Natural Language Processing (NLP) Project

Author: Argha Sarkar (1221352) a.sarkar@warwick.ac.uk

For the NLP project, the first task is to pre-process the corpus of the news articles. The directory path and the file name might need to be edited in the first and the second code cells to load up the corpus and the list of positive and negative words. The file paths are validated before anything else happens. Additionally, there has been a code block added which ensures the **nltk** path has been updated for Joshua and DCS machines to avoid the need to download the required corpuses or other files.

For handling the corpus and add some helpful functions, I've created a corpus object. The corpus class loads the converts the *json lines* into separate json documents. It then adds them to a list called **json_docs**. Using a pointer to keep track of the current object, its possible to iterate over the corpus processing each document step by step. This can also be done from any given index and not just from the start. Furthermore, the corpus class also offloads the hassle of dealing with the storage of the pre-processed news articles.

Pre-processing

The pre-processing of text takes place here. The following tasks are performed:

- All the text is converted to lower case.
- All URLs are removed from the text using the following regex: `\b(http)(s)?:\/\/((([\w])(\.)?)+)\b`

  The regex looks for boundaries at the start and the end. After that, it looks for "http" at the start with an optional "s". After that the semi-colon and two slashes are used. After that, any number (one or greater) of words and full-stops will be matched.

- All numbers are removed which are not concatenated with any non-numeric characters. EG: 5 is removed but not 5pm.
- All non-alphanumeric characters expect for spaces are removed.
- All words of length below 4 characters are removed. This has been implemented by removing all words of length between one and three characters. `\b[A-Za-z0-9]{1}\b`

  All boundary alphanumeric characters which are one character in length will be removed.

Using the **WordNetLemmatizer** from NLTK's stem module, all the words are lemmatized using the default (noun) Part Of Speech tagging. This aims to reduce the vocabulary and make the words more consistent.

Part B: N-grams

The pre-processing of the data has been performed.

For the first part of this task, the vocab size and the number of tokens needs to be calculated. For this, a class has been created which uses a **dictionary** with the words as the key and their counts as the values. The **number of keys** will be the **vocab size** and the **token** will be the **total sum of the word counts**.

The **TokenCounter** class uses a dictionary to keep the count of how many times a word has occurred in the corpus so far. If the word is already in the dictionary, it's value is incremented by one. If the word is appearing for the first time, then it's added to the dictionary and it's value is set to 1.

The class **VocabAndTokens** class iterates through the pre-processed corpus. For each of the pre-processed documents, the text is split into words. After this, each of the word is added to the **TokenCounter** class. After calculating the size of the vocabulary and the number of tokens, it's printed out.

Part B.1) N Print out here is the number of tokens and V is the vocabulary size.¶

Using the get_V() and get_N() methods of the VocabsAndToken counter class, get the vocab size and the total number of tokens. In this case, **V: 102166 N: 5775947.**

Trigrams calculator: NGramHolder

The **NGramHodler** the class uses a dictionary to store the most common trigrams and provide helper functionalities. For example, it's able to take a list of words as an input and generate a list of n_grams for it. The "n" depends on the class instantiation. Furthermore, it's also able to generate the top n_grams in a descending order of count. Given an n_gram, it's able to return how many times it's occured so far in the text processed. Also, it's able to search n_grams which begin with a certain phrase. For example, searching for **"is this"** using the get_n_grams_with_phrase(phrase) will return all n_grams which begins with "is this" for example: **"is this the"**.

Top 25 trigrams

For the pre-processing that took place earlier, punctuation and all words with fewer than 2 characters were removed. As a result, the trigrams listed below consists of only words which are two or more characters long.

```
[('one of the', 2439), ('on shares of', 2093), ('day moving average', 1
972), ('on the stock', 1567), ('as well as', 1427), ('in research repor
t', 1417), ('in research note', 1375), ('the year old', 1255), ('the un
ited states', 1225), ('for the quarter', 1221), ('average price of', 11
93), ('research report on', 1177), ('research note on', 1138), ('the en
d of', 1134), ('in report on', 1124), ('earnings per share', 1123), ('s
hares of the', 1081), ('buy rating to', 1075), ('cell phone plan', 1073
), ('phone plan details', 1070), ('according to the', 1068), ('of the c
ompany', 1039), ('appeared first on', 995), ('moving average price', 99
5), ('price target on', 968)]
```

Part B.3) Calculating the list of positive and negative words in the corpus

Use the class **WordCount** to read the file with the positive and negative words. After reading the file, the words are stripped of any trailing spaces or new line characters using **rstrip**. I had to do this as I found there were trailing **\n** characters to these words.

After the words have been added to the dictionary, the **add_word** method can be used for increasing the count for a positive or negative word in the dictionary if it already exists.

Alternatively, **add_sentence** can be used for processing all the words in the whole sentence at the same time.

Iterate through all the documents and adds each document to the positive and negative word counter.

Calculates the positive and negative words. After that, it prints out the counts of the positive and negative words in the corpus and draws a bar graph to show the difference. The result is given below:

```
Positive word count: 171508
Negative word count: 125916
```

Part B.4: Calculating the number of new stories with more positive than negative words

A class **ArticleSentimentAnalyser** has been used to calculate the news stories' positive and negative words.

Firstly, the class takes the file paths of the two files containing the positive and negative words. After that, the two list of words are loaded into **sets**. (Sets are used instead of list to increase performance). Additionally, there are two counters for positive and negative articles. Using the **corpus** class's iterator, all the articles are processed.

For each of the article, the number of positive and negative words are calculated. If the number of positive words in the article is greater than the number of the negative words then the positive article counter is incremented. Similarly, if the number of negative words in the article is greater than the number of postive words, then the negative counter is incremented. If the number of positive and negative words are equal, nothing happens.

After the processing takes places, the number of positive and negative methods can be accessed via `analyser.get_positive_article_count()` and `analyser.get_negative_article_count()`

Before running the analyser, the pointer of the corpus is reset to 0. This ensures that all the articles are processed.

The **ArticleSentimentAnalyser** class is instantiated with the positive, negative file paths as well as the corpus being passed in as arguments.

```
Positive article count: 11044
Negative article count: 6255
```

Part C: Language models

To do this, the trigrams, bigrams and unigrams were generated from the first 16000 rows of the corpus.

Part C.2: Sentence generation

For generating the sentence, I am taking the last two words and searching all trigrams where those two words are the first two words.

For example, if there is a phrase **"is this"**, all trigrams like **"is this the"** are returned. Getting the count of the appearance of **is this the** and dividing it by the count of the bi-gram **is this**. All the possible trigrams are divided by the count of the bi-gram to get the probabilities. The trigram for which the highest probability occurs is used. The last word in that trigram is taken and that's the next word that's added to the sentence that's being generated.

P("the" | "is this") = P("is this the") / P("is this")

P("the" | "is this") = count(tri_grams("is this the")) / count(bi_grams("is this"))

In the above example, the trigrams of all different words are used instead of "the". The one with the highest probability is appended to the current sentence. The generated sentence is:

is this the company has market capitalization of billion and

Part C.2: Evaluation and Perplexity

After building the language model on the first 16000 articles, it's time to see how well the models performs. This can be done by calculating the perplexity.

For calculating the perplexity, the articles from 16001 to the end will be used. The perplexity will be calculated on the entire document.

Laplace smoothing

In order to smooth out the results for unseen ngrams in the training corpus, **Laplace smoothing** has been used. This means that 1 has been added to the numerator and the value of **V** (the vocabulary size) has been added to the denominator. It's not usually the best solution for language models due to the huge number of 0's that's replaced. However, in this case it has been used as it's very easy to implement.

Calculating Perplexity

1) To calculate the perplexity, generate bigram for the article.

2) For each of the bigrams in the article, get the count for that bigram from the language model generated in part c.1. and add 1.

3) For the first word in the bigram, get it's count from the unigram of the language model and add the value of the vocab size. This has been done according to the "add-1" estimate.

4) Calculate the probability and store it in a list of probabilities

5) iterate through the list of probabilities and divide 1 by the probabilities

6) Iterate over this new list and multiply all the probabilities together to get the perplexity.

After the perplexities for all the test articles have been calculated, they've been stored in a list called **perplexities**.

For some of the perplexities, the perplexity shows up as infinite.