**Practical No: 1**

# Aim: Document Indexing and Retrieval

# • Implement an inverted index construction algorithm.

# • Build a simple document retrieval system using the constructed index.

**Theory:**

● An 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 organised by terms (words), and each term points to a list of documents or web pages that contain that term.

● Inverted indexes are widely used in search engines, database systems, and other applications where efficient text search is required.

● They are especially useful for large collections of documents, where searching through all the documents would be prohibitively slow. An inverted index is an index data structure storing a mapping from content, such as words or numbers, to its locations in a document or a set of documents.

**Rules to create an inverted index** –

1) The text of each document is first preprocessed by removing stop words: Stop words are the most occurring and useless words in documents like “I”, “the”, “we”, “is”, and “an”.

2) The text is tokenized, meaning that it is split into individual terms.

3) The terms are then added to the index, with each term pointing to the documents in which it appears.

**Input:**

import nltk  # Import NLTK to download stopwords

from nltk.corpus import stopwords # Import stopwords from NLTK

# Define the documents

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

document2 = "The lazy dog slept in the sun"

# Get the stopwords for English language from NLTK

nltk.download('stopwords')

stopWords = stopwords.words('english')

# Step 1: Tokenize the documents

# Convert each document to lowercase and split it into words

tokens1 = document1.lower().split()

tokens2 = document2.lower().split()

# Combine the tokens into a list of unique terms

terms = list(set(tokens1 + tokens2))

# Step 2: Build the inverted index

# Create an empty dictionary to store the inverted index as well as a dictionary

#to store number of occurrences

inverted\_index = {}

occ\_num\_doc1 = {}

occ\_num\_doc2 = {}

# For each term, find the documents that contain it

for term in terms:

    if term in stopWords:

        continue

    documents = []

    if term in tokens1:

        documents.append("Document 1")

        occ\_num\_doc1[term] = tokens1.count(term)

    if term in tokens2:

        documents.append("Document 2")

        occ\_num\_doc2[term] = tokens2.count(term)

    inverted\_index[term] = documents

# Step 3: Print the inverted index

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

    print(term, "->", end=" ")

for doc in documents:

    if doc == "Document 1":

        print(f"{doc} ({occ\_num\_doc1.get(term, 0)}),", end=" ")

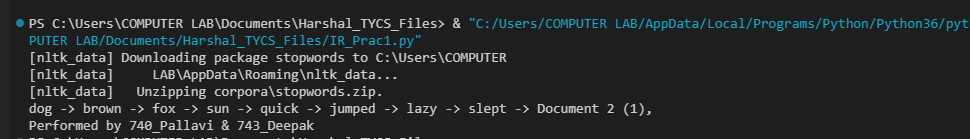
    else:

        print(f"{doc} ({occ\_num\_doc2.get(term, 0)}),", end=" ")

    print()

print("Performed by 740\_Pallavi & 743\_Deepak")

**Output:**

****

**Practical No: 2**

# Aim: Retrieval Models

# ● Implement the Boolean retrieval model and process queries.

# ● Implement the vector space model with TF-IDF weighting and cosine similarity.

**Implement the Boolean retrieval model and process queries:**

documents = {

    1: "apple banana orange",

    2: "apple banana",

    3: "banana orange",

    4: "apple"

}

# Function to build an inverted index using dictionaries

def build\_index(docs):

    index = {}  # Initialize an empty dictionary to store the inverted index

    for doc\_id, text in docs.items():  # Iterate through each document and its text

        terms = set(text.split())  # Split the text into individual terms

        for term in terms:  # Iterate through each term in the document

            if term not in index:

                index[term] = {doc\_id}  # If the term is not in the index, create a new set with document ID

            else:

                index[term].add(doc\_id)  # If the term exists, add the document ID to its set

    return index  # Return the built inverted index

# Building the inverted index

inverted\_index = build\_index(documents)

# Function for Boolean AND operation using inverted index

def boolean\_and(operands, index):

    if not operands:  # If there are no operands, return all document IDs

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

    result = index.get(operands[0], set())  # Get the set of document IDs for the first operand

    for term in operands[1:]:  # Iterate through the rest of the operands

        result = result.intersection(index.get(term, set()))  # Compute intersection with sets of document IDs

    return list(result)  # Return the resulting list of document IDs

# Function for Boolean OR operation using inverted index

def boolean\_or(operands, index):

    result = set()  # Initialize an empty set to store the resulting document IDs

    for term in operands:  # Iterate through each term in the query

        result = result.union(index.get(term, set()))  # Union of sets of document IDs for each term

    return list(result)  # Return the resulting list of document IDs

# Function for Boolean NOT operation using inverted index

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

    operand\_set = set(index.get(operand, set()))  # Get the set of document IDs for the operand

    all\_docs\_set = set(range(1, total\_docs + 1))  # Create a set of all document IDs

    return list(all\_docs\_set.difference(operand\_set))  # Return documents not in the operand set

# Example queries

query1 = ["apple", "banana"]  # Query for documents containing both "apple" and "banana"

query2 = ["apple", "orange"]  # Query for documents containing "apple" or "orange"

# Performing Boolean Model queries using inverted index

result1 = boolean\_and(query1, inverted\_index)  # Get documents containing both terms

result2 = boolean\_or(query2, inverted\_index)  # Get documents containing either of the terms

result3 = boolean\_not("orange", inverted\_index, len(documents))  # Get documents not containing "orange"

# Printing results

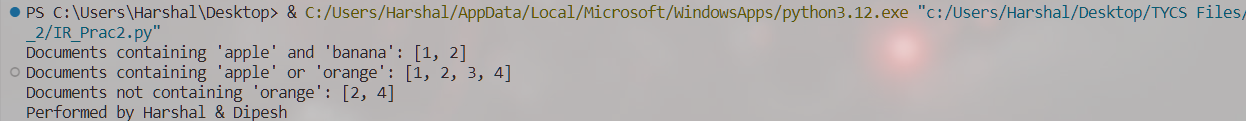
print("Documents containing 'apple' and 'banana':", result1)

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

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

print("Performed by Harshal & Dipesh")

**Output:**

****

**Implement the vector space model with TF-IDF weighting and cosine similarity:**

# Import necessary libraries

from sklearn.feature\_extraction.text import CountVectorizer, TfidfTransformer

import nltk  # Import NLTK to download stopwords

from nltk.corpus import stopwords  # Import stopwords from NLTK

import numpy as np  # Import NumPy library

from numpy.linalg import norm  # Import norm function from NumPy's linear algebra module

# Define the training and test sets of text documents

train\_set = ["The sky is blue.", "The sun is bright."]  # Documents

test\_set = ["The sun in the sky is bright."]  # Query

# Get the stopwords for English language from NLTK

nltk.download('stopwords')

stopWords = stopwords.words('english')

# Initialize CountVectorizer and TfidfTransformer objects

vectorizer = CountVectorizer(stop\_words=stopWords)  # Convert text to matrix of token counts

transformer = TfidfTransformer()  # Convert matrix of token counts to TF-IDF representation

# Convert the training and test sets to arrays of TF-IDF features

trainVectorizerArray = vectorizer.fit\_transform(train\_set).toarray()  # Fittransform training set

testVectorizerArray = vectorizer.transform(test\_set).toarray()  # Transform test set

# Display the TF-IDF arrays for training and test sets

print('Fit Vectorizer to train set:', trainVectorizerArray)

print('Transform Vectorizer to test set:', testVectorizerArray)

# Define a lambda function to calculate cosine similarity between vectors

cx = lambda a, b: round(np.inner(a, b) / (norm(a) \* norm(b)), 3)

# Iterate through each vector in the training set

for vector in trainVectorizerArray:

    print('Train Vector:', vector)

    # Iterate through each vector in the test set

    for testV in testVectorizerArray:

        print('Test Vector:', testV)

        cosine = cx(vector, testV)  # Calculate cosine similarity between vectors

        print('Cosine Similarity:', cosine)

# Fit the transformer to the training set and transform it to TF-IDF representation

transformer.fit(trainVectorizerArray)

print('\nTF-IDF for train set:')

print(transformer.transform(trainVectorizerArray).toarray())

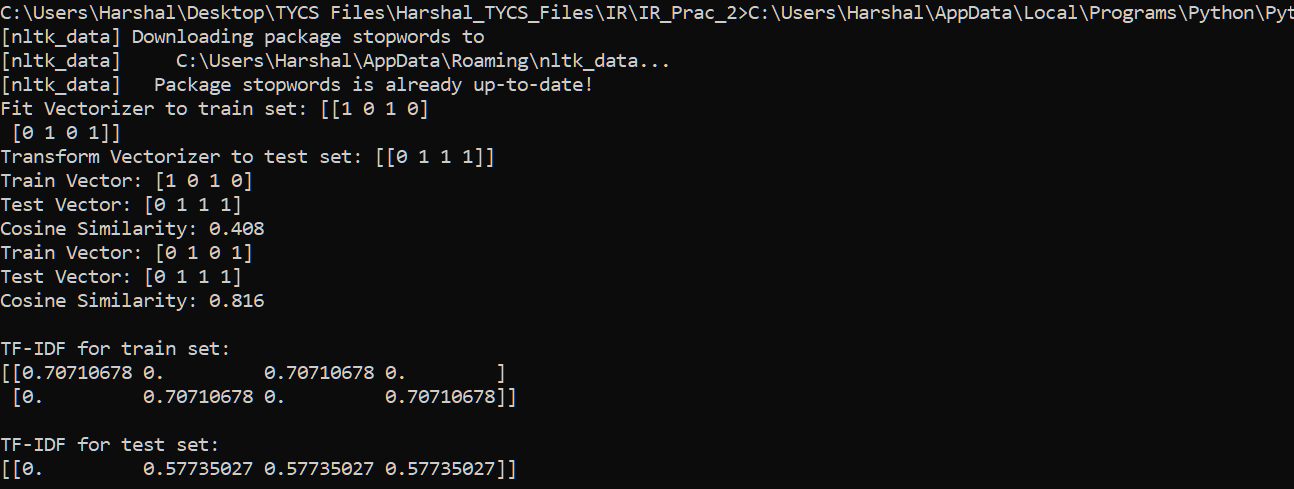
# Fit the transformer to the test set and transform it to TF-IDF representation

transformer.fit(testVectorizerArray)

print('\nTF-IDF for test set:')

tfidf = transformer.transform(testVectorizerArray)

print(tfidf.todense())

****

**Practical No: 3**

# Aim: Spelling Correction in IR Systems

**● Develop a spelling correction module using edit distance algorithms.**

**● Integrate the spelling correction module into an information retrieval system.**

**Input :**

# A naive recursive Python program to find the minimum number

# of operations to convert str1 to str2

def editDistance(str1, str2, m, n):

    # If first string is empty, insert all characters of second string

    if m == 0:

        return n

    # If second string is empty, remove all characters of first string

    if n == 0:

        return m

    # If last characters of both strings are the same, ignore them and recurse for the remaining part

    if str1[m - 1] == str2[n - 1]:

        return editDistance(str1, str2, m - 1, n - 1)

    # If last characters are not the same, consider all three operations:

    # Insert, Remove, and Replace. Compute the minimum cost among them.

    return 1 + min(

        editDistance(str1, str2, m, n - 1),    # Insert

        editDistance(str1, str2, m - 1, n),    # Remove

        editDistance(str1, str2, m - 1, n - 1) # Replace

    )

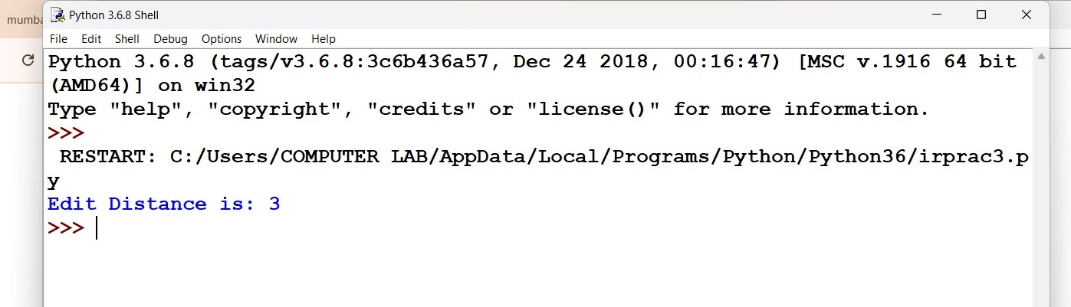
# Driver code

str1 = "sunday"

str2 = "saturday"

print('Edit Distance is:', editDistance(str1, str2, len(str1), len(str2)))

**Output:**

****

**Practical No: 4**

# Aim: Evaluation Metrics for IR Systems

# A) Calculate precision, recall, and F-measure for a given set of retrieval results.

# B) Use an evaluation toolkit to measure average precision and other evaluation metrics.

# A) Calculate precision, recall, and F-measure for a given set of retrieval results.

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))

    '''

    (Optional)

    PPT values:

    true\_positive = 20

    false\_positive = 10

    false\_negative = 30

    '''

    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) Use an evaluation toolkit to measure average precision and other evaluation metrics.**

from sklearn.metrics import average\_precision\_score

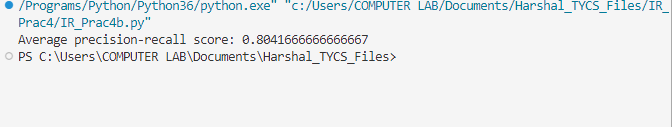
y\_true = [0, 1, 1, 0, 1, 1] #Binary Prediction

y\_scores = [0.1, 0.4, 0.35, 0.8, 0.65, 0.9] #Model's estimation score

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

print(f'Average precision-recall score: {average\_precision}')

**Output:**

****

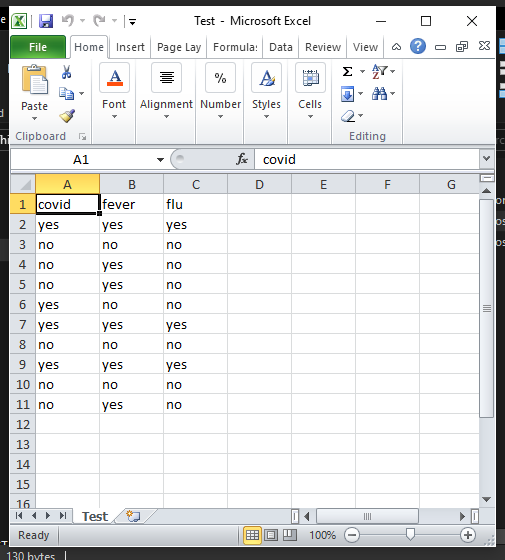
**Practical No: 5**

# Aim: Text Categorization

# A) Implement a text classification algorithm (e.g., Naive Bayes or Support Vector Machines).

# B) Train the classifier on a labelled dataset and evaluate its performance.

**Create a Dataset file:**

**Test.csv:**  
****

# A) Implement a text classification algorithm (e.g., Naive Bayes or Support Vector Machines).

# Import necessary libraries

import pandas as pd

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

from sklearn.feature\_extraction.text import CountVectorizer

from sklearn.naive\_bayes import MultinomialNB

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

# Load the CSV file

df = pd.read\_csv(r"C:\Users\Harshal\Desktop\TYCS Files\Harshal\_TYCS\_Files\IR\IR\_Prac5\Test.csv")

# Combine 'covid' and 'fever' columns into a single feature column

data = df["covid"] + " " + df["fever"]

X = data.astype(str)  # Test data

y = df['flu']  # Labels

# Splitting the data into training and test data

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

# Converting data into bag-of-words format

vectorizer = CountVectorizer()  # Initialize CountVectorizer

# Transform training and test data

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

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

# Initialize and train the Multinomial Naive Bayes classifier

classifier = MultinomialNB()

classifier.fit(X\_train\_counts, y\_train)

# Load another dataset to test the model

data1 = pd.read\_csv(r"C:\Users\Harshal\Desktop\TYCS Files\Harshal\_TYCS\_Files\IR\IR\_Prac5\Test.csv")

new\_data = data1["covid"] + " " + data1["fever"]

# Transform new data using the trained vectorizer

new\_data\_counts = vectorizer.transform(new\_data.astype(str))

# Make predictions on the new dataset

predictions = classifier.predict(new\_data\_counts)

# Output the predictions

print("Predictions for new data:")

print(predictions)

# Evaluate the model using test data

accuracy = accuracy\_score(y\_test, classifier.predict(X\_test\_counts))

print(f"\nAccuracy: {accuracy:.2f}")

print("Classification Report:")

print(classification\_report(y\_test, classifier.predict(X\_test\_counts)))

# Convert predictions to DataFrame

predictions\_df = pd.DataFrame(predictions, columns=['flu\_prediction'])

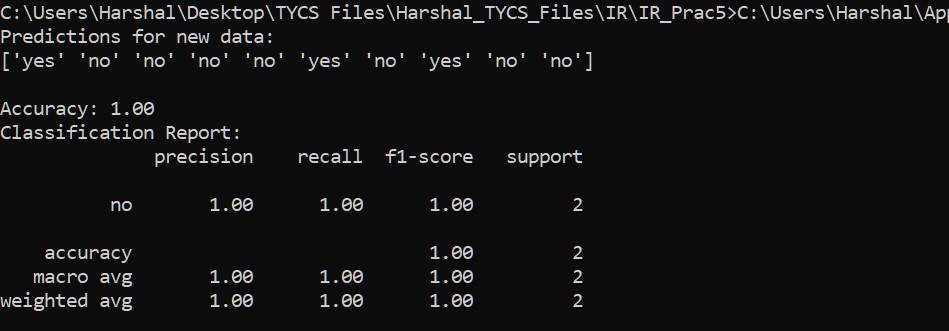
# Concatenate predictions with the original test DataFrame

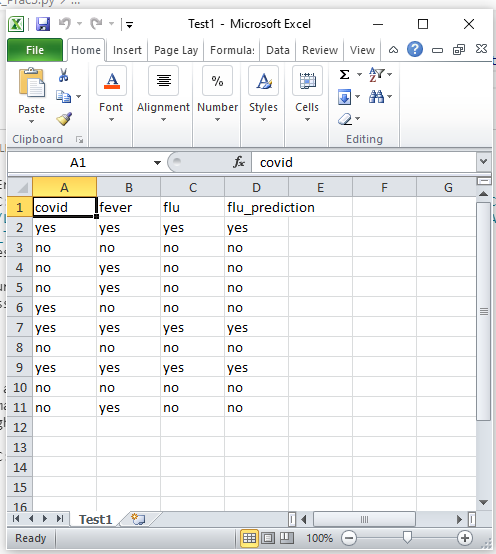
data1 = pd.concat([data1, predictions\_df], axis=1)

# Save the updated DataFrame to a new CSV file

data1.to\_csv(r"C:\Users\Harshal\Desktop\TYCS Files\Harshal\_TYCS\_Files\IR\IR\_Prac5\Test1.csv", index=False)

**Output:**

****

****

**Practical No: 6**

**Aim: Clustering for Information Retrieval**

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

**Input:**

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

             ]

# Create a TfidfVectorizer object

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

# Learn vocabulary and idf from training set.

X = vectorizer.fit\_transform(documents)

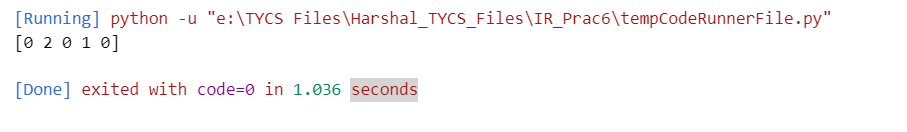
# Perform k-means clustering

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

# Print cluster labels for each document

print(kmeans.labels\_)

**Output:**

****

**Practical No: 7**

# Aim: Web Crawling and Indexing

# A) Develop a web crawler to fetch and index web pages.

# B) Handle challenges such as robots.txt, dynamic content, and crawling delays.

**Input:**#pip install beautifulsoup4

import requests

from bs4 import BeautifulSoup

import time

from urllib.parse import urljoin, urlparse

from urllib.robotparser import RobotFileParser

def get\_html(url):

    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'}

    try:

        response = requests.get(url, headers=headers)

        response.raise\_for\_status()

        return response.text

    except requests.exceptions.HTTPError as errh:

        print(f"HTTP Error: {errh}")

    except requests.exceptions.RequestException as err:

        print(f"Request Error: {err}")

    return None

def save\_robots\_txt(url):

    try:

        robots\_url = urljoin(url, '/robots.txt')

        robots\_content = get\_html(robots\_url)

        if robots\_content:

            with open('robots.txt', 'wb') as file:

                file.write(robots\_content.encode('utf-8-sig'))

    except Exception as e:

        print(f"Error saving robots.txt: {e}")

def load\_robots\_txt():

    try:

        with open('robots.txt', 'rb') as file:

            return file.read().decode('utf-8-sig')

    except FileNotFoundError:

        return None

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

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

    links = []

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

        absolute\_url = urljoin(base\_url, link['href'])

        links.append(absolute\_url)

    return links

def is\_allowed\_by\_robots(url, robots\_content):

    parser = RobotFileParser()

    parser.parse(robots\_content.split('\n'))

    return parser.can\_fetch('\*', url)

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

    visited\_urls = set()

    def recursive\_crawl(url, depth, robots\_content):

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

            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, robots\_content)

    save\_robots\_txt(start\_url)

    robots\_content = load\_robots\_txt()

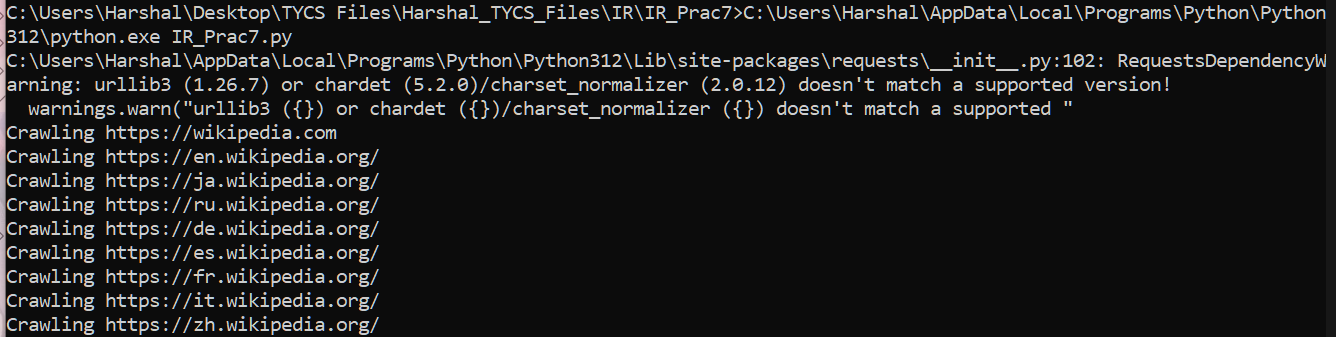
    if not robots\_content:

        print("Unable to retrieve robots.txt. Crawling without restrictions."

    recursive\_crawl(start\_url, 1, robots\_content)

# Example usage:

crawl('https://wikipedia.com', max\_depth=2, delay=2)

**Output:   
**

**robots.txt file:**

# robots.txt for http://www.wikipedia.org/ and friends

#

# Please note: There are a lot of pages on this site, and there are

# some misbehaved spiders out there that go \_way\_ too fast. If you're

# irresponsible, your access to the site may be blocked.

#

# Observed spamming large amounts of https://en.wikipedia.org/?curid=NNNNNN

# and ignoring 429 ratelimit responses, claims to respect robots:

# http://mj12bot.com/

User-agent: MJ12bot

Disallow: /

# advertising-related bots:

User-agent: Mediapartners-Google\*

Disallow: /

# Wikipedia work bots:

User-agent: IsraBot

Disallow:

User-agent: Orthogaffe

Disallow:

# Crawlers that are kind enough to obey, but which we'd rather not have

# unless they're feeding search engines.

User-agent: UbiCrawler

Disallow: /

User-agent: DOC

Disallow: /

User-agent: Zao

Disallow: /

# Some bots are known to be trouble, particularly those designed to copy

# entire sites. Please obey robots.txt.

User-agent: sitecheck.internetseer.com

Disallow: /

User-agent: Zealbot

Disallow: /

**Practical No: 8**

# Aim: : Link Analysis and PageRank

# A) Implement the PageRank algorithm to rank web pages based on link analysis.

# B) Apply the PageRank algorithm to a small web graph and analyse the results.

**Python Code:**import numpy as np

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

    # Validate damping factor

    if not (0 < damping\_factor < 1):

        raise ValueError("Damping factor must be between 0 and 1.")

    # Get the number of nodes

    num\_nodes = len(graph)

    # Initialize PageRank values

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

    # Iterative PageRank calculation

    for \_ in range(max\_iterations):

        prev\_page\_ranks = np.copy(page\_ranks)

        for node in range(num\_nodes):

            # Calculate the contribution from incoming links

            incoming\_links = [i for i, v in enumerate(graph) if node in v]

            if not incoming\_links:

                continue

            # Calculate the PageRank for the current node

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

                               damping\_factor \* sum(prev\_page\_ranks[link] / len(graph[link]) for link in incoming\_links if len(graph[link]) > 0)

        # Check for convergence

        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

    ]

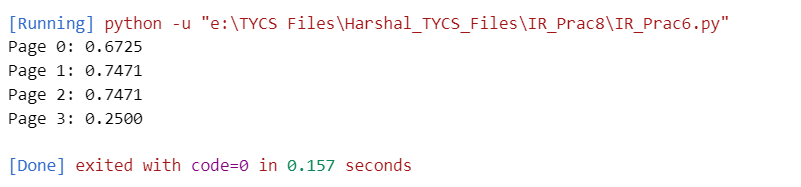
    # Calculate PageRank

    result = page\_rank(web\_graph)

    # Display PageRank values

    for i, pr in enumerate(result):

        print(f"Page {i}: {pr:.4f}")

**Output:   
**