CSE 4334 Programming Assignment 1 (P1)

Spring 2021

Due: 11:59pm Central Time, Saturday, March 5, 2021

In this assignment, you will implement a toy "search engine" in Python. You code will read a corpus and produce TF-IDF vectors for documents in the corpus. Then, given a query string, you code will return the query answer--the document with the highest cosine similarity score for the query. Instead of computing cosine similarity score for each and every document, you will implement a smarter threshold-bounding algorithm which shares the same basic principle as real search engines.

The instructions on this assignment are written in an .ipynb file. You can use the following commands to install the Jupyter notebook viewer. You can use the following commands to install the Jupyter notebook viewer. "pip" is a command for installing Python packages. You are required to use Python 3.5.1 or more recent versions of Python in this project.

```
pip install jupyter

pip install notebook (You might have to use "sudo" if you are i
nstalling them at system level)
```

To run the Jupyter notebook viewer, use the following command:

```
jupyter notebook P1.ipynb
```

The above command will start a webservice at http://localhost:8888/ (http://localhost:8888/) and display the instructions in the '.ipynb' file.

Requirements

- This assignment must be done individually. You must implement the whole assignment by yourself. Academic dishonety will have serious consequences.
- You can discuss topics related to the assignment with your fellow students. But you are not allowed to discuss/share your solution and code.

Dataset

We use a corpus of all the general election presidential debates from 1960 to 2012. We processed the corpus and provided you a .zip file, which includes 30 .txt files. Each of the 30 files contains the transcript of a debate and is named by the date of the debate. The .zip file can

be downloaded from Canvas ("Course Materials" > "Programming Assignment 1" > "Attached Files: presidential_debates.zip").

Programming Language

1. You are required to use Python 3.5.1 or more recent versions of Python. You are required to submit a single .py file of your code.
2. You are expected to use several modules in NLTK--a natural language processing toolkit for Python. NLTK doesn't come with Python by default. You need to install it and "import" it in your .py file. NLTK's website (http://www.nltk.org/index.html (http://www.nltk.org/index.html)) provides a lot of useful information, including a book http://www.nltk.org/book/ (http://www.nltk.org/book/), as well as installation instructions (http://www.nltk.org/install.html) (http://www.nltk.org/install.html)).
3. In programming assignment 1, other than NLTK, you are not allowed to use any other non-standard Python package. However, you are free to use anything from the the Python Standard Library that comes with Python (https://docs.python.org/3/library/) (https://docs.python.org/3/library/)).

Tasks

You code should accomplish the following tasks:

(1) **Read** the 30 .txt files, each of which has the transcript of a presidential debate. The following code does it. Make sure to replace "corpusroot" by your directory where the files are stored. In the example below, "corpusroot" is a sub-folder named "presidential_debates" in the folder containing the python file of the code.

In this assignment we ignore the difference between lower and upper cases. So convert the text to lower case before you do anything else with the text. For a query, also convert it to lower case before you answer the query.

In [3]:
```
import os
corpusroot = './presidential_debates/presidential_debates'
for filename in os.listdir(corpusroot):
    file = open(os.path.join(corpusroot, filename), "r", encoding='UTF-8
    doc = file.read()
    file.close()
    doc = doc.lower()
print(doc)
```

```
october 3, 2000

the first gore-bush presidential debate

moderator: good evening from the clark athletic center at the univers
ity of massachusetts in boston. i'm jim lehrer of the newshour on pb
s, and i welcome you to the first of three 90-minute debates between
the democratic candidate for president, vice president al gore and th
e republican candidate, governor george w. bush of texas. the debates
are sponsored by the commission on presidential debates and they will
be conducted within formats and rules agreed to between the commissio
n and the two campaigns. we'll have the candidates at podiums. no ans
wer to a question can exceed two minutes. rebuttal is limited to one
minute. but as moderator i have the option to follow up and to extend
any particular give and take another three-and-a-half minutes. even t
hen, no single answer can exceed two minutes. the candidates under th
eir rules may not question each other directly. there will be no open
ing statements, but each candidate may have up to two minutes for a c
losing statement. the questions and the subjects were chosen by me al
```

(2) **Tokenize** the content of each file. For this, you need a tokenizer. For example, the following piece of code uses a regular expression tokenizer to return all course numbers in a string. Play with it and edit it. You can change the regular expression and the string to observe different output results.

For tokenizing the Presidential debate speeches, let's all use RegexpTokenizer(r'[a-zA-Z]+'). What tokens will it produce? What limitations does it have?

In [14]:
```
from nltk.tokenize import RegexpTokenizer
tokenizer = RegexpTokenizer(r'[a-zA-Z]+')
tokens = tokenizer.tokenize(doc)
print(tokens)
```

```
['october', 'the', 'first', 'gore', 'bush', 'presidential', 'debate',
'moderator', 'good', 'evening', 'from', 'the', 'clark', 'athletic',
'center', 'at', 'the', 'university', 'of', 'massachusetts', 'in', 'bo
ston', 'i', 'm', 'jim', 'lehrer', 'of', 'the', 'newshour', 'on', 'pb
s', 'and', 'i', 'welcome', 'you', 'to', 'the', 'first', 'of', 'thre
e', 'minute', 'debates', 'between', 'the', 'democratic', 'candidate',
'for', 'president', 'vice', 'president', 'al', 'gore', 'and', 'the',
'republican', 'candidate', 'governor', 'george', 'w', 'bush', 'of',
'texas', 'the', 'debates', 'are', 'sponsored', 'by', 'the', 'commissi
on', 'on', 'presidential', 'debates', 'and', 'they', 'will', 'be', 'c
onducted', 'within', 'formats', 'and', 'rules', 'agreed', 'to', 'betw
een', 'the', 'commission', 'and', 'the', 'two', 'campaigns', 'we', 'l
l', 'have', 'the', 'candidates', 'at', 'podiums', 'no', 'answer', 't
o', 'a', 'question', 'can', 'exceed', 'two', 'minutes', 'rebuttal',
'is', 'limited', 'to', 'one', 'minute', 'but', 'as', 'moderator',
'i', 'have', 'the', 'option', 'to', 'follow', 'up', 'and', 'to', 'ext
end', 'any', 'particular', 'give', 'and', 'take', 'another', 'three',
'and', 'a', 'half', 'minutes', 'even', 'then', 'no', 'single', 'answe
r', 'can', 'exceed', 'two', 'minutes', 'the', 'candidates', 'under',
```

(3) Perform **stopword removal** on the obtained tokens. NLTK already comes with a stopword list, as a corpus in the "NLTK Data" (http://www.nltk.org/nltk_data/) (http://www.nltk.org/nltk_data/)). You need to install this corpus. Follow the instructions at http://www.nltk.org/data.html (http://www.nltk.org/data.html). You can also find the instruction in this book: http://www.nltk.org/book/ch01.html (http://www.nltk.org/book/ch01.html) (Section 1.2 Getting Started with NLTK). Basically, use the following statements in Python interpreter. A pop-up window will appear. Click "Corpora" and choose "stopwords" from the list.

In [15]:
```
import nltk
nltk.download()
```

```
showing info https://raw.githubusercontent.com/nltk/nltk_data/gh-page
s/index.xml (https://raw.githubusercontent.com/nltk/nltk_data/gh-page
s/index.xml)
```

Out[15]:
```
True
```

After the stopword list is downloaded, you will find a file "english" in folder nltk_data/corpora/stopwords, where folder nltk_data is the download directory in the step above. The file contains 127 stopwords. nltk.corpus.stopwords will give you this list of stopwords. Try the following piece of code.

In [16]:
```
from nltk.corpus import stopwords
print(stopwords.words('english'))
print(sorted(stopwords.words('english')))
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you',
"you're", "you've", "you'll", "you'd", 'your', 'yours', 'yourself', 'y
ourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'her
s', 'herself', 'it', "it's", 'its', 'itself', 'they', 'them', 'their',
'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'tha
t', "that'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'b
e', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'di
d', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'a
s', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'again
st', 'between', 'into', 'through', 'during', 'before', 'after', 'abov
e', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'ov
er', 'under', 'again', 'further', 'then', 'once', 'here', 'there', 'wh
en', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'mor
e', 'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'ow
n', 'same', 'so', 'than', 'too', 'very', 's', 't', 'can', 'will', 'jus
t', 'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', 'o',
're', 've', 'y', 'ain', 'aren', "aren't", 'couldn', "couldn't", 'did
n', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't",
'haven', "haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'must
n', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "shoul
dn't", 'wasn', "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn',
"wouldn't"]
['a', 'about', 'above', 'after', 'again', 'against', 'ain', 'all', 'a
m', 'an', 'and', 'any', 'are', 'aren', "aren't", 'as', 'at', 'be', 'be
cause', 'been', 'before', 'being', 'below', 'between', 'both', 'but',
'by', 'can', 'couldn', "couldn't", 'd', 'did', 'didn', "didn't", 'do',
'does', 'doesn', "doesn't", 'doing', 'don', "don't", 'down', 'during',
'each', 'few', 'for', 'from', 'further', 'had', 'hadn', "hadn't", 'ha
s', 'hasn', "hasn't", 'have', 'haven', "haven't", 'having', 'he', 'he
r', 'here', 'hers', 'herself', 'him', 'himself', 'his', 'how', 'i', 'i
f', 'in', 'into', 'is', 'isn', "isn't", 'it', "it's", 'its', 'itself',
'just', 'll', 'm', 'ma', 'me', 'mightn', "mightn't", 'more', 'most',
'mustn', "mustn't", 'my', 'myself', 'needn', "needn't", 'no', 'nor',
'not', 'now', 'o', 'of', 'off', 'on', 'once', 'only', 'or', 'other',
'our', 'ours', 'ourselves', 'out', 'over', 'own', 're', 's', 'same',
'shan', "shan't", 'she', "she's", 'should', "should've", 'shouldn', "s
houldn't", 'so', 'some', 'such', 't', 'than', 'that', "that'll", 'th
e', 'their', 'theirs', 'them', 'themselves', 'then', 'there', 'these',
'they', 'this', 'those', 'through', 'to', 'too', 'under', 'until', 'u
p', 've', 'very', 'was', 'wasn', "wasn't", 'we', 'were', 'weren', "wer
en't", 'what', 'when', 'where', 'which', 'while', 'who', 'whom', 'wh
y', 'will', 'with', 'won', "won't", 'wouldn', "wouldn't", 'y', 'you',
"you'd", "you'll", "you're", "you've", 'your', 'yours', 'yourself', 'y
ourselves']
```

(4) Also perform **stemming** on the obtained tokens. NLTK comes with a Porter stemmer. Try the following code and learn how to use the stemmer.

In [17]:
```
from nltk.stem.porter import PorterStemmer
stemmer = PorterStemmer()
print(stemmer.stem('studying'))
print(stemmer.stem('vector'))
print(stemmer.stem('entropy'))
print(stemmer.stem('hispanic'))
print(stemmer.stem('ambassador'))
```

```
studi
vector
entropi
hispan
ambassador
```

(5) Using the tokens, compute the **TF-IDF vector** for each document. Use the following equation that we learned in the lectures to calculate the term weights, in which t is a token and d is a document:

$$w_{t,d} = (1 + log_{10} tf_{t,d}) \times (log_{10} \frac{N}{df_t}).$$

Note that the TF-IDF vectors should be normalized (i.e., their lengths should be 1).

Represent a TF-IDF vector by a dictionary. The following is a sample TF-IDF vector.

In [6]:
```
{'sanction': 0.014972337775895645, 'lack': 0.008576372825970286, 'regre
```

Out[6]:
```
{'sanction': 0.014972337775895645,
 'lack': 0.008576372825970286,
 'regret': 0.009491784747267843,
 'winter': 0.030424375278541155}
```

(6) Given a query string, calculate the query vector. (Remember to convert it to lower case.) In calculating the query vector, don't consider IDF. I.e., use the following equation to calculate the term weights in the query vector, in which t is a token and q is the query:

$$w_{t,q} = (1 + log_{10} tf_{t,q}).$$

The vector should also be normalized.

(7) Find the document that attains the highest **cosine similarity** score. If we compute the cosine similarity between the query vector and every document vector, it is too inefficient. Instead, implement the following method:

(7.1) For each token t that exists in the corpus, construct its **postings list**---a sorted list in which each element is in the form of (document d, TF-IDF weight w). Such an element provides t's weight w in document d. The elements in the list are sorted by weights in descending order.

(7.2) For each token t in the query, return the top-10 elements in its corresponding postings list. If the token t doesn't exist in the corpus, ignore it.

(7.3) If a document d appears in the top-10 elements of every query token, calculate d's cosine similarity score. Recall that the score is defined as follows. Since d appears in top-10 of all query tokens, we have all the information to calculate its actual score $sim(q, d)$.

$$sim(q,d) = \vec{q} \cdot \vec{d} = \sum_{t \text{ in both q and d}} w_{t,q} \times w_{t,d}.$$

(7.4) If a document d doesn't appear in the top-10 elements of some query token t, use the weight in the 10th element as the upper-bound on t's weight in d's vector. Hence, we can calculate the upper-bound score for d using the query tokens' actual and upper-bound weights with respect to d's vector, as follows.

$$\overline{sim(q,d)} = \sum_{t \in T_1} w_{t,q} \times w_{t,d} + \sum_{t \in T_2} w_{t,q} \times \overline{w_{t,d}}.$$

In the above equation, T_1 includes query tokens whose top-10 elements contain d. T_2 includes query tokens whose top-10 elements do not contain d. $\overline{w_{t,d}}$ is the weight in the 10-th element of t's postings list. As a special case, for a document d that doesn't appear in the top-10 elements of any query token t, its upper-bound score is thus:

$$\overline{sim(q,d)} = \sum_{t \in q} w_{t,q} \times \overline{w_{t,d}}.$$

(7.5) If a document's actual score is better than or equal to the actual scores and upper-bound scores of all other documents, it is returned as the query answer.

If there isn't such a document, it means we need to go deeper than 10 elements into the postings list of each query token.

What to Submit

Submit through Canvas your source code in a single .py file. You can use any standard Python library. The only non-standard library/package allowed for this assignment is NLTK. You .py file must define at least the following functions:

- getidf(token): return the inverse document frequency of a token. If the token doesn't exist in the corpus, return -1. The parameter 'token' is already stemmed. Note the differences between getidf("hispan") and getidf("hispanic") in the examples below.
- getweight(filemae,token): return the TF-IDF weight of a token in the document named 'filename'. If the token doesn't exist in the document, return 0. The parameter 'token' is already stemmed. Note that both getweight("1960-10-21.txt","reason") and getweight("2012-10-16.txt","hispanic") return 0, but for different reasons.
- query(qstring): return a tuple in the form of (filename of the document, score), where the document is the query answer with respect to "qstring" according to (7.5). If no document contains any token in the query, return ("None",0). If we need more than 10 elements from each posting list, return ("fetch more",0).

Some sample results that we should expect from a correct implementation:

- print("%.12f" % getidf("health"))

0.079181246048

- print("%.12f" % getidf("agenda"))

0.363177902413

- print("%.12f" % getidf("vector"))

-1.000000000000

- print("%.12f" % getidf("reason"))

0.000000000000

- print("%.12f" % getidf("hispan"))

0.632023214705

- print("%.12f" % getidf("hispanic"))

-1.000000000000

- print("%.12f" % getweight("2012-10-03.txt","health"))

0.008528366190

- print("%.12f" % getweight("1960-10-21.txt","reason"))

0.000000000000

- print("%.12f" % getweight("1976-10-22.txt","agenda"))

0.012683891289

- print("%.12f" % getweight("2012-10-16.txt","hispan"))

0.023489163449

- print("%.12f" % getweight("2012-10-16.txt","hispanic"))

0.000000000000

- print("(%s, %.12f)" % query("health insurance wall street"))

(2012-10-03.txt, 0.033877975254)

- print("(%s, %.12f)" % query("particular constitutional amendment"))

(fetch more, 0.000000000000)

- print("(%s, %.12f)" % query("terror attack"))

(2004-09-30.txt, 0.026893338131)

- print("(%s, %.12f)" % query("vector entropy"))

(None, 0.000000000000)

Evaluation

Your program will be evaluated using the following criteria:

- Correctness (75 Points)

We will evaluate your code by calling the functions specificed above (getidf - 20 points; getweight - 25 points; query - 30 points). So, make sure to use the same function names, parameter names/types/orders as specified above. We will use the above test cases and other queries and tokens to test your program.

- Efficiency (15 Points)

Don't be satisfied by exhaustive, straightforward implementation. Keep improving its efficiency. An efficient solution should be able to answer a query in a few seconds. Also, it should consider the boundary cases. Your program should behave correctly under special cases and even incorrect input.

- Clarity, organization, modularity, documentation (10 Points)

Follow good coding standards to make your program easy to understand by others and easy to maintain/extend.