Université d'Ottawa
Faculté de génie

École de science d'informatique et de génie électrique

uOttawa

L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Electrical Engineering and Computer Science

CSI 4107 Information Retrieval Systems

Winter 2023

ASSIGNMENT-2

Implement a neural Information Retrieval system based on the vector space model, for a collection of documents

Course Coordinator: **Diana Inkpen**

Submitted by:

Group -03
Rakshita Mathur St#300215340
Fatimetou Fah St#300101359

Table of Content

Task Division

Experiment-1 code : Fatimetou
Experiment-2 code : Rakshita
Report Writing : Rakshita & Fatimetou

Introduction

Out of the three experiments we decided to perform a modified version of experiment 1 and 2

1. Use your system for Assignment 1 to produce initial results (1000 documents for each query), then re-rank them based on a new similarity score between the query and each selected document. You can produce vectors for the query and each of the selected documents using various versions of sent2vec, doc2vec, BERT, or the universal sentence encoder. You can also use pre-trained word embeddings and assemble them to produce query/document embeddings.

2. Query vector modification or query expansion based on pre-trained word embeddings or other methods. For example, add synonyms to the query if there is similarity with more than one word in the query (or with the whole query vector). You can use pre-trained word embeddings (such as FastText, word2vec, GloVe, and others), preferably some built on a Twitter corpus, to be closer to your collection of documents.

For experiment 1 we are using Doc2Vec whereas for experiment 2 we utilized word2vec to produce vectors for each word. After that we utilized cosine similarity to compare and rank the documents for each query.

Instructions on how to run the program

We used a jupyter notebook for this assignment. To run the program, we need to first install the packages that are imported. These are the instructions to run on the terminal/command line to have them installed:

- pip install re
- pip3 install Collections
- pip install nltk
- pip install tqdm
- pip install bs4

To be able to access the code, you need to download the .ipynb file to your desktop. Then, launch the Jupyter Notebook App. When you do so, the notebook will show all the files on your desktop, choose the notebook you want to access; open the ipynb file and the code will appear and click on the cell “Cell” and select “Run All”.

Experiment 1 - Doc2Vec and Re-Ranking

Introduction to Doc2vec

Doc2Vec is a method based on neural networks that creates representations of text files that are fixed in length. It was developed as an expansion of the word2vec algorithm, a well-liked technique for discovering word embeddings.

Each document is represented as a vector in the Doc2Vec, which learns this representation through an unsupervised training procedure. The algorithm learns to predict a target word or document given the context of adjacent words or documents using a corpus of documents as its input.

Functionality of the program

The Doc2Vec technique is implemented in this code for information retrieval jobs. It tokenizes the text at the document level, removes stopwords, and short tokens (length = 2) as part of the preprocessing procedure for a group of documents. The Doc2Vec model is then trained using these documents, after which it generates tagged documents for it.

The function loops over all queries after training, obtains the inferred vector for each query, calculates the cosine similarity between the query and all documents, and ranks the documents according to the similarity score. The top 1000 documents with the highest similarity are then written to an output file.

Step 1- Preprocessing

Preprocessing is a term used to describe a set of actions carried out on raw text to transform it into a format that machine learning algorithms can use. It is the initial phase of the code and involves preprocessing a collection of documents. The preprocessing in this instance is done expressly for Doc2Vec, an unsupervised learning technique that discovers fixed-length representations of texts.

Step 2- Trains a Doc2Vec model using the preprocessed documents

The code then trains a Doc2Vec model using the preprocessed documents after the documents have been preprocessed. This entails creating vectors from the preprocessed text that represent each document in a fixed-length vector space. The next step is to locate related documents based on a query using the learned Doc2Vec model.

Step 3- Ranking

The algorithm utilizes the Doc2Vec model to rank documents according to similarity scores for each query after it has been trained. It entails calculating the degree of similarity between the vector representations of the query and each document in the collection.

Step 4- Computes average precision for each query using relevance judgments

Using relevance assessments, the code determines the average precision for each query. Humans often provide relevance assessments, which identify the documents that are pertinent to a given query. Higher numbers indicate greater performance. The average precision measures how closely the ranked papers fit the relevant judgements.

Step-5 Map score and P@10

Finally, depending on the average precision scores, the algorithm calculates the Mean Average Precision (MAP) and Precision at 10 (P@10) metrics. P@10 measures the precision of the top 10 ranked documents for each query, whereas MAP assesses the average performance over all queries.

Dependency library used

In [1]:
```python
import os
import re
from gensim.models.doc2vec import TaggedDocument, Doc2Vec
from nltk.tokenize import word_tokenize
from collections import defaultdict
import math
from tqdm import tqdm
from bs4 import BeautifulSoup
import nltk
from sklearn.metrics.pairwise import cosine_similarity
nltk.download('punkt')
nltk.download('stopwords')
from nltk.corpus import stopwords
```

Experiment 2 - Query Expansion

Introduction to Query Expansion

A method for improving the retrieval of pertinent documents from a collection based on a user's search query is query expansion utilizing Word2Vec in information retrieval (IR).

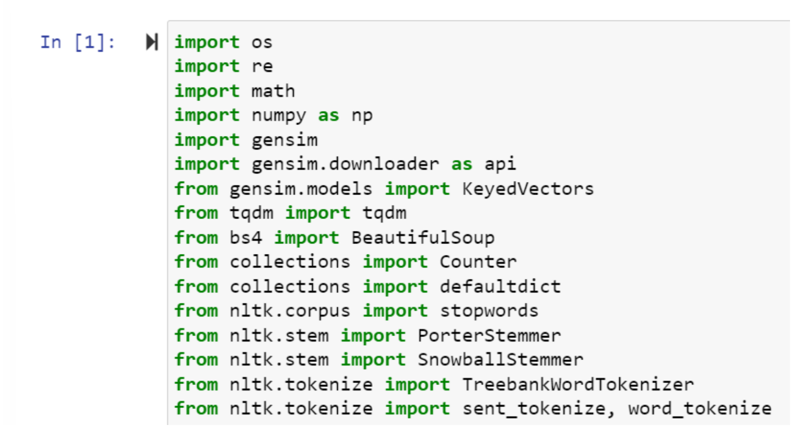
This method builds a semantic representation of the query terms using Word2Vec, a form of neural network that learns word embeddings. Word embeddings, which are created by analyzing a lot of text data, are essentially a numerical representation of a word's meaning.

Similar words or synonyms can be found based on their proximity in the embedding space once the query phrases have been transformed into their respective word embeddings. These related terms can then be added to the first query to broaden it, possibly capturing more pertinent documents that the original query might not have been able to retrieve.

For instance, the Word2Vec model would advise adding "automobile" or "vehicle" to a user's search for "car" in order to get more pertinent search results.

Overall, Word2Vec query expansion can increase an IR system's precision and recall, resulting in better search results for users.

Dependency library used



```
In [1]:  import os
         import re
         import math
         import numpy as np
         import gensim
         import gensim.downloader as api
         from gensim.models import KeyedVectors
         from tqdm import tqdm
         from bs4 import BeautifulSoup
         from collections import Counter
         from collections import defaultdict
         from nltk.corpus import stopwords
         from nltk.stem import PorterStemmer
         from nltk.stem import SnowballStemmer
         from nltk.tokenize import TreebankWordTokenizer
         from nltk.tokenize import sent_tokenize, word_tokenize
```

Step 1- Preprocessing

This programme performs text pre-processing and tokenization. The preprocessing function uses the following steps to carry out the following operations on a text of a document:
1. utilizes a regular expression to get rid of all markup and other non-text elements.
2. tokenizes the text into individual words using the nltk toolkit's word tokenize function.
3. eliminates "the" and "and" as stop words. In order to remove any terms that are in the stop word set, the tokenized words are filtered by the stopwords.words function from the nltk package using the stopwords.words collection of stop words.
4. Checks to see if the token is alphabetical before removing punctuation numbers.

The stem tokens function uses the PorterStemmer from the nltk package to take a list of tokens and return a list of stemmed tokens. After that, the main loop reads each file in the "coll" directory, preprocesses the text, stems the tokens, and saves the stemmed tokens for each file in the documents dictionary, where the file name acts as the key and the list of stemmed tokens acts as the value.

Step-2 Query Expansion

We processed a dictionary of queries in this stage. It initially filters out stopwords, punctuation, and words with a length of 2 or less for each query. After that, the remaining words are subjected to stemming. Then, using word embeddings from the Google News dataset, it widens the query. If the top three most comparable words are not already in the original query, are not stopwords or punctuation, and have stem lengths greater than 2, they are added to the enlarged query for each token in the original query.

The processed and expanded queries are contained in the processed queries dictionary, which has a list of the processed and expanded query words as the value and the query number as the key.

Step-3 Indexing

For this stage, a document dictionary's inverted index is produced. An inverted index is a data structure that associates each distinct term in the documents with a list of papers that also include the term frequently.

The inverted index is initially stored in an empty defaultdict by the code. The process then repeats over all of the documents, which are represented as a dictionary with a document ID as the key and a list of tokens for the words in each document as the value. The function builds a defaultdict to hold the term frequencies for each document and iterates through the document's tokens. It increases the matching term frequency for each token in the defaultdict.

The function adds (doc id, term freq) pairs to the inverted index for each term in the document after calculating the term frequencies for each document. The term is the key and a list of (doc id, term freq) tuples for each document that contains the term is the value of the inverted index, which is kept as a defaultdict of lists.

The function then outputs the list of documents and their term frequencies that contain each phrase, along with the inverted index for each term.

Step-4 Ranking and Retrieval

This code implements an information retrieval system with TF-IDF as its core. The method accepts a set of documents and a set of questions and generates a list of the top 10 documents for each query based on the cosine similarity between the TF-IDF representations of the query and the documents.

Here is a quick summary of what the code accomplishes:

To get the document frequencies (df) for each term, make a list of all the different terms used in each document. Determine the inverse document frequency for each phrase (idf). Calculate the TF-IDF representation for each document.
To determine the cosine similarity between a query and each document, first calculate the TF-IDF representation of each query in a loop. A file called "Results.txt" should be added, listing the top 10 documents for each query in order of cosine similarity.
The TF-IDF representation of a document is used to determine the word importance in that document. The code calculates the TF-IDF representation for each page and each query using the formula below:

(0.5 + 0.5 tfiq)idf = tf-idf

where tf is the term frequency in the document or query and idf is the inverse document frequency for the term. The cosine similarity between two vectors is calculated by dividing the dot product of the two vectors by the product of their norms. The Counter class from the collections module is used in the code to keep track of the frequency of terms in documents and queries. The code adds computes logarithms and square roots in the math module to compute the idf and cosine similarity, respectively.

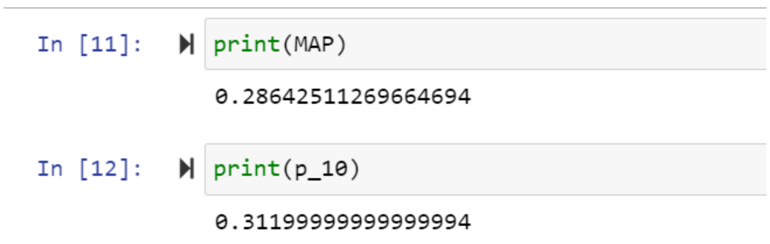
Step-5 Map score and P@10

The average of the average precision values across all queries is known as MAP. An indicator of how relevant the retrieved documents are on average to a query is average precision. It is determined by dividing the total number of relevant documents by the sum of the precision values at all relevant documents. The percentage of the top 10 documents returned that are relevant is called precision at 10.

Result-Comparing the MAP scores

Assignment-1

For Assignment-1 we added the code for the MAP score, the same if what we are using



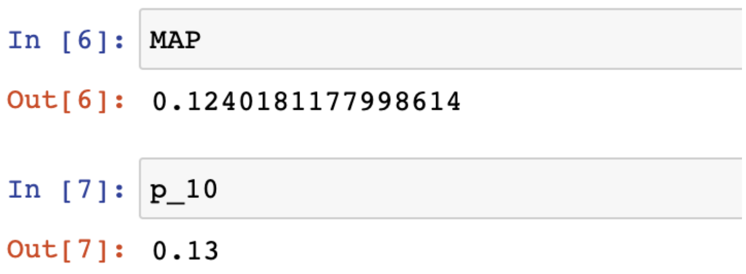
```
In [11]:  print(MAP)

          0.28642511269664694

In [12]:  print(p_10)

          0.31199999999999994
```

Assignment-2 Experiment-1



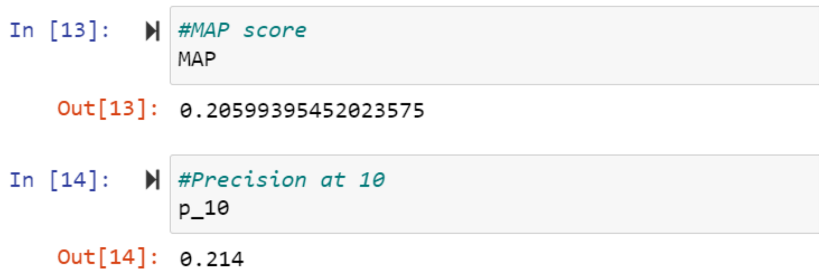
```
In [6]: MAP

Out[6]: 0.1240181177998614

In [7]: p_10

Out[7]: 0.13
```

Assignment-2 Experiment- 2



```
In [13]:  #MAP score
          MAP

Out[13]:  0.20599395452023575

In [14]:  #Precision at 10
          p_10

Out[14]:  0.214
```

Contents of the submitted archive

The archive A2_300215340_300101359.zip contains this file, Assignment-1- score.ipynb which is the code with the output after each steps, test_50.txt which is the test query provided, coll is the collection of documents with 322 files, and Results.txt which is the final file produced after running the last step of the code which has the 50 queries with the top 10 results file. It computes the map score and the p@10 for assignment 1. Assignment-2 Doc2vec Re-Rank.ipynb, contains the code for Doc2vec code as well as the map score and p@10. Finally, Assignment-2 Query Expansion.ipynb, contains the code for Query expansion code as well as the map score and p@10.

Conclusion

Unfortunately, both our experiments couldn't give us a better map score or p@10 compared to our first assignment. We tried our best to get the intended results but nothing would work.

References

- Detecting Document Similarity With Doc2vec: https://towardsdatascience.com/detecting-document-similarity-with-doc2vec-f8289a9a7db7
- Information Retrieval using word2vec based Vector Space Model: https://www.analyticsvidhya.com/blog/2020/08/information-retrieval-using-word2vec-based-vector-space-model/
- Query Expansion for Information Retrieval: https://link.springer.com/referenceworkentry/10.1007/978-0-387-39940-9_947
- **4. Ranked Retrieval with real example** https://www.youtube.com/watch?v=moHvTfZgPGQ