List Of Lab Questions

1. **Implement Naïve Bayes classifier for document classification. You should take any dataset related to this task.**

CODE:

```
import os
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.pipeline import Pipeline

def load_documents(folder_path, category):
    documents = []
    for filename in os.listdir(folder_path):
        with open(os.path.join(folder_path, filename), 'r', encoding='utf-8') as file:
            content = file.read()
            documents.append((content, category))
    return documents

if __name__ == "__main__":
    # Replace 'your_folder_path' with the path to the folder containing your text documents
    folder_path_positive = '/content/drive/MyDrive/data'
    folder_path_negative = '/content/drive/MyDrive/data'

    # Load positive and negative documents
    positive_documents = load_documents(folder_path_positive, 'positive')
    negative_documents = load_documents(folder_path_negative, 'negative')

    # Combine positive and negative documents
    all_documents = positive_documents + negative_documents

    # Split the data into training and testing sets
    documents_train, documents_test, categories_train, categories_test = train_test_split(
        [doc[0] for doc in all_documents],
        [doc[1] for doc in all_documents],
        test_size=0.2,
        random_state=42
    )

    # Create a Naive Bayes classifier pipeline
    text_clf = Pipeline([
        ('vect', CountVectorizer()),
```

```
    ('tfidf', TfidfTransformer()),
    ('clf', MultinomialNB())
])

# Train the classifier
text_clf.fit(documents_train, categories_train)

# Make predictions on the test set
predicted = text_clf.predict(documents_test)

# Print classification report
print(metrics.classification_report(categories_test, predicted))

# Print confusion matrix
print("Confusion Matrix:")
print(metrics.confusion_matrix(categories_test, predicted))
```

2. Implement k-Nearest Neighbors classifier for document classification. You should take any dataset related to this task.

CODE:

```
import os
import glob
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report

def load_data(folder_path):
    documents = []
    labels = []
    for file_path in glob.glob(os.path.join(folder_path, '*.txt')):
        with open(file_path, 'r', encoding='utf-8') as file:
            content = file.read()
            documents.append(content)
            # Extract label from the filename
            labels.append(file_path.split('/')[-1].split('_')[0])  # Assuming labels are before the first underscore
    return documents, labels

folder_path = '/content/drive/MyDrive/data'
documents, labels = load_data(folder_path)

vectorizer = TfidfVectorizer(stop_words='english')
X = vectorizer.fit_transform(documents)

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

k = 5
knn_classifier = KNeighborsClassifier(n_neighbors=k)
knn_classifier.fit(X_train, y_train)

predictions = knn_classifier.predict(X_test)

accuracy = accuracy_score(y_test, predictions)
print(f'Accuracy: {accuracy:.2f}')

print('Classification Report:')
print(classification_report(y_test, predictions))
```

3. Implement Support vector machine classifier for document classification. You should take any dataset related to this task.

CODE:

```
import os
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report

# Step 1: Load and Preprocess Data

# Load text data from your documents
document_directory = "/content/drive/MyDrive/data"

documents = []  # List to store document texts
labels = []     # List to store labels (0 or 1 for negative and positive, respectively)
query = "biology"  # Replace with your query

for filename in os.listdir(document_directory):
   if filename.endswith(".txt"):
      with open(os.path.join(document_directory, filename), "r", encoding="utf-8") as file:
         doc_text = file.read()
         documents.append(doc_text)
         # Determine the label based on relevance to the query
         label = 1 if query in doc_text else 0
         labels.append(label)

# Step 2: Split the Data
X_train, X_test, y_train, y_test = train_test_split(documents, labels, test_size=0.2, random_state=42)

# Step 3: Preprocess and Vectorize Data

# Create a TF-IDF vectorizer
vectorizer = TfidfVectorizer(max_features=1000)  # You can adjust the number of features as needed
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)

# Step 4: Train an SVC Model

# Create and train an SVC model
svc_classifier = SVC(kernel='linear')
svc_classifier.fit(X_train_tfidf, y_train)
```

```
# Step 5: Make Predictions and Evaluate the Model

# Make predictions on the test set
y_pred = svc_classifier.predict(X_test_tfidf)

# Calculate and print the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Print a classification report for precision, recall, and F1-score
print('Classification Report:')
print(classification_report(y_test, y_pred))
```

4. Implement Decision trees algorithm for document classification. You should take any dataset related to this task.

CODE:

```
import os
import re
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

document_dir = '/content/drive/MyDrive/data'
document_list = []

for filename in os.listdir(document_dir):
   with open(os.path.join(document_dir, filename), 'r', encoding='utf-8') as file:
      document_list.append(file.read())

def preprocess_text(text):
   text = text.lower()
   text = re.sub(r'[^a-z\s]', '', text)  # Remove non-alphabetic characters
   text = text.split()
   return ' '.join(text)

preprocessed_documents = [preprocess_text(doc) for doc in document_list]

X = preprocessed_documents
y = np.arange(len(X)) // (len(X) // num_classes)  # Assign labels based on the number of classes

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

vectorizer = TfidfVectorizer()
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)

decision_tree_classifier = DecisionTreeClassifier()
decision_tree_classifier.fit(X_train_tfidf, y_train)

y_pred = decision_tree_classifier.predict(X_test_tfidf)

accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")
```

5. Implement Rocchio classification algorithm to classify the relevant documents from given collection of documents for a given query. You should take any dataset related to this task.

CODE:

```
import os
import re
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.neighbors import NearestCentroid
document_dir = '/content/drive/MyDrive/data'
document_list = []

for filename in os.listdir(document_dir):
   with open(os.path.join(document_dir, filename), 'r', encoding='utf-8') as file:
      document_list.append(file.read())
def preprocess_text(text):
   text = text.lower()
   text = re.sub(r'[^a-z\s]', '', text)  # Remove non-alphabetic characters
   text = text.split()
   return ' '.join(text)

preprocessed_documents = [preprocess_text(doc) for doc in document_list]num_classes = 3  # Replace
with the actual number of classes

# Assign labels based on the number of classes
y = np.arange(len(X)) // (len(X) // num_classes)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
vectorizer = TfidfVectorizer()
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)
rocchio_classifier = NearestCentroid()
rocchio_classifier.fit(X_train_tfidf, y_train)

y_pred = rocchio_classifier.predict(X_test_tfidf)

from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")
```

6. Implement k-means clustering algorithm to divide the given collection of text documents into clusters.

CODE:

```
import os
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

def load_documents(folder_path):
    documents = []
    for filename in os.listdir(folder_path):
        with open(os.path.join(folder_path, filename), 'r', encoding='utf-8') as file:
            content = file.read()
            documents.append(content)
    return documents

def preprocess_documents(documents):
    # Create a TF-IDF vectorizer
    vectorizer = TfidfVectorizer(stop_words='english')

    # Transform the documents to TF-IDF vectors
    tfidf_matrix = vectorizer.fit_transform(documents)

    return tfidf_matrix

def k_means_clustering(tfidf_matrix, n_clusters):
    # Perform k-means clustering
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    cluster_labels = kmeans.fit_predict(tfidf_matrix)

    return cluster_labels

if __name__ == "__main__":
    # Replace 'your_folder_path' with the path to the folder containing your text documents
    folder_path = '/content/drive/MyDrive/data'
    documents = load_documents(folder_path)

    # Preprocess documents
    tfidf_matrix = preprocess_documents(documents)

    # Replace 'your_number_of_clusters' with the desired number of clusters
```

```
num_clusters = 2

# Perform k-means clustering
cluster_labels = k_means_clustering(tfidf_matrix, num_clusters)

# Print the document clusters
for cluster_id in range(num_clusters):
    print(f"Cluster {cluster_id + 1}:")
    cluster_indices = np.where(cluster_labels == cluster_id)[0]
    for index in cluster_indices:
        print(f"  Document {index + 1}")
    print("\n")
```

7. Implement Agglomerative clustering algorithm to divide the given collection of text documents into clusters.

CODE:

```
import os
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

def load_documents(folder_path):
    documents = []
    for filename in os.listdir(folder_path):
        with open(os.path.join(folder_path, filename), 'r', encoding='utf-8') as file:
            content = file.read()
            documents.append(content)
    return documents

def preprocess_documents(documents):
    # Create a TF-IDF vectorizer
    vectorizer = TfidfVectorizer(stop_words='english')

    # Transform the documents to TF-IDF vectors
    tfidf_matrix = vectorizer.fit_transform(documents)

    return tfidf_matrix

def hierarchical_clustering(tfidf_matrix, n_clusters):
    # Compute pairwise cosine similarity
    pairwise_similarity = cosine_similarity(tfidf_matrix)

    # Apply Agglomerative Clustering
    clustering = AgglomerativeClustering(n_clusters=n_clusters, affinity='precomputed', linkage='average')
    cluster_labels = clustering.fit_predict(1 - pairwise_similarity)  # 1 minus similarity to convert to
distance

    return cluster_labels

if __name__ == "__main__":
    # Replace 'your_folder_path' with the path to the folder containing your text documents
    folder_path = '/content/drive/MyDrive/data'
    documents = load_documents(folder_path)

    # Preprocess documents
```

```
tfidf_matrix = preprocess_documents(documents)

# Replace 'your_number_of_clusters' with the desired number of clusters
num_clusters = 5

# Perform Agglomerative Clustering
cluster_labels = hierarchical_clustering(tfidf_matrix, num_clusters)

# Print the document clusters
for cluster_id in range(num_clusters):
    print(f"Cluster {cluster_id + 1}:")
    cluster_indices = np.where(cluster_labels == cluster_id)[0]
    for index in cluster_indices:
        print(f"  Document {index + 1}")
    print("\n")
```

8. Implement Divisive clustering algorithm to divide the given collection of text documents into clusters.

CODE:

```
import os
import nltk
import string
import re
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans
nltk.download('punkt')
nltk.download('stopwords')
def read_documents_from_directory(directory):
    documents = []
    for filename in os.listdir(directory):
        if filename.endswith(".txt"):
            with open(os.path.join(directory, filename), "r") as file:
                text = file.read()
                documents.append(text)
    return documents
def preprocess_text(text):
    # Tokenization
    words = nltk.word_tokenize(text)

    # Remove punctuation and lowercase
    words = [word.lower() for word in words if word.isalpha()]

    # Remove stopwords
    stopwords = set(nltk.corpus.stopwords.words('english'))
    words = [word for word in words if word not in stopwords]

    # Stemming (you can use other stemming methods as well)
    stemmer = nltk.PorterStemmer()
    words = [stemmer.stem(word) for word in words]

    return " ".join(words)

document_directory = "/content/drive/MyDrive/data"
documents = read_documents_from_directory(document_directory)
preprocessed_documents = [preprocess_text(doc) for doc in documents]
vectorizer = TfidfVectorizer()
tfidf_matrix = vectorizer.fit_transform(preprocessed_documents)
n_clusters = 3  # You can set the desired number of clusters here
```

```
kmeans = KMeans(n_clusters=n_clusters, random_state=0)
cluster_labels = kmeans.fit_predict(tfidf_matrix)
for doc_index, cluster_label in enumerate(cluster_labels):
    print(f"Document {doc_index + 1} is in Cluster {cluster_label}")
```

9. Implement a Web Crawler.

CODE:

```
!pip install requests beautifulsoup4

import requests
from bs4 import BeautifulSoup

def crawl_web(url, depth=3):
    visited_urls = set()
    crawl_queue = [(url, 0)]

    while crawl_queue:
        current_url, current_depth = crawl_queue.pop(0)

        if current_url not in visited_urls and current_depth <= depth:
            try:
                # Fetch the web page
                response = requests.get(current_url)
                if response.status_code == 200:
                    # Parse the HTML content
                    soup = BeautifulSoup(response.text, 'html.parser')

                    # Extract information or perform desired actions here
                    # For example, print the title of the page
                    print(f"Depth {current_depth}: {soup.title.text}")

                    # Add links to the crawl queue for further exploration
                    links = soup.find_all('a', href=True)
                    for link in links:
                        next_url = link['href']
                        crawl_queue.append((next_url, current_depth + 1))

                    # Mark the current URL as visited
                    visited_urls.add(current_url)

            except Exception as e:
                print(f"Error processing {current_url}: {e}")

if _name_ == "_main_":
    # Start crawling from a given URL
    starting_url = "link of website"
    crawl_web(starting_url)
```

10. Implement a program to perform the following sequence of tasks on a given text document.

a. **Tokenization**
b. **Stemming**
c. **Stop words removal.**
d. **Inverted index construction**

CODE:

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from collections import defaultdict

nltk.download('punkt')
nltk.download('stopwords')

def tokenize(text):
    # Tokenization
    tokens = word_tokenize(text.lower())
    return tokens

def stem(tokens):
    # Stemming using Porter Stemmer
    stemmer = PorterStemmer()
    stemmed_tokens = [stemmer.stem(token) for token in tokens]
    return stemmed_tokens

def remove_stopwords(tokens):
    # Stop words removal
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [token for token in tokens if token not in stop_words]
    return filtered_tokens

def build_inverted_index(doc_id, tokens, inverted_index):
    # Inverted index construction
    for position, token in enumerate(tokens):
        inverted_index[token].append((doc_id, position))

if __name__ == "__main__":
    # Replace 'your_document_path' with the actual path to your text document
    document_path = '/content/drive/MyDrive/data/1.txt'

    with open(document_path, 'r', encoding='utf-8') as file:
```

```
    document_content = file.read()

# Tokenization
tokens = tokenize(document_content)

# Stemming
stemmed_tokens = stem(tokens)

# Stop words removal
filtered_tokens = remove_stopwords(stemmed_tokens)

# Inverted index construction
inverted_index = defaultdict(list)
build_inverted_index(1, filtered_tokens, inverted_index)

# Print the results
print("Original Tokens:")
print(tokens)
print("\nStemmed Tokens:")
print(stemmed_tokens)
print("\nTokens after Stop Words Removal:")
print(filtered_tokens)
print("\nInverted Index:")
for term, postings in inverted_index.items():
    print(f"{term}: {postings}")
```

11. Write a program to compute the TF-IDF weighted vectors for each document in the given collection. (Note: You should collect a list of minimum 10 text documents earlier)

CODE:

```
!pip install scikit-learn

import os
from sklearn.feature_extraction.text import TfidfVectorizer

def load_documents(folder_path):
    documents = []
    for filename in os.listdir(folder_path):
        with open(os.path.join(folder_path, filename), 'r', encoding='utf-8') as file:
            content = file.read()
            documents.append(content)
    return documents

def compute_tfidf_vectors(documents):
    # Create a TfidfVectorizer
    vectorizer = TfidfVectorizer(stop_words='english')

    # Fit and transform the documents
    tfidf_matrix = vectorizer.fit_transform(documents)

    # Get feature names (words)
    feature_names = vectorizer.get_feature_names_out()

    # Convert the TF-IDF matrix to a list of dictionaries
    tfidf_vectors = []
    for i in range(len(documents)):
        feature_index = tfidf_matrix[i, :].nonzero()[1]
        tfidf_scores = zip(feature_index, [tfidf_matrix[i, x] for x in feature_index])
        tfidf_vectors.append(dict((feature_names[i], score) for i, score in tfidf_scores))

    return tfidf_vectors

if __name__ == "__main__":
    # Replace 'your_folder_path' with the path to the folder containing your text documents
    folder_path = '/content/drive/MyDrive/data'
    documents = load_documents(folder_path)

    if not documents:
        print("No documents found in the specified folder.")
    else:
        tfidf_vectors = compute_tfidf_vectors(documents)
```

```
# Print TF-IDF vectors for each document
for i, vector in enumerate(tfidf_vectors):
    print(f"Document {i + 1} TF-IDF Vector:")
    print(vector)
    print("\n")
```

12. **Write a program to compute the cosine similarity between given query Q and each text document in a collection of documents (Note: You should collect a list of minimum 10 text documents earlier).**

CODE:

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


def load_documents(folder_path):
    documents = []
    for filename in os.listdir(folder_path):
        with open(os.path.join(folder_path, filename), 'r', encoding='utf-8') as file:
            content = file.read()
            documents.append(content)
    return documents


def compute_cosine_similarity(query, documents):
    # Combine the query and documents
    all_text = [query] + documents


    # Create a TF-IDF vectorizer
    vectorizer = TfidfVectorizer()


    # Transform the documents to TF-IDF vectors
    tfidf_matrix = vectorizer.fit_transform(all_text)


    # Compute cosine similarity between the query and each document
    cosine_similarities = cosine_similarity(tfidf_matrix[0:1], tfidf_matrix[1:]).flatten()
```

```
    return cosine_similarities

if __name__ == "__main__":
    # Replace 'your_folder_path' with the path to the folder containing your text documents
    folder_path = '/content/drive/MyDrive/data'
    documents = load_documents(folder_path)

    # Replace 'your_query' with the actual query text
    query = 'your query'

    # Compute cosine similarity
    similarities = compute_cosine_similarity(query, documents)

    # Print the cosine similarity for each document
    for i, similarity in enumerate(similarities):
        print(f"Similarity with Document {i + 1}: {similarity}")
```