WEEK – 4

Experiment Title:

Classification of a Set of Text Documents into Known Classes

Aim:

To perform classification of text documents into known classes using machine learning algorithms such as Naive Bayes

Objective:

To understand and implement text classification techniques using supervised learning algorithms on standard datasets.

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 labeling
- News categorization

Naive Bayes Classifier:

Concept:

Naive Bayes is a probabilistic classification algorithm based on **Bayes' Theorem**, assuming independence among features. Despite this "naive" assumption, it works effectively in text classification tasks.

Program:

```
# Import necessary libraries
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
```

Experiment 7: WEEK-7 - Parse XML Text, Generate Web Graph, and Compute Topic-Specific PageRank

AIM

To parse XML text, generate a web graph, and compute topic-specific PageRank.

DESCRIPTION

- **Parse XML**: Read XML structure to extract pages, links, titles, topics (using xml.etree.ElementTree).
- **Generate Web Graph**: Directed graph where nodes = pages, edges = hyperlinks (citations).
- **Topic-Specific PageRank**: Modified PageRank biasing towards topic-relevant pages.
    - Standard PageRank: Importance based on incoming links from important pages.
    - Topic-Specific: Teleportation vector favors topic pages (e.g., "science").
    - Formula: PR = α * M * PR + (1-α) * v (M=transition matrix, v=topic teleport vector, α=0.85 damping).
- Process: Parse XML → Build adjacency matrix → Compute ranks → Visualize graph.

PROGRAM

```
import xml.etree.ElementTree as ET
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt

# Sample XML (use your own or this for demo)
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>'''

# 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

# 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
    return M, pages
```

```
# 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.get(p, []) else 0.0 for p in pages])
    if teleport.sum() == 0:
        teleport = np.ones(n)
    teleport /= teleport.sum()  # Normalize
    r = np.ones(n) / n
    for _ 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))

# Visualize Graph
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 = ["lightgreen" if topic in topics_map.get(page, []) else "skyblue" for page in
G.nodes()]
    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()

# Execute
graph, topics_map = parse_xml(xml_text)
M, pages = build_adj_matrix(graph)
topic = "science"
draw_web_graph(graph, topics_map, topic)
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}")
```

Experiment 10: WEEK-10 - Implementation of PageRank on Scholarly Citation Network

AIM

To implement PageRank on a scholarly citation network.

DESCRIPTION

- **PageRank**: Measures node importance in a directed graph (e.g., citations). A paper is important if cited by important papers.
- **Scholarly Network**: Nodes = papers, Edges = citations (directed from citing to cited).
- **Use**: Ranks papers by impact, beyond citation count.
- Process: Build graph → Compute PageRank (α=0.85) → Output scores.
- Example: Cyclic citations distribute ranks based on influence.

PROGRAM

python

```
import networkx as nx

# Sample citation network
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)
0pagerank_scores = nx.pagerank(G, alpha=0.85, max_iter=100)
print("PageRank Scores:")
for paper, score in pagerank_scores.items():
    print(f"{paper}: {score:.4f}")
```

OUTPUT

text

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