### CS F469

### Information Retrieval

### Assignment - 1

### Inverted Index Construction and Preprocessing

import nltk  
import textract  
from nltk.corpus import \*  
from nltk.stem.porter import \*  
import os  
import pickle  
from nltk.tokenize import sent\_tokenize, word\_tokenize  
from pathlib import Path

Before constructing the inverting index we need to install essential datasets for stopword removal and normalization done during the preprocessing step. We have used the available nltk datasets for the same purpose.

# ! Uncomment in first run   
# nltk.download('punkt')  
# nltk.download('words')  
# nltk.download('stopwords')

### Normalization

Porter stemming algorithm is used for the purpose of normalization of tokenized words. It reduces a given term to its root form, thus the query is robust to more typical morphological and inflexional endings. We have the used the PorterStemmer by nltk library for the same purpose.

# stemming porter object  
stemmer = PorterStemmer()  
root = Path(".")

Before construction of inverted index, we have converted all the pdf files given to word documents as directly reading from pdfs lead to a higher loss as compared to word documents and the pdfs might contain watermarks or unselectable text which can hamper our text extraction purpose. We have used **textract** and **sent\_tokenize** by nltk library to extract the raw text from the documents and tokenize every sentence to preserve the context.

# list of all doc names  
files = list()  
for dir in [r"\Auto", r"\Property"]:  
 cur\_dir = r".\Docs" + dir  
 for file in os.listdir(cur\_dir):  
 cur\_path = r".\Docs" + dir + "\\" + file  
 files.append(cur\_path)  
files.sort()

docs = list()  
for x in range(len(files)):  
 for i in sent\_tokenize(textract.process(files[x]).decode("utf8")):   
 docs.append((i, x))

### Inverted Index Construction

The structure of our inverted index is a list of tuples of form *(doc index, file index, frequency)* where doc index is the index of the newly transformed documents, file index is the index of file the transformed document belongs to, and the frequency is the number of times the normalized term or the **key** appears in the transformed document. The size of our transformed documents is dynamic in nature, the minimum threshold we have kept is 500 characters. We keep on adding sentences to a document until we get atleast 500 characters. Benefit of adding sentences instead of words is that context of each transformed document is preserved, it starts with a sentence and there is no abrupt ending of the context when the user queries for any document.

For a given sentence, we tokenize it to words using **nltk.word\_tokenize** and convert those words to lower case. if the word is a stopword, we skip it, otherwise we normalize the word using porter stemming and construct the inverted index with **key** as the normalized term. Our inverted\_index is a dictionary whose **key** is the normalized term and **value** is the list of tuples as mentioned above.

After the construction of the inverted index table, we sort each list by the third element of the tuple, that is frequency of the **key** so that we get relevant results whenever user queries for anything.

# set of Stop words  
stop\_words = set(stopwords.words('english'))  
  
"""  
{  
 key : string (normalized)  
 value: list of ("doc index", file\_index, frequency)   
}  
"""  
inverted\_idx = dict()  
  
# list of string modified document  
documents = list()  
  
count\_id = 0  
  
def process(doc\_index):  
 """  
 Reads file, tokenize it, normalizes it and builds the inverted index  
 """  
  
 result = doc\_index + 1  
 global count\_id  
 text = docs[doc\_index][0]  
 file\_index = docs[doc\_index][1]  
 while doc\_index + 1 < len(docs):  
 doc\_index += 1  
 if docs[doc\_index][1] == docs[doc\_index - 1][1] and len(text + docs[doc\_index][0]) <= 500:  
 text += docs[doc\_index][0]  
 else:  
 result = doc\_index  
 break  
  
 tokens = nltk.tokenize.word\_tokenize(str(text))  
  
 new\_token = list()  
 for i in tokens:  
 new\_token.append(i.lower())  
 tokens = new\_token  
  
 curr\_str = ""  
 normalised\_word\_freq = dict()  
 for j in range(len(tokens)):  
 curr\_str += tokens[j] + " "  
  
 normal = stemmer.stem(tokens[j].lower())  
 if normalised\_word\_freq.get(normal) != None:  
 normalised\_word\_freq[normal] += 1  
 else:  
 normalised\_word\_freq[normal] = 1   
  
 documents.append((curr\_str, files[file\_index]))  
   
 visited = set()  
 for j in range(len(tokens)):  
 normalised\_word = stemmer.stem(tokens[j].lower())  
 if tokens[j].lower() not in stop\_words and normalised\_word not in visited:  
 visited.add(normalised\_word)  
  
 if inverted\_idx.get(normalised\_word) != None:  
 inverted\_idx[normalised\_word].append((count\_id, file\_index, normalised\_word\_freq[normalised\_word]))  
 else:  
 inverted\_idx[normalised\_word] = [(count\_id, file\_index, normalised\_word\_freq[normalised\_word])]  
 count\_id += 1  
  
 return result  
  
i = 0  
while i < len(docs):  
 i = process(i)  
  
for x in inverted\_idx:  
 inverted\_idx[x] = sorted(inverted\_idx[x], key=lambda y: -y[2])

After the entire preprocessing stage inverted index table, transformed document list and the list of original files is dumped to respective binary files so that whenever user wants to query, our search engine can just open those files and use those data structures for querying directly instead of preprocessing again and again.

my\_path = root / "Pickled\_files" / "Inverted\_index"  
dbfile = open(my\_path, 'wb')  
pickle.dump(inverted\_idx, dbfile)   
dbfile.close()  
  
my\_path = root / "Pickled\_files" / "Documents"  
dbfile = open(my\_path, 'wb')  
pickle.dump(documents, dbfile)   
dbfile.close()  
  
my\_path = root / "Pickled\_files" / "Files"  
dbfile = open(my\_path, 'wb')  
pickle.dump(files, dbfile)   
dbfile.close()