**Practical-1**

**AIM :- Document Indexing and Retrieval**

1. Implement an inverted index construction algorithm.
2. Build a simple document retrieval system using the constructed index.

**Source Code :-**

document1="The quick brown fox jumped over the lazy dog."

document2="The lazy dog slept in the sun."

tokens1=document1.lower().split()

tokens2=document2.lower().split()

terms=list(set(tokens1+tokens2))

inverted\_index={}

**for** term **in** terms:

documents = []

**if** term **in** tokens1:

documents.append("Document 1")

**if** term **in** tokens2:

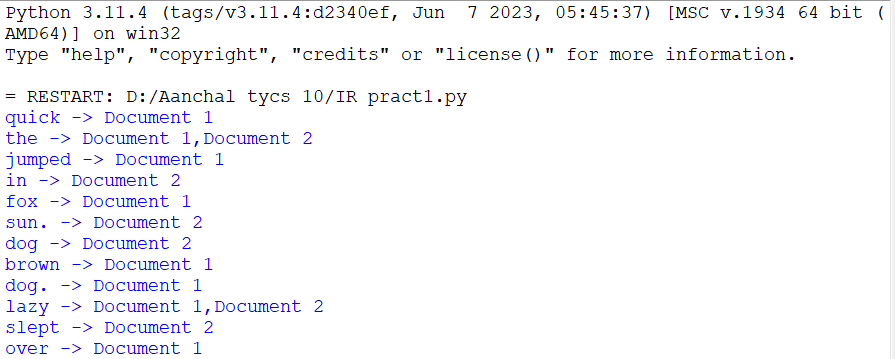
documents.append("Document 2")

inverted\_index[term]=documents

**for** term,documents **in** inverted\_index.items():

**print** (term ,"->",",".join(documents))

**Output :-**



**Practical-2**

**AIM :-** **Retrieval Models**

1. Implement the Boolean retrieval model and process queries.
2. Implement the vector space model with TF-IDF weighting and cosine similarity.

**A)**

**Source Code :-**

documents={

1:"apple banana orange",

2:"apple banana",

3:"banana orange",

4:"apple"

}

**def** build\_index(docs):

index={}

**for** doc\_id, text **in** docs.items():

terms=set(text.split())

**for** term **in** terms:

**if** term **not** **in** index:

index[term]={doc\_id}

**else**:

index[term].add(doc\_id)

**return** index

inverted\_index=build\_index(documents)

**def** boolean\_and(operands,index):

**if** **not** operands:

**return** list(range(1,len(documents) +1))

result=index.get(operands[0],set())

**for** term **in** operands[1:]:

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

**return** list(result)

**def** boolean\_or(operands,index):

result=set()

**for** term **in** operands:

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

**return** list(result)

**def** boolean\_not(operand,index,total\_docs):

operand\_set=set(index.get(operand,set()))

all\_docs\_set=set(range(1,total\_docs+1))

**return** list(all\_docs\_set.difference(operand\_set))

query1=["apple","banana"]

query2=["apple","orange"]

result1=boolean\_and(query1,inverted\_index)

result2=boolean\_or(query2,inverted\_index)

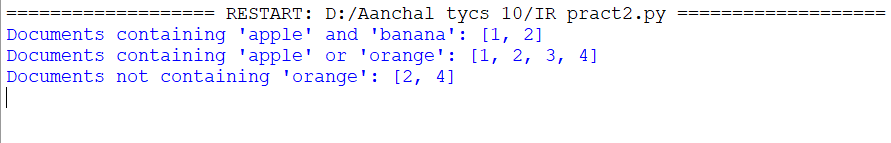
result3=boolean\_not("orange",inverted\_index , len(documents))

**print**("Documents containing 'apple' and 'banana':",result1)

**print**("Documents containing 'apple' or 'orange':",result2)

**print**("Documents not containing 'orange':",result3)

**Output :-**



**B)**

**Source Code :-**

**import** math

**from** collections **import** Counter

**def** tokenize(text):

# Simple tokenization, you might want to improve it based on your specific needs

**return** text.lower().split()

**def** compute\_tf(doc):

# Computes term frequency (TF) for each term in the document

word\_count = Counter(tokenize(doc))

total\_words = len(tokenize(doc))

tf = {word: count/total\_words **for** word, count **in** word\_count.items()}

**return** tf

**def** compute\_idf(documents):

# Computes inverse document frequency (IDF) for each term in the collection of documents

N = len(documents)

idf = {}

**for** doc **in** documents:

terms = set(tokenize(doc))

**for** term **in** terms:

idf[term] = idf.get(term, 0) + 1

idf = {term: math.log(N/frequency) **for** term, frequency **in** idf.items()}

**return** idf

**def** compute\_tf\_idf(tf, idf):

# Computes TF-IDF for each term in the document

tf\_idf = {term: tf[term] \* idf.get(term, 0) **for** term **in** tf}

**return** tf\_idf

**def** compute\_cosine\_similarity(query\_tf\_idf, doc\_tf\_idf):

# Computes cosine similarity between query and document TF-IDF vectors

common\_terms = set(query\_tf\_idf.keys()) & set(doc\_tf\_idf.keys())

dot\_product = sum(query\_tf\_idf[term] \* doc\_tf\_idf[term] **for** term **in** common\_terms)

query\_norm = math.sqrt(sum(value\*\*2 **for** value **in** query\_tf\_idf.values()))

doc\_norm = math.sqrt(sum(value\*\*2 **for** value **in** doc\_tf\_idf.values()))

**if** query\_norm == 0 **or** doc\_norm == 0:

**return** 0

similarity = dot\_product / (query\_norm \* doc\_norm)

**return** similarity

similarity = dot\_product / (query\_norm \* doc\_norm)

**return** similarity

**def** vector\_space\_model(documents, query):

# Build TF-IDF vectors for documents and query

tf\_idf\_vectors = []

**for** doc **in** documents:

tf = compute\_tf(doc)

idf = compute\_idf(documents)

tf\_idf = compute\_tf\_idf(tf, idf)

tf\_idf\_vectors.append(tf\_idf)

query\_tf = compute\_tf(query)

query\_idf = compute\_idf(documents)

query\_tf\_idf = compute\_tf\_idf(query\_tf, query\_idf)

# Compute cosine similarity between query and each document

similarities = []

**for** doc\_tf\_idf **in** tf\_idf\_vectors:

similarity = compute\_cosine\_similarity(query\_tf\_idf, doc\_tf\_idf)

similarities.append(similarity)

**return** similarities

# Example usage:

documents = [

"This is the first document.",

"This document is the document.",

"And this is the third one.",

"Is this the document?",

]

query = "This is a query document."

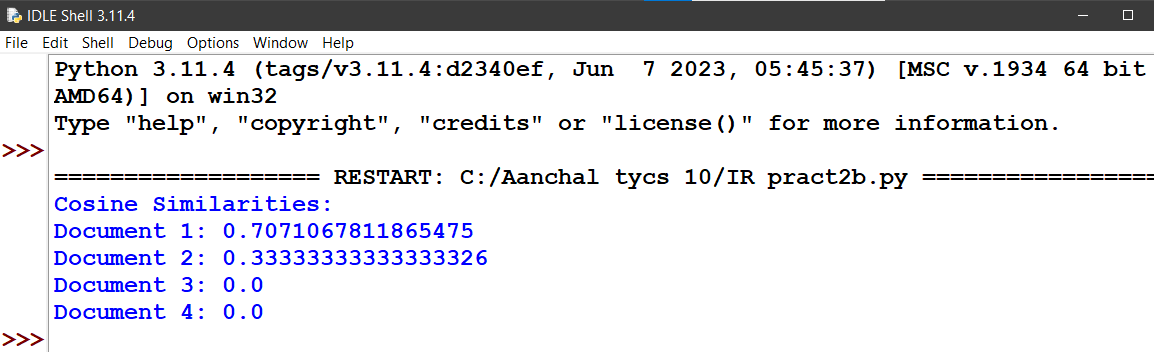
similarities = vector\_space\_model(documents, query)

**print**("Cosine Similarities:")

**for** i, sim **in** enumerate(similarities):

**print**(f"Document {i + 1}: {sim}")

**Output :-**



**Practical-3**

**AIM :-** **Spelling Correction in IR Systems**

1. Develop a spelling correction module using edit distance algorithms.
2. Integrate the spelling correction module into an information retrieval system.

**Source Code :-**

**def** edit\_distance(str1, str2):

m, n = len(str1), len(str2)

dp = [[0] \* (n + 1) **for** \_ **in** range(m + 1)]

**for** i **in** range(m + 1):

**for** j **in** range(n + 1):

**if** i == 0:

dp[i][j] = j

**elif** j == 0:

dp[i][j] = i

**elif** str1[i-1] == str2[j-1]:

dp[i][j] = dp[i-1][j-1]

**else**:

dp[i][j] = 1 + min(dp[i][j-1], # Insert

dp[i-1][j], # Remove

dp[i-1][j-1] # Replace

)

**return** dp[m][n]

**def** correct\_spelling(query, dictionary):

corrected\_query = query.lower() # Convert to lowercase for case-insensitive correction

suggestions = []

**for** word **in** dictionary:

distance = edit\_distance(corrected\_query, word.lower())

**if** distance <= 10: # You can adjust this threshold based on your preference

suggestions.append((word, distance))

suggestions.sort(key=**lambda** x: x[1]) # Sort by distance in ascending order

**return** suggestions

# Example usage

dictionary = ["information", "retrieval", "spelling", "correction", "system"]

query = input("Enter your query: ")

suggestions = correct\_spelling(query, dictionary)

**if** suggestions:

**print**("Did you mean:")

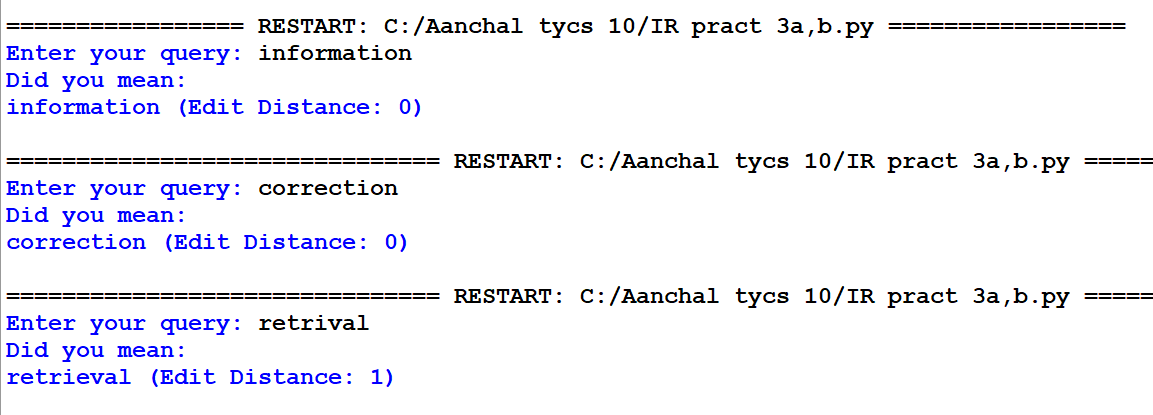
**for** suggestion, distance **in** suggestions:

**print**(f"{suggestion} (Edit Distance: {distance})")

**else**:

**print**("No suggestions found.")

**Output :-**



**Practical-4**

**AIM :- Evaluation Metrics for IR Systems**

1. Calculate precision, recall, and F-measure for a given set of retrieval results.
2. Use an evaluation toolkit to measure average precision and other evaluation metrics.

**A)**

**Source Code :-**

**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)

recall = true\_positive / (true\_positive + false\_negative)

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

**return** precision, recall, f\_measure

retrieved\_set = set(["doc1", "doc2", "doc3"]) # Predicted set

relevant\_set = set(["doc1", "doc4"]) # Actually Needed set (Relevant)

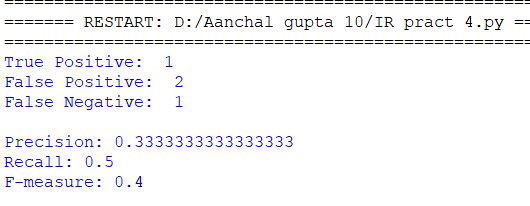
precision, recall, f\_measure = calculate\_metrics(retrieved\_set, relevant\_set)

**print**(f"Precision: {precision}")

**print**(f"Recall: {recall}")

**print**(f"F-measure: {f\_measure}")

**Output :-**

****

**B)**

**Source Code :-**

**from** sklearn.metrics **import** average\_precision\_score, precision\_recall\_curve, auc

**from** sklearn.metrics **import** accuracy\_score, precision\_score, recall\_score, f1\_score, roc\_auc\_score

**import** matplotlib.pyplot **as** plt

**import** numpy **as** np

**def** calculate\_classification\_metrics(y\_true, y\_pred\_prob):

"""

Calculate various classification metrics.

Parameters:

- y\_true: True labels (ground truth).

- y\_pred\_prob: Predicted probabilities for the positive class.

Returns:

- Dictionary containing different evaluation metrics.

"""

# Convert probabilities to binary predictions

y\_pred\_binary = (np.array(y\_pred\_prob) >= 0.5).astype(int)

# Accuracy

accuracy = accuracy\_score(y\_true, y\_pred\_binary)

# Precision, Recall, F1 Score

precision = precision\_score(y\_true, y\_pred\_binary)

recall = recall\_score(y\_true, y\_pred\_binary)

f1 = f1\_score(y\_true, y\_pred\_binary)

# ROC-AUC

roc\_auc = roc\_auc\_score(y\_true, y\_pred\_prob)

# Average Precision and Precision-Recall curve

average\_precision = average\_precision\_score(y\_true, y\_pred\_prob)

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

area\_under\_pr\_curve = auc(recall\_curve, precision\_curve)

# Print metrics

**print**(f'Accuracy: {accuracy}')

**print**(f'Precision: {precision}')

**print**(f'Recall: {recall}')

**print**(f'F1 Score: {f1}')

**print**(f'ROC-AUC: {roc\_auc}')

**print**(f'Average Precision: {average\_precision}')

**print**(f'Area under Precision-Recall curve: {area\_under\_pr\_curve}')

# Plot Precision-Recall curve

plt.figure(figsize=(8, 6))

plt.plot(recall\_curve, precision\_curve, color='blue', lw=2, label='Precision-Recall curve')

plt.xlabel('Recall')

plt.ylabel('Precision')

plt.title('Precision-Recall Curve')

plt.legend()

plt.show()

# Example usage:

# Replace the following placeholders with your actual data

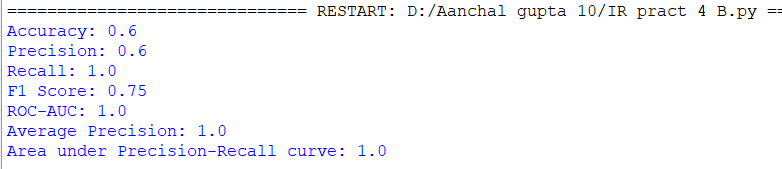
# Example for binary classification

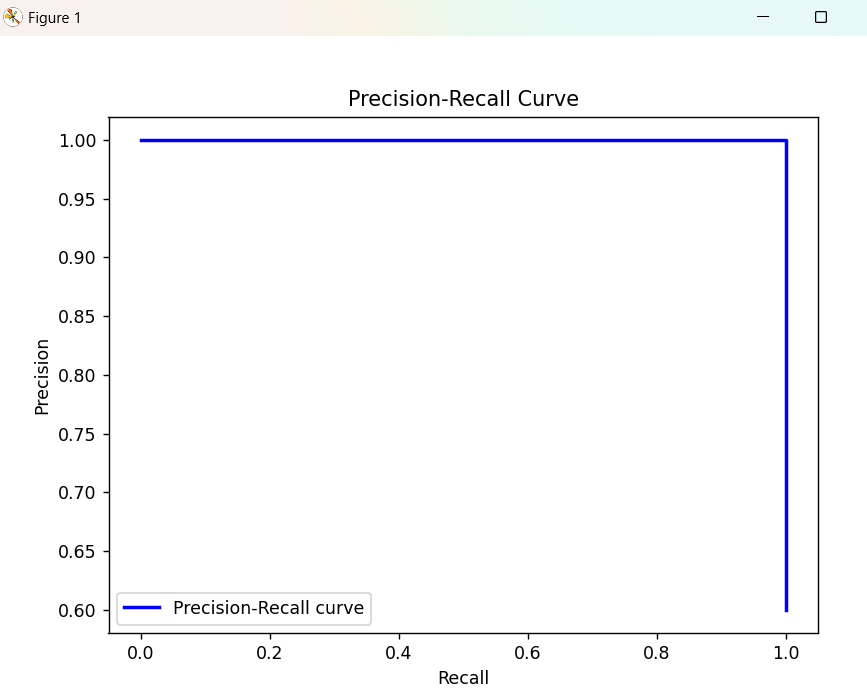
y\_true\_binary = [1, 0, 1, 0, 1] # True binary labels (1 for positive, 0 for negative)

y\_pred\_prob\_binary = [0.8, 0.6, 0.9, 0.5, 0.7] # Predicted probabilities for the positive class

calculate\_classification\_metrics(y\_true\_binary, y\_pred\_prob\_binary)

**Output :-**

****



**Practical-5**

**AIM :- Text Categorization**

1. Implement a text classification algorithm (e.g., Naive Bayes or Support Vector Machines).
2. Train the classifier on a labelled dataset and evaluate its performance.

**Source Code :-**

# Import necessary libraries

**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

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)

# Convert text data to TF-IDF features

vectorizer = TfidfVectorizer()

X\_train\_tfidf = vectorizer.fit\_transform(X\_train)

X\_test\_tfidf = vectorizer.transform(X\_test)

# Train a Support Vector Machine (SVM) classifier

svm\_classifier = SVC(kernel='linear', C=1.0)

svm\_classifier.fit(X\_train\_tfidf, y\_train)

# Make predictions on the test set

y\_pred = svm\_classifier.predict(X\_test\_tfidf)

# Evaluate the performance of the classifier

accuracy = accuracy\_score(y\_test, y\_pred)

classification\_rep = classification\_report(y\_test, y\_pred, target\_names=newsgroups.target\_names)

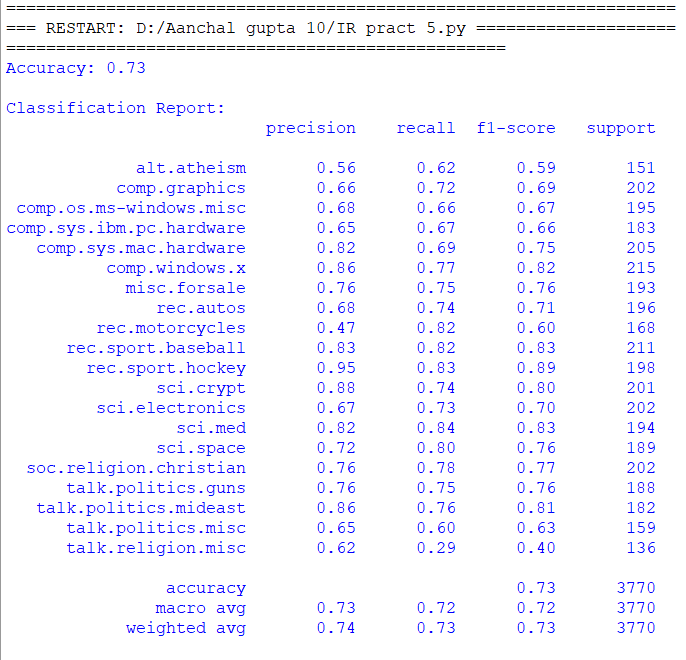
# Print the results

**print**(f"Accuracy: {accuracy:.2f}")

**print**("\nClassification Report:")

**print**(classification\_rep)

**Output :-**



**Practical-6**

**AIM :- Clustering for Information Retrieval**

1. Implement a clustering algorithm (e.g., K-means or hierarchical clustering).
2. Apply the clustering algorithm to a set of documents and evaluate the clustering results.

**Source Code :-**

# Import necessary libraries

**from** sklearn.feature\_extraction.text **import** TfidfVectorizer

**from** sklearn.cluster **import** KMeans

**from** sklearn.metrics **import** silhouette\_score

# Sample documents

documents = [

"This is the first document.",

"This document is the second document.",

"And this is the third one.",

"Is this the first document?",

"The last document is here."

]

# Convert documents to TF-IDF matrix

vectorizer = TfidfVectorizer()

X = vectorizer.fit\_transform(documents)

# Apply K-means clustering

num\_clusters = 2 # You can adjust the number of clusters as needed

n\_init\_value = 10 # You can adjust this value based on your needs

kmeans = KMeans(n\_clusters=num\_clusters, n\_init='auto')

kmeans.fit(X)

# Get cluster labels

cluster\_labels = kmeans.labels\_

# Calculate silhouette score using the original documents and cluster labels

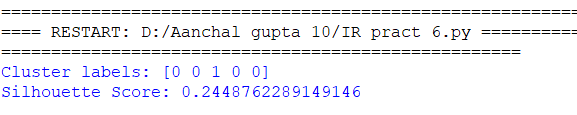
silhouette\_avg = silhouette\_score(X, cluster\_labels)

# Print results

**print**("Cluster labels:", cluster\_labels)

**print**("Silhouette Score:", silhouette\_avg)

**Output :-**



**Practical-7**

**AIM :- Web Crawling and Indexing**

1. Develop a web crawler to fetch and index web pages.
2. Handle challenges such as robots.txt, dynamic content, and crawling delays.

**Source Code :-**

**import** requests

**from** bs4 **import** BeautifulSoup

**import** time

**from** urllib.parse **import** urlparse, urljoin

**from** urllib.robotparser **import** RobotFileParser

**class** WebCrawler:

**def** \_\_init\_\_(self, seed\_url, max\_pages=100):

self.seed\_url = seed\_url

self.base\_url = urlparse(seed\_url).scheme + '://' + urlparse(seed\_url).hostname

self.visited\_urls = set()

self.queue = [seed\_url]

self.max\_pages = max\_pages

self.pages\_crawled = 0

self.robot\_parser = RobotFileParser()

self.robot\_parser.set\_url(urljoin(self.base\_url, '/robots.txt'))

self.robot\_parser.read()

**def** is\_allowed\_by\_robots(self, url):

**return** self.robot\_parser.can\_fetch('\*', url)

**def** crawl(self):

**while** self.queue **and** self.pages\_crawled < self.max\_pages:

current\_url = self.queue.pop(0)

**if** current\_url **not** **in** self.visited\_urls **and** self.is\_allowed\_by\_robots(current\_url):

**try**:

response = requests.get(current\_url)

**if** response.status\_code == 200:

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

# Here you can implement your indexing logic

# For example, extract text, store in a database, etc.

**print**(f"Crawled: {current\_url}")

# Extract links from the page and add them to the queue

**for** link **in** soup.find\_all('a', href=True):

new\_url = urljoin(current\_url, link['href'])

**if** new\_url.startswith(self.base\_url) **and** new\_url **not** **in** self.visited\_urls:

self.queue.append(new\_url)

self.visited\_urls.add(current\_url)

self.pages\_crawled += 1

**except** Exception **as** e:

**print**(f"Error crawling {current\_url}: {str(e)}")

# Introduce a delay to avoid being too aggressive

time.sleep(1)

**if** \_\_name\_\_ == "\_\_main\_\_":

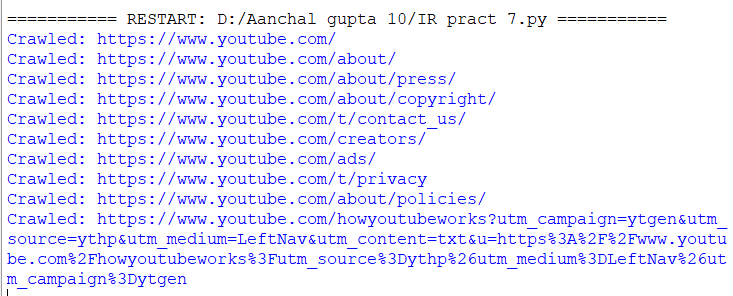
seed\_url = "https://www.youtube.com/"

max\_pages\_to\_crawl = 10

crawler = WebCrawler(seed\_url, max\_pages\_to\_crawl)

crawler.crawl()

**Output :-**



**Practical-8**

**AIM :- Link Analysis and PageRank**

1. Implement the PageRank algorithm to rank web pages based on link analysis.
2. Apply the PageRank algorithm to a small web graph and analyze the results.

**Source Code :-**

**import** numpy **as** np

**def** pagerank(graph, damping\_factor=0.85, max\_iterations=100, epsilon=1e-8):

num\_pages = len(graph)

initial\_page\_rank = np.ones(num\_pages) / num\_pages

teleportation\_prob = (1 - damping\_factor) / num\_pages

page\_rank = initial\_page\_rank.copy()

**for** \_ **in** range(max\_iterations):

prev\_page\_rank = page\_rank.copy()

**for** i **in** range(num\_pages):

incoming\_links = np.where(graph[:, i] == 1)[0]

sum\_page\_rank = np.sum(prev\_page\_rank[incoming\_links] / len(incoming\_links)) **if** incoming\_links.size > 0 **else** 0

page\_rank[i] = teleportation\_prob + damping\_factor \* sum\_page\_rank

# Check for convergence

**if** np.linalg.norm(page\_rank - prev\_page\_rank, 1) < epsilon:

**break**

**return** page\_rank

**if** \_\_name\_\_ == "\_\_main\_\_":

# Example web graph (adjacency matrix)

web\_graph = np.array([

[0, 1, 1, 1],

[1, 0, 0, 0],

[1, 1, 0, 1],

[0, 0, 1, 0]

])

# Calculate PageRank

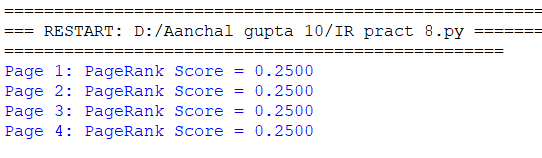
page\_rank\_scores = pagerank(web\_graph)

# Display results

**for** i, score **in** enumerate(page\_rank\_scores):

**print**(f"Page {i + 1}: PageRank Score = {score:.4f}")

**Output :-**

****

**Practical-9**

**AIM :- Learning to Rank**

1. Implement a learning to rank algorithm (e.g., RankSVM or RankBoost).
2. Train the ranking model using labelled data and evaluate its effectiveness.

**Source Code :-**

**from** sklearn.datasets **import** make\_classification

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

**from** sklearn.preprocessing **import** StandardScaler

**from** sklearn.svm **import** SVC

**from** sklearn.metrics **import** ndcg\_score

# Generate synthetic labeled data (replace with your actual labeled data)

X, y = make\_classification(n\_samples=1000, n\_features=10, n\_classes=2, random\_state=42)

# Split data into train and test sets

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

# Normalize features

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# Train RankSVM model

ranksvm\_model = SVC(kernel='linear')

ranksvm\_model.fit(X\_train\_scaled, y\_train)

# Predict rankings on test set

y\_pred = ranksvm\_model.decision\_function(X\_test\_scaled)

# Evaluate model effectiveness using NDCG (Normalized Discounted Cumulative Gain)

ndcg = ndcg\_score(y\_test.reshape(1, -1), y\_pred.reshape(1, -1))

**print**("NDCG Score:", ndcg)

**Output :-**

