Vector Space Model

In this week’s assignment you will implement the core parts of a ranked retrieval system which uses the vector space model to score documents for a given query.

The overall structure of this assignment is similar to the Boolean Retrieval programming exercises in previous weeks.

Note, however, that for this system, you do not (and indeed cannot without considerable effort) handle boolean queries, but just free form text queries.

Because everything is free-form, and we still remove stop words when adding documents to the index, we give you a function `remove_stop_words` which will take a query string and return a list of query terms with the stop words removed.

In [ ]:
```
from textutils import tokenize_document, remove_stop_words
import math, glob
from collections import defaultdict
```

We have global variables for the index, the document id to document mapping and a documentid counter.

In [ ]:
```
'''Map document titles to document ids'''
documents = {}

'''A running counter for assigning numerical IDs to documents'''
docid_counter = 1

'''The document-term frequencies'''
the_index = dict()
```

First we need to build the index. For the vector space model, we need to have different information in the index than just regular postings lists. We now need to store the frequency of each term for each document.

The two quantities we need to know for each term in the collection of documents are (i) the number of documents in which the term appears, and (ii) the number of times a term appears for each of the documents that contain the term.

We leave it up to you how exactly you want to store these quantities, and ask you to implement getter functions that match your storage scheme for the two quantities below.
In [ ]:

def add_document(doc):
    
    Add a document to the inverted index. Returns the document's ID
    
    global documents, docid_counter, the_index
    
    # do not re-add the same document.
    if doc in documents.values():
        return
    docid = docid_counter
    documents[docid] = doc
    docid_counter += 1
    print("Adding document %s to inverted index with document ID %d" % (doc, docid))
    
for term in tokenize_document(doc):
    # TODO: collect term frequencies and document frequencies per term.
    pass

Now we can build the index:

In [ ]:

for document in glob.glob('..\shared\corpus\*.txt'):
    add_document(document)

Next we need wrappers to extract and compute the various numbers and formulas presented in the book in sections 6.2 and 6.3.

First you have to implement the function $tf$ that gives us the term frequency $tf_{i,d}$ of a term $term$ in document with the id $docid$:

In [ ]:

def tf(term, docid):
    
    Calculate term frequency for term in docid. Return 0 if term not in index,
    or term does not appear in document.
    
    # TODO: implement
    return 0

Next, you have to implement function $df$ which gives us the document frequency $df_{i}$ of a term $term$.

In [ ]:

def df(term):
    
    Extract frequency of term for document with id docid from index
    
    # TODO: implement
    return 0

You have to implement the two functions $tf$ and $df$ to extract term and per-term document frequencies from the index you’ve created in add_document.

From here, you should use the functions $tf$ and $df$ where appropriate.
Next you can implement the function \( \text{idf} \) to compute the inverse document frequency of a term \( t \):
\[
\text{idf}_t = \log \frac{N}{df_t},
\]

In [ ]:
```python
def idf(term):
    
    Compute \( \text{idf}_t \) for a term
    
    # TODO: implement
    return 0
```

Now we have all the parts that we need to compute the \( \text{tf-idf} \) weight for a term in a document.

The \( \text{tf-idf} \) weighting scheme assigns weights with the following formula:
\[
\text{tf-idf}_{t,d} = \text{tf}_{t,d} \cdot \text{idf}_t.
\]

In [ ]:
```python
def tf_idf(term, docid):
    
    Compute \( \text{tf-idf} \) for term and docid
    
    # TODO: implement
    return 0
```

The last helper function we will need is a normalization function. For this assignment we use cosine normalization, which is
\[
C = \sqrt{w_1^2 + w_2^2 + \ldots + w_M^2}
\]

where \( w_t = \text{tf-idf}_{t,docid} \forall t \in \{ \text{terms contained in document with docid} \} \).

Note that in the book the cosine normalization is given as \( \frac{1}{C} \) in Figure 6.15, but if you already compute the reciprocal of \( C \) in \text{norm_cosine} you would have to substitute a multiplication for the division in line 9 of the algorithm in Figure 6.14.

In [ ]:
```python
def norm_cosine(docid):
    
    Compute cosine normalization for docid
    
    # TODO: implement
    return 0
```

Now we can implement the cosine scoring function given as pseudo-code in Figure 6.14. We give the variable \( K \) which determines how many results we return as an optional parameter.

We give you the step “Initialize Length[N]”, as (i) the book is not very clear on what you should do for that step and (ii) this array is strictly not necessary, as we could just call \text{norm_cosine} when we normalize the scores in step 2 of \text{cosine_score}.

We leave it up to you to implement \text{cosine_score} using either the term-at-a-time or document-at-a-time strategies, as the choice of strategy should not change the results in any way.

For the terms \( w_{t,d} \) and \( w_{t,q} \) shown on line 6 of the algorithm in Figure 6.14 you should substitute the weighted frequency of term \( t \) for document \( d \) and the weight of term \( t \) with respect to the query.
You can pick a formula for each weight by picking a pair of expressions from the first two columns (Term frequency and Document frequency) of the table in Figure 6.15.

The provided code uses cosine normalization for the document weights -- that's the provided code for step "Initialize Length[N]".

In [ ]:

```python
def cosine_score(query, K=10):
    
    # Initialize arrays so we can use docids as indices
    scores = [0] * (len(documents.keys())+1)
    length = [0] * (len(documents.keys())+1)
    # Precompute length array values -- these are the normalization factors
    for d in documents.keys():
        length[d] = norm_cosine(d)
    # TODO: implement
    # 1. compute scores for each document, store in scores array
    # 2. Normalize using the length array
    # 3. Return top-K: sort by descending score and return first K elements of result.
    return []
```

Now that we have a scoring function, we can write `execute_query` as a thin wrapper around the scoring function:

In [ ]:

```python
def execute_query(query):
    
    # We use the new remove_stop_words helper function to split and normalize
    # the query. After this call query is a list of words to query for
    query = remove_stop_words(query)
    return cosine_score(query)
```

As usual, we provide a `print_result` function:

In [ ]:

```python
def print_result(results):
    if not results:
        print("No documents found")
        print()
        return
    else:
        # If we got some results, print them
        for docid, score in results:
            print('%2d(%6.3f) -> %s' % (docid, score, documents[docid]))
        print()
        return
```

Now we can run some queries on our index:
In [ ]:

```python
print_result(execute_query("Brutus Calpurnia"))
print_result(execute_query("Caesar"))
```

For the query "Brutus Calpurnia" the document `the_tragedy_of_julius_caesar.txt` should have by far the highest score.

For the query "Caesar" the top three documents should be (in order):

- `the_tragedy_of_julius_caesar.txt`
- `the_tragedy_of_antony_and_cleopatra.txt`
- `cybeline.txt`