EECS 767 Project work plan part 1

1. Document Processing
   1. Create a linked-list class in Python <http://stackabuse.com/python-linked-lists/>
   2. Store stop words in a “stop list”
      1. # this may be a third part list containing ‘no value words’ such as: is, a, the, then, an, and, are, that, this… (discussed on pg 25 of text)
   3. Store index files in their own directory (prior to program being run)
      1. #note this may change after implementing the web crawler
   4. Accept directory path as input from user
   5. Open directory containing files to be indexed
   6. Create index list to store normalized tokens
   7. For each file in the directory
      1. Read document and:
         1. (remove html tags) Ignore all characters between “<” and “\>
         2. Remove all punctuation/special characters [e.g. ignore []{}!@#$%^&\*()-+]
         3. Ignore all “stop words” in stop list
         4. Convert remaining terms (characters delimited by spaces) to lower case and store as tuple in list with format (word, document#)
   8. Quicksort index list to arrange terms in alphabetical order ascending and document ID order ascending.
   9. Create hash table terms\_dictionary to store (Key term, Value tuple(document frequency [number of document containing term], term frequency [total count of term])
   10. Calculate term frequency (TF) and document frequency (DF) from index list
       1. Initialize variable term as first item in index list tuple[0]
       2. For each tuple in index list
          1. Initialize variable previous\_document\_ID as -1
          2. Initialize variable DTF as 0. #This will store term frequency per document
          3. While tuple[0] == term #check if term changes
             1. docID = tuple[0,1] #store docID
             2. TF +=1 #count occurrence of term
             3. DTF +=1
             4. If document ID tuple[0,1] is not previous\_document\_id,

Create linked\_list\_node (docID,DTF)

DF +=1 #add 1 to document freq

DTF =0 #reset document term freq count

Add Key tuple[0] value list(tuple(TF,DF),linked\_list) to hash table

* + - * 1. Next term in index

1. Query processing
   1. Accept query from user input #note query only stores terms, not operators (And, or,like etc.)
   2. Sterilize query input for stop words and special characters similar to step 1g
   3. Store query as list query\_terms
   4. Conduct binary search for each term in query\_terms against keys in hash table
      1. For each term match
      2. Append key value pair from hash table to result list
2. Vector Space model
   1. Rank query results
      1. Determine number of documents returned
         1. For each item in result list
            1. For each document in linked\_list in item
            2. Add document\_ID to set document\_set{}# document\_ID islinked\_list\_node[0]
         2. N = length (document\_set{})
         3. Create multisets for each document\_set# AKA bags described in class
            1. Create list multiset\_list to store multisets
            2. For each item in document\_set

Append multiset Dn to multiset\_list # n is the document number

* + 1. Calculate TFIDF for each term and document
       1. For each key in terms\_dictionary
          1. DF=Key[1][0] #first index of second item in item list
          2. IDF = log10(N/DF) #weighted document frequency of the term # calculate weighted IDF

For each document in document\_set #Calc TFIDF

Check if document ID exists in linked list

For each item in linked\_list

If document ID

TFDT=document[1] #the second index in the document, the term frequency in that document linked\_list node

TFIDF=TFDT\*IDF

Append TFIDF to multiset Dn

Else TFIDF=0

Append TFIDF to multiset Dn

* + 1. Calculate TFIDF for Query vector
       1. Create multiset for query query\_multiset
       2. For each Key in terms\_dictionary
          1. DF=Key[1][0] #first index of second item in item list
          2. IDF = log10(N/DF) #weighted document frequency of the term # calculate weighted IDF

Check if string(key) exists in query\_terms

If string(key) in query\_terms

TFDT= count term in query\_terms

TFIDF=TFDT\*IDF

Append TFIDF to multiset Dn

Else TFIDF=0

Append TFIDF to query\_multiset

1. Calculate distance between query and document vectors (slide 42 of preso)
   1. Create list document\_query\_similarity
   2. For every document\_vector in multiset\_list
      1. For every document\_tfidf in document\_vector
         1. i=0 #instantiate loop counter for query\_multiset
         2. Numerator+=(Document\_tfidf\*query\_multiset[i])
         3. Document\_denominator+=(document\_tfidf)^2
         4. i+=1
      2. For every query\_tfidf in query\_multiset
         1. Query\_denominator+=(query\_tdidf)^2
      3. Similarity = Numerator / (sqrt(Query\_denominator)\*sqrt(Document\_denominator))
      4. Append tuple(document\_vector index, similarity) to list document\_query\_similarity
   3. Quicksort list document\_query\_similarity by similarity descending
   4. Return top 10 items from document\_query\_similarity
2. Web crawler (2nd half of semester)

Project guidelines from Blackboard:

1. **Document processing and indexing**. You will be provided with a zip file that contains 63 HTML documents collected from Wikipedia. First, pre-process the documents by removing all HTML tags and convert everything into lower case. Implement a stop list and a stemmer to pre-process the documents (for the stop list and stemmer, you are allowed to use third-party open source code). Second, build an inverted index (including dictionary and posting lists) for the documents. Please make sure to keep all the frequency information.

2. **Vector Space model**. The goal is to provide a TF-IDF-based ranking for the documents. Since you have already collected frequency information in step 1, please further compute IDF for each term. For each document, find a way to calculate the length of the corresponding document vector. For each incoming query, pre-process the query with the stop list and stemmer. Identify candidate documents that contain at least one query term. Meanwhile, compute the length of the query vector. Finally, compute the TF-IDF similarity score between the query and each candidate document [hint: there is no need to construct the complete document vector, or loop through all dimensions in the vector space], and sort the documents by the score.

3. **Niche crawler**. You should identify a domain of interest (e.g., ku, Wikipedia, nfl, etc.). Ideally, the size of the domain should be manageable, and the link structure is not too complicate to follow. Your crawler should contain at least three components: (1) a multi-threaded spider that fetches and parses webpages, (2) the URL frontier which stores to-be-crawled URLs; and (3) the URL repository that stores crawled URLs. Please be polite to the site. Please collect a few hundreds to a few thousands of pages.

4. Please feed the collected documents to the search engine that you implemented in step 2. Please implement a Web-based interface to take user queries and return answers (document names, snapshot with search term(s) highlighted, and URL) to the user. You only need to provide a reasonable (not so fancy) interface, you can use WYSIWYG editors to generate HTML. Keep this version of your search engine, since it will be compared with two future versions.

5. Add term proximity into your scoring mechanism. Define your own score that reflects the proximity of search terms in each document. Define your own algorithm to integrate term proximity score with the tf-idf score from step 2.

6. Add one of the following to your search engine:

6.1. Search personalization: use cookies to track users. Record each search and each click-through. For a new query, add a small component of the "search history" as query expansion.

6.2. Relevance feedback. For each query, allow the user to identify a set of "positive" and "negative" results. Use user feedback to update the query and return new (refined) results to the user.

7. Please evaluate and compare the performance of the original search engine (step 4), and the new versions (step 5 and 6).

You are expected to do a project presentation during the last two weeks of the semester. Please make a few powerpoint slides to introduce your search engine, especially the system architecture and design notes. Please also prepare a short demonstration of your search engine.

You need to submit a project report will all your source code (please do NOT include crawled documents). It is strongly suggested that you make the search engine available when we grade your project report.