README

1. **Names**:
   Lara Karacasu - lk2859
   Ulas Onat Alakent - ua2182

2. **List of files to submit:**
   a. search_engine.py
   b. stop_word.txt
   c. html_tags.txt
   d. README.pdf
   e. transcript.pdf

3. **How to run program:**
   a. Install sklearn:
      pip3 install --upgrade setuptools wheel pip
      pip3 install scikit-learn
   b. Unzip file:
      tar -xvf filename.tar.gz
   c. Run implementation:
      python3 search_engine.py
      AIzaSyDugAgxKOHTfHRymoH7-ZyUjANV5vKW01Q a33a167b34969b3a8
      <Precision> <Query>

4. **Description of the internal design:**

We start by reading in the stop words and html tags from the respective txt files. If the command-line arguments are entered incorrectly, we terminate the program. After accessing the Google API, we go through all the words in the query and exclude them from the list of stop words if the query term is also a stop word.

In our main loop, we make sure to terminate if the results found are less than 10, per the requirements. Otherwise, then we iterate through all the results we received using the Google Search API, and create a lexicon by processing HTML objects in a way that we remove the HTML tags and stopwords, and only keep in the alphanumeric terms. After printing out the respective URL, title, and summary for the result, we make sure to skip it if it’s not an HTML file, so that we won’t get an input from the user in that case. Only after building the lexicon and calculating the running sum of relevant and nonrelevant documents, we execute the Rocchio’s algorithm by using the beta value of 0.75 and gamma value of 0.15 (Relevance Feedback in

Information Retrieval, 1983). We also make sure that after the calculations for the running sums, a value will be set back to 0 if it becomes negative.

Our code considers only the HTML files when calculating the precision value for each iteration as we divide the current precision value by the number of html files. From there on, we call the orderGenerator function which generates all possible query terms and decides on the order with highest ngrams. More details on how we used Rocchio's feedback and how we implemented the order generator can be found in step 5. We also just terminate (without printing anything else) if the desired precision is reached.

External Libraries:

1. **sys:** In order to work with command-line arguments
2. **html:** The library for working with HTML documents, we use the unescape function to print out the text written in html in an appropriate way.
3. **sklearn:** The machine learning library, we take advantage of the CountVectorizer to transform the words into vector space, which then we use to generate all orders of the query terms to pick the order with the highest ngrams.
4. **itertools:** We call a function from this library to generate all the possible permutations of a query
5. **googleapiclient:** In order to interact with the Google Search API.

5. **Query modification method:**
   Our query modification method uses two components: our own implementation of Rocchio's feedback algorithm (to determine the two new keywords added in each iteration) and a function orderGenerator (to determine the order of the keywords). Our implementation of Rocchio's feedback algorithm follows these main steps:
   (1) Create a variable final_vector to hold the final query vector. It's initially defined as a copy of our lexicon, which holds all 0s as values and all seen terms as keys. This is meant to represent a vector of all 0s initially, with one element per key.
   (2) Find the summation vector of the relevant documents, represented as a dictionary where keys are relevant terms and values are term frequencies
   (3) Find the summation vector of the non-relevant documents, represented as a dictionary where keys are nonrelevant terms and values are term frequencies
   (4) Divide each value in the relevant summation vector by the size of the set of relevant documents, to obtain the average relevant document vector, and then multiply these values by the beta coefficient of 0.75. This choice of beta coefficient is from the paper "Relevance Feedback in Information Retrieval," 1983. Then, add each of those elements to the final_vector.

(5) Divide each value in the nonrelevant summation vector by the size of the set of nonrelevant documents, to obtain the average nonrelevant document vector, and then multiply these values by the gamma coefficient of 0.15. Then, subtract each of those elements from the final vector. If a value is negative, assign the value as 0 instead. This choice of gamma coefficient is from the paper "Relevance Feedback in Information Retrieval," 1983.

(6) We are left with our final_vector, a dictionary with all keys from the lexicon, mapped to values that represent the result of the computation from Rocchio's feedback algorithm.

(7) Obtain the keys associated with the max two elements.

(8) Add these keys to the current list of query terms, to be reordered with the orderGenerator method.

Our process for determining the new query term order follows these main steps:

(1) The function orderGenerator() takes as input the new list of query terms derived from Rocchio's algorithm and the list of relevant documents, represented as strings. It then outputs the reordered query list, where the order is most frequently seen in the relevant documents (nonrelevant documents are not considered for this step).

(2) Use the sklearn.feature_extraction.text package from the scikit-learn library to initialize a CountVectorizer for unigrams, bigrams, and trigrams — we call this our engine_vectorizer. We then fit that object on the relevant documents. This will represent the relevant documents as a vector where the dimensions of the vector corresponds to the document's lexicon, or the unique words seen in the document. We do this referring to the sklearn documentation for CountVectorizer: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html.

(3) Generate all possible orders, or permutations, of the terms in our new query list using the itertools.permutations() function. This is because we will need to consider all possible orders of the query, to evaluate them according to their frequency in our relevant documents. We referred to the section on itertools.permutations here: https://docs.python.org/3/library/itertools.html.

(4) Join the query terms list into a single string for each query order. Then, use the engine_vectorizer object to map the strings to vectors using the transform() function: https://scikit-learn.org/stable/modules/feature_extraction.html. Append each transformation to our n-grams list.

(5) Find the count of the number of n-grams found in the relevant documents, for each query order generated in Step 3.

(6) Get the index of the query order (from the list generated previously) that maximizes the n-gram count from the relevant documents.

(7) Return the query order corresponding to this index of the query_order list. At the very end, map this list query order back to a string so that it can processed properly in the API call in the next iteration.

6. **Google Custom Search Engine JSON API Key:**
   AIzaSyDugAgxKOHTfHRymoH7-ZyUjANV5vKW01Q
   Engine ID: a33a167b34969b3a8