Vector Space Model:

import os

import math

from nltk.tokenize import word\_tokenize

from nltk.stem import PorterStemmer

from nltk.corpus import stopwords

import nltk

nltk.download('punkt')

nltk.download('stopwords')

def preprocess\_text(text):

ps = PorterStemmer()

stop\_words = set(stopwords.words('english'))

words = word\_tokenize(text)

# Preprocess: lowercase, remove stopwords, and apply stemming

preprocessed = [ps.stem(word.lower()) for word in words if word.isalnum() and word.lower() not in stop\_words]

return preprocessed

def build\_index(file\_path, doc\_id, index, raw\_tf):

with open(file\_path, 'r') as file:

text = file.read().strip()

tokens = preprocess\_text(text)

# Compute raw term frequency (count) and update the index

term\_count = {}

for word in tokens:

term\_count[word] = term\_count.get(word, 0) + 1

for term, count in term\_count.items():

# Update raw term frequency (count)

if term not in raw\_tf:

raw\_tf[term] = {}

raw\_tf[term][doc\_id] = count

# Update Index

if term not in index:

index[term] = []

if doc\_id not in index[term]:

index[term].append(doc\_id)

def compute\_tf(raw\_tf):

tf = {}

for term, doc\_counts in raw\_tf.items():

tf[term] = {}

for doc\_id, count in doc\_counts.items():

if count > 0:

tf[term][doc\_id] = 1 + math.log2(count)

else:

tf[term][doc\_id] = 0

return tf

def compute\_idf(index, num\_docs):

idf = {}

for term, postings in index.items():

doc\_freq = len(postings)

idf[term] = math.log(num\_docs / doc\_freq, 2) # Log base 2

return idf

def compute\_tf\_idf(tf, idf):

tf\_idf = {}

for term, doc\_tf in tf.items():

tf\_idf[term] = {}

for doc\_id, tf\_value in doc\_tf.items():

tf\_idf[term][doc\_id] = tf\_value \* idf.get(term, 0)

return tf\_idf

def compute\_document\_norms(tf\_idf, num\_docs):

doc\_norms = {doc\_id: 0 for doc\_id in range(num\_docs)}

for term, doc\_weights in tf\_idf.items():

for doc\_id, weight in doc\_weights.items():

doc\_norms[doc\_id] += weight \*\* 2

# Take the square root for the norms

for doc\_id in doc\_norms:

doc\_norms[doc\_id] = math.sqrt(doc\_norms[doc\_id])

return doc\_norms

def preprocess\_query(query):

return preprocess\_text(query)

def calculate\_query\_vector(query\_tokens, tf\_idf, idf):

query\_tf = {}

query\_vector = {}

# Calculate raw TF for query

for token in query\_tokens:

query\_tf[token] = query\_tf.get(token, 0) + 1

# Convert query TF to TF-IDF

for token, count in query\_tf.items():

if token in idf:

tf = 1 + math.log2(count)

query\_vector[token] = tf \* idf[token]

return query\_vector

def calculate\_cosine\_similarity(query\_vector, tf\_idf, doc\_norms):

doc\_similarities = defaultdict(float)

query\_norm = math.sqrt(sum(value \*\* 2 for value in query\_vector.values()))

for term, query\_value in query\_vector.items():

if term in tf\_idf:

for doc\_id, doc\_value in tf\_idf[term].items():

doc\_similarities[doc\_id] += query\_value \* doc\_value # Dot product

# Normalize by query norm and document norms

for doc\_id, dot\_product in doc\_similarities.items():

if query\_norm > 0 and doc\_norms[doc\_id] > 0:

doc\_similarities[doc\_id] /= (query\_norm \* doc\_norms[doc\_id])

else:

doc\_similarities[doc\_id] = 0.0

return sorted(doc\_similarities.items(), key=lambda x: x[1], reverse=True)

# Main execution

input\_dir = "/content/drive/MyDrive/fulldata"

output\_tf\_idf\_file = "/content/drive/MyDrive/output\_dynamic/tf\_idf.txt"

doc\_paths = [os.path.join(input\_dir, file) for file in os.listdir(input\_dir) if file.endswith('.txt')]

# Initialize structures

index = {}

raw\_tf = {}

# Build the index and raw term frequency

for doc\_id, doc\_path in enumerate(doc\_paths):

build\_index(doc\_path, doc\_id, index, raw\_tf)

# Convert raw term frequency to log-based TF

tf = compute\_tf(raw\_tf)

# Compute IDF

num\_docs = len(doc\_paths)

idf = compute\_idf(index, num\_docs)

# Compute TF-IDF

tf\_idf = compute\_tf\_idf(tf, idf)

# Compute Document Norms

doc\_norms = compute\_document\_norms(tf\_idf, num\_docs)

# Query processing and similarity calculation

query = "example query"

query\_tokens = preprocess\_query(query)

query\_vector = calculate\_query\_vector(query\_tokens, tf\_idf, idf)

similarities = calculate\_cosine\_similarity(query\_vector, tf\_idf, doc\_norms)

print("Query:", query)

print("\nSimilarity Scores:")

for doc\_id, score in similarities:

print(f"Document {doc\_id}: {score:.4f}")

Binary Independence Model:

import os

import math

from collections import defaultdict

def preprocess\_text(text):

return text.lower().split()

def build\_index(file\_paths):

index = defaultdict(lambda: defaultdict(int))

doc\_term\_counts = []

for doc\_id, file\_path in enumerate(file\_paths):

with open(file\_path, 'r') as f:

text = f.read()

terms = set(preprocess\_text(text))

for term in terms:

index[doc\_id][term] = 1

return index

def compute\_term\_probabilities(index, num\_docs):

term\_probabilities = {}

terms = set()

for doc in index.values():

for t in doc.keys():

terms.add(t)

for term in terms:

term\_in\_docs = []

for doc\_id, doc in index.items():

if term in doc:

term\_in\_docs.append(doc\_id)

p\_relevant = len(term\_in\_docs) / num\_docs

p\_not\_relevant = 1 - p\_relevant

term\_probabilities[term] = (p\_relevant, p\_not\_relevant)

return term\_probabilities

def score\_document(query\_terms, doc\_id, term\_probabilities, index):

score = 1

for term in query\_terms:

if term in index[doc\_id]: # If term is present in the document

p\_relevant, p\_not\_relevant = term\_probabilities[term]

score \*= p\_relevant / p\_not\_relevant

else: # If term is not in the document

p\_relevant, p\_not\_relevant = term\_probabilities[term]

score \*= (1 - p\_relevant) / (1 - p\_not\_relevant)

return score

# Example usage

input\_dir = "/path/to/documents"

query = "example query"

doc\_paths = [os.path.join(input\_dir, file) for file in os.listdir(input\_dir) if file.endswith('.txt')]

# Build index

index = build\_index(doc\_paths)

num\_docs = len(doc\_paths)

# Compute term probabilities for relevant and non-relevant documents

term\_probabilities = compute\_term\_probabilities(index, num\_docs)

# Query processing

query\_terms = preprocess\_text(query)

# Score all documents

doc\_scores = {}

for doc\_id in index.keys():

doc\_scores[doc\_id] = score\_document(query\_terms, doc\_id, term\_probabilities, index)

# Sort documents by score (higher score = more relevant)

sorted\_docs = sorted(doc\_scores.items(), key=lambda x: x[1], reverse=True)

# Print ranked documents

for doc\_id, score in sorted\_docs:

print(f"Document {doc\_id}: Score = {score:.4f}")

SVM:

import os

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.model\_selection import train\_test\_split

from sklearn.svm import LinearSVC

from sklearn.metrics import classification\_report, accuracy\_score

# Path to the folder containing subfolders for each label

DATA\_FOLDER = "path\_to\_your\_folder"

# Step 1: Load text data and labels

def load\_data(data\_folder):

texts = []

labels = []

for label in os.listdir(data\_folder):

label\_folder = os.path.join(data\_folder, label)

if os.path.isdir(label\_folder):

for filename in os.listdir(label\_folder):

file\_path = os.path.join(label\_folder, filename)

with open(file\_path, 'r', encoding='utf-8') as file:

texts.append(file.read())

labels.append(label)

return texts, labels

documents, labels = load\_data(DATA\_FOLDER)

# Step 2: Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(documents, labels, test\_size=0.3, random\_state=42)

# Step 3: Vectorize the text data using TfidfVectorizer

vectorizer = TfidfVectorizer(stop\_words='english', max\_features=10000) # Adjust features as needed

X\_train\_vectors = vectorizer.fit\_transform(X\_train)

X\_test\_vectors = vectorizer.transform(X\_test)

# Step 4: Train the SVM classifier

classifier = LinearSVC()

classifier.fit(X\_train\_vectors, y\_train)

# Step 5: Make predictions on the test data

y\_pred = classifier.predict(X\_test\_vectors)

# Step 6: Evaluate the model

print("Classification Report:\n", classification\_report(y\_test, y\_pred))

print("Accuracy:", accuracy\_score(y\_test, y\_pred))