Sentiment Classification using Word Embeddings (Word2Vec)

Dipika Baad · Follow
Published in The Startup
11 min read · Mar 3, 2020

Background to Word Embeddings and Implementing Sentiment Classification on Yelp Restaurant Review Text Data using Word2Vec.

How the word embeddings are learned and used for different tasks will be explored in the beginning followed by using Word2Vec vectors for doing sentiment classification on Yelp Restaurant Review Dataset.

In my previous posts of Sentiment Classification using BOW and Sentiment Classication using TFIDF , I have covered the topics of preprocessing the text and loading the data. This will be similar to those posts and we can quickly compare the results to those at the end as well. This post is one step further where little more complex method for representing the text is used to get the vectors for documents which tries to capture more than just the word information/ importance. Let's begin with loading the data.

Restaurant Reviews by Sentiment Example by Dipika Baad

Load the data

Yelp restaurant review dataset can be downloaded from their site and the format of the data present there is JSON. The data provided is actually not in correct json format readable for python. Each row is dictionary but for it to be a valid json format, a square bracket should be at the start and end of the file with , being added at end of each row. Define the `INPUT_FOLDER` as folder path in your local directory where yelp review.json file is present. Declare `OUTPUT_FOLDER` as a path where you want to write the output from the following function. Loading of json

data and writing the top 100,000 rows is done in the following function:

```
import pandas as pd

def load_yelp_orig_data():
    PATH_TO_YELP_REVIEWS = INPUT_FOLDER + '/review.json'

    # read the entire file into a python array
    with open(PATH_TO_YELP_REVIEWS, 'r') as f:
        data = f.readlines()

    # remove the trailing "\n" from each line
    data = map(lambda x: x.rstrip(), data)

    data_json_str = "[" + ','.join(data) + "]"

    # now, load it into pandas
    data_df = pd.read_json(data_json_str)

    data_df.head(100000).to_csv(OUTPUT_FOLDER + '/output_reviews_top.csv')

load_yelp_orig_data()
```

Once the above function has been run, you are ready to load it in pandas dataframe for the next steps. For the experiment, only small amount of data is taken so that it can be run faster to see the results.

Exploring data

After the data is loaded, new column for sentiment indication is created. It is not always the situation that some column with the prediction label you want to do is present in the original dataset. This can be a derived column in most of the cases. For this case, `stars` column in the data is used to derive sentiment.

```
top_data_df = pd.read_csv(INPUT_FOLDER + 'output_reviews_top.csv')
print("Columns in the original dataset:\n")
print(top_data_df.columns)
```

loading_in_dataframe.py hosted with ❤ by **GitHub** **view raw**

Output:

```
Columns in the original dataset:

Index(['Unnamed: 0', 'business_id', 'cool', 'date', 'funny', 'review_id',
       'stars', 'text', 'useful', 'user_id'],
      dtype='object')
```

After the data is available, mapping from stars to sentiment is done and distribution for each sentiment is plotted.

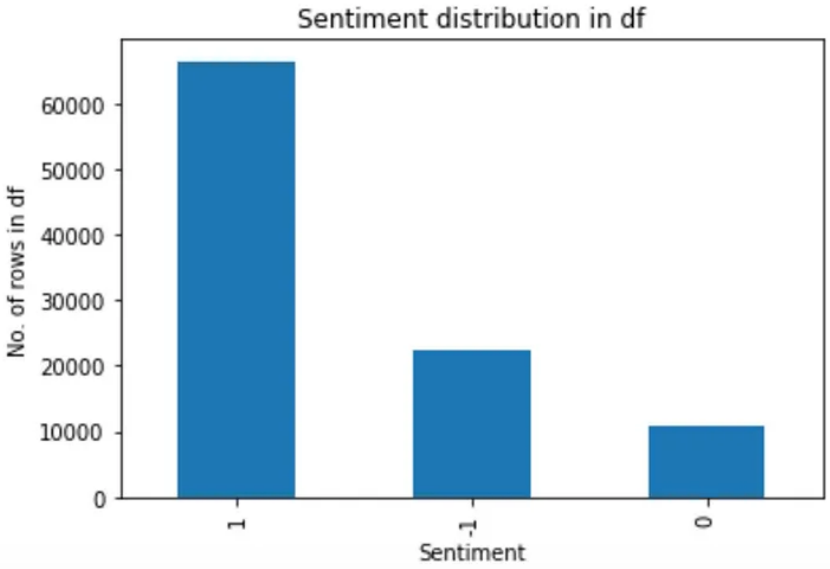
```python
import matplotlib.pyplot as plt

print("Number of rows per star rating:")
print(top_data_df['stars'].value_counts())

# Function to map stars to sentiment
def map_sentiment(stars_received):
    if stars_received <= 2:
        return -1
    elif stars_received == 3:
        return 0
    else:
        return 1
# Mapping stars to sentiment into three categories
top_data_df['sentiment'] = [ map_sentiment(x) for x in top_data_df['stars']]
# Plotting the sentiment distribution
plt.figure()
pd.value_counts(top_data_df['sentiment']).plot.bar(title="Sentiment distribution in df")
plt.xlabel("Sentiment")
plt.ylabel("No. of rows in df")
plt.show()
```

Output:

```
Number of rows per star rating:
5    44367
4    22103
1    14585
3    10970
2     7975
Name: stars, dtype: int64
```



Once that is done, number of rows for each sentiment is checked. Sentiment Classes are as follows:

1. Positive : 1
2. Negative: -1
3. Neutral: 0

Number of rows are not equally distributed across these three sentiments. In this post, problem of imbalanced classes won't be dealt that is why, simple function to retrieve the top few records for each sentiment is written. In this example, `top_n` is 10000 which means total of 30,000 records will be taken.

```python
# Function to retrieve top few number of each category
def get_top_data(top_n = 5000):
    top_data_df_positive = top_data_df[top_data_df['sentiment'] == 1].head(top_n)
    top_data_df_negative = top_data_df[top_data_df['sentiment'] == -1].head(top_n)
    top_data_df_neutral = top_data_df[top_data_df['sentiment'] == 0].head(top_n)
    top_data_df_small = pd.concat([top_data_df_positive, top_data_df_negative, top_data_df_n
    return top_data_df_small

# Function call to get the top 10000 from each sentiment
top_data_df_small = get_top_data(top_n=10000)

# After selecting top few samples of each sentiment
print("After segregating and taking equal number of rows for each sentiment:")
print(top_data_df_small['sentiment'].value_counts())
top_data_df_small.head(10)
```

load_top_reviews.py hosted with ❤ by GitHub view raw

Output:

```
After segregating and taking equal number of rows for each sentiment:
-1    10000
 1    10000
 0    10000
Name: sentiment, dtype: int64
```

	Unnamed: 0	business_id	cool	date	funny	review_id	stars	text	useful	user_id	sentiment
1	1	NZnhc2sEQy3RmzKTZnqtwQ	0	2017-01-14 21:30:33	0	GJXCdrto3ASJOqKeVWPi6Q	5	I *adore* Travis at the Hard Rock's new Kelly ...	0	yXQM5uF2jS6es16SJzNHfg	1
2	2	WTqjgwHlXbSFevF32_DJVw	0	2016-11-09 20:09:03	0	2TzJjDVDEuAW6MR5Vuc1ug	5	I have to say that this office really has it t...	3	n6-Gk65cPZL6Uz8qRm3NYw	1
3	3	ikCg8xy5JIg_NGPx-MSIDA	0	2018-01-09 20:56:38	0	yi0R0Ugj_xUx_Nek0-_Qig	5	Went in for a lunch. Steak sandwich was delici...	0	dacAIZ6fTM6mqwW5uxkskg	1
5	5	eU_713ec6fTGNO4BegRaww	0	2013-01-20 13:25:59	0	fdiNeiN_hoCxCMy2wTRW9g	4	I'll be the first to admit that I was not exci...	0	w31MKYsNFMrjhWxxAb5wIw	1
10	10	8mIrX_LrOnAqWsB5JrOojQ	0	2011-11-30 02:11:15	0	kbtscdyz6lvrtGjD1quQTg	4	Like walking back in time, every Saturday morn...	0	Flk4lQQu1eTe2EpzQ4xhBA	1
12	12	FxLfqxdYPA6Z85PFKaqLrg	0	2016-05-07 01:36:53	0	Z7wgXp98wYB57QdRY3HQ3w	4	Wow. So surprised at the one and two star revi...	0	GYNnVehQeXjty0xH7-6Fhw	1
13	13	LUN6swQYa4xJKaM_UEUOEw	0	2018-04-27 20:25:26	0	qlXw1JQ0UodW7qrmVgwCXw	4	Michael from Red Carpet VIP is amazing ! I rea...	0	bAhqAPoWaZYcyYi7bs024Q	1

How to preprocess text data?

Preprocessing involves many steps like tokenization, removing stop words, stemming/lemmatization etc. These commonly used techniques were explained in detail in my previous post of BOW. Here, only the necessary steps are explained in the next phase.

Why do you need to preprocess this text? — Not all the information is useful in making predictions or doing classifications. Reducing the number of words will reduce the input dimension to your model. The way the language is written, it contains lot of information which is grammar specific. Thus when converting to numeric format, word specific characteristics like capitalisation, punctuations, suffixes/prefixes etc. are redundant. Cleaning the data in a way that similar words map to single word and removing the grammar relevant information from text can tremendously reduce the vocabulary. Which methods to apply and which ones to skip depends on the problem at hand.

1. Removal of Stop Words

Stop words are the words which are commonly used and removed from the sentence as pre-step in different Natural Language Processing (NLP) tasks. Example of stop words are: 'a', 'an', 'the', 'this', 'not' etc. Every tool uses a bit different set of stop words list that it removes but this technique is avoided in cases where phrase structure matters like in this case of Sentiment Analysis.

Example of removing stop words:

```
# Removing the stop words
from gensim.parsing.preprocessing import remove_stopwords
print(remove_stopwords("Restaurant had a really good service!!"))
print(remove_stopwords("I did not like the food!!"))
print(remove_stopwords("This product is not good!!"))
```

stop_words_removal.py hosted with ❤ by GitHub view raw

Output:

```
Restaurant good service!!
I like food!!
This product good!!
```

As it can be seen from the output, removal of stop words removes necessary words required to get the sentiment and sometimes it can totally change the meaning of the sentence. In the examples printed by above piece of code, it is clear that it can convert a negative statement into positive sentence. Thus, this step is skipped for Sentiment Classification.

2. Tokenization

Tokenization is the process in which the sentence/text is split into array of words called tokens. This helps to do transformations on each words separately and this is also required to transform words to numbers. There are different ways of performing tokenization. I have explained these ways in my previous post under Tokenization section, so if you are interested you can check it out.

Gensim's `simple_preprocess` allows you to convert text to lower case and remove punctuations. It has `min` and `max` length parameters as well which help to filter out rare words and most commonly words which will fall in that range of lengths.

Here, `simple_preprocess` is used to get the tokens for the dataframe as it does most of the preprocessing already for us. Let's apply this method to get the tokens for the dataframe:

```
1  from gensim.utils import simple_preprocess
2  # Tokenize the text column to get the new column 'tokenized_text'
3  top_data_df_small['tokenized_text'] = [simple_preprocess(line, deacc=True) for line in top_da
4  print(top_data_df_small['tokenized_text'].head(10))
```

tokenize_with_simple_preprocess.py hosted with ❤ by GitHub view raw

Output:

```
1     [adore, travis, at, the, hard, rock, new, kell...
2     [have, to, say, that, this, office, really, ha...
3     [went, in, for, lunch, steak, sandwich, was, d...
5     [ll, be, the, first, to, admit, that, was, not...
10    [like, walking, back, in, time, every, saturda...
12    [wow, so, surprised, at, the, one, and, two, s...
13    [michael, from, red, carpet, vip, is, amazing,...
15    [you, can, really, find, anything, wrong, with...
16    [great, lunch, today, staff, was, very, helpfu...
18    [we, ve, been, huge, slim, fan, since, they, o...
Name: tokenized_text, dtype: object
```

3. Stemming

Stemming process reduces the words to its' root word. Unlike Lemmatization which uses grammar rules and dictionary for mapping words to root form, stemming simply removes suffixes/prefixes. Stemming is widely used in the application of SEOs, Web search results, and information retrieval since as long as the root matches in the text somewhere it helps to retrieve all the related

documents in the search.

There are different algorithms used to do the stemming. PorterStammer(1979), LancasterStammer (1990), and SnowballStemmer (can add custom rules). NLTK or Gensim package can be used for implementing these algorithms for stemming. Lancaster is bit slower than Porter so we can use it according to size and response time required. Snowball stemmer is a slightly improved version of the Porter stemmer and is usually preferred over the latter. It is not very clear which one will produce accurate results, so one has to experiment different methods and choose the one that gives better results. In this example, Porter Stemmer is used which is simple and speedy. Following code shows how to implement stemming on dataframe and new column `stemmed_tokens` is created:

```
from gensim.parsing.porter import PorterStemmer
porter_stemmer = PorterStemmer()
# Get the stemmed_tokens
top_data_df_small['stemmed_tokens'] = [[porter_stemmer.stem(word) for word in tokens] for tok
top_data_df_small['stemmed_tokens'].head(10)
```

stemming.py hosted with ❤ by GitHub view raw

Output:

```
1     [ador, travi, at, the, hard, rock, new, kelli,...
2     [have, to, sai, that, thi, offic, realli, ha, ...
3     [went, in, for, lunch, steak, sandwich, wa, de...
5     [ll, be, the, first, to, admit, that, wa, not,...
10    [like, walk, back, in, time, everi, saturdai, ...
12    [wow, so, surpris, at, the, on, and, two, star...
13    [michael, from, red, carpet, vip, is, amaz, re...
15    [you, can, realli, find, anyth, wrong, with, t...
16    [great, lunch, todai, staff, wa, veri, help, i...
18    [we, ve, been, huge, slim, fan, sinc, thei, op...
Name: stemmed_tokens, dtype: object
```

Splitting into Train and Test Sets:

Train data would be used to train the model and test data is the data on which the model would predict the classes and it will be compared with original labels to check the accuracy or other model test metrics.

- Train data (Subset of data for training ML Model) ~70%
- Test data (Subset of data for testing ML Model trained from the train data) ~30%

Try to balance the number of classes in both the sets so that the results are not biased or one of the reasons for insufficient model training. This is a crucial part of machine learning model. In real-world problems, there are cases of imbalanced classes which needs using techniques like oversampling minority class, undersampling majority class (Resample function from scikit-learn packaged or generating synthetic samples using SMOTE functionality in Imblearn package .

For this case, the data is split into two parts, train and test with 70% in train and 30% in test. While making the splitting, it is better to have equal distribution of classes in both train and test data. Here, function train_test_split from scikit-learn package is used.

```
from sklearn.model_selection import train_test_split
# Train Test Split Function
def split_train_test(top_data_df_small, test_size=0.3, shuffle_state=True):
    X_train, X_test, Y_train, Y_test = train_test_split(top_data_df_small[['business_id', 'c
                                                        top_data_df_small['sentiment'],
                                                        shuffle=shuffle_state,
                                                        test_size=test_size,
                                                        random_state=15)
    print("Value counts for Train sentiments")
    print(Y_train.value_counts())
    print("Value counts for Test sentiments")
    print(Y_test.value_counts())
    print(type(X_train))
    print(type(Y_train))
    X_train = X_train.reset_index()
    X_test = X_test.reset_index()
    Y_train = Y_train.to_frame()
    Y_train = Y_train.reset_index()
    Y_test = Y_test.to_frame()
    Y_test = Y_test.reset_index()
    print(X_train.head())
    return X_train, X_test, Y_train, Y_test

# Call the train_test_split
X_train, X_test, Y_train, Y_test = split_train_test(top_data_df_small)
```

split_data_train_test.py hosted with ❤ by GitHub view raw

Output:

```
Value counts for Train sentiments
 1    7036
-1    7008
 0    6956
Name: sentiment, dtype: int64
Value counts for Test sentiments
 0    3044
-1    2992
 1    2964
Name: sentiment, dtype: int64
<class 'pandas.core.frame.DataFrame'>
<class 'pandas.core.series.Series'>
   index  ...                                    stemmed_tokens
0  34963  ...  [wait, minut, onli, to, get, to, front, of, th...
1  21265  ...  [chicken, bacon, ranch, pizza, wa, pretti, bom...
2  36472  ...  [as, sketchi, as, hell, thi, place, strike, me...
3  36589  ...  [last, time, tri, bubbl, tea, and, it, wa, dis...
4   3117  ...  [so, for, the, game, last, week, wa, visit, th...

[5 rows x 11 columns]
```

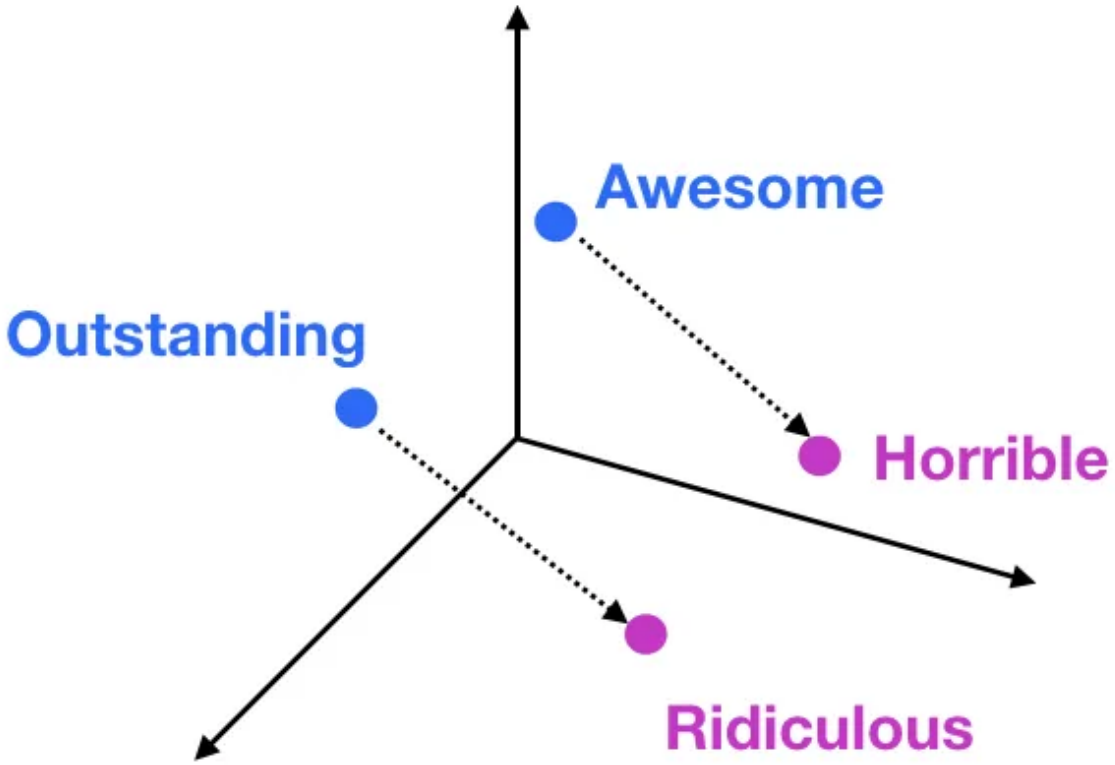
As it can be seen from the above output, data is distributed for each classes proportionately. Number of rows for each sentiment in train and test are printed.

Understanding Word2Vec Model

Word Embeddings

Word embeddings are words mapped to real number vectors such that it can capture the semantic meaning of words. The methods tried in my previous posts of BOW and TFIDF do not capture the meaning between the words, they consider the words seperately as features. Word embeddings use some models to map a word into vectors such that similar words will be closer to each other. As shown in the below figure, for example some of the positive words which are adjectives will be closer to each other and vice versa for negative adjectives. It captures semantical and syntactical information of words. To train this model it takes into consideration the words surrounding that word of particular window size. There are different ways of deriving the word embedding vectors. Word2vec is one such method where neural embeddings model is used to learn that. It uses following two architectures to achieve this.

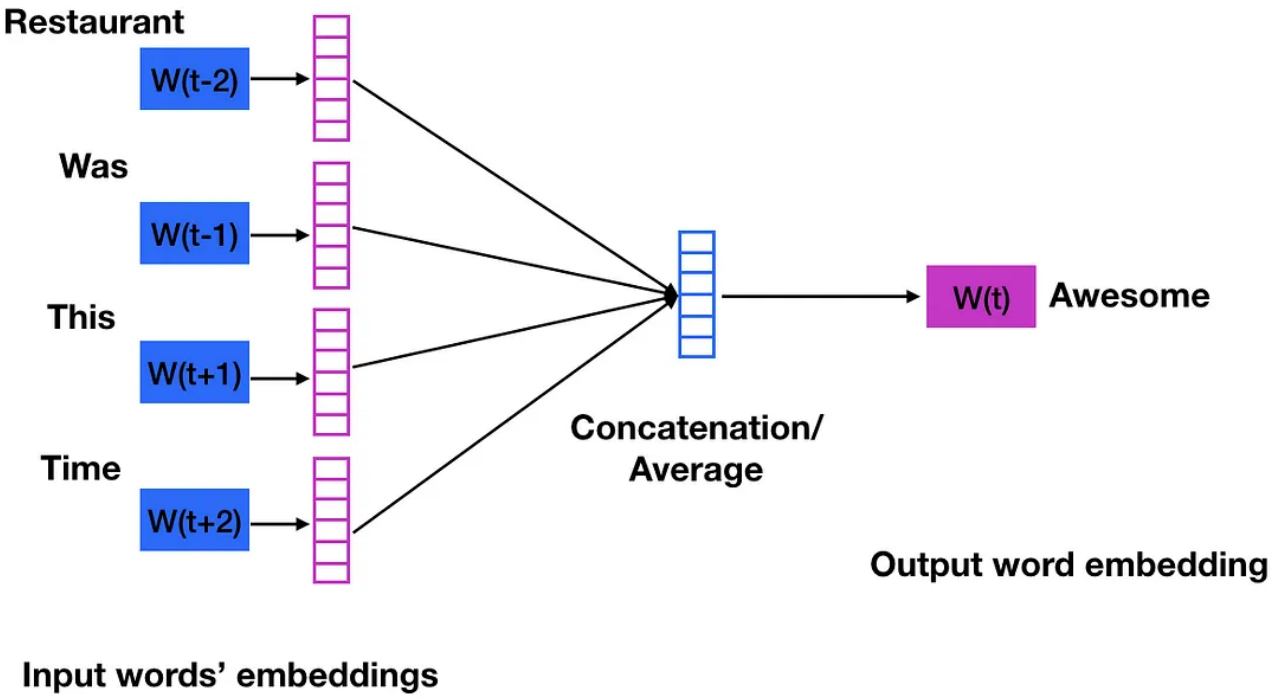
- CBOW
- Skip Gram



Word vectors mapped into space for few words by Dipika Baad

CBOW (Continuous bag of words)

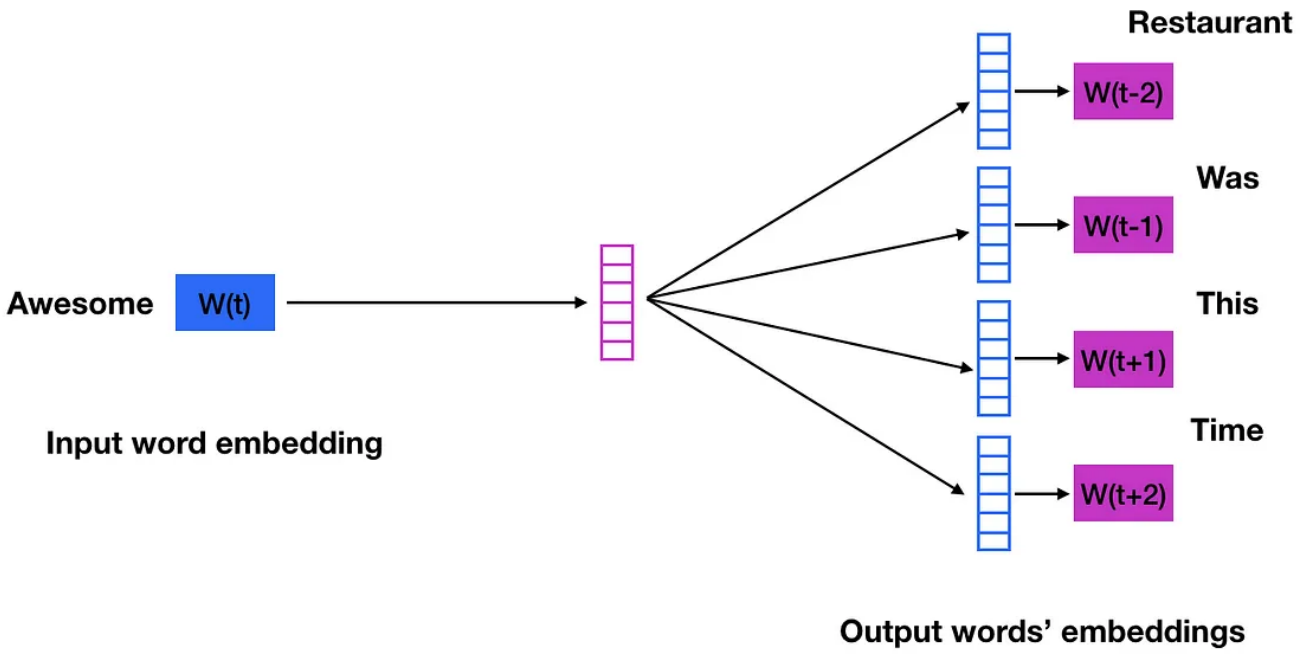
Here the model predicts the word under consideration given context words within specific window. The hidden layer has the number of dimensions in which the current word needs to be represented at the output layer. Following diagram shows as example with window of size 2 for predicting vector for word ‘awesome’ given a sentence ‘Restaurant was awesome this time’.



CBOW diagram by Dipika Baad

Skip Gram

Skip gram is opposite of CBOW where it predicts embeddings for the surrounding context words in the specific window given a current word. The input layer contains the current word and the output layer contains the context words. The hidden layer contains the number of dimensions in which we want to represent current word present at the input layer. Following shows an example with window size of 2.



SG diagram by Dipika Baad

In this case, we will be using gensim's Word2Vec for creating the model. Some of the important parameters are as follows:

Params -

- **size**: The number of dimensions of the embeddings and the default is 100.
- **window**: The maximum distance between a target word and words around the target word. The default window is 5.
- **min_count**: The minimum count of words to consider when training the model; words with occurrence less than this count will be ignored. The default for min_count is 5.
- **workers**: The number of partitions during training and the default workers is 3.
- **sg**: The training algorithm, either CBOW(0) or skip gram(1). The default training algorithm is CBOW.

```python
from gensim.models import Word2Vec
import time
# Skip-gram model (sg = 1)
size = 1000
window = 3
min_count = 1
workers = 3
sg = 1

word2vec_model_file = OUTPUT_FOLDER + 'word2vec_' + str(size) + '.model'
start_time = time.time()
stemmed_tokens = pd.Series(top_data_df_small['stemmed_tokens']).values
# Train the Word2Vec Model
w2v_model = Word2Vec(stemmed_tokens, min_count = min_count, size = size, workers = workers,
print("Time taken to train word2vec model: " + str(time.time() - start_time))
w2v_model.save(word2vec_model_file)
```

train_word2vec.py hosted with ❤ by **GitHub** **view raw**

Output:

```
Time taken to train word2vec model: 107.16288304328918
/usr/local/lib/python3.6/dist-packages/smart_open/smart_open_lib.py:402: UserWarning: This function is deprecated, use smart_open.open instead.
  'See the migration notes for details: %s' % _MIGRATION_NOTES_URL
```

It is usually better to save the model in some file so you don't have to rerun it every time doing the training for the classifier. In the next part, we will reload the model and see how to access the word2vec dictionary as well.

```python
import numpy as np
# Load the model from the model file
sg_w2v_model = Word2Vec.load(word2vec_model_file)
# Unique ID of the word
print("Index of the word 'action':")
print(sg_w2v_model.wv.vocab["action"].index)
# Total number of the words
print(len(sg_w2v_model.wv.vocab))
# Print the size of the word2vec vector for one word
print("Length of the vector generated for a word")
print(len(sg_w2v_model['action']))
# Get the mean for the vectors for an example review
print("Print the length after taking average of all word vectors in a sentence:")
print(np.mean([sg_w2v_model[token] for token in top_data_df_small['stemmed_tokens'][0]], axi
```

load_word2vec_model.py hosted with ❤ by GitHub view raw

Output:

```
/usr/local/lib/python3.6/dist-packages/smart_open/smart_open_lib.py:402: UserWarning: This function is deprecated, use smart_open.open instead.
  'See the migration notes for details: %s' % _MIGRATION_NOTES_URL
Index of the word 'action':
2154
30056
Length of the vector generated for a word
1000
Print the length after taking average of all word vectors in a sentence:
1000
```

Next will see how to use the Word2Vec model to get the vector for documents in the dataset.

Generating Word2Vec Vectors

Word2Vec vectors are generated for each review in train data by traversing through the `X_train` dataset. By simply using the model on each word of the review, we get the word embedding vectors for those words. We will be implementing average over all the vectors of words in a sentence and that will represent a sentence from our dataset. These vectors are stored in a `csv` file. You can directly create this in a dataframe but when there is a large amount of data it is better to write to a file as and when the vector is created and if the code breaks you can start from the point where it had broken. Following code, writes the vectors in the `OUTPUT_FOLDER` defined in the first step.

```
# Store the vectors for train data in following file
word2vec_filename = OUTPUT_FOLDER + 'train_review_word2vec.csv'
with open(word2vec_filename, 'w+') as word2vec_file:
    for index, row in X_train.iterrows():
        model_vector = (np.mean([sg_w2v_model[token] for token in row['stemmed_tokens']], ax
        if index == 0:
            header = ",".join(str(ele) for ele in range(1000))
            word2vec_file.write(header)
            word2vec_file.write("\n")
        # Check if the line exists else it is vector of zeros
        if type(model_vector) is list:
            line1 = ",".join( [str(vector_element) for vector_element in model_vector] )
        else:
            line1 = ",".join([str(0) for i in range(1000)])
        word2vec_file.write(line1)
        word2vec_file.write('\n')
```

generate_word2vec_vectors_for_traindata.py hosted with ❤ by GitHub view raw

Training Sentiment Classification Model using Word2Vec Vectors

Once the Word2Vec vectors are ready for training, we load it in dataframe. `DecisionTreeClassifier` is used here to do the sentiment classification. Decision tree classifier is Supervised Machine learning algorithm for classification. In this example, scikit-learn package is used for implementing the decision tree classifier class. The `fit` function is used to fit the input feature vectors against the sentiments in train data. Following code shows how to train the classifier with Word2Vec vectors.

```
import time
#Import the DecisionTreeeClassifier
from sklearn.tree import DecisionTreeClassifier
# Load from the filename
word2vec_df = pd.read_csv(word2vec_filename)
#Initialize the model
clf_decision_word2vec = DecisionTreeClassifier()

start_time = time.time()
# Fit the model
clf_decision_word2vec.fit(word2vec_df, Y_train['sentiment'])
print("Time taken to fit the model with word2vec vectors: " + str(time.time() - start_time))
```

train_classifier_with_w2v_vectors.py hosted with ❤ by GitHub view raw

Output:

```
Time taken to fit the model with word2vec vectors: 36.71265363693237
```

This took ~36 seconds to train for our input data. `clf_decision_word2vec` variable can be now used to do the predictions.

Testing the Model

Time to see how the model worked out at the end of this all facade.

```
from sklearn.metrics import classification_report
test_features_word2vec = []
for index, row in X_test.iterrows():
    model_vector = np.mean([sg_w2v_model[token] for token in row['stemmed_tokens']], axis=0)
    if type(model_vector) is list:
        test_features_word2vec.append(model_vector)
    else:
        test_features_word2vec.append(np.array([0 for i in range(1000)]))
test_predictions_word2vec = clf_decision_word2vec.predict(test_features_word2vec)
print(classification_report(Y_test['sentiment'],test_predictions_word2vec))
```

test_word2vec_for_classification.py hosted with ❤ by GitHub view raw

Output:

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: DeprecationWarning: Call to deprecated `__getitem__`
  This is separate from the ipykernel package so we can avoid doing imports until
              precision    recall  f1-score   support

          -1       0.56      0.55      0.55      2992
           0       0.46      0.47      0.46      3044
           1       0.55      0.56      0.55      2964

    accuracy                           0.52      9000
   macro avg       0.52      0.52      0.52      9000
weighted avg       0.52      0.52      0.52      9000
```

Classification Report shows the average accuracy which is `0.52` . This is a good result compared to the amount of data used for training. The `predict` function can be used on the model object to get the predicted class for the test data. Accuracy for positive and negative sentiments is better than neutral which makes sense as it is hard to distinguish the neutral comments compared to commonly used words in the positive and negative sentiment.

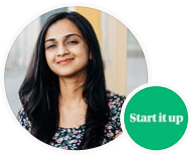
This accuracy is little bit less compared to BOW Classification and TFIDF

Classification done in previous posts. One thing to notice is that the total input dimension has reduced from vocab size of `30056` to `1000` in case of Word2Vec. That is here the dimension can be made custom of less size but as it is capturing the necessary things and only limited things to describe the words it is a good compromise between accuracy and computational complexity for the classification model.

So now you can easily experiment with your own dataset with this method! I hope this helped you to understand how to use Word2Vec vectors to do the sentiment analysis on restaurant reviews data. Feel free to extend this code! This is applicable to any other text classification problems where multiple classes are there. If I can think about improving this model, I would use different hyper-parameters for decision classifier or even try out other classification models. Input parameters like `size`, `min_count` and `window_size` of word2vec function can be experimented to get a better accuracy than this. Instead of using average, min and max of word vectors of words in a sentence can be taken as well. Preprocessing can be changed to use lemmatization or other stemming algorithms to see how the results change.

As always — Happy experimenting and learning :)

Machine Learning | NLP | Sentiment Classification | Text Classification | Restaurant Review



Written by Dipika Baad

304 Followers · Writer for The Startup