Student Name: Niketan Eshwar Moon
Student Id: a1790186
Course: Applied Natural Language Processing (Semester 1/ 2022)

ANLP Assignment 2
Building a Text Matching algorithm for question retrieval

Importing Libraries

I have imported the libraries that were required for data preprocessing, text preprocessing stage. Some of them are as follows:
re - used to find match between pattern for removing punctuation and other functionalities.
Pandas and numpy - used for reading data and doing some calculations
nltk - used for tokenization and removing normal english words called as stopwords. Need to download stopwords. Also used this package for applying lemmatization technique to the text.

Importing the Dataset

The dataset file used was “data.tsv” which is a tab-separated file. So I used pd.read_csv function with sep = “\t” for reading the file. It may also happen that for bad lines it may give errors to just get warnings on such rows and skipping them I used **on_bad_lines = “warn”.** I have also stored file data in another variable so that when text preprocessing is applied, the file data is clean to get original questions.

Data Exploration

I used data.head() and data.info() to get insight into the data. The dataset consists of 363192 rows and 6 columns.
id - considering this as a docId
qid1- question1 id
qid2- question2 id
question1 - the full question1 consider this as input queries where only is_duplicate is 1
question2 - Full question2. Consider this as data on which the input queries will be matched
is_duplicate - Tells us the ground truth for question1. For is_duplicate = 1 then question2 is a match for input query question1.

Data Preprocessing

- As the dataset consisted of some null values, I first dropped the rows with null values using **dropna()** function.

- It was also observed that there were possibilities that question2 has duplicates and for easing the computing load, dropped the rows with duplicates using drop_duplicates([“question2”])

Text Preprocessing

I have used the same preprocessing techniques that were used in Assingment 1 for sentiment analysis.

Data Cleaning -

In data cleaning, I removed punctuations, numbers, any HTML tags or URLs. This matching was done using regular expression. Later the whole text was converted into lowercase.

Tokenization and stopwords removal

I have used tokenization to convert the text into tokens i.e. text is divided into words separated by a space. Later I removed all the occurrences of stopwords in these tokens. For this, I used stopwords taken from nltk.corpus.

Lemmatization

Converted words to their base root using WordNetLemmatizer. For example the root word for “hoping” is hope. So lemmatization applies methods to convert any word into its root form.

Creating an Inverted File Mapping of the words

I created a class MapInvertFile. When the object is instantiated, the constructor first initialises variables as follows
self.N - Keeps track of total number of documents
self.invertFile - Converts the words into inverted file data structure.
{
word: (docId, tfidf score)
}
self.numberOfWordInDoc - Keeps track of count of words for each document. Stored in this format
{(doc, word) : count}
self.wordInDocuments - Tells us about the number of documents the word existed in.
self.tfWord - which stores the tfidf score. Mapping is done like this
{(doc, word): tfidfscore}

I created a getInvertedFile function to get us the words converted into inverted file structure. In this method, I first calculated the number of documents. Then I took

each entry from the dataset.question2 feature and converted into list. Now I used **getCountTermInDocument()** function to store the **self.numberOfWordInDoc** and **self.wordInDocuments.** The logic was to loop through the questions list and convert each sentence into words and then calculate the above parameters.

Calculating TFIDF Score

Once this is calculated, I calculated tf for each word in document. For calculating TF score I applied formula

TF = log(1 + count(t, d))

where count(t, d) is the count of words in each document and TF is the score for each word in a specific document.

Then in setInvertedFile, I calculated the IDF score for each word in document using the formula

$$IDF = \frac{N \text{ i.e total Number of documents}}{\text{Number of documents in which the word existed}}$$

Once the IDF score is calculated I calculated the TFIDF score by first taking the TF score from self.tfWord and then multiplying with IDF score that we just calculated. Later, I stored the score inside tfWord

This is how the TFIDF score was calculated and stored.

Mapping into inverted file

Once the tfidf score is calculated, I stored the word and (doc,tfidf) as key value pair which is simply the data structure of Inverted File. The logic was to loop through all the keys inside self.tfWord and then we calculate tfidf score and store the word as key and (doc, tfidfscore) as value in self.invertedFile.

For getting the inverted file mapping for all the entries in question2, we just have to instantiate the class and then call **instance.getInvertedFile(data)** which will return the inverted file mapping.

Text Matching for Inverted File

For text matching, the input queries are 100 such that is_duplicated is 1.0. I used dataH to take 100 such entries. Now I applied all the Text Preprocessing techniques that was done above when we did text processing for data.

For calculating the rank of the entries, I created a class TextMaching, which has two functions **getTop5TextMatching** and **getTop2TextMatching.**

The logic is to first take the input query, then split it into words and for each word look into the inverted file mapping and calculate the total tfidf score for each docId. So if a word appears in doc1 with tfidf score of 0.05 and doc2 with tfidf score of 0.02 then

S(doc1) = 0.05, S(doc2) = 0.02

Now if another word occurs in doc2 with tfidf score of 0.04 then we will first look into the hashmap if we have doc2 inside it and then add its score to the current score. So now the score ranking for documents become

S(doc1) = 0.05 and S(doc2) = 0.06
Therefore, doc2 gets a higher ranking than doc1. So **getTop5TextMatching gives us the docId for top 5 ranks for each input query**
Similarly, getTop2TextMatching gives the docId for top 2 ranks for each query.
This is how text retrieval using Inverted File Structure with tfidf score is done.
After calculating top 5 matches for each input query, I uploaded each input query with top 5 matched questions into a file called **invertedFileTextMatchingOutput.txt**

Performance Evaluation Top 5 vs Top 2

I calculated the probability for questions if the docId matches the docId inside top matches.

$$Probability = \frac{correct\ value\ inside\ top\ matches}{total\ number\ of\ entries}$$

	Probability
Top 5 matches	0.37 or 37%
Top 2 matches	0.23 or 23%

Search Question Interface

Created one function searchQuestionMatch which takes an input question from the user and then applied text preprocessing techniques on the given input.
Later, each word inside the input question is checked with inverted File Mapping that we created before. If the word exists, we calculate the document rank as explained above in the text-matching section.
Later, the function returns top 5 docIds that matched the data.
Then I just extracted question2 with these docIds.

Sentence Embedding using average

Pretrained Word Embeddings

For sentence Embedding, I used pre-trained word embeddings from the glove. I used 300-dimensional word embeddings. I first downloaded the glove file then I read the file. Each line has the format word “vectors separated by space”. So I just split each line and my word is the first index element and the others are its vectors. Then mapped the word with its vectors. This way we got the pre-trained word embeddings.

Generating Sentence Embeddings

For this I created a class **GenerateAverageSentenceEmbedding.**
The **generateEmbeddings** method first goes into each sentences and then split them to make into words and for each word we do sum of the vectors. Once all the sum for each word in that sentence is done and we get

self.**sumWordEmbeddingVector.** Then for taking the average we just divide the total sum of vectors by the length of words in that sentence. This way we generate average sentence embedding for each sentence. Then we store the sentence as key and its average vectors as values in self.averageSentenceEmbedding in format
{
sentence : average vectors
}

1. Generating data embeddings for our data in question2. I took all question2 entries and converted it into a list and each sentence inside the entry generated sentence embeddings. For generating embeddings I called **generateEmbeddings** method described above.
2. Generated sentence embeddings for our input query data. I took all the question1 entries inside dataH which is only 100 entries and then I calculated its embeddings again calling the generateEmbeddings function.
3. Calculating Cosine Similarity: For this I used **getCosineSimilarity** function which returns the cosine similarity calculated by formula

$$\text{cosine}(\vec{v},\vec{w}) = \frac{\vec{v}\cdot\vec{w}}{|\vec{v}||\vec{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}}$$

4. Then I went through each embedding inside input query embeddings and compared it with all the embeddings inside data embeddings or question2 sentence embeddings. Later I calculated top5 matches and top2 matches by simply getting the top 5 values and top 2 values for cosine similarity respectively.
5. Then I took all the top 5 values for each input query and store it in file **“sentenceEmbeddingOutput.txt”**

Performance Evaluation Top 5 vs Top 2

Calculated the probability of top 5 matches and top 2 matches by the formula

$$Probability = \frac{correct\ value\ inside\ top\ matches}{total\ number\ of\ entries}$$

	Probability
Top 5	0.28 or 28%
Top 2	0.12 or 12%

Search Question Average Sentence Embedding

For this I created method **searchQuestionSentenceEmbedding** which takes in question and applies text preprocessing on it.
Later, we calculate sentence embeddings for the input question with **generateEmbeddings** method described above.
Once we get the embeddings we do the cosine similarity input query with the sentence embeddings that we calculated before i.e question2 sentence embeddings.
Then the function returns the top 5 questions matched.

Sentence Embedding using Downweight frequent words

- Used the same pre-trained embeddings
- Generate Sentence Embedding using down weight technique
  Here I created a class called **GenerateDownWeightSentenceEmbedding.**
  Here while taking sum of vectors with the pre-trained vectors we are first multiplying the vector with $\frac{a}{a + p(w)}$
  where "a" is a parameter of value 10^{-4}. Now p(w) is the frequency of word w in total words. So, to calculate p(w) for each word. I created a function called probabilityW() which goes through each word in the sentence and increments the count of the word. Later I loop through the probWMap created, that has word count for now, and store the p(w) in probWMap with w as key and v as value.
  Now once we get probability of word w for each word, then we will multiply $\frac{a}{a + p(w)}$ with the vector, and later after calculating for each word in a sentence we will store the average of the down weight vectors in self.downWeightSentenceEmbedding in format
  {sentence: down weight vectors average }
  This is how sentence embedding with down weight vectors is calculated.
  For top 5 text we will do the following things

1. First we will calculate sentence embeddings for data in question2
2. Later we calculate embeddings for input query for data in question1 in dataH
3. Later we do the cosine similarity between question1 and question2 and find out the top 5 max values according to cosine similarity.
4. Once we get the index for the cosine similarity we get the top 5 matches. We store the top 5 matches for each query in **downWeightOutput.txt** file.

Performance Evaluation

We use the same matrix as used in the above 2 methods. We find out the probability of top 2 and top 5 matches.

	Probability
Top 5	0.19 or 19%
Top 2	0.04 or 4%

Search Question

A input question is asked from the user and we apply the text preprocessing techniques discussed above on this. Later, we generate embeddings for the question and compare them with downWeightData Embeddings that we calculated for the sentences in question2 by cosine similarity. The function at last return top 5 matches questions.

a. Comparing the TFIDF based and sentence embedding based sentence matching system

	TFIDF	**Average Sentence Embedding**	**DownWeight Average Sentence Embedding**
Probability Top 5	0.37 or 37%	0.28 or 28%	0.19 or 19%
Probability Top 2	0.23 or 23%	0.12 or 12%	0.04 or 4%
Time for creating Inverted File	3.21 seconds	No inverted file technique applied	No inverted file technique applied
Time for creating sentence embeddings	No sentence embedding applied	16.1 seconds	17.7 seconds
Time for text matching	Overall 4 seconds including inverted file structuring and getting top 5 text matching	Around 12 mins (Cosine Similarity) 11 min 57 seconds to be exact	Around 12 mins (Cosine Similarity) 11 min 55 seconds to be exact
Total Time of execution	3.8 seconds	12 minutes 18 seconds	12 minutes 28 seconds

- The above table compares the parameters between TFIDF and Sentence based Embedding. From the above table, we can say that the time complexity for getting text matching in TFIDF is more than 150 times faster as compared to sentence embedding.

- The probability of getting the correct top 5 is 37% for TFIDF as compared to sentence embedding where it is only 28%.
- The probability of getting the correct top 2 is 23% for TFIDF as compared to sentence embedding where it is only 12%.

b. Comparing different ways of creating sentence embedding

- From the above table we can say that the average sentence embedding performs better than down weight frequent word sentence embedding
- The time complexity for average sentence embedding is smaller than the time complexity for down weight sentence embedding. Down weight frequent word sentence embedding takes a little(approx 10 seconds) more to compute than average sentence embedding.
- The probabilities for top 5 and top 2 performance is better for average sentence embedding than down weight frequent words sentence embedding.
- Time for calculating cosine similarity is the same as the size of both the vectors are same but just the values for vectors in Average Sentence Embedding are different than the values for vectors in Down Weight Frequent Words Sentence Embedding.
- A detailed explanation of how directly averaging word vectors is used and calculated in average sentence embedding and how the down weight frequent word technique is used to calculate the sentence embeddings is discussed above in each of their particular section.