

Code:

from collections import defaultdict

import re

# Sample documents

documents = {

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

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

}

# Preprocessing: tokenize, lowercase, remove punctuation

def preprocess(text):

    text = text.lower()

    text = re.sub(r'[^\w\s]', '', text)  # remove punctuation

    return text.split()

# Build inverted index

def build\_inverted\_index(docs):

    inverted\_index = defaultdict(set)

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

        tokens = preprocess(text)

        for token in tokens:

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

    return dict(inverted\_index)

# Search function

def search(index, query\_terms):

    results = [index.get(term, set()) for term in query\_terms]

    if results:

        return set.intersection(\*results)

    return set()

# Construct index

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

# Query example

query = ["computer", "science"]

result\_docs = search(inverted\_index, query)

print("Inverted Index:")

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

    print(f"{term}: {doc\_ids}")

print("\nSearch Result for 'computer science':", result\_docs)

output:

the: {'doc1'}

computer: {'doc2', 'doc1'}

science: {'doc2', 'doc1'}

students: {'doc1'}

are: {'doc1'}

appearing: {'doc1'}

for: {'doc1'}

practical: {'doc2', 'doc1'}

examination: {'doc2', 'doc1'}

will: {'doc2'}

B] code:

import re

from collections import defaultdict

# Define the documents first

documents = {

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

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

}

# Preprocessing

def preprocess(text):

    text = text.lower()

    text = re.sub(r'[^\w\s]', '', text)

    return text.split()

# Build inverted index

def build\_inverted\_index(docs):

    inverted\_index = defaultdict(set)

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

        tokens = preprocess(text)

        for token in tokens:

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

    return dict(inverted\_index)

# Search documents for terms

def search(index, query\_terms):

    results = [index.get(term, set()) for term in query\_terms]

    if results:

        return set.intersection(\*results)

    return set()

# Question Answering System

def answer\_question(question, docs):

    keywords = preprocess(question)

    best\_match = ""

    max\_overlap = 0

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

        sentences = re.split(r'[.!?]', text)

        for sentence in sentences:

            tokens = preprocess(sentence)

            overlap = len(set(tokens) & set(keywords))

            if overlap > max\_overlap:

                max\_overlap = overlap

                best\_match = sentence.strip()

    return best\_match if best\_match else "No relevant information found."

# Build index and search

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

query = ["computer", "science"]

print("Search Result:", search(inverted\_index, query))

# Answer a question

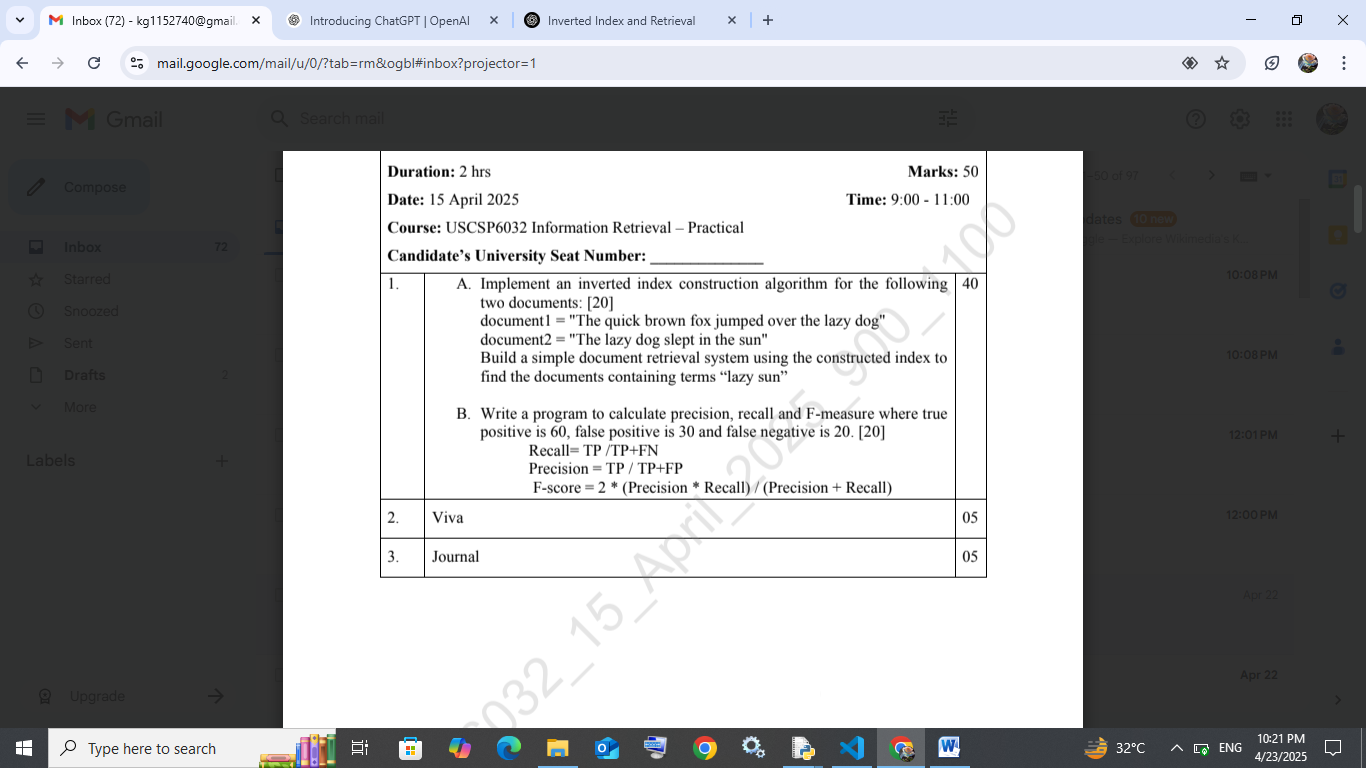
question = "When is the practical examination?"

answer = answer\_question(question, documents)

print("Answer to Question:", answer)

output:

Answer to Question: The computer science students are appearing for practical examination



Code:

from collections import defaultdict

import re

# Step 1: Define the documents

documents = {

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

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

}

# Step 2: Preprocessing - lowercase, remove punctuation, tokenize

def preprocess(text):

    text = text.lower()

    text = re.sub(r'[^\w\s]', '', text)

    return text.split()

# Step 3: Build Inverted Index

def build\_inverted\_index(docs):

    index = defaultdict(set)

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

        for word in preprocess(text):

            index[word].add(doc\_id)

    return dict(index)

# Step 4: Search for documents containing all query terms

def search(index, query\_terms):

    results = [index.get(term, set()) for term in query\_terms]

    return set.intersection(\*results) if results else set()

# Execution

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

query = ["lazy", "sun"]

result\_docs = search(inverted\_index, query)

# Output

print("Inverted Index:")

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

    print(f"{term}: {doc\_ids}")

print("\nDocuments containing 'lazy' and 'sun':", result\_docs)

output:

the: {'doc2', 'doc1'}

quick: {'doc1'}

brown: {'doc1'}

fox: {'doc1'}

jumped: {'doc1'}

over: {'doc1'}

lazy: {'doc2', 'doc1'}

dog: {'doc2', 'doc1'}

slept: {'doc2'}

in: {'doc2'}

sun: {'doc2'}

Documents containing 'lazy' and 'sun': {'doc2'}

B] code:

# Given values

TP = 60

FP = 30

FN = 20

# Calculate metrics

precision = TP / (TP + FP)

recall = TP / (TP + FN)

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

# Output

print(f"\nPrecision: {precision:.2f}")

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

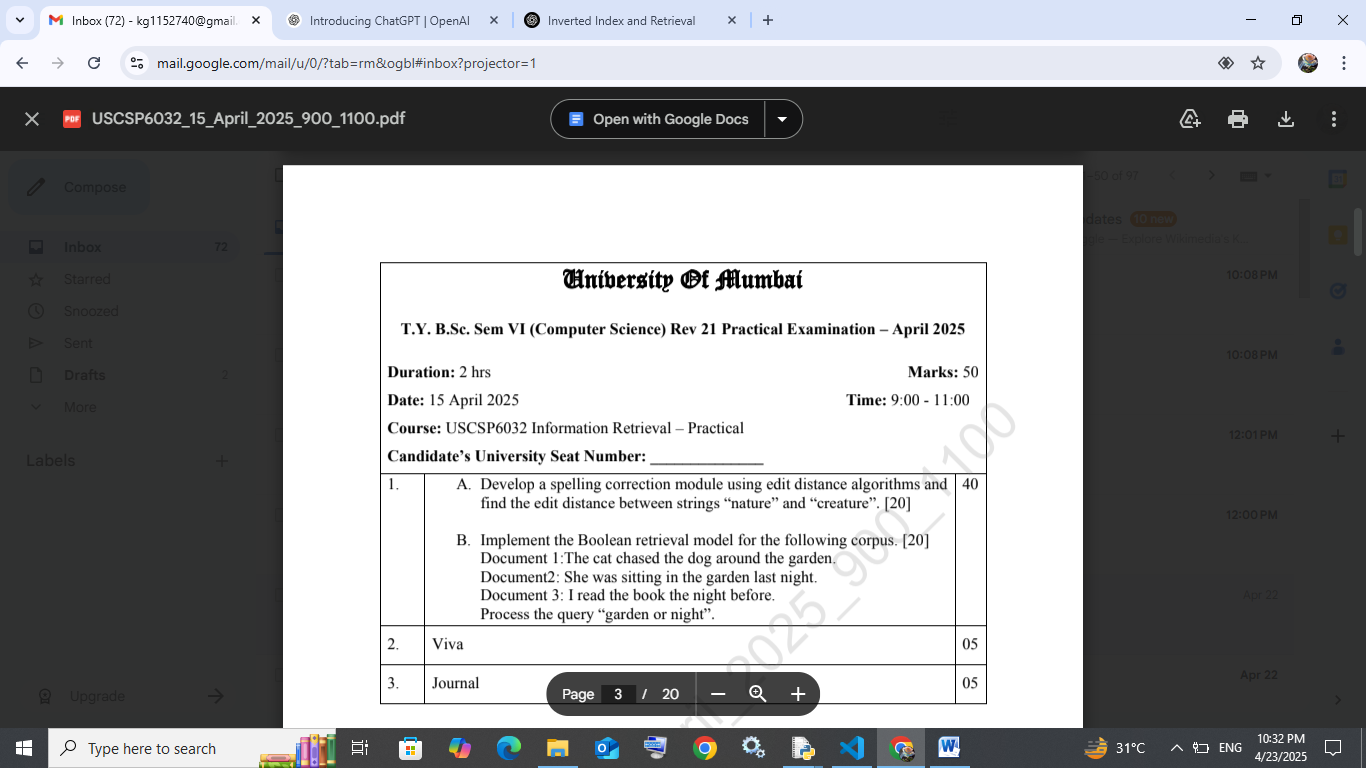
print(f"F-score: {f\_score:.2f}")

output:

Precision: 0.67

Recall: 0.75

F-score: 0.71



Code:

# Levenshtein Distance (Edit Distance)

def edit\_distance(str1, str2):

    n, m = len(str1), len(str2)

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

    for i in range(n+1):

        for j in range(m+1):

            if i == 0:

                dp[i][j] = j  # Insert all characters

            elif j == 0:

                dp[i][j] = i  # Remove all characters

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

                dp[i][j] = dp[i-1][j-1]  # No operation

            else:

                dp[i][j] = 1 + min(

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

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

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

                )

    return dp[n][m]

# Example: Distance between "nature" and "creature"

word1 = "nature"

word2 = "creature"

distance = edit\_distance(word1, word2)

print(f"Edit Distance between '{word1}' and '{word2}' is {distance}")

output:

Edit Distance between 'nature' and 'creature' is 3

B] code:

import re

from collections import defaultdict

# Documents

corpus = {

    "doc1": "The cat chased the dog around the garden.",

    "doc2": "She was sitting in the garden last night.",

    "doc3": "I read the book the night before."

}

# Preprocess text

def preprocess(text):

    text = re.sub(r'[^\w\s]', '', text.lower())

    return text.split()

# Build inverted index

def build\_inverted\_index(docs):

    index = defaultdict(set)

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

        for term in preprocess(text):

            index[term].add(doc\_id)

    return dict(index)

# Boolean Query Evaluation (OR)

def boolean\_or\_query(index, term1, term2):

    return index.get(term1, set()) | index.get(term2, set())

# Execution

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

result = boolean\_or\_query(inverted\_index, "garden", "night")

print("\nInverted Index:")

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

    print(f"{term}: {doc\_ids}")

print("\nDocuments matching query 'garden OR night':", result)

output:

Inverted Index:

the: {'doc3', 'doc2', 'doc1'}

cat: {'doc1'}

chased: {'doc1'}

dog: {'doc1'}

around: {'doc1'}

garden: {'doc2', 'doc1'}

she: {'doc2'}

was: {'doc2'}

sitting: {'doc2'}

in: {'doc2'}

last: {'doc2'}

night: {'doc3', 'doc2'}

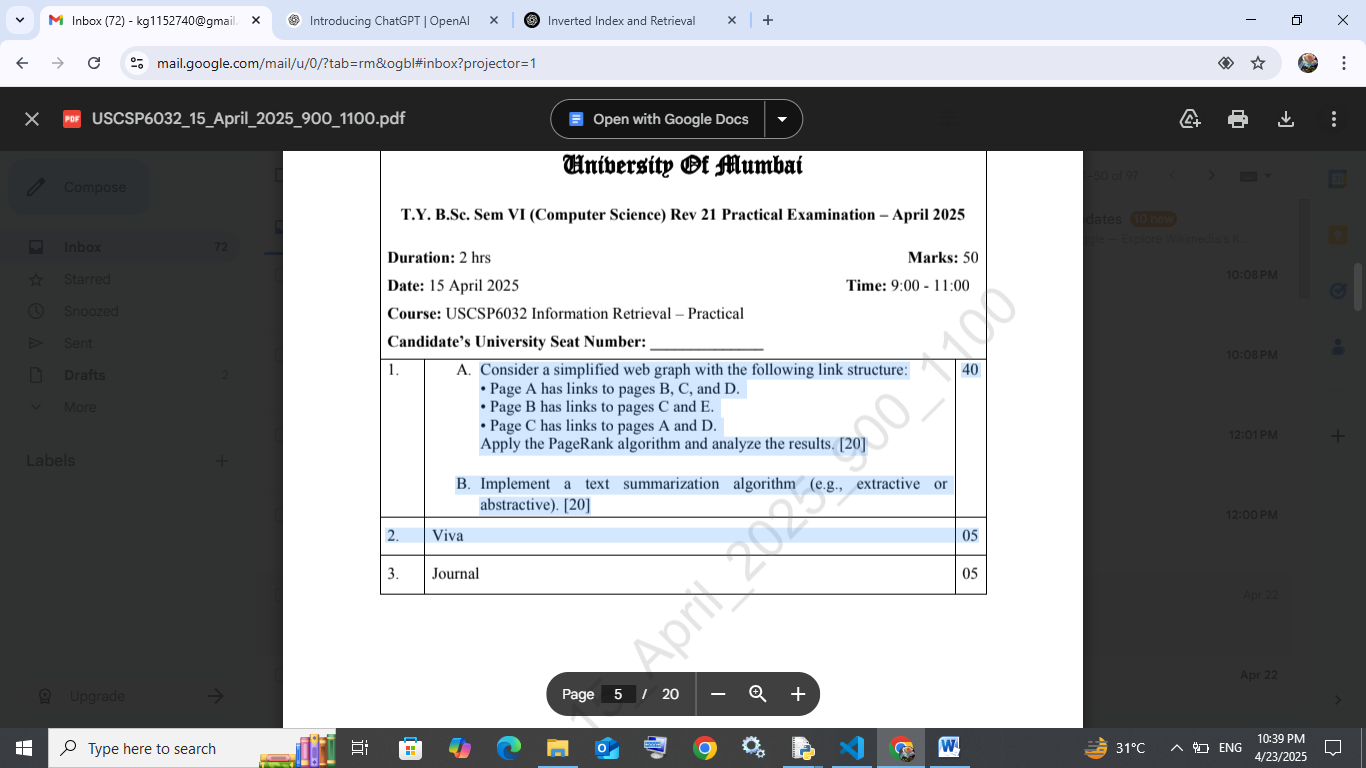
i: {'doc3'}

read: {'doc3'}

book: {'doc3'}

before: {'doc3'}

Documents matching query 'garden OR night': {'doc3', 'doc2', 'doc1'}



Code:

def pagerank(graph, damping=0.85, num\_iterations=100):

    pages = list(graph.keys())

    N = len(pages)

    ranks = {page: 1 / N for page in pages}

    for \_ in range(num\_iterations):

        new\_ranks = {}

        for page in pages:

            incoming = [p for p in pages if page in graph[p]]

            rank\_sum = 0

            for p in incoming:

                out\_links = len(graph[p]) if graph[p] else N  # handle sink nodes

                rank\_sum += ranks[p] / out\_links

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

        ranks = new\_ranks

    return ranks

# Define the graph

graph = {

    "A": ["B", "C", "D"],

    "B": ["C", "E"],

    "C": ["A", "D"],

    "D": [],  # sink node

    "E": []   # sink node

}

ranks = pagerank(graph)

print("PageRank Results:")

for page, rank in sorted(ranks.items()):

    print(f"{page}: {rank:.4f}")

output:

PageRank Results:

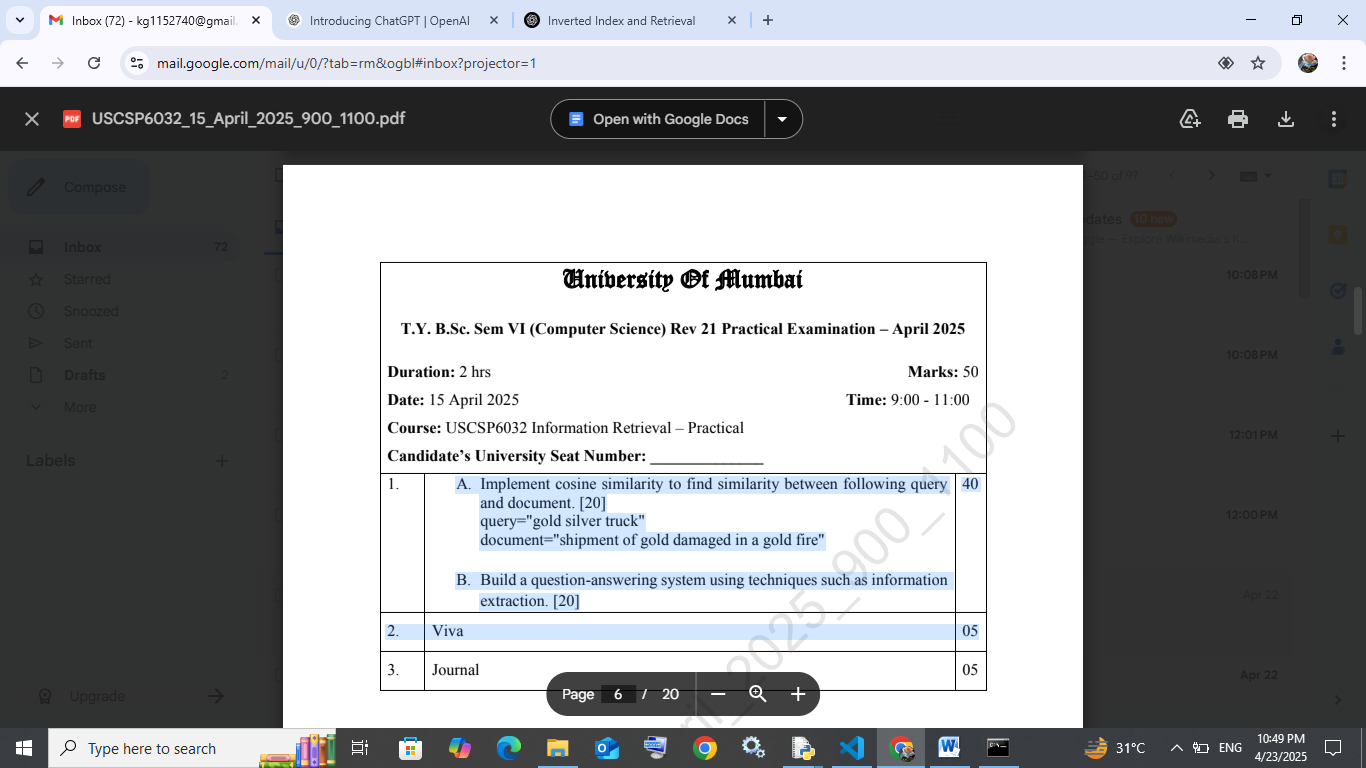
A: 0.0581

B: 0.0465

C: 0.0662

D: 0.0746

E: 0.0498



Code:

import math

from collections import Counter

def cosine\_similarity(text1, text2):

    # Tokenize

    words1 = text1.lower().split()

    words2 = text2.lower().split()

    # Create vocabulary

    vocab = list(set(words1 + words2))

    # Term frequency

    tf1 = Counter(words1)

    tf2 = Counter(words2)

    # Convert to vector

    vec1 = [tf1[word] for word in vocab]

    vec2 = [tf2[word] for word in vocab]

    # Dot product

    dot\_prod = sum(a \* b for a, b in zip(vec1, vec2))

    # Magnitudes

    mag1 = math.sqrt(sum(a \* a for a in vec1))

    mag2 = math.sqrt(sum(b \* b for b in vec2))

    # Cosine similarity

    if mag1 == 0 or mag2 == 0:

        return 0.0

    return dot\_prod / (mag1 \* mag2)

# Test

query = "gold silver truck"

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

similarity = cosine\_similarity(query, document)

print(f"Cosine Similarity: {similarity:.4f}")

output:

Cosine Similarity: 0.3651