# PRACTICAL NO 1 [A]

**AIM:** Document Indexing and Retrieval

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

**SOLUTION:**

1. Implement an inverted index construction algorithm.

**INPUT:** document1 = "The quick brown fox jumped over the lazy dog" document2 = "The lazy dog slept in the sun" import nltk

from nltk.corpus import stopwords stop\_words = set(stopwords.words('english'))

# Tokenize and filter stopwords

tokens1 = [word for word in document1.lower().split() if word not in stop\_words] tokens2 = [word for word in document2.lower().split() if word not in stop\_words]

# Build inverted index and occurrence counts inverted\_index = {}

occurrences = {"Document 1": {}, "Document 2": {}} for term in set(tokens1 + tokens2):

inverted\_index[term] = [] if term in tokens1:

inverted\_index[term].append("Document 1") occurrences["Document 1"][term] = tokens1.count(term) if term in tokens2:

inverted\_index[term].append("Document 2") occurrences["Document 2"][term] = tokens2.count(term)

# Print results

print("Inverted Index:", inverted\_index)

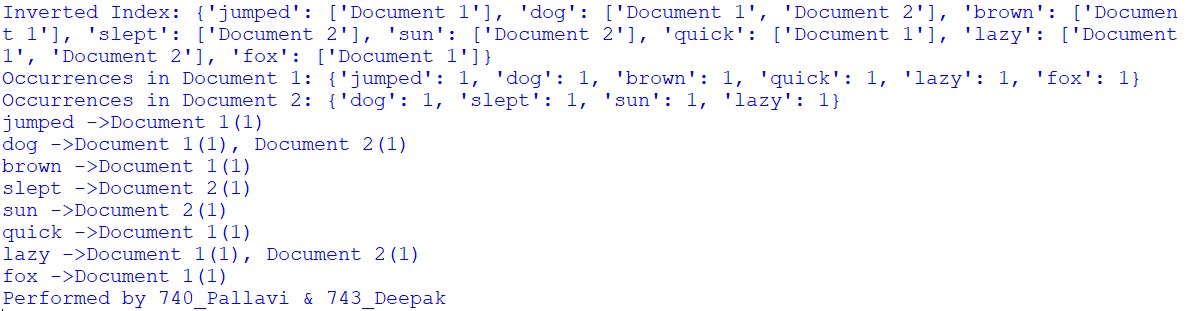
print("Occurrences in Document 1:", occurrences["Document 1"]) print("Occurrences in Document 2:", occurrences["Document 2"])

# Print inverted index with occurrences for term, docs in inverted\_index.items():

print(f"{term} ->", end="")

print(", ".join(f"{doc}({occurrences[doc].get(term, 0)})" for doc in docs)) print("Performed by 740\_Pallavi & 743\_Deepak")

**OUTPUT:**



# PRACTICAL NO 1 [B]

2. Build a simple document retrieval system using the constructed index.

**INPUT:** import re

from collections import defaultdict class DocumentRetrievalSystem: def \_\_init\_\_(self):

self.index = defaultdict(list) self.documents = [] def add\_documents(self, documents): for doc\_id, document in enumerate(documents):

self.documents.append(document) for term in self.tokenize(document): self.index[term].append(doc\_id) def search(self, query):

query\_terms = self.tokenize(query)

result\_docs = set(self.index[query\_terms[0]]) if query\_terms and query\_terms[0] in self.index else set() for term in query\_terms[1:]:

result\_docs &= set(self.index[term])

return [self.documents[doc\_id] for doc\_id in result\_docs] def tokenize(text):

return re.findall(r'\b\w+\b', text.lower()) if \_\_name\_\_ == "\_\_main\_\_":

retrieval\_system = DocumentRetrievalSystem() retrieval\_system.add\_documents([

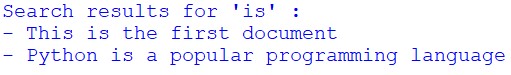
"This is the first document",

"Python is a popular programming language", "Document retrieval systems are important"]) query = "is"

results = retrieval\_system.search(query) if results:

print(f"Search results for '{query}':") for result in results: print("-", result) else: print(f"No results found for '{query}'.")

**OUTPUT:**



PRACTICAL NO 2[A]

**AIM :** Retrieval Models

Implement the Boolean retrieval model and process queries.

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

**SOLUTION:** 1) Implement the Boolean retrieval model and process queries **INPUT:** 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(): for term in set(text.split()):

index.setdefault(term, set()).add(doc\_id) return index

inverted\_index = build\_index(documents) def boolean\_and(operands, index): result = index.get(operands[0], set()) for term in operands[1:]:

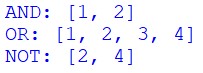
result &= index.get(term, set()) return list(result)

def boolean\_or(operands, index, total\_docs):

result = set() for term in operands:

result |= index.get(term, set()) return list(result | set(range(1, total\_docs + 1))) def boolean\_not(operand, index, total\_docs):

return list(set(range(1, total\_docs + 1)) - index.get(operand, set())) query1, query2, query3 = ["apple", "banana"], ["apple", "orange"], "orange" print("AND:", boolean\_and(query1, inverted\_index)) print("OR:", boolean\_or(query2, inverted\_index, len(documents))) print("NOT:", boolean\_not(query3, inverted\_index, len(documents))) **OUTPUT:**



PRACTICAL NO 2[B]

2) Implement the vector space model with TF-IDF weighting and cosine similarity.

**INPUT:** from sklearn.feature\_extraction.text import CountVectorizer, TfidfTransformer import nltk

from nltk.corpus import stopwords import numpy as np from numpy.linalg import norm

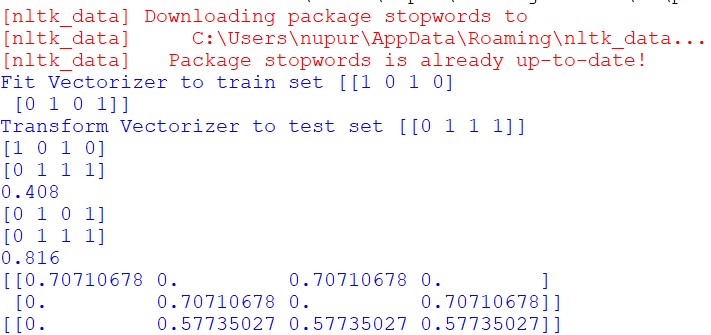
train\_set = ["The sky is blue.", "The sun is bright."] test\_set = ["The sun in the sky is bright."] nltk.download('stopwords') stopWords = stopwords.words('english') vectorizer = CountVectorizer(stop\_words=stopWords) trainVectorizerArray = vectorizer.fit\_transform(train\_set).toarray() testVectorizerArray = vectorizer.transform(test\_set).toarray() print('Fit Vectorizer to train set', trainVectorizerArray) print('Transform Vectorizer to test set', testVectorizerArray) cx = lambda a, b: round(np.inner(a, b) / (norm(a) \* norm(b)), 3) for vector in trainVectorizerArray: print(vector) for testV in testVectorizerArray:

print(testV)

cosine = cx(vector, testV) print(cosine)

transformer = TfidfTransformer()

print(transformer.fit\_transform(trainVectorizerArray).toarray()) print(transformer.fit\_transform(testVectorizerArray).todense()) **OUTPUT:**



# 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:** def editDistance(str1, str2, m, n): if m==0: return n if n==0: return m if str1[m-1] == str2[n-1]:

return editDistance(str1, str2, m-1, n-1) return 1 + min(editDistance(str1, str2, m, n-1), editDistance(str1, str2, m-1, n), editDistance(str1, str2, m-1, n-1) ) str1 = "sunday" str2 = "saturday"

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

**OUTPUT:**



# PRACTICAL NO 4[A]

**AIM :** Evaluation Metrics for IR Systems

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

**SOLUTION:**

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

**INPUT:**

def calculate\_metrics(retrieved\_set, relevant\_set): tp = len(retrieved\_set & relevant\_set) # True Positives fp = len(retrieved\_set - relevant\_set) # False Positives fn = len(relevant\_set - retrieved\_set) # False Negatives

print(f"True Positive: {tp}\nFalse Positive: {fp}\nFalse Negative: {fn}\n")

precision = tp / (tp + fp) if (tp + fp) > 0 else 0 recall = tp / (tp + fn) if (tp + fn) > 0 else 0

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

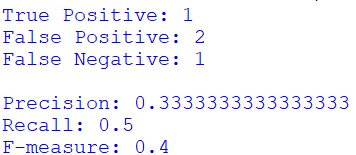
return precision, recall, f\_measure

# Example input

retrieved\_set = {"doc1", "doc2", "doc3"} # Predicted set relevant\_set = {"doc1", "doc4"} # Relevant set

# Calculate and display metrics

precision, recall, f\_measure = calculate\_metrics(retrieved\_set, relevant\_set) print(f"Precision: {precision}\nRecall: {recall}\nF-measure: {f\_measure}") **OUTPUT:**



PRACTICAL NO 4[B]

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

**INPUT:** 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

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

**SOLUTION :**

Dataset.csv file

**INPUT:**

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 df=pd.read\_csv(r"C:\Users\nupur\sem 6 journals\IR\Dataset.csv") data= df["covid"]+" "+df["fever"] X=data.astype(str) y=df['flu']

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

X\_train\_counts=vectorizer.fit\_transform(X\_train) X\_test\_counts = vectorizer.transform(X\_test) classifier=MultinomialNB() classifier.fit(X\_train\_counts, y\_train) data1=pd.read\_csv(r"C:\Users\nupur\sem 6 journals\IR\Test.csv") new\_data=data1["covid"]+" "+data1["fever"]

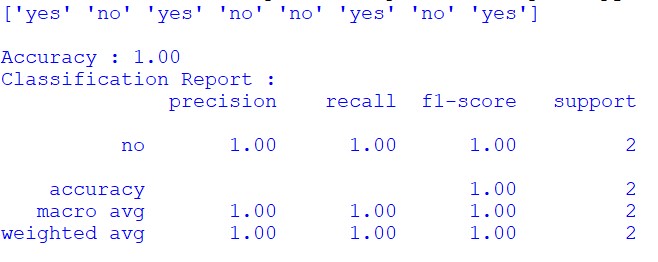
new\_data\_counts=vectorizer.transform(new\_data.astype(str)) predictions=classifier.predict(new\_data\_counts) new\_data=predictions print(new\_data)

accuracy=accuracy\_score(y\_test, classifier.predict(X\_test\_counts)) print(f"\nAccuracy : {accuracy:.2f}") print("Classification Report :")

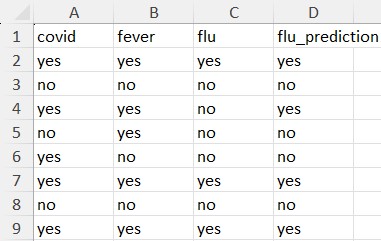
print(classification\_report(y\_test,classifier.predict(X\_test\_counts))) predictions\_df=pd.DataFrame(predictions,columns=['flu\_prediction']) data1=pd.concat([data1,predictions\_df],axis=1)

data1.to\_csv(r"C:\Users\nupur\sem 6 journals\IR\test1.csv",index=False)

**OUTPUT:**



test1.csv



# 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",

"Dogs are often called 'Man's best friend'.",

"Some digs are trained to assist people with disablities.",

"The sun rises in the east and set on the west.",

"Many cats enjoy climbing trees and chasing toys.",

]

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

X=vectorizer.fit\_transform(documents)

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

**OUTPUT:**



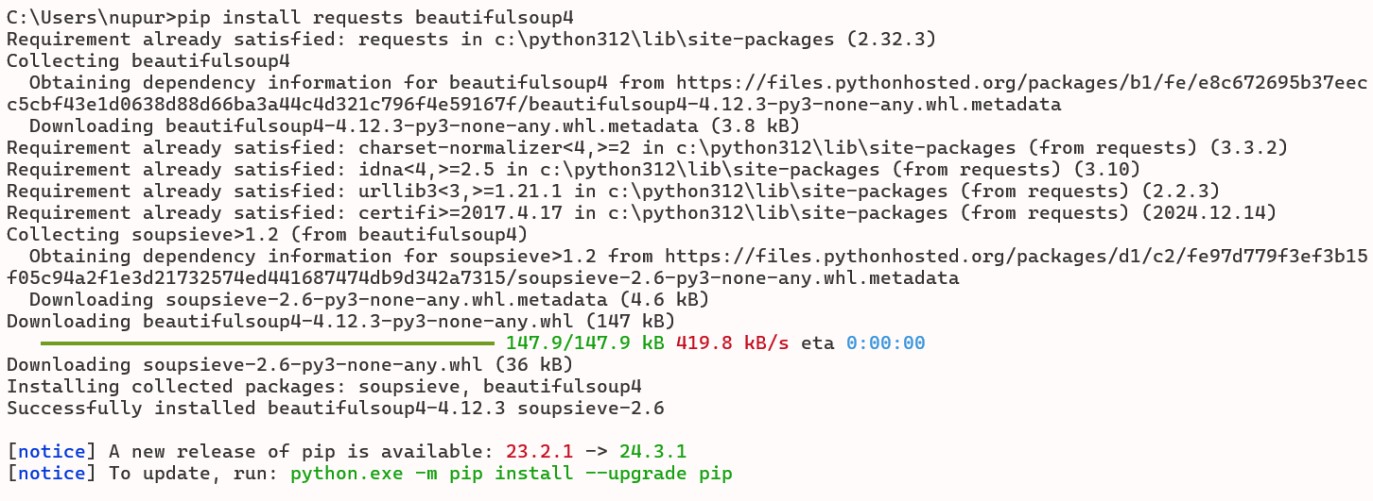
# PRACTICAL NO 7

**AIM :** Web Crawling and Indexing

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

**SOLUTION:**

Install the following modules using pip: pip install requests beautifulsoap4



**INPUT:**

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): print("Hello") 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:

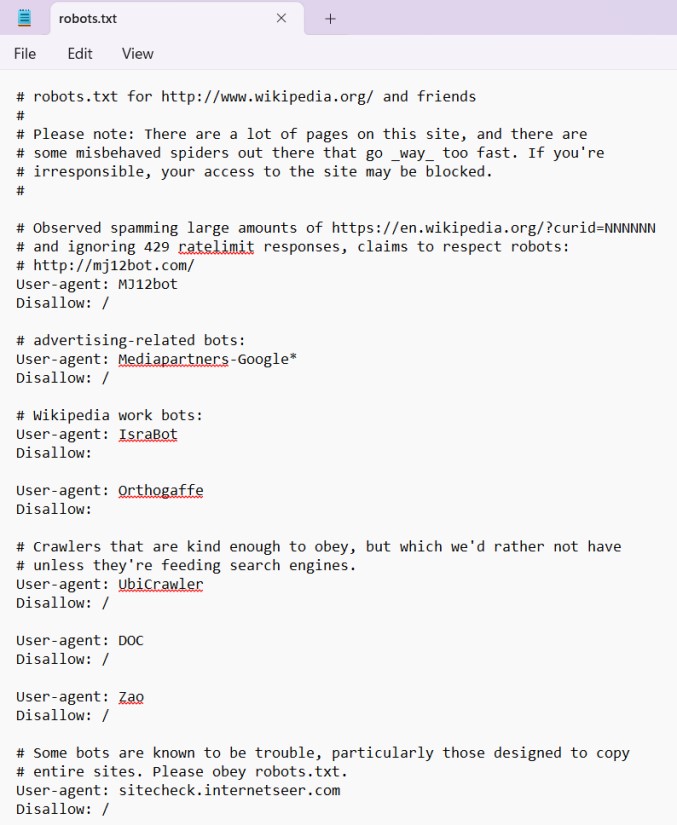
print("Unable to retrieve robots.txt. Crawling without restrictions.") else:

print("Using robots.txt for crawling restrictions.") recursive\_crawl(start\_url,1,robots\_content) print("Performed by Nupur Karpe") crawl('https://', max\_depth=2, delay=2)

**OUTPUT:**



Robot.txt file



# PRACTICAL NO 8

**AIM:** Link Analysis and PageRank

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

**INPUT:**

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

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

web\_graph = [

[1, 2],

[0, 2],

[0, 1],

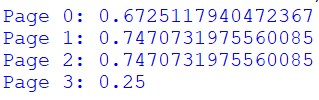
[1, 2],

]

result = page\_rank(web\_graph) for i, pr in enumerate(result):

print(f"Page {i}: {pr}")

**OUTPUT:**



# PRACTICAL NO 9

**AIM:** Learning to Rank

Implement a learning to rank algorithm (e.g., RankSVM or RankBoost).

Train the ranking model using labelled data and evaluate its effectiveness.

**INPUT:**

import numpy as np from sklearn.svm import SVC

from sklearn.model\_selection import train\_test\_split from sklearn.metrics import ndcg\_scor

# Example dataset

X = np.array([[3, 2, 1], [2, 1, 0], [0, 1, 2], [1, 2, 0], [2, 1, 3], [1, 0, 2]]) y = np.array([1, 0, 0, 1, 0, 1])

# Split data

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

# Train RankSVM model model = SVC(kernel='linear', C=1.0) model.fit(X\_train, y\_train)

# Evaluate model

ndcg = ndcg\_score([y\_test], [model.predict(X\_test)]) print(f"NDCG Score: {ndcg:.4f}")

**OUTPUT:**



# PRACTICAL NO 10

**AIM :** Advanced Topics in Information Retrieval

Implement a text summarization algorithm (e.g., extractive or abstractive).

Build a question-answering system using techniques such as information extraction

**INPUT:**

import nltk nltk.download('punkt') nltk.download('stopwords')

from nltk.tokenize import sent\_tokenize, word\_tokenize from nltk.corpus import stopwords from collections import Counter def generate\_summary(text, num\_sentences=2):

sentences = sent\_tokenize(text)

words = [word.lower() for word in word\_tokenize(text) if word.isalnum()] words = [word for word in words if word not in stopwords.words("english")] word\_freq = Counter(words)

sentence\_scores = {sent: sum(word\_freq[word] for word in word\_tokenize(sent.lower()) if word in word\_freq) for sent in sentences} summary = ' '.join(sorted(sentence\_scores, key=sentence\_scores.get, reverse=True)[:num\_sentences]) return summary

# Example usage

text = """

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

As such, NLP is related to the area of human–computer interaction.

Many challenges in NLP involve natural language understanding, that is, enabling computers to derive meaning from human or natural language input, and others involve natural language generation.

"""

print(generate\_summary(text))

**OUTPUT:**

