WEEK-1

Representation of a Text Document in Vector Space Model and Computing Similarity between two documents.

AIM:

To compute the similarity between two text documents using **TF-IDF vectorization** and **Cosine Similarity** in Python.

DESCRIPTION:

Cosine Similarity – Definition:

Cosine Similarity is a metric used to measure the similarity between two non-zero vectors by calculating the **cosine of the angle** between them.

In the context of text documents, it compares two documents represented as TF-IDF vectors and returns a value between **0** and **1**:

- **1** indicates identical documents (high similarity).
- **0** indicates completely different documents.

It is widely used in **text mining**, **information retrieval**, and **natural language processing** to compare the semantic similarity between documents.

Vector Space Model – Definition:

The Vector Space Model is a mathematical model used to represent text documents as vectors in a multi-dimensional space.

Each unique term in the entire corpus forms a separate dimension. The values in the vector are typically **term weights**, such as **TF-IDF scores**, which indicate the importance of a word in a document.

In this model:

- Documents are treated as vectors.
- Similarity can be computed using mathematical techniques like **cosine similarity**.

This model enables efficient **document comparison**, **ranking**, and **clustering** in text-based applications.

This experiment demonstrates how to measure the similarity between two documents using Natural Language Processing techniques.

The program uses **TF-IDF (Term Frequency–Inverse Document Frequency)** to convert textual data into numerical form, where each word is assigned a weight based on its importance in the document and across documents.

Once the documents are represented as TF-IDF vectors, the **cosine similarity** between them is calculated. Cosine similarity is a metric that quantifies the cosine of the angle between two vectors in a multi-dimensional space. A similarity score close to 1 indicates high similarity, while a score near 0 indicates low similarity.

Cosine Similarity – Formula:

$$\text{Cosine Similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$

Where:

- A and B are two vectors (e.g., TF-IDF vectors of two documents).
- $A \cdot B$ is the **dot product** of the vectors.
- $\|A\|$ and $\|B\|$ are the **magnitudes (lengths)** of vectors A and B.

Step-by-Step Process to Calculate Cosine Similarity:

1. **Convert documents into vectors**
   Use a technique like **TF-IDF** to convert each document into a numeric vector.
2. **Compute the dot product $A \cdot B$**
   Multiply corresponding values of the two vectors and sum them.
3. **Compute the magnitude of each vector**

$$\|A\| = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}$$

$$\|B\| = \sqrt{b_1^2 + b_2^2 + \cdots + b_n^2}$$

4. **Apply the cosine similarity formula**

$$\text{Cosine Similarity} = \frac{A \cdot B}{\|A\| \times \|B\|}$$

Interpretation of Result:

- **1** → Documents are exactly the same in direction (very similar).
- **0** → Documents are completely different (orthogonal vectors).
- **Closer to 1** → More similar.

PROGRAM:

```
doc1 = "Machine learning is amazing and fun"
doc2 = "Deep learning and machine learning are parts of artificial intelligence" documents = [doc1, doc2]
from sklearn.feature_extraction.text import TfidfVectorizer from sklearn.metrics.pairwise import cosine_similarity
# Create TF-IDF vectorizer vectorizer = TfidfVectorizer()
# Fit and transform the documents
tfidf_matrix = vectorizer.fit_transform(documents)
# Display the feature names (terms)
print("Vocabulary:", vectorizer.get_feature_names_out())
# Display TF-IDF matrix
print("TF-IDF Matrix:\n", tfidf_matrix.toarray())
# Compute cosine similarity between the two documents similarity =
cosine_similarity(tfidf_matrix[0:1], tfidf_matrix[1:2])
print("Cosine Similarity between doc1 and doc2:", similarity[0][0])
```

OUTPUT:

```
Vocabulary: ['amazing' 'and' 'are' 'artificial' 'deep' 'fun' 'intelligence' 'is'
 'learning' 'machine' 'of' 'parts']
TF-IDF Matrix:
[[0.447 0.447 0.    0.    0.    0.447 0.    0.447 0.447 0.447 0.    0.   ]
 [0.    0.265 0.265 0.265 0.265 0.    0.265 0.    0.529 0.529 0.265 0.265]]
Cosine Similarity between doc1 and doc2: 0.487
```

WEEK-2

Pre-processing of a Text Document: stop word removal and stemming.

AIM:

To preprocess a text document by performing stop word removal and stemming

DESCRIPTION:

Understanding Stop Words

Stop words in Natural Language Processing (NLP) refer to the most common words in a language. Examples include "and", "the", "is", and others that do not provide significant meaning and are often removed to speed up processing without losing crucial information. For this purpose, Python's Natural Language Tool Kit (NLTK) provides a pre-defined list of stop words.

Introduction to Stemming

Stemming is a technique that reduces a word to its root form. Although the stemmed word may not always be a real or grammatically correct word in English, it does help to consolidate different forms of the same word to a common base form, reducing the complexity of text data. This simplification leads to quicker computation and potentially better performance when implementing Natural Language Processing (NLP) algorithms, as there are fewer unique words to consider.

For example, the words "run", "runs", "running" might all be stemmed to the common root "run". This helps our algorithm understand that these words are related and they carry a similar semantic meaning.

PROGRAM:

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize from
nltk.stem import PorterStemmer nltk.download('punkt')
nltk.download('stopwords')
```

```
def preprocess_text(text):
    text = text.lower()
    words = word_tokenize(text)

    stop_words = set(stopwords.words('english'))

    filtered_words = [word for word in words if word.isalnum() and word not in stop_words]

    stemmer = PorterStemmer()
    stemmed_words = [stemmer.stem(word) for word in filtered_words]
    return stemmed_words
text = "Machine learning algorithms are revolutionizing the world of artificial intelligence."
print("Orginal Text:",text)
processed = preprocess_text(text)
processed_text = ' '.join(processed)
print("Processed Text:", processed_text)
print("Preprocessed Words:", processed)
```

OUTPUT:

Orginal Text: Machine learning algorithms are revolutionizing the world of artificial intelligence.

Processed Text: machin learn algorithm revolution world artifici intellig

Preprocessed Words: ['machin', 'learn', 'algorithm', 'revolution', 'world', 'artifici', 'intellig']

WEEK-14

Construction of an Inverted Index for a given document collection comprising of at least 50 documents with a total vocabulary size of at least 1000 words.

AIM:

To construct an inverted index for a collection of 50 documents and vocabulary of 1000 words.

DESCRIPTION:

An **inverted index** is a fundamental data structure in the field of information retrieval. It is designed to enable fast and efficient full-text searches over a large collection of documents. Instead of storing documents sequentially as they are, an inverted index reverses the natural relationship between documents and terms by mapping each **unique word (term)** in the corpus to a list of **documents** in which it appears. This allows for quick lookup of all documents containing a given word. In constructing an inverted index, the document collection must undergo several preprocessing steps to normalize and clean the data. These steps typically include

- converting all text to lowercase, removing punctuation and stopwords (common, non-informative words like “the,” “is,” “and”),
- tokenizing the text into words,
- applying stemming or lemmatization to reduce words to their root forms (e.g., “running” becomes “run”).

PYTHON CODE:

```
import os import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word_tokenize

from nltk.stem import PorterStemmer

from collections import defaultdict import json

nltk.download('punkt')

 nltk.download('stopwords')
```

```
def preprocess(text):
    text = text.lower()
    tokens = word_tokenize(text)
    stop_words = set(stopwords.words('english')) stemmer = PorterStemmer()
    words = [stemmer.stem(word) for word in tokens if word.isalnum() and word not in
    stop_words]
    return words documents = {}
for filename in os.listdir():
    if filename.endswith(".txt"):
        with open(filename, 'r', encoding='utf-8', errors='ignore') as f:
            text = f.read()
            documents[filename] = preprocess(text)
print(f"Total documents loaded: {len(documents)}")
inverted_index = defaultdict(set)
for doc_id, words in documents.items():
    for word in set(words): # avoid duplicates per document
        inverted_index[word].add(doc_id)
vocab_size = len(inverted_index)
print(f"\nVocabulary Size: {vocab_size} words")
print("\nSample inverted index terms:")
for term in list(inverted_index)[:10]:
    print(f"{term}: {sorted(inverted_index[term])}")
```

OUTPUT:

Total documents loaded: 11

defaultdict(<class 'set'>, {'today': {'HI.txt'}, 'work': {'HI.txt'}, 'warm': {'untitled4.txt', 'HI.txt'}, 'hi': {'HI.txt'}, 'professor': {'HI.txt'}, 'aditya': {'HI.txt'}, 'sushuma': {'HI.txt'}, 'assist': {'HI.txt'},

'sunni': {'untitled4.txt'}})

Vocabulary Size: 9 words

Sample inverted index terms:

hi: ['HI.txt']

sushuma: ['HI.txt']

today: ['HI.txt']

aditya: ['HI.txt']

work: ['HI.txt']

professor: ['HI.txt']

assist: ['HI.txt']

warm: ['HI.txt', 'untitled4.txt']

sunni: ['untitled4.txt']

WEEK-4

Classification of a set of Text Documents into known classes (You may use any of the Classification algorithms like Naive Bayes, Max Entropy, Rochio‟s, Support Vector Machine). Standard Datasets will have to be used to show the results.

AIM:

To perform classification of text documents into known classes.

DESCRIPTION:

Introduction to Text Classification

Text classification is a fundamental task in Natural Language Processing (NLP) where a piece of text (e.g., a document, sentence, or paragraph) is assigned to one or more predefined categories. Applications include:

- Spam detection
- Sentiment analysis
- Topic labelling
- News categorization

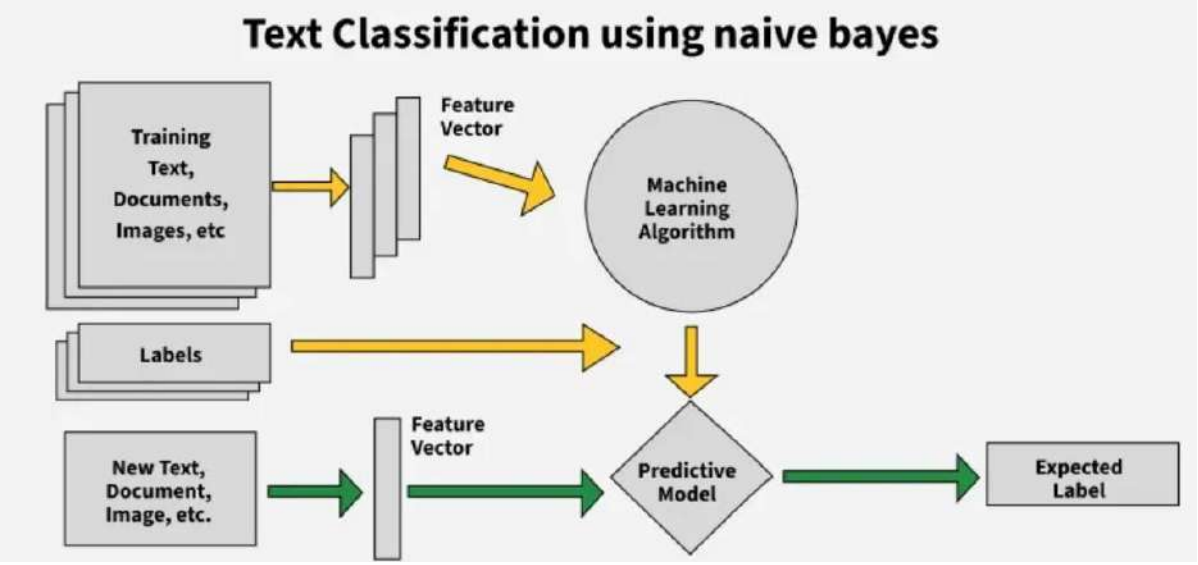
Naive Bayes

Naive Bayes is a classification algorithm that uses probability to predict which category a data point belongs to, assuming that all features are unrelated. It is named as "Naive" because it assumes the presence of one feature does not affect other features.

Why Naive Bayes?

The Naive Bayes classifier is a popular probabilistic machine learning algorithm based on Bayes' Theorem with an assumption of independence among features. It is particularly well- suited for text classification due to:

- **Efficiency** in handling high-dimensional data (like text)
- **Robustness** even with small training datasets
- **Scalability** and speed of training and prediction



PROGRAM:

```
from sklearn.datasets import fetch_20newsgroups

from sklearn.model_selection import train_test_split

from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.naive_bayes import MultinomialNB

from sklearn.metrics import accuracy_score, classification_report

categories = ['sci.space', 'rec.sport.hockey', 'comp.graphics', 'alt.atheism']

newsgroups= fetch_20newsgroups(subset='all',categories=categories,shuffle=True, random_state=42)

print(f"Total documents: {len(newsgroups.data)}")

print(f"Target classes: {newsgroups.target_names}")

X_train, X_test, y_train, y_test = train_test_split (newsgroups.data, newsgroups.target, test_size=0.2, random_state=42 )

vectorizer = TfidfVectorizer(stop_words='english')

X_train_tfidf = vectorizer.fit_transform(X_train)

X_test_tfidf = vectorizer.transform(X_test)

nb = MultinomialNB()
```

```
nb.fit(X_train_tfidf, y_train)
y_pred = nb.predict(X_test_tfidf)
print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n")
print(classification_report(y_test, y_pred, target_names=newsgroups.target_names))
for i in range(5):
    print("\nText:\n", X_test[i])
    print("Actual:", newsgroups.target_names[y_test[i]])
    print("Predicted:", newsgroups.target_names[y_pred[i]])
```

OUTPUT:

Accuracy: 0.9840425531914894

Classification Report:

	precision	recall	f1-score	support
alt.atheism	1.00	1.00	1.00	152
comp.graphics	0.96	0.99	0.97	196
rec.sport.hockey	0.99	1.00	1.00	194
sci.space	0.99	0.95	0.97	210
accuracy			0.98	752
macro avg	0.99	0.99	0.99	752
weighted avg	0.98	0.98	0.98	752

WEEK-5

Text Document Clustering using K-means. Demonstrate with a standard dataset and compute performance measures- Purity, Precision, Recall and F-measure.

AIM:

To cluster a document using K-Means and compute the performance measures.

DESCRIPTION:

Text Clustering

Text document clustering is an **unsupervised machine learning** technique used to **group similar documents** into clusters based on their content. Unlike classification, clustering doesn't use labeled data. It's widely used in:

- Document organization
- Topic discovery
- Information retrieval

K-Means Clustering Algorithm

K-means clustering is a type of unsupervised learning method, which is used when we don't have labelled data as in our case, we have unlabelled data (means, without defined categories or groups). The goal of this algorithm is to find groups in the data, whereas the no. of groups is represented by the variable K. The data have been clustered on the basis of high similarity points together and low similarity points in the separate clusters.

Algorithm Steps:

1. **Preprocessing**: Convert raw documents into a numerical format (e.g., TF-IDF vectors).
2. **Initialization**: Choose k initial centroids randomly.
3. **Assignment Step**: Assign each document to the nearest centroid (based on Euclidean or Cosine distance).
4. **Update Step**: Recompute centroids based on the assigned documents.

5. **Repeat** the assignment and update steps until convergence (no change in centroids or assignments).

3. Text Representation using TF-IDF

Before clustering, text documents are converted into vector form using:

- **Bag of Words (BoW)** or
- **TF-IDF (Term Frequency-Inverse Document Frequency)**

TF-IDF reduces the impact of commonly used words and emphasizes more informative terms.

Performance Evaluation Metrics

In clustering (especially with known ground truth), we compare the clustering results with actual labels using the following metrics:

1. Purity

Purity measures the extent to which clusters contain documents from primarily one class.

Formula:

$$\text{Purity} = \frac{1}{N} \sum_{k} \max_{j} |C_k \cap L_j|$$

- Ck: Cluster k
- Lj : Class j
- N: Total number of documents

A higher purity means better clustering. Maximum value = 1.

2. Precision (per class)

Precision measures how many documents assigned to a cluster are actually relevant (i.e., from the correct class).

Formula:

$$\text{Precision} = \frac{TP}{TP + FP}$$

TP: True Positives

FP: False Positives

3. Recall (per class)

Recall measures how many relevant documents (from a true class) are correctly assigned to a cluster.

Formula:

$$\text{Recall} = \frac{TP}{TP + FN}$$

TP: True Positives

FN: False Negatives

4. F-measure (F1 Score)

F-measure is the harmonic mean of precision and recall. It balances both metrics.

Formula:

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

PROGRAM:

```
import numpy as np
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans
from sklearn.metrics import precision_score, recall_score, f1_score
from scipy.stats import mode
```

 # Step 2: Load Dataset (using 4 categories for clarity)

```
categories = ['alt.atheism', 'comp.graphics', 'sci.space', 'talk.religion.misc']

newsgroups = fetch_20newsgroups(subset='all', categories=categories, remove=('headers', 'footers', 'quotes'))
```

 # Step 3: Convert Text to TF-IDF Features

```
vectorizer = TfidfVectorizer(stop_words='english', max_features=1000)

X = vectorizer.fit_transform(newsgroups.data)

y_true = newsgroups.target
```

 # Step 4: Apply K-Means Clustering

```
k = len(categories)

kmeans = KMeans(n_clusters=k, random_state=42)

y_pred = kmeans.fit_predict(X)


def purity_score(y_true, y_pred):
    clusters = np.unique(y_pred)
    classes = np.unique(y_true)
    contingency_matrix = np.zeros((len(clusters), len(classes)))
    for i, cluster in enumerate(clusters):
        indices = np.where(y_pred == cluster)[0]
        true_labels = y_true[indices]
        if len(true_labels) == 0:
            continue
        most_common = mode(true_labels, keepdims=True).mode[0]
        count = np.sum(true_labels == most_common)
        j = np.where(classes == most_common)[0][0]
```

```
            contingency_matrix[i][j] = count
return np.sum(np.max(contingency_matrix, axis=1)) / np.sum(contingency_matrix)
# Create a label mapping from cluster to majority class
def map_clusters_to_labels(y_true, y_pred):
    label_mapping = {}
    for cluster in np.unique(y_pred):
        indices = np.where(y_pred == cluster)[0]
        if len(indices) == 0:
            continue
        majority_label = mode(y_true[indices], keepdims=True).mode[0]
        label_mapping[cluster] = majority_label
    # Map each prediction to the true label
    mapped_preds = np.array([label_mapping[cluster] for cluster in y_pred])
    return mapped_preds
y_pred_mapped = map_clusters_to_labels(y_true, y_pred)
# Compute Metrics
purity = purity_score(y_true, y_pred)
precision = precision_score(y_true, y_pred_mapped, average='macro') recall =
recall_score(y_true, y_pred_mapped, average='macro')
f1 = f1_score(y_true, y_pred_mapped, average='macro')
# Print Results
print("Purity Score:", round(purity, 4))
print("Precision:", round(precision, 4))
print("Recall:", round(recall, 4))
print("F1-Score:", round(f1, 4))
```

OUTPUT:

Purity Score: 1.0

Precision: 0.6767

Recall: 0.5225

F1-Score: 0.5253

WEEK-6

Crawling/ Searching the Web to collect news stories on a specific topic (based on user input). The program should have an option to limit the crawling to certain selected websites only.

AIM:

To Crawl/ Search the Web to collect news stories on a specific topic.

DESCRIPTION:

Crawling

Crawling is the process by which a program (called a **crawler** or **spider**) automatically browses the web to collect and download information from websites.

Introduction

Web crawling is the automated process of systematically browsing and extracting information from web pages. In the context of news story collection, a crawler is programmed to search the internet for news articles related to a **specific topic** provided by the user. The goal is to collect relevant content from the web and present it in a usable format.

How It Works

1. **User Input**
    - The user specifies a **topic** (e.g., "Climate Change", "Artificial Intelligence", "Olympics 2024").
    - Optionally, the user can **limit the search to specific websites** (e.g., *bbc.com*, *nytimes.com*).
2. **URL Queue Initialization**
    - The crawler starts with a set of **seed URLs** (either from the user or pre-defined news portals).
3. **Fetching Pages**
    - The crawler downloads the HTML content of these web pages.

4. **Parsing & Extraction**
    - The HTML is analyzed to extract:
        - Headlines
        - Publication dates
        - Article content
        - Author names
    - Relevant **links to other news pages** are also extracted.
5. **Filtering by Topic**
    - Using **keyword matching** or **Natural Language Processing (NLP)**, the crawler selects only those articles related to the given topic.
6. **Storing Results**
    - The collected news stories are saved in a database or file for further reading, analysis, or summarization.

Install package: pip install serpapi beautifulsoup4 requests About package:

- **serpapi**: A Python wrapper for SerpAPI, which lets you retrieve Google/Bing/News results through their API (requires a SerpAPI key).
- **beautifulsoup4**: A Python library for parsing HTML and XML documents; commonly used to extract data from web pages.
- **requests**: A popular library to send HTTP requests easily in Python.

PROGRAM:

```
pip install serpapi beautifulsoup4 requests

import requests

from bs4 import BeautifulSoup

topic = input("Enter the news topic to search for: ")
```

```
websites = input("Enter comma-separated websites to limit crawling (e.g., bbc.com,cnn.com):
        ").split(',')
SERP_API_KEY = '8d6bc2b3eef2e66c277a5a34be29b70d490834e929934539b15ae91c71dd569c'
search_url = 'https://serpapi.com/search.json'
def search_news(topic, websites):
    all_results = []
    for site in websites:
        params = {
            "engine": "google",
            "q": f"{topic} site:{site.strip()}",
            "api_key": SERP_API_KEY
        }
        response = requests.get(search_url, params=params)
        data = response.json()
        if "organic_results" in data:
            for result in data["organic_results"]:
                title = result.get("title")
                link = result.get("link")
                snippet = result.get("snippet", "")
                all_results.append((title, link, snippet))
    return all_results
def display_results(results):
    for idx, (title, link, snippet) in enumerate(results, start=1):
        print(f"\nNews {idx}:")
        print(f"Title  : {title}")
```

```
            print(f"URL     : {link}")
            print(f"Summary : {snippet}")
results = search_news(topic, websites)
if results:
    display_results(results)
else:
    print("No results found.")
```

OUTPUT:

Enter the news topic to search for: AI in healthcare

Enter comma-separated websites to limit crawling (e.g., bbc.com,cnn.com): bbc.com,cnn.com

News 1:

Title : AI in healthcare: what are the risks for the NHS?

URL : https://www.bbc.com/news/articles/c6233x9k4dlo

Summary : Generative AI will be transformative for NHS patient outcomes, a senior government advisor says.

News 2:

Title : How AI can spot diseases that doctors aren't looking for

URL : https://www.bbc.com/news/articles/c9q7zqy1xlpo

Summary : AI can take a second look at medical scans and flag up potential problems that doctors might not see.

News 3:

Title : How AI Has Transformed Healthcare's Future

URL : https://www.bbc.com/storyworks/hpe-greenlake/how-ai-has-transformed-healthcares-future

Summary : AI can link seemingly unrelated information to reveal new research pathways that yield better results. For example, AI models have identified potential ...

News 4:

Title : Hospitals will use AI to speed up patient care

URL : https://www.bbc.com/news/articles/cye0yywdegdo

Summary : Hospitals across the region are to use artificial intelligence (AI) technology to reduce unnecessary admissions and lengthy stays, ...

News 5:

Title : Can AI help modernise Ireland's healthcare system?

URL : https://www.bbc.com/news/articles/cly7yxm3py5o

Summary : Ireland is investing billions of euros to revamp its healthcare service - will AI help?

News 6:

Title : How artificial intelligence is matching drugs to patients

URL : https://www.bbc.com/news/business-65260592

Summary : Health-tech firms around the world are increasingly using AI to help tailor drugs for patients.

WEEK-7

To parse XML text, generate Web graph and compute topic specific page rank Parse XML Text

AIM:

To parse XML text, generate Web graph and compute topic specific page rank .

DESCRIPTION:

Parsing XML means reading the XML data, understanding its structure, and extracting the relevant information from it.

- Purpose: Often, XML files store web crawl data — URLs, hyperlinks, anchor texts, and metadata.
- Process:
    1. Load XML file into memory.
    2. Use an XML parser (like Python's xml.etree.ElementTree or lxml) to navigate the tree structure.
    3. Extract required fields — e.g., <page>, <link>, <title>, <content>.
    4. Store these in data structures like dictionaries or adjacency lists for further graph processing.

Generate Web Graph

The web graph is a directed graph where:

- Nodes (vertices) = web pages
- Edges = hyperlinks from one page to another
- Construction:
    1. Create a mapping from URL → node ID.
    2. For each <page> in the XML, list its outgoing <link>s.
    3. Build an adjacency list or adjacency matrix.
- Purpose: This graph becomes the foundation for PageRank calculations.

Compute Topic-Specific PageRank

Topic-specific PageRank modifies the standard PageRank by biasing the random surfer towards a set of topic-related pages.

- Idea: Instead of teleporting uniformly to any page, the algorithm teleports more often to pages relevant to a given topic (e.g., "sports" or "technology").

- Steps:
    1. Identify a topic seed set — pages known to be about the topic.
    2. Initialize teleportation vector v where:
        - Higher values for topic-relevant pages
        - Lower (or zero) for irrelevant pages
    3. Use the PageRank equation:

        $$PR=\alpha MPR+(1-\alpha)v$$

        - M: transition matrix from the web graph
        - α (alpha): damping factor (usually 0.85)
        - v: topic-biased teleportation vector
    4. Iterate until convergence (small changes between iterations).

PROGRAM:

```
import xml.etree.ElementTree as ET

import numpy as np

import networkx as nx

import matplotlib.pyplot as plt

# Step 1: Parse XML

def parse_xml(xml_text):

    tree = ET.fromstring(xml_text)

    graph = {}
```

```
    topics_map = {}
    for page in tree.findall('page'):
        title = page.find('title').text.strip()
        links = [link.text.strip() for link in page.findall('link')]
        topics = page.find('topics').text.strip().split(",") if page.find('topics') is not
                None else []
        graph[title] = links
        topics_map[title] = [t.strip() for t in topics]
    return graph, topics_map
```

 # Step 2: Build Adjacency Matrix

```
def build_adj_matrix(graph):
    pages = list(graph.keys())
    idx = {page: i for i, page in enumerate(pages)}
    n = len(pages)
    M = np.zeros((n, n))
    for page, links in graph.items():
        if links:
            for link in links:
                if link in idx:
                    M[idx[link]][idx[page]] = 1 / len(links)
        else:
            M[:, idx[page]] = 1 / n # dangling node handling
    return M, pages
```

 # Step 3: Compute Topic-Specific PageRank

```
def topic_specific_pagerank(M, pages, topics_map, topic, d=0.85, tol=1e-6, max_iter=100):
```

```
    n = len(pages)
    teleport = np.array([1.0 if topic in topics_map[p] else 0.0 for p in pages])
    if teleport.sum() == 0:
        teleport = np.ones(n)
    teleport = teleport / teleport.sum() # normalize
    r = np.ones(n) / n # initial rank
    for i in range(max_iter):
        r_new = d * M @ r + (1 - d) * teleport
        if np.linalg.norm(r_new - r, 1) < tol:
            break
        r = r_new
    return dict(zip(pages, r))
```

 # Step 4: Visualize the Web Graph with Topic Highlight

```
def draw_web_graph(graph, topics_map, topic):
    G = nx.DiGraph()
    for page, links in graph.items():
        for link in links:
            G.add_edge(page, link)
```

 # Node colors: highlight pages having the topic

```
node_colors = []
for page in G.nodes():
    if topic in topics_map.get(page, []):
        node_colors.append("lightgreen") # highlight topic pages
    else:
```

```
                node_colors.append("skyblue")          # normal pages
plt.figure(figsize=(6, 4))
pos = nx.spring_layout(G, seed=42)
nx.draw(G, pos, with_labels=True, node_color=node_colors, node_size=1500,
                        font_size=10, arrowsize=15, edge_color="gray")
plt.title(f"Web Graph (Highlighted Topic: {topic})")
plt.show()
```

 # Step 5: Input and Execute

```
xml_text = '''<web>
        <page>
                <title>PageA</title>
                <link>PageB</link>
                <link>PageC</link>
                <topics>science,education</topics>
        </page>
        <page>
                <title>PageB</title>
                <link>PageC</link>
                <topics>science</topics>
        </page>
        <page>
                <title>PageC</title>
                        <topics>sports</topics>
        </page>
</web>'''
```

graph, topics_map = parse_xml(xml_text)

M, pages = build_adj_matrix(graph)

 # Draw the web graph with topic highlighting

topic = "science"

draw_web_graph(graph, topics_map, topic)

 # Compute topic-specific PageRank

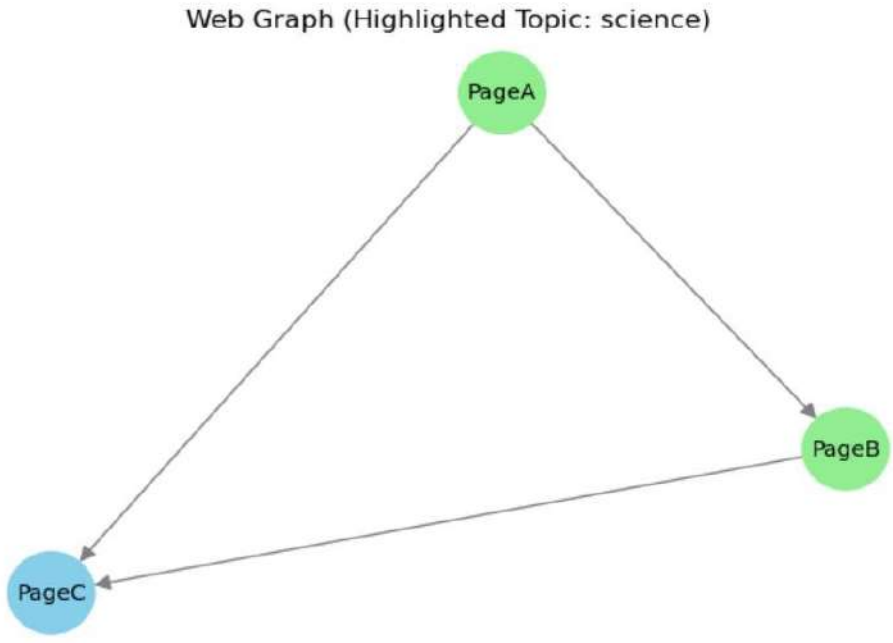
ranks = topic_specific_pagerank(M, pages, topics_map, topic)

print("\nTopic-Specific PageRank (Topic: science):")

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

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

OUTPUT:




```
Topic-Specific PageRank (Topic: science):
PageC: 0.4849
PageB: 0.3027
PageA: 0.2124
```

WEEK-8

Implement Matrix Decomposition and LSI for a standard dataset.

AIM:

To implement matrix decomposition and LSI for a standard dataset.

DESCRIPTION:

Matrix Decomposition

Matrix decomposition (or matrix factorization) is a mathematical technique used to break down a large matrix into smaller, simpler matrices.

- In text mining, we typically start with a document-term matrix (DTM) or TF-IDF matrix where:
    - Rows = documents
    - Columns = terms (words)
    - Values = frequency or importance of terms in documents

Latent Semantic Indexing (LSI)

Latent Semantic Indexing (also called Latent Semantic Analysis, LSA) is a technique in information retrieval and natural language processing that uses SVD on the term-document matrix.

Idea behind LSI

- Natural language has synonyms (different words with similar meaning) and polysemy (same word with multiple meanings).
- A raw term-document matrix treats each word independently, which misses semantic relationships.
- LSI reduces the matrix dimensions to capture hidden (latent) semantic structures in text.

Process of LSI

1. Construct a TF-IDF matrix from the dataset.
2. Apply Truncated SVD to decompose into lower-rank matrices.
3. Represent documents and terms in this reduced semantic space.
4. Use cosine similarity or other metrics to find similarity between documents/queries**. Singular Value Decomposition (SVD)**

- SVD is a mathematical technique that breaks a large matrix into three smaller matrices.
- For a document-term matrix (like TF-IDF), SVD helps find patterns/relationships between terms and documents.
- It identifies the most important concepts (latent topics) in the data

PROGRAM:

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
# Step 1: Load dataset (subset for speed)
categories = ['sci.space', 'rec.sport.hockey', 'comp.graphics']
newsgroups = fetch_20newsgroups(subset='train', categories=categories, remove=('headers',
                'footers', 'quotes'))
# Step 2: TF-IDF Vectorization
vectorizer = TfidfVectorizer(stop_words='english', max_features=1000)
X_tfidf = vectorizer.fit_transform(newsgroups.data)
print(f"Original TF-IDF shape: {X_tfidf.shape}") # (docs x terms)
# Step 3: SVD for LSI (Latent Semantic Indexing)
k = 100 # number of latent dimensions
svd = TruncatedSVD(n_components=k)
X_lsi = svd.fit_transform(X_tfidf)
print(f"Reduced LSI shape: {X_lsi.shape}") # (docs x k topics)
# Step 4: Show similarity between some documents
def show_similar_docs(query_idx, top_n=5):
```

```
    similarities = cosine_similarity([X_lsi[query_idx]], X_lsi)[0]
    top_indices = similarities.argsort()[::-1][1:top_n+1]
    print(f"\nQuery Document {query_idx}:\n{newsgroups.data[query_idx][:300]}...\n")
    print("Top similar documents:")
    for i in top_indices:
        print(f"\nDoc #{i} (Similarity: {similarities[i]:.3f}):\n{newsgroups.data[i][:300]}...")
```

 # Example: Show top 5 similar documents to doc #0

```
show_similar_docs(query_idx=0, top_n=5)
```

OUTPUT:

```
Original TF-IDF shape: (1777, 1000)
Reduced LSI shape: (1777, 100)
Query Document #0:
Mike Vernon is now 3 wins 11 losses plus that All-Star game debacle in
afternoon games during his career...with another afternoon game with
Los Angeles next Sunday...has the ABC deal doomed the Flames?...
Top similar documents:
Doc #342 (Similarity: 0.686):
Dale Hunter ties the game, scoring his third goal of the game
with 2.7 seconds remaining in regulation.
    You could feel it coming on.
    "Due to contractual agreements, ESPN will be unable to carry
the rest of this game live, so that we may show you a worthless early-season battle between th...
Doc #1208 (Similarity: 0.658):
Showing a meaningless (relatively) baseball game over the overtime of
game that was tied up with less than 3 seconds left on the clock?
Gimme a break!  Where does ESPN get these BRILLIANT decisions from?...
```

WEEK-9

Mining Twitter to identify tweets for a specific period (and/or from a geographical location) and identify trends and named entities.

AIM:

To mine twitter for identifying tweets for a specific period.

DESCRIPTION

This program is designed to mine Twitter data in order to analyze social media activity within a specified **time period** and/or **geographical location**. By leveraging the Twitter API, it collects tweets that match a given query or hashtag, applies filters such as date ranges and geotags, and then processes the data to extract meaningful insights.

The workflow of the program includes:

1. **Data Collection**

- Connects to the Twitter API using a bearer token.
- Retrieves tweets based on user-defined criteria such as keywords, hashtags, time period, and geographical coordinates.

2. **Preprocessing**

- Cleans tweet text by removing URLs, mentions, hashtags, emojis, and stopwords.
- Normalizes the text for further linguistic analysis.

3. **Trend Analysis**

- Identifies the most frequent hashtags, keywords, and topics used during the selected period or within the chosen location.
- Detects trending discussions by analyzing tweet frequency patterns.

4. **Named Entity Recognition (NER)**

- Applies natural language processing (NLP) techniques to detect and classify named entities such as persons, organizations, locations, products, or events.
- Helps to understand which entities dominate public discourse in the given timeframe or region.

5. **Visualization & Insights**

- Generates word clouds, frequency charts, and time-based trend graphs.

- Provides insights into emerging topics, popular hashtags, and influential entities.

PROGRAM:

```
import tweepy

# Replace with your own Bearer Token from Twitter Developer Portal

bearer_token =
"AAAAAAAAAAAAAAAAAAAAADhM4QEAAAAAEhGXBfqa4kNwgb3%2F3XEC8JceL
Ys%3D0AVX5bRfhoQTvuRjjokbg7zOQ6egn1VOGtL2xEXIW4N7IGsX9P"

# Initialize Tweepy client with bearer token

client = tweepy.Client(bearer_token=bearer_token)

# Define your search query

query = "AI OR Machine Learning"

# Fetch recent tweets matching the query

tweets = client.search_recent_tweets(

    query=query,

    max_results=100,              # maximum results per request (up to 100)

    tweet_fields=['created_at', 'text']  # request tweet creation time and text

)

# Check if tweets are returned

if tweets.data is not None:

    # Print tweet creation date and text

    for tweet in tweets.data:

        print(f"Created at: {tweet.created_at}")

        print(f"Tweet text: {tweet.text}\n")

else:

    print("No tweets found for this query.")
```

OUTPUT:

Created at: 2025-09-23 03:51:48+00:00

Tweet text: RT @leiane1: Good morning, family

How are you?

The @recallnet Arena is NOW open.

Trade proven, high-volume pairs with real liquidity.…

Created at: 2025-09-23 03:51:48+00:00

Tweet text: @icanvardar @stripe Stripe is becoming an ai labs

Created at: 2025-09-23 03:51:48+00:00

Tweet text: RT @GaiAIio: 🚀 GaiAI Discord is live!

Join our growing community of creators, developers, and Web3 AI explorers.

Discuss ideas, share gen…

Created at: 2025-09-23 03:51:48+00:00

Tweet text: RT @psicolut: a virginia sambando daquele jeito como rainha de bateria e voce aí se cobrando pra tirar um projeto do papel porque ainda não…

Created at: 2025-09-23 03:51:48+00:00

Tweet text: @JnglJourney LOL....AI...UFOs....the spooky ghouls of Halloween arriving early....

Or is it EU countries confabulating fake narratives to blame Russia for these mystery sightings.

Created at: 2025-09-23 03:51:48+00:00

Tweet text: @OpenledgerFdn @kbwofficial @OpenledgerHQ Openledger is really building the fair layer of the OPEN internet AI

Created at: 2025-09-23 03:51:48+00:00

Tweet text: @69on_ai 这是哪部作品的人物呀

Created at: 2025-09-23 03:51:48+00:00

Tweet text: RT @FractionAI_xyz: Here’s a crazy thought:

Every Tuesday, we’ve been shipping something new and exciting, week in and week out.

This sh…

Created at: 2025-09-23 03:51:48+00:00

Tweet text: @darwinmda_ @jaofranko As vezes é mais sobre o traços do artista, mas acho que se ele fizesse o cara chorando ai sim seria

Created at: 2025-09-23 03:51:48+00:00

Tweet text: RT @skywongraveee: ผมเก็บสิ่งนี้มาได้มันคืออะไร ถาม ai ก็ไม่รู้ ใครทราบรบกวนแจ้งที

#MuTeLuvNotMyFatherEP1 https://t.co/VzYjCk7Dv5

Created at: 2025-09-23 03:51:48+00:00

Tweet text: @KAMADAN_AI めっちゃ共感です！

WEEK-10

Implementation of PageRank on Scholarly Citation Network.

AIM:

To implement Page Rank on scholarity Citation Network.

DESCRIPTION:

What is PageRank?

- PageRank is an algorithm originally developed by Google founders Larry Page and Sergey Brin.
- It measures the *importance* of nodes in a directed graph, based on the idea that a node (like a web page or research paper) is important if other important nodes link (or cite) it.
- In a scholarly citation network:
    - **Nodes = Research papers**
    - **Edges = Citations (directed links from citing paper to cited paper) Use of**

PageRank in citation networks

- To identify **influential research papers**.
- A paper is important not just because it has many citations, but because it is cited by other important papers.
- It helps rank scholarly articles in terms of *impact* rather than just *counting citations.*

Example

Suppose we have a small citation network:

- Paper1 → cites Paper2, Paper3
- Paper2 → cites Paper3
- Paper3 → cites Paper1

This forms a cycle. Running PageRank will eventually distribute scores showing which paper is most central in the citation loop.

PROGRAM:

```
pip install --upgrade numpy scipy network

import networkx as nx

# Example scholarly citation network

# Each node is a paper, edges represent citations

citations = {

    "Paper1": ["Paper2", "Paper3"],

    "Paper2": ["Paper3"],

    "Paper3": ["Paper1"],
    "Paper4": ["Paper2", "Paper3"],

    "Paper5": ["Paper3", "Paper4"]

}

# Build directed graph G = nx.DiGraph()

for paper, cited_papers in citations.items():

    for cited in cited_papers:

        G.add_edge(paper, cited)

# Compute PageRank manually (no scipy backend needed) pagerank_scores = nx.pagerank(G,

alpha=0.85, max_iter=100) print("\n PageRank Scores:")

for paper, score in pagerank_scores.items():

    print(f"{paper}: {score:.4f}")
```

OUTPUT:

```
PageRank Scores:
Paper1: 0.3515
Paper2: 0.1975
Paper3: 0.3782
Paper4: 0.0428
Paper5: 0.0300
```