Experiment 1

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

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

Description:

Objective: To understand and implement how text documents can be represented numerically using the **Vector Space Model (VSM)**. To compute the **similarity** between two documents using measures like **Cosine Similarity**.

What is Vector Space Model (VSM)?

VSM is an algebraic model that represents text documents as **vectors** in a multi-dimensional space. Each **dimension** corresponds to a unique **term (word)** in the corpus vocabulary. The **value** in each dimension may represent raw frequency, TF (Term Frequency), or TF-IDF (Term Frequency– Inverse Document Frequency) score.

Why Vector Representation?

Enables application of mathematical and statistical operations like distance or similarity.
Useful in information retrieval, document clustering, classification, and recommendation systems.
Cosine Similarity: Measures cosine of the angle between two vectors.

Ranges from 0 (no similarity) to 1 (identical direction).

Formula:

$$\cos(\theta) = \frac{\vec{A} \cdot \vec{B}}{||\vec{A}|| \cdot ||\vec{B}||}$$

Implementation:

Preprocessing: Clean and tokenize documents.
Vectorization: Convert documents into numerical vectors using TfidfVectorizer.
Similarity Computation: Use cosine_similarity from sklearn.metrics.pairwise.

Program:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Sample documents
doc1 = "Information retrieval is the process of obtaining relevant information from a collection of resources."
doc2 = "Text mining and information retrieval are important components of data science."

# Vectorize the documents using TF-IDF
vectorizer = TfidfVectorizer()
tfidf_matrix = vectorizer.fit_transform([doc1, doc2])

# Compute cosine similarity
similarity = cosine_similarity(tfidf_matrix[0:1], tfidf_matrix[1:2])

# Display similarity
print("Cosine Similarity between doc1 and doc2:", similarity[0][0])
```

Output:

```
Cosine Similarity between doc1 and doc2: 0.2315321376566199 2
```

Experiment-2

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

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

Description:

Objective: To perform text preprocessing by implementing two fundamental steps:

Stop word removal
Stemming

These steps reduce noise in text data and improve the performance of text mining and information retrieval tasks.

What is Text Preprocessing?

Text preprocessing refers to the transformation of raw text into a clean and analyzable format. It is the first and essential step in any natural language processing (NLP) or information retrieval task.

Stop Word Removal: Stop words are commonly used words (like is, the, and, in, etc.) that carry little meaningful information for analysis.

- Removing them reduces dimensionality and computational complexity.
- Libraries like NLTK, spaCy, and Scikit-learn offer predefined stop word lists.

Stemming: Stemming is the process of reducing inflected or derived words to their root or base form (e.g., playing, played, plays → play).

- Porter Stemmer and Lancaster Stemmer are commonly used.
- Unlike lemmatization, stemming is rule-based and faster but less accurate grammatically.

Implementation:

- **Input**: Raw text document.
- **Tokenization**: Split text into individual words.
- **Stopword Removal**: Remove all stop words using a stop word list.
- **Stemming**: Apply a stemming algorithm to the remaining tokens.
- **Output**: List of cleaned, stemmed tokens.

Program:

```
# Install spaCy and download model
!pip install -q spacy
!python -m spacy download en_core_web_sm

# Now import libraries
import spacy
from nltk.stem import PorterStemmer

# Load spaCy English model
nlp = spacy.load("en_core_web_sm")

# Sample text
text = "Text mining is the process of deriving meaningful information from natural
language text."

# Process the text
doc = nlp(text)

# Tokenize and remove stopwords using spaCy
tokens = [token.text.lower() for token in doc if token.is_alpha and not token.is_stop]

# Apply stemming using NLTK's PorterStemmer
stemmer = PorterStemmer()
```

```
stemmed_tokens = [stemmer.stem(token) for token in tokens]

# Output results
print("Original Text:\n", text)
print("\nTokens after Stop Word Removal:\n", tokens)
print("\nStemmed Tokens:\n", stemmed_tokens)
```

Output:

Original Text:

Text mining is the process of deriving meaningful information from natural language text.

Tokens after Stop Word Removal:
['text', 'mining', 'process', 'deriving', 'meaningful', 'information', 'natural', 'language', 'text']

Stemmed Tokens:
['text', 'mine', 'process', 'deriv', 'meaning', 'inform', 'natur', 'languag', 'text']

Experiment-3

3. Write a Python Program To Implement Signature Files by taking 2 documents.

Aim: To implement Signature files by taking 2 documents.

Description:

Signature File in Information Retrieval

What is a Signature File?

A Signature File is a classic indexing technique used in Information Retrieval (IR) systems to quickly find documents that may contain a given search term. Instead of storing the

entire text, each document is represented by a compact binary signature (bit string) that acts like a fingerprint.

- Each bit position in the signature corresponds to a word (or a group of words) in the vocabulary.
- If the word is present in the document → bit = 1
- If the word is absent → bit = 0

Thus, searching becomes faster since the system only needs to check these binary vectors rather than scanning the whole document.

How it Works (Steps)

1. Tokenization– Break documents into words.
2. Vocabulary Creation– Collect all unique words from the collection.
3. Signature Generation– For each document:
    - Initialize a binary array of length = number of unique words.
    - Set 1 in the positions where words occur, else keep 0.
4. Query Processing–
    - Convert the query word into its index in the signature.
    - Check all document signatures to see where the bit = 1.
    - Retrieve those documents as relevant results.

Program:

```
#Step 1: Create two documents

doc1 = "sachin virat dhoni anu"

doc2 = "anu chinna purna ram"

documents = [doc1, doc2]

#Step 2: Extract all unique words from both documents

unique_words = set()

tokenized_docs = []
```

```
for doc in documents:
    words = doc.lower().split()
    tokenized_docs.append(words)
    unique_words.update(words)
unique_words = sorted(unique_words)  # keep consistent order
print("Unique Words:", unique_words)
# Step 3: Generate signature for each unique word
word_index = {word: i for i, word in enumerate(unique_words)}
def create_signature(words):
    signature = [0] * len(unique_words)
    for word in words:
        if word in word_index:
            signature[word_index[word]] = 1
    return signature
# Create signature for each document
doc_signatures = [create_signature(words) for words in tokenized_docs]
# Show document signatures
print("\nDocument Signatures:")
for i, sig in enumerate(doc_signatures):
    print(f"Document {i+1}: {sig}")
# Step 4: Query and retrieve relevant documents
```

```
query = input("\nEnter a word to search: ").lower()
if query in word_index:
    query_idx = word_index[query]
    print(f"\nDocuments containing the word '{query}':")
    found = False
    for i, signature in enumerate(doc_signatures):
        if signature[query_idx] == 1:
            print(f"- Document {i+1}: \"{documents[i]}\"")
            found = True
    if not found:
        print("No documents found.")
else:
    print(f"The word '{query}' is not in the indexed vocabulary.")
```

Output:

```
Unique Words: ['anu', 'chinna', 'dhoni', 'purna', 'ram', 'sachin', 'virat']
Document Signatures:
Document 1: [1, 0, 0, 1, 0, 1, 1]
Document 2: [0, 1, 1, 0, 1, 1, 0]
Enter a word to search: sachin
Documents containing the word 'sachin':
- Document 1: "sachin virat dhoni anu"
```

Experiment-4

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 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.

Description:

What is Text Classification?

Text classification is a supervised learning task where the goal is to assign a label or category to a text document based on its content.

Examples:

- Classifying emails as spam or not spam
- Categorizing news into sports, politics, technology, etc.

Common Algorithms Used:

Algorithm	Description
Naive Bayes	Probabilistic classifier assuming word independence
SVM	Finds the optimal hyperplane separating classes

Implementation Steps:

1. **Dataset**: Use a **standard text dataset** like:
    - 20 Newsgroups (from sklearn.datasets)
    - SMS Spam Collection, or other labeled corpora.
2. **Preprocessing**:
    - Lowercasing
    - Stop-word removal
    - Vectorization using **TF-IDF**
3. **Model Training**:
    - Split dataset into training and testing sets
    - Train the chosen classifier on training data
4. **Evaluation**:
    - Predict classes on test data
    - Compute accuracy, precision, recall, and F1-score

Program:

```
# Install required packages (if needed)
!pip install -q scikit-learn
```

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import LinearSVC
from sklearn.metrics import classification_report, accuracy_score

# Load 20 Newsgroups dataset
categories = ['alt.atheism', 'comp.graphics', 'sci.med', 'soc.religion.christian']
newsgroups = fetch_20newsgroups(subset='all', categories=categories, shuffle=True,
random_state=42)

#  Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
  newsgroups.data, newsgroups.target, test_size=0.3, random_state=42
)

#  Vectorize text using TF-IDF
vectorizer = TfidfVectorizer(stop_words='english', max_df=0.5)
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)

# Train Naive Bayes Classifier
nb_model = MultinomialNB()
nb_model.fit(X_train_tfidf, y_train)
nb_preds = nb_model.predict(X_test_tfidf)

#  Train SVM Classifier
svm_model = LinearSVC()
svm_model.fit(X_train_tfidf, y_train)
svm_preds = svm_model.predict(X_test_tfidf)

#  Evaluate both models
print("\n Naive Bayes Classifier Results:")
print(classification_report(y_test, nb_preds,
target_names=newsgroups.target_names))
print("Accuracy:", accuracy_score(y_test, nb_preds))
```

```
print("\n SVM Classifier Results:")
print(classification_report(y_test, svm_preds,
target_names=newsgroups.target_names))
print("Accuracy:", accuracy_score(y_test, svm_preds))
```

Output:

```
🔍 Naive Bayes Classifier Results:
                          precision    recall  f1-score   support

             alt.atheism       1.00      0.88      0.93       252
           comp.graphics       0.97      0.99      0.98       295
                 sci.med       0.99      0.94      0.97       299
soc.religion.christian       0.87      0.98      0.92       282

                accuracy                           0.95      1128
               macro avg       0.96      0.95      0.95      1128
            weighted avg       0.95      0.95      0.95      1128

Accuracy: 0.950354609929078
```

```
🔍 SVM Classifier Results:
                          precision    recall  f1-score   support

             alt.atheism       0.98      0.96      0.97       252
           comp.graphics       0.95      1.00      0.97       295
                 sci.med       0.98      0.96      0.97       299
soc.religion.christian       0.97      0.96      0.97       282

                accuracy                           0.97      1128
               macro avg       0.97      0.97      0.97      1128
            weighted avg       0.97      0.97      0.97      1128

Accuracy: 0.9689716312056738
```

Experiment-5

5. Text Document Clustering using K-means. Demonstrate with a standard dataset and compute performance measures- Purity.

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

Description:

What is Document Clustering?

- Document clustering is an unsupervised learning technique used to group similar text documents together into clusters.
- Unlike classification, clustering does not require labeled data.

What is K-Means Clustering?

- A centroid-based algorithm that partitions the dataset into K clusters.
- It minimizes the distance between data points and the centroid of the assigned cluster.
- In text mining, documents are first converted into TF-IDF vectors, and then clustered using Euclidean or cosine distance.

Evaluation Metrics:

Since clustering is unsupervised, evaluation requires comparing clusters with **true class labels (if available)**.

Metric	Description
Purity	Measures the extent to which each cluster contains documents from a single class

Implementation Steps:

1. **Dataset**: Use a labeled text dataset (like 20 Newsgroups with subset categories).
2. **Preprocessing**:
   - Lowercase conversion
   - Stopword removal
   - TF-IDF vectorization
3. **Clustering**:
   - Apply **K-Means algorithm** from sklearn.cluster
   - Set k to the number of true categories
4. **Evaluation**:
   - Match clusters to true labels
   - Compute **Purity**.

Program:

```
#Install scikit-learn (if not already)
!pip install -q scikit-learn

from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans
import numpy as np

#Load 4 categories from 20 Newsgroups dataset for simplicity
categories = ['alt.atheism', 'comp.graphics', 'sci.med', 'soc.religion.christian']
newsgroups = fetch_20newsgroups(subset='all', categories=categories, shuffle=True,
random_state=42)

#Vectorize using TF-IDF
vectorizer = TfidfVectorizer(stop_words='english', max_df=0.5)
X = vectorizer.fit_transform(newsgroups.data)
y_true = newsgroups.target

#Apply K-Means clustering
k = len(categories)
model = KMeans(n_clusters=k, init='k-means++', max_iter=100, n_init=10,
random_state=42)
model.fit(X)
y_pred = model.labels_
```

```
#--------- Purity Calculation ----------
def purity_score(y_true, y_pred):
  contingency = confusion_matrix(y_true, y_pred)
  return np.sum(np.amax(contingency, axis=0)) / np.sum(contingency)

#--------- Evaluation Metric ----------
print("Purity Score:", purity_score(y_true, y_pred))
```

Output:

Purity Score: 0.5163607342378292

Experiment-6

6. Building an Inverted Files using Sorted Array than search a documents by using the created Inverted File.

Aim: To implement Building an Inverted Files using Sorted Array than search a documents by using the created Inverted File.

Description:

Definition:

An Inverted File (or Inverted Index) is a data structure that maps terms (keywords) to their occurrences in documents. It essentially inverts the typical structure of documents containing words into a structure where words point to documents.

Structure:

Dictionary (Vocabulary List): Contains all unique terms in the document collection.

Posting Lists: For each term, a list of document identifiers (DocIDs) where the term appears.

Program:

```
# Step 1: Create two documents
doc1 = "anu chinna satya sudha"
doc2 = "kiran diya radha anu"
documents = [doc1, doc2]

# Step 2: Tokenize documents and build inverted index
inverted_index = {}

for doc_id, doc in enumerate(documents):
  words = doc.lower().split()
  for word in words:
    if word not in inverted_index:
      inverted_index[word] = set()
    inverted_index[word].add(doc_id)

# Step 3: Convert sets to sorted arrays
for word in inverted_index:
  inverted_index[word] = sorted(inverted_index[word])

# Step 4: Display inverted index
print("Inverted Index (word -> document IDs):")
for word in sorted(inverted_index):
  print(f"{word}: {inverted_index[word]}")

# Step 5: Query
query = input("\nEnter a word to search: ").lower()

if query in inverted_index:
  doc_ids = inverted_index[query]
  print(f"\nDocuments containing the word '{query}':")
  for doc_id in doc_ids:
    print(f"- Document {doc_id + 1}: \"{documents[doc_id]}\"")
else:
```

```
print(f"The word '{query}' is not in the indexed vocabulary.")
```

Output:

```
Inverted Index (word -> document IDs):
anu: [0, 1]
chinna: [0]
diya: [1]
kiran: [1]
radha: [1]
satya: [0]
sudha: [0]

Enter a word to search: radha

Documents containing the word 'radha':
- Document 2: "kiran diya radha anu"
```

Experiment-7

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

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

Description:

Objective: Conceptual Overview:

What is XML Parsing?

XML (eXtensible Markup Language) is a standard format for representing structured data. In the context of the Web, XML is often used to describe the structure of websites, including links between pages.

What is a Web Graph?

A **Web Graph** is a directed graph where:

- **Nodes** represent web pages.
- **Edges** represent hyperlinks between them.

What is PageRank?

PageRank is an algorithm used to rank web pages based on their **link structure**. It assumes that **important pages are linked to by many other important pages**.

Topic-Specific PageRank:

Unlike global PageRank, **Topic-Specific PageRank** biases the random surfer model towards a **specific topic or set of nodes**. The teleportation step prefers pages related to a specific topic, helping in **focused web crawling** and **topic-sensitive search**.

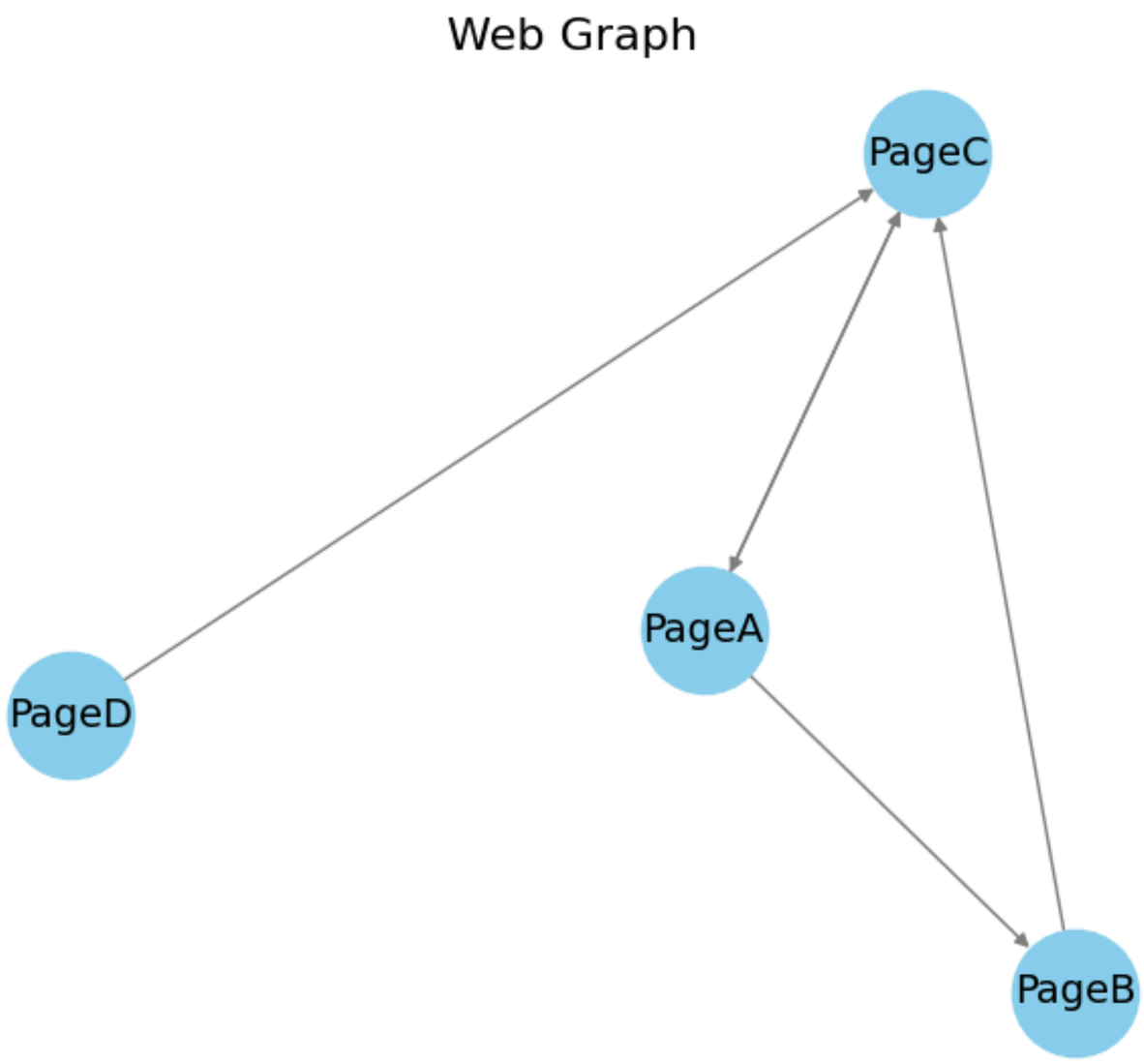
Program:

```
# Install required libraries
!pip install -q networkx
import xml.etree.ElementTree as ET
import networkx as nx
import matplotlib.pyplot as plt
#  Sample XML Data (you can replace this with actual XML content)
xml_data = '''
```

```
<pages>
 <page name="PageA">
  <link>PageB</link>
  <link>PageC</link>
 </page>
 <page name="PageB">
  <link>PageC</link>
 </page>
<page name="PageC">
 <link>PageA</link>
</page>
<page name="PageD">
 <link>PageC</link>
 </page>
</pages>
'''
# Parse XML
root = ET.fromstring(xml_data)
# Create directed graph
web_graph = nx.DiGraph()
# Add nodes and edges
for page in root.findall('page'):
  src = page.get('name')
  web_graph.add_node(src)
  for link in page.findall('link'):
```

```
    dest = link.text
    web_graph.add_edge(src, dest)
# Display the graph
plt.figure(figsize=(6, 5))
nx.draw(web_graph, with_labels=True, node_color='skyblue', edge_color='gray',
node_size=2000, font_size=14)
plt.title("Web Graph", fontsize=16)
plt.show()
#  Compute PageRank
pagerank_scores = nx.pagerank(web_graph, alpha=0.85)
#  Display results
print(" PageRank Scores:")
for page, score in pagerank_scores.items():
  print(f"{page}:{score:.4f}")
```

Output:



PageRank Scores:

PageA: 0.3725

PageB: 0.1958

PageC: 0.3942

PageD: 0.0375

Experiment-8

8. Implement Matrix Decomposition and LSI for a standard dataset.

Aim: To Implement Matrix Decomposition and LSI for a standard dataset.

Description:

Objective:

Matrix Decomposition

- Matrix decomposition means breaking a large matrix into smaller pieces that are easier to work with.
- In text mining / information retrieval, we usually represent data in a term-document matrix:
    - Rows = words (terms)
    - Columns = documents
    - Each cell = frequency (or weight like TF-IDF) of a word in a document

Latent Semantic Indexing (LSI)

- LSI is an application of matrix decomposition (SVD) in text mining / search engines.
- Idea: Words that appear in similar contexts have similar meanings (even if they don’ t exactly match).

Program:

Install scikit-learn if not already

```
!pip install -q scikit-learn
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD
import numpy as np
#  Sample Document Collection (you can load files instead)
documents = [
  "The cat sat on the mat.",
  "Dogs and cats are both pets.",
  "The dog chased the cat.",
  "Cats sleep on warm mats.",
  "Dogs bark and guard the house."
]
# Step 1: Convert to TF-IDF matrix
vectorizer = TfidfVectorizer(stop_words='english')
X_tfidf = vectorizer.fit_transform(documents)
# Step 2: Apply Truncated SVD (LSI)
num_topics = 2  # You can increase this number
svd_model = TruncatedSVD(n_components=num_topics, random_state=42)
X_lsi = svd_model.fit_transform(X_tfidf)
# Step 3: Display LSI Topics (term importance in each topic)
terms = vectorizer.get_feature_names_out()
for i, comp in enumerate(svd_model.components_):
  terms_in_topic = zip(terms, comp)
  sorted_terms = sorted(terms_in_topic, key=lambda x: x[1], reverse=True)
  print(f"\nTopic {i + 1}:")
```

```
    for term, weight in sorted_terms[:5]:
        print(f"  {term} ({weight:.4f})")
# Step 4: Show document-topic matrix
print("\n Document representation in reduced LSI space:")
for i, doc_vec in enumerate(X_lsi):
    print(f"Doc {i + 1}: {doc_vec}")
```

Output:

Topic 1:

cats (0.5115)

dogs (0.5115)

pets (0.4060)

bark (0.2280)

guard (0.2280)

Topic 2:

cat (0.6279)

chased (0.3891)

dog (0.3891)

mat (0.3891)

sat (0.3891)

Document representation in reduced LSI space:

Doc 1: [7.41494481e-17 7.89158991e-01]

Doc 2: [8.11647332e-01 -6.13828483e-17]

Doc 3: [-1.90576631e-16 7.89158991e-01]

Doc 4: [5.73921332e-01 2.01344316e-16]

Doc 5: [5.73921332e-01 3.43134775e-16]

Experiment-9

9. Search a Substring in a string by using Navie Algorithm.

Aim: To Search a Substring in a string by using Navie Algorithm.

Description:

The naive string searching algorithm is the simplest approach to finding a pattern within a given text. The idea is to align the pattern at the beginning of the text and check character by character to see if the entire pattern matches. If a mismatch occurs at any position, the algorithm shifts the pattern by one character to the right and repeats the process. This continues until the end of the text is reached.

Program:

```
def search_pattern(pattern, text):
```

```
  m = len(pattern)
  n = len(text)
  for i in range(n - m + 1):
    j = 0
    while j < m and text[i + j] == pattern[j]:
      j += 1
    if j == m:
      print(f"Pattern found at index {i}")
if __name__ == "__main__":
  text1 = input("Enter the Text: ")
  pattern1 = input("Enter the Pattern: ")
  search_pattern(pattern1, text1)
```

Output:

Enter the Text: AI&ML students comes to IR Lab

Enter the Pattern: students

Pattern found at index 6

Experiment-10

10. Implementation of PageRank on Scholarly Citation Network.

Aim: To implementation of PageRank on Scholarly Citation Network.

Description:

Objective: To implement the **PageRank algorithm** on a **Scholarly Citation Network** to determine the relative importance of research papers based on citations, similar to how Google ranks web pages.

Conceptual Overview:

What is PageRank?

PageRank is an algorithm originally developed by **Google** to rank web pages. It assigns a numerical weight to each node (e.g., a webpage or paper) based on the **number and quality of links (citations)** it receives.

In the context of scholarly citations:

- Each **node** represents a research paper.
- A **directed edge** from paper A to paper B implies that A cites B.
- A paper that is cited by many important papers gets a **higher PageRank score**.

Why PageRank for Citation Networks?

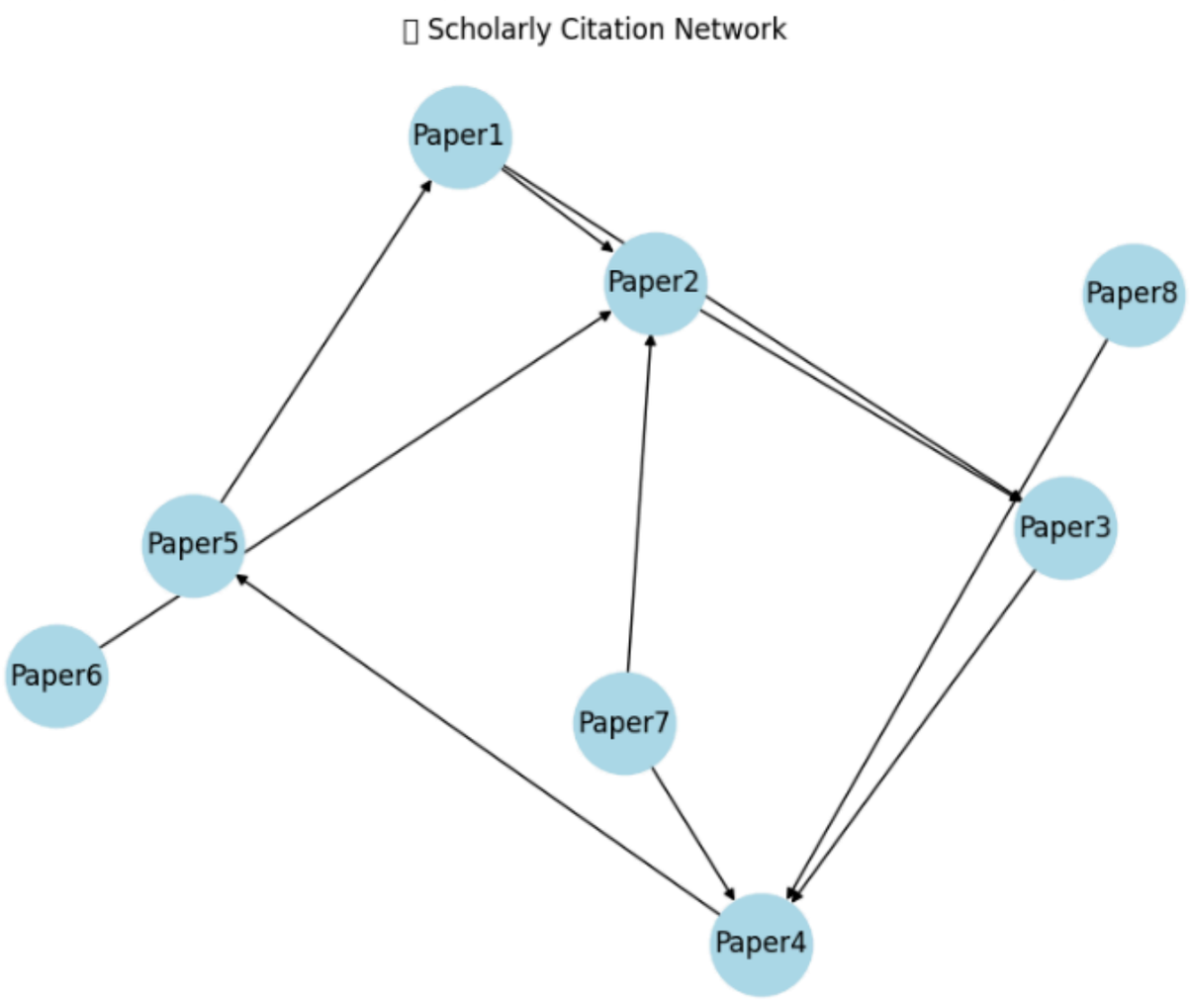
- Helps identify **influential or foundational research papers**.
- Better than just counting citations, since it also considers **who is citing** the paper.
- Useful in **literature reviews, recommender systems**, and **academic search engines**.

Program:

```
# Install required library
!pip install -q networkx matplotlib
import networkx as nx
import matplotlib.pyplot as plt
#  Step 1: Simulate a Scholarly Citation Network
# Nodes represent papers, edges represent citations (Paper A cites Paper B)
citation_edges = [
    ("Paper1", "Paper2"),
    ("Paper1", "Paper3"),
    ("Paper2", "Paper3"),
    ("Paper3", "Paper4"),
    ("Paper4", "Paper5"),
    ("Paper5", "Paper1"), # Circular citation
    ("Paper6", "Paper2"),
    ("Paper7", "Paper2"),
    ("Paper7", "Paper4"),
```

```
    ("Paper8", "Paper4"),
]
# Step 2: Create a directed graph
G = nx.DiGraph()
G.add_edges_from(citation_edges)
# Step 3: Visualize the graph
plt.figure(figsize=(8,6))
pos = nx.spring_layout(G, seed=42)
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=2000,
arrows=True, font_size=12)
plt.title("Scholarly Citation Network")
plt.show()
# Step 4: Compute PageRank
pagerank_scores = nx.pagerank(G, alpha=0.85)
# Step 5: Display PageRank scores
print("\n PageRank Scores (Influence of Papers):")
sorted_scores = sorted(pagerank_scores.items(), key=lambda x: x[1], reverse=True)
for paper, score in sorted_scores:
    print(f"{paper}: {score:.4f}")
```

Output:



PageRank Scores (Influence of Papers):
Paper4: 0.2177
Paper3: 0.2060
Paper5: 0.2038

Paper1: 0.1920
Paper2: 0.1243
Paper6: 0.0188
Paper7: 0.0188
Paper8: 0.0188