ALGORITHMS FOR INFORMATION RETRIEVAL

ASSIGNMENT-1 ANALYSIS REPORT

TEAM NUMBER - 26

NAME	SRN
CHINMAY A	PES2UG20CS902
VIJAY J	PES2UG20CS815
ATHARVA ANIL CHANDA	PES2UG20CS075

CORPUS

Context

IMDb (also known as the Internet Movie Database) is an online database of information related to movies, it combines movie plot description, Metastore ratings, critic and user ratings and reviews, release dates, and many more aspects.

Most data in the database is provided by volunteer contributors.

Content

This dataset includes IMDB top 1,000 movies of all time with attributes such as Title, Certificate, Duration, Genre, etc.

Acknowledgements

The data in this dataset has been scraped using selenium from the publicly available website https://www.imdb.com

Data Structure Used:

List,Dictionary,Array,String

Dataset link:

https://www.kaggle.com/datasets/omarhanyy/imdb-top-1000

Notebook Link:

https://colab.research.google.com/drive/1pCSYNe4u8rghaNb-_NwwewmBc8ibL7cT?usp=sharing

SECTION 2

- PREPROCESSING
- FREE TEXT QUERY USING INVERTED INDEX
- RETRIEVAL BASED ON SIMILARITY INDEX
- BOOLEAN RETRIEVAL
- INVERTED INDEX CREATION
- WILD CARD QUERIES
- PHRASE QUERIES
- ADVANCE QUERY RE-RANK

PREPROCESSING:

- In the first part of the project, the dataset is analyzed. There is a table containing Overview where all the reviews of each movie is present. It is possible to visualize the reviews by a word cloud. After that, the texts of the reviews are cleaned, removing stop words and punctuation and performing stemming. Converting all words to lowercase case (case folding) generating tokens, Removing Stop words, stemming , lemmatization to obtain a filtered sentence

- Preprocessing techniques such as case folding, stop word removal, stemming, and lemmatization can help in information retrieval of this dataset in the following ways:

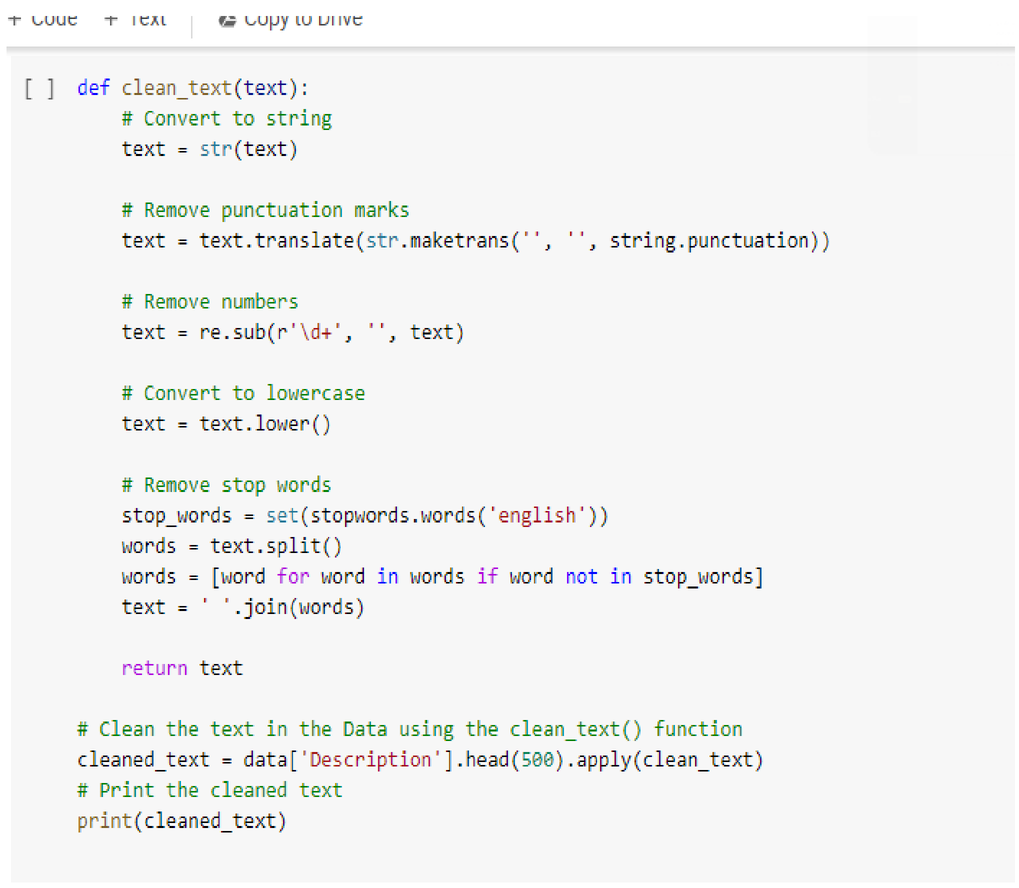
1. **Case Folding**: Converting all words to lowercase can help to normalize the text and reduce the number of variations of the same word. In this dataset, this can be particularly useful in grouping together the reviews that use different capitalizations of the same words, such as "Good" and "good"
2. **Stop word removal:** Stop words such as "the", "a", "an", and "and" can be removed from the reviews to focus on the more meaningful words that carry the sentiment of the review. This can help to improve the accuracy of sentiment analysis models that are built on top of this dataset
3. **Stemming**: Stemming can be applied to reduce words to their base or root form, which can help to group together different variations of the same word. For example,

"watching", "watched", and "watches" could all be stemmed to "watch". This can help to improve the accuracy of topic modeling and keyword extraction models

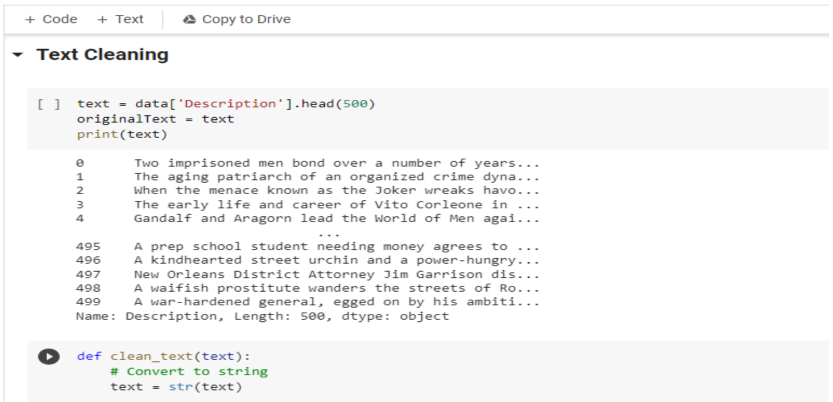
4. **Lemmatization**: Lemmatization can be applied to convert words to their base form using a dictionary, which can result in more accurate results than stemming. This is particularly useful for reviews that use more complex words or have different variations of the same word based on the context. For example, "amazing" and "amaze" could both be lemmatized to "amaze".

5. **Vectorization**: In information retrieval, vectorization is a technique used to transform textual data into numerical vectors that can be used for various tasks such as search, classification, clustering, and recommendation. In the case of the IMDb top 1000 dataset, vectorizing the preprocessed data would involve representing each movie review as a numerical vector

Overall, these preprocessing techniques can help to clean and normalize the text data, which can lead to more accurate and efficient information retrieval and analysis. This can be particularly useful for sentiment analysis, topic modeling, keyword extraction, and other natural language processing tasks.



```
[ ]  def clean_text(text):
         # Convert to string
         text = str(text)

         # Remove punctuation marks
         text = text.translate(str.maketrans('', '', string.punctuation))

         # Remove numbers
         text = re.sub(r'\d+', '', text)

         # Convert to lowercase
         text = text.lower()

         # Remove stop words
         stop_words = set(stopwords.words('english'))
         words = text.split()
         words = [word for word in words if word not in stop_words]
         text = ' '.join(words)

         return text

     # Clean the text in the Data using the clean_text() function
     cleaned_text = data['Description'].head(500).apply(clean_text)
     # Print the cleaned text
     print(cleaned_text)
```

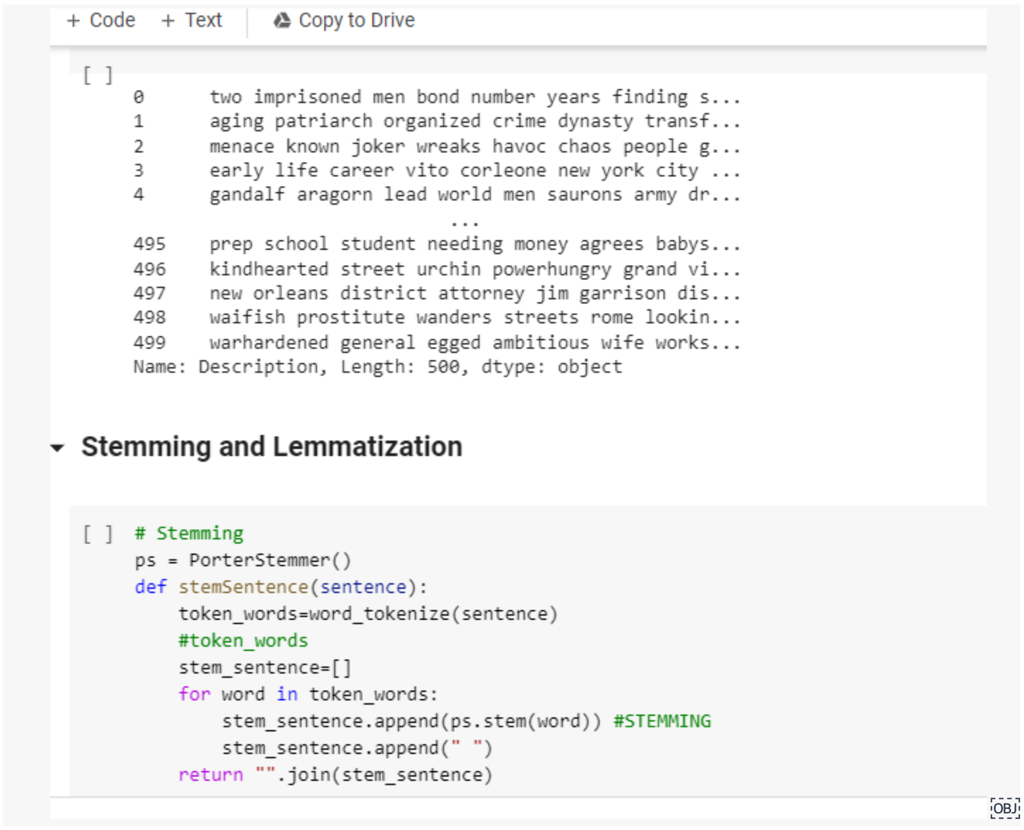


Text Cleaning

```
[ ]  text = data['Description'].head(500)
     originalText = text
     print(text)
```

```
0      Two imprisoned men bond over a number of years...
1      The aging patriarch of an organized crime dyna...
2      When the menace known as the Joker wreaks havo...
3      The early life and career of Vito Corleone in ...
4      Gandalf and Aragorn lead the World of Men agai...
                              ...
495    A prep school student needing money agrees to ...
496    A kindhearted street urchin and a power-hungry...
497    New Orleans District Attorney Jim Garrison dis...
498    A waifish prostitute wanders the streets of Ro...
499    A war-hardened general, egged on by his ambiti...
Name: Description, Length: 500, dtype: object
```

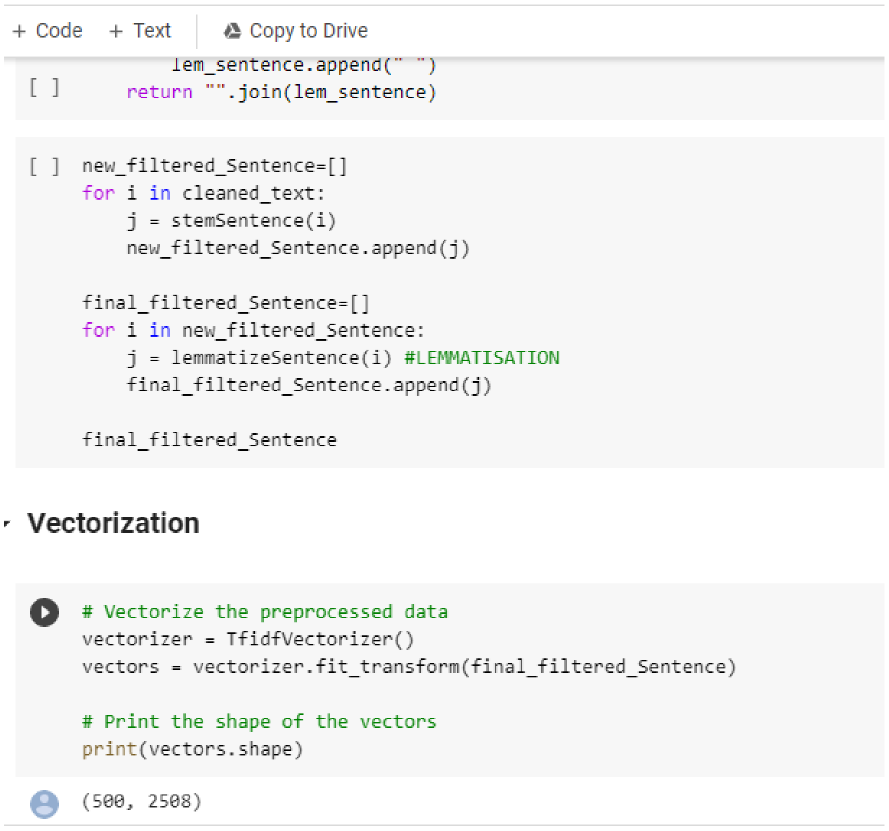
```
def clean_text(text):
    # Convert to string
    text = str(text)
```



```
[ ]
     0      two imprisoned men bond number years finding s...
     1      aging patriarch organized crime dynasty transf...
     2      menace known joker wreaks havoc chaos people g...
     3      early life career vito corleone new york city ...
     4      gandalf aragorn lead world men saurons army dr...
                                  ...
     495    prep school student needing money agrees babys...
     496    kindhearted street urchin powerhungry grand vi...
     497    new orleans district attorney jim garrison dis...
     498    waifish prostitute wanders streets rome lookin...
     499    warhardened general egged ambitious wife works...
     Name: Description, Length: 500, dtype: object
```

Stemming and Lemmatization

```
[ ] # Stemming
    ps = PorterStemmer()
    def stemSentence(sentence):
        token_words=word_tokenize(sentence)
        #token_words
        stem_sentence=[]
        for word in token_words:
            stem_sentence.append(ps.stem(word)) #STEMMING
            stem_sentence.append(" ")
        return "".join(stem_sentence)
```



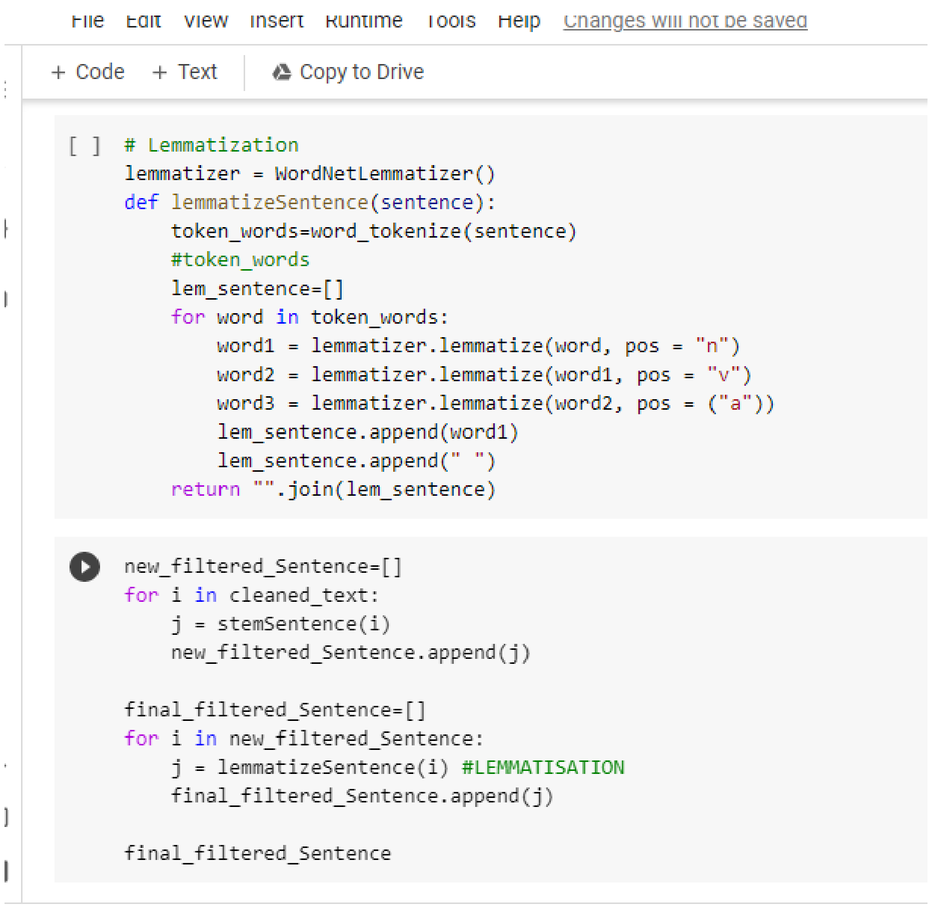
```
            lem_sentence.append(" ")
[ ]     return "".join(lem_sentence)
```

```
[ ] new_filtered_Sentence=[]
    for i in cleaned_text:
        j = stemSentence(i)
        new_filtered_Sentence.append(j)

    final_filtered_Sentence=[]
    for i in new_filtered_Sentence:
        j = lemmatizeSentence(i) #LEMMATISATION
        final_filtered_Sentence.append(j)

    final_filtered_Sentence
```

Vectorization

```
# Vectorize the preprocessed data
vectorizer = TfidfVectorizer()
vectors = vectorizer.fit_transform(final_filtered_Sentence)

# Print the shape of the vectors
print(vectors.shape)
```

```
(500, 2508)
```


File Edit View Insert Runtime Tools Help Changes will not be saved
+ Code + Text
Copy to Drive

```
[ ] # Lemmatization
    lemmatizer = WordNetLemmatizer()
    def lemmatizeSentence(sentence):
        token_words=word_tokenize(sentence)
        #token_words
        lem_sentence=[]
        for word in token_words:
            word1 = lemmatizer.lemmatize(word, pos = "n")
            word2 = lemmatizer.lemmatize(word1, pos = "v")
            word3 = lemmatizer.lemmatize(word2, pos = ("a"))
            lem_sentence.append(word1)
            lem_sentence.append(" ")
        return "".join(lem_sentence)
```

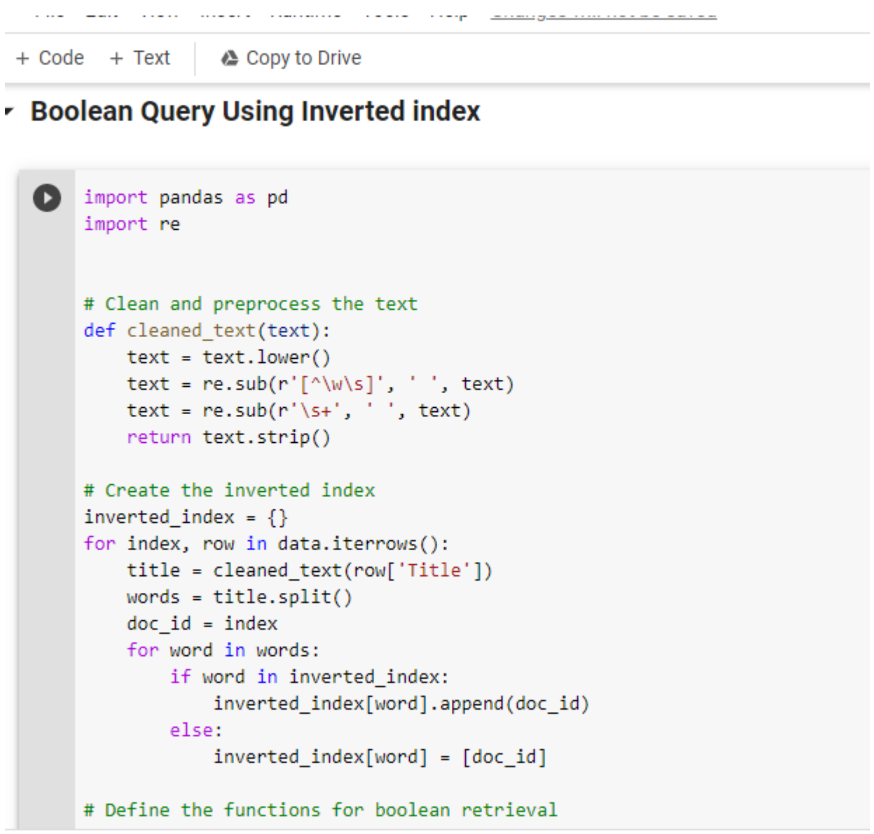
```
new_filtered_Sentence=[]
for i in cleaned_text:
    j = stemSentence(i)
    new_filtered_Sentence.append(j)

final_filtered_Sentence=[]
for i in new_filtered_Sentence:
    j = lemmatizeSentence(i) #LEMMATISATION
    final_filtered_Sentence.append(j)

final_filtered_Sentence
```

BOOLEAN RETRIEVAL USING INVERTED INDEX

- Boolean searches allow you to combine words and phrases using the words AND, OR, NOT (known as Boolean operators) to limit, broaden, or define your search. A good researcher should know how to do a Boolean Search
- An inverted index is a data structure that stores a mapping from terms (in this case, words from the titles of documents) to the documents that contain them. This makes it efficient to retrieve documents that match a given query by looking up the query terms in the index and finding the corresponding documents.
- The code starts by defining a function cleaned_text to preprocess the text by converting it to lowercase, removing non-alphanumeric characters, and removing extra whitespaces. Then, it creates an inverted index by iterating through each document (assuming a variable data containing the document titles) and splitting the title into words, adding the document index to the inverted index for each word.

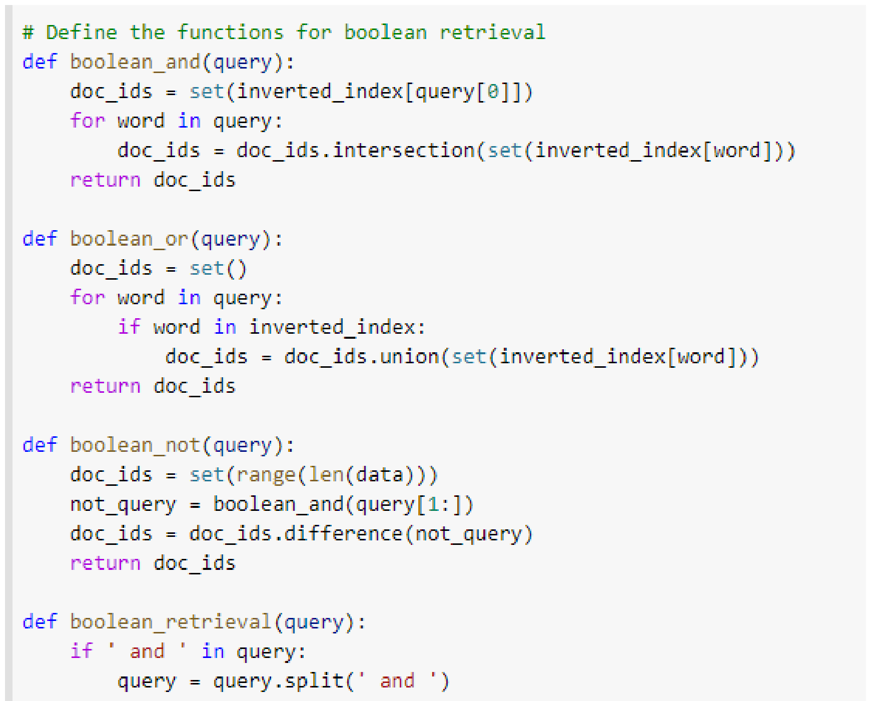
- The code then defines three Boolean retrieval functions, boolean_and, boolean_or, and boolean_not, which implement the Boolean operations AND, OR, and NOT, respectively, using the inverted index to retrieve documents that match the query.

- Finally, the main boolean_retrieval function takes a query as input and uses the appropriate Boolean retrieval function based on the query type (AND, OR, NOT, or a single word) to retrieve a set of matching document IDs. It then calculates the cosine similarity between the query and each of the matching documents (assuming a variable cosine_similarities containing the precomputed cosine similarity scores) and returns a list of document scores, sorted by rank.

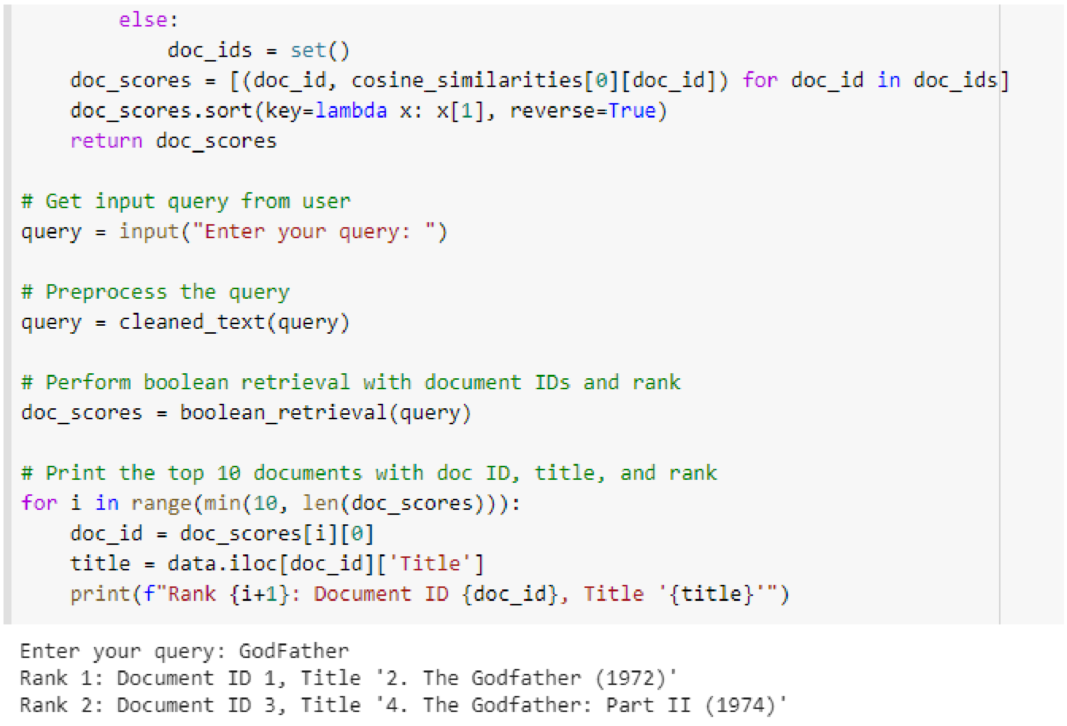


Boolean Query Using Inverted index

```python
import pandas as pd
import re


# Clean and preprocess the text
def cleaned_text(text):
    text = text.lower()
    text = re.sub(r'[^\w\s]', ' ', text)
    text = re.sub(r'\s+', ' ', text)
    return text.strip()

# Create the inverted index
inverted_index = {}
for index, row in data.iterrows():
    title = cleaned_text(row['Title'])
    words = title.split()
    doc_id = index
    for word in words:
        if word in inverted_index:
            inverted_index[word].append(doc_id)
        else:
            inverted_index[word] = [doc_id]

# Define the functions for boolean retrieval
```

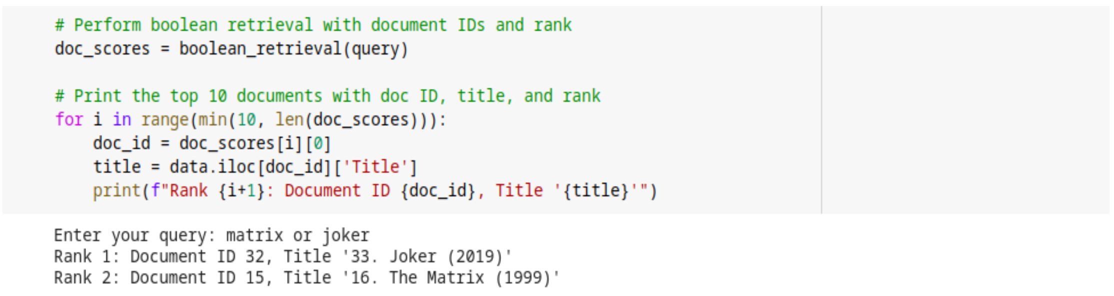


```python
# Define the functions for boolean retrieval
def boolean_and(query):
    doc_ids = set(inverted_index[query[0]])
    for word in query:
        doc_ids = doc_ids.intersection(set(inverted_index[word]))
    return doc_ids

def boolean_or(query):
    doc_ids = set()
    for word in query:
        if word in inverted_index:
            doc_ids = doc_ids.union(set(inverted_index[word]))
    return doc_ids

def boolean_not(query):
    doc_ids = set(range(len(data)))
    not_query = boolean_and(query[1:])
    doc_ids = doc_ids.difference(not_query)
    return doc_ids

def boolean_retrieval(query):
    if ' and ' in query:
        query = query.split(' and ')
```



```
        else:
            doc_ids = set()
    doc_scores = [(doc_id, cosine_similarities[0][doc_id]) for doc_id in doc_ids]
    doc_scores.sort(key=lambda x: x[1], reverse=True)
    return doc_scores

# Get input query from user
query = input("Enter your query: ")

# Preprocess the query
query = cleaned_text(query)

# Perform boolean retrieval with document IDs and rank
doc_scores = boolean_retrieval(query)

# Print the top 10 documents with doc ID, title, and rank
for i in range(min(10, len(doc_scores))):
    doc_id = doc_scores[i][0]
    title = data.iloc[doc_id]['Title']
    print(f"Rank {i+1}: Document ID {doc_id}, Title '{title}'")
```

```
Enter your query: GodFather
Rank 1: Document ID 1, Title '2. The Godfather (1972)'
Rank 2: Document ID 3, Title '4. The Godfather: Part II (1974)'
```



```
# Perform boolean retrieval with document IDs and rank
doc_scores = boolean_retrieval(query)

# Print the top 10 documents with doc ID, title, and rank
for i in range(min(10, len(doc_scores))):
    doc_id = doc_scores[i][0]
    title = data.iloc[doc_id]['Title']
    print(f"Rank {i+1}: Document ID {doc_id}, Title '{title}'")
```

```
Enter your query: matrix or joker
Rank 1: Document ID 32, Title '33. Joker (2019)'
Rank 2: Document ID 15, Title '16. The Matrix (1999)'
```

<u>**INVERTED INDEX CREATION AND EXECUTION ON FREE TEXT QUERIES**</u>

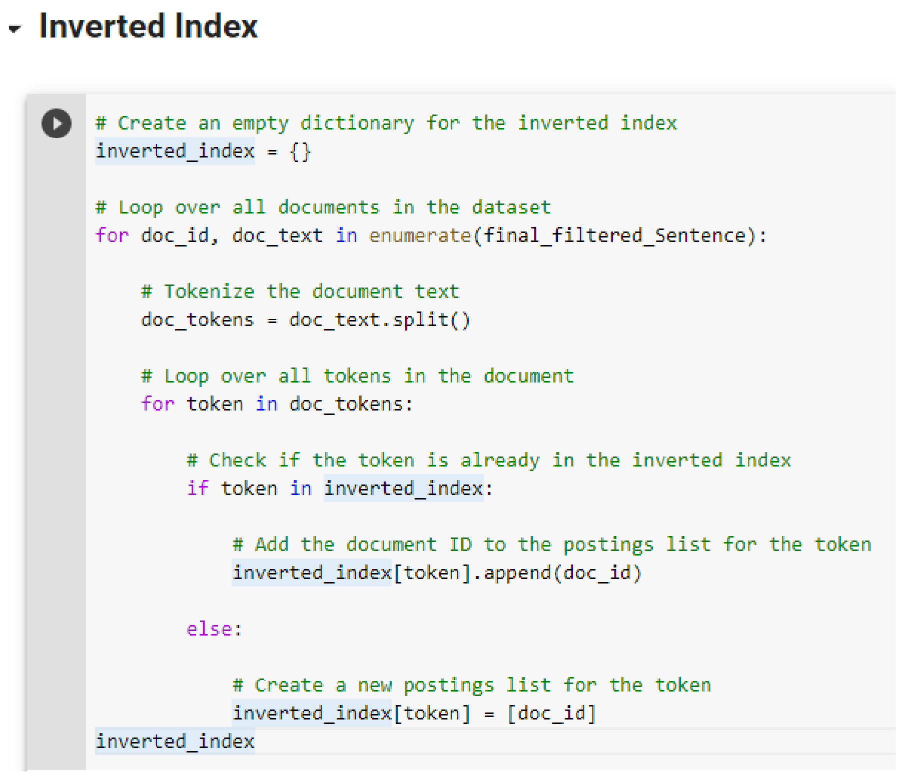
- Inverted index is a data structure used in information retrieval systems to efficiently retrieve documents or web pages containing a specific term or set of terms. In an inverted index, the index is organized by terms (words), and each term points to a list of documents or web pages that contain that term.

- The code starts by creating an empty dictionary called inverted_index, which will be used to store the inverted index.

- Then, the code loops over each document in the dataset using a for loop with the enumerate function. For each document, the text is tokenized into individual words using the split() method.

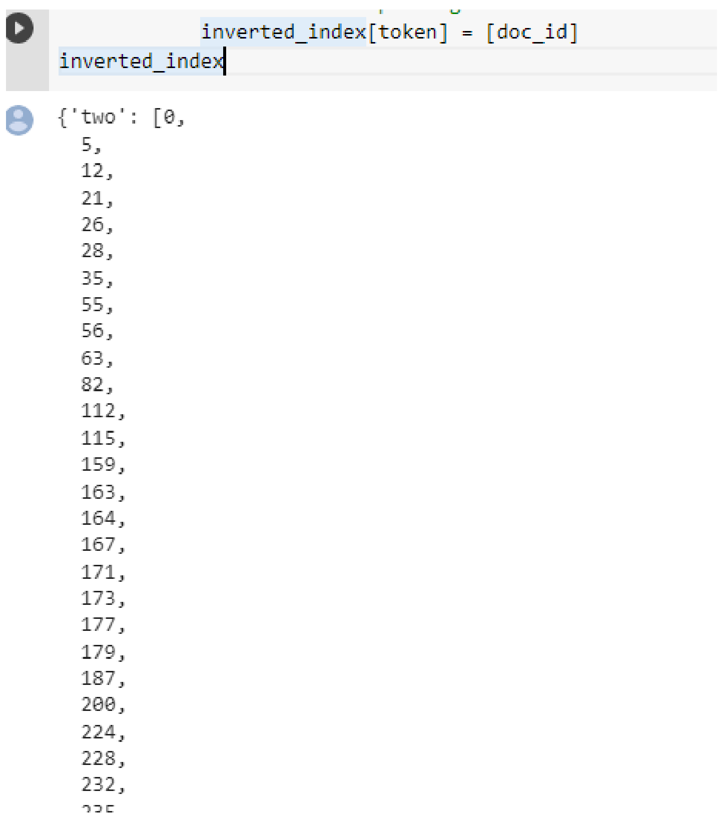
- Next, the code loops over each token in the document, and checks if the token is already in the inverted index. If it is, the document ID (represented by the variable doc_id) is added to the postings list for the token (which is a list of document IDs that contain the token). If the token is not yet in the inverted index, a new postings list is created for the token, and the current document ID is added to it.

- Finally, after processing all of the documents, the resulting inverted_index dictionary contains a set of keys that are the unique tokens in the dataset, and the corresponding values are lists of document IDs that contain each token. This data structure can be used for efficient text search, for example by finding all documents that contain a particular query term.



Inverted Index

```
# Create an empty dictionary for the inverted index
inverted_index = {}

# Loop over all documents in the dataset
for doc_id, doc_text in enumerate(final_filtered_Sentence):

    # Tokenize the document text
    doc_tokens = doc_text.split()

    # Loop over all tokens in the document
    for token in doc_tokens:

        # Check if the token is already in the inverted index
        if token in inverted_index:

            # Add the document ID to the postings list for the token
            inverted_index[token].append(doc_id)

        else:

            # Create a new postings list for the token
            inverted_index[token] = [doc_id]
inverted_index
```



```
            inverted_index[token] = [doc_id]
inverted_index
```

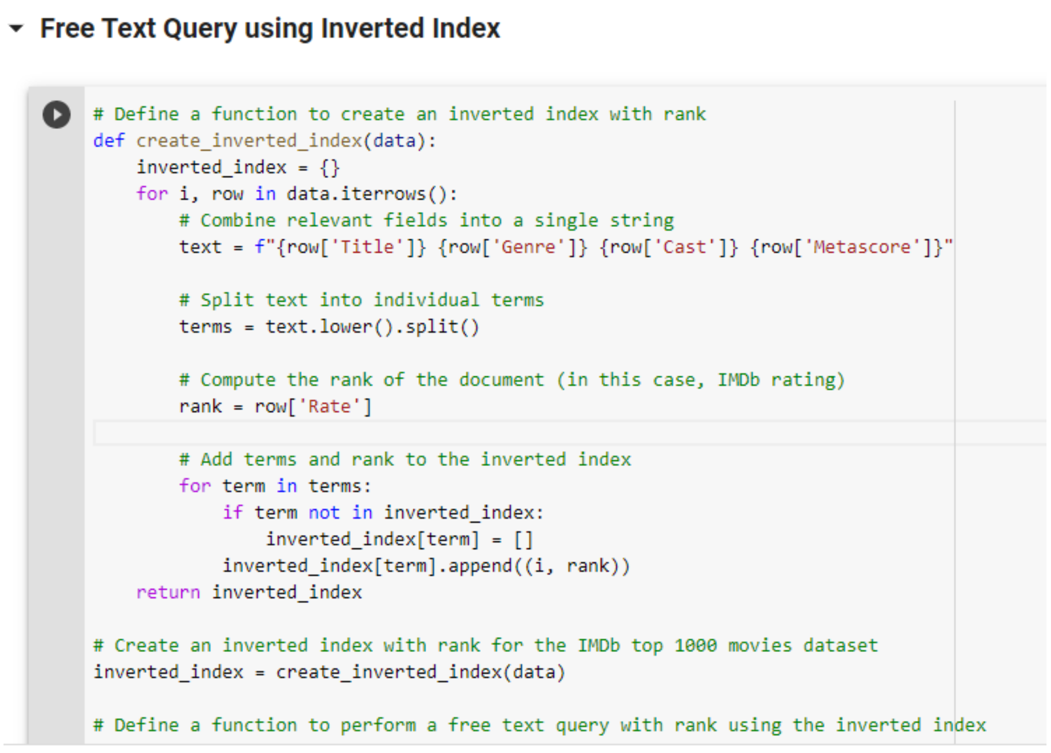
```
{'two': [0,
  5,
  12,
  21,
  26,
  28,
  35,
  55,
  56,
  63,
  82,
  112,
  115,
  159,
  163,
  164,
  167,
  171,
  173,
  177,
  179,
  187,
  200,
  224,
  228,
  232,
```

FREE TEXT QUERIES EXECUTION WITH RANK

- An inverted index is a data structure used in information retrieval to efficiently store and retrieve information about the terms in a corpus of documents. In this case, the IMDb top 1000 movies dataset is being used as the corpus. The inverted index is created by parsing through the dataset and creating a mapping between each term and the documents (movies) that contain that term, along with a rank or weight assigned to that term.

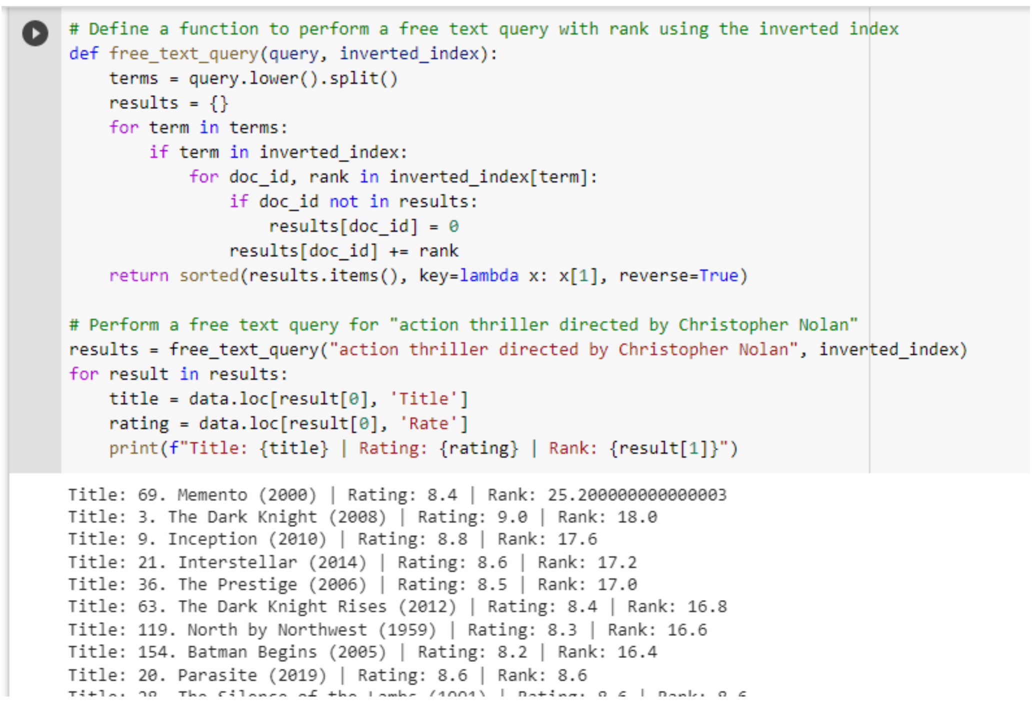
- The function free_text_query takes a query string as input and performs a search on the inverted index. The query string is first converted to lowercase and split into individual terms. For each term in the query, the function looks up the corresponding document IDs and ranks in the inverted index. If a document ID is already in the results dictionary, the rank for that term is added to the existing value. If not, a new entry is created in the results dictionary with the rank for that term.

- Finally, the results are sorted based on the total rank for each document and printed in descending order, along with the corresponding movie title and rating. The rank is a measure of the relevance of the movie to the query, calculated by summing up the ranks of all the terms in the query that appear in the movie's description.



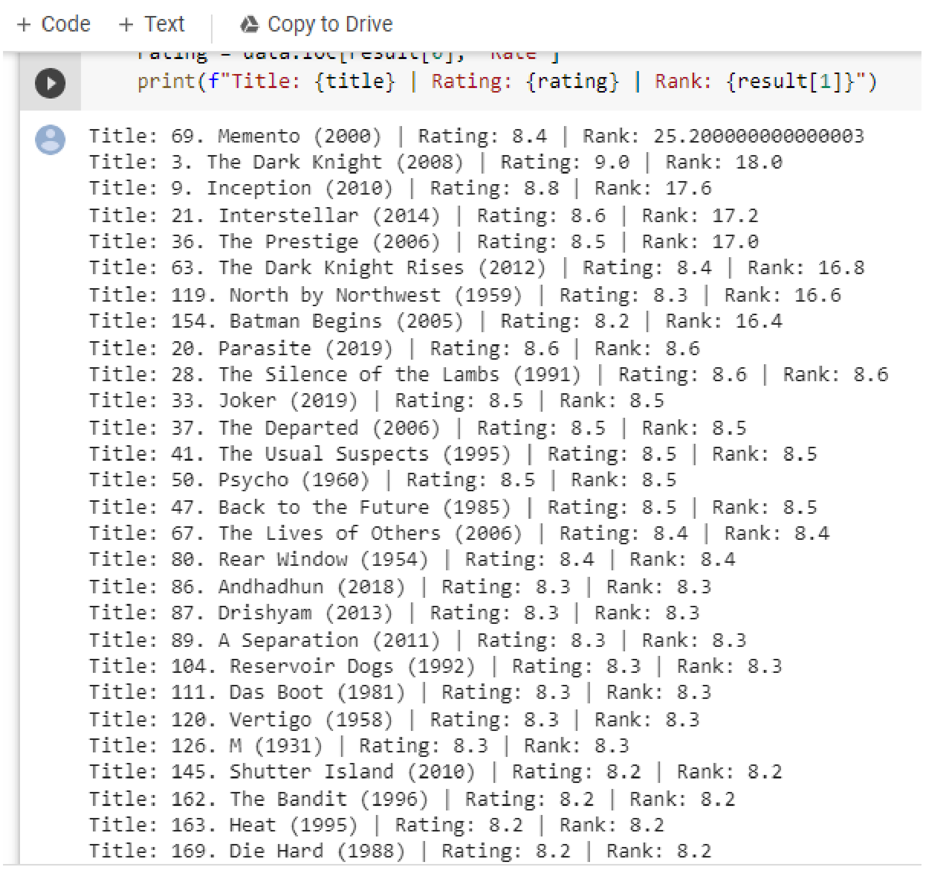
Free Text Query using Inverted Index

```python
# Define a function to create an inverted index with rank
def create_inverted_index(data):
    inverted_index = {}
    for i, row in data.iterrows():
        # Combine relevant fields into a single string
        text = f"{row['Title']} {row['Genre']} {row['Cast']} {row['Metascore']}"

        # Split text into individual terms
        terms = text.lower().split()

        # Compute the rank of the document (in this case, IMDb rating)
        rank = row['Rate']

        # Add terms and rank to the inverted index
        for term in terms:
            if term not in inverted_index:
                inverted_index[term] = []
            inverted_index[term].append((i, rank))
    return inverted_index

# Create an inverted index with rank for the IMDb top 1000 movies dataset
inverted_index = create_inverted_index(data)

# Define a function to perform a free text query with rank using the inverted index
```



```python
# Define a function to perform a free text query with rank using the inverted index
def free_text_query(query, inverted_index):
    terms = query.lower().split()
    results = {}
    for term in terms:
        if term in inverted_index:
            for doc_id, rank in inverted_index[term]:
                if doc_id not in results:
                    results[doc_id] = 0
                results[doc_id] += rank
    return sorted(results.items(), key=lambda x: x[1], reverse=True)

# Perform a free text query for "action thriller directed by Christopher Nolan"
results = free_text_query("action thriller directed by Christopher Nolan", inverted_index)
for result in results:
    title = data.loc[result[0], 'Title']
    rating = data.loc[result[0], 'Rate']
    print(f"Title: {title} | Rating: {rating} | Rank: {result[1]}")
```

```
Title: 69. Memento (2000) | Rating: 8.4 | Rank: 25.200000000000003
Title: 3. The Dark Knight (2008) | Rating: 9.0 | Rank: 18.0
Title: 9. Inception (2010) | Rating: 8.8 | Rank: 17.6
Title: 21. Interstellar (2014) | Rating: 8.6 | Rank: 17.2
Title: 36. The Prestige (2006) | Rating: 8.5 | Rank: 17.0
Title: 63. The Dark Knight Rises (2012) | Rating: 8.4 | Rank: 16.8
Title: 119. North by Northwest (1959) | Rating: 8.3 | Rank: 16.6
Title: 154. Batman Begins (2005) | Rating: 8.2 | Rank: 16.4
Title: 20. Parasite (2019) | Rating: 8.6 | Rank: 8.6
```



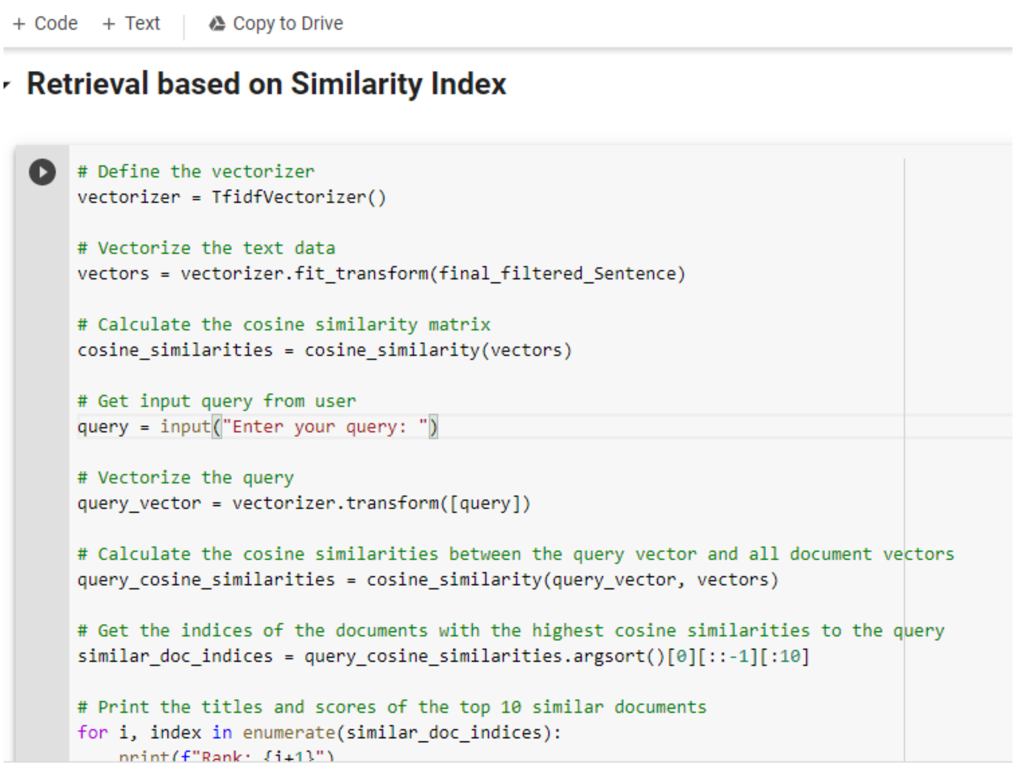
```
    print(f"Title: {title} | Rating: {rating} | Rank: {result[1]}")
```

```
Title: 69. Memento (2000) | Rating: 8.4 | Rank: 25.200000000000003
Title: 3. The Dark Knight (2008) | Rating: 9.0 | Rank: 18.0
Title: 9. Inception (2010) | Rating: 8.8 | Rank: 17.6
Title: 21. Interstellar (2014) | Rating: 8.6 | Rank: 17.2
Title: 36. The Prestige (2006) | Rating: 8.5 | Rank: 17.0
Title: 63. The Dark Knight Rises (2012) | Rating: 8.4 | Rank: 16.8
Title: 119. North by Northwest (1959) | Rating: 8.3 | Rank: 16.6
Title: 154. Batman Begins (2005) | Rating: 8.2 | Rank: 16.4
Title: 20. Parasite (2019) | Rating: 8.6 | Rank: 8.6
Title: 28. The Silence of the Lambs (1991) | Rating: 8.6 | Rank: 8.6
Title: 33. Joker (2019) | Rating: 8.5 | Rank: 8.5
Title: 37. The Departed (2006) | Rating: 8.5 | Rank: 8.5
Title: 41. The Usual Suspects (1995) | Rating: 8.5 | Rank: 8.5
Title: 50. Psycho (1960) | Rating: 8.5 | Rank: 8.5
Title: 47. Back to the Future (1985) | Rating: 8.5 | Rank: 8.5
Title: 67. The Lives of Others (2006) | Rating: 8.4 | Rank: 8.4
Title: 80. Rear Window (1954) | Rating: 8.4 | Rank: 8.4
Title: 86. Andhadhun (2018) | Rating: 8.3 | Rank: 8.3
Title: 87. Drishyam (2013) | Rating: 8.3 | Rank: 8.3
Title: 89. A Separation (2011) | Rating: 8.3 | Rank: 8.3
Title: 104. Reservoir Dogs (1992) | Rating: 8.3 | Rank: 8.3
Title: 111. Das Boot (1981) | Rating: 8.3 | Rank: 8.3
Title: 120. Vertigo (1958) | Rating: 8.3 | Rank: 8.3
Title: 126. M (1931) | Rating: 8.3 | Rank: 8.3
Title: 145. Shutter Island (2010) | Rating: 8.2 | Rank: 8.2
Title: 162. The Bandit (1996) | Rating: 8.2 | Rank: 8.2
Title: 163. Heat (1995) | Rating: 8.2 | Rank: 8.2
Title: 169. Die Hard (1988) | Rating: 8.2 | Rank: 8.2
```

RETRIEVAL BASED ON SIMILARITY INDEX

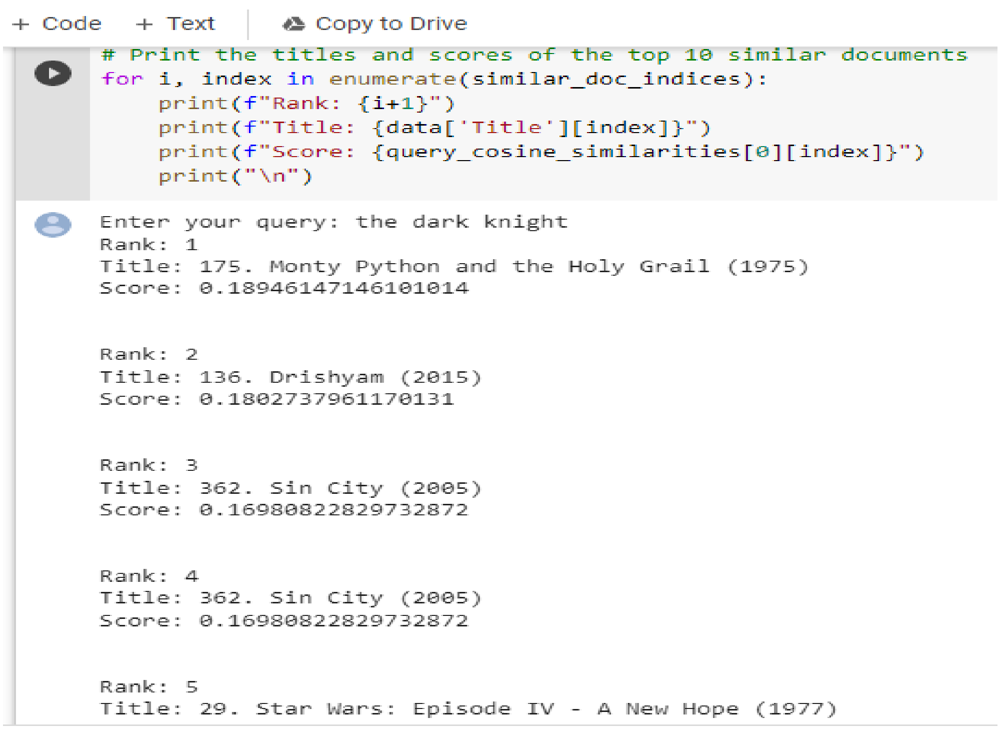
- We are implementing a simple information retrieval system using the Vector Space Model (VSM) and cosine similarity to rank documents based on their relevance to a user's input query.
- Here is a step-by-step breakdown of the code:
- The TfidfVectorizer class from scikit-learn is used to transform the raw text data (stored in final_filtered_Sentence variable) into a matrix of TF-IDF (Term Frequency-Inverse Document Frequency) features. This converts the text data into a numerical representation that can be used for further processing.
- The cosine_similarity function from scikit-learn is used to calculate the cosine similarity matrix between all the document vectors in the transformed dataset.
- The user is prompted to enter a query, which is stored in the query variable.

- The query is vectorized using the same TfidfVectorizer object created earlier, and the resulting query vector is stored in the query_vector variable.
- The cosine_similarity function is used again to calculate the cosine similarities between the query vector and all document vectors in the dataset.
- The argsort() function is used to sort the cosine similarities in descending order, and the first 10 indices of the sorted list are stored in the similar_doc_indices variable. These indices correspond to the 10 most similar documents to the user's input query.
- A for loop is used to iterate through the top 10 similar documents and print their rank, title, and score (cosine similarity value) to the user.
- Overall, this code demonstrates a simple implementation of a text-based search engine using the Vector Space Model and cosine similarity to retrieve documents based on their relevance to a user's input query.



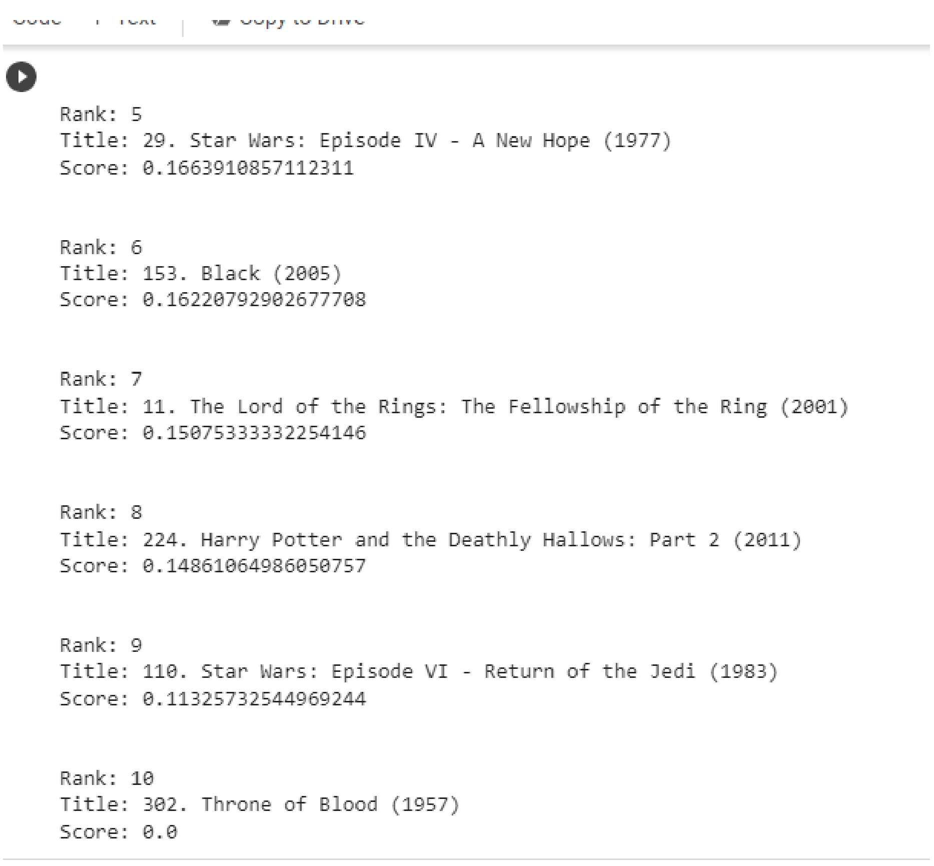
Retrieval based on Similarity Index

```
# Define the vectorizer
vectorizer = TfidfVectorizer()

# Vectorize the text data
vectors = vectorizer.fit_transform(final_filtered_Sentence)

# Calculate the cosine similarity matrix
cosine_similarities = cosine_similarity(vectors)

# Get input query from user
query = input("Enter your query: ")

# Vectorize the query
query_vector = vectorizer.transform([query])

# Calculate the cosine similarities between the query vector and all document vectors
query_cosine_similarities = cosine_similarity(query_vector, vectors)

# Get the indices of the documents with the highest cosine similarities to the query
similar_doc_indices = query_cosine_similarities.argsort()[0][::-1][:10]

# Print the titles and scores of the top 10 similar documents
for i, index in enumerate(similar_doc_indices):
    print(f"Rank: {i+1}")
```



```
# Print the titles and scores of the top 10 similar documents
for i, index in enumerate(similar_doc_indices):
    print(f"Rank: {i+1}")
    print(f"Title: {data['Title'][index]}")
    print(f"Score: {query_cosine_similarities[0][index]}")
    print("\n")
```

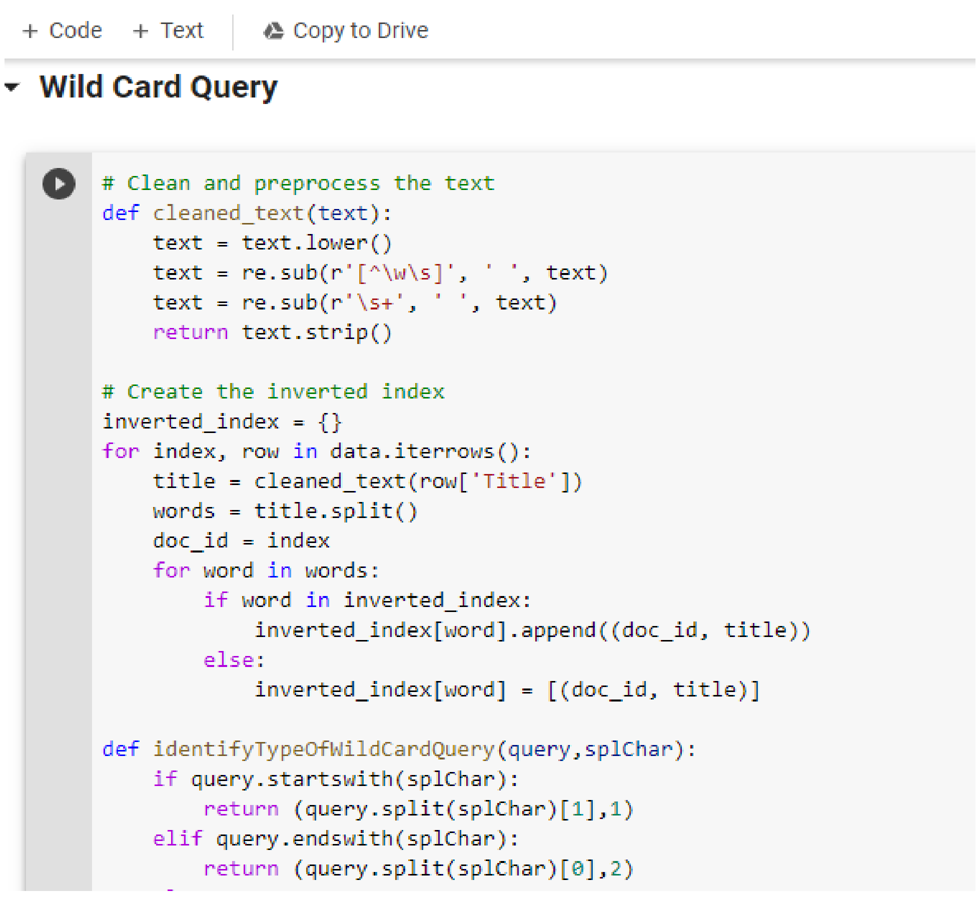
```
Enter your query: the dark knight
Rank: 1
Title: 175. Monty Python and the Holy Grail (1975)
Score: 0.18946147146101014


Rank: 2
Title: 136. Drishyam (2015)
Score: 0.1802737961170131


Rank: 3
Title: 362. Sin City (2005)
Score: 0.16980822829732872


Rank: 4
Title: 362. Sin City (2005)
Score: 0.16980822829732872


Rank: 5
Title: 29. Star Wars: Episode IV - A New Hope (1977)
```



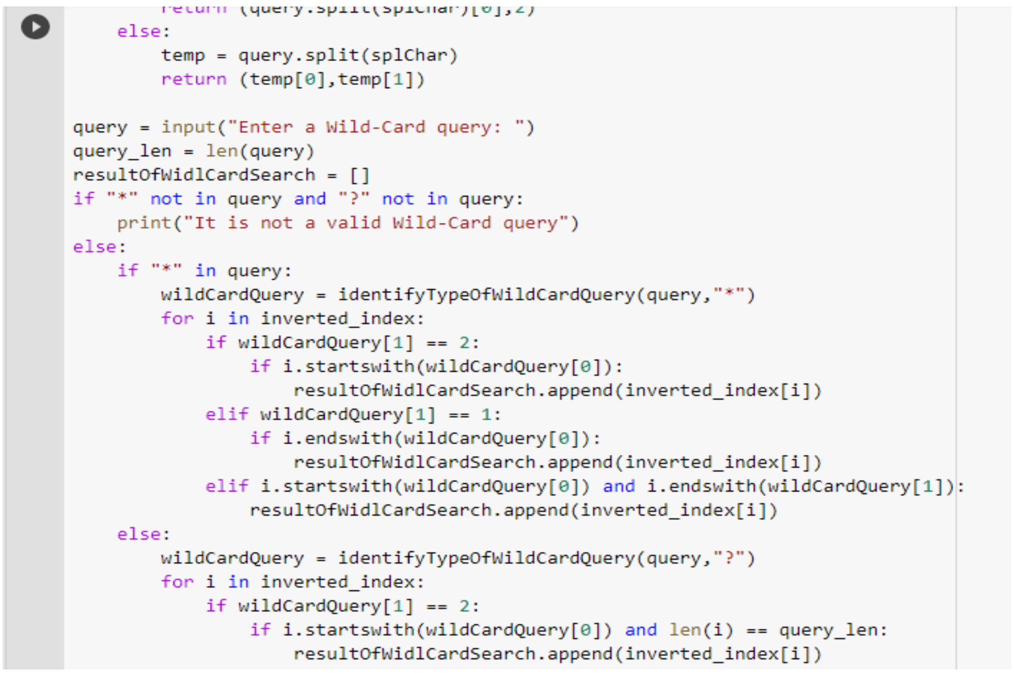
```
Rank: 5
Title: 29. Star Wars: Episode IV - A New Hope (1977)
Score: 0.1663910857112311


Rank: 6
Title: 153. Black (2005)
Score: 0.16220792902677708


Rank: 7
Title: 11. The Lord of the Rings: The Fellowship of the Ring (2001)
Score: 0.15075333332254146


Rank: 8
Title: 224. Harry Potter and the Deathly Hallows: Part 2 (2011)
Score: 0.14861064986050757


Rank: 9
Title: 110. Star Wars: Episode VI - Return of the Jedi (1983)
Score: 0.11325732544969244


Rank: 10
Title: 302. Throne of Blood (1957)
Score: 0.0
```

WILDCARD QUERIES

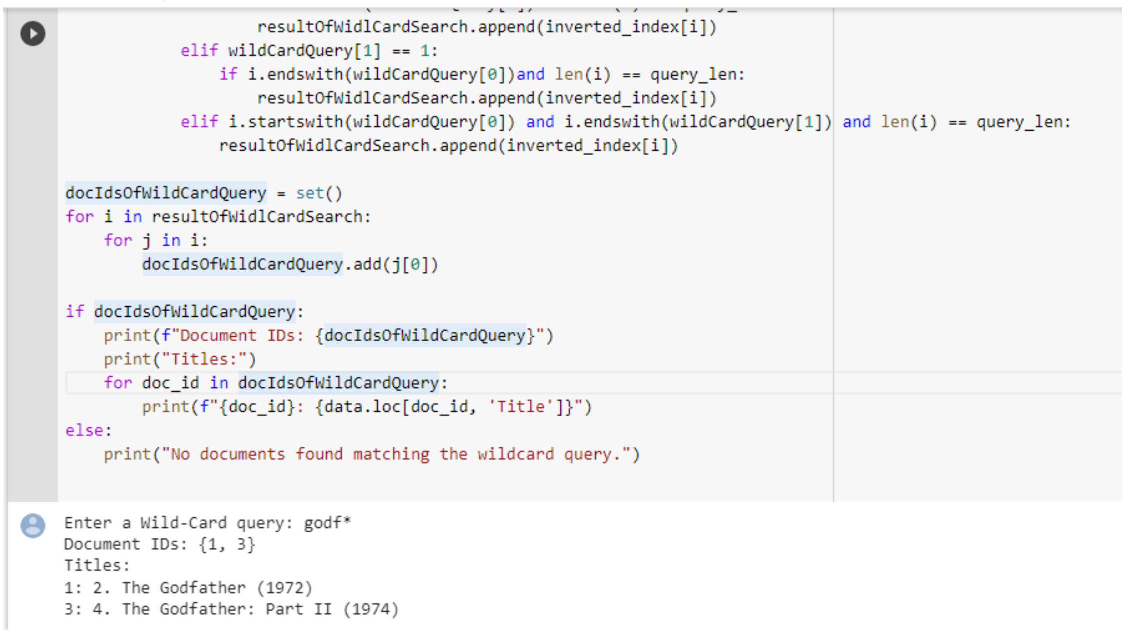
- The code is an implementation of a wildcard search using an inverted index. The code first defines a function called cleaned_text() which converts all characters to lowercase and removes all punctuation from the text.
- Next, the code creates an inverted index from a Pandas dataframe called data. The inverted index is a dictionary where each key is a word from the text, and the value is a list of tuples. Each tuple contains the document ID and the title of the document where the word appears.
- The code then prompts the user to enter a wildcard query. If the query contains "*" or "?", the program identifies the type of the wildcard query using a function called identifyTypeOfWildCardQuery().
- The code then searches the inverted index for words that match the wildcard query. If the wildcard query ends with "", the program searches for words that start with the characters before the "". If the wildcard query starts with "", the program searches for words that end with the characters after the "". If the wildcard query contains "" in the middle, the program searches for words that start with the characters before the "" and end with the characters after the "*". Similarly, if the wildcard query contains "?", the program searches for words that match the characters before and after the "?".
- The code then creates a set of document IDs that match the wildcard query and prints the titles of the documents that match the query. If no documents are found, the program prints a message indicating that no documents match the query.



```
# Clean and preprocess the text
def cleaned_text(text):
    text = text.lower()
    text = re.sub(r'[^\w\s]', ' ', text)
    text = re.sub(r'\s+', ' ', text)
    return text.strip()

# Create the inverted index
inverted_index = {}
for index, row in data.iterrows():
    title = cleaned_text(row['Title'])
    words = title.split()
    doc_id = index
    for word in words:
        if word in inverted_index:
            inverted_index[word].append((doc_id, title))
        else:
            inverted_index[word] = [(doc_id, title)]

def identifyTypeOfWildCardQuery(query,splChar):
    if query.startswith(splChar):
        return (query.split(splChar)[1],1)
    elif query.endswith(splChar):
        return (query.split(splChar)[0],2)
```



```
        return (query.split(splChar)[0],2)
    else:
        temp = query.split(splChar)
        return (temp[0],temp[1])

query = input("Enter a Wild-Card query: ")
query_len = len(query)
resultOfWidlCardSearch = []
if "*" not in query and "?" not in query:
    print("It is not a valid Wild-Card query")
else:
    if "*" in query:
        wildCardQuery = identifyTypeOfWildCardQuery(query,"*")
        for i in inverted_index:
            if wildCardQuery[1] == 2:
                if i.startswith(wildCardQuery[0]):
                    resultOfWidlCardSearch.append(inverted_index[i])
            elif wildCardQuery[1] == 1:
                if i.endswith(wildCardQuery[0]):
                    resultOfWidlCardSearch.append(inverted_index[i])
            elif i.startswith(wildCardQuery[0]) and i.endswith(wildCardQuery[1]):
                resultOfWidlCardSearch.append(inverted_index[i])
    else:
        wildCardQuery = identifyTypeOfWildCardQuery(query,"?")
        for i in inverted_index:
            if wildCardQuery[1] == 2:
                if i.startswith(wildCardQuery[0]) and len(i) == query_len:
                    resultOfWidlCardSearch.append(inverted_index[i])
```



```
                    resultOfWidlCardSearch.append(inverted_index[i])
            elif wildCardQuery[1] == 1:
                if i.endswith(wildCardQuery[0])and len(i) == query_len:
                    resultOfWidlCardSearch.append(inverted_index[i])
            elif i.startswith(wildCardQuery[0]) and i.endswith(wildCardQuery[1]) and len(i) == query_len:
                resultOfWidlCardSearch.append(inverted_index[i])

docIdsOfWildCardQuery = set()
for i in resultOfWidlCardSearch:
    for j in i:
        docIdsOfWildCardQuery.add(j[0])

if docIdsOfWildCardQuery:
    print(f"Document IDs: {docIdsOfWildCardQuery}")
    print("Titles:")
    for doc_id in docIdsOfWildCardQuery:
        print(f"{doc_id}: {data.loc[doc_id, 'Title']}")
else:
    print("No documents found matching the wildcard query.")
```

```
Enter a Wild-Card query: godf*
Document IDs: {1, 3}
Titles:
1: 2. The Godfather (1972)
3: 4. The Godfather: Part II (1974)
```

PHRASE QUERY USING BI-WORD INDEX AND POSITIONAL INDEXING

- One approach to handling phrases is to consider every pair of consecutive terms in a document as a phrase. For example, the text Friends, Romans, Countrymen would generate the bi-words :
- friends romans
  romans countrymen
- The code first defines a function cleaned_text that takes a text and applies some preprocessing steps to it, including converting it to lowercase, removing non-alphanumeric characters, and removing extra whitespace.

- Then, it creates two indexes: biword_index and positional_index. biword_index is a dictionary that maps bi-words (pairs of consecutive words) to the documents that contain them and their positions. positional_index is a dictionary that maps individual words to the documents that contain them and their positions.

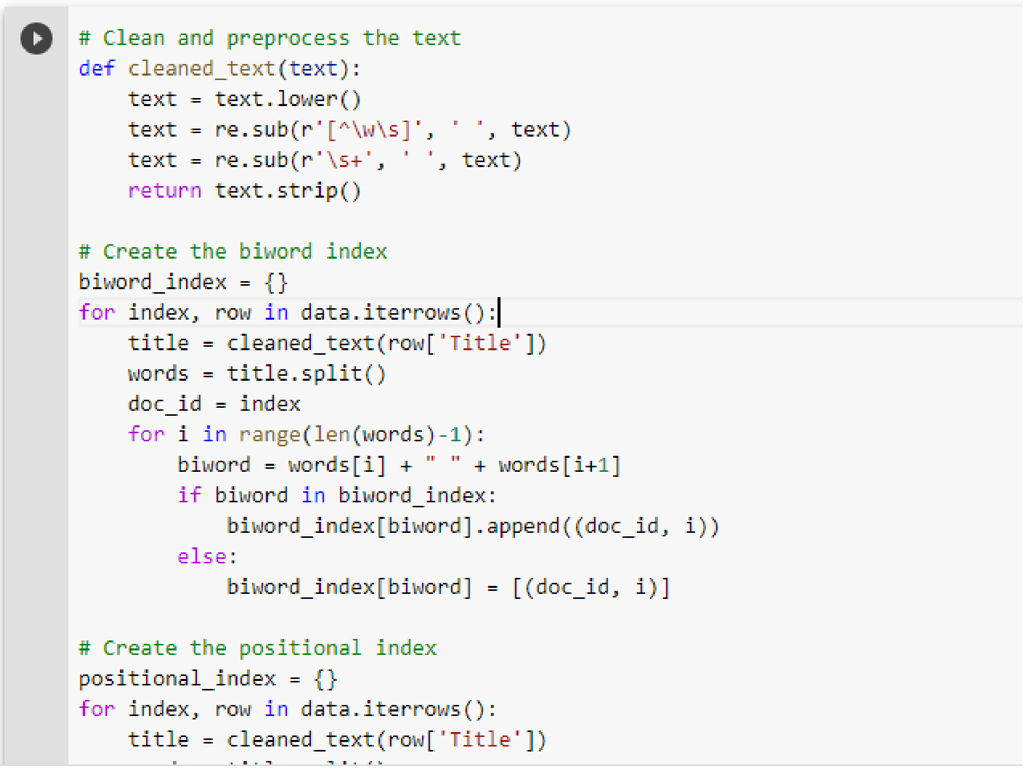
- The code then prompts the user to input a phrase query and whether they want the results to be ranked by position or not.

- Using the bi-word index, the code finds the candidate documents that contain any of the bi-words in the query. Then, using the positional index, the code checks for each candidate document whether it contains all the query words in the correct order and calculates the positions where they appear. The matched documents and their matching positions are stored in the matched_docs list.

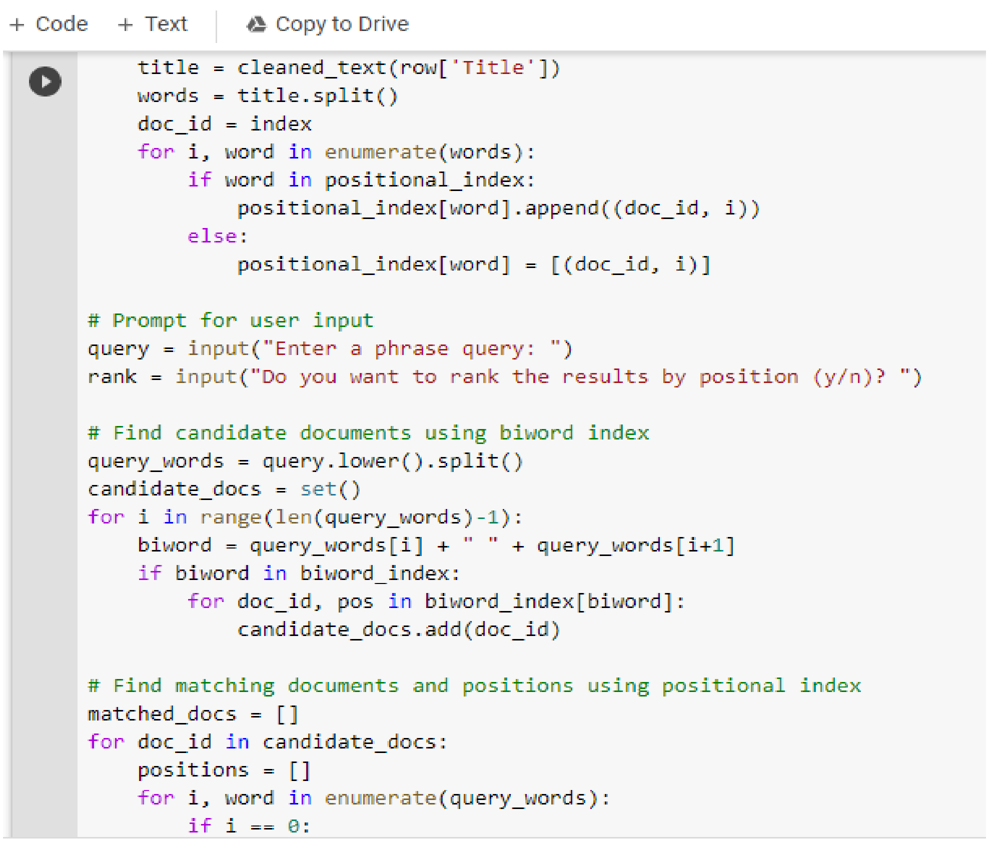
- If the user chooses to rank the results by position, the matched_docs list is sorted by the number of matching positions in each document.

- Finally, the code displays the matching documents and their matching positions, or a message if no documents were found. The output includes the document ID, title, and the positions where the query appears in the title.

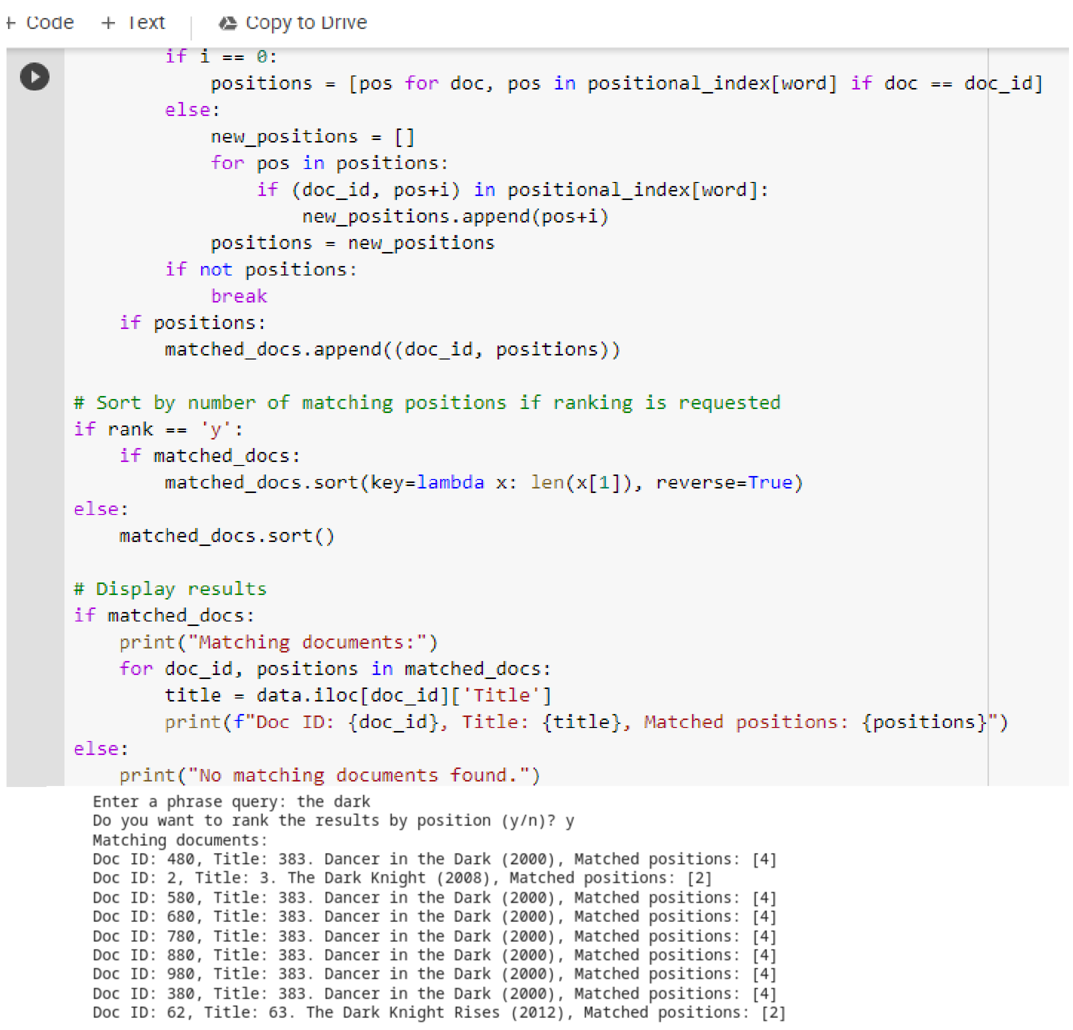
Pharse Query using Bi-word Index and Positional Indexing



```python
# Clean and preprocess the text
def cleaned_text(text):
    text = text.lower()
    text = re.sub(r'[^\w\s]', ' ', text)
    text = re.sub(r'\s+', ' ', text)
    return text.strip()

# Create the biword index
biword_index = {}
for index, row in data.iterrows():
    title = cleaned_text(row['Title'])
    words = title.split()
    doc_id = index
    for i in range(len(words)-1):
        biword = words[i] + " " + words[i+1]
        if biword in biword_index:
            biword_index[biword].append((doc_id, i))
        else:
            biword_index[biword] = [(doc_id, i)]

# Create the positional index
positional_index = {}
for index, row in data.iterrows():
    title = cleaned_text(row['Title'])
```



```
    title = cleaned_text(row['Title'])
    words = title.split()
    doc_id = index
    for i, word in enumerate(words):
        if word in positional_index:
            positional_index[word].append((doc_id, i))
        else:
            positional_index[word] = [(doc_id, i)]

# Prompt for user input
query = input("Enter a phrase query: ")
rank = input("Do you want to rank the results by position (y/n)? ")

# Find candidate documents using biword index
query_words = query.lower().split()
candidate_docs = set()
for i in range(len(query_words)-1):
    biword = query_words[i] + " " + query_words[i+1]
    if biword in biword_index:
        for doc_id, pos in biword_index[biword]:
            candidate_docs.add(doc_id)

# Find matching documents and positions using positional index
matched_docs = []
for doc_id in candidate_docs:
    positions = []
    for i, word in enumerate(query_words):
        if i == 0:
```

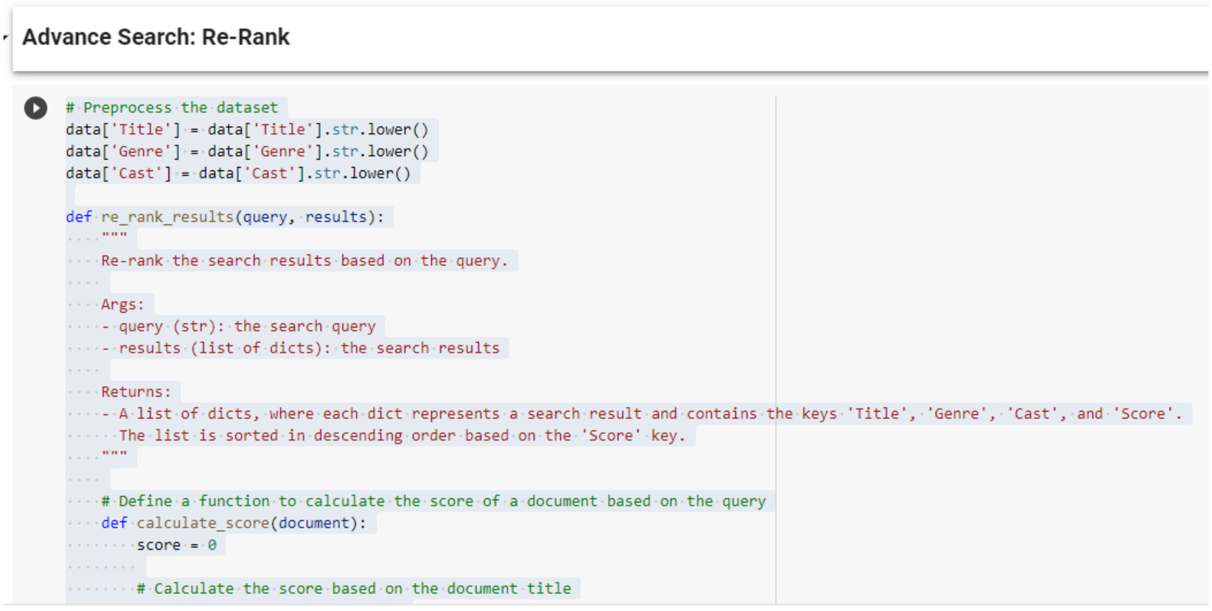


```
        if i == 0:
            positions = [pos for doc, pos in positional_index[word] if doc == doc_id]
        else:
            new_positions = []
            for pos in positions:
                if (doc_id, pos+i) in positional_index[word]:
                    new_positions.append(pos+i)
            positions = new_positions
        if not positions:
            break
    if positions:
        matched_docs.append((doc_id, positions))

# Sort by number of matching positions if ranking is requested
if rank == 'y':
    if matched_docs:
        matched_docs.sort(key=lambda x: len(x[1]), reverse=True)
else:
    matched_docs.sort()

# Display results
if matched_docs:
    print("Matching documents:")
    for doc_id, positions in matched_docs:
        title = data.iloc[doc_id]['Title']
        print(f"Doc ID: {doc_id}, Title: {title}, Matched positions: {positions}")
else:
    print("No matching documents found.")
```

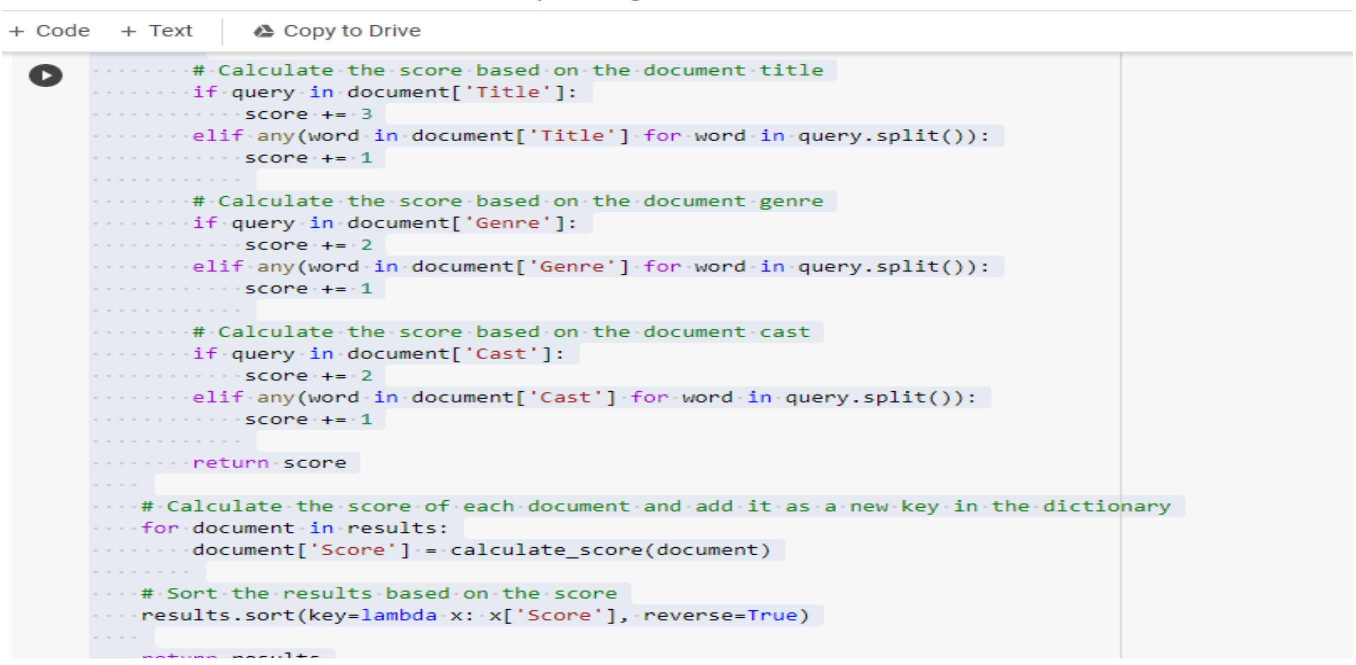
```
Enter a phrase query: the dark
Do you want to rank the results by position (y/n)? y
Matching documents:
Doc ID: 480, Title: 383. Dancer in the Dark (2000), Matched positions: [4]
Doc ID: 2, Title: 3. The Dark Knight (2008), Matched positions: [2]
Doc ID: 580, Title: 383. Dancer in the Dark (2000), Matched positions: [4]
Doc ID: 680, Title: 383. Dancer in the Dark (2000), Matched positions: [4]
Doc ID: 780, Title: 383. Dancer in the Dark (2000), Matched positions: [4]
Doc ID: 880, Title: 383. Dancer in the Dark (2000), Matched positions: [4]
Doc ID: 980, Title: 383. Dancer in the Dark (2000), Matched positions: [4]
Doc ID: 380, Title: 383. Dancer in the Dark (2000), Matched positions: [4]
Doc ID: 62, Title: 63. The Dark Knight Rises (2012), Matched positions: [2]
```

Advance Search: Re-Rank

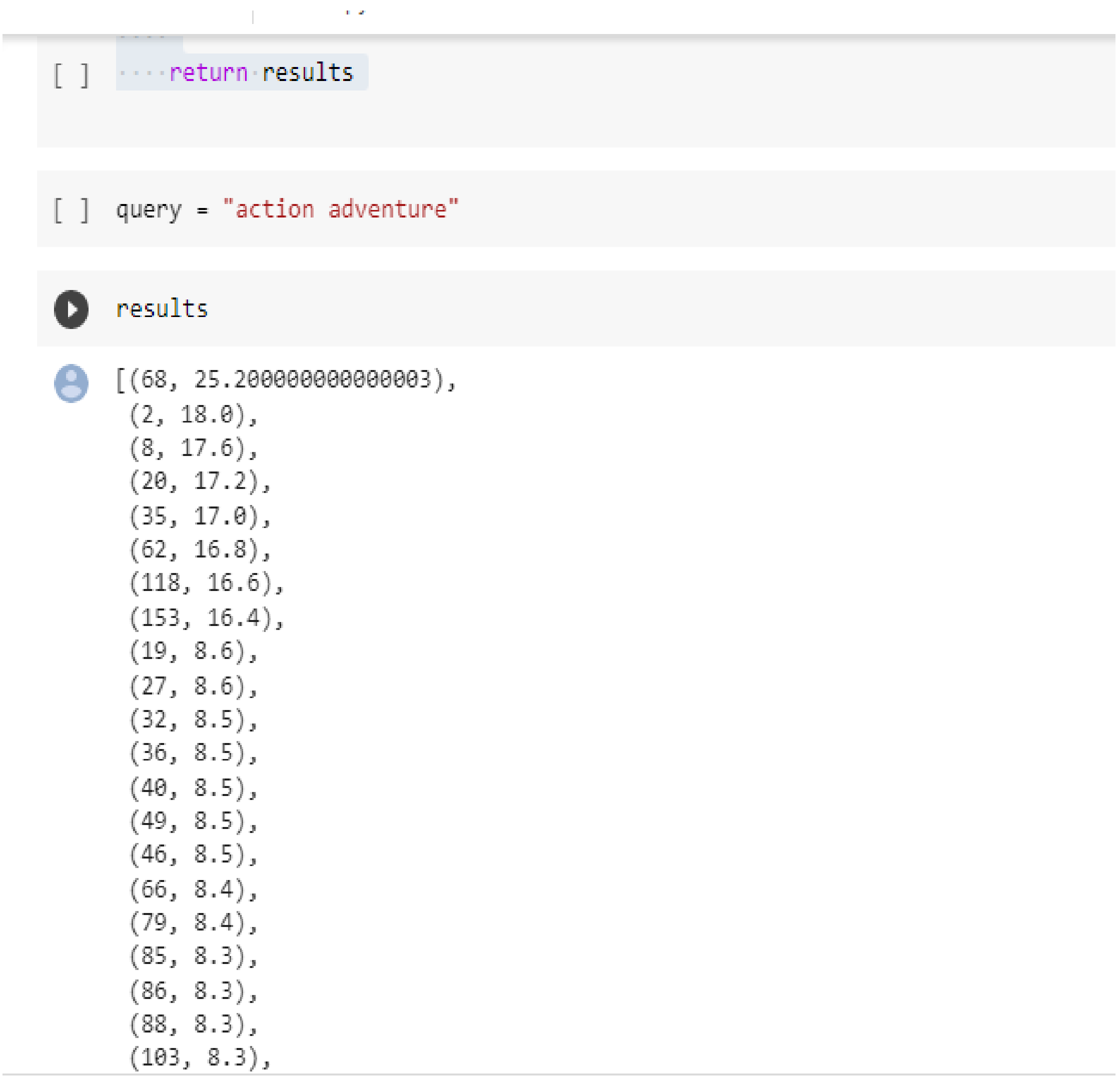
- The code defines a function called re_rank_results that takes a search query and a list of search results as input, and returns a list of the same search results re-ranked based on their relevance to the query.
- Before the re-ranking, the input dataset is preprocessed by converting the 'Title', 'Genre', and 'Cast' columns to lowercase.
- The re_rank_results function uses a nested function called calculate_score to assign a score to each search result based on its relevance to the search query. The calculate_score function takes a single search result as input and calculates its score based on how many times the search query appears in the 'Title', 'Genre', and 'Cast' columns of the search result. A higher score means the search result is more relevant to the search query.
- Finally, the function sorts the list of search results in descending order based on the score and returns the sorted list. The search results in the returned list are ordered from the most relevant to the least relevant to the search query.


Advance Search: Re-Rank

```
# Preprocess the dataset
data['Title'] = data['Title'].str.lower()
data['Genre'] = data['Genre'].str.lower()
data['Cast'] = data['Cast'].str.lower()

def re_rank_results(query, results):
    """
    Re-rank the search results based on the query.

    Args:
    - query (str): the search query
    - results (list of dicts): the search results

    Returns:
    - A list of dicts, where each dict represents a search result and contains the keys 'Title', 'Genre', 'Cast', and 'Score'.
      The list is sorted in descending order based on the 'Score' key.
    """

    # Define a function to calculate the score of a document based on the query
    def calculate_score(document):
        score = 0

        # Calculate the score based on the document title
```



```
        # Calculate the score based on the document title
        if query in document['Title']:
            score += 3
        elif any(word in document['Title'] for word in query.split()):
            score += 1

        # Calculate the score based on the document genre
        if query in document['Genre']:
            score += 2
        elif any(word in document['Genre'] for word in query.split()):
            score += 1

        # Calculate the score based on the document cast
        if query in document['Cast']:
            score += 2
        elif any(word in document['Cast'] for word in query.split()):
            score += 1

        return score

    # Calculate the score of each document and add it as a new key in the dictionary
    for document in results:
        document['Score'] = calculate_score(document)

    # Sort the results based on the score
    results.sort(key=lambda x: x['Score'], reverse=True)

```

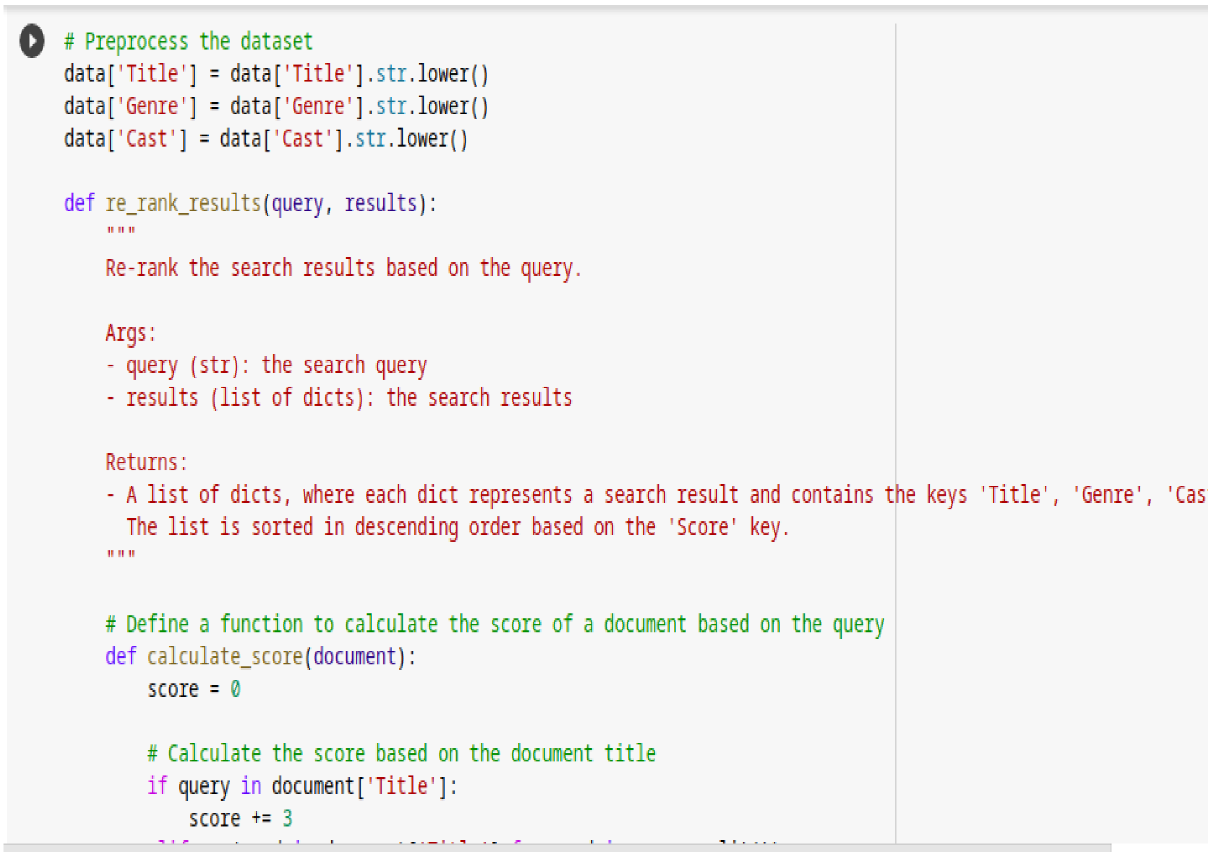


```
    return results
```

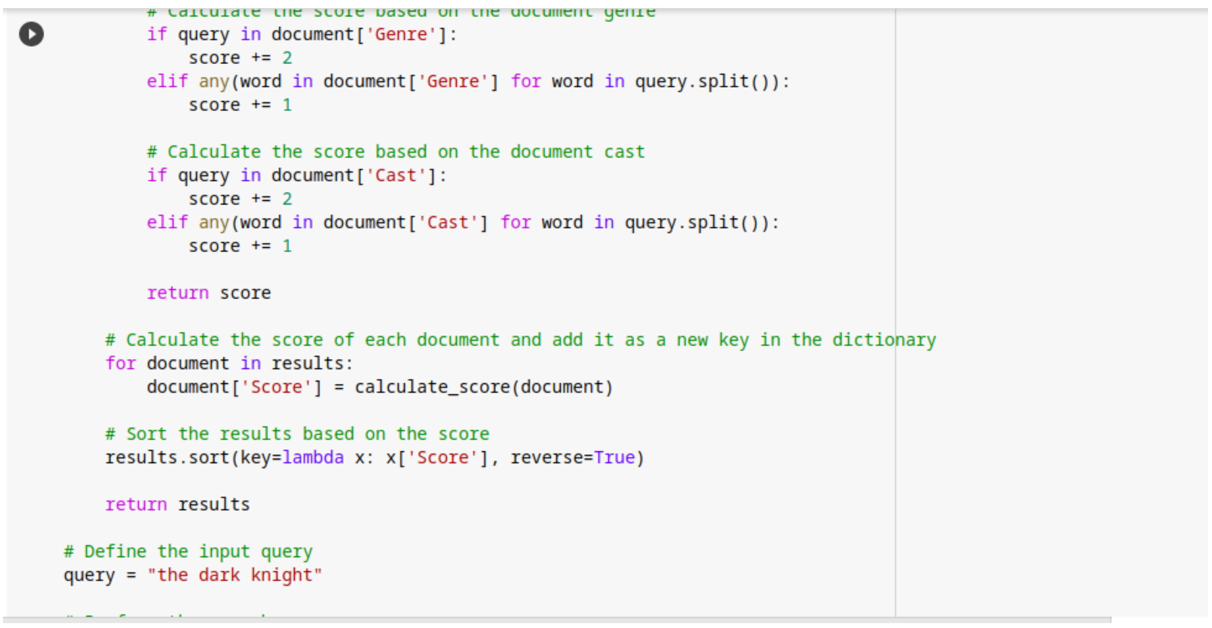
```
query = "action adventure"
```

```
results
```

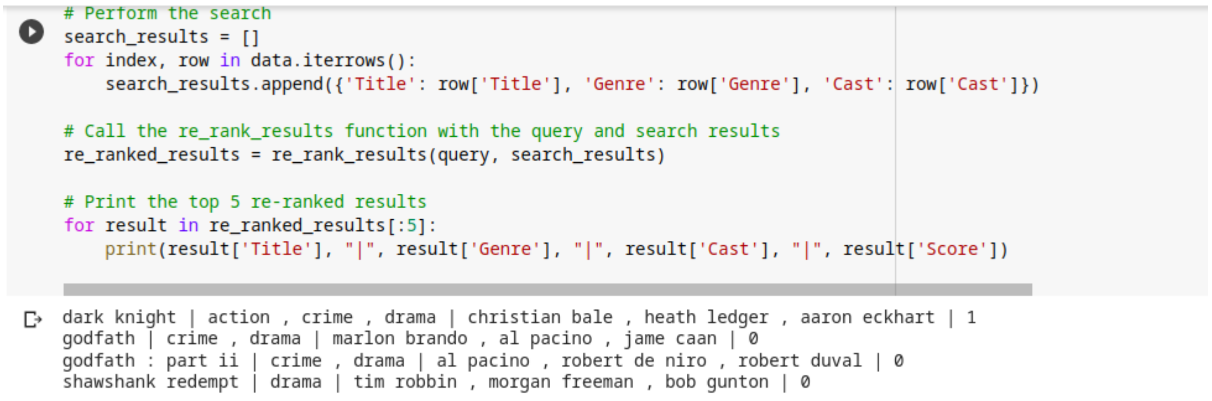
```
[(68, 25.200000000000003),
 (2, 18.0),
 (8, 17.6),
 (20, 17.2),
 (35, 17.0),
 (62, 16.8),
 (118, 16.6),
 (153, 16.4),
 (19, 8.6),
 (27, 8.6),
 (32, 8.5),
 (36, 8.5),
 (40, 8.5),
 (49, 8.5),
 (46, 8.5),
 (66, 8.4),
 (79, 8.4),
 (85, 8.3),
 (86, 8.3),
 (88, 8.3),
 (103, 8.3),
```



```python
# Preprocess the dataset
data['Title'] = data['Title'].str.lower()
data['Genre'] = data['Genre'].str.lower()
data['Cast'] = data['Cast'].str.lower()

def re_rank_results(query, results):
    """
    Re-rank the search results based on the query.

    Args:
    - query (str): the search query
    - results (list of dicts): the search results

    Returns:
    - A list of dicts, where each dict represents a search result and contains the keys 'Title', 'Genre', 'Cas
      The list is sorted in descending order based on the 'Score' key.
    """

    # Define a function to calculate the score of a document based on the query
    def calculate_score(document):
        score = 0

        # Calculate the score based on the document title
        if query in document['Title']:
            score += 3
```



```python
        # Calculate the score based on the document genre
        if query in document['Genre']:
            score += 2
        elif any(word in document['Genre'] for word in query.split()):
            score += 1

        # Calculate the score based on the document cast
        if query in document['Cast']:
            score += 2
        elif any(word in document['Cast'] for word in query.split()):
            score += 1

        return score

    # Calculate the score of each document and add it as a new key in the dictionary
    for document in results:
        document['Score'] = calculate_score(document)

    # Sort the results based on the score
    results.sort(key=lambda x: x['Score'], reverse=True)

    return results

# Define the input query
query = "the dark knight"
```



```python
# Perform the search
search_results = []
for index, row in data.iterrows():
    search_results.append({'Title': row['Title'], 'Genre': row['Genre'], 'Cast': row['Cast']})

# Call the re_rank_results function with the query and search results
re_ranked_results = re_rank_results(query, search_results)

# Print the top 5 re-ranked results
for result in re_ranked_results[:5]:
    print(result['Title'], "|", result['Genre'], "|", result['Cast'], "|", result['Score'])
```

```
dark knight | action , crime , drama | christian bale , heath ledger , aaron eckhart | 1
godfath | crime , drama | marlon brando , al pacino , jame caan | 0
godfath : part ii | crime , drama | al pacino , robert de niro , robert duval | 0
shawshank redempt | drama | tim robbin , morgan freeman , bob gunton | 0
```