**AIWR ASSIGNMENT -1**

**Team Members:**

| SHARATH K | PES2UG20CS552 |
| --- | --- |
| NITHISH S | PES2UG20CS531 |
| PRATHAM HEGDE | PES2UG20CS536 |
| MANIK | PES2UG20CS576 |

Assignment github link: <https://github.com/sharathkrishnegowda/AIWR_Assignment1>

**Corpus Details and Source:**

Used Dataset: **IMDB Top 250 Movies**

Dataset Link: <https://www.kaggle.com/datasets/rajugc/imdb-top-250-movies-dataset>

Dataset Description:

This dataset contains the top 250 rated movies on IMDB as of 2021, providing a snapshot of the most popular and highly rated movies of recent times. By analyzing this dataset, one can gain insights into the movie industry, such as trends in movie ratings and popular genres

Data structures used:

In order to create a search engine we created a dictionary which contains an array where token is stored as key and document IDs are stored as array of values

Set of free text wild card queries, Phrase queries

Wildcard Queries :

Input string: ‘picture\*’

Phrase queries:

Input string: “day”

**Pre Processing of raw data:**

import nltk

import numpy as np

import pandas as pd

nltk.download("punkt")

nltk.download("stopwords")

nltk.download("wordnet")

nltk.download('omw-1.4')

from nltk.tokenize import word\_tokenize

stopwords = nltk.corpus.stopwords.words('english')

stemmer = nltk.stem.SnowballStemmer('english')

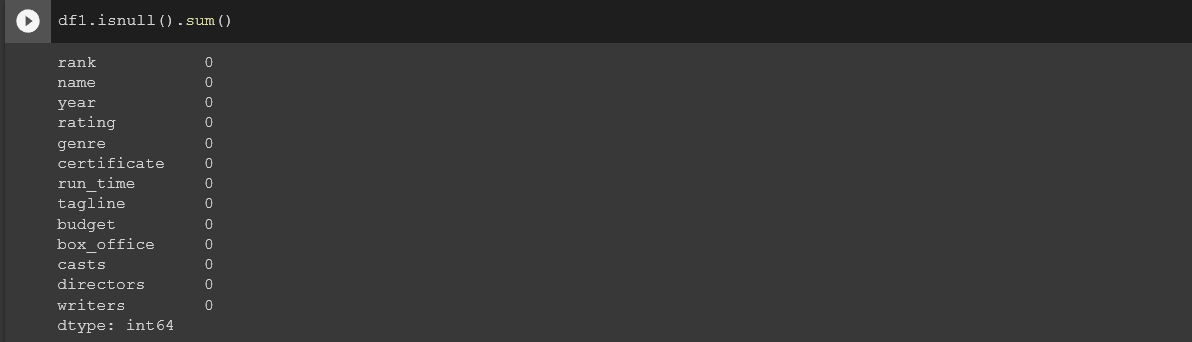
lemmatizer = nltk.stem.WordNetLemmatizer()

from google.colab import drive

drive.mount('/content/drive')

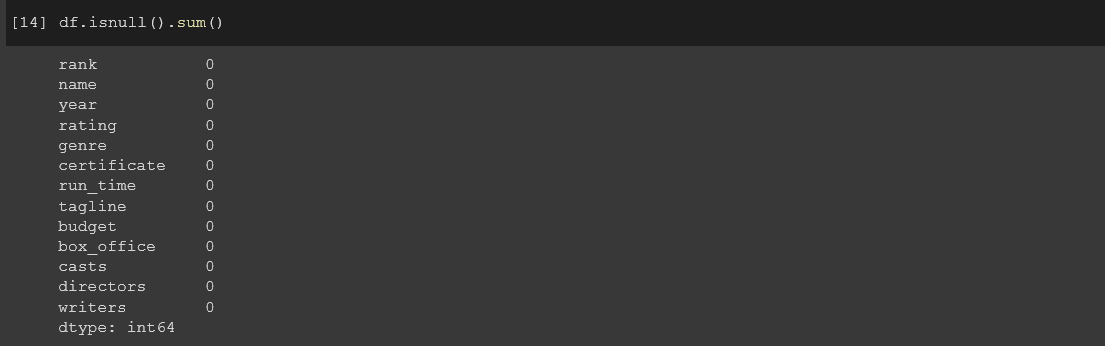
df1=pd.read\_csv('/content/drive/MyDrive/Colab Notebooks/AIWR/IMDB Top 250 Movies.csv')

df1.isnull().sum()



df = df1.dropna()

df.isnull().sum()



df.head()



"""TOKENIZING THE \*\*CORPUS\*\*"""

Tokenizing a corpus means breaking down the textual data into individual words or tokens. In other words, tokenization is the process of splitting a text into smaller units called tokens, which are usually words but can also be subwords or characters, depending on the specific tokenization approach.

The function **tokenizing\_corpus()** which takes a parameter type (which should be a list of strings) and applies the **word\_tokenize()** function from the nltk library to each string in the list using the map() function. The result is a list of lists where each inner list contains the tokenized words of a single string

**Code**:

corpus = df["tagline"].values

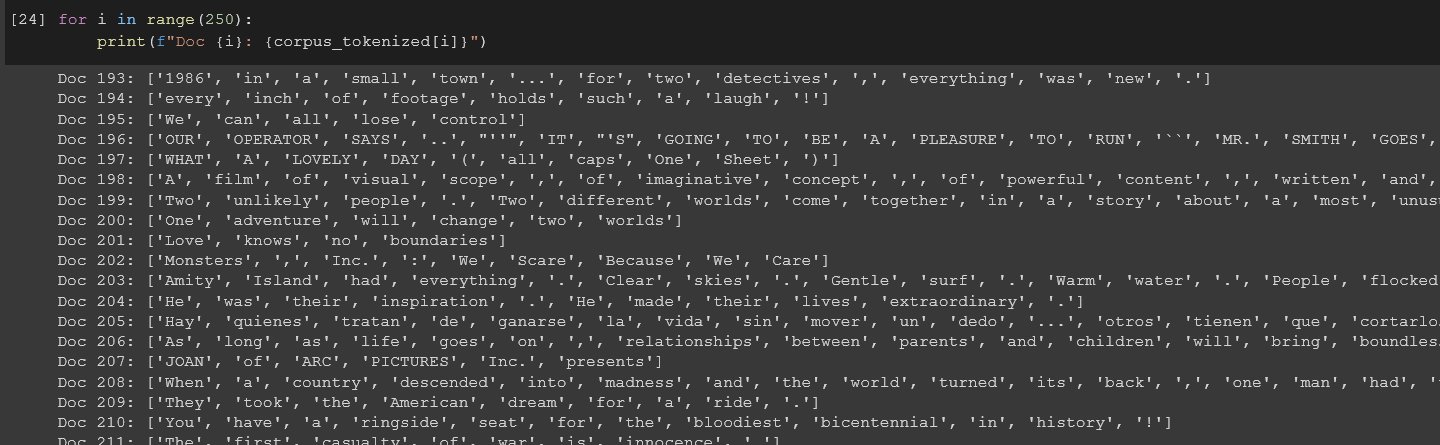
def tokenizing\_corpus(type):

return list(map(word\_tokenize, type))

corpus\_tokenized = tokenizing\_corpus(corpus)

for i in range(250):

print(f"Doc {i}: {corpus\_tokenized[i]}")



"""REMOVING THE STOPWORDS"""

Stopwords are often removed from text data because they are commonly used words that do not carry much meaning and can often be ignored without affecting the overall meaning of the text. Examples of stopwords include "the", "and", "a", "an", "in", "of", etc.

In this code we are removing stop words which is an important part of preprocessing.

**Code:**

verify=[',',':',';','.','#','(',')','{','[',']',"}",'/','?']

def removing\_stopwords(tokenized\_review):

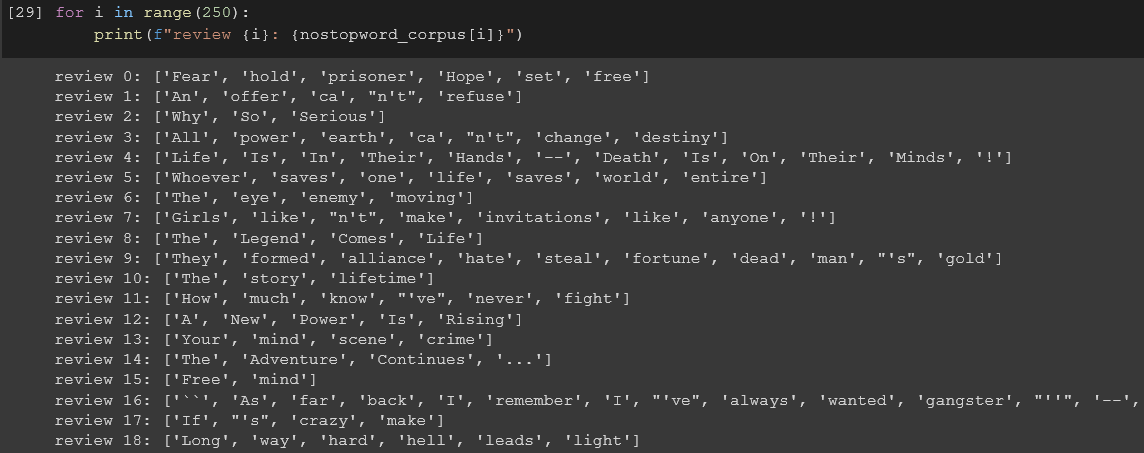
\_function = lambda review: [word for word in review if word not in stopwords and word not in verify]

return list(map(\_function, tokenized\_review))

nostopword\_corpus = removing\_stopwords(corpus\_tokenized)

for i in range(250):

print(f"review {i}: {nostopword\_corpus[i]}")



"""**CASE FOLDING**"""

This code performs a normal step in preprocessing called case folding which involves converting all the words in the text to lowercase.

By converting all words to lowercase, we eliminate any differences in case that might appear due to differences in how the text was originally written. For example, "The", "the", and "tHE" would all be converted to "the", making it easier to compare and analyze them.

**Code**:

def case\_folding(tokenized\_reviews):

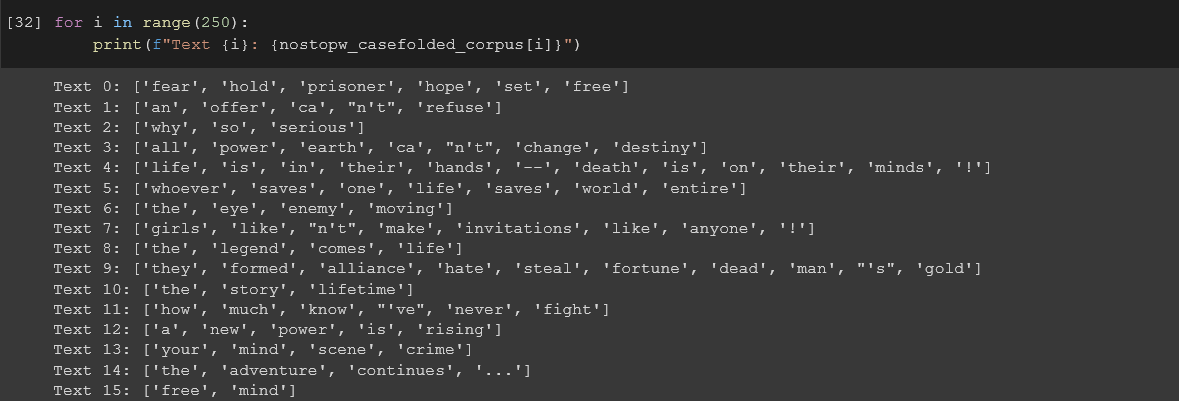
\_function = lambda review: [word.lower() for word in review]

return list(map(\_function, tokenized\_reviews))

nostopw\_casefolded\_corpus = case\_folding(nostopword\_corpus)

for i in range(250):

print(f"Text {i}: {nostopw\_casefolded\_corpus[i]}")



"""LEMMATIZING WORDS"""

This code performs another important preprocessing step called "lemmatization", which involves converting words to their base or dictionary form

By converting words to their lemmas, we can reduce the overall number of unique words in the text and capture more of the underlying meaning.

**Code**:

def lemmatize\_words(tokenized\_reviews):

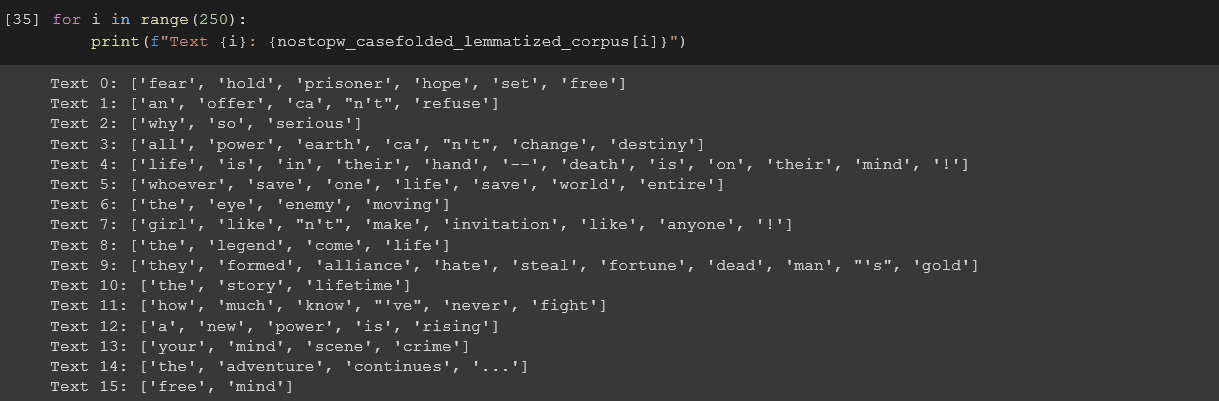
function = lambda review: [lemmatizer.lemmatize(word) for word in review]

return list(map(function, tokenized\_reviews))

nostopw\_casefolded\_lemmatized\_corpus = lemmatize\_words(nostopw\_casefolded\_corpus)

for i in range(250):

print(f"Text {i}: {nostopw\_casefolded\_lemmatized\_corpus[i]}")



"""STEMMING """

def stem\_words(tokenized\_reviews):

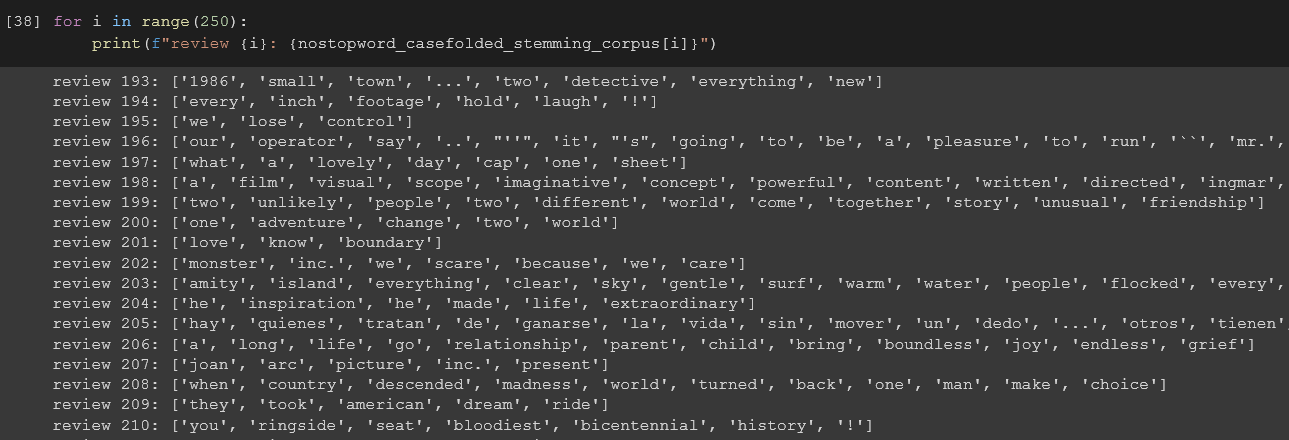
function = lambda reviews: [stemmer.stem(word) for word in reviews]

return list(map(function, tokenized\_reviews))

nostopword\_casefolded\_stemming\_corpus = lemmatize\_words(nostopw\_casefolded\_lemmatized\_corpus)

for i in range(250):

print(f"review {i}: {nostopword\_casefolded\_stemming\_corpus[i]}")



**Implementation of Inverted Index:**

The function **create\_invert** takes a list of tokenized reviews and returns an inverted index, which is a dictionary that maps each word in the reviews to a list of indices of the reviews in which the word appears.

**Code:**

def create\_invert(tokenized\_reviews):

inverted\_index = dict()

for index in range(len(tokenized\_reviews)):

for word in tokenized\_reviews[index]:

if word not in inverted\_index:

inverted\_index[word] = list()

inverted\_index[word].append(index)

for key in inverted\_index:

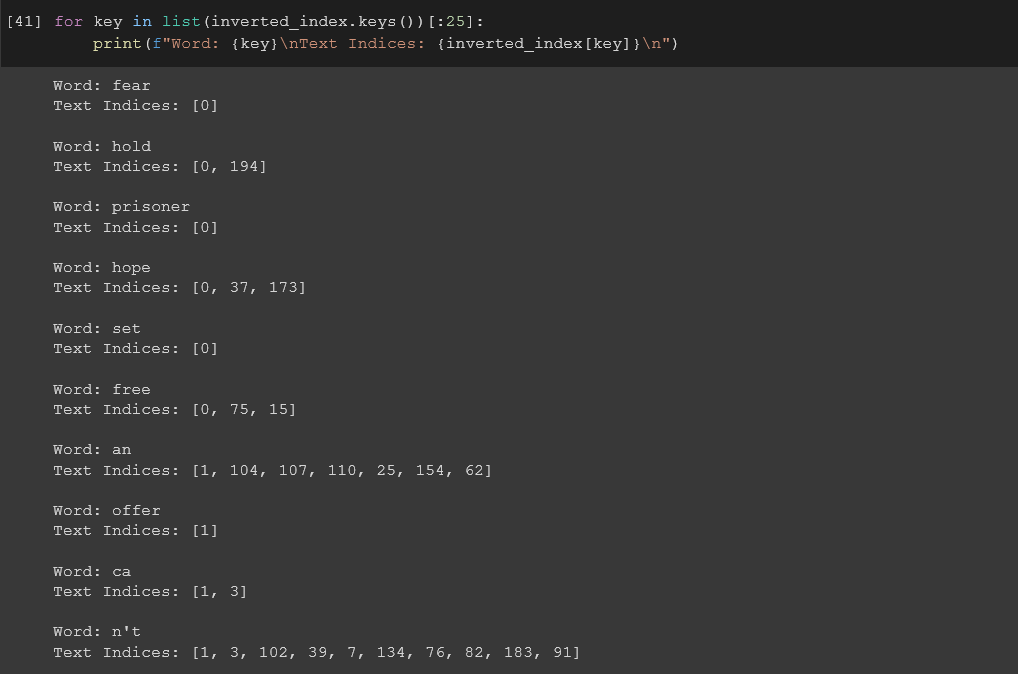
inverted\_index[key] = list(set(inverted\_index[key]))

return inverted\_index

inverted\_index = create\_invert(nostopword\_casefolded\_stemming\_corpus)

for key in list(inverted\_index.keys())[:25]:

print(f"Word: {key}\nText Indices: {inverted\_index[key]}\n")



Boolean Queries:

The function **boolean\_query** takes a query string and an inverted index dictionary, and returns a set of document ids that match the query.

Overall, the function allows users to perform boolean queries on a set of documents using the inverted index, which enables fast lookup of documents containing specific words

We write a query using AND,OR and this returns the document id that match.

**Code**:

def query\_parsed(infix):

order = {}

order['('] = 0

order[')'] = 0

order['OR'] = 1

order['NOT'] = 3

order['AND'] = 2

output = []

stack = []

for token in infix:

if (token == '('):

stack.append(token)

elif (token == ')'):

operator = stack.pop()

while operator != '(':

output.append(operator)

operator = stack.pop()

elif (token in order):

if (stack):

current\_operator = stack[-1]

while (stack and order[current\_operator] > order[token]):

output.append(stack.pop())

if (stack):

current\_operator = stack[-1]

stack.append(token) # add token to stack

else:

output.append(token.lower())

while (stack):

output.append(stack.pop())

return output

def boolean\_query(query, inverted\_index):

query = query.strip()

query\_tokens = query.split()

boolean\_query = query\_parsed(query\_tokens)

result\_stack = list()

for idx, token in enumerate(boolean\_query):

if token not in ["AND", "NOT", "OR"]:

result = set(inverted\_index[token])

else:

if token in ['AND', 'OR']:

right\_operand = result\_stack.pop()

left\_operand = result\_stack.pop()

if token == 'AND':

operation = set.intersection

else:

operation = set.union

result = operation(left\_operand, right\_operand)

else:

operand = result\_stack.pop()

complement\_document\_ids = inverted\_index[boolean\_query[idx-1]]

result = list()

for word in inverted\_index:

result.extend([\_id for \_id in inverted\_index[word] if \_id not in complement\_document\_ids])

result = set(result)

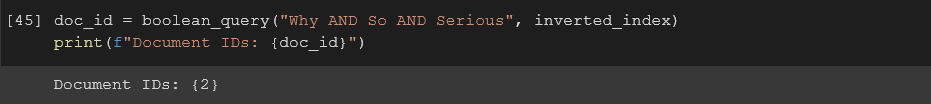
result\_stack.append(result)

return result\_stack.pop()

"""TESTING WITH QUERIES"""

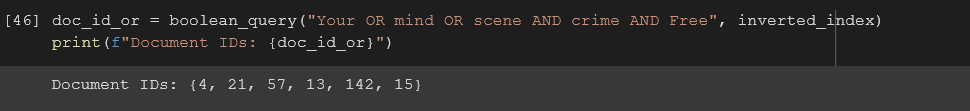
doc\_id = boolean\_query("Why AND So AND Serious", inverted\_index)

print(f"Document IDs: {doc\_id}")



doc\_id\_or = boolean\_query("Your OR mind OR scene AND crime AND Free", inverted\_index)

print(f"Document IDs: {doc\_id\_or}")



Handling wild card and phrase queries:

Wild card queries allow us to search for words or phrases with variations, such as different spellings or forms of a word.

Phrase queries, on the other hand, allow us to search for words or phrases in a specific order, which can be important for finding relevant information on a specific topic.

In the **first screenshot** we have used wild card queries,we used the word **picture\***

this gives us the document ids which have words either starting or ending with picture

In the **second screenshot** we have used phrase queries,we use the word **day** this gives us the document ids which have the exact **word** day in it

**Code**:

def search(query):

if '\*' in query:

query = query.replace('\*', '')

result = []

for word in inverted\_index:

if query in word:

result.extend(inverted\_index[word])

return result

elif '"' in query:

query = query.replace('"', '')

result = []

for word in inverted\_index:

if query == word:

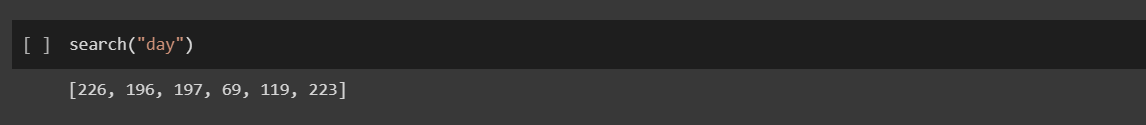
result.extend(inverted\_index[word])

return result

else:

return inverted\_index[query]





Retrieve relevant text using similarity index:

The similarity index is a measure of how closely a query matches a document in a collection. It is used to rank the documents in order of relevance to a given query.

def retrieve\_relevant\_text(query):

docs = search(query)

# get the relevant text

relevant\_text = []

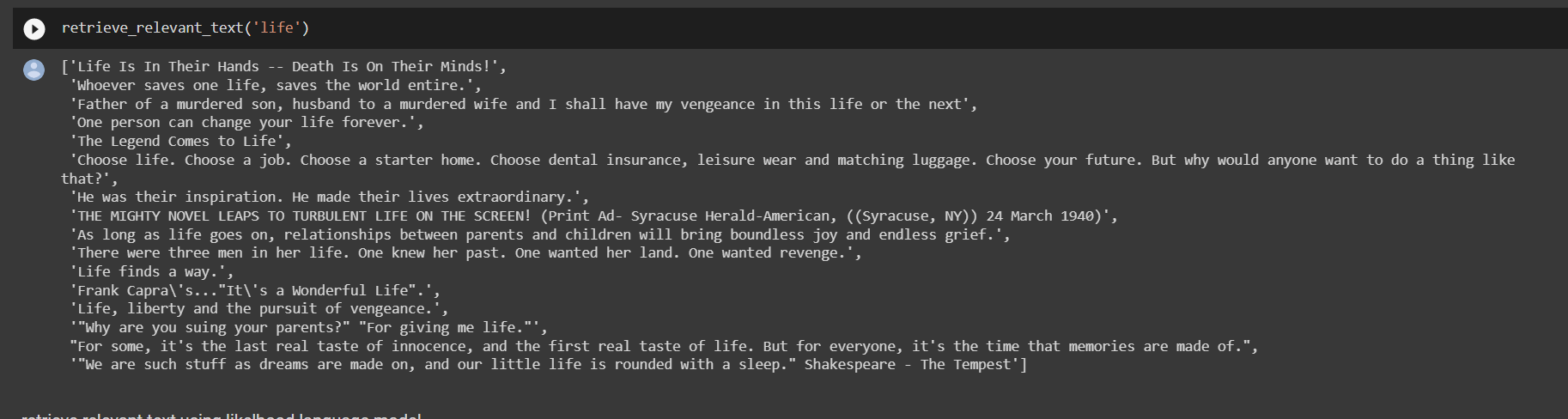
for doc in docs:

relevant\_text.append(df.loc[df['id'] == doc]['tagline'].values[0])

return relevant\_text

df['id'] = range(0, len(df))

retrieve\_relevant\_text('life')



Ranking of retrieved documents:

This code takes a query as input, searches for documents that match the query using the search function, retrieves the relevant text for each document, calculates a score for each document based on the frequency of query terms in the relevant text, and finally sorts the documents based on their scores

def rank\_retrieved\_documents(query):

# handle wildcard and phase queries

docs = search(query)

# get the relevant text

relevant\_text = []

for doc in docs:

relevant\_text.append(df.loc[df['id'] == doc]['tagline'].values[0])

# calculate the score for each document

scores = {}

for doc in relevant\_text:

scores[doc] = 0

for word in query.split():

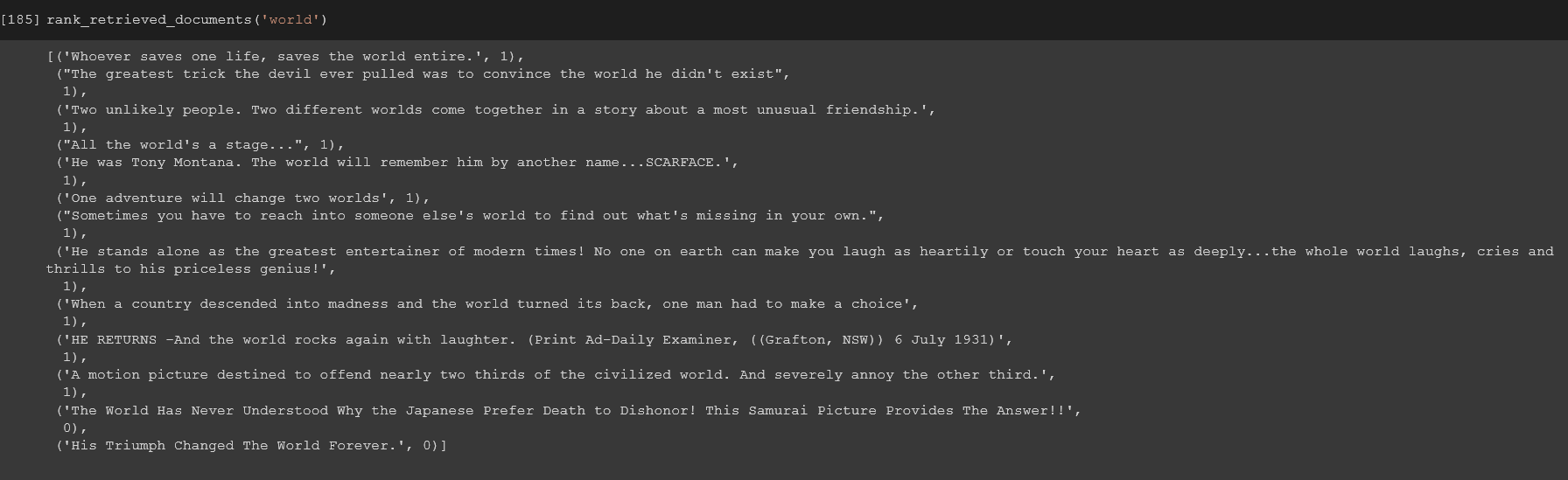
scores[doc] += doc.count(word)

# sort the documents based on the score

sorted\_scores = sorted(scores.items(), key=lambda x: x[1], reverse=True)

return sorted\_scores

rank\_retrieved\_documents('world')



Advanced search: relevance feedback, semantic matching, reranking of results, finding out query intention:

def advanced\_search(query):

# get the relevant text

relevant\_text = retrieve\_relevant\_text(query)

# calculate the score for each document

scores = {}

for doc in relevant\_text:

scores[doc] = 0

for word in query.split():

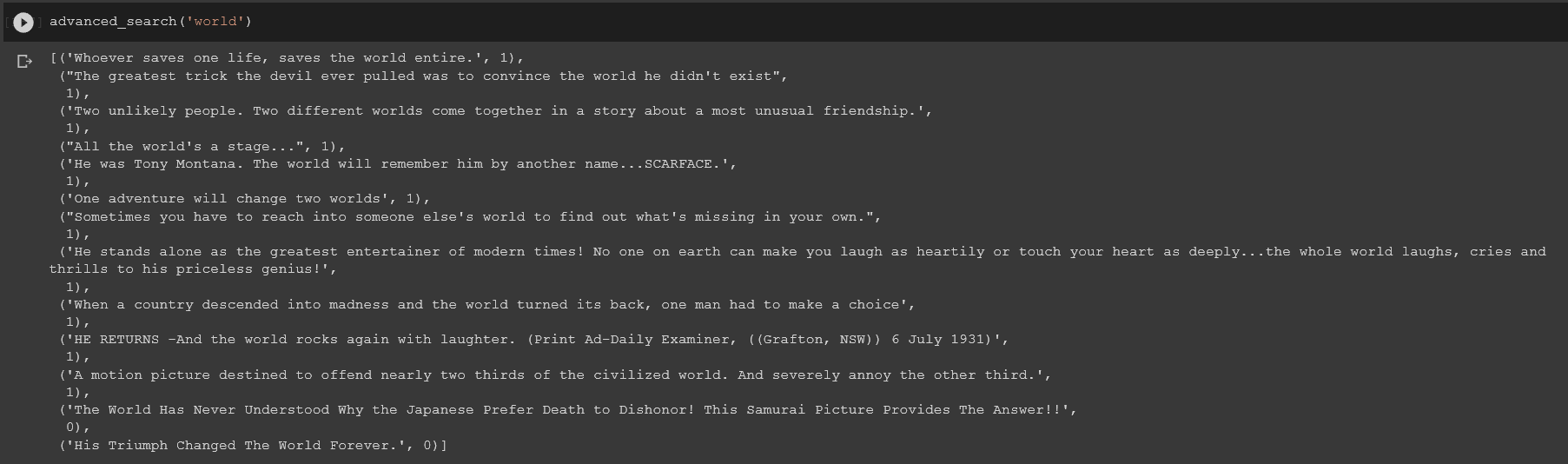
scores[doc] += doc.count(word)

# sort the documents based on the score

sorted\_scores = sorted(scores.items(), key=lambda x: x[1], reverse=True)

return sorted\_scores

advanced\_search('world')



"""RERANKING RESULTS"""

Reranking of results refers to the process of reordering the initial list of retrieved documents based on a new ranking model. The aim of reranking is to improve the quality of the search results by elevating the relevance of more relevant documents and demoting the relevance of less relevant documents.

def rerank\_results(query):

index = inverted\_index

documents = search(query)

# Create a list to store document scores

scores = []

# Split the query into individual terms

query\_terms = query.split()

# Iterate over each document

for doc\_id in documents:

# Initialize the score for this document

score = 0

# Iterate over each query term

for term in query\_terms:

# If the term appears in the document

if term in index and doc\_id in index[term]:

# Increment the score by the frequency of the term in the document

try:

score += index[term][doc\_id]

except:

pass

# Add the document score to the list of scores

scores.append((doc\_id, score))

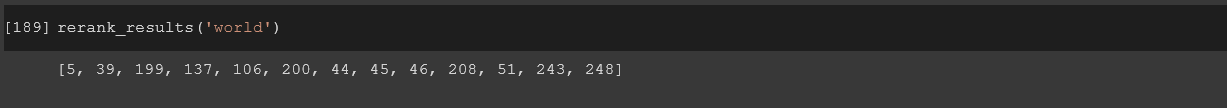
# Sort the list of scores in descending order

scores.sort(key=lambda x: x[1], reverse=True)

# Return the sorted list of document IDs

return [doc\_id for doc\_id, score in scores]

rerank\_results('world')



"""SEMANTIC MATCHING"""

Semantic matching refers to the process of comparing two pieces of text (such as a query and a document) and determining the degree to which they are related in meaning or intent. The goal of semantic matching is to identify documents or information that are relevant to a user's search query, even if they don't contain the exact same words as the query.

The code below uses the NLTK library to perform semantic matching between a query and a set of documents. Specifically, it calculates the semantic similarity between each query token and each document token using WordNet, and then aggregates the scores to calculate a similarity score for each document.

**Code**:

from nltk.corpus import wordnet as wn

from nltk.tokenize import word\_tokenize

# define a function to calculate semantic similarity between two words using WordNet

def calculate\_similarity(word1, word2):

synsets1 = wn.synsets(word1)

synsets2 = wn.synsets(word2)

if not synsets1 or not synsets2:

return 0.0

max\_sim = -1

for synset1 in synsets1:

for synset2 in synsets2:

sim = wn.path\_similarity(synset1, synset2)

if sim is not None and sim > max\_sim:

max\_sim = sim

return max\_sim

# define a function to perform semantic matching of a query against a document

def semantic\_matching(query):

documents = inverted\_index

scores = []

query\_tokens = word\_tokenize(query)

for document in documents:

doc\_tokens = word\_tokenize(document)

similarity\_score = 0.0

for query\_token in query\_tokens:

max\_sim = -1

for doc\_token in doc\_tokens:

sim = calculate\_similarity(query\_token, doc\_token)

if sim > max\_sim:

max\_sim = sim

similarity\_score += max\_sim

scores.append((document, similarity\_score / len(query\_tokens)))

return scores

semantic\_matching('why so serious')

