Algorithmic Details of the Vector Space Retrieval Model

invidx_cons.py

This script helps to create inverted postings list and dictionary by scanning through the documents.

First we extract the path and indexfile name from the command line arguments using sys library.

Then we get the list of files containing a collection of docs in the given path using os library. We start iterating over the list to scan through each document. While iterating, we keep track of number of docs as it will be later used in normalization. We tokenize the lines using nltk tokenizer. We extract the docid from the first line. Now for proceeding lines, if the line starts with "<\DOC>", we treat it as the end of document and extract doc id for the next doc from the next line by breaking out of the outermost while loop. If it starts with "<TEXT>", we start scanning for words. While scanning we also search for tags from placing them properly in the inverted list. Also, if we encounter a tag, we store it and search for a similar tag in the next word for merging such instances (like P:Barack P:Obama-→ we store it as P:barack, P:obama, P:barack :Obama). Also we lowercase all the word before storing. For storing, we use dictionary as it will be handy in quick retrieval. Keys of the dictionary are the terms/words while value is another dictionary containing docids (in which it occurs) as the keys and the corresponding document frequency (num of times it appears in that doc) as the value.

{term1: {doc1:13,doc4:12…….},term2:{doc52:15,doc90:31…..}…………}

While storing the key we also check if its already in the posting list. Same goes for docids for each term. Dict.get() function turns out to be really helpful for the same.

Now we do the required normalizing. In the dictionary corresponding to each term, we also add another key called "inv_doc_freq" which stores the inverse document frequency of the term. Its found by -> np.log2(1+num_docs/n) , where np~numpy, num_docs~total number of docs and n~number of docs containing that term.

We also calculate log normalized terms frequnecies and multiply them by idfs to store the final term weights. We also maintain another dictionary called the norms which stores the norm values of doc vectors so as to avoid storing the whole sparse vector. We progressively calculate the norms while calculating the weights by adding the squared weights. Finally we take np.sqrt() of all the values in the norms dictionary.

In the intermediate calculations, we limit the num of decimals to 4 for speed efficiency and finally while storing in the binary files, we limit the num of decimals to 2 for storage efficiency.

Finally after completing all this, we use pickle and bzip libraries for dumping and compression. Although bzipping requires extra time, but it almost reduces the size to 1/3rd of that achieved only by pickling. Also while pickling we use pickling.HIGHEST_PROTOCOL as the protocol parameter value in the pickle.dump() function which facilitates the most efficient dumping.

In the first file named indexfile.dict we dump a sorted list of tuples of the form (keys, document frequency). First term of tuple is used as the sorting key. Then we dump the inverted postings list and norms in the second file called indexfile.idx

printdict.py

First we load the dictionary of terms using pickle and the filename extracted from the command line argument. Next start a for loop for printing the dictionary-

```
i=0

for term in dictionary:

    print(term[0]+":"+str(term[1])+":"+str(i))

    i+=1
```

Here term[0] represents the key, term[1] the doc frequency i is the index of that term in the sorted list.

vecsearch.py

First we extract the cut-off k, query file, result file, dict file and index file from the command line argument. Then we load the dictionary, norms and inverted posting list. Then we start scanning through the queries file. If the line starts with <num>, we extract the qid. If it starts with we extract the query. We tokenize the query through nltk library and then tag it through STANFORD NER tagger. We then process the query. If it ends with * then we find a term word that starts withat prefix using binary search in the sorted list and then iterate in the previous and next consecutive terms till the term starts with that prefix, otherwise break. We also check if the term starts with O:/P:/L: or N: so as to add it according to the appropriate tag. If it starts with N:, we add it with all three tags. After creating the query vector, we log normalize the frequencies and multiply it with the inverse doc frequency of that term. We also calculate the norm of that vector. Then to find the similarity scores, we iterate through doc ids (maintained in the posting list) for each term in the query and simply multiply and add it to similarity score of the corresponding doc. We then normalize all the similarity scores with norm of query vector and corresponding doc vector (stored in the norms dictionary). We then find the top k similarity scores and write it out in the results file in the required format.

For tunings-

I removed all the numbers from the document text. I tried using different tokenizers like spacy, nltk, .split()… and finally selected nltk. I tried using nltk stemmer and stop word removal but that didn't help. In the query I replaced "/" with " " to avoid skipping words. Also I split using the delimiter "-" to again avoid skipping individual terms. Eg- iran-contra in query will be stored as iran, contra, iran-contra.