Project 1: Information Retrieval System

Group Members:
Suchit Sahoo(ss6630)
Aarushi Jain(aj3087)

List of Files:
Main.py
googleSearchModule.py
metricsModule.py
queryExapansionModule.py
requirements.txt
Config.json
sampleResponse.json

Problem Statement:
Given a query, exploit user-provided relevance feedback to generate the top-k results from the web.

Setting up the project:

- Extract the proj1.tar.gz file
- Navigate to the proj1 directory
- Run the following command
    - pip3 install -r requirements.txt
- To start using the program run the following command
    - python3 main.py <google api key> <google engine id> <desired precision> <query>

Google API Key: AIzaSyAMw1otSb1Tuh6Qe81Eo799bNSO55cqtTI
Google Engine Id: 59dddcc6d7ae4afdd

Design Components:

Main.py File: The entry point to our application is the main method defined in the main.py file. It takes 4 arguments-Google API key, Google Engine Id, query and precision. It returns the top 10 results to the user and takes the user relevance feedback. If the desired precision is not achieved, it calls the queryExpansion method which modifies the query and the top 10 results are returned based on the new query. This process is repeated till the desired precision is achieved.

googleSearchModule.py File: This module filters out the non-html files and returns the top 10 google search results. It takes 2 arguments-Google API key and Google Engine Id.

metricsModule.py File: It returns the precision value based on user-provided relevance feedback.

queryExapansionModule.py File: This module modifies the query. It computes raw features:tfIdf for each term, proximity to the query words, raw term frequency and calculates the score based on these features.

Handling Non-Html Files:

When we receive a non Html file in the google search engine API we don't include it in the set of documents returned to the user. Therefore all the results that the user receives are from Html files. The Precision@10 is also computed on the number of non-html files received.

Algorithm Design and Design choice:

1. **Preprocessing**:
    a. **Preprocessing of query**: To standardize the queries, first, the queries are **striped** to remove any **additional blank spaces** at the start and end of the query. Apart from that **case folding** is done to convert the query into **lowercase**.
    b. Pre-Processing of the Google Search Results: For the relevant documents obtained from google search results we extract the **fileformat, title, snippet and link** properties. **Casefolding and Removal of punctuations** are performed on properties of **title and snippet/summary** of the retrieved document.
2. **Creating Document Collections:** We take a combination of **relevant search result**'s URL domain Name, title and snippets as contents of a Document. We maintain an in-memory documents array that stores all the relevant documents retrieved till the given iteration.
3. **Query Expansion Algorithm:** For each query, we compute a unigram based and a bigram-based Query Expansion.
    a. Preprocessing:
        i. Tokenization: nltk's word_tokenizer module is used to tokenize the query and the document collection.
        ii. StopWords Removal: StopWords are commonly occurring words that don't contribute a lot to the meaning of a document. Including stopwords in the query will not yield more relevant documents and rather may make the results more irrelevant. Therefore stopwords are removed from the document collections as a preprocessing step. **Nltk's English stopword** list with a few additional terms was used to curate the stopwords list.
    b. **Features**: To compute the query expansion we use the following features.
        i. **Term Collection Frequency**: Instead of considering document-wise term frequency we have considered a collection-wide term frequency. Since the objective is to augment the query with terms that could provide more relevant information, document-wise term frequency doesn't provide any

insights. Rather we need a metric that captures the entire collection hence collection wide frequency is more insightful.

$$cf(t,c) = log(count(t,c)),\ cf: collection\ frequency,\ t: term,\ c: collection$$

ii. **Inverse Document Frequency**: Inverse Document Frequency is utilized to identify rarer terms across the document collection. Although we may have removed commonly used stopwords, but specific to a given context we might have a term occurring in all documents. Such terms might not be very useful to delineate the search results.

For example: For a query **banking products** all the relevant documents will have the term **financial** appearing in them. Including this term in the query and now searching for **banking products financial** will not yield the specific banking product we are looking for. However, including a rarer term like **bonds** may provide more relevant results.

$$idf(t) = log(N/df(t)),\ N: Total\ no\ of\ documents,\ t: term,\ d: document$$
$$df(t): no\ of\ documents\ that\ contain\ term\ t$$

iii. **Proximity:** A netCorpus is created by combining all the contents of the document collection. Within this document collection, we determine the min distance and direction(if the term in collection lies after the query term 1 else it is -1) of each collection term with each of the query term.

For example: q = [“cat”,”eating”] , corpus=[“cat”,”is”,”eating”,” a”,”mice”]
mindist((”mice”,“cat”)) = 4 , dir((“cat”,”mice”)) = 1
mindist((”mice”,“eating”)) = 2 , dir((“eating”,”mice”)) = 1

$$proximity(t) = min(mindist(t,q_i)),\ for\ all\ q_i \epsilon q,\ t: term,\ q: query$$

iv. **POS Tagging:** During our experiments, we found that using Nouns terms for query expansion yielded more relevant documents as compared to other parts of speech (POS) tags like adjectives, adverbs, prepositions etc.
This observation is grounded in the linguistic fact that Nouns represent the central idea of the text. To account for this fact we increased the term collection frequency of terms containing a noun by a factor of 2.
Nltk ‘s pos_tagger was used to identify each term POS tag in the collection.

$$if\ term\ contains\ "NN"\ pos\ tag$$
$$cf(t,c) = 2 * cf(t,c)$$

For example: While searching with query **ms** for Microsoft-related articles. We found that without using the POS tagging the subsequent query was goals ms which didn't yield good results. After introducing the POS tagging-based rule now the same query after the first iteration yielded a query of microsoft ms which produced more relevant documents.

c. **UniGram Score:**
$unigramScore(t) = \alpha * cf(t,c) + \beta * log(1/proximity(t)) + (1 - \alpha - \beta) * idf(t)$
Empirically the value of 0.5,0.3 for alpha and beta produced the best results.

d. **BiGram Score:**
   i. We have used **nltk's ngrams** modules to obtain the biGrams present in the query and document collection.
   ii. While computing the biGram Score we have ignored the idf term. BiGrams are inherently rare than unigrams and since the document collection is very small usually ranging from 10-20 documents. The idf for bigrams were mostly equal and didn't provide additional insights.

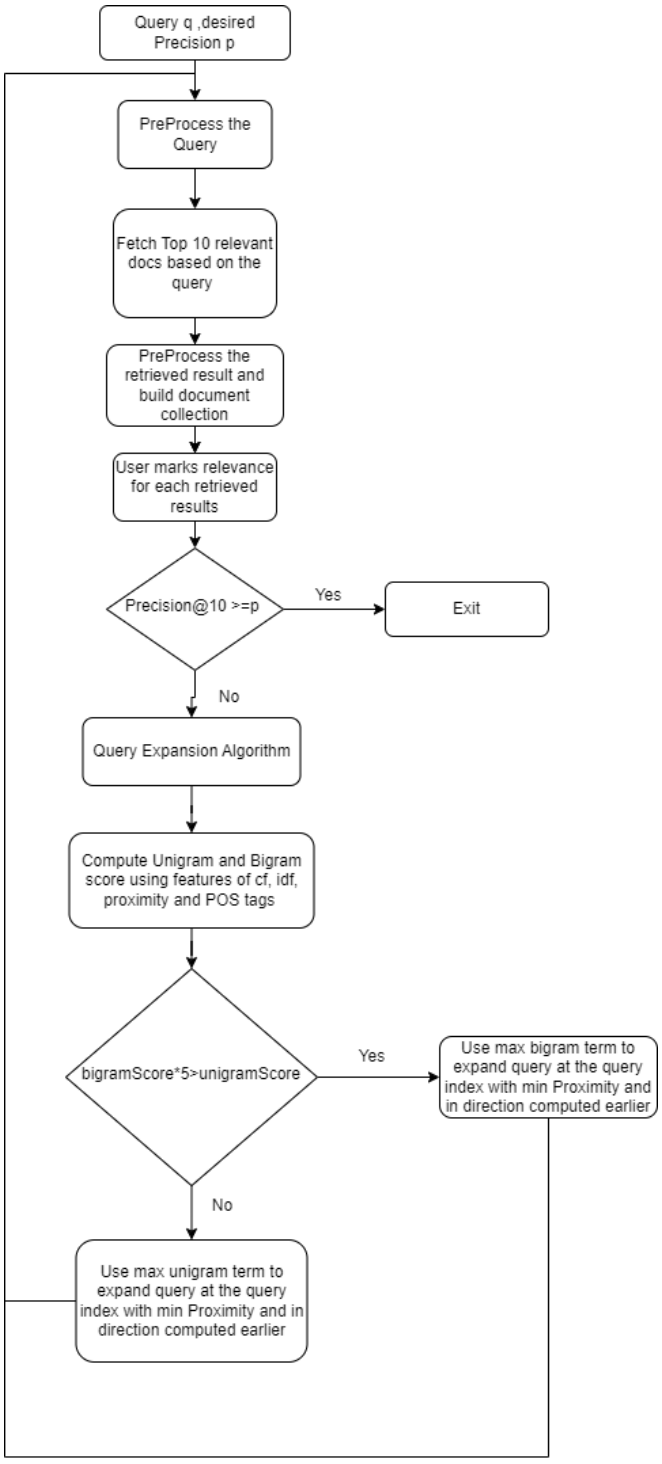
$bigramScore(t) = \alpha * cf(t,c) + (1 - \alpha) * log(1/proximity(t))$
Empirically the alpha value of 0.6 provides the best result.

e. **Comparing UniGram and BiGram Scores:** BiGrams are rare than unigram and capture more context and structure about the query. To account for their more importance BiGram Scores are scaled by 5 before comparing them to Unigram Scores.

$$bigramScore(t) = 5 * bigramScore(t)$$
$$maxBigram = argmax(bigramScore(t))$$
$$maxUnigram = argmax(unigramScore(t))$$
$$term\ to\ added = argmax(bigramScore(maxBigram), unigramScore(MaxUnigram))$$

f. **Query Expansion:** After identifying the term(t) with max score. We need to augment the query with the term. We use proximity scores and direction to identify the exact location of the term.
Query index(i) for Expansion is determined as follows,
$i = argmin_i(proximity(t, q_i))$, $where\ t{:}\ term\ to\ be\ added,\ q{:}\ tokenized\ query$
Direction in which the term t should be added is determined by dir of $proximity(t, q_i)$. If it is 1 t is appended after q_i else it is appended before q_i.

Flow Diagram:



References:

- Afuan, Lasmedi. “A study: query expansion methods in information retrieval.” *Journal of Physics: Conference Series*, 2019, p. 12, https://iopscience.iop.org/article/10.1088/1742-6596/1367/1/012001.

- Claveau, Vincent. “Query expansion with artificially generated texts.” *CoRR*, vol. abs/2012.08787, 2020.