**Specification Document**

Table of content

[Part 1- work-flow plans: 3](#_Toc165142076)

[Part 1: Explanations 4](#_Toc165142077)

[Part 2- work-flow plans: 7](#_Toc165142078)

[Part 2: Explanations 12](#_Toc165142079)

[Part 3- work-flow plans: 14](#_Toc165142080)

[Part 3: Explanations 15](#_Toc165142081)

# Part 1- work-flow plans:

Plan for task 1:

1. Convert the json to a df.
2. Create Word List named “tokenized\_docs” variable that contains all the words from title + abstract for each document and tokenize all the words in it.

Title

Abstract

tokenized\_doc

|  |
| --- |
| tokenized\_doc |
| ........................ |

Then, create a df named df\_tokenized\_docs that will contains all the tokenized words for each document.

1. For phase a, create a dictionary of all the words in corpus with **key=word**, **value=num of repetitions** in the corpus.

Then, output all the words that appear at least ones.

1. For phase b, do the next things:

* Apply the phase and store the results in a variable.
* Create a df for the phase with columns “AllWordsInDoc” + phase definition like this:

|  |  |
| --- | --- |
| AverageUniqueWordSum | AllWordsInDoc |
| ........................ |  |

1. For phase c, base the calculation on the dictionary from phase a and present the results in df for 20 most frequent and 20 most rarest like this:

|  |  |
| --- | --- |
| Frequency | Word |
| ........................ |  |

Plans for task 2:

1. Copy the “tokenized\_docs” to a new one called “PreproccecedList” and preform the preprocessing steps:

* Removal of Punctuations
* Removal of Stopwords
* Lemmatization & POS Tagging
* Stemming

1. Preform again steps 3 and 4 from question 1 plans.
2. Show both final df’s before and after the preprocessing.

Plans for task 3:

1. Explain the lemmatization and steaming techniques.
2. Apply lemmatization over the “PreproccecedList” and show the results.
3. Apply steaming over the “PreproccecedList” and show the results.

# Part 1: Explanations

1. D) In my code, I arranged words based on how often they appeared in many documents across the corpus. I listed the top 20 and bottom 20 words separately. Then, I used sorting to organize the words by their frequency and then selected the desired sets. This helped me see which words were common and which were rare, making it easier to identify important terms and outliers.

After examining the data, I found that common words were mostly stopwords and punctuations, while rare words included float-type numbers.

1. B) After cleaning and preprocessing the corpus, I noticed these significant changes:

* The total of unique words decreased from 325745 to 219869. A total of 105876 characters and words were removed.
* The most 20’s frequent words are medical and biological terms like: cell, dna, gene, human, tumor and etc.
* The most 20’s rarest words are still numbers and odd terms that I don’t know yet.

1. The ways to improve the words in the corpus was applying the stemming and lemmatization&POS tagging over the tokenized words and by that some words were “united” to it’s root form.

Moreover, I discovered that employing POS tagging alongside lemmatization boosted accuracy. Various parts of speech have distinct base forms. For instance, "fly" can function as a verb ("flying") or a noun ("fly"). By tagging words with their POS before lemmatizing, I ensured that the correct base form was selected for each word, depending on its usage. This refinement enhanced the precision of our text analysis.

Example of words that morphed after applying these techniques from the document indexed 2995:

* tumorigenic>tumorigen (this may be meaningless in English but it can be understood as a medical term and maybe a basic of autocomplete search engine).
* suggested>suggest
* clones>clone

1. Bonus:

A: Utilizing WordNet, Elasticsearch, or semantic search can be an effective approach for finding similar words with the same meaning. Here's how each method can be employed:

WordNet: WordNet is a lexical database of the English language that groups words into sets of synonyms called synsets, provides short, general definitions, and records the various semantic relations between these synonym sets. Using WordNet, you can find synonyms, hypernyms (more general terms), hyponyms (more specific terms), and other related words for a given word.

Elasticsearch: Elasticsearch is a powerful search engine that can be used to index and search through vast amounts of textual data. By leveraging Elasticsearch, you can create custom indexes of your text corpus and utilize its search capabilities to find similar words based on various criteria such as similarity score, word frequency, or semantic context.

Semantic search: Semantic search involves understanding the meaning of words and phrases in a search query to retrieve relevant results. Techniques such as natural language processing (NLP), word embeddings, and semantic similarity measures can be applied to perform semantic search. This approach enables finding similar words not only based on lexical similarity but also on semantic relatedness.

B: Finding similar words with the same meaning offers several benefits:

* Improved search relevance: When users search for a particular term, providing them with a list of similar words enhances search relevance by offering alternative options that might better match their intent.
* Enhanced user experience: By presenting users with a variety of similar words, similarly to the previous reason, you can help them discover alternative vocabulary that they may not have considered initially, enriching their experience and aiding in communication.
* Diverse content generation: For content creators, having access to similar words allows for more diverse and engaging content generation. It helps avoid repetitive language use and fosters creativity in writing.
* Language learning and comprehension: For language learners, exploring synonyms and related terms aids in understanding nuances in meaning and usage, thereby facilitating language comprehension and vocabulary expansion.
* Semantic understanding: By finding similar words based on semantic relationships, rather than just lexical similarity, you gain deeper insights into the connections between words and concepts, contributing to a better understanding of language semantics.

# Part 2- work-flow plans:

Plan for task 2:

Before the implementation of the different searches, I need prepare the queries

to facilitate our further tasks by providing a structured and tokenized representation of the queries (as I did to the corpus in part 1).

1. Convert the json file of the queries to a df.
2. Create Word List named “tokenized\_queries” variable that contains all the words from title + context for each query and tokenize all the words in it.

Title

Context

tokenized\_queries

|  |
| --- |
| tokenized\_query |
| ........................ |

Then, create a df named df\_tokenized\_queries that will contains all the tokenized words for each query.

Preprocessing and cleaning the queries:

This step is important because it ensures that the queries' data undergoes uniform treatment, aligning with the pre-processing conducted on the corpus. This alignment is imperative to avoid missing words during subsequent analysis tasks such as binary search and TF-IDF calculations.

1. Copy the “tokenized\_queries” to a new one called “df\_preprocessed\_queries” and preform the preprocessing steps:

* Removal of Punctuations
* Removal of Stopwords
* Lemmatization & POS Tagging
* Stemming

Plans for task 2:

Before starting with the searches methods, I need create the dictionary set of all the unique word in the corpus (the dictionary vector).

1. Binary search implementation:

* Create a function ”compute\_binary\_search\_df” that calculates word counts for all documents in a corpus and returns the results in a DataFrame.
* Initialize variables: Initialize an empty list word\_counts\_list to store word counts for each document.
* Iterate through each Document: Loop through each document (our bag of words) in the corpus.
* Initialize word counts dictionary: For each document, create an empty dictionary word\_counts with keys from the unique\_words\_set and initial values set to 0.
* Count words: Split each document into words and iterate through them.

Check if each word exists in the unique\_words\_set.

If a word is in the unique\_words\_set, set its count to 1 in the word\_counts dictionary.

* Append the word counts to a list: Append the word\_counts dictionary to the word\_counts\_list.
* Create DataFrame based on the list: Convert the word\_counts\_list into a DataFrame df\_word\_counts.
* Return DataFrame: Return the DataFrame df\_word\_counts containing word counts for all documents.

Apply the same function over the queries.

1. Co-occurrence search implementation:

* Create a function “compute\_co\_occurrence\_df” that calculates word co-occurrences for all documents in a corpus and returns the results in a DataFrame.
* Initialize variables: Initialize an empty list word\_counts\_list to store word co-occurrences for each document.
* Iterate through each document: Loop through each document (our bag of words) in the corpus.
* Initialize word counts Dictionary: For each document, create an empty dictionary word\_counts with keys from the unique\_words\_set and initial values set to 0.
* Count the word’s co-occurrences: Split each document into words and iterate through them. Check if each word exists in the unique\_words\_set. If a word is in the unique\_words\_set, increment its count in the word\_counts dictionary.
* Append the word co-occurrences to List: Append the word\_counts dictionary to the word\_counts\_list.
* Create DataFrame: Convert the word\_counts\_list into a DataFrame df\_word\_counts.
* Return DataFrame: Return the DataFrame df\_word\_counts containing word co-occurrences for all documents.

Also here, apply the same function over the queries.

1. There are 2 main options that I may use:
2. TF-IDF implementation using libraries (The chosen option):

* TF implementation:
* Create a function called tf\_calculation to calculate the term frequency (TF) values for each document in the corpus and return the results as a DataFrame.
* Initialize variables: Initialize an empty DataFrame tf\_df with columns representing unique words from the provided list unique\_words\_list.
* Select documents: Assign the entire corpus to the variable documents.
* Initialize TF-IDF Vectorizer: Initialize a TF-IDF vectorizer tf\_vectorizer with the parameter use\_idf set to False and the vocabulary set to the unique\_words\_list.
* Fit and Transform: Fit the TF-IDF vectorizer to the documents and transform them to calculate the TF values.
* Convert to DataFrame: Convert the TF values (tf\_values) to an array and then to a DataFrame and then, set the column names of the DataFrame.

Return DataFrame: Return the DataFrame tf\_df containing the calculated TF values for each document.

Apply the same function over the queries.

* IDF implementation:

This function is very similar in implementation to the tf\_calculation function, but with a small changes.

* Create a function called idf\_calculation to calculate the inverse document frequency (IDF) values for each term in the corpus and return the results as a DataFrame.
* Initialize variables: Initialize an empty DataFrame idf\_df with columns representing unique words from the provided list unique\_words\_list.
* Select documents: as in the tf\_calculation, assign the entire corpus to the variable documents.
* Initialize TF-IDF Vectorizer: Initialize a TF-IDF vectorizer idf\_vectorizer with the parameter use\_idf set to True and the vocabulary set to the unique\_words\_list.
* Fit and Transform: Fit the TF-IDF vectorizer to the documents and transform them to calculate the IDF values.
* Convert to DataFrame: Convert the IDF values (idf\_values) to an array and then to a DataFrame and then, set the column names of the DataFrame.
* Return DataFrame: Return the DataFrame idf\_df containing the calculated IDF values for each term.

Also here, apply the same function over the queries.

* Complete TF-IDF implementation:
* Create a function named tfidf\_calculation to calculate the TF-IDF values for each term in the corpus and return the results as a DataFrame.
* Initialize variables: Initialize an empty DataFrame tfidf\_df with columns representing unique words from the provided list unique\_words\_list.
* Select documents: Assign the entire corpus to the variable documents.
* Initialize TF-IDF Vectorizer: Initialize a TF-IDF vectorizer tfidf\_vectorizer with the parameter use\_idf set to True and the vocabulary set to the unique\_words\_list.
* Fit and transform: Fit the TF-IDF vectorizer to the documents and transform them to calculate the TF-IDF values.
* Convert to DataFrame: Convert the TF-IDF values (tfidf\_values) to an array and then to a DataFrame and then, set the column names of the DataFrame.
* Return DataFrame: Return the DataFrame tfidf\_df containing the calculated TF-IDF values for each term.

Also here, I’ll apply this over the queries.

1. Manual TF-IDF implementation plan (I’m not sure if I’ll use it because it has a very long runtime):

* TF implementation:
* Create a function called computeTF3 to compute the term frequency (TF) values for each document in a collection and return the results as a DataFrame.
* Initialize variables: Initialize an empty list tf\_results to store the TF values for each document.
* Iterate over the documents: Iterate through each document in the documents collection.
* Initialize document TF Dictionary: For each document, create an empty dictionary tf\_doc to store the TF values for each unique word.
* Count words: Count the total number of words in the document, then Iterate through each unique word in the unique\_words set.
* Calculate TF: Calculate the TF value for each word in the document by dividing the count of occurrences of the word by the total word count in the document.

Then, Store the TF value for each word in the tf\_doc dictionary.

* Append the results: Append the tf\_doc dictionary containing TF values for the current document to the tf\_results list.
* Create DataFrame: Convert the list of dictionaries (tf\_results) into a DataFrame.
* Return DataFrame: Return the DataFrame containing the calculated TF values for each document.

Apply also over the queries.

* IDF implementation:
* Create a function called computeIDF3 to calculate the inverse document frequency (IDF) values for each term in a collection of documents and return the results as a DataFrame.
* Initialize variables: Initialize an empty dictionary idf\_results to store the IDF values for each unique word.
* Iterate through Unique Words: Iterate through each unique word in the unique\_words set.
* Count documents containing the word: Count the number of documents in which the current word occurs. Then, Iterate through each document in the documents collection and check if the word is present in the document.
* Calculate IDF value: Calculate the IDF value for the current word using the formula: 1 + log((N + 1) / (1 + doc\_count)), where N represents the total number of documents and doc\_count represents the number of documents containing the word. Then, the calculated IDF value to the idf\_results dictionary with the current word as the key.
* Create DataFrame: Convert the dictionary idf\_results into a DataFrame.
* Return DataFrame: Return the DataFrame containing the calculated IDF values for each term.

Apply this also over the queries.

* Complete TF-IDF implementation:
* Create a function called computeTFIDF3 to calculate the TF-IDF values for each term in a collection of documents and return the results as a DataFrame.
* TF calculation: Use the computeTF3 function by calling it to calculate the term frequency (TF) values for each term in the documents.
* IDF calculation: Use the computeIDF3 function by calling it to calculate the inverse document frequency (IDF) values for each term in the documents.
* Combine TF and IDF: Create a copy of the TF results DataFrame to store TF-IDF values and then, iterate through each unique word in the unique\_words set.
* Calculate the TF-IDF value: For each word, multiply its TF value (from tf\_results) by its IDF value (from idf\_results) and store the result in the TF-IDF DataFrame.
* Return TF-IDF DataFrame: Return the DataFrame tfidf\_results containing the calculated TF-IDF values for each term in the documents.

Apply this over the queries too.

# Part 2: Explanations

1. I decided to use the 3WH format, focusing on the Who, What, Where, and How aspects of the research questions.

* 3WH table for query 9:

|  |  |  |  |
| --- | --- | --- | --- |
| Who | What | When | How |
| Researchers, Scientists Physicians, Biologists, or individuals who interested in understanding the complex function of mutY in humans. | Articles about the function of mutY in humans. | Not applicable. | Exploration of the functional roles and mechanisms of mutY (also known as hMYH) in humans. |

* 3WH table for query 12:

|  |  |  |  |
| --- | --- | --- | --- |
| Who | What | When | How |
| 1. Physicians and Scientists. 2. Patients or experimental model organisms with altered Smad4 expression and without it. | Identification of genes regulated by the signal transducing molecule Smad4. | Not applicable. | Characterization of gene expression patterns in Smad4 knockout mice compared to normal mice. |

* 3WH table for query 15:

|  |  |  |  |
| --- | --- | --- | --- |
| Who | What | When | How |
| 1. Scientists who work in a lab. 2. Cells, biological systems or organs that undergoing apoptosis. | Role of ATPases in apoptosis. | During apoptosis. | Investigation into the involvement and mechanisms of ATPases during apoptosis. |

* 3WH table for query 20:

|  |  |  |  |
| --- | --- | --- | --- |
| Who | What | When | How |
| Cells or biological systems where proteins are modified by covalent attachment to ubiquitin or ubiquitin-like proteins. | Identification of biological processes regulated by substrate modification by ubiquitin or ubiquitin-like proteins. | During biological process. | Examination of the impact of substrate modification by ubiquitin or ubiquitin-like proteins on various cellular\biological processes. |

* 3WH table for query22:

|  |  |  |  |
| --- | --- | --- | --- |
| Who | What | When | How |
| Cells exposed to DNA-damaging agents causing single-stranded or double-stranded DNA breaks. | Differential response of p53 family members to single-stranded versus double-stranded DNA breaks. | During DNA damage. | Investigation into the distinct responses of p53 family members to different types of DNA damage and their downstream effects. |

# **Part 3- work-flow plans**:

The first thing I need to do is to convert the qrel\_14 (the gold standard) to a DataFrame for later usage.

Plan for building the bases of the search model:

1. Computing the Cosine Similarity: Compute cosine similarity between a query TF-IDF vector and a corpus TF-IDF matrix.

I’ll do it by Using the cosine\_similarity function to calculate cosine similarity between the query TF-IDF vector and each document's TF-IDF vector in the corpus. And then, return a 1D array of cosine similarity scores.

1. Transform Cosine Similarities: After I tested the first function, I didn’t feel comfortable with it’s presentation. That’s why I want to transform the cosine similarity scores into a readable representation for each query by Iterating through each query in the queries DataFrame and retrieve the cosine similarity scores for the current query. Then, build a representation string for the query, including its ID and similarity scores, store each representation in a dictionary with the query ID as the key and finally return the dictionary of transformed results.
2. Map the scores to the documents (individually) and sort: Map cosine similarity scores to corresponding document IDs, filter out zero scores, sort scores, and extract top 100 documents by Iterating through the transformed results dictionary. Then, Extract the document scores from the representation and convert them into a dictionary mapping document IDs to scores. Next filter out documents with zero scores and sort the document-score mappings in descending order. After it, extract the top 100 document IDs and update the representation with sorted document IDs. Store the mapped representation in a dictionary with query IDs as keys. Finally, return the dictionary of mapped results.

Later, I want to create the search model function that will be based on these 3 parts.

Plan for evaluation stage:

Evaluate the performance of a retrieval system by comparing predicted relevant documents to gold standard (the qrel\_14 JSON file) relevant documents.

Firstly, Iterating through each row in the predicted DataFrame. For each row, extract the query ID and the set of predicted relevant documents and then check if there are matching rows in the gold standard DataFrame for the current query. If relevant documents are found in gold\_df Then It will Extracting the relevant documents for the current query from gold\_df.

Next, create binary arrays representing the presence or absence of each document ID in both the predicted and gold relevant document sets.

Afterwards, calculating evaluation metrics such as precision, recall, and F1-score using the precision\_score, recall\_score, and f1\_score functions and Storing the evaluation results (precision, recall, and F1-score) in a dictionary with the query ID as the key and Print the details of the evaluation for each query if desired.

Finally, Return a dictionary containing evaluation results for each query. If no relevant documents are found in gold\_df for a query, print a message indicating that.

# Part 3: Explanations

1. A. i) I don’t think that binary search is much applicable for the queries and corpus format due to the next reasons:

* In binary search, you can’t understand from the results the relevance of a word from the dictionary to a document and question, but only if it’s appeared in it.
* Binary search may not be suitable for all types of data structures or search scenarios. It is most effective when dealing with sorted data sets and may not be as efficient in dynamic or unsorted environments.
* For more optimized results, it is crucial to preprocess and clean the data before preforming this search.

However, there is one advantage, although I don’t think it is dominant in our project (compared to working with TF-IDF for example):

* Efficiency, Binary search is pretty efficient and has a good runtime (based on my experience with working on our corpus and the queries). It reduces the time complexity of searching for words in dictionaries, leading to faster retrieval of information.

ii) In this project, the way I implemented binary search was to iterate over each document in the corpus or query in queries file, and check if each word exists in the dictionary set, if it is exists, the value will be 1, otherwise 0. I honestly don’t see any way to improve it. But from what I did in this project, I can tell that the preprocessing\cleaning stage was extremely important, and in my opinion optimized the binary search process by its running time complex because many irrelevant words and characters were dropped off during the cleaning stage.

1. C. Analyzing the evaluation results:

* For each query:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Query id** | **Precision** | **Recall** | **F1-score** | **Results analysis** |
| **9** | 0.6 | 1.0 | 0.749 | This query got the precision value of 0.6, indicating some irrelevant documents were found, while the recall is perfect, 1.0, meaning all relevant documents were found (high sensitivity). The F1-score, reflecting overall performance, stands at 0.75, indicating reasonably good results. |
| **12** | 0.81 | 1.0 | 0.895 | For this query, the precision value is high at 0.81, indicating that most retrieved documents were relevant. The perfect recall score of 1.0 means that all relevant documents were retrieved (high sensitivity). The F1-score, at 0.895, signifies an excellent overall performance. |
| **15** | 0.49 | 1.0 | 0.657 | For this query, the precision is 0.49, a poor result indicating a significant number of irrelevant items among the retrieved ones. However, the recall is perfect at 1.0, meaning all relevant items were retrieved (high sensitivity). The F1-score, at 0.658, suggests a moderate overall performance. |
| **20** | 0.65 | 1.0 | 0.787 | This query, achieved a precision of 0.65, indicating a reasonable proportion of relevant documents among those retrieved. Perfect recall of 1.0, it successfully retrieved all relevant documents (high sensitivity). The F1-score of 0.788 reflects pretty good overall performance. |
| **22** | 0.97 | 1.0 | 0.984 | This query, attained an exceptionally high precision of 0.97, indicating that almost all retrieved documents were relevant. Additionally, it achieved a perfect recall, retrieving all relevant documents (high sensitivity). The F1-score of 0.985 demonstrates an outstanding overall performance. |

* Average of the queries evaluation results:

|  |  |  |
| --- | --- | --- |
| **Precision** | **Recall** | **F1-score** |
| 0.704 | 1.0 | 0.815 |

Average precision: The average precision across all queries is 0.704, indicating that about 70.4% of the retrieved documents were relevant on average. This suggests the system is proficient at finding relevant documents while minimizing irrelevant ones.

Average recall: The average recall across all queries is 1.0, demonstrating that all relevant documents were retrieved for each query on average. This perfect recall underscores the system's effectiveness in not missing any relevant documents, essential for a robust search process (high sensitivity).

Average F1-score: The average F1-score across all queries is 0.815, representing a balanced measure of precision and recall. This score suggests an overall effective retrieval performance, with a good balance between precision and recall.

In conclusion, these results indicate that the search process performs well, consistently retrieving all relevant documents, maintaining a good balance between precision and recall, and effectively meeting users' needs in accessing relevant information.

1. Yes! In did!

This project of building and developing the basics of search engine model and it’s final model can be useful to me in many ways in future projects. Firstly, during this course and while working on the project I learned how to work with textual data and how to understand the methods behind search engines. Secondly, as a future data analyst, it is very important for me to know how to work with documents and textual data in addition to data stored in a table for instants. Practically nowadays, it is important to be familiar with extracting data efficiently and as a student, it helped me with apply some of the methods while searching medical documents for different uses. Moreover, I don’t know if this idea is implemented somewhere, but for example, the search model of the project can be used as a root of developing a browser plugin that can rate the reliability of retrieved results of medical question on different search engines such as google, yahoo and etc, and it can be used by patients, doctors and anyone who need a medical information online.