**Q1) Inverted index construction for**

**doc1: The computer science students are appearing for practical examination.**

**doc2: computer science practical examination will start tomorrow.**  
  
Code :

from collections import defaultdict

documents = {

"doc1": "The computer science students are appearing for practical examination.",

"doc2": "computer science practical examination will start tomorrow."

}

def preprocess(text):

return text.lower().replace(".", "").split()

inverted\_index = defaultdict(list)

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

words = preprocess(text)

for position, word in enumerate(words):

if doc\_id not in inverted\_index[word]:

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

print("Inverted Index:")

for term in sorted(inverted\_index):

print(f"{term}: {inverted\_index[term]}")

query\_terms = ["computer", "science"]

result\_docs = set(documents.keys())

for term in query\_terms:

if term in inverted\_index:

result\_docs = result\_docs.intersection(set(inverted\_index[term]))

else:

result\_docs = set()

break

print("\nDocuments containing the terms 'computer science':")

print(result\_docs)

**Q2) Build a question-answering system using techniques such as information extraction**  
Code :

import spacy

nlp = spacy.load("en\_core\_web\_sm")

context = """Dr. A.P.J. Abdul Kalam was born in Rameswaram, India, in 1931. He became the 11th President of India in 2002."""

doc = nlp(context)

def answer(question):

question = question.lower()

for ent in doc.ents:

if "who" in question and ent.label\_ == "PERSON":

return ent.text

if "when" in question and ent.label\_ == "DATE":

return ent.text

if "where" in question and ent.label\_ == "GPE":

return ent.text

return "No answer found."

print("Q: Where was Kalam born?")

print("A:", answer("Where was Kalam born?"))

print("Q: When was Kalam born?")

print("A:", answer("When was Kalam born?"))

print("Q: Who became President in 2002?")

print("A:", answer("Who became President in 2002?"))

**Q3) Inverted index construction for**

**doc1: The quick brown fox jumped over the lazy dog**

**doc2: The lazy dog slept in the sun.**

Code:

from collections import defaultdict

documents = {

"doc1": "The quick brown fox jumped over the lazy dog.",

"doc2": "The lazy dog slept in the sun."

}

def preprocess(text):

return text.lower().replace(".", "").split()

inverted\_index = defaultdict(list)

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

words = preprocess(text)

for position, word in enumerate(words):

if doc\_id not in inverted\_index[word]:

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

print("Inverted Index:")

for term in sorted(inverted\_index):

print(f"{term}: {inverted\_index[term]}")

query\_terms = ["lazy", "sun"]

result\_docs = set(documents.keys())

for term in query\_terms:

if term in inverted\_index:

result\_docs = result\_docs.intersection(set(inverted\_index[term]))

else:

result\_docs = set()

break

print("\nDocuments containing the terms 'lazy sun':")

print(result\_docs)  
  
**Q4) calculate precision, recall & F-measure: true positive is 60, false positive is 30 & false negative is 20**  
  
Code:

true\_positive = 60

false\_positive = 30

false\_negative = 20

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

precision = true\_positive / (true\_positive + false\_positive)

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

print(f"Recall: {recall:.2f}")

print(f"Precision: {precision:.2f}")

print(f"F-score: {f\_score:.2f}")

**Q5) spelling correction module using edit distance : “nature” and “creature”**

Code:

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 = "nature"

str2 = "creature"

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

**Q6) Boolean retrieval model for  
Doc1:The cat chased the dog around the garden,**

**Doc2: She was sitting in the garden last night ,**

**Doc 3: I read the book the night before.**

**Process the query “garden or night”.**

Code:  
documents = {

1: "The cat chased the dog around the garden.",

2: "She was sitting in the garden last night.",

3: "I read the book the night before."

}

def build\_index(docs):

index = {}

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

for term in set(text.lower().split()):

index.setdefault(term.strip(".,!?"), set()).add(doc\_id)

return index

index = build\_index(documents)

def boolean\_or(terms):

return list(set.union(\*(index.get(term, set()) for term in terms)))

query = ["garden", "night"]

result = boolean\_or(query)

print("Documents matching 'garden OR night':", result)  
  
**Q7) Web Crawler**   
Code:

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)

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

**Q8) Page Rank : for**

**Page A has links to pages B, C, and D.**

**Page B has links to pages C and E.**

**Page C has links to pages A and D.**  
  
Code:

links = {

'A': ['B', 'C', 'D'],

'B': ['C', 'E'],

'C': ['A', 'D'],

'D': [],

'E': []

}

pages = links.keys()

n = len(pages)

damping = 0.85

pagerank = {page: 1 / n for page in pages}

from collections import defaultdict

incoming\_links = defaultdict(list)

for src, targets in links.items():

for target in targets:

incoming\_links[target].append(src)

def compute\_pagerank(iterations=10):

global pagerank

for \_ in range(iterations):

new\_rank = {}

for page in pages:

rank\_sum = 0

for incoming in incoming\_links[page]:

rank\_sum += pagerank[incoming] / len(links[incoming])

new\_rank[page] = (1 - damping) / n + damping \* rank\_sum

pagerank = new\_rank

compute\_pagerank()

print("Final PageRank scores:")

for page, score in sorted(pagerank.items(), key=lambda x: x[1], reverse=True):

print(f"Page {page}: {score:.4f}")

**Q9) Implement a text summarization algorithm .**

Code:  
from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

import re

# Input text

text = """

India is a country in South Asia. It is the seventh-largest country by land area,

the second-most populous country, and the most populous democracy in the world.

Bounded by the Indian Ocean on the south, the Arabian Sea on the southwest, and the Bay of Bengal on the southeast,

it shares land borders with Pakistan to the northwest; China, Nepal, and Bhutan to the north; and Bangladesh and Myanmar to the east.

India has a rich cultural heritage. It is known for its diversity in languages, traditions, and festivals.

India has made remarkable progress in science, technology, and space research in the recent decades.

"""

# Split text into sentences using regex

sentences = re.split(r'(?<=[.!?])\s+', text.strip())

# TF-IDF Vectorizer

vectorizer = TfidfVectorizer()

X = vectorizer.fit\_transform(sentences)

# Calculate similarity matrix

similarity\_matrix = cosine\_similarity(X)

# Score each sentence by summing its similarities to other sentences

sentence\_scores = similarity\_matrix.sum(axis=1)

# Get indices of top 3 sentences

top\_n = 3

top\_sentence\_indices = sentence\_scores.argsort()[-top\_n:][::-1]

# Get the top sentences in original order

summary\_sentences = [sentences[i] for i in sorted(top\_sentence\_indices)]

# Print the summary

summary = ' '.join(summary\_sentences)

print("Summary:")

print(summary)

**Q10) cosine similarity for**

**query="gold silver truck"**

**document="shipment of gold damaged in a gold fire"**

Code:

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

# Input text

query = "gold silver truck"

document = "shipment of gold damaged in a gold fire"

# Create the TF-IDF vectorizer

vectorizer = TfidfVectorizer()

# Combine the query and document into a single corpus

corpus = [query, document]

# Convert the text into TF-IDF vectors

tfidf\_matrix = vectorizer.fit\_transform(corpus)

# Calculate cosine similarity between the query and document

cos\_sim = cosine\_similarity(tfidf\_matrix[0:1], tfidf\_matrix[1:2])

print(f"Cosine Similarity: {cos\_sim[0][0]}")

**Q11) Boolean Retreival Model :**

**Document 1:BSc lectures start at 7.**

**Document 2:My lectures are over.**

**Document 3: Today is a holiday.**

**Process the query “not lectures”**  
  
Code:  
documents = {

1: ":BSc lectures start at 7.",

2: "My lectures are over..",

3: "Today is a holiday."

}

def build\_index(docs):

index = {}

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

for term in set(text.lower().split()):

index.setdefault(term.strip(".,!?"), set()).add(doc\_id)

return index

index = build\_index(documents)

def boolean\_or(terms):

return list(set.union(\*(index.get(term, set()) for term in terms)))

query = ["not","lectures"]

result = boolean\_or(query)

print("Documents matching 'not lectures':", result)

**Q12). vector space model with TF-IDF weighting**

**Document 1: "Document about python programming language and data analysis."**

**Document 2: "Document discussing machine learning algorithms and programming techniques.", Document 3: "Overview of natural language processing and its applications."**

**query = "python programming"**  
  
Code:

from sklearn.feature\_extraction.text import TfidfVectorizer

# Documents provided

documents = [

"Document about python programming language and data analysis.",

"Document discussing machine learning algorithms and programming techniques.",

"Overview of natural language processing and its applications."

]

# Query

query = ["python programming"]

# Create a TF-IDF vectorizer

vectorizer = TfidfVectorizer()

# Fit on the documents + query to ensure consistent vectorization

tfidf\_matrix = vectorizer.fit\_transform(documents + query)

# Separate documents and query

doc\_vectors = tfidf\_matrix[:-1]

query\_vector = tfidf\_matrix[-1]

# Show TF-IDF feature names and vectors (optional debug output)

feature\_names = vectorizer.get\_feature\_names\_out()

print("TF-IDF Feature Names:")

print(feature\_names)

print("\nTF-IDF Matrix (Documents):")

print(doc\_vectors.toarray())

print("\nTF-IDF Vector (Query):")

print(query\_vector.toarray())

**Q. 13) Calculate the cosine similarity between the query and each document from the above problem.**

Code:  
from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

# ----- Documents and Query -----

documents = [

"Document about python programming language and data analysis.",

"Document discussing machine learning algorithms and programming techniques.",

"Overview of natural language processing and its applications."

]

query = ["python programming"]

# ----- TF-IDF Vectorization -----

vectorizer = TfidfVectorizer()

tfidf\_matrix = vectorizer.fit\_transform(documents + query)

# Split document vectors and query vector

doc\_vectors = tfidf\_matrix[:-1]

query\_vector = tfidf\_matrix[-1]

# ----- Cosine Similarity Calculation -----

cosine\_similarities = cosine\_similarity(query\_vector, doc\_vectors).flatten()

# ----- Display Results -----

for i, score in enumerate(cosine\_similarities):

print(f"Cosine similarity between Query and Document {i+1}: {score:.4f}")

**Q14) Use an evaluation toolkit to measure average precision and other evaluation metrics.**  
Code:

from sklearn.metrics import precision\_score, recall\_score, f1\_score, average\_precision\_score

# Simulated binary predictions and labels (for toolkit usage)

# 1 = Positive, 0 = Negative

y\_true = [1]\*60 + [0]\*30 + [1]\*20 # 60 TP, 30 FP, 20 FN => 80 actual positives, 30 actual negatives

y\_pred = [1]\*60 + [1]\*30 + [0]\*20 # Model predicted 90 positives (60 correct, 30 incorrect)

# Calculate metrics

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

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

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

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

# Output

print(f"Toolkit Precision: {precision:.4f}")

print(f"Toolkit Recall: {recall:.4f}")

print(f"Toolkit F1-score: {f1:.4f}")

print(f"Average Precision Score: {avg\_precision:.4f}")

**15] Naïve bayes (20 newsgroups not find)**

import nltk

from nltk.corpus import movie\_reviews

import random

from sklearn.naive\_bayes import MultinomialNB

from sklearn.feature\_extraction.text import CountVectorizer

from sklearn.pipeline import make\_pipeline

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

from sklearn import metrics

nltk.download('movie\_reviews')

docs = [(movie\_reviews.raw(fileid), category)

for category in movie\_reviews.categories()

for fileid in movie\_reviews.fileids(category)]

random.shuffle(docs)

X = [doc for doc, label in docs]

y = [label for doc, label in docs]

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

model = make\_pipeline(CountVectorizer(), MultinomialNB())

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

print(metrics.classification\_report(y\_test, y\_pred))

print("Accuracy:", metrics.accuracy\_score(y\_test, y\_pred))

**16] SVM (20 newsgroups not found)**

import nltk

from nltk.corpus import movie\_reviews

import random

from sklearn.svm import SVC

from sklearn.feature\_extraction.text import CountVectorizer

from sklearn.pipeline import make\_pipeline

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

from sklearn import metrics

# Download dataset if not already downloaded

nltk.download('movie\_reviews')

docs = [(movie\_reviews.raw(fileid), category)

for category in movie\_reviews.categories()

for fileid in movie\_reviews.fileids(category)]

random.shuffle(docs)

X = [doc for doc, label in docs]

y = [label for doc, label in docs]

# Split into training and test sets

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

# Build a pipeline: CountVectorizer + SVM

model = make\_pipeline(CountVectorizer(), SVC(kernel='linear'))

# Train the model

model.fit(X\_train, y\_train)

# Predict on the test set

y\_pred = model.predict(X\_test)

print(metrics.classification\_report(y\_test, y\_pred))

print("Accuracy:", metrics.accuracy\_score(y\_test, y\_pred))

**17] Question-answering (national bird)**

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

corpus = [

"India has the second-largest population in the world.",

"It is surrounded by oceans from three sides which are Bay Of Bengal in the east, the Arabian Sea in the west and Indian Ocean in the south.",

"Tiger is the national animal of India.",

"Peacock is the national bird of India.",

"Mango is the national fruit of India."

]

query = "Which is the national bird of India?"

documents = corpus + [query]

vectorizer = TfidfVectorizer()

tfidf\_matrix = vectorizer.fit\_transform(documents)

cosine\_similarities = cosine\_similarity(tfidf\_matrix[-1], tfidf\_matrix[:-1]).flatten()

most\_similar\_idx = cosine\_similarities.argmax()

most\_similar\_sentence = corpus[most\_similar\_idx]

print(f"Question: {query}")

print(f"Answer: {most\_similar\_sentence}")

**18]precision , recall , f-measure(60,30,20)**

# Given values

true\_positive = 60

false\_positive = 30

false\_negative = 20

precision = true\_positive / (true\_positive + false\_positive)

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

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

print(f"Precision: {precision:.2f}")

print(f"Recall: {recall:.2f}")

print(f"F-measure: {f\_score:.2f}")

**19] pagerank algorithm ([a-b,c,d] [b-c,e], [c-a,d])**

import networkx as nx

G = nx.DiGraph()

G.add\_edges\_from([

('A', 'B'), ('A', 'C'), ('A', 'D'),

('B', 'C'), ('B', 'E'),

('C', 'A'), ('C', 'D')

# D and E have no outgoing links

])

pagerank\_scores = nx.pagerank(G, alpha=0.85)

print("PageRank Scores:")

for page, score in pagerank\_scores.items():

print(f"{page}: {score:.4f}")

**20] precision , recall , f-measure(20 , 10 , 30)**

# Given values

TP = 20

FP = 10

FN = 30

# Calculate Precision, Recall, F-score

precision = TP / (TP + FP)

recall = TP / (TP + FN)

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

print(f"Precision: {precision:.2f}")

print(f"Recall: {recall:.2f}")

print(f"F-measure: {f\_score:.2f}")

**21]pagerank algorithm ([a-b,c] [b-c,d] [c-a,d] [d-b])**

import networkx as nx

# Create a directed graph

G = nx.DiGraph()

# Add edges based on the given structure

G.add\_edges\_from([

('A', 'B'), ('A', 'C'),

('B', 'C'), ('B', 'D'),

('C', 'A'), ('C', 'D'),

('D', 'B')

])

# Compute PageRank with damping factor 0.85

pagerank\_scores = nx.pagerank(G, alpha=0.85)

# Display the PageRank values

print(“ PageRank Scores:")

for page, score in sorted(pagerank\_scores.items(), key=lambda x: -x[1]):

print(f"{page}: {score:.4f}")

**22] text summarization algorithm**

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

import re

# Input text

text = """

India is a country in South Asia. It is the seventh-largest country by land area,

the second-most populous country, and the most populous democracy in the world.

Bounded by the Indian Ocean on the south, the Arabian Sea on the southwest, and the Bay of Bengal on the southeast,

it shares land borders with Pakistan to the northwest; China, Nepal, and Bhutan to the north; and Bangladesh and Myanmar to the east.

India has a rich cultural heritage. It is known for its diversity in languages, traditions, and festivals.

India has made remarkable progress in science, technology, and space research in the recent decades.

"""

# Split text into sentences using regex

sentences = re.split(r'(?<=[.!?])\s+', text.strip())

# TF-IDF Vectorizer

vectorizer = TfidfVectorizer()

X = vectorizer.fit\_transform(sentences)

# Calculate similarity matrix

similarity\_matrix = cosine\_similarity(X)

# Score each sentence by summing its similarities to other sentences

sentence\_scores = similarity\_matrix.sum(axis=1)

# Get indices of top 3 sentences

top\_n = 3

top\_sentence\_indices = sentence\_scores.argsort()[-top\_n:][::-1]

# Get the top sentences in original order

summary\_sentences = [sentences[i] for i in sorted(top\_sentence\_indices)]

# Print the summary

summary = ' '.join(summary\_sentences)

print("Summary:")

print(summary)

**23] vector space with TF-IDF (python programming)**

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

# Step 1: Define the corpus and query

documents = [

"Document about python programming language and data analysis.",

"Document discussing machine learning algorithms and programming techniques.",

"Overview of natural language processing and its applications."

]

query = "python programming"

# Step 2: Combine documents and query

corpus = documents + [query]

# Step 3: Vectorize using TF-IDF

vectorizer = TfidfVectorizer()

tfidf\_matrix = vectorizer.fit\_transform(corpus)

# Step 4: Calculate cosine similarity (last row is query)

cosine\_sim = cosine\_similarity(tfidf\_matrix[-1], tfidf\_matrix[:-1])

# Step 5: Rank documents

scores = cosine\_sim.flatten()

ranking = scores.argsort()[::-1]

# Step 6: Print results

print("Query:", query)

print("\n Document Ranking based on TF-IDF Cosine Similarity:")

for idx in ranking:

print(f"Document {idx + 1} | Score: {scores[idx]:.4f} | Text: {documents[idx]}")

**24] cosine similarity(gold)**

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

query = "gold silver truck"

document = "shipment of gold damaged in a gold fire"

texts = [query, document]

vectorizer = TfidfVectorizer()

tfidf\_matrix = vectorizer.fit\_transform(texts)

similarity = cosine\_similarity(tfidf\_matrix[0:1], tfidf\_matrix[1:2])[0][0]

print("Cosine Similarity (TF-IDF) between Query and Document:", round(similarity, 4))

**25] k-means algorithms (machine learning)**

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.cluster import KMeans

import pandas as pd

documents = [

"Machine learning is the study of computer algorithms that improve through experience.",

"Deep learning is a subset of machine learning.",

"Natural language processing is a field of artificial intelligence.",

"Computer vision is a field of study that enables computers to interpret the visual world.",

"Reinforcement learning is a machine learning algorithm.",

"Information retrieval is the process of obtaining information from a collection.",

"Text mining is the process of deriving high-quality information from text.",

"Data clustering is the task of dividing a set of objects into groups.",

"Hierarchical clustering builds a tree of clusters.",

"K-means clustering is a method of vector quantization."

]

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

X = vectorizer.fit\_transform(documents)

k = 3 # Define number of clusters

kmeans = KMeans(n\_clusters=k, random\_state=42)

kmeans.fit(X)

labels = kmeans.labels\_

print(f"Documents grouped into {k} clusters:\n")

df = pd.DataFrame({

'Document': documents,

'Cluster': labels

})

print(df)

**26] precision, recall (true , score)**

from sklearn.metrics import precision\_score, recall\_score, f1\_score, average\_precision\_score

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

y\_scores = [0.1, 0.8, 0.6, 0.3, 0.9]

y\_pred = [1 if score >= 0.5 else 0 for score in y\_scores]

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

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

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

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

print(f"Precision: {precision:.4f}")

print(f"Recall: {recall:.4f}")

print(f"F1-score: {f1:.4f}")

print(f"Average Precision: {avg\_precision:.4f}")

**27] inverted index (class starts)**

from collections import defaultdict

document1 = "our class meeting starts soon"

document2 = "my class starts at 6."

documents = [document1, document2]

def build\_inverted\_index(documents):

inverted\_index = defaultdict(list)

for doc\_id, doc in enumerate(documents):

tokens = doc.lower().split()

for token in tokens:

if doc\_id not in inverted\_index[token]:

inverted\_index[token].append(doc\_id)

return inverted\_index

inverted\_index = build\_inverted\_index(documents)

print("Inverted Index:")

for word, doc\_ids in inverted\_index.items():

print(f"{word}: {doc\_ids}")

**28] document retrieval system (class meeting)**

from collections import defaultdict

docs = ["our class meeting starts soon", "my class starts at 6."]

index = defaultdict(list)

for i, doc in enumerate(docs):

for word in doc.lower().split():

index[word].append(i)

query = ["class", "meeting"]

result\_docs = set(index[query[0]]) # Start with docs containing the first query term

for term in query[1:]:

result\_docs &= set(index[term]) # Keep only docs containing all query terms

print("Documents containing the query:", [docs[i] for i in result\_docs] if result\_docs else "No documents found.")

**29] vector space with TF-IDF (solar system)**

from sklearn.feature\_extraction.text import TfidfVectorizer

import numpy as np

documents = [

"The sun is the star at the center of the solar system.",

"She wore a beautiful dress to the party last night.",

"The book on the table caught my attention immediately."

]

query = ["solar system"]

vectorizer = TfidfVectorizer()

tfidf\_matrix = vectorizer.fit\_transform(documents)

query\_tfidf = vectorizer.transform(query)

cosine\_similarities = np.dot(tfidf\_matrix, query\_tfidf.T).toarray().flatten()

print("Cosine Similarities:", cosine\_similarities)

most\_similar\_doc\_index = np.argmax(cosine\_similarities)

print(f"The document most relevant to the query '{query[0]}':")

print(documents[most\_similar\_doc\_index])

**30] clustering [k-means ] (evaluate results)**

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.cluster import KMeans

import matplotlib.pyplot as plt

from sklearn.decomposition import PCA

# Sample documents

documents = [

"Machine learning is the study of computer algorithms that improve through experience.",

"Deep learning is a subset of machine learning.",

"Natural language processing is a field of artificial intelligence.",

"Computer vision is a field of study that enables computers to interpret the visual world.",

"Reinforcement learning is a machine learning algorithm.",

"Information retrieval is the process of obtaining information from a collection.",

"Text mining is the process of deriving high-quality information from text.",

"Data clustering is the task of dividing a set of objects into groups.",

"Hierarchical clustering builds a tree of clusters.",

"K-means clustering is a method of vector quantization."

]

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

X = vectorizer.fit\_transform(documents)

n\_clusters = 3 # Number of clusters

kmeans = KMeans(n\_clusters=n\_clusters, random\_state=42)

kmeans.fit(X)

labels = kmeans.labels\_

for i, doc in enumerate(documents):

print(f"Document {i+1} is in cluster {labels[i]}")

pca = PCA(n\_components=2)

X\_pca = pca.fit\_transform(X.toarray())

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

plt.scatter(X\_pca[:, 0], X\_pca[:, 1], c=labels, cmap='viridis')

plt.title("K-means Clustering of Documents")

plt.xlabel("PCA Component 1")

plt.ylabel("PCA Component 2")

plt.colorbar(label='Cluster Label')

plt.show()

**31] Boolean retrieval (university and Mumbai)**

document1 = "The university exam is scheduled next week."

document2 = "The university of mumbai has declared the result."

import re

def tokenize(document):

return re.findall(r'\b\w+\b', document.lower())

tokens\_doc1 = set(tokenize(document1))

tokens\_doc2 = set(tokenize(document2))

inverted\_index = {}

def build\_inverted\_index(doc\_id, tokens, inverted\_index):

for token in tokens:

if token not in inverted\_index:

inverted\_index[token] = set()

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

build\_inverted\_index('doc1', tokens\_doc1, inverted\_index)

build\_inverted\_index('doc2', tokens\_doc2, inverted\_index)

query = "university and mumbai"

query\_terms = set(query.lower().split(' and '))

def boolean\_retrieval(query\_terms, inverted\_index):

result\_docs = None

for term in query\_terms:

if term in inverted\_index:

if result\_docs is None:

result\_docs = inverted\_index[term]

else:

result\_docs = result\_docs.intersection(inverted\_index[term])

else:

result\_docs = set()

break

return result\_docs

matching\_docs = boolean\_retrieval(query\_terms, inverted\_index)

print("Matching Documents for query '{}':".format(query))

for doc in matching\_docs:

print(doc)

**32] question answering (peacock)**

import re

corpus = [

"India has the second-largest population in the world.",

"It is surrounded by oceans from three sides which are Bay Of Bengal in the east, the Arabian Sea in the west and Indian oceans in the south.",

"Tiger is the national animal of India.",

"Peacock is the national bird of India.",

"Mango is the national fruit of India."

]

# Preprocess the text by removing punctuation and converting to lowercase

def preprocess\_text(text):

text = text.lower() # Convert to lowercase

text = re.sub(r'[^a-zA-Z\s]', '', text) # Remove punctuation

return text

# Preprocess the entire corpus

corpus = [preprocess\_text(doc) for doc in corpus]

# Function to search for the answer based on the query

def search\_answer(query, corpus):

# Preprocess the query and extract keywords

query = preprocess\_text(query)

# Define relevant keywords (since it's a specific question, focus on key terms)

query\_keywords = ["national", "bird", "india"] # These keywords are derived from the query

# Iterate over the corpus to check for documents that contain the query keywords

for doc in corpus:

# Check if the document contains the relevant keywords

if all(keyword in doc for keyword in query\_keywords):

return doc # Return the document containing the answer

return "Answer not found."

query = "Which is the national bird of India?"

answer = search\_answer(query, corpus)

print(f"Answer to the query '{query}':")

print(answer)print(f"Answer to the query '{query}':")

print(answer)

**33]text summarization (NLP data given)**

import numpy as np

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

import networkx as nx

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, natural language generation,

and machine learning.

Text summarization is the process of distilling the most important information from a source

(text) to produce an abridged version for a particular user or task.

Automatic text summarization methods are greatly needed to address the

ever-growing amount of text data available online to both better help

discover relevant information and to consume the vast amount of text data available more efficiently.

"""

def split\_into\_sentences(text):

return text.strip().split('\n')

sentences = split\_into\_sentences(text)

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

X = vectorizer.fit\_transform(sentences)

similarity\_matrix = cosine\_similarity(X)

graph = nx.from\_numpy\_array(similarity\_matrix)

scores = nx.pagerank(graph)

ranked\_sentences = sorted(((score, sentence) for sentence, score in zip(sentences, scores.values())), reverse=True)

top\_n = 3

summary = ' '.join([sentence for score, sentence in ranked\_sentences[:top\_n]])

print("Summary:")

print(summary)