Movie Reviews Sentiment Analysis

Statistics 199: Directed Research in Statistics

Kaixin Wang

kaixinwang@ucla.edu

Advisor: Miles Chen

Project Report
Adopted from the book ***Applied Text Analysis with Python*** [1]

Department of Statistics
University of California, Los Angeles
Spring 2020

[1] Bengfort et al. (2018)

Contents

1 Introduction 3

2 Corpus 3

3 Methods 3

3.1 Import the corpus . . . 3

3.2 Document Tokenization . . . 3

3.3 Adding the Part-Of-Speech Tagging . . . 5

3.4 Creating the Training and Testing Sets . . . 5

3.5 `TfidfVectorizer` Method . . . 5

3.6 `KeyphraseExtractor` Class . . . 6

3.7 Logistic Regression Classifier . . . 6

3.8 Linear Support Vector Machine Classifier . . . 7

3.9 Keras Classifier . . . 7

4 Results and Analysis 8

5 Future Steps 9

A Corpus 9

B Code 9

C Textbook 9

List of Figures

1 Flowchart of moview reviews sentiment analysis . . . 3

2 Flowchart of creating the frequency distributions . . . 4

3 Frequency distribution of words in the training set . . . 4

4 Frequency distribution of words in the testing set . . . 5

5 Pipelines of using LogisticClassifier and SGDClassifier . . . 6

6 Confusion matrix of using LogisticRegression classifier . . . 6

7 Confusion matrix of using SGDClassifier . . . 7

8 Pipelines of using KerasClassifier . . . 7

9 Confusion matrix of using KerasClassifier (text normalization) . . . 8

10 Confusion matrix of using KerasClassifier (keyphrase) . . . 8

List of Tables

1 Classification accuracy on the testing set . . . 8

1 Introduction

Sentiment analysis is one of the main topics in Natural Language Processing (NLP). In this project, we will classify the sentiment of movie reviews that are scrapped from the IMDB website. The reviews will be marked as either positive (`pos`) or negative (`neg`). We will build several classifiers, including machine learning classification methods such as the logistic regression classifier and the support vector machine classifier, as well as deep learning neural network such as the keras classifier.

We will introduce the corpus next, and start to extract the features and build the classifiers.

2 Corpus

The corpus that we will be using is the *Large Movie Review Dataset* collected by Stanford University (Maas et al., 2011).

This is a dataset for binary sentiment classification containing substantially more data than previous benchmark datasets. There is a set of around 25,000 highly polar movie reviews for training, and 20,000 for testing, and each piece of review is tagged as either positive (`pos`) or negative (`neg`):

- There are 20268 documents in the training set, where 7768 (38.3%) are tagged as `pos` and 12500 (61.7%) are `neg`.
- There are 20001 documents in the testing set, where 7501 (37.5%) are tagged as `pos` and 12500 (62.5%) are `neg`.

3 Methods

The overall sentiment analysis procedure is shown in the flowchart in Figure 1:

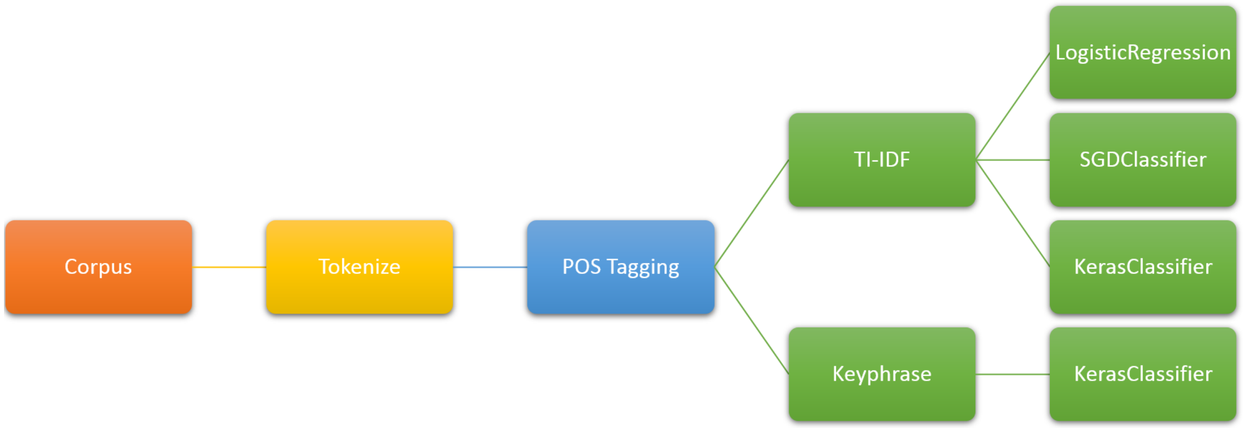

Figure 1: Flowchart of moview reviews sentiment analysis

3.1 Import the corpus

Since the documents are grouped into two categories, `pos` and `neg`, we will use the built-in corpus reader, `CategorizedPlaintextCorpusReader`, to import the corpus. To do this, we need to specify the `root` path of the corpus, a `DOC_PATTERN` to specify the file name of each document in the corpus, as well as a `CAT_PATTERN` that specifies the format of the category that each document belongs to.

After reading in the corpus, we can access properties of the corpus such as the `fileid` and `category` by calling its `fileids()` and `categories()` functions.

3.2 Document Tokenization

After reading in the corpus, we will now tokenize each document into individual words. To do this, we first define a `TextNormalizer` class, and use it in the function `tokenize`, which will check if the word (1) contains any punctuation marks and (2) is a stop word. If the word has no punctuation and is not a stop word, the normalizer will convert the word into all lower cases and output the filtered word in a list.

We can visualize the frequency distribution of words in the training set and the testing set by (1) re-combining the individual words into sentences using the `getDocument` function first, (2) getting the frequency distribution via using `CountVectorizer`, and (3) plotting the distribution using `FreqDistVisualizer`. The procedure is shown in Figure 2:

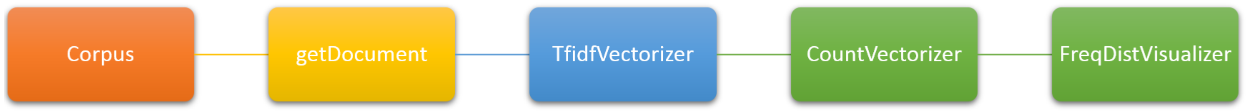

Figure 2: Flowchart of creating the frequency distributions

The frequency distributions are shown in Figure 3 and Figure 4.

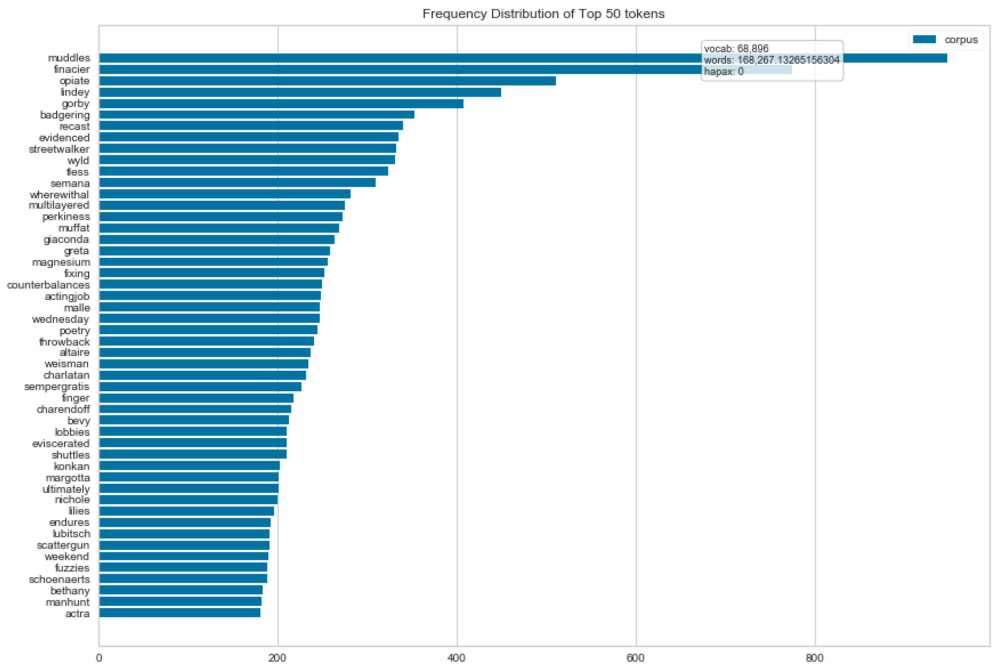

Figure 3: Frequency distribution of words in the training set

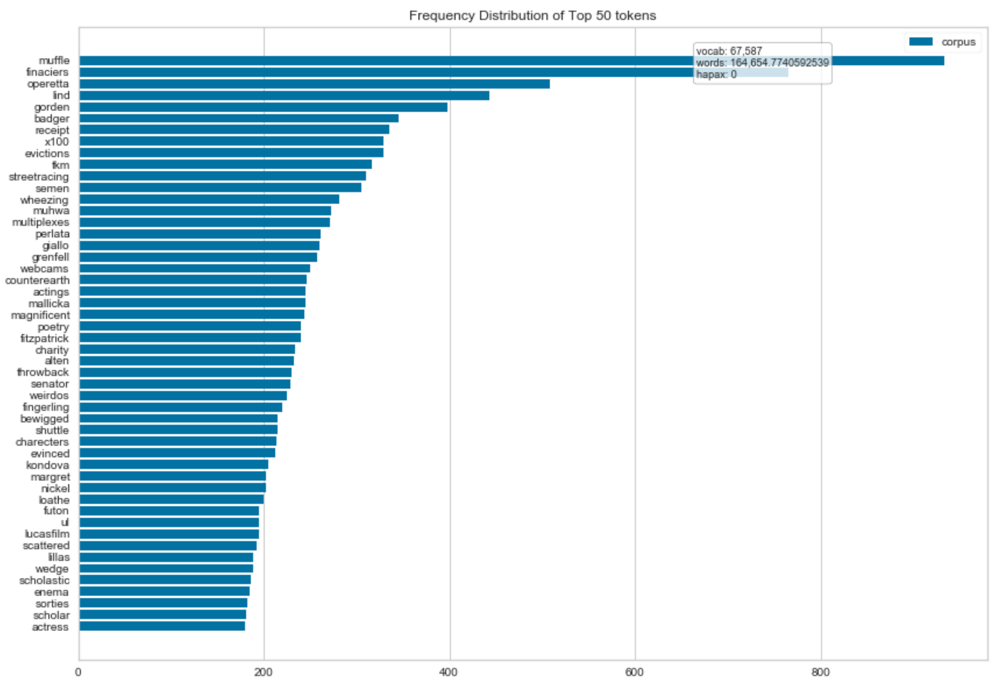

Figure 4: Frequency distribution of words in the testing set

3.3 Adding the Part-Of-Speech Tagging

After breaking the documents into individual words, we can now add the Part-Of-Speech (POS) tags to each word. We will achieve this by defining a `postag` function, which iterates through each document in the corpus, apply the `tokenize` function on each word and then add the POS tag to them. The output of `postag` will be a list of list of list of tuples in the form of `(word, POS)`.

3.4 Creating the Training and Testing Sets

After defining the functions `postag` and `tokenize`, we will now create the training and testing set. We can obtain the features by applying the `postag` function to the training corpus and testing corpus, respectively, to obtain `X_train` and `X_test` sets, which are the transformed (numeric) features that machine learning methods can understand. As for the testing set, we will define a `CorpusLoader` class to extract the label of each document. By creating a `CorpusLoader` object for both the training corpus and the testing corpus, we can obtain `y_train` and `y_test`, which are the target variables in the training and testing sets.

Since we have obtained the all four pieces of information that are usually required by machine learning methods, we can start to build different types of classifiers.

3.5 `TfidfVectorizer` Method

TF-IDF is computed on a per-term basis, such that the relevance of a token to a document is measured by the scaled frequency of the appearance of the term in the document, normalized by the inverse of the scaled frequency of the term in the entire corpus.

1. The *term frequency of a term given a document*, `tf(t, d)`, can be the boolean frequency (as in one-hot encoding, 1 if t occurs in d 0 otherwise), or the count. However, generally both the term frequency and inverse document frequency are scaled logarithmically to prevent bias of longer documents or terms that appear much more frequently relative to other terms: `tf(t, d) = 1 + log ft`,d.

2. Similarly, *the inverse document frequency of a term given the set of documents* can be logarithmically scaled as follows: `idf(t, D) = log 1 + N nt`, where N is the number of documents and nt is the number of occurrences of the term t in all documents. TF-IDF is then computed completely as `tfidf(t, d, D) = tf(t, d)` $\times$ `idf(t, D)`.

Note that:

(1) Generally both the term frequency and inverse document frequency are scaled logarithmically to prevent bias of longer documents or terms that appear much more frequently relative to other terms.
(2) Because the ratio of the `idf log` function is greater or equal to 1, the TF-IDF score is always greater than or equal to zero:

- the closer the TF-IDF score of a term is to 1, the more informative that term is to that document;
- the closer the score is to zero, the less informative that term is.

We will be using the `TfidfVectorizer` method to convert the raw documents to a matrix of TF-IDF features.

3.6 `KeyphraseExtractor` Class

Instead of normalizing each word in the document, we can also choose to extract the keyphrases in the document. We will define a `KeyphraseExtractor` class and use its `extract_candidate_phrases` method to extract the keyphrases.

3.7 Logistic Regression Classifier

The first type of classifier that we will explore is the logistic regression classifier. To build the classifier, we will create a `Pipeline` consisting of the `TextNormalizer`, `TfidfVectorizer`, and finally the `LogisticRegression` classifier. We will also apply dimension reduction to the features by adding the `TruncatedSVD` to reduce the number of features to 1000. After creating the pipeline object, we will fit the model with the training set, `X_train` and `y_train`, which will fit each object in the pipeline with the training set data. Then we can predict the output of the testing set by calling the `predict` function on the pipeline object.

The structure of the pipeline is shown in Figure 5.

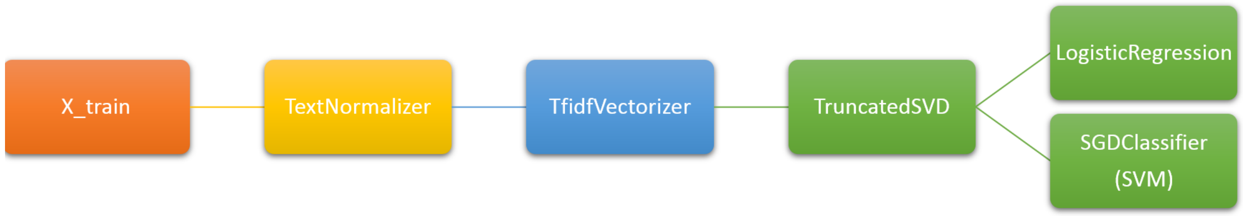

Figure 5: Pipelines of using LogisticClassifier and SGDClassifier

To visualize the performance of the logistic regression classifier, we can create a confusion matrix for predicting the testing set, as shown in Figure 6.

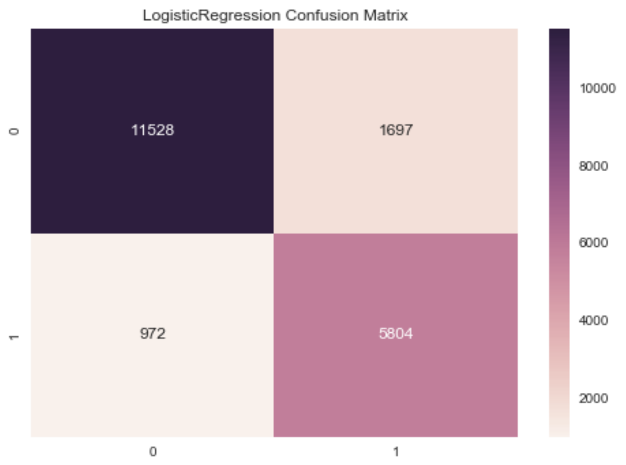

Figure 6: Confusion matrix of using LogisticRegression classifier

3.8 Linear Support Vector Machine Classifier

The second type of classifier that we will explore is the stochastic gradient descent (SGD) learning classifier. `SGDClassifier` is a collection of linear classifiers (SVM, logistic regression, a.o.) with SGD training. By default, it fits a linear support vector machine (SVM).

To build the linear SVM classifier, similar to building the logistics regression classifier, we will create a `Pipeline` consisting of the `TextNormalizer`, `TfidfVectorizer`, and finally the `LogisticRegression` classifier. We will also apply dimension reduction to the features by adding the `TruncatedSVD` to reduce the number of features to 1000. After creating the pipeline object, we will fit the model with the training set, `X_train` and `y_train`, which will fit each object in the pipeline with the training set data. Then we can predict the output of the testing set by calling the `predict` function on the pipeline object.

The structure of the pipeline is shown in Figure 5.

To visualize the performance of the linear SVM classifier, we can create a confusion matrix for predicting the testing set, as shown in Figure 7.

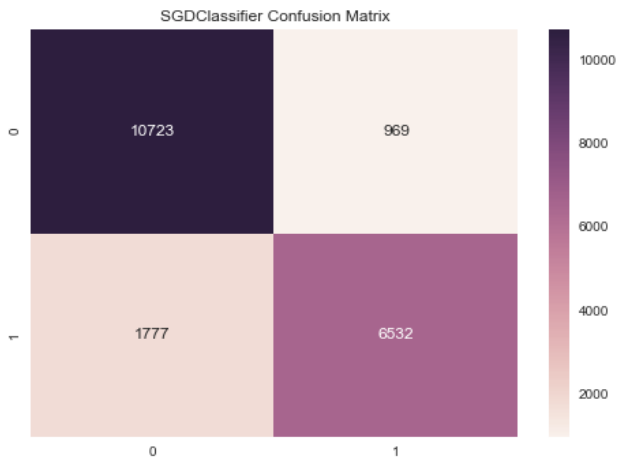

Figure 7: Confusion matrix of using SGDClassifier

3.9 Keras Classifier

The third type of classifier that we will implement is the `KerasClassifier`, which is one of the most basic deep learning classifiers in the Keras module.

We will transform the features in two different approaches, where one uses the `TextNormalizer` class to normalize the features, while the other one uses the `KeyphraseExtractor` class to extract the features. The difference between two approaches is shown in Figure 8.

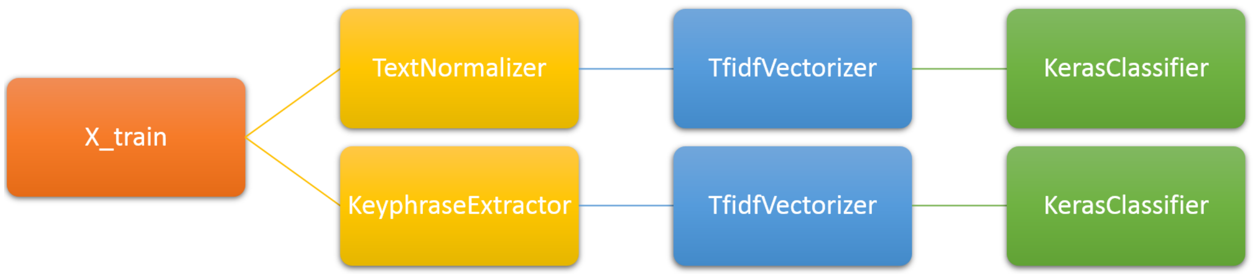

Figure 8: Pipelines of using KerasClassifier

To visualize the performance of the keras classifiers, we can create a confusion matrix for predicting the testing set for each model, as shown in Figure 9 and Figure 10.

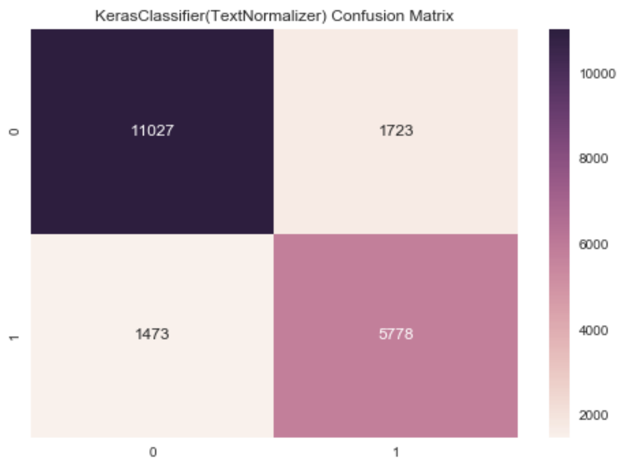

Figure 9: Confusion matrix of using KerasClassifier (text normalization)

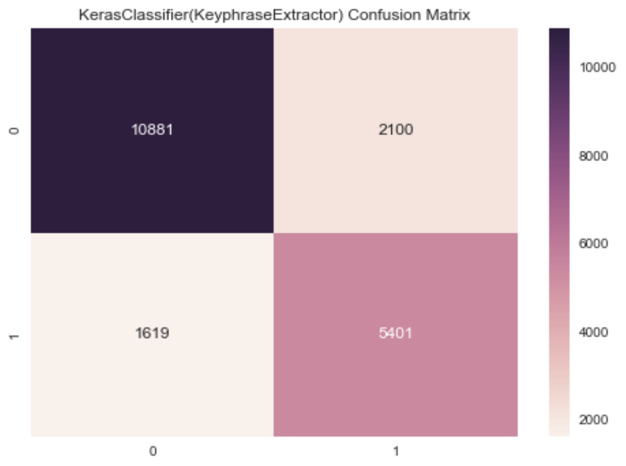

Figure 10: Confusion matrix of using KerasClassifier (keyphrase)

4 Results and Analysis

Table 1: Classification accuracy on the testing set

Classifier	**Logistic Regression**	**Linear SVM**	**Keras Classifier (TF-IDF)**	**Keras Classifier (Keyphrase)**
Accuracy	**0.8666**	**0.8627**	**0.8402**	**0.8141**

From Table 4, we observe that:

- `LogisticRegression`: most accurate classifier among all four methods, but it takes relatively long to fit the model
- `SGDClassifier`: also has a relatively high classification accuracy, and it also takes quite a bit to fit the model
- `KerasClassifier`: it takes very small amount of time to build the classifier pipelines, although the overall performance on the testing set is not as high; one possible issue is overfitting on the training set

5 Future Steps

We saw from the analysis that the logistic regression and support vector machine classifiers, which are traditional machine learning methods, had relatively better classification performances than the deep learning neural networks. One possible reason for this is that the ANNs are overfitting the training set. So we can further explore how to balance the trade-offs between model complexity and model generality.

Another possible improvement is on the speed of executing the entire program. In the current version, the step of transforming the raw text into numeric features so that machine learning can understand takes quite a long time, and the process of building the classification pipelines also takes time. To speed up the computations, we can scale the text analytics through using multiprocessing or Spark, two methods that are frequently used to distribute a large workload to multiple computational resources (e.g., processes, disks, memory).

A Corpus

The corpus used in this project is available at http://ai.stanford.edu/~amaas/data/sentiment/.

B Code

The Jupyter Notebook of this project can be found at https://nbviewer.jupyter.org/github/kaixin-wang/MovieReviewsNLP/blob/master/Notebook/STATS199-NLP-Code-KaixinWang.ipynb.

C Textbook

The textbook *Applied Text Analysis with Python* (Bengfort et al., 2018) can be accessed at https://proquest.safaribooksonline.com/book/programming/python/9781491963036.

References

Bengfort, B., Bilbro, R., and Ojeda, T. (2018). *Applied Text Analysis with Python*. O'Reilly Media, Inc.

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. (2011). Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA. Association for Computational Linguistics.