|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Practical No. 1**

**Title: Document Indexing and Retrieval**

**Aim: To implement a basic document retrieval system using an inverted index and use it to efficiently search for documents containing specific terms.**

Given a set of documents, build an inverted index data structure that maps each unique term to the documents in which it appears. The program should allow users to input a query consisting of one or more terms and retrieve the documents that contain all the terms in the query. The program should provide users as output, the relevant documents based on the terms present in their query.

**Program**

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

# Define the documents documents = {

"Document 1": "The quick brown fox jumped over the lazy dog", "Document 2": "The lazy dog slept in the sun",

"Document 3": "The quick brown fox and the lazy dog are friends", "Document 4": "The lazy cat watched the lazy dog",

"Document 5": "The quick brown fox ran through the forest"

}

# Step 0: Import NLTK stopwords stopWords = stopwords.words('english')

# Step 1: Tokenize the documents tokens = {}

for doc\_id, doc\_text in documents.items(): tokens[doc\_id] = set(word\_tokenize(doc\_text.lower()))

# Step 2: Build the inverted index inverted\_index = {}

for doc\_id, doc\_tokens in tokens.items(): for term in doc\_tokens:

if term not in stopWords:

if term not in inverted\_index: inverted\_index[term] = set()

inverted\_index[term].add(doc\_id)

# Step 3: Document Retrieval System def search\_documents(query):

query\_terms = set(word\_tokenize(query.lower())) - set(stopWords) matching\_documents = set()

for term in query\_terms:

term\_documents = inverted\_index.get(term, set()) if not matching\_documents:

matching\_documents = term\_documents else:

matching\_documents = matching\_documents.intersection(term\_documents)

return matching\_documents

# Step 4: User Input and Query Processing

user\_query = input("Enter your query").strip().lower()

matching\_docs = search\_documents(user\_query) if matching\_docs:

print(f"\nDocuments containing the terms '{user\_query}':

{matching\_docs}")

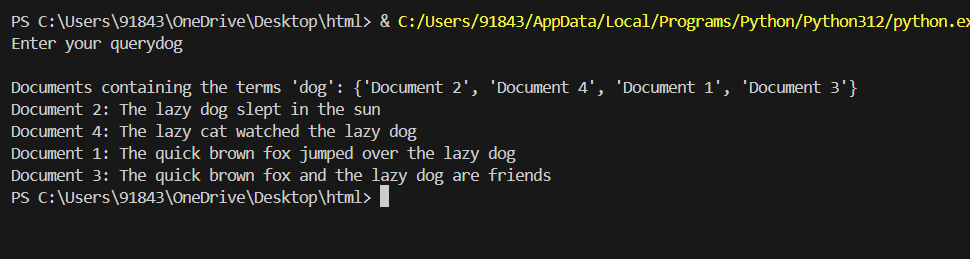
for doc\_id in matching\_docs:

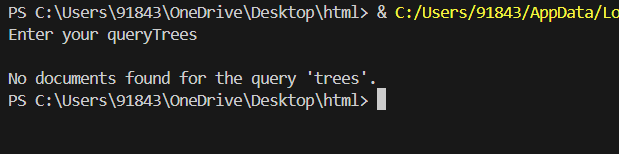
print(f"{doc\_id}: {documents[doc\_id]}")

else:

print(f"\nNo documents found for the query '{user\_query}'.")

**Output:**





**Practical No. 2**

**Title: Retrieval Models (Boolean Model and Vector Space Model) Aim:**

1. **to implement a simple Boolean retrieval system using an inverted index.**
2. **to implement document retrieval using the TF-IDF (Term Frequency-Inverse Document Frequency) vectorization technique and cosine similarity.**

Develop a basic information retrieval system capable of handling Boolean queries over a collection of text documents. The system should provide functionality for performing Boolean AND, OR, and NOT operations on terms extracted from the documents. The goal is to efficiently retrieve relevant documents that match the user’s query based on the presence or absence of specific terms.

**Program**

from nltk.tokenize import word\_tokenize from nltk.corpus import stopwords

# Example documents documents = [

"The quick brown fox jumped over the lazy dog", "The lazy dog slept in the sun",

"The quick brown dog chased the squirrel up the tree", "The sun shines bright in the sky",

"The brown fox is quick and agile"

]

# Tokenize and create the inverted index def build\_inverted\_index(documents):

inverted\_index = {}

stop\_words = set(stopwords.words('english'))

for doc\_id, document in enumerate(documents, start=1):

words = set(word\_tokenize(document.lower())) - stop\_words for word in words:

if word not in inverted\_index: inverted\_index[word] = []

inverted\_index[word].append(doc\_id) return inverted\_index

# Boolean AND operation using inverted index def boolean\_and(operands, index, total\_docs):

if not operands:

return list(range(1, total\_docs + 1))

result = set(index.get(operands[0], set())) for term in operands[1:]:

result = result.intersection(index.get(term, set())) return list(result)

# Boolean OR operation using inverted index def boolean\_or(operands, index):

result = set()

for term in operands:

result = result.union(index.get(term, set())) return list(result)

# Boolean NOT operation using inverted index def boolean\_not(operand, index, total\_docs):

operand\_set = set(index.get(operand, set())) all\_docs\_set = set(range(1, total\_docs + 1))

not\_result = all\_docs\_set.difference(operand\_set) return list(not\_result)

# Tokenize the user query and perform boolean operation based on the input def perform\_boolean\_operation(query, inverted\_index, total\_docs):

query\_terms = word\_tokenize(query.lower())

if 'and' in query\_terms:

operands = query.split(' AND ')

query\_result = boolean\_and(operands, inverted\_index, total\_docs) elif 'or' in query\_terms:

operands = query.split(' OR ')

query\_result = boolean\_or(operands, inverted\_index) elif 'not' in query\_terms:

operand = query.split('NOT ')[1]

query\_result = boolean\_not(operand, inverted\_index, total\_docs) else:

# Default to OR operation if no operator is specified operands = query\_terms

query\_result = boolean\_or(operands, inverted\_index) print(operands)

return query\_result

# Example usage

inverted\_index = build\_inverted\_index(documents) total\_docs = len(documents)

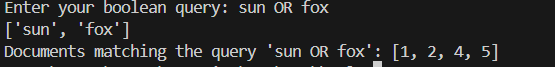
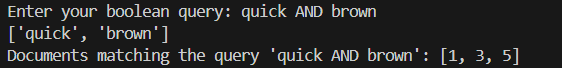
# Take query input from the user

query = input("Enter your boolean query: ")

# Perform boolean operation based on the user query

query\_result = perform\_boolean\_operation(query, inverted\_index, total\_docs) print(f"Documents matching the query '{query}': {query\_result}")

**Output:**



Given a query and a corpus of text documents, to rank the documents based on their similarity to the query. The relevance of a document should be determined by its similarity to the query, computed using the cosine similarity metric.

**Program**

from sklearn.feature\_extraction.text import TfidfVectorizer from sklearn.metrics.pairwise import cosine\_similarity from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

# Example documents documents = [

"The quick brown fox jumped over the lazy dog", "The lazy dog slept in the sun",

"The quick brown dog chased the squirrel up the tree", "The sun shines bright in the sky",

"A brown fox is quick and agile", "The lazy cat slept on the windowsill"

]

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

# Tokenize the documents and query stop\_words = set(stopwords.words('english'))

tokenized\_documents = [" ".join([word for word in word\_tokenize(doc.lower()) if word.isalnum() and word not in stop\_words]) for doc in documents] tokenized\_query = " ".join([word for word in word\_tokenize(query.lower()) if word.isalnum() and word not in stop\_words])

# TF-IDF Vectorization vectorizer = TfidfVectorizer()

tfidf\_matrix = vectorizer.fit\_transform(tokenized\_documents) query\_vector = vectorizer.transform([tokenized\_query])

# Calculate cosine similarity between the query vector and document vectors cosine\_similarities = cosine\_similarity(query\_vector, tfidf\_matrix).flatten()

# Display the cosine similarities print("Cosine Similarities:")

for i, similarity in enumerate(cosine\_similarities, start=1): print(f"Document {i}: {similarity}")

# Rank documents based on cosine similarity ranked\_indices = cosine\_similarities.argsort()[::-1]

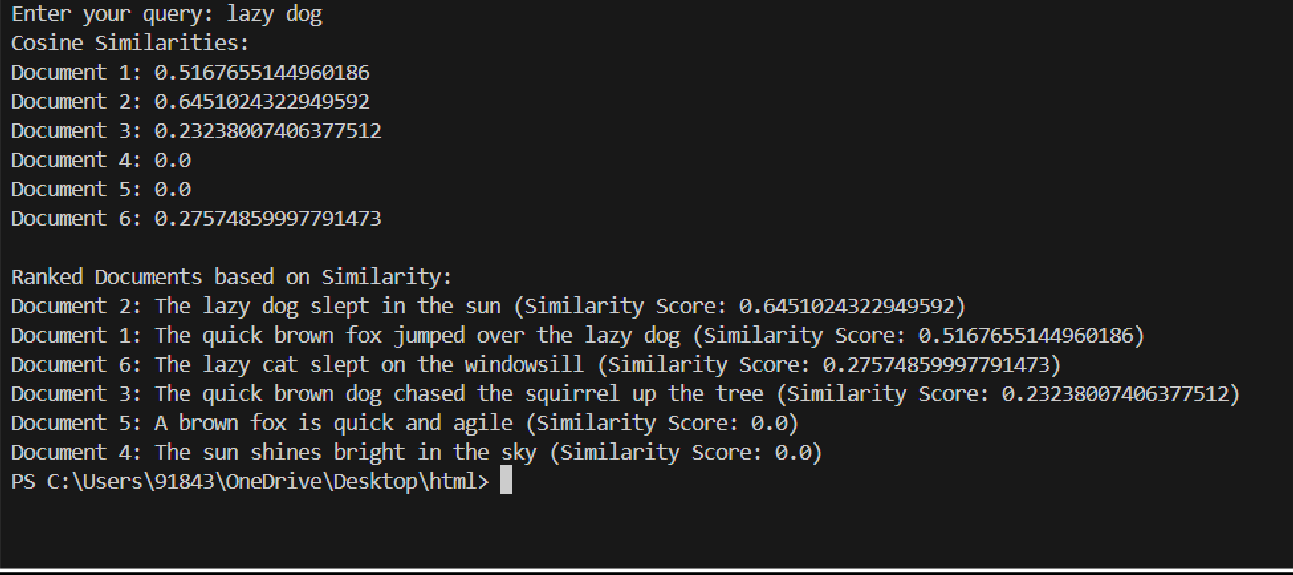
# Display the ranked documents

print("\nRanked Documents based on Similarity:") for index in ranked\_indices:

print(f"Document {index + 1}: {documents[index]} (Similarity Score:

{cosine\_similarities[index]})")

**Output:**



**Practical No. 3**

**Title: Spelling Correction in IR Systems**

**Aim: To implement an information retrieval system that corrects spelling using a spell-correcting algorithm (Levenshtein distance)**

Develop a program that can efficiently correct misspelled words in a user query to improve the accuracy of information retrieval. Given a set of documents and a user query containing potentially misspelled words, the program aims to identify and correct these misspellings using the Levenshtein distance algorithm.

**Program**

import numpy as np import nltk

from nltk.corpus import words

# Load the NLTK words corpus nltk.download('words') nltk\_words = set(words.words())

# Spelling correction function

def levenshtein\_distance(word1, word2): len1, len2 = len(word1), len(word2)

distances = np.zeros((len1 + 1, len2 + 1), dtype=int) for i in range(len1 + 1):

distances[i, 0] = i for j in range(len2 + 1):

distances[0, j] = j

for i in range(1, len1 + 1): for j in range(1, len2 + 1):

cost = 0 if word1[i - 1] == word2[j - 1] else 1 distances[i, j] = min(

distances[i - 1, j] + 1,

distances[i, j - 1] + 1, distances[i - 1, j - 1] + cost

)

return distances[len1, len2]

def correct\_spelling(input\_word, candidate\_words): min\_distance = float('inf')

corrected\_word = input\_word

for candidate in candidate\_words:

distance = levenshtein\_distance(input\_word, candidate) if distance < min\_distance:

min\_distance = distance corrected\_word = candidate

return corrected\_word

# Information retrieval function def search(query, dictionary):

# Split the query into individual words query\_terms = query.split()

# Spell-correct each query term

corrected\_query\_terms = [correct\_spelling(term, dictionary) for term in query\_terms]

#Return the corrected query terms return corrected\_query\_terms

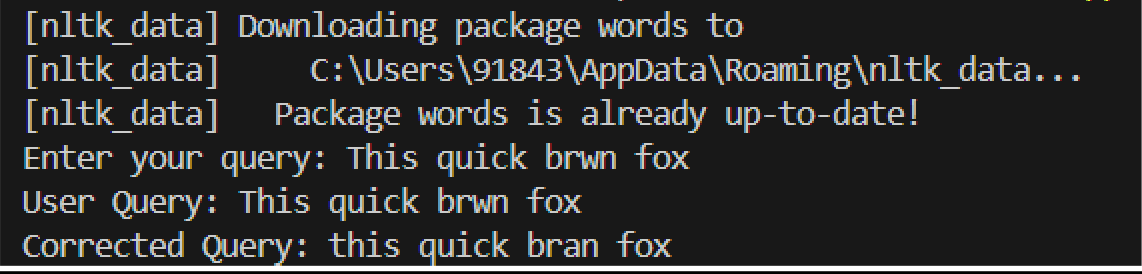
# Example usage

user\_query = input("Enter your query: ") corrected\_query = search(user\_query, nltk\_words)

print(f"User Query: {user\_query}")

print(f"Corrected Query: {' '.join(corrected\_query)}")

**Output:**



**Practical No. 4**

**Title: Evaluation Metrics for IR Systems**

**Aim: To calculate precision, recall, and F-measure**

Design a program to calculate precision, recall, and F1-score based on the user-provided values of true positives (TP), false positives (FP), and false negatives (FN). The program should prompt the user to input the number of true positives (TP), false positives (FP), and false negatives (FN) and Calculate precision, recall, and F1-score using the provided inputs.

**Program**

def calculate\_precision\_recall\_f1(tp, fp, fn): precision = tp / (tp + fp) if (tp + fp) > 0 else 0 recall = tp / (tp + fn) if (tp + fn) > 0 else 0

f1 = (2 \* precision \* recall) / (precision + recall) if (precision + recall) > 0 else 0

return precision, recall, f1

# User input for TP, FP, FN

tp = int(input("Enter the number of True Positives (TP): ")) fp = int(input("Enter the number of False Positives (FP): ")) fn = int(input("Enter the number of False Negatives (FN): "))

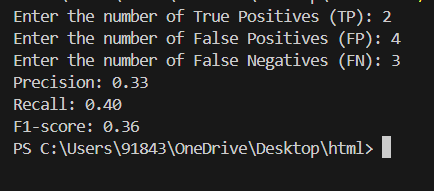
# Calculate precision, recall, and F1-score

precision, recall, f1 = calculate\_precision\_recall\_f1(tp, fp, fn)

# Print the results print(f"Precision: {precision:.2f}") print(f"Recall: {recall:.2f}")

print(f"F1-score: {f1:.2f}")

**Output:**



Calculate the precision, recall, and F-measure metrics for an information retrieval system based on the number of true positives (TP), false positives (FP), and false negatives (FN) obtained from comparing two sets of retrieved documents and relevant documents.

**Program**

def calculate\_metrics(retrieved\_set, relevant\_set): true\_positive = len(retrieved\_set.intersection(relevant\_set)) false\_positive = len(retrieved\_set.difference(relevant\_set)) false\_negative = len(relevant\_set.difference(retrieved\_set))

print("True Positive: ", true\_positive, "\nFalse Positive: ", false\_positive,

"\nFalse Negative: ", false\_negative, "\n")

precision = true\_positive / (true\_positive + false\_positive) if (true\_positive + false\_positive) > 0 else 0

recall = true\_positive / (true\_positive + false\_negative) if (true\_positive + false\_negative) > 0 else 0

f\_measure = 2 \* precision \* recall / (precision + recall) if (precision

+ recall) > 0 else 0

return precision, recall, f\_measure # Create an exhaustive list of terms

all\_terms = ["apple", "banana", "orange", "grape", "melon","papaya"] retrieved=["banana", "orange","papaya"] relevant=["orange","papaya","melom"]

# Example usage:

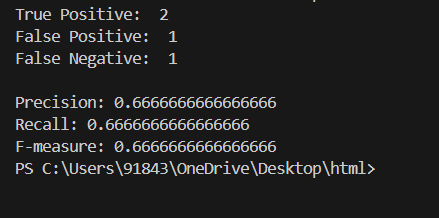
retrieved\_set = set(retrieved) # Predicted set

relevant\_set = set(relevant) # Actually Needed set (Relevant)

precision, recall, f\_measure = calculate\_metrics(retrieved\_set, relevant\_set)

print(f"Precision: {precision}") print(f"Recall: {recall}") print(f"F-measure: {f\_measure}")

**Output:**



Given a set of predicted binary labels (y\_pred) and their corresponding true binary labels (y\_true), evaluate the performance by calculating precision, recall, f1-score and average precision of a binary classification model using standard metrics from scikit-learn module

**Program**

from sklearn.metrics import precision\_recall\_curve, average\_precision\_score, roc\_curve, roc\_auc\_score

# Example data

y\_true = [0, 1, 1, 0, 1]

y\_scores = [0.1, 0.8, 0.6, 0.3, 0.9] # Predicted scores (probability of positive class)

# Calculate precision-recall curve

precision, recall, \_ = precision\_recall\_curve(y\_true, y\_scores)

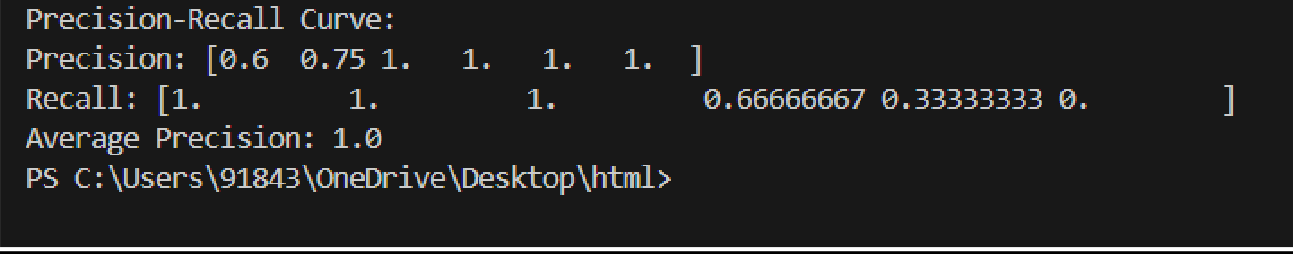
# Calculate average precision

avg\_precision = average\_precision\_score(y\_true, y\_scores)

# Print the results print("Precision-Recall Curve:") print("Precision:", precision) print("Recall:", recall)

print("Average Precision:", avg\_precision)

**Output:**



**Practical No. 5**

**Title: Text Categorization (using Naive Bayes or Support Vector Machines)**

**Aim: To construct a text classification system using the Naive Bayes or SVM classifier.**

Given a dataset of 20\_newsgroups, build a classification model using Naïve Bayes classifier that can accurately classify each review as either positive or negative.

**Program**

from sklearn.feature\_extraction.text import TfidfVectorizer from sklearn.naive\_bayes import MultinomialNB

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

from sklearn.datasets import fetch\_20newsgroups

# Load the 20 newsgroups dataset

newsgroups = fetch\_20newsgroups(subset='all', remove=('headers', 'footers', 'quotes'))

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(newsgroups.data, newsgroups.target, test\_size=0.2, random\_state=42)

# Vectorize the text data using TF-IDF vectorizer = TfidfVectorizer(max\_features=10000) X\_train\_tfidf = vectorizer.fit\_transform(X\_train) X\_test\_tfidf = vectorizer.transform(X\_test)

# Train a Multinomial Naive Bayes classifier classifier = MultinomialNB() classifier.fit(X\_train\_tfidf, y\_train)

# Make predictions on the test set y\_pred = classifier.predict(X\_test\_tfidf)

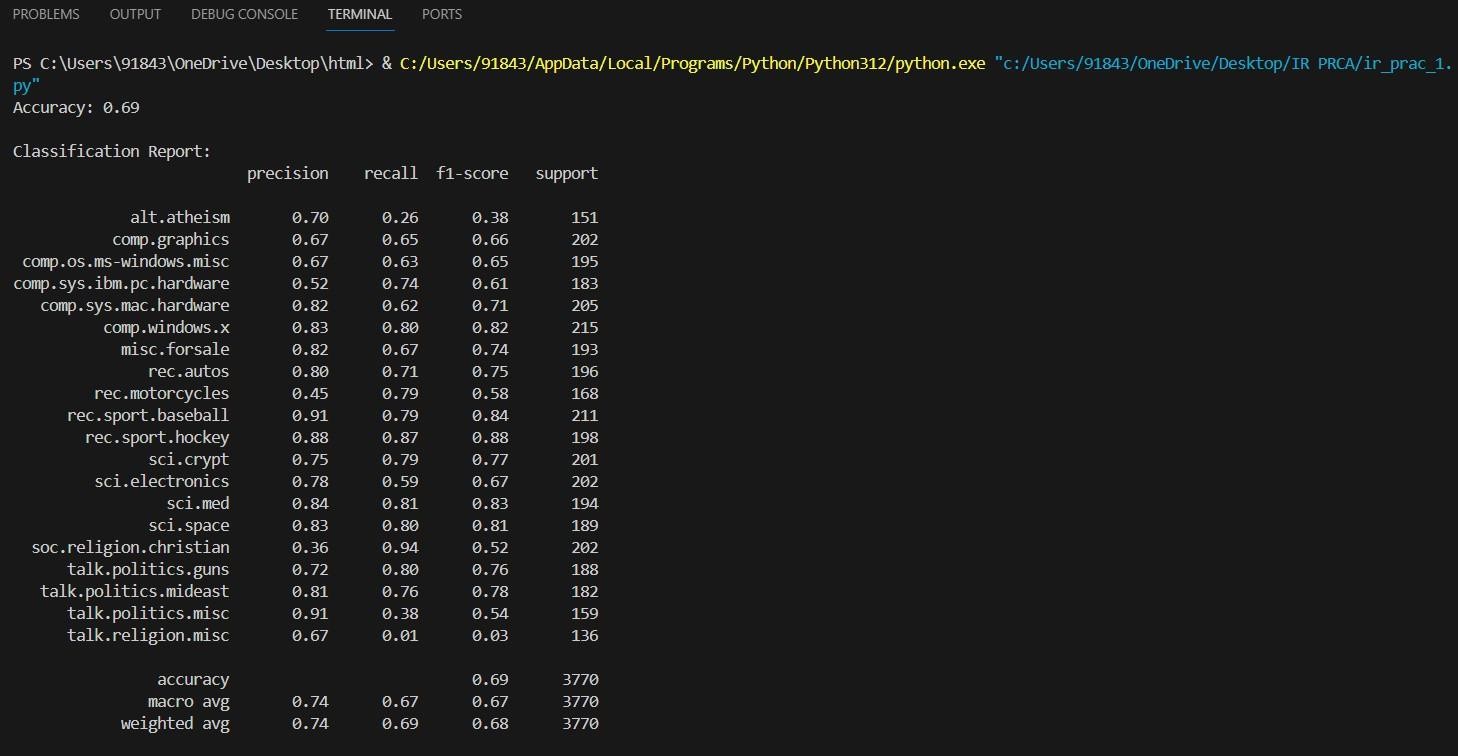
# Calculate accuracy

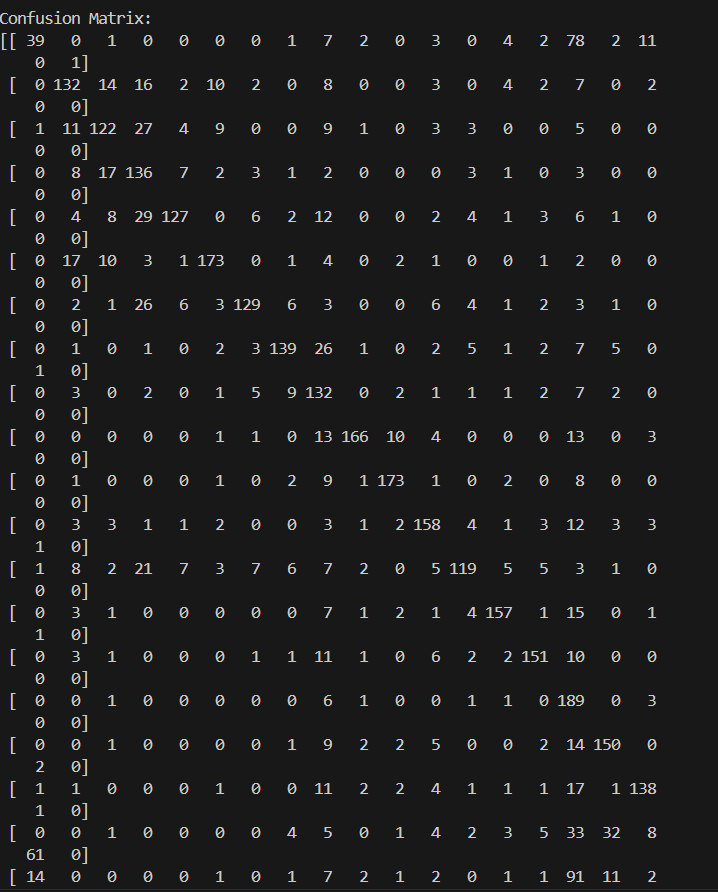
accuracy = accuracy\_score(y\_test, y\_pred) print(f"Accuracy: {accuracy:.2f}")

# Display classification report print("\nClassification Report:") print(classification\_report(y\_test, y\_pred, target\_names=newsgroups.target\_names))

# Display confusion matrix print("\nConfusion Matrix:") print(confusion\_matrix(y\_test, y\_pred))

**Output:**





Given a dataset of 20\_newsgroups, build a classification model using SVM classifier that can accurately classify documents into categories like “alt.atheism,” “soc.religion.christian,” “comp.graphics,” or “sci.med,”

**Program**

from sklearn.datasets import fetch\_20newsgroups

from sklearn.feature\_extraction.text import TfidfVectorizer from sklearn.model\_selection import train\_test\_split

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score, classification\_report

# Load the 20 Newsgroups dataset

categories = ['alt.atheism', 'soc.religion.christian', 'comp.graphics', 'sci.med']

newsgroups\_train = fetch\_20newsgroups(subset='train', categories=categories) newsgroups\_test = fetch\_20newsgroups(subset='test', categories=categories)

# Extract features from text using TF-IDF Vectorizer vectorizer = TfidfVectorizer()

X\_train = vectorizer.fit\_transform(newsgroups\_train.data) X\_test = vectorizer.transform(newsgroups\_test.data) y\_train = newsgroups\_train.target

y\_test = newsgroups\_test.target

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_train, y\_train, test\_size=0.2, random\_state=42)

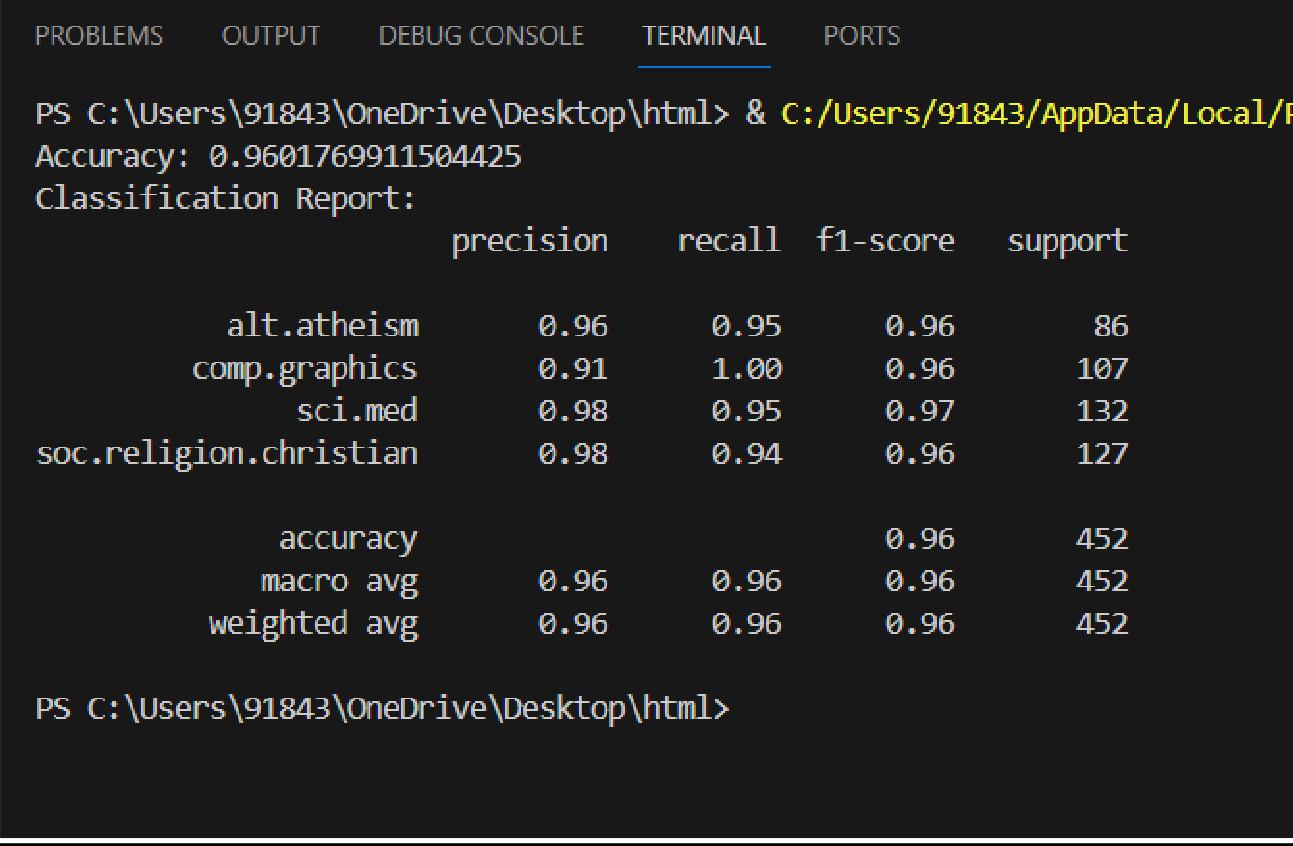
# Train the SVM classifier svm\_classifier = SVC(kernel='linear') svm\_classifier.fit(X\_train, y\_train)

# Make predictions on the testing set predictions = svm\_classifier.predict(X\_test)

# Evaluate the model

accuracy = accuracy\_score(y\_test, predictions) print("Accuracy:", accuracy) print("Classification Report:") print(classification\_report(y\_test, predictions, target\_names=newsgroups\_test.target\_names))

**Output:**



**Practical No. 6**

**Title: Clustering for Information Retrieval (K-means clustering)**

**Aim: To perform text clustering on the 20 Newsgroups dataset using the K-means algorithm and evaluate the quality of clustering results.**

Given a collection of text documents, group similar documents into clusters using the K-means clustering algorithm.

**Program**

from sklearn.feature\_extraction.text import TfidfVectorizer from sklearn.cluster import KMeans

from sklearn.feature\_extraction.text import TfidfVectorizer from sklearn.cluster import KMeans

documents = [

"Cats are known for their agility and grace", # cat doc1 "Dogs are often called ‘man’s best friend’.", # dog doc1

"Some dogs are trained to assist people with disabilities.", # dog doc2

"The sun rises in the east and sets in the west.", # sun doc1 "Many cats enjoy climbing trees and chasing toys.", # cat doc2

"Birds are fascinating creatures that can fly in the sky.", # bird doc1

"Fish live in water and breathe through gills.", # fish doc1 "Reptiles are cold-blooded animals with scales or plates.", # reptile

doc1

]

"Owls are nocturnal birds of prey that hunt at night." # bird doc2

# Create a TfidfVectorizer object

vectorizer = TfidfVectorizer(stop\_words='english')

# Learn vocabulary and idf from training set.

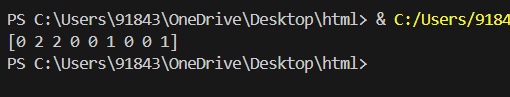
X = vectorizer.fit\_transform(documents)

# Perform k-means clustering

kmeans = KMeans(n\_clusters=3, random\_state=0).fit(X)

# Print cluster labels for each document print(kmeans.labels\_)

**Output:**



Given a collection of text documents from the 20 Newsgroups dataset, group similar documents into clusters using the K-means clustering algorithm.

**Program**

from sklearn.feature\_extraction.text import TfidfVectorizer from sklearn.cluster import KMeans

from sklearn import metrics

from sklearn.datasets import fetch\_20newsgroups

# Step 1: Load and Preprocess the Data

dataset = fetch\_20newsgroups(subset='all', shuffle=True, random\_state=42) documents = dataset.data

# Step 2: Feature Extraction

vectorizer = TfidfVectorizer(stop\_words='english')

X = vectorizer.fit\_transform(documents)

# Step 3: Apply K-means Clustering num\_clusters = 20 # Number of clusters

kmeans = KMeans(n\_clusters=num\_clusters, random\_state=42) kmeans.fit(X)

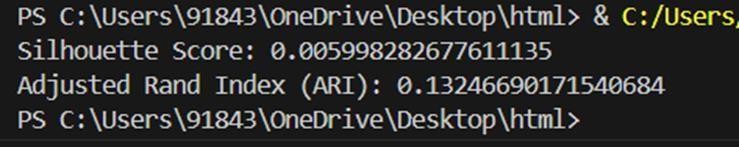
# Step 4: Evaluate Clustering Results

silhouette\_score = metrics.silhouette\_score(X, kmeans.labels\_, metric='euclidean')

ari\_score = metrics.adjusted\_rand\_score(dataset.target, kmeans.labels\_)

print("Silhouette Score:", silhouette\_score) print("Adjusted Rand Index (ARI):", ari\_score)

**Output:**



Given a collection of text documents, group similar documents into clusters using the hierarchical clustering algorithm.

**Program**

from sklearn.feature\_extraction.text import TfidfVectorizer from sklearn.cluster import AgglomerativeClustering

documents = [

"Cats are known for their agility and grace", # cat doc1 "Dogs are often called ‘man’s best friend’.", # dog doc1

"Some dogs are trained to assist people with disabilities.", # dog doc2

"The sun rises in the east and sets in the west.", # sun doc1 "Many cats enjoy climbing trees and chasing toys.", # cat doc2

"Birds are fascinating creatures that can fly in the sky.", # bird doc1

"Fish live in water and breathe through gills.", # fish doc1 "Reptiles are cold-blooded animals with scales or plates.", # reptile

doc1

]

"Owls are nocturnal birds of prey that hunt at night." # bird doc2

# Create a TfidfVectorizer object

vectorizer = TfidfVectorizer(stop\_words='english')

# Learn vocabulary and idf from training set.

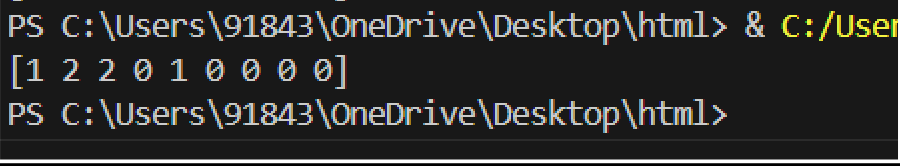
X = vectorizer.fit\_transform(documents)

# Perform hierarchical clustering

agg\_clustering = AgglomerativeClustering(n\_clusters=3).fit(X.toarray())

# Print cluster labels for each document print(agg\_clustering.labels\_)

**Output:**



Given a collection of text documents from the 20 Newsgroups dataset, group similar documents into clusters using the hierarchial clustering algorithm.

**Program**

from sklearn.feature\_extraction.text import TfidfVectorizer from sklearn.cluster import AgglomerativeClustering

from sklearn import metrics

from sklearn.datasets import fetch\_20newsgroups

# Step 1: Load and Preprocess the Data

dataset = fetch\_20newsgroups(subset='all', shuffle=True, random\_state=42) documents = dataset.data

# Step 2: Feature Extraction

vectorizer = TfidfVectorizer(stop\_words='english')

X = vectorizer.fit\_transform(documents)

# Step 3: Apply Hierarchical Clustering num\_clusters = 20 # Number of clusters

hierarchical\_clustering = AgglomerativeClustering(n\_clusters=num\_clusters) hierarchical\_clustering.fit(X.toarray())

# Step 4: Evaluate Clustering Results silhouette\_score = metrics.silhouette\_score(X, hierarchical\_clustering.labels\_, metric='euclidean')

ari\_score = metrics.adjusted\_rand\_score(dataset.target, hierarchical\_clustering.labels\_)

print("Silhouette Score:", silhouette\_score) print("Adjusted Rand Index (ARI):", ari\_score)

**Output:**

**Practical No. 7**

**Title: Web Crawling and Indexing**

**Aim: Develop a web crawler to fetch and index web pages.**

**Program**

import requests

from bs4 import BeautifulSoup from urllib.parse import urljoin

from urllib.robotparser import RobotFileParser import time

def get\_html(url): try:

headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.3'}

response = requests.get(url, headers=headers) response.raise\_for\_status()

return response.text

except requests.exceptions.RequestException as err: print(f"Request Error: {err}")

return None

def load\_robots\_txt(url): try:

robots\_url = urljoin(url, '/robots.txt') robots\_content = get\_html(robots\_url) robot\_parser = RobotFileParser()

if robots\_content: robot\_parser.parse(robots\_content.split('\n'))

return robot\_parser except Exception as e:

print(f"Error loading robots.txt: {e}") return None

def extract\_links(html, base\_url):

soup = BeautifulSoup(html, 'html.parser')

links = [urljoin(base\_url, link['href']) for link in soup.find\_all('a', href=True)]

return links

def is\_allowed\_by\_robots(robot\_parser, url): return robot\_parser.can\_fetch('\*', url)

def crawl(start\_url, max\_depth=3, delay=1): visited\_urls = set()

robot\_parser = load\_robots\_txt(start\_url)

def recursive\_crawl(url, depth):

if depth > max\_depth or url in visited\_urls or not is\_allowed\_by\_robots(robot\_parser, url):

return visited\_urls.add(url) time.sleep(delay) html = get\_html(url) if html:

print(f"Crawling {url}")

links = extract\_links(html, url) for link in links:

recursive\_crawl(link, depth + 1)

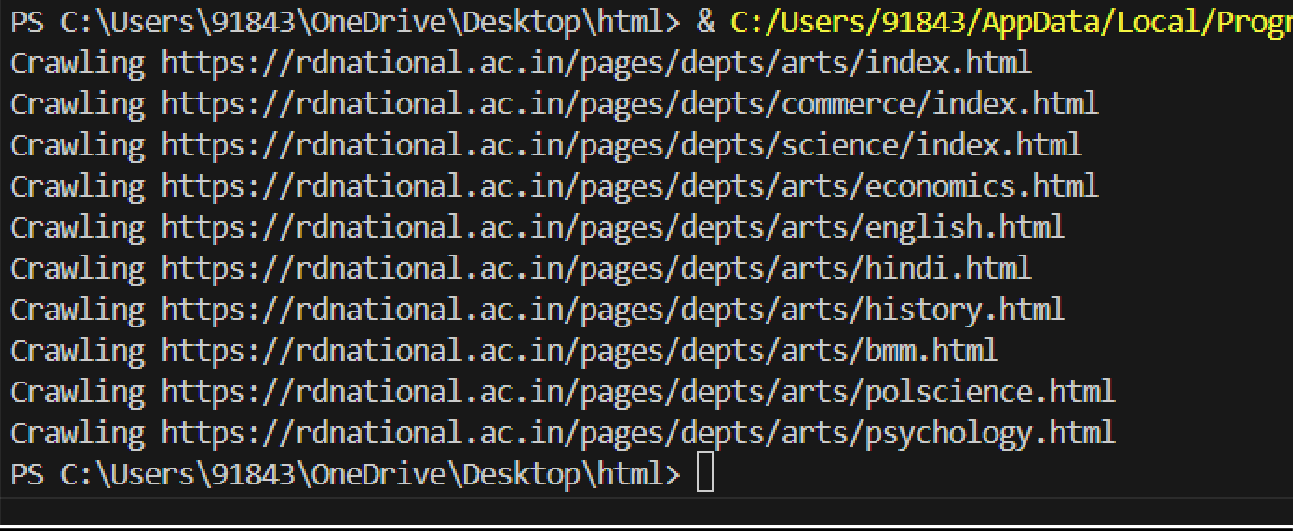
if not robot\_parser or not is\_allowed\_by\_robots(robot\_parser, start\_url):

print(f"Access to {start\_url} is restricted by robots.txt.

Crawling aborted.")

return recursive\_crawl(start\_url, 1)

**Output:**



**Practical No. 8**

**Title: Link Analysis and PageRank (implementation of PageRank algorithm) Aim: Implement the PageRank algorithm to rank web pages based on link analysis.**

**Program**

import numpy as np

def page\_rank(graph, damping\_factor=0.85, max\_iterations=100, tolerance=1e- 6):

"""

Compute PageRank scores for each node in the graph represented by an adjacency list.

Parameters:

* graph: list of lists, adjacency list representing the graph
* damping\_factor: float, damping factor (default: 0.85)
* max\_iterations: int, maximum number of iterations (default: 100)
* tolerance: float, tolerance for convergence (default: 1e-6)

Returns:

* pagerank\_scores: numpy array, PageRank scores for each node """

num\_nodes = len(graph)

page\_ranks = np.ones(num\_nodes) / num\_nodes

for \_ in range(max\_iterations): prev\_page\_ranks = np.copy(page\_ranks)

for node in range(num\_nodes):

incoming\_links = [i for i, v in enumerate(graph) if node in v] if not incoming\_links:

continue

page\_ranks[node] = (1 - damping\_factor) / num\_nodes + \

damping\_factor \* sum(prev\_page\_ranks[link] / len(graph[link]) for link in incoming\_links)

if np.linalg.norm(page\_ranks - prev\_page\_ranks, 2) < tolerance: break

return page\_ranks # Example usage

if name == " main ":

# Define a simple directed graph as an adjacency list web\_graph = [

[1, 2], # Node 0 has links to Node 1 and Node 2

[0, 2], # Node 1 has links to Node 0 and Node 2

[0, 1], # Node 2 has links to Node 0 and Node 1

[1, 2], # Node 3 has links to Node 1 and Node 2

]

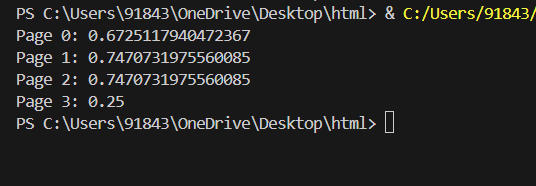
# Calculate PageRank

result = page\_rank(web\_graph)

# Display PageRank values

for i, pr in enumerate(result): print(f"Page {i}: {pr}")

**Output:**



**Practical 10**

**Title:** Advanced Topics in Information Retrieval

**Aim:** Implement a text summarization algorithm (e.g., extractive or abstractive) and Build a question- answering system using techniques such as information extraction

**Program:**

from sklearn.feature\_extraction.text import TfidfVectorizer from sklearn.metrics.pairwise import cosine\_similarity from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize import string

# Sample corpus of text data (answers) corpus = [

"Python is an interpreted, high-level, general-purpose programming language.",

"Machine learning is the scientific study of algorithms and statistical models that computer systems use to perform a specific task without using explicit instructions.",

"Natural language processing (NLP) is a subfield of linguistics, computer science, and artificial intelligence concerned with the interactions between computers and human language.",

"Deep learning is a subset of machine learning in artificial intelligence (AI) that has networks capable of learning unsupervised from data that is unstructured or unlabeled.",

"Reinforcement learning is an area of machine learning concerned with how software agents ought to take actions in an environment in order to maximize the notion of cumulative reward."

]

# Preprocess the corpus def preprocess(text):

stop\_words = set(stopwords.words('english')) text = text.lower() # Convert to lowercase

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

Remove punctuation

words = word\_tokenize(text) # Tokenize words

words = [word for word in words if word not in stop\_words] # Remove stopwords

return ' '.join(words)

processed\_corpus = [preprocess(text) for text in corpus] # TF-IDF Vectorization

vectorizer = TfidfVectorizer()

tfidf\_matrix = vectorizer.fit\_transform(processed\_corpus)

# User question

question = input("Enter your question: ")

# Preprocess the question processed\_question = preprocess(question)

# Calculate cosine similarity between question and answers question\_vector = vectorizer.transform([processed\_question])

similarity\_scores = cosine\_similarity(question\_vector, tfidf\_matrix)

# Retrieve the most similar answer most\_similar\_index = similarity\_scores.argmax() answer = corpus[most\_similar\_index]

print("Question:", question) print("Answer:", answer)

**output:**

