**Developing a Naïve Search Engine Using MapReduce Technology**

Assignment 02

22i-1865 Aqsa Fayaz

22i-1940 Manal Aamir

22i-1868 Muhammad Malik

**Introduction:**

In order to create a naïve search engine capable of processing large volumes of data we’ll follow the following steps:  
**Dataset Preparation:**

We'll start by dividing the 5 GB Wikipedia dataset into smaller, manageable chunks to facilitate easier processing and analysis.

**Data Preprocessing:**

Our code will clean and standardize the text data, removing stopwords, and normalizing terms for consistency across the dataset.

**TF-IDF Score Calculation:**

We will calculate Term Frequency (TF) and Inverse Document Frequency (IDF) scores to evaluate the importance of words within documents relative to the entire dataset.

**Vector Space Model Implementation:**

This step involves coding a model to represent both documents and queries as vectors, enabling us to measure similarities for ranking purposes.

**Developing the Search Engine with MapReduce:**

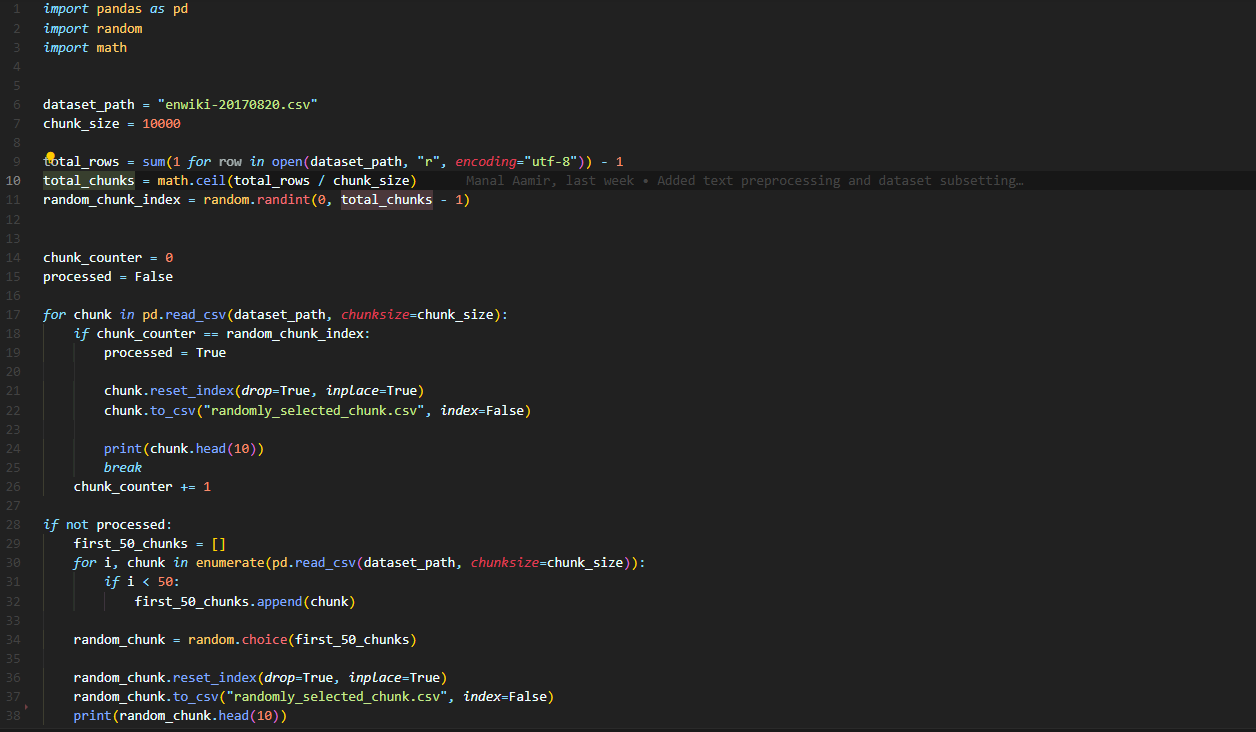
* Word Enumeration: Our code will scan the dataset to identify unique words, assigning each a unique identifier.
* Document Count: We'll compute the IDF for each term, essentially counting the documents each term appears in.
* Indexing: The script will process each document to create a TF/IDF vector representation, forming the basis of our search index.
* Query Processing: We'll develop functions to convert user queries into vectors and find the most relevant documents by comparing these vectors with our document index.

**1- Data Preparation**

In our project, the initial step involves preparing the dataset for further processing. To manage the extensive Wikipedia dataset effectively, we employ a Python script that segments the data into more manageable pieces. This approach not only makes the dataset easier to handle but also optimizes it for the MapReduce processing framework.

**Procedure**:

* **Reading the Dataset**: The script initiates by setting a path to the dataset, **enwiki-20170820.csv**, which contains the entire Wikipedia dump. Given the vast size of this dataset, direct processing is impractical due to hardware limitations and efficiency concerns.
* **Determining Chunk Size**: We've decided on a chunk size of 10,000 rows. This size is a balanced choice that ensures each chunk is substantial enough for meaningful analysis but not so large as to hinder processing speed or efficiency.
* **Calculating Total Chunks**: The script calculates the total number of chunks by dividing the total number of rows by the chunk size. This calculation is crucial for understanding how the dataset will be broken down and for iterating over the entire dataset effectively.
* **Random Chunk Selection for Sampling**: As a unique approach to data preparation, the script selects a random chunk from the total chunks calculated. This method is particularly useful for initial testing and analysis, allowing us to work with a representative sample of the data without overwhelming our processing capabilities.
* **Fallback Strategy**: If, for any reason, the process fails to select a chunk, the script has a fallback mechanism. It compiles the first 50 chunks into a list and randomly selects one to work with. This ensures that even in edge cases, we have a manageable dataset ready for preprocessing and further analysis.
* **Output**: The selected chunk is then written to a new CSV file, **randomly\_selected\_chunk.csv**, resetting its index to ensure data consistency. This file serves as the basis for all subsequent data processing steps.



**2- Data Preprocessing**

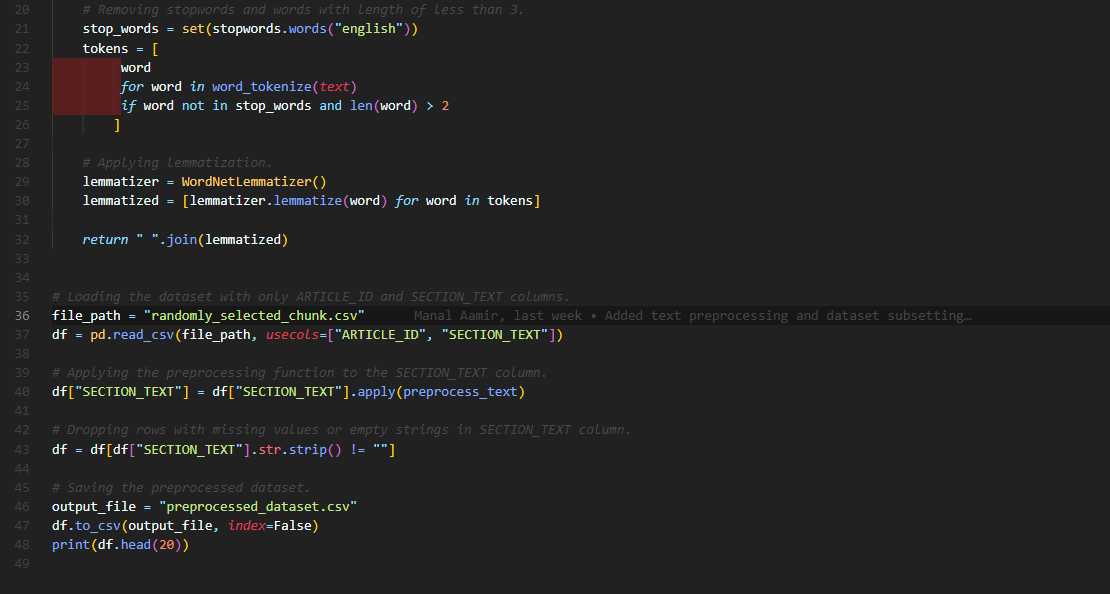
After preparing our dataset into manageable chunks, the next crucial step in our project is data preprocessing. This stage is essential for cleaning and refining the dataset, ensuring that it's in the optimal format for analysis and further processing. We utilize a Python script, **preprocessing.py**, to automate this preprocessing step.

**Key Functions of the Script**:

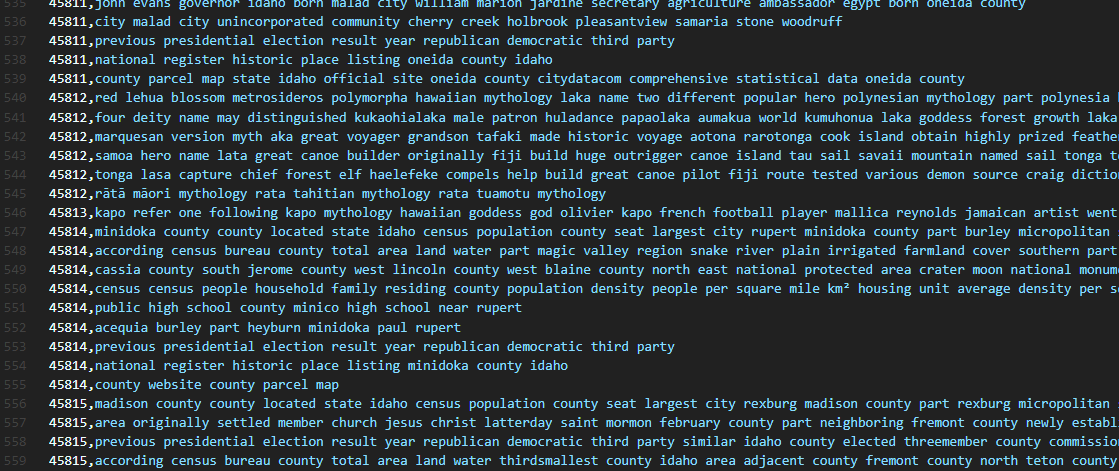
* **Text Normalization**: The script begins by converting all text to lowercase to ensure uniformity. This step is crucial for consistency, as it prevents the same word in different cases from being treated as distinct.
* **Removing Numbers and URLs**: We use regular expressions to remove numbers and URLs from the text. This is important because numbers and URLs are usually not relevant to our analysis and can skew the results if not removed.
* **Punctuation Removal**: Another regular expression is employed to strip away punctuation. Punctuation marks can interfere with text analysis, so their removal is necessary for clean data.
* **Stop Words Removal**: Common words that offer little value in understanding the text's context, known as stop words, are removed. This includes words like "the", "is", "at", which are prevalent in English but don't contribute significantly to the semantic meaning of the text.
* **Lemmatization**: The script applies lemmatization to condense words to their base or dictionary form. Unlike stemming, lemmatization considers the context of a word, leading to more accurate reductions. This process helps in reducing the complexity of the text data and standardizing variations of the same word.

**Data Cleaning Process**:

* The script loads the dataset, focusing on the **ARTICLE\_ID** and **SECTION\_TEXT** columns, as these contain the key information needed for our search engine.
* It applies the **preprocess\_text** function to the **SECTION\_TEXT** column, systematically cleaning each text entry according to the steps outlined above.
* Post-processing, rows with missing values or empty strings in the **SECTION\_TEXT** column are dropped to ensure that only complete and relevant data is retained for analysis.
* The cleaned and processed dataset is then saved to a new file, **preprocessed\_dataset.csv**, and ready for the next stages of our project.



**Sample of preprocessed\_dataset.csv:**

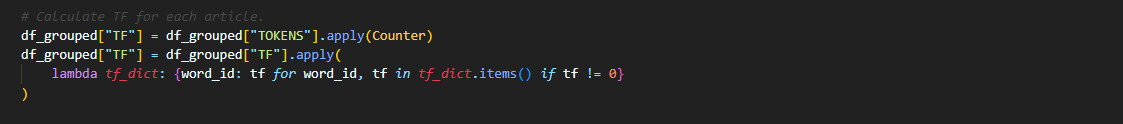


**3- TF-IDF Score Calculation and Indexing**

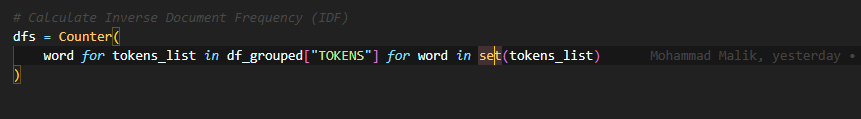
The core of our project's analytical engine is the calculation of TF-IDF scores and the creation of an indexing system that enables efficient document retrieval. The **MainProcess.py** script encapsulates this critical phase, transitioning from the preprocessed dataset to a structured form that our search engine can utilize for query processing.

**Key Components of the Script:**

* **Dataset Integration**: Initially, the script reads the preprocessed dataset from **preprocessed\_dataset.csv**, where the text has already been cleaned and normalized.
* **Article Aggregation**: To handle articles that might be split across multiple rows, the script aggregates text by **ARTICLE\_ID**, concatenating **SECTION\_TEXT** entries. This ensures that each article is treated as a single document for analysis.
* **Tokenization**: The aggregated text for each article is tokenized into words. This step transforms the text into a list of tokens that can be analyzed individually.
* **Vocabulary Construction**: From the tokens, a vocabulary set is created, comprising all unique words found in the dataset. This vocabulary serves as the basis for indexing, with each word assigned a unique identifier.
* **Term Frequency (TF) Calculation**: For each article, the script calculates the term frequency, essentially counting how many times each word appears. This frequency is stored in a dictionary that maps words (or rather, their IDs) to their counts.



* **Inverse Document Frequency (IDF) Calculation**: The script computes the IDF for each word in the vocabulary. This involves counting the number of documents each word appears in across the entire dataset. IDF reflects the importance of a word; words that appear in many documents are considered less significant than those that appear in fewer documents.



* **TF-IDF Calculation**: Combining TF and IDF, the script calculates the TF-IDF score for each word in each article. This score is a statistical measure used to evaluate the importance of a word to a document in the context of a corpus. The higher the TF-IDF score, the more relevant the word is to the specific document.

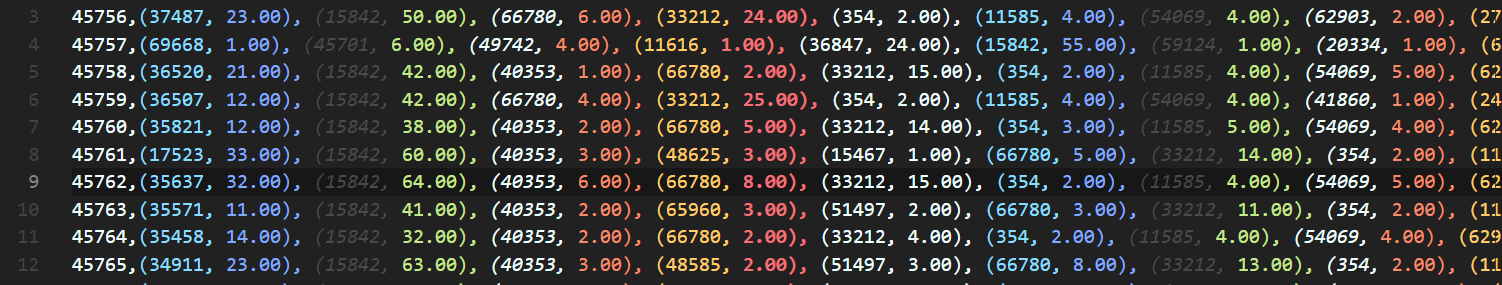


* **Output Generation**: Finally, the script outputs several CSV files, including a vocabulary list, term frequencies, document frequencies, and TF-IDF scores for each article. These files collectively form the indexed database our search engine will query against.

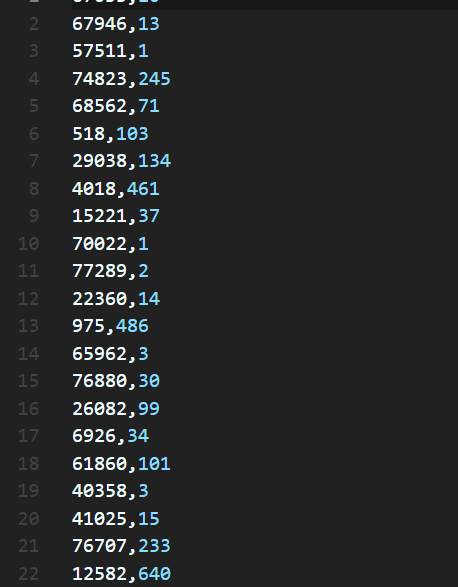
**Processing Logic**:

The script’s logic is rooted in the foundational principles of information retrieval. By calculating TF-IDF scores, it provides a quantitative basis for comparing the relevance of documents to a given search query. The indexing system not only organizes this data efficiently but also ensures that query processing can be performed quickly, even over large datasets.

**Sample of article\_term\_frequencies.csv:**

****

**Sample of article\_document\_frequencies.csv:**

****

**Sample of article\_tfidf\_scores.csv:**

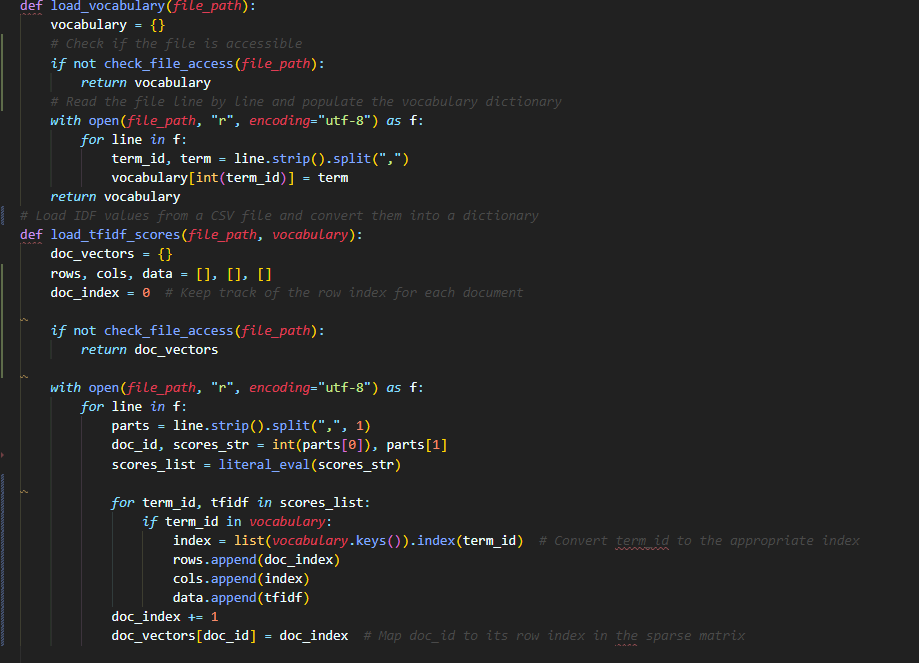


**4-Vector Space Model**

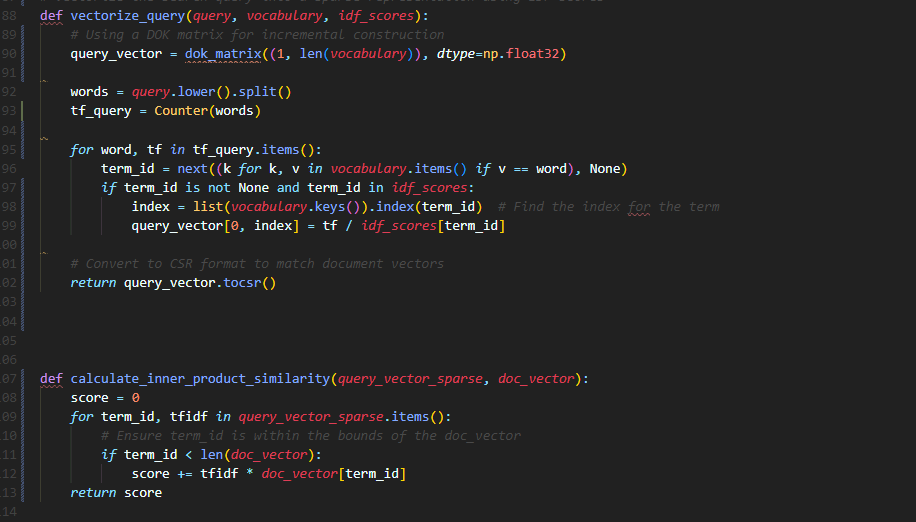
The **VectorSpaceModel.py** script demonstrates the implementation of the Vector Space Model (VSM) for our search engine, focusing on document and query processing without leveraging MapReduce. This approach allows us to understand and apply fundamental information retrieval techniques in a Python environment, using libraries like Pandas and SciPy to handle data and sparse matrices efficiently.

Overview of the Script's Functionalities:

* **File Accessibility Checks**: Ensures that necessary files, such as IDF scores and vocabulary listings, are available and accessible for the script to function correctly. This step is crucial to prevent runtime errors due to missing data.
* **Loading Data**:
  + **IDF Scores**: Imports IDF values from a specified file, converting these values into a dictionary for quick access. IDF scores help in determining the importance of terms across the document corpus.
  + **Vocabulary**: Reads a vocabulary file to map terms to unique identifiers, facilitating consistent reference to terms throughout the process.

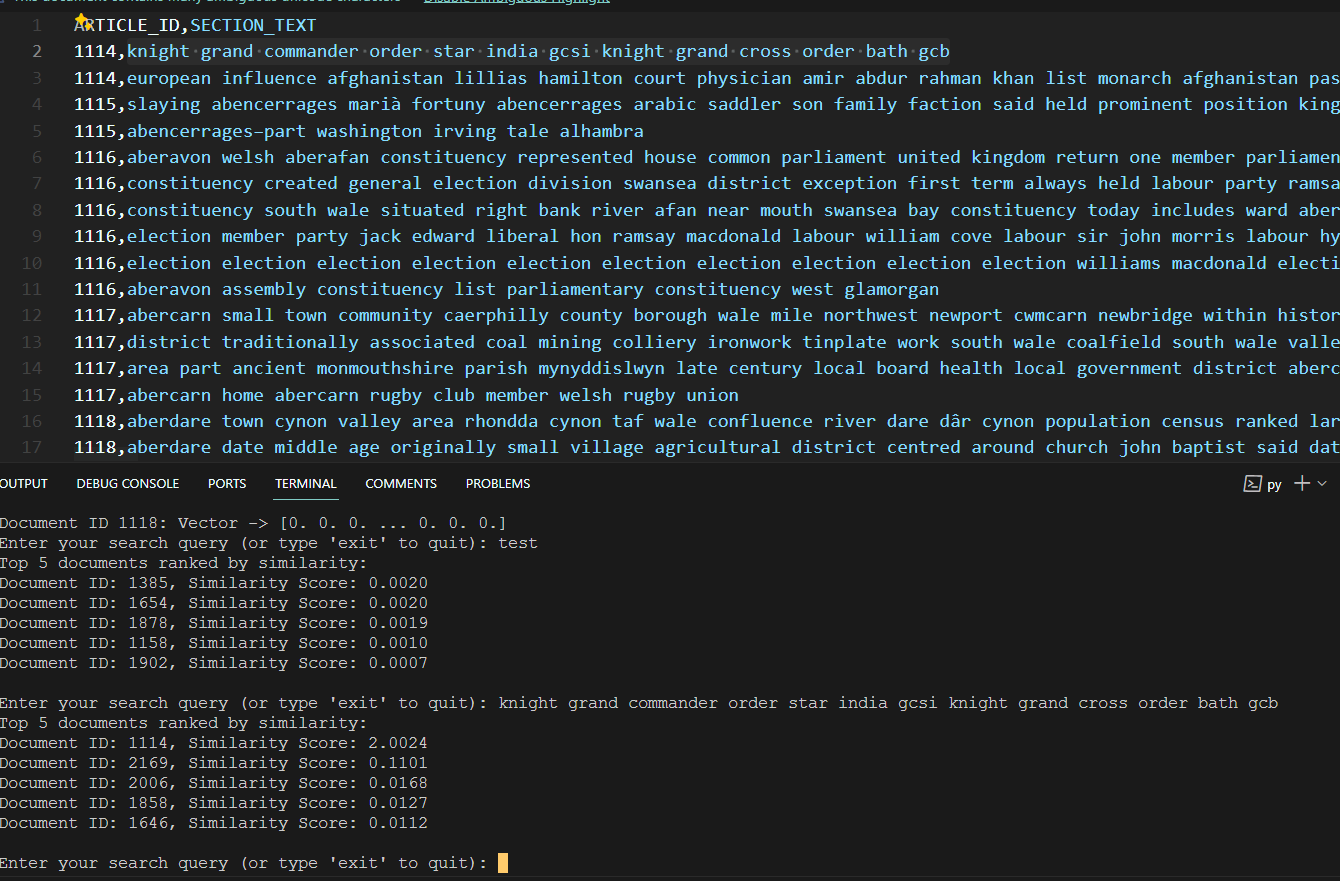


* **Constructing Document Vectors**: Utilizes TF-IDF scores to build sparse representations of documents as vectors within a high-dimensional space defined by the vocabulary. This step is fundamental to comparing document similarity.
* **Sparse Matrix Representation**: For efficiency, document vectors are stored in a Compressed Sparse Row (CSR) matrix. This format significantly reduces memory usage and computation time when dealing with large, sparse datasets.
* **Query Processing**:
  + Converts user queries into vectors using the same IDF scores and vocabulary as the documents. This vectorization involves tokenizing the query, mapping tokens to the vocabulary, and applying IDF scores.
  + Computes similarity scores between the query vector and document vectors, effectively determining the relevance of each document to the query.
* **Ranking Documents**: Based on computed similarity scores, documents are ranked to identify the most relevant ones to the user's query. The script outputs a list of top documents, ranked by their relevance.



**Implementation Steps:**

* **Preparation**: The script starts by loading the necessary components, such as IDF scores and the vocabulary, from their respective files.
* **Document Vectorization**: Each document in the dataset is represented as a vector in a sparse matrix, with TF-IDF scores indicating the significance of each term within the document.
* **Query Vectorization**: Converts a user's search query into a vector using the same method as document vectorization, allowing for direct comparison.
* **Similarity Calculation**: Computes the similarity between the query vector and each document vector to assess relevance.
* **Output**: Displays the top documents that most closely match the search query, serving as the search engine's response.



**5- Vector Space Model with MapReduce**

In a distributed computing environment like Hadoop, MapReduce enables the processing of large datasets across multiple nodes efficiently. Integrating the VSM into a MapReduce framework would involve dividing the task into smaller sub-tasks that can be executed in parallel across different nodes. Here's a conceptual overview:

**Word Enumeration and IDF Calculation with MapReduce**

* **Map Phase**: Each mapper takes a chunk of the dataset and processes it to extract words, creating key-value pairs where the key is the word and the value is the document ID. For IDF calculation, the mapper would output each word-document pair only once.
* **Reduce Phase**: Reducers receive key-value pairs sorted by key (the word). For each word, the reducer compiles a list of unique documents in which the word appears and calculates the IDF by dividing the total number of documents by the number of documents containing the word, followed by taking the logarithm of this quotient.

**TF Calculation with MapReduce**

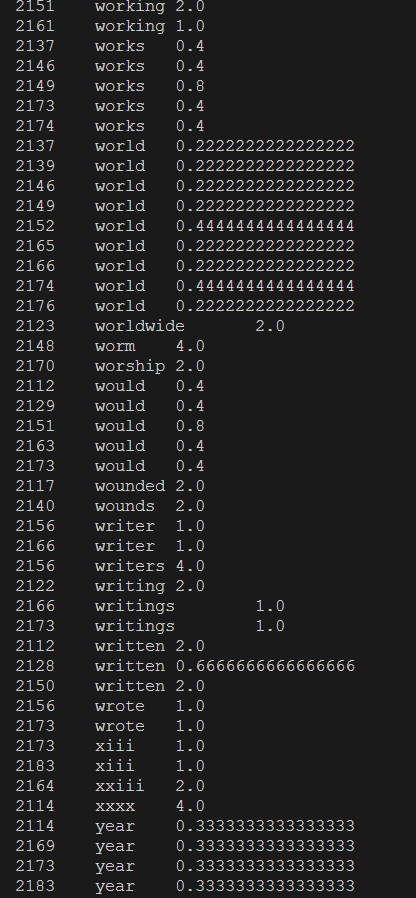
* **Map Phase**: Mappers process documents to count word occurrences, outputting key-value pairs with the word and count for each document.
* **Reduce Phase**: Reducers sum up the counts for each word-document pair, resulting in the final term frequency (TF) for each word in each document.

**TF-IDF Score Calculation**

With TF and IDF values calculated, a final MapReduce job can combine these to calculate the TF-IDF score for each word in each document. This could be done in a single reduce phase, where the reducer has access to both TF and IDF values and calculates the TF-IDF by multiplying the TF by the IDF for each word-document pair.

**Query Processing and Document Ranking**

* **Vectorization and Query Processing**: This could be handled outside the MapReduce jobs due to the interactive nature of queries. However, once a query is vectorized, its similarity to document vectors could be assessed using a MapReduce job.
* **Map Phase for Query Processing**: For each document vector, the mapper calculates the dot product between the document vector and the query vector (or another similarity measure), outputting the document ID and its similarity score.
* **Reduce Phase for Ranking**: Reducers receive the document IDs and their similarity scores, sorting them to identify the top N documents to be returned as search results.



**Conclusion:**  
In summary, our project to develop a naïve search engine using MapReduce technology provided a practical, end-to-end experience in applying big data analytics. From initial data preparation and preprocessing to implementing complex TF-IDF calculations and the Vector Space Model, each step was crucial in building a functional search engine capable of efficiently handling large datasets. This project not only enhanced our technical skills but also demonstrated the power of distributed computing in processing and analyzing big data. Looking forward, opportunities to improve the search engine abound, including algorithm refinement and dataset expansion. The project underscored the importance of collaboration and persistence, offering invaluable lessons and confidence for future challenges in data science and analytics.

Top of Form