Documentation

Problem statement

In the given problem statement, we were supposed to implement the unigram inverted index data structure on the given dataset. The given dataset consisted of files having various conversations and dialogues which needed to be imported and preprocessed before applying any computation. This required thorough reading and discarding the file which were not encoded in utf-8.

After the successful import, we then would convert this data into a data structure (given unigram inverted index) which will be used to solve the following queries. The queries will be input in the manner given in the doc.

The function **importDocument** imports all the files from the given dataset and discards the file which does not follow the utf-8 encoding. This results in **5** discarded files and **1128** total processed files. All the files are stored in the dictionary **documents** and the name of the files in **files**. Each value in the dictionary is a 2d object with ith iteration containing the lines and the jth words in each line.

Preprocessing

Objective is to make the raw data clean and computable for the given problem statement. It has been done in 3 steps:

- **Step 1 : onlyWords**
  Iterate over the given **documents** and iterate over each line in the document. White spaces in each word are removed and with that we check the length of the word in order to eliminate the white spaces. After this, **nltk.RegexpTokenizer(r"\w+")** is applied on each word in order to remove any extra special characters. This leaves the word with alphanumeric and underscore values. Again the length is checked to eliminate the empty words and all the remaining words are lowercase.

  Returned - a dictionary containing documents having only alpha-numberic and underscore values.

- **Step 2 : removeStopWords**
  set(stopwords.words('english')) is used to collect all the stop words in the dictionary. We iterate through the dictionary and each line and for every line, we check if the word is in the stopwords and remove it.

  Returned - a dictionary containing only non stop words.

- **Step 3 : lemmatization**
  WordNetLemmatizer() is used to collect all the lemmatized words and an inbuilt function to lemmatize the words. We iterate through the dictionary and each line and for every line, we lemmatize the word

  Returned - a dictionary containing only base/lemmatized words.

There's another preprocessing step but this is done after converting into a unigram index. It is written in the next part.

Unigram Data Index

Now, we have all the required words for our unigram index, we create a dictionary of these words having the following norm for the value:

- The value is a list of length 2
- First index contains the frequency of documents
- Second index contains the sorted linked list of all the document indices which contain this word.

This data structure creation has been done in 2 steps:

- **Creation : uniqueWords**
  We iterate over the preprocessed **documents** and apply an enumerate function to it in order to obtain indices of the files. Now we iterate over the lines and find the words. Before iteration of words, we create a set of all the words in the doc to eliminate repeating words. If the word does not exist in the dictionary then we initialize it with size 1 and list containing the current document. Since the document is already in sorted order, we don't need to sort the linked list. If the word already exists, then we just update the values and insert the document number.

  Returned : a dictionary of unigram index data structure and a list of unique words

- **Refining : removeUnderscore**
  There are still some words which are either irrelevant or redundant (for example _____ and apple_pie respectively). So in this function, we aim to remove all the underscores and repeat the preprocessing steps. First we replace all underscores with spaces and then strip them to get clean alphanumeric words only. We apply the above regex function (**nltk.RegexpTokenizer(r"\w+")**) to remove all the extra symbols and make a dictionary of words (since there might be more than one word coming from each **word**). Then we lemmatized, lowercase and iterated over the list of words that came from the **word**. In this iteration we simply check if the word is relevant, existing or redundant and apply the following cases as done in the above stages.

Returned : a dictionary of unigram index data structure of only alphanumeric words and respective list of unique words

Our total number of words comes out to be **64239**.

Query Computation

We are ready with our data structure. Now we will test it using the given queries. We follow the given guidelines on how to take the input. The n and query case is similar to the given conditions in the document. Operations will take input as : **"op1,op2,op3"**. Also operation cannot start with **NOT**.

A function **compute** runs on every query and operation. The first step is to preprocess the query using the same steps in the preprocessing and initializing **doclist** with the dictionary value of the first word in the query. Then we run a loop over operations and send the **doclist** and the next query **query(i+1)** to the function. At the same time we check the **NOT** condition and according to that alter the parameters. A variable size contains the size of all the input documents registered in the code i.e. **1128**.

The function **queryOperation** merges the following linked list of documents and calculates the intersection as well. Both these steps have been done simultaneously. An if condition checks the **NOT** condition and if true, a function **queryNOT** returns the complement of the list. A 2 pointer loop is run to merge the 2 lists in **O(x+y)** and a number of comparisons are stored in the **comp** variable. After the merging step, we check the query and according to that, we either return union (in case of **OR**) or intersection (in case of **AND**).

Returned - a linked list containing all the required documents and no. of comparisons