

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

        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 Retrieval 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'[^\w\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)
```