \mathbf{h},$$

Where v_w' is the j-th column of the matrix W'. We then use softmax activation function to calculate the final distribution of words and compare with the target.

$$p(w_j|w_I) = y_j = \frac{\exp(u_j)}{\sum_{j'=1}^{V} \exp(u_{j'})},$$

The target is one hot encoded with the target word being 1 and the rest of term being 0. Error between output and target will be calculated and propagated back to re-adjust the weights. Since the objective function is $E = -\log p(w_o|w_i)$, taking the derivative of E with regard to j-th unit's net input u_j will obtain:

$$\frac{\partial E}{\partial u_j} = y_j - t_j := e_j$$

Where $t_j = 1(j = j^*)$, and e_j refers to the prediction error on the output layer. The gradient on the weights of W' can be calculated as:

$$\frac{\partial E}{\partial w'_{ij}} = \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial w'_{ij}} = e_j \cdot h_i$$

From here, we can use stochastic gradient descent to update the weight using this equation:

$$\mathbf{v}'^{(\text{new})}_{w_j} = \mathbf{v}'^{(\text{old})}_{w_j} - \eta \cdot e_j \cdot \mathbf{h} \qquad \text{for } j = 1, 2, \cdots, V.$$

Where $\eta > 0$ is the learning rate. The input vector W is then updated in similar procedures. With increasing iteration these two vectors will be updated such that the prediction error is reducing. The final output vector W' will be taken as the final vector representation of the words.

2.2 CBOW (Multi-word Context)

Usually the context of input for CBOW model has more than one word. The structure of the model is illustrated by figure 4:

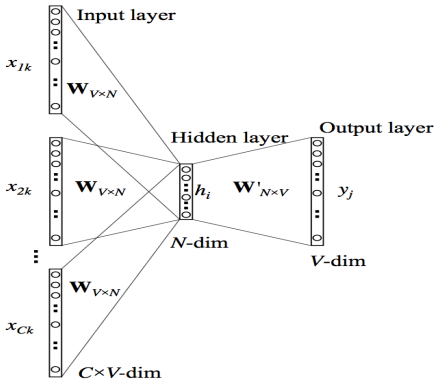

Fig 4: CBOW model with Multi-word Context

While most of the procedure is similar to the one-word setting, the input vector regarding the context words will be averaged to provide a single vector to be multiplied with the hidden-output vector W'. After computing the final output, we sum the prediction error for every context word and average over the number of context words to obtain average prediction error. This error is used to back propagate and update the input and output vector, similar to the one-word setting.

2.3 Skip-Gram Model

In contrast to CBOW model where it tries to predict the word given a context, the aim of Skip-Gram Model is to predict the context given a word.

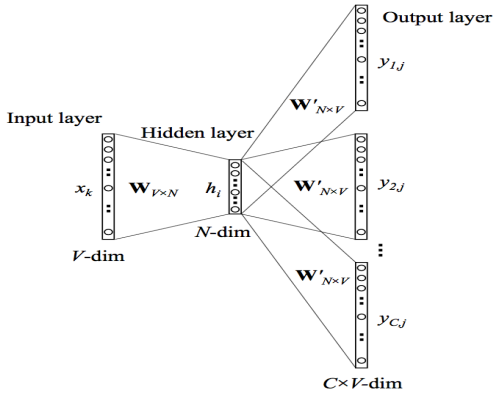

Figure 5: The skip-gram model.

The structure of Skip-Gram Model is similar to that of CBOW model. The only difference occurs in the output layer. In the ouput layer, C separate errors are calculated with respect to the C target variables. This is computed by subtracting the target vectors from the same vector containing softmax possibilities. These C error vectors are then summed together to obtain a final error vector that is propagated back to update the weights.

Implementation of Word Embedding

In this session, I will apply word-embedding techniques on a real world dataset to examine its effectiveness while comparing the classification results with other traditional methods. The dataset used is the "Twitter Sentiment Corpus" by Niek Sanders available at http://thinknook.com/twitter-sentiment-analysis-training-corpus-dataset-2012-09-22/. The dataset contains 1,578,627 classified tweets; each row is marked as 1 for positive sentiment and 0 for negative sentiment.

1. Examining most frequent words

To understand the dataset, I first tokenize the texts and find out which are top words in positive tweets negative tweets. In this session I used the regular expression from NLTK library and remove the punctuations while tokenize the words. The top 15 words in both categories are shown below:

```
pos_topwords[:15]
```

```
[('good', 60865),
 ('day', 54488),
 ('love', 50255),
 ('http', 46264),
 ('quot', 45471),
 ('wa', 45392),
 ('u', 39735),
 ('like', 38297),
 ('get', 38031),
 ('lol', 36000),
 ('com', 34833),
 ('thanks', 34607),
 ('â', 34263),
 ('time', 33983),
 ('going', 30546)]
```

```
neg_topwords[:15]
```

```
[('wa', 59386),
 ('day', 50092),
 ('get', 47872),
 ('go', 47605),
 ('work', 45586),
 ('like', 41367),
 ('today', 38104),
 ('want', 33688),
 ('going', 33418),
 ('got', 33031),
 ('back', 32674),
 ('miss', 31551),
 ('time', 31319),
 ('really', 31172),
 ('im', 30988)]
```

We observe common words like “day” and “go”, but there are also different words such as “good” and “thanks” in positive tweet and “miss” in negative tweets. Now we try to train a word2vec model to see whether it can capture the information in these tweets.

2. Training Word2vec model

We now try to build a word2vec model using the texts. After some research, I found out the tweet tokenizor from NLTK library for tokenizing tweets. I fit the tokenizer to the texts and use the sentences as the input for the word2vec model.

To train the word2vec model, I utilized the Gensim library and used the default parameters to create a vector representation of words with 100 dimensions. I then examine the effectiveness of the model in performing some of the famous capabilities of word2vec model.

To evaluate the performances of word embedding models, Google have released their testing set of about 20,000 syntactic and semantic test examples, following the "A is to B as C is to D" task. The test examines whether the model is able to capture the similarity and differences between words.

The total accuracy is 25.8%, which shows that the accuracy is not very high for most of the questions. However we observe that it has obtained 58.2% accuracy for family-related words and decent scores for gram3-comparative (44.9%), gram5-present-participle (53.8%), gram7-past-tense (42.7%) and gram9-plural-verbs (38.1%). From these we can interpret that the tweets are more related to these topics. Besides, since most of the tweets are informal and more related to personal emotions and experiences, this text may not be representative because it also includes topics like currency and location information (state names).

Besides the test, I also explore the effectiveness of the model in performing the famous task "king + man – women = queen" task. The similarity tests also show that our word2vec model has preserved the relationship between the words.

```
print(model.most_similar(positive=['woman', 'actress'], negative=['man'], topn=1))
print(model.most_similar(positive=['dad', 'family'], negative=['man'], topn=2))
```

```
[('actor', 0.8064113259315491)]
[('grandma', 0.721605658531189), ('mom', 0.7162371873855591)]
```

```
print("model.similarity('dad', 'mom'): %f" %model.similarity('dad', 'mom'))
print("model.similarity('happy','homework'): %f" %model.similarity('happy','homework'))
```

```
model.similarity('dad', 'mom'): 0.957098
model.similarity('happy','homework'): 0.069131
```

3. word2vec model for classification

In my study, I found out that as compared to the tweet tokenizer, the regular expression tokenizer results in better performance in classification task. Therefore, I build a new word2vec model with regular expression tokernized texts.

According to Tomas Mikolov, very minimal text cleaning is required when learning a word-embedding model. Hence, I just tokenize the words using regular expression to remove the punctuations and change the words to lower case. Stop words are not removed and no forms of stemming and lemmatization have been performed. After training the w2v model, I created a dictionary "w2v" to store the words and its associated 100 vectors.

4. Traditional classification methods

In this session I apply some of the traditional classification methods on the dataset. The benchmark model is Multinomial Naïve Bayes, which ends up having a better performance than some of the more complex models. Other models include logistic regression with cross validation, random forest classifier, gradient boosting classifier and stochastic gradient descent (SGD) classifier. Random forest is included due to its

effectiveness in dealing with large number of features, and SGD has proven to be an efficient approach to discriminative learning of linear classifiers under convex loss functions such as support vector machines.

During the training, some parameters are regulated to cope with the large number of features and the risk of over fitting. Max_feature, max_depth and subsample are controlled for random forest, while penalty of elastic net has been introduced in SGD.

The data is then split into 75% of training set and 25% of testing set. To prepare the data for training, I utilized the sklearn library to tokenize the sentences into words and output a sparse matrix. I use the TfidfVectorizer to vectorize the words while assigning weights through TF-IDF, removing stopwords and converting the text to lowercase. The output matrix for training data has a shape of (1183971, 562809), while 118397 is the number of documents and 562809 is the number of tokens in the documents.

The various models then fit on the training set and validate on the testing set. Performance metrics like prediction accuracy, AUC, confusion matrix and classification reports are evaluated.

The classification result is as followed:

Model	Multinomial NB	Logistic Regression	Random Forest	Gradient Boosting	SGD Classifier
Prediction Accuracy	0.754309	0.770769	0.739393	0.705119	0.754985
AUC	0.838127	0.850202	0.812129	0.773016	NA

We observe that logistic regression obtained the best result as represented by the accuracy scores and other performance metrics. Surprisingly, Multinomial Naïve Bayes is second despite its simple structure.

5. Classification using averaged w2v output

Now we try to classify the texts using the word2vec model and compare with the other models. To utilize the w2v output from our study, I learnt from a blog post online (see reference) and converted the vector representation from step 3 to an input matrix that can be fit in to traditional classification methods.

After apply TfidfVectorizer to tokenize each documents, for each word in the text, we look for its 100 vector representations in the dictionary and then average to obtain its weights. The output matrix thus has a shape of (1578627,683263), where 1578627 is the number of documents and 683263 is the number of unique tokens.

Now we fit the logistic regression model on the input data because it is the best model from step 4. The performance of the model is shown together with the previous models.

Model	Multinomial NB	Logistic Regression	Random Forest	Gradient Boosting	SGD Classifier	W2V LRcv
Prediction Accuracy	0.754309	0.770769	0.739393	0.705119	0.754985	0.753804
AUC	0.838127	0.850202	0.812129	0.773016	NA	0.832537

The result shows that logistic regression with word2vec result does not perform as well as the original logistic regression. However it is better than random forest and gradient boosting in terms of both prediction accuracy and AUC score. The reason of its unsatisfying result could be due to the reason of increasing features due to different tokenization techniques in dealing with the texts (w2v produced more tokens). Besides, averaging the vectors may result in loss of information and might not be representative of the words.

6. Classification using CNN with w2v

From step 5 we realized that averaging the vectors might not be the best way to utilize the output of word2vec models. The more logical way would then be using a neural network. In this session, I used the Keras library to build a 1D convolutional neural networks for classification task.

Firstly, I created a sequence with dimension (1578627, 303) to represent the words in the texts. This format allows us to feed the text samples and labels into a neural network. The texts are then randomized and 75% of the data are selected as training set. Secondly, the embedding matrix is created by parsing the w2v vector representation to each unique token. The embedding layer output is a 3D tensor of shape (samples, sequence_length(303), embedding_dim(100)). We then build a simple CNN to classify the texts. However, due to the large data size, the program crashes in my laptop and we obtained result as followed:

```
model = Model(sequence_input, preds)
model.compile(loss='categorical_crossentropy',optimizer='rmsprop',metrics=['acc'])

model.fit(x_train, y_train,batch_size=128,epochs=2,validation_data=(x_val, y_val))
```

```
Train on 1183971 samples, validate on 394656 samples
Epoch 1/2
 180608/1183971 [===>..........................] - ETA: 47:09 - loss: 0.5567 - acc: 0.7117
```

In order to examine the effectiveness of CNN on w2v, I decided to train on a smaller subset of the data (10%). After selecting 10% of the data and separating them into training and testing set, I run the model with 2 epochs and obtained following results:

```
model = Model(sequence_input, preds)
model.compile(loss='categorical_crossentropy',optimizer='rmsprop',metrics=['acc'])

model.fit(x_train, y_train,batch_size=128,epochs=2,validation_data=(x_val, y_val))
```

```
Train on 118397 samples, validate on 39465 samples
Epoch 1/2
118397/118397 [==============================] - 363s 3ms/step - loss: 0.5878 - acc: 0.6855 - val_loss: 0.5667 - val_
acc: 0.7069
Epoch 2/2
118397/118397 [==============================] - 398s 3ms/step - loss: 0.5538 - acc: 0.7129 - val_loss: 0.5681 - val_
acc: 0.7044
```

Even though we observe significant improvement of prediction accuracy from 0.6855 to 0.7129 after two epochs, the result is still not as good as the previous methods. I feel that it might be meaningful to use pre-trained word embedding which captures more information.

7. Classification using CNN with pre-trained word embedding

Since we are studying tweets, I chose the Glove vectors for tweets with 100 dimensions. After importing the data, I created the new embedding layer by mapping the vector representation from the imported embedding to the tokens. Due to the limitation of commutation power, I run the model for six epochs manually, two in each row, and obtained the result as followed:

```
model = Model(sequence_input, preds)
model.compile(loss='categorical_crossentropy',optimizer='rmsprop',metrics=['acc'])
model.fit(x_train, y_train, batch_size=128, epochs=2,validation_data=(x_val, y_val))
```

```
Train on 118397 samples, validate on 39465 samples
Epoch 1/2
118397/118397 [==============================] - 368s 3ms/step - loss: 0.5884 - acc: 0.6861 - val_loss: 0.5672 - val_
acc: 0.7008
Epoch 2/2
118397/118397 [==============================] - 392s 3ms/step - loss: 0.5525 - acc: 0.7137 - val_loss: 0.5588 - val_
acc: 0.7077

<keras.callbacks.History at 0x28e84f6a0>
```

```
model.fit(x_train, y_train, batch_size=128, epochs=2,validation_data=(x_val, y_val))
```

```
Train on 118397 samples, validate on 39465 samples
Epoch 1/2
118397/118397 [==============================] - 366s 3ms/step - loss: 0.5276 - acc: 0.7317 - val_loss: 0.5754 - val_
acc: 0.7001
Epoch 2/2
118397/118397 [==============================] - 395s 3ms/step - loss: 0.5039 - acc: 0.7462 - val_loss: 0.5694 - val_
acc: 0.7076

<keras.callbacks.History at 0x28ec98438>
```

```
model.fit(x_train, y_train, batch_size=128, epochs=2,validation_data=(x_val, y_val))
```

```
Train on 118397 samples, validate on 39465 samples
Epoch 1/2
118397/118397 [==============================] - 370s 3ms/step - loss: 0.4801 - acc: 0.7613 - val_loss: 0.5863 - val_
acc: 0.7001
Epoch 2/2
118397/118397 [==============================] - 394s 3ms/step - loss: 0.4575 - acc: 0.7732 - val_loss: 0.6121 - val_
acc: 0.7013
```

We observed that accuracy increases from 0.6861 to 0.7732 after six epochs.

Evaluation

From the implementation on the twitter dataset, we evaluated the effectiveness of word2vec model in capturing the information of the words and compare its performance with other traditional classification methods.

Firstly, although CNN obtains the highest accuracy score after training on six epochs, it only utilized 10% of the data due to computational limitations and we cannot guarantee that it will out perform the traditional methods for the whole dataset. Therefore, the long training time and high requirement of computational power of CNN may make it difficult for us to implement word2vec in classification tasks. As compared to CNN, averaging the vector representations of word2vec output can be an easier way to make use of word embedding information in classification task. However, this method is also subject to how we tokenize the words.

Secondly, while the classification result utilizing word2vec are not very satisfying in this study, we should note that this is a tweet dataset that includes many informal words, symbols and text emoticons. It also suggests that we might need to spend more time in preprocessing the texts before feeding them into the word2vec model.

Thirdly, as Mikolov and many others argue, we should not evaluate the effectiveness of word2vec model solely based on its classification accuracy. Word2vec models are famous for its capacities in capturing the semantic relationship of the words and retaining valuable information for further studies, such as the similarity tests that we observed in part 2.

Overall, this study explore the effectiveness of word2vec in various text analytics tasks, demonstrated its strengths in retaining semantic relationship among the words and explore some of the possibilities of implementing word2vec result in text classification task. The code of the study is attached in the appendix.

References

Bengio, Y., Ducharme, R., Vincent, P., & Janvin, C. (2003). A Neural Probabilistic Language Model. The Journal of Machine Learning Research, 3, 1137–1155. Retrieved from http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf

GloVe: Global Vectors for Word Representation, https://nlp.stanford.edu/projects/glove/

Google testing set, https://raw.githubusercontent.com/RaRe-Technologies/gensim/develop/gensim/test/test_data/questions-words.txt.

Keras CNN,

https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html

https://github.com/keras-team/keras/blob/master/examples/pretrained_word_embeddings.py

Kerr, S. J. (2017, October 03). Getting Started With Word Embedding in R. Retrieved March 11, 2018, from http://programminghistorian.github.io/ph-submissions/lessons/getting-started-with-word-embeddings-in-r#fnref:4

Mikolov, T., Corrado, G., Chen, K., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. Proceedings of the International Conference on Learning Representations (ICLR 2013), 1–12.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. NIPS, 1–9.

N., Dar, P., & Srivastava, P. (2017, June 06). Intuitive Understanding of Word Embeddings: Count Vectors to Word2Vec. Retrieved March 11, 2018, from https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/

Ruder, S. (2018, March 03). On word embeddings - Part 1. Retrieved March 11, 2018, from http://ruder.io/word-embeddings-1/

Rong, X. (2014). word2vec Parameter Learning Explained. arXiv:1411.2738 [cs].

Sahlgren, M. (2008). The distributional hypothesis. Italian Journal of Linguistics, 20 (1), 33–53.

Traditional classification methods by word embedding,

http://nadbordrozd.github.io/blog/2016/05/20/text-classification-with-word2vec/, https://github.com/nadbordrozd/blog_stuff/blob/master/classification_w2v/benchmarking_python3.ipynb

Twitter Sentiment Corpus, http://thinknook.com/twitter-sentiment-analysis-training-corpus-dataset-2012-09-22/

Appendix