Information Retrieval Practical

Practical 1

Code 1 :

```
import nltk
from nltk.corpus import stopwords

# Define the documents
document1 = "The quick brown fox jumped over the lazy dog"
document2 = "The lazy dog slept in the sun"

# Download stopwords and get the list for English
nltk.download('stopwords')
stopWords = stopwords.words('english')

# Tokenize and preprocess the documents
tokens1 = document1.lower().split()
tokens2 = document2.lower().split()

# Combine the tokens into a list of unique terms
terms = list(set(tokens1 + tokens2))

# Initialize dictionaries for the inverted index and occurrences
inverted_index = {}
occ_num_doc1 = {}
occ_num_doc2 = {}

# Build the inverted index
for term in terms:
    if term in stopWords:
        continue
    documents = []
    if term in tokens1:
        documents.append("Document 1")
        occ_num_doc1[term] = tokens1.count(term)
    if term in tokens2:
        documents.append("Document 2")
        occ_num_doc2[term] = tokens2.count(term)
    inverted_index[term] = documents

# Print the inverted index
for term, documents in inverted_index.items():
    print(term, "->", end=" ")
    for doc in documents:
        if doc == "Document 1":
            print(f"{doc} ({occ_num_doc1.get(term, 0)}),", end=" ")
        else:
```

```
            print(f"{doc} ({occ_num_doc2.get(term, 0)}),", end=" ")
    print()
```

Output 1 :

==========END OF PRACTICAL==========

Practical 2

Code 2a :

```
# Documents
documents = {
    1: "apple banana orange",
    2: "apple banana",
    3: "banana orange",
    4: "apple"
}

# Function to build an inverted index
def build_index(docs):
    index = {}
    for doc_id, text in docs.items():
        for term in set(text.split()):
            index.setdefault(term, set()).add(doc_id)
    return index

# Boolean AND operation
def boolean_and(operands, index):
    if not operands:
        return list(range(1, len(documents) + 1))
    result = index.get(operands[0], set())
    for term in operands[1:]:
        result &= index.get(term, set())
    return list(result)
```

```
# Boolean OR operation
def boolean_or(operands, index):
    result = set()
    for term in operands:
        result |= index.get(term, set())
    return list(result)

# Boolean NOT operation
def boolean_not(operand, index, total_docs):
    all_docs_set = set(range(1, total_docs + 1))
    return list(all_docs_set - index.get(operand, set()))

# Build the inverted index
inverted_index = build_index(documents)

# Example queries
query1 = ["apple", "banana"]  # AND query
query2 = ["apple", "orange"]  # OR query
query3 = "orange"             # NOT query

# Perform queries
result1 = boolean_and(query1, inverted_index)
result2 = boolean_or(query2, inverted_index)
result3 = boolean_not(query3, inverted_index, len(documents))

# Output results
print("Documents containing 'apple' AND 'banana':", result1)
print("Documents containing 'apple' OR 'orange':", result2)
print("Documents NOT containing 'orange':", result3)
print("Performed by 740_Pallavi & 743_Deepak")
```

Output 2a :

Documents containing 'apple' AND 'banana': [1, 2]
Documents containing 'apple' OR 'orange': [1, 2, 3]
Documents NOT containing 'orange': [2, 4]

Code 2b :

```
from sklearn.feature_extraction.text import CountVectorizer,
TfidfTransformer
import numpy as np
from numpy.linalg import norm

# Training and test documents
train_set = ["The sky is blue.", "The sun is bright."]
test_set = ["The sun in the sky is bright."]

# Initialize vectorizer (with stopwords) and transformer
vectorizer = CountVectorizer(stop_words="english")
transformer = TfidfTransformer()

# Generate TF-IDF features
train_vectors = vectorizer.fit_transform(train_set).toarray()
test_vectors = vectorizer.transform(test_set).toarray()

# Display TF-IDF features
print("Training Set TF-IDF:", train_vectors)
print("Test Set TF-IDF:", test_vectors)

# Function to compute cosine similarity
def cosine_similarity(vec1, vec2):
    return round(np.dot(vec1, vec2) / (norm(vec1) * norm(vec2)), 3)

# Compute cosine similarity between training and test vectors
for train_vec in train_vectors:
    for test_vec in test_vectors:
        print("Cosine Similarity:", cosine_similarity(train_vec,
test_vec))
```

Output 2b :

==========END OF PRACTICAL==========

Practical 3

Code 3 :

```
# A Naive recursive python program to find minimum number of operations to
convert str1 to str2
def editDistance(str1, str2, m, n):
    # If first string is empty, the only option is to insert all
characters of second string into first
    if m == 0:
        return n
    # If second string is empty, the only option is to remove all
characters of first string
    if n == 0:
        return m
    # If last characters of two strings are same, nothing much to do.
Ignore last characters and get count for remaining strings.
    if str1[m-1] == str2[n-1]:
        return editDistance(str1, str2, m-1, n-1)
    # If last characters are not same, consider all three operations on
last character of first string,
    # recursively compute minimum cost for all three operations and take
minimum of three values.
    return 1 + min(editDistance(str1, str2, m, n-1),   # Insert
                   editDistance(str1, str2, m-1, n),   # Remove
                   editDistance(str1, str2, m-1, n-1))  # Replace

# Driver code
str1 = "sunday"
```

```
str2 = "saturday"
print('Edit Distance is:', editDistance(str1, str2, len(str1), len(str2)))
```

Output 3 :

==========END OF PRACTICAL==========

Practical 4

Code 4a :

```
def calculate_metrics(retrieved_set, relevant_set):
    # Calculate True Positive, False Positive, and False Negative
    true_positive = len(retrieved_set.intersection(relevant_set))
    false_positive = len(retrieved_set.difference(relevant_set))
    false_negative = len(relevant_set.difference(retrieved_set))

    # Print the metrics
    print("True Positive:", true_positive)
    print("False Positive:", false_positive)
    print("False Negative:", false_negative)

    # Calculate Precision, Recall, and F-measure
    precision = true_positive / (true_positive + false_positive)
    recall = true_positive / (true_positive + false_negative)
    f_measure = 2 * precision * recall / (precision + recall)

    return precision, recall, f_measure

# Example sets
retrieved_set = {"doc1", "doc2", "doc3"}  # Predicted set
relevant_set = {"doc1", "doc4"}  # Actually needed set (Relevant)

# Calculate and print Precision, Recall, and F-measure
precision, recall, f_measure = calculate_metrics(retrieved_set,
relevant_set)
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F-measure: {f_measure}")
```

Output 4a :

True Positive: 1
False Positive: 2
False Negative: 1
Precision: 0.3333333333333333

Recall: 0.5
F-measure: 0.4

Code 4b :

```
from sklearn.metrics import average_precision_score

# Binary ground truth labels and model scores
y_true = [0, 1, 1, 0, 1, 1]  # True labels (binary)
y_scores = [0.1, 0.4, 0.35, 0.8, 0.65, 0.9]  # Predicted scores from the
model

# Calculate and print the average precision-recall score
avg_precision = average_precision_score(y_true, y_scores)
print(f'Average Precision-Recall Score: {avg_precision}')
```

Output 4b :

Average Precision-Recall Score: 0.75

==========END OF PRACTICAL==========

Practical 5

Code 5 :

==========END OF PRACTICAL==========

Practical 6

Install library:

File name: kmeans_clustering.py

Code 6 :

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans

# Input documents
documents = [
    "Cats are known for their agility and grace",  # cat doc1
    "Dogs are often called 'man's best friend'.",  # dog doc1
    "Some dogs are trained to assist people with disabilities.",  # dog
doc2
    "The sun rises in the east and sets in the west.",  # sun doc1
    "Many cats enjoy climbing trees and chasing toys.",  # cat doc2
```

```
]

# Create a TfidfVectorizer object
vectorizer = TfidfVectorizer(stop_words='english')

# Transform documents into TF-IDF vectors
X = vectorizer.fit_transform(documents)

# Perform k-means clustering
kmeans = KMeans(n_clusters=3, random_state=0).fit(X)

# Print cluster labels for each document
print("Cluster Labels for Documents:")
for i, label in enumerate(kmeans.labels_):
    print(f"Document {i + 1}: Cluster {label}")
```

Output 6 :

==========END OF PRACTICAL==========

Practical 7

Library install :

File name : web_crawler.py

Code 7 :

```
import requests
from bs4 import BeautifulSoup
import time
from urllib.parse import urljoin
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
        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)

# Example usage:
print("Performed by 740_Pallavi & 743_Deepak")
crawl('https://wikipedia.com', max_depth=2, delay=2)
```

Output 7 :

Performed by 740_Pallavi & 743_Deepak

Crawling https://wikipedia.com
Crawling https://www.wikipedia.org/
Crawling https://meta.wikimedia.org/
...

==========**END OF PRACTICAL**==========

Practical 8

Library required:

File name: page_rank.py

Code 8 :

```
import numpy as np

def page_rank(graph, damping_factor=0.85, max_iterations=100,
tolerance=1e-6):
    # Get the number of nodes
    num_nodes = len(graph)
```

```
    # Initialize PageRank values
    page_ranks = np.ones(num_nodes) / num_nodes

    # Iterative PageRank calculation
    for _ in range(max_iterations):
        prev_page_ranks = np.copy(page_ranks)

        for node in range(num_nodes):
            # Calculate the contribution from incoming links
            incoming_links = [i for i, v in enumerate(graph) if node in v]
            if not incoming_links:
                continue

            page_ranks[node] = (1 - damping_factor) / num_nodes + \
                damping_factor * sum(prev_page_ranks[link] /
                                     len(graph[link]) for link in
incoming_links)

        # Check for convergence
        if np.linalg.norm(page_ranks - prev_page_ranks, 2) < tolerance:
            break

    return page_ranks

# Example usage
if __name__ == "__main__":
    # Define a simple directed graph as an adjacency list
    # Each index represents a node, and the list at that index contains
nodes to which it has outgoing links
    web_graph = [
        [1, 2],  # Node 0 has links to Node 1 and Node 2
        [0, 2],  # Node 1 has links to Node 0 and Node 2
        [0, 1],  # Node 2 has links to Node 0 and Node 1
        [1, 2],  # Node 3 has links to Node 1 and Node 2
    ]

    # Calculate PageRank
    result = page_rank(web_graph)

    # Display PageRank values
    for i, pr in enumerate(result):
        print(f"Page {i}: {pr}")
```

Output 8 :

==========END OF PRACTICAL==========

Practical 9

Library required:

```
pip install numpy scikit-learn
```

pip install numpy scikit-learn sumy

File name: rank_svm.py

If upgrade error comes then run command in terminal : **python -m pip install --upgrade pip**

Code 9a :

```
import numpy as np
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Generating synthetic ranking dataset
X = np.array([[1, 2], [2, 3], [3, 3], [4, 5], [5, 5], [6, 7]])  # Features
y = np.array([0, 0, 1, 1, 2, 2])  # Relevance scores (higher is better)

# Splitting into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Using an SVM classifier as a simple ranking model
model = SVC(kernel="linear", probability=True)
model.fit(X_train, y_train)

# Predicting relevance scores
y_pred = model.predict(X_test)
```

```
# Evaluating model performance
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.2f}")
```

Output 9a :

Code 9b :

```
import numpy as np
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Synthetic ranking dataset
X = np.array([[1, 2], [2, 3], [3, 3], [4, 5], [5, 5], [6, 7]])  # Features
y = np.array([0, 0, 1, 1, 2, 2])  # Relevance scores (higher is better)

# Increase the test size to make sure we have enough samples to stratify
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5,
stratify=y, random_state=42)

# Using SVM classifier
model = SVC(kernel="linear", probability=True)
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Model accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.2f}")

# Display predictions for understanding
print("True labels: ", y_test)
print("Predicted labels: ", y_pred)
```

Output 9b :

==========END OF PRACTICAL==========

Practical 10

Library required:

pip install numpy scikit-learn sumy

File name: text_summary.py

Code 10 :

```
from sumy.parsers.plaintext import PlaintextParser
from sumy.nlp.tokenizers import Tokenizer
from sumy.summarizers.lsa import LsaSummarizer

# Sample text for summarization
text = """
Information Retrieval is a field concerned with searching for relevant
documents
within a large dataset. Text summarization is the process of reducing a
document's
length while maintaining its main points. Extractive methods identify and
extract
key sentences, while abstractive methods generate summaries in their own
words.
"""
# Parse the text

parser = PlaintextParser.from_string(text, Tokenizer("english"))

# Summarize using LSA (Latent Semantic Analysis)
summarizer = LsaSummarizer()
summary = summarizer(parser.document, 2)  # Number of sentences in summary

# Print the summary
for sentence in summary:
    print(sentence)
```

Output 10 :

==========END OF PRACTICAL==========