Report

Assignment-1

CS F469 INFORMATION RETRIEVAL

Problem statement:

Retrieving a passage/paragraph from the sections like exclusions/inclusions for the documents (Any kind of documents like word/PDF/Image). Each policy documents have multiple sections like inclusions, exclusions, conditions, definitions, extensions, covered sections, and so on. Each document must be extracted with its text information. From the policy documents, the section's (Already mentioned above) entire passage/paragraph needs to be retrieved depending on the query.

TEAM

Siddhartha Kankipati – 2020B4A70841H

Pranshul Bhatnagar – 2019A1PS0886H

Shlok Mongia – 2019B2A71527H

Nikhil Raj – 2020A7PS0093H

Application Architecture

Our IR Model provides a basic implementation for retrieving a passage/paragraph information for a specific section from a list of documents. The model architecture consists of the following components: a text extractor, a text preprocessor, a paragraph tokenizer, an inverted index, a query preprocessor, and a scoring function.

Text Extractor: This function traverses all the directories and subdirectories in the parent folder. For each directory, it loops through all the files and checks the file type (whether it is PDF/word/Image). Then, it gets the full path of the file and uses different libraries, such as PyPDF2, docx2txt, pytesseract, etc., to extract text and return them into a single string.

Text Preprocessor: This function cleans and tokenizes the text before adding them to the inverted index using the NLTK library. The text preprocessor consists of the following sub-components: a tokenizer, a stopword remover, and a lemmatizing function. Word tokenizer and sentence tokenizer splits the text into words and sentences. The stopword remover removes common words of the English alphabet, and the lemmatize function reduces words to their root form (e.g., "running" becomes "run").

Paragraph tokenizer: The documents used for this model have a very unstructured textual form(irregular paragraph breaks), so we decided to tokenize five sentences into a paragraph. The paragraph tokenizer takes extracted text as input and tokenizes the text into paragraphs by combining every five sentences.

Inverted Index: We used an inverted index implemented as a hash table where each term points to a linked list of paragraph IDs and their term frequencies. The inverted index also stores each term's Document Frequency(DF) & Inverse Document Frequency(IDF).

Query Preprocessor: It performs the same task as the text preprocessor on the query given by the user, tokenizes it, removes stopwords, and applies a limiting function.

Scoring Function: We have used the cosine similarity measure to calculate the relevance score between the query and each paragraph. The scoring function calculates the weight of each term in the query and then calculates the dot product of the query terms and each paragraph. Then we calculate the cosine similarity between the query and each document considering the frequency of the terms in both the query and the document and their overall frequency in the corpus. The resulting scores are ranked in descending order to provide the user with the most relevant documents.

Cosine Similarity: Cosine similarity is a metric used to measure the similarity of two vectors. Specifically, it measures the similarity in the vectors' direction or orientation, ignoring differences in their magnitude or scale. Both vectors must be part of the same inner product space, producing a scalar through inner product multiplication. The cosine of the angle between them measures the similarity of the two vectors.

Cosine similarity is beneficial for applications that utilize sparse data, such as word documents, transactions in market data, and recommendation systems, because cosine similarity ignores 0-0 matches. Counting 0-0 matches in sparse data would inflate similarity scores. The decision as to which metric to use depends on the particular task that we have to perform. Tasks such as retrieval of the most similar texts to a given document generally perform better with cosine similarity measure than others like euclidean distance measure.

Data Structures Used

1. “**Hash Map**” stores inverted indices, where the unique terms are the keys, and the posting lists, which consist of the paragraph they belong to and their frequency, are the values.

2. “**Lists**” store tokenized sentences and paragraphs collection. Lists are the most used container in this assignment.

3. **“Inverted-Index”** is a data structure that is used instead of an Index Matrix to save on space. Inverted-Index is a high-level implementation of a “Hash Map.”

Normalization

Normalization refers to transforming text into a standard format or structure. This can involve various steps, such as converting text to lowercase, removing punctuation, and expanding contractions. Normalization aims to reduce variation in the text and make it easier to process and analyze.

Word-level tokenization involves breaking down the text into individual words or word-like units, using whitespace or punctuation marks as delimiters.

Sentence-level tokenization involves breaking down the text into individual sentences, using punctuation marks or other indicators as delimiters.

Removal of Stop Words: These words are considered noise in the data and can often be safely ignored without affecting the text's overall meaning.

Lemmatization: The goal is to normalize words to a standard form that can be easily compared and analyzed.

Query Search

We have created the following functions to search a query :

cosine_similarity(inverted_index, paragraphs, query): It calculates cosine similarity between a query and a collection of documents represented by an inverted index. The function first calculates the weight of each term in the query using the term frequency (TF) scheme, then computes the query vector's length. It then iterates over the terms in the query and, for each term, retrieves the set of documents that contain it and its IDF value from the inverted index. It then calculates the dot product of the query vector and

the document vectors, weighted by the IDF values, and stores the results in a dictionary. Next, the function computes the length of each document vector and the cosine similarity between the query vector and each document vector and stores the results in a dictionary. Finally, the function returns the top five paragraphs with the highest cosine similarity score with the query and the sorted similarity scores for all the documents.

PhraseQuery(sentences, query): A function that takes in a list of paragraphs and returns a list of paragraphs that contain the exact phrase

AdvanceBOOLQuery(sentences, query): This function takes in a list of paragraphs and queries as input and returns the paragraphs which satisfy the **BOOLEAN query**.

Time Dependency

The program's runtime is contingent on two factors: the size of the input documents and the length of the target section and query. The initial tokenization step necessitated dividing the documents into paragraphs, which took **13.5 milliseconds** for the specified dataset. Preprocessing involved a variety of measures, including removing special characters and digits, converting text to lowercase, splitting text into words, lemmatizing each word, and then joining the words together again as a string. For the provided dataset, preprocessing took **7.6 seconds**. During the retrieval step, the program loops through each document, break it down into sections and determines the match score between the query and section text using cosine similarity. The time required for searching is influenced by the size of the inverted index, the complexity of the search query, and the hardware used to run the program. In this example, a small index and simple queries are employed, resulting in a very short search time of approximately **0.281 seconds**. However, the search time may be considerable in real-world situations with large indexes and complex queries, necessitating techniques such as approximate nearest neighbor search to expedite the process.

Performance of the Search Engine

Precision and recall are two measures that are commonly used for assessing a search engine's performance. Precision measures the proportion of relevant documents among the documents retrieved by the search engine. Recall measures the proportion of relevant documents retrieved by the search engine among all the relevant documents. However, we need a set of relevant documents for each query to calculate precision and recall. We do not have a set of relevant documents, so we cannot calculate precision and recall. To create a list of documents relevant to a particular query, we need the subject expert to rank the documents in order of importance to a query.

Therefore, to evaluate the performance of our search engine, we have used a metric called MRR (Mean Reciprocal Rank). The MRR (Mean Reciprocal Rank) measures the position of the first relevant document within a ranking. A value close to 1 indicates that relevant results are near the top of the search results, which is desirable. Conversely, lower MRR values indicate poorer search quality, with relevant results farther down the list. For a Q number of queries, the mean reciprocal rank is given by:

$$\mathrm{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{\mathrm{rank}_i}$$

If no correct answer was returned in the query, then the reciprocal rank is 0. Our search engine has an MRR of **0.8239,** indicating that the top-ranked result will likely be relevant to the user's query.

In evaluating our model, we utilized MRR instead of precision or recall. Precision and recall require knowledge of all relevant documents to determine which are pertinent to a

given query. However, we could not calculate precision and recall since we lacked ground-truth results. To calculate the MRR, we created an Excel file named "query_file" that contained the paragraph index corresponding to each query from which the query was directly copied.