Practical : - 1

Document Indexing and Retrieval

a) Implement an inverted index construction algorithm.

b) Build a simple document retrieval system using the constructed index.

```
import nltk
nltk.download('stopwords')
nltk.download('punkt')
from collections import defaultdict
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
class SimpleSearchEngine:
    def __init__(self):
        self.inverted_index = defaultdict(list)
        self.stop_words = set(stopwords.words('english'))
        self.stemmer = PorterStemmer()
    def index_document(self, doc_id, document):
        words = [self.stemmer.stem(word.lower()) for word in word_tokenize(document) if 
word.isalnum() and word not in self.stop_words]
        for word in set(words):  # Using set to ensure unique terms in a document
            self.inverted_index[word].append(doc_id)
    def print_inverted_index(self):
        print("Inverted Index:")
        for term, doc_ids in self.inverted_index.items():
            print(f"{term}: {doc_ids}")
    def search(self, query):
        query_terms = set([self.stemmer.stem(term.lower()) for term in word_tokenize(query) if 
term.isalnum() and term not in self.stop_words])
        relevant_docs = set()
        for term in query_terms:
            relevant_docs.update(self.inverted_index.get(term, []))
        return relevant_docs
search_engine = SimpleSearchEngine()

search_engine.index_document(1, "This is a sample document about python.")
search_engine.index_document(2, "Python programming language is widely used.")
```

```
search_engine.index_document(3, "Document indexing and retrieval is important in
information retrieval.")
search_engine.print_inverted_index()

user_query = input("\nEnter your search query: ")
result = search_engine.search(user_query)
print("\nRelevant Documents:", result)
```
****

Practical :-2A

Retrieval Models

a) Implement the Boolean retrieval model and process queries.

b) Implement the vector space model with TF-IDF weighting and cosine similarity.

```
class BooleanRetrievalModel:
    def __init__(self, documents):
        self.inverted_index = self.build_inverted_index(documents)
    def build_inverted_index(self, documents):
        inverted_index = {}
        for doc_id, document in enumerate(documents):
            terms = set(document.split())
            for term in terms:
                inverted_index.setdefault(term, set()).add(doc_id)
        return inverted_index
    def boolean_search(self, query):
        query_terms = set(query.split())
        result = set(range(len(self.inverted_index)))
        for term in query_terms:
            result = result.intersection(self.inverted_index.get(term, set()))
            if not result:
                break
        return result

documents = [
    "This is a sample document about Python programming.",
    "Python is a widely used programming language.",
    "Document retrieval is important in information systems."
]
boolean_model = BooleanRetrievalModel(documents)
```

```
boolean_result = boolean_model.boolean_search("Python")
if boolean_result:
    print("Boolean Retrieval Result:", boolean_result)
else:
    print("No matching documents found for the Boolean query.")
```

Spelling Correction in IR Systems

a) Develop a spelling correction module using edit distance algorithms.

b) Integrate the spelling correction module into an information retrieval system.

Practical :-2B

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
class VectorSpaceModel:
    def __init__(self, documents):
        self.documents = documents
        self.vectorizer = TfidfVectorizer(stop_words='english')
        self.tf_idf_matrix = self.vectorizer.fit_transform(documents)
    def vector_space_search(self, query):
        query_vector = self.vectorizer.transform([query])
        cosine_similarities = cosine_similarity(query_vector, self.tf_idf_matrix).flatten()
        ranked_documents = sorted(enumerate(cosine_similarities), key=lambda x: x[1],
reverse=True)
        return [doc_id for doc_id, similarity in ranked_documents]

documents = [
    "This is a sample document about Python programming.",
    "Python is a widely used programming language.",
    "Document retrieval is important in information systems."
]
vector_space_model = VectorSpaceModel(documents)

vector_space_result = vector_space_model.vector_space_search("Python")
if vector_space_result:
    print("Vector Space Model Result:", vector_space_result)
else:
    print("No matching documents found for the Vector Space Model.")
```

Practical:- 2(COMBINED)

```
import nltk
```

```
nltk.download('punkt')
from collections import defaultdict
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
class SearchEngine:
    def __init__(self, documents):
        self.documents = documents
        self.boolean_model = self.build_boolean_model()
        self.vector_space_model = self.build_vector_space_model()
    def build_boolean_model(self):
        inverted_index = defaultdict(set)
        for doc_id, document in enumerate(self.documents):
            terms = set(word_tokenize(document.lower()))
            for term in terms:
                inverted_index[term].add(doc_id)
        return inverted_index
    def boolean_search(self, query):
        query_terms = set(word_tokenize(query.lower()))
        result = set(range(len(self.documents)))
        for term in query_terms:
            result = result.intersection(self.boolean_model.get(term, set()))
            if not result:
                break
        return result
    def build_vector_space_model(self):
        vectorizer = TfidfVectorizer(stop_words='english')
        tf_idf_matrix = vectorizer.fit_transform(self.documents)
        return vectorizer, tf_idf_matrix
    def vector_space_search(self, query):
        query_vector = self.vector_space_model[0].transform([query])
        cosine_similarities = cosine_similarity(query_vector, self.vector_space_model[1]).flatten()
        ranked_documents = sorted(enumerate(cosine_similarities), key=lambda x: x[1],
reverse=True)
        return [doc_id for doc_id, similarity in ranked_documents]

documents = [
```

```
    "This is a sample document about Python programming.",
    "Python is a widely used programming language.",
    "Document retrieval is important in information systems."
]
search_engine = SearchEngine(documents)

boolean_result = search_engine.boolean_search("Python programming")
if boolean_result:
    print("Boolean Retrieval Result:", boolean_result)
else:
    print("No matching documents found for the Boolean query.")

vector_space_result = search_engine.vector_space_search("Python used")
if vector_space_result:
    print("Vector Space Model Result:", vector_space_result)
else:
    print("No matching documents found for the Vector Space Model.")
```

****
****

Practical : - 3

Spelling Correction in IR Systems

a) Develop a spelling correction module using edit distance algorithms.

b) Integrate the spelling correction module into an information retrieval system.

(METHOD-1)

```
'''This program creates a simple spelling correction module (SpellingCorrection) using the
Levenshtein distance algorithm
and then integrates it into an information retrieval system (InformationRetrievalSystem).
The suggest_correction method in the spelling correction module provides a corrected word
based on the minimum edit distance and word frequency.
'''
import nltk
nltk.download('punkt')
from nltk.metrics import edit_distance
from collections import defaultdict
class SpellingCorrection:
    def __init__(self, documents):
        self.corpus = [word.lower() for document in documents for word in
```

```
nltk.word_tokenize(document)]
        self.word_frequency = defaultdict(int)
        self.build_word_frequency()
    def build_word_frequency(self):
        for word in self.corpus:
            self.word_frequency[word] += 1
    def suggest_correction(self, query_word):
        suggestions = []
        for word in self.word_frequency:
            distance = edit_distance(query_word, word)
            suggestions.append((word, distance))

        suggestions.sort(key=lambda x: (x[1], -self.word_frequency[x[0]]))
        return suggestions[0][0] if suggestions else query_word
class InformationRetrievalSystem:
    def __init__(self, documents):
        self.documents = documents
        self.spelling_correction = SpellingCorrection(documents)
    def search(self, query):
        corrected_query = ' '.join([self.spelling_correction.suggest_correction(word.lower()) for
word in nltk.word_tokenize(query)])
        print("Corrected Query:", corrected_query)

        print("Search Results:")
        for doc in self.documents:
            print(doc)

documents = [
    "This is a sample document about Python programming.",
    "Python is a widely used programming language.",
    "Document retrieval is important in information systems."
]
ir_system = InformationRetrievalSystem(documents)

query = "pythton progrmming"
ir_system.search(query)
```

****

(METHOD-2)

```
'''
In this code, the InformationRetrievalSystem class uses the TF-IDF vector space model for
search.
It corrects the query using the SpellingCorrection module and then calculates the
cosine similarities between the corrected query and all documents.
Finally, it ranks the documents by similarity and prints the search results.'''

import nltk
nltk.download('punkt')
from nltk.metrics import edit_distance
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from collections import defaultdict
class SpellingCorrection:
    def __init__(self, documents):
        self.corpus = [word.lower() for document in documents for word in
nltk.word_tokenize(document)]
        self.word_frequency = defaultdict(int)
        self.build_word_frequency()
    def build_word_frequency(self):
        for word in self.corpus:
            self.word_frequency[word] += 1
    def suggest_correction(self, query_word):
        suggestions = []
        for word in self.word_frequency:
            distance = edit_distance(query_word, word)
            suggestions.append((word, distance))

        suggestions.sort(key=lambda x: (x[1], -self.word_frequency[x[0]]))
        return suggestions[0][0] if suggestions else query_word
class InformationRetrievalSystem:
    def __init__(self, documents):
        self.documents = documents
        self.spelling_correction = SpellingCorrection(documents)
        self.vectorizer, self.tf_idf_matrix = self.build_vector_space_model()
    def build_vector_space_model(self):
        vectorizer = TfidfVectorizer(stop_words='english')
        tf_idf_matrix = vectorizer.fit_transform(self.documents)
        return vectorizer, tf_idf_matrix
```

```
    def search(self, query):
        corrected_query = ' '.join([self.spelling_correction.suggest_correction(word.lower()) for
word in nltk.word_tokenize(query)])
        print("Corrected Query:", corrected_query)

        query_vector = self.vectorizer.transform([corrected_query])

        cosine_similarities = cosine_similarity(query_vector, self.tf_idf_matrix).flatten()

        ranked_documents = sorted(enumerate(cosine_similarities), key=lambda x: x[1],
reverse=True)
        print("Search Results:")
        for idx, similarity in ranked_documents:
            print(f"Document {idx + 1}: Similarity = {similarity:.4f}")
            print(self.documents[idx])

documents = [
    "This is a sample document about Python programming.",
    "Python is a widely used programming language.",
    "Document retrieval is important in information systems."
]
ir_system = InformationRetrievalSystem(documents)

query = "pythton progrmming"
ir_system.search(query)
```

****
****

Practical:- 4

Evaluation Metrics for IR Systems

Calculate precision, recall, and F-measure for a given set of retrieval results.

Use an evaluation toolkit to measure average precision and other evaluation metrics.

```
from sklearn.metrics import precision_score, recall_score, f1_score, average_precision_score,
precision_recall_curve, roc_auc_score
import numpy as np

true_labels = [1, 0, 1, 1, 0, 1, 0, 0, 1, 0]  # Ground truth relevance labels
predicted_labels = [1, 1, 0, 1, 1, 0, 1, 0, 0, 1]  # Predicted relevance labels
scores = np.array([0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.0])  # Predicted relevance scores
```

```
precision = precision_score(true_labels, predicted_labels)
recall = recall_score(true_labels, predicted_labels)
f1 = f1_score(true_labels, predicted_labels)
print("Precision:", precision)
print("Recall:", recall)
print("F-measure:", f1)
average_precision = average_precision_score(true_labels, predicted_labels)
print("\nAverage Precision:", average_precision)

precision, recall, _ = precision_recall_curve(true_labels, scores)

roc_auc = roc_auc_score(true_labels, scores)
print("\nROC-AUC Score:", roc_auc)
print("\nPrecision-Recall Curve Points:")
for p, r in zip(precision, recall):
    print(f"Precision: {p:.2f}, Recall: {r:.2f}")
```

****

Practical: - 5

Text Categorization

Implement a text classification algorithm (e.g., Naive Bayes or Support Vector Machines).

Train the classifier on a labelled dataset and evaluate its performance.

```
'''Text Categorization
Implement a text classification algorithm (e.g., Support Vector Machines or Naive Bayes).
Train the classifier on a labelled dataset and evaluate its performance.'''
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
'''from sklearn.naive_bayes import MultinomialNB'''
from sklearn.metrics import classification_report, accuracy_score
from sklearn.datasets import fetch_20newsgroups
data = fetch_20newsgroups(subset='all', remove=('headers', 'footers', 'quotes'))
documents = data.data
labels = data.target

X_train, X_test, y_train, y_test = train_test_split(documents, labels, 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)

svm_classifier = SVC(kernel='linear')  # You can try different kernels (e.g., 'rbf') and
parameters
svm_classifier.fit(X_train_tfidf, y_train)
'''# Train the classifier (Multinomial Naive Bayes)
naive_bayes_classifier = MultinomialNB()
naive_bayes_classifier.fit(X_train_tfidf, y_train)'''

y_pred = svm_classifier.predict(X_test_tfidf)
'''y_pred = naive_bayes_classifier.predict(X_test_tfidf)'''

accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
classification_rep = classification_report(y_test, y_pred, target_names=data.target_names)
print("Classification Report:")
print(classification_rep)
```

****

Practical:- 6

Clustering for Information Retrieval

Implement a clustering algorithm (e.g., K-means or hierarchical clustering).

Apply the clustering algorithm to a set of documents and evaluate the clustering results.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.datasets import fetch_20newsgroups
import matplotlib.pyplot as plt
from sklearn.decomposition import TruncatedSVD

data = fetch_20newsgroups(subset='all', remove=('headers', 'footers', 'quotes'))
documents = data.data
vectorizer = TfidfVectorizer(stop_words='english')
tfidf_matrix = vectorizer.fit_transform(documents)
```

```
num_clusters = 5  # You can adjust the number of clusters based on your needs
kmeans = KMeans(n_clusters=num_clusters, n_init=10, random_state=42)  # Set n_init
explicitly to avoid the warning
cluster_assignments = kmeans.fit_predict(tfidf_matrix)
silhouette_avg = silhouette_score(tfidf_matrix, cluster_assignments)
print(f"Silhouette Score: {silhouette_avg}")

print("\nCluster Assignments (Showing only the first 10 documents):")
for doc_id, cluster_id in enumerate(cluster_assignments[:10]):
    print(f"Document {doc_id + 1} -> Cluster {cluster_id + 1}")

svd = TruncatedSVD(n_components=2, random_state=42)
tfidf_matrix_reduced = svd.fit_transform(tfidf_matrix)
plt.figure(figsize=(10, 6))
colors = ['red', 'blue', 'green', 'purple', 'orange']  # Adjust colors based on the number of
clusters
for cluster_id in range(num_clusters):
    cluster_points = tfidf_matrix_reduced[cluster_assignments == cluster_id]
    plt.scatter(cluster_points[:, 0], cluster_points[:, 1], label=f'Cluster {cluster_id + 1}',
color=colors[cluster_id], alpha=0.7)
plt.title('K-means Clustering of Documents')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.show()
```

****
****

Practical: - 7

Web Crawling and Indexing

Develop a web crawler to fetch and index web pages.

Handle challenges such as robots.txt, dynamic content, and crawling delays.

```
import requests
from bs4 import BeautifulSoup
import time
def fetch_page(url):
    try:
        response = requests.get(url)
        if response.status_code == 200:
```

```
            return response.text
        else:
            print(f"Failed to fetch page: {url}")
            return None
    except Exception as e:
        print(f"Error fetching page: {e}")
        return None
def parse_page(html):
    soup = BeautifulSoup(html, 'html.parser')
    # Extract links from the page
    links = [link.get('href') for link in soup.find_all('a', href=True)]
    print("Links:")
    for link in links:
        print(link)
    images = [image.get('src') for image in soup.find_all('img', src=True)]
    print("\nImages:")
    for image in images:
        print(image)
    metadata = {meta.get('name'): meta.get('content') for meta in soup.find_all('meta', attrs=
{'name': True})}
    print("\nMetadata:")
    for name, content in metadata.items():
        print(f"{name}: {content}")

def crawl(start_url, max_pages):
    visited = set()
    queue = [start_url]
    while queue and len(visited) < max_pages:
        url = queue.pop(0)
        if url in visited:
            continue
        html = fetch_page(url)
        if html:
            print(f"\nParsing page: {url}")
            parse_page(html)
            visited.add(url)

            soup = BeautifulSoup(html, 'html.parser')
            links = soup.find_all('a', href=True)
```

```
        for link in links:
            new_url = link.get('href')
            if new_url and new_url.startswith('http') and new_url not in visited:
                queue.append(new_url)

        time.sleep(1)

start_url = 'https://google.com'
max_pages = 1
crawl(start_url, max_pages)
```

****
****

Practical: - 8

Link Analysis and PageRank

Implement the PageRank algorithm to rank web pages based on link analysis.

Apply the PageRank algorithm to a small web graph and analyze the results.

```
import numpy as np
def pagerank(adj_matrix, damping_factor=0.85, max_iter=100, tol=1e-6):

    num_pages = adj_matrix.shape[0]
    pr = np.ones(num_pages) / num_pages

    for _ in range(max_iter):
        pr_new = np.zeros(num_pages)
        for i in range(num_pages):
            for j in range(num_pages):
                if adj_matrix[j, i] != 0:
                    pr_new[i] += pr[j] / np.sum(adj_matrix[j])
        pr_new = damping_factor * pr_new + (1 - damping_factor) / num_pages

        if np.linalg.norm(pr_new - pr) < tol:
            break
        pr = pr_new
    return pr

adj_matrix = np.array([
    [0, 1, 0, 0],
    [0, 0, 1, 0],
```

```
    [0, 0, 0, 1],
    [1, 0, 0, 0]
])

page_ranks = pagerank(adj_matrix)

for i, pr in enumerate(page_ranks):
    print(f"Page {chr(65 + i)}: {pr:.4f}")
```

****

Practical:- 9

Learning to Rank

Implement a learning to rank algorithm (e.g., RankSVM or RankBoost).

Train the ranking model using labelled data and evaluate its effectiveness.

```
from sklearn.datasets import load_svmlight_file
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import mean_squared_error

X, y = load_svmlight_file("path...\\labelled_data-2.txt")

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

rank_svm = SVC(kernel='linear')
rank_svm.fit(X_train, y_train)

train_score = rank_svm.score(X_train, y_train)
test_score = rank_svm.score(X_test, y_test)
print("Training Accuracy:", train_score)
print("Testing Accuracy:", test_score)

train_predictions = rank_svm.predict(X_train)
test_predictions = rank_svm.predict(X_test)
train_mse = mean_squared_error(y_train, train_predictions)
test_mse = mean_squared_error(y_test, test_predictions)
print("Training Score:", train_score)
print("Testing Score:", test_score)
```

```
print("Training MSE:", train_mse)
print("Testing MSE:", test_mse)
```

****
****
****

Advanced Topics in Information Retrieval

Implement a text summarization algorithm (e.g., extractive or abstractive).

Build a question-answering system using techniques such as information

Practical-10(A):

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
def extractive_summarize(text, num_sentences=3):
    sentences = text.split('.')  # Split text into sentences
    num_sentences = min(num_sentences, len(sentences))


    count_vectorizer = CountVectorizer(stop_words='english')
    count_matrix = count_vectorizer.fit_transform(sentences)


    tfidf_transformer = TfidfTransformer()
    tfidf_matrix = tfidf_transformer.fit_transform(count_matrix)


    similarity_matrix = cosine_similarity(tfidf_matrix, tfidf_matrix)


    scores = np.sum(similarity_matrix, axis=1)


    ranked_sentences = [sentences[idx] for idx in np.argsort(scores)[::-1]]


    summary = '. '.join(ranked_sentences[:num_sentences])

    return summary
# Sample text
text = "Text summarization is the process of distilling the most important information from a
```

source (or sources) to produce a concise summary. Extractive summarization methods select the most representative sentences directly from the source text. They do not generate new sentences but rather choose the most significant ones. In this implementation, we'll use an extractive approach to summarize text using cosine similarity between sentence embeddings. GlassFish is an open-source Jakarta EE platform application server project started by Sun Microsystems, then sponsored by Oracle Corporation, and now living at the Eclipse Foundation and supported by OmniFish, Fujitsu and Payara.[2] The supported version under Oracle was called Oracle GlassFish Server. GlassFish is free software and was initially dual-licensed under two free software licences: the Common Development and Distribution License (CDDL) and the GNU General Public License (GPL) with the Classpath exception. After having been transferred to Eclipse, GlassFish remained dual-licensed, but the CDDL license was replaced by the Eclipse Public License (EPL).[3]"
summary = extractive_summarize(text)
print(summary)

Practical 10(B):

```
#pip install transformers
from transformers import pipeline
# Load SQuAD (Stanford Question Answering Dataset) model (default model name =
"distilbert-base-cased-distilled-squad", revision = "626af31")
qa_pipeline = pipeline("question-answering")
# Context (a short paragraph from Wikipedia about NLP)
context = """Natural language processing (NLP) is a subfield of linguistics,
computer science, information engineering, and artificial intelligence concerned with the
interactions between computers and human language, in particular how to program computers
to
process and analyze large amounts of natural language data."""
# Ask a question
question = "What is Natural Language Processing?"
result = qa_pipeline(question=question, context=context)
# Print the answer
print(f"Question: {question}")
print(f"Answer: {result['answer']}")
```

Practical 10(COMBINED):

```
#pip install spacy
#python -m spacy download en_core_web_sm
import spacy
```

```
# Load English language model
nlp = spacy.load("en_core_web_sm")
# Sample text data
text = """
William Shakespeare was an English playwright, poet,
and actor, widely regarded as the greatest writer in the English language
and the world's greatest dramatist. He is often called
England's national poet and the "Bard of Avon". His extant works,
including collaborations, consist of some 39 plays, 154 sonnets,
three long narrative poems, and a few other verses, some of uncertain
authorship. His plays have been translated into every major living language
and are performed more often than those of any other playwright.
"""
# Function to answer questions using information extraction
def answer_question(question, text):
    doc = nlp(text)
    answer = None
    if question.lower().startswith("who"):
        for entity in doc.ents:
            if entity.label_ == "PERSON":
                answer = entity.text
                break
    elif question.lower().startswith("where"):
        for entity in doc.ents:
            if entity.label_ == "GPE":
                answer = entity.text
                break
    elif question.lower().startswith("how"):
        for token in doc:
            if token.pos_ == "NUM":
                answer = token.text
                break
    elif question.lower().startswith("when"):
        for entity in doc.ents:
            if entity.label_ == "DATE":
                answer = entity.text
                break
    return answer
```

```
# User input loop
while True:
    user_input = input("Please enter your question (or 'exit' to quit): ").strip()
    if user_input.lower() == "exit":
        print("Exiting the program...")
        break
    else:
        answer = answer_question(user_input, text)
        if answer:
            print("Answer:", answer)
        else:
            print("Sorry, I don't know the answer.")
```