Exploration of Sentiment Analysis Techniques in Python for Classifying Film Reviews

Aim: Apply the skills gained from 'Text Mining' and 'Machine Learning' to implement a sentiment prediction algorithm that trains on and classifies movie reviews as `pos` or `neg`.
Tagged Release: Challenge-3-Choi.

Description of my solution: MultinomialNB

Much of my submitted implementation is inspired by the functions we have implemented as exercise in our 'Text Mining' and 'Machine Learning' notebooks.

Initially, I check if the required nltk packages, corpora, and packages are already installed in the machine that is running the program using the command `nltk.find(...)`. If a `LookupError` is raised (indicating that the requirement is not met), the necessary dependencies are downloaded. The reason for the initial check is to prevent and remove unnecessary side-effects from my implementation. These nltk dependencies include:

- wordnet – A package that provides the functionalities and algorithms that are necessary in stemming and lemmatization
- movie_reviews – A corpora which contains a large set of movie review text data. The entirety of the movie reviews available in this corpora will serve as our training data (as the testing data is provided separately)
- stopwords – This corpus provides the list of common, neutral words which we filter out from our data (to increase efficiency and optimize space/time complexity of our implementation).
- punkt – This module allows us to use the `nltk.word_tokenize(...)` function which is nltk's recommended word tokenizer that is crucial for the Tokenization step.

After the dependency requirements have all been met, I load the training dataset as a pandas Dataframe by opening the `.txt` files from nltk's movie_reviews directory. This operation takes linear time as each element (i.e. a movie review) is visited once (O(n)).

Then, I define the x variable (i.e. `x_training: Series`) by applying a helper function `clean_text(text)` to each cell under the column 'text' of our training dataset. `clean_text(text)` tokenizes the movie review (`: str -> lst(str)`), filters for stopwords, and lemmatizes each of the word using our selected Lemmatizer WordNetLemmatizer(), and finally joins each of the lemmatized strings with white space delimiters. The WordNetLemmatizer() was selected out of a variety of stemming/lemmatization algorithms as it maximized the accuracy of our program. With all other parameters set constant (MultinomialNB), the following stemming/lemmatization algorithms performed as follows [algorithm name : accuracy] :

- SnowballStemmer : 0.784
- LancasterStemmer : 0.796
- **WordNetLemmatizer : 0.801**
- PorterStemmer : 0.798

Afterwards, I use a TfidfVectorizer() to transform the long string of (cleaned, tokenized, filtered, lemmatized) words to transform the series into a csr_matrix. Upon research, I learned that a sparse data structure like csr_matrix is the golden standard for storing data about counts, and especially text word-counts. I learned that using a sparse data structure like csr_matrix makes the process of using some classification algorithms (e.g. GaussianNB) less stream-lined as they may require dense data structures (and hence, some work around/transformation). Nevertheless, it seems that for the majority of use cases (i.e. in Keras and scikit-learn), sparse data structures are seamlessly integrated.

With the training data all prepared, I instantiate a MultinomialNB classifier and fit the vectorized x matrix and the y variable (series of 'pos'/'neg') to it. I selected the multinomial Naive Bayes classifier instead of other alternatives as it is most suitable for classification with discrete features such as word counts for text classification. It is also convenient that fractional counts such as tf-idf also work with MultinomialNB. I have empirically tested the performance of this classifier against its counterparts, and it has proven its great fit for the given task.

With the training of the model out of the way, we can know implement the relatively simple predict_sentiment function, which takes a movie review string as an argument. The function cleans the text using the aforementioned clean_text function then vectorizes the text using our TfidfVectorizer(). Then, we use `.predict()` to use the trained classifier/-model to predict whether the review is positive or negative. Finally, we return `res[0]` to output the first (and only) element of the output list of `.predict()`.

Solutions Attempted

`MultinomialNB.` A supervised, Naïve-Bayes learning algorithm that I used for my final implementation of my algorithm. [**tester outcome (avg.): (8.5473, 1.7353, 0.801)**]

`ComplementNB.` Another Bayesian supervised learning algorithm that was as accurate and efficient as `MultinomialNB`. Implementing this method is as simple as replacing all occurances of MultinomialNB in the script with ComplementNB. Its strikingly similar performance to MultinomialNB is to be expected as this classifier was designed with the purpose of replacing MultinomialNB while correcting the "severe assumptions" made by the standard Multinomial Naive Bayes classifier. Learning about this, I considered whether I should use ComplementNB instead. However, my research showed that the golden standard for reliable, accurate text classification remains to be MultinomialNB. [**tester outcome (avg.): (8.5713, 1.8322, 0.801)**]

`BernoulliNB.` A Bayesian supervised learning algorithm that was significantly less accurate and less efficient than its aforementioned counterparts. This method can be implemented the same way as ComplementNB. The poor performance is expected as BernoulliNB is designed to handle binary/boolean features rather than occurrence counts. [**tester outcome (avg.): (8.7568, 2.2431, 0.588)**]

`Lexicon-based analysis.` I have also attempted to use the nltk-provided lexicon (e.g. vader) to convert the string-type data into a float-type data. Then, this dense data structure (very different compared to the sparse CSR Matrix used in the NB approaches above) was used by sklearn machine learning algorithms to train a classifier model. Unfortunately, the performance of this method was poor in comparison to MultinomialNB. To implement this approach, I had to load the lexicon (as a dictionary) for fast access. A sample code is shown below for demonstration of that process (and the algorithm that computes the sentiment for each of the movie reviews using the lexicon). [**tester outcome (avg.): (8.2893, 3.4342, 0.562)**]

```
def load_lexicon(path):
    with open(path) as f:
        text=f.read()
    resDict = {'TOKEN':[], 'MEAN-SENTIMENT-RATING':[], 'STANDARD
        DEVIATION':[], 'RAW-HUMAN-SENTIMENT-RATINGS':[]}
    for line in text.split("\n"):
        splitLine=line.split("\t")
        resDict['TOKEN'].append(splitLine[0])
        resDict['MEAN-SENTIMENT-RATING'].append(float(splitLine[1]))
        resDict['STANDARD DEVIATION'].append(float(splitLine[2]))
        resDict['RAW-HUMAN-SENTIMENT-RATINGS'].append(splitLine[3])
    return pd.DataFrame(resDict)

def use_lexicon(review, lexicon):
    lexiconDict = lexicon[['TOKEN', 'MEAN-SENTIMENT-RATING'
        ]].set_index('TOKEN').to_dict()['MEAN-SENTIMENT-RATING']
    res = 0
    wList = nltk.word_tokenize(review)
    for w in wList:
        res += lexiconDict.get(w, 0)
    return res
```

Performance Analysis

Approach	Import Time	Prediction Time	Accuracy
MultinomialNB	8.5473	1.7353	0.801
ComplementNB	8.5713	1.8322	0.801
BernoulliNB	8.7568	2.2431	0.588
Lexicon-based	8.2893	3.4342	0.562

As is demonstrated above, our chosen implementation empirically performs the best out of the four solutions that I have implemented in terms of accuracy and speed. This makes sense as MultinomialNB is designed for the exact purpose of text classification and sentiment analysis from text-based data. Therefore, although I am unaware of how the sklearn algorithm works under-the-hood, it is reasonable to assume that MultinomialNB in particular is optimized for this type of operation and is therefore faster and more accurate.

With regards to the functions and implementations that I have coded in this project, I made sure to be as efficient as possible. All the code I have written are max O(n) (linear time) and I have made conscious efforts to keep the constant to a minimum through several modifications. For instance, my first implementation conducted tokenization, filtering, and lemmatization in three separate steps – each requiring the whole dataset to be traversed row-by-row. Seeing how this could slow down the performance of my algorithm, I redesigned the three processes to occur during the same traversal. Furthermore, I also removed/reduced unnecessary variable assignments as memory access can be time consuming (though the cost can be minimized through caching).

I have tested my implementation of predict_sentiment with my own set of tests to see how it performs with certain texts. I was quite satisfied with the results although occasional some results were surprising (e.g. clearly negative sentence being classified as positive, certain punctuations being associated with pos/neg, etc...).

One surprising phenomenon I observed while trying to optimize the performance of my algorithm is that filtering out some words which I thought would be critical words to the classifier actually improved the accuracy of the classifier. For instance, I filtered out ‘terribl’ and ‘bad’ from every review (by adding them to the stopWords set) and the accuracy of our MultinomialNB increased from 0.801 to 0.803. Though it is a marginal increase that may not be statistically significant, I was surprised to observe that the accuracy is not severely undermined. Perhaps it is because of the insufficient sample size or the testing sample, however this observation made me wonder about the astounding potential for machine learning to see trends and patterns that are often invisible to the human eye and mind.