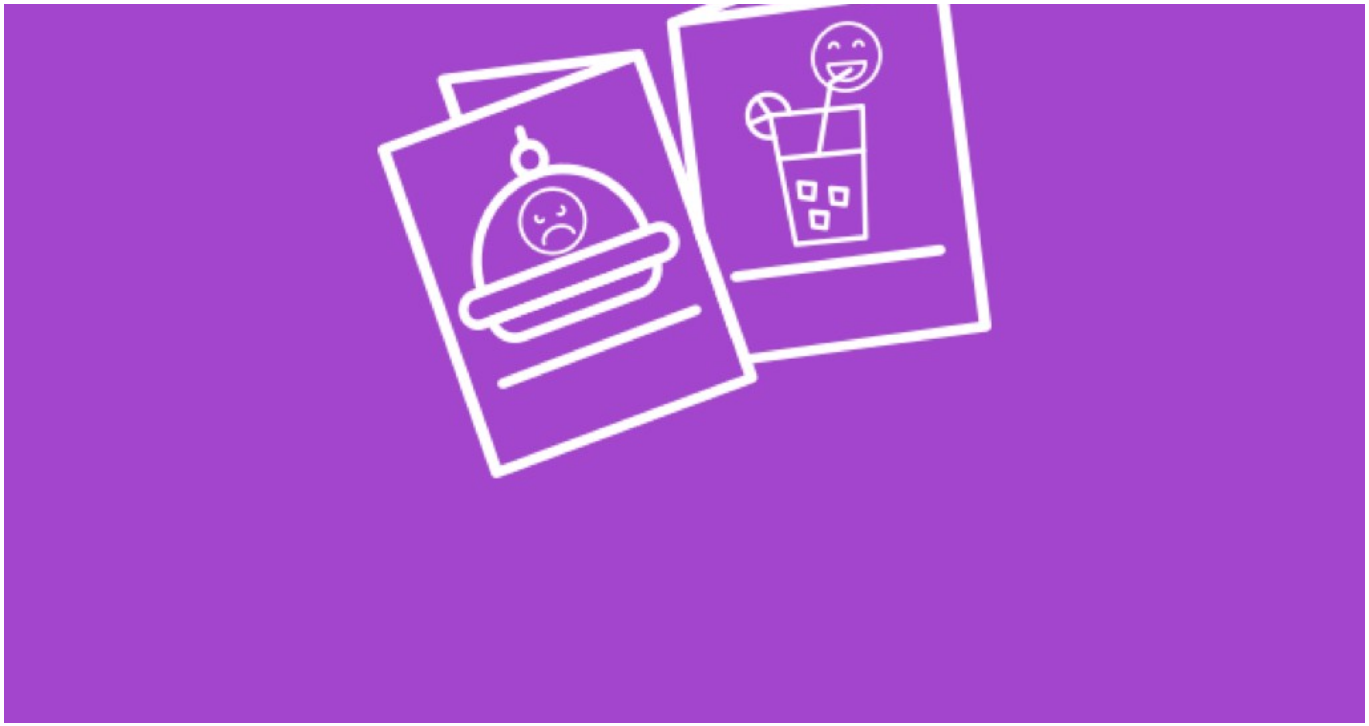
This is your **last** free member-only story this month. Sign up for Medium and get an extra one

Sentiment Classification for Reviews using Doc2Vec

Dipika Baad Follow

Mar 9 · 11 min read ★

Sentiment Classification for Yelp Restaurant Review Text Data using Doc2Vec vectors for reviews

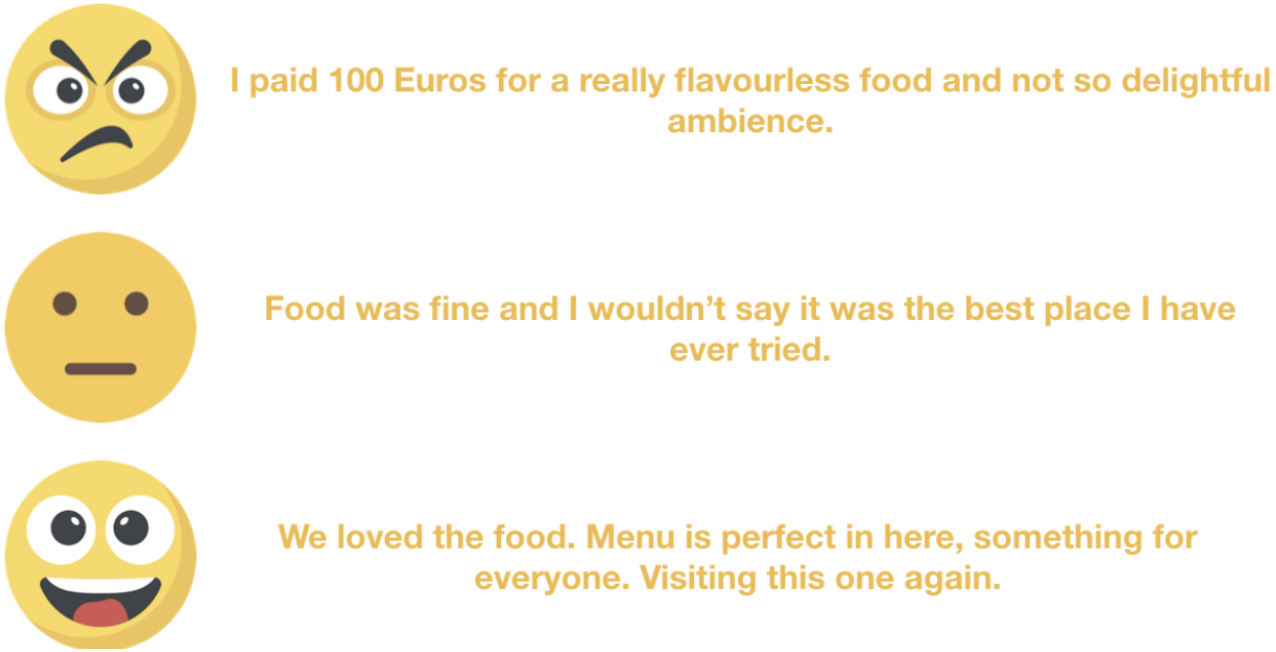


Sentiment Classification For Reviews Using Doc2Vec by Dipika Baad

In this post, sentiment classification on text data will be done by using Doc2Vec to get the document vectors which will be used as input to the classification model. This method can be used for any other multi-class text classification problems as well. Doc2Vec is based on Word2Vec which was used to do sentiment classification in my previous post. Doc2Vec instead of just getting the word vectors which correlates words with surrounding words, it adds another vector for document id. This is more tuned towards getting a document vector, hence this is more preferred way of getting

document vectors instead of Word2Vec vectors averaged on document level as explained in previous.

In my previous posts of Sentiment Classification using BOW, Sentiment Classication using TFIDF and Sentiment Classification using Word2Vec 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 by loading the data first.



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:

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.

Output:

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

Output:

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.

Output:

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:

Output:

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:

Output:

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:

Output:

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.

Output:

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 Doc2Vec Model

The main objective of Doc2Vec is to convert the sentence (or paragraph) into a vector. It is generalisation of the Word2Vec model. You can use the averaged Word2Vec vectors of each words to get the sentence vector as explained in the previous post. Doc2Vec generates an efficient and high-quality distributed vectors that is it captures the precise syntactic and semantic word relationships (ref).

The way of training involves adding another vector called paragraph id vector in addition to the surrounding vector so specifically adding an extra parameter for identifying the doc. Along with training the word vectors it learns the document vector. This extension of CBOW is called Distributed Memory version of Paragraph Vector **(PV-DM)**. Another algorithm which is an extension of Skip-gram is called Distributed Bag of Words version of Paragraph Vector **(PV-DBOW)**. In this input is the paragraph id and the output layer is the surrounding word vectors. This is actually faster and uses less memory. PV-DM has been used quite often as it gives state-of-the-art results.

Let's see in short how those two algorithms work for getting the document vector.

1. PV-DM

This is similar to CBOW in Word2Vec (explained in my Word2Vec post) where it tries to predict the embedding for target word based on it's context word but here it adds paragraph id as the input along with rest of surrounding words as per window size mentioned. In the following diagram, example of sentence "Restaurant was awesome this time" is taken where target word is "awesome".

PV-DM diagram by Dipika Baad

2. PV-DBOW

This is similar to Skip Gram(SG) of Word2Vec model (explained in my Word2Vec post). In SG, target word is used as input and context words are predicted in the neural network. Here, instead of the target word, paragraph id as the input is used. Example sentence in the document will be used same as above.

PV-DBOW diagram by Dipika Baad

Here, Gensim's Doc2Vec is used to create the vectors. Corresponding important parameters are explained below. PV-DM is chosen for training this with vector size of 1000.

Params -

- **documents**: Iterable list of TaggedDocument or can be simply a list of elements. Prefer the iterable for large datasets.
- **dm**: Decides which algorithm out of PV-DM and PV-DBOW. By default dm=1 (PV-DM) and if dm=0 then PV-DBOW.

- **vector_size**: Dimensionality of the feature vectors.
- **window**: The maximum distance between the current and predicted word within a sentence.
- **min_count**: Ignores all words with total frequency lower than this.

Output:

Part of the output (TaggedDocument is huge)

Next will see how to use the Doc2Vec model to get the vector for documents in the dataset.

Generate Doc2Vec Vectors

Doc2Vec vectors are generated for each review in train data by traversing through the `X_train` dataset. By inferring over the stemmed tokens using Doc2Vec model trained above, vectors are generated for each review. 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.

Training Sentiment Classification Model using Doc2Vec Vectors

Once the Doc2Vec 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 Doc2Vec vectors.

Output:

Testing the Model

Time to see how the model turned out to be by testing it on the test data.

Output:

Classification Report shows the average accuracy which is `0.46`. This is a good enough 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 less compared to BOW Classification, TFIDF Classification and Word2Vec Classification done in previous posts. Even though Doc2Vec is go to option for getting the document vector, in this case we found that

just averaging Word2Vec vectors had given us better results with same vector size & window size compared to using Doc2Vec where extra parameter of paragraph id is used and easier to use compared to Word2Vec. I experimented with multiple options to check if the accuracy improves but it didn't come close to Word2Vec accuracy. It is good to experiment these two methods to see which one gives better results. One of the reasons could be that reviews differ from each other a lot so adding parameter of document id is not helping much for the model. Relations between words near by is representing the sentiment class more than considering their whole relationship to complete document. Results for multiple combinations of window and vector sizes is given below.

Accuracies for different combinations of parameters for Doc2Vec by Dipika Baad

As it can be seen, increasing the dimension even close to the vocab size didn't give a result as good as it was obtained with Word2Vec. Total vocab size was `30056` and with Word2Vec or Doc2Vec one can reduce the dimension and obtain a closer accuracy compared to BOW and TFIDF. 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 documents it is a good compromise between accuracy and computational complexity for the classification model. But for this review dataset, Word2Vec worked better.

So now you can easily experiment with your own dataset with this method! I hope this helped you to understand how to use Doc2Vec 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 `vector_size`, `min_count` and `window` of doc2vec function can be experimented to get a better accuracy than this. Preprocessing can be changed to use lemmatization or other stemming algorithms to see how the results change.

Have fun experimenting with NLP problems :)

NLP Sentiment Classification Machine Learning Restaurant Review Text Classification

Learn more.

Medium is an open platform where 170 million readers come to find insightful and dynamic thinking. Here, expert and undiscovered voices alike dive into the heart of any topic and bring new ideas to the surface. Learn more

Make Medium yours.

Follow the writers, publications, and topics that matter to you, and you'll see them on your homepage and in your inbox. Explore

Share your thinking.

If you have a story to tell, knowledge to share, or a perspective to offer — welcome home. It's easy and free to post your thinking on any topic. Write on Medium

Medium

About Help Legal