### **1: Boolean Retrieval Model with Inverted Index**

#### **Implementation**

The goal of this part is to construct an **Inverted Index** for a document collection and implement a Boolean retrieval model to support **AND**, **OR**, and **NOT** operations in search queries. Below are the key components and steps involved in the implementation:

##### **1. Preprocessing:**

The document collection must undergo several preprocessing steps before building the index. The steps include:

* **Case Folding**: All words in the documents are converted to lowercase to avoid case-sensitive mismatches during search.
* **Tokenization**: The text is split into individual words or tokens, which serve as the basic searchable units.
* **Non-alphabetic Token Removal**: Non-alphabetic tokens such as numbers, punctuation, and symbols are removed to focus purely on the content words.
* **Stop Word Removal**: Common stop words (e.g., "the," "is," "and") are removed to reduce noise in the search results.
* **Lemmatization**: Words are reduced to their base forms (e.g., "running" becomes "run"), allowing for more consistent and accurate matching between query terms and documents.

##### **2. Inverted Index Construction:**

After preprocessing the documents, an **inverted index** is constructed. The index maps each word to the set of documents (identified by unique document IDs) where the word appears.

The inverted index construction process includes:

* For each document, preprocessing its content.
* For each word in the preprocessed content, adding the document's ID to the list of documents associated with that word in the index.

The inverted index allows efficient retrieval of documents by associating words with the documents in which they occur.

##### **3. Boolean Search Query Handling:**

The retrieval model supports Boolean search queries with **AND**, **OR**, and **NOT** operators. These queries allow users to combine search terms logically to refine the results. The system processes queries as follows:

* **AND**: Returns the intersection of the sets of documents containing each term. For example, a query like "dog AND cat" will return only the documents containing both "dog" and "cat."
* **OR**: Returns the union of the sets of documents containing any of the terms. For example, "dog OR cat" will return all documents containing either "dog" or "cat."
* **NOT**: Returns the difference between the set of documents containing a term and the set of documents containing the excluded term. For example, "dog NOT cat" will return documents containing "dog" but not "cat."

Boolean queries are parsed, and the system uses a stack to manage the logical operations between tokens. After processing the query, the resulting set of document IDs is returned as the answer.

#### **Example 1:**

Let’s assume the system processes the following Boolean query:

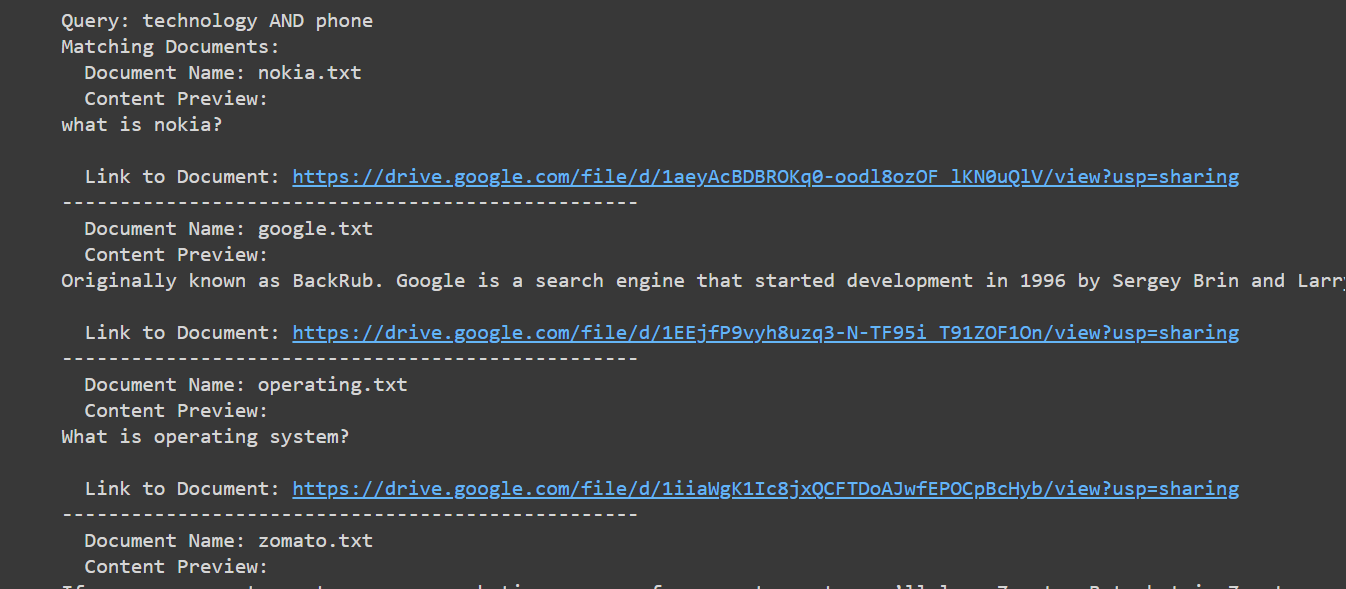
* **Query**: "technology AND phone"

Steps:

1. **Find** documents that contain both "technology" and "phone."
2. **Display** the final set of matching documents.

* **Matching Documents**:
  + nokia.txt
  + google.txt
  + operating.txt
  + zomato.txt
  + huawei.txt
  + skype.txt
  + messenger.txt

**Explanation**: The system searches for documents containing both "technology" and "phone." As a result, it retrieves documents related to companies and services involved in technology and telecommunications, such as Nokia, Google, and Skype.



#### **Example 2:**

Let’s assume the system processes the following Boolean query:

* **Query**: "deliveries OR foods"

Steps:

1. **Find** documents that contain either "deliveries" or "foods" (or both).
2. **Display** the final set of matching documents.

* **Matching Documents**:
  + flipkart.txt
  + google.txt
  + blackberry.txt
  + zomato.txt
  + swiggy.txt

**Explanation**: The system retrieves documents related to either deliveries or foods. Results such as "swiggy.txt" and "zomato.txt" likely discuss food deliveries, while "flipkart.txt" and "google.txt" may involve e-commerce or logistics.

#### **Output:**



The system returns the filenames of the matching documents from the collection, making it easier for users to locate relevant files.

#### **Conclusion:**

The Boolean retrieval model implemented in Part 1 allows users to perform complex queries using logical operators. The inverted index ensures efficient retrieval of documents matching the search criteria. The preprocessing steps ensure that the index is concise and handles variations in word forms.

### **2. Extend the Boolean retrieval model implemented in part (1) to Extended Boolean Retrieval model with following features:**

#### **a) Phrase Queries search using Biword index**

### **Implementation**

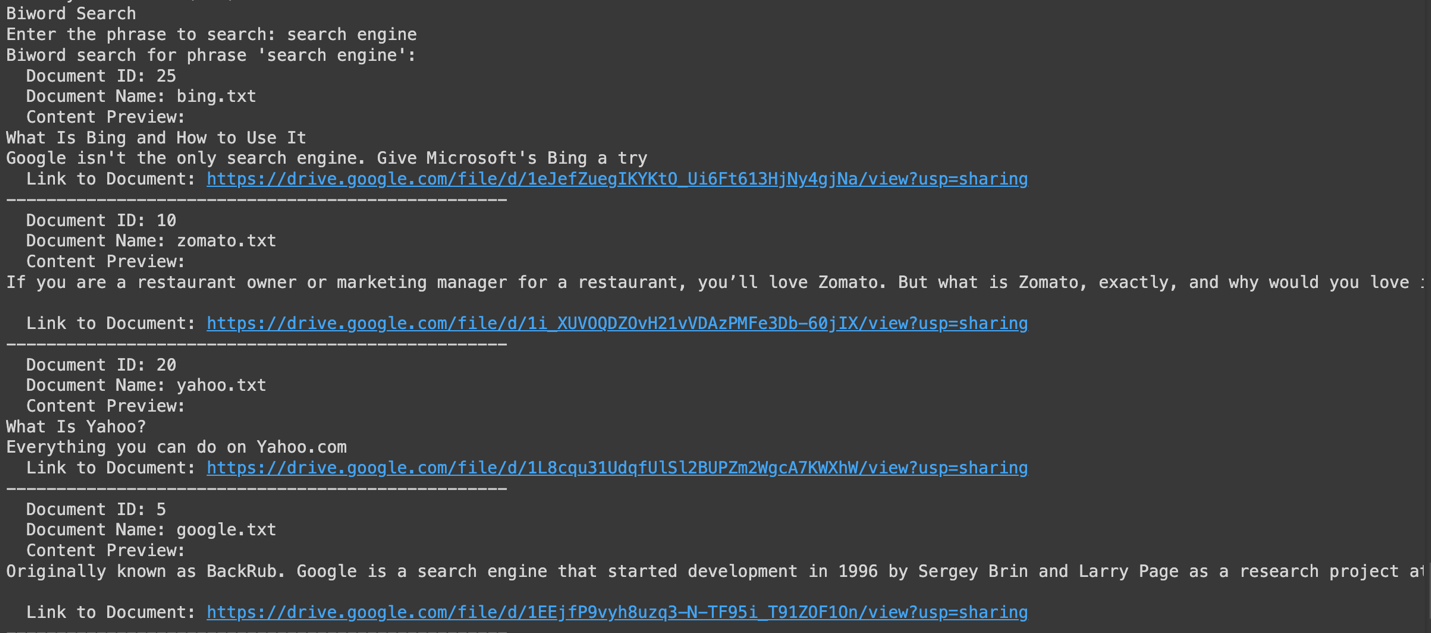
* **Preprocessing**: The text is converted to lowercase, tokenized into words, filtered to remove non-alphabetic tokens and stopwords, and lemmatized to handle variations of words consistently.
* **Biword Construction**: After preprocessing, biwords (pairs of consecutive words) are constructed from the tokenized words.
* **Indexing**: An inverted index is built to map each word to the documents and their positions. A biword index maps each biword to the set of document IDs containing that biword.
* **Search**: Given a search phrase, the biwords are constructed and matched against the biword index. Only documents containing all the specified biwords in sequence are returned.

### **Example**

For a query like "search engine", the system:

* Preprocesses the phrase to obtain biwords.
* Retrieves documents containing these biwords in sequence.
* Displays relevant documents.

**Output**



b) **Proximity queries search by adding positional index**

### **Implementation**

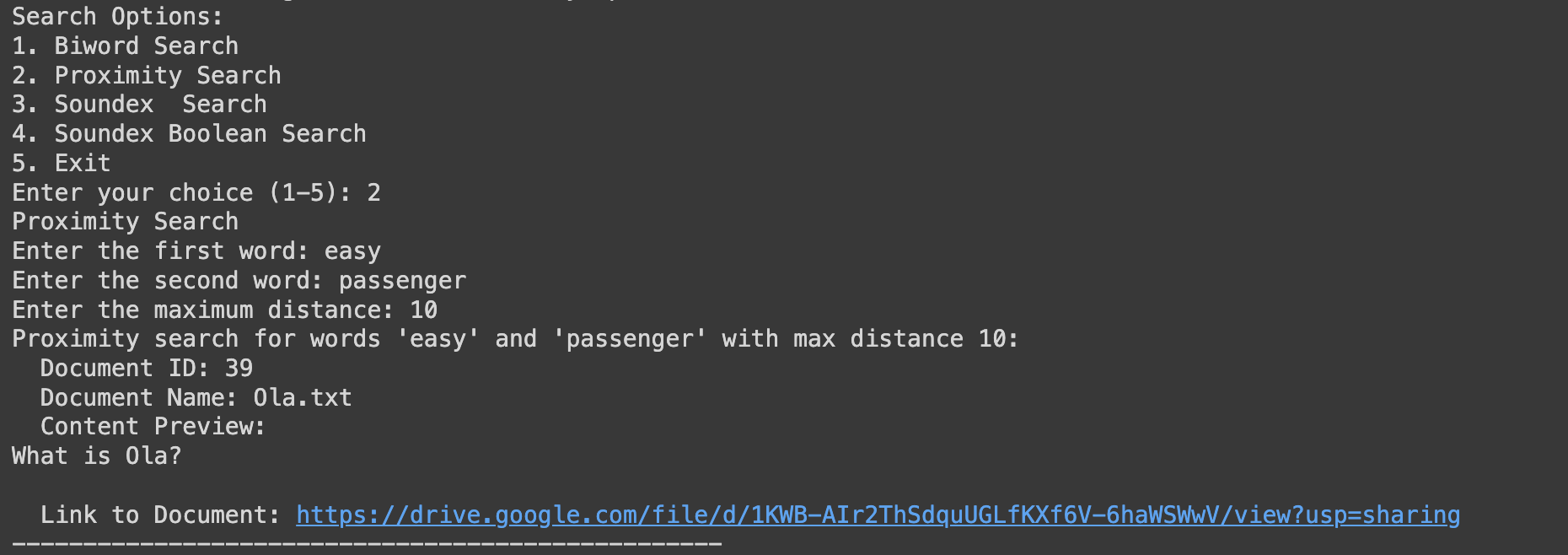
* **Preprocessing**: Similar to the biword search, but the focus is on the positions of words within documents.
* **Indexing**: An inverted index is built where each word maps to its positions in various documents.
* **Search**: For a given pair of words and a maximum distance, the algorithm checks whether the words appear within the specified distance in the same document.

### **Example**

For a query like " easy and passenger " with a maximum distance of 10, the system:

* Retrieves positions of both words in the documents.
* Checks if the words are within the specified distance of each other.
* Returns the documents meeting the proximity criteria.

Output



c) **Use Soundex algorithm for spelling matching of names**

Soundex search is used to find words that sound similar but may be spelled differently. It’s useful for dealing with spelling variations or typos.

### **Implementation**

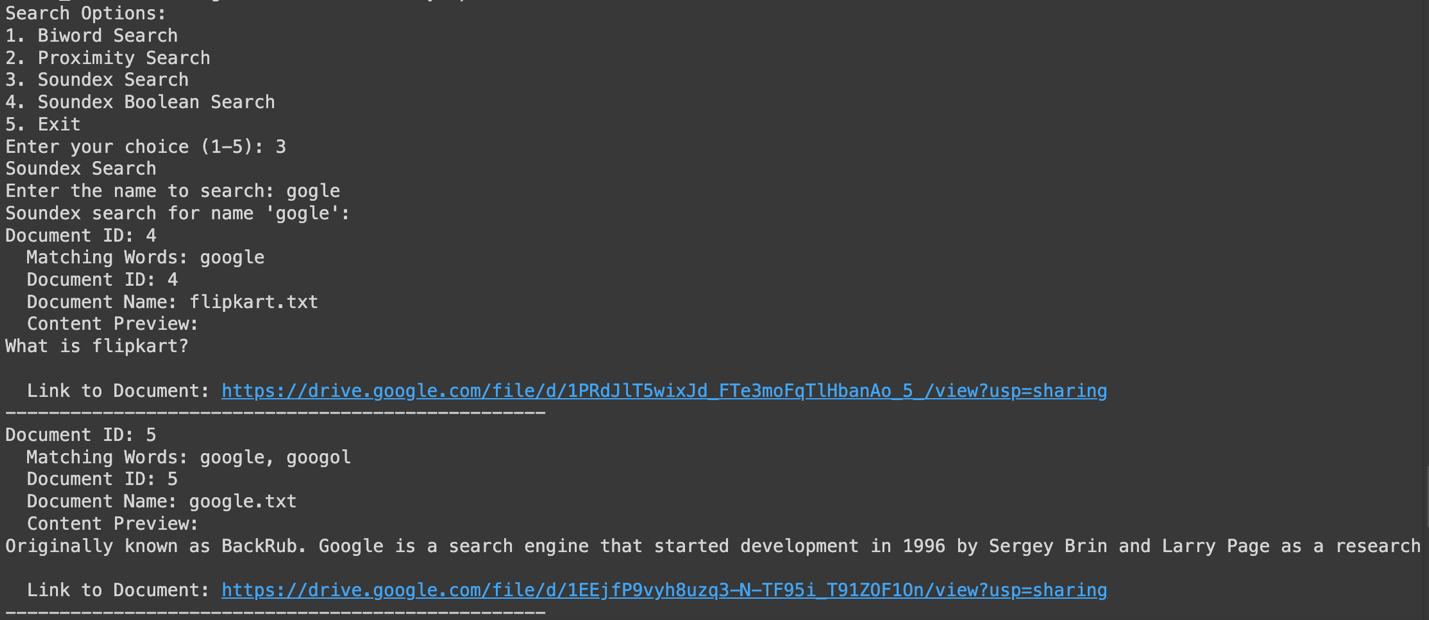
* **Soundex Algorithm**: Converts words into a code based on their pronunciation.
* **Indexing**: Each word is indexed by its Soundex code.
* **Search**: Given a name or term, the system converts it into a Soundex code and retrieves documents containing words with matching Soundex codes.

### **Example**

For a name like "Gogle", the system:

* Converts "Gogle" to its Soundex code.
* Searches for documents containing words with the same Soundex code.
* Displays the matching documents.

Output



d)**Soundex search with Boolean queries**

Soundex Boolean search combines Soundex with Boolean operators (AND, OR, NOT) to refine searches based on phonetic similarity and logical relationships between terms.

### **Implementation**

* **Soundex Conversion**: Converts each term in the query into its Soundex code.
* **Query Parsing**: Parses the Boolean query to handle operators and operands.
* **Search**: Applies Boolean logic to the set of documents identified by Soundex codes.

### **Example**

For a query like "lehri and stainford", the system:

* Converts "lehri" and "stainford" to their respective Soundex codes.
* Finds documents containing words with the corresponding Soundex codes.
* Applies the Boolean operator (AND) to return documents containing both terms.

