Setup our environement

```
%%capture

!pip install cassandra-driver
!pip install ipywidgets
!pip install ipyplot

!pip install cohere
!pip install tiktoken

## used for summarisation step so no need to run now as all modules have been summarised and stored
#!pip install transformers

##for the webpage
!pip install anvil-uplink

import anvil.server


import ipyplot
import os



## For connecting to the database
import pandas as pd #just using this for ipy output formating

!pip install cassandra-driver
from cassandra.cluster import Cluster
from cassandra.auth import PlainTextAuthProvider
from cassandra import ConsistencyLevel
from cassandra.query import dict_factory
import ipywidgets as widgets
import requests
import IPython

## For mounting google drive
from google.colab import drive
import chardet

## For similarity search functions
from scipy.spatial import distance
import numpy as np
from sklearn.preprocessing import normalize
from sklearn.metrics.pairwise import cosine_similarity

## For training the model
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
from sklearn.model_selection import train_test_split

## For attempting retries
import time

## For inserting documents with queries
from cassandra.query import BoundStatement
from cassandra.query import BatchStatement

## For evaluation metrics
from sklearn.metrics import accuracy_score, precision_score


## For barts version of summarisation step (don't need to run now as all modules have been summarised and stored)
#import re
#from transformers import BartForConditionalGeneration, BartTokenizer

## For evaluating the hyperparamter grid to find the best possible set up for the model
from sklearn.model_selection import ParameterGrid
from sklearn.preprocessing import MultiLabelBinarizer
from itertools import islice
import IPython.display as display
import shutil
import pickle


## Setting the random seed for the runtime to ensure reproducibility of results
import random
import tensorflow as tf
```

```
## Set the random seed for Python's random module
random.seed(42)

## Set the random seed for NumPy
np.random.seed(42)

## Set the random seed for TensorFlow
tf.random.set_seed(42)
```

Setup AstraDB Connection

Setup our AstraDB Credentials

```
ASTRA_CLIENT_ID: '' Insert text here ''
ASTRA_CLIENT_SECRET: '' Insert text here ''
ASTRACS_TOKEN: '' Insert text here ''
Show code
Database: Semantic_Search_Using_Doc2'
Download Secure B...
Downloaded secure bundle for Semantic_Search_Using_Doc2Vec at /content/Semantic_Search_Using_Doc2Vec_secure_bundle.zip
```

Connect to our AstraDB VectorDB

```
#@title Connect to our AstraDB VectorDB
SECURE_CONNECT_BUNDLE_PATH = scb_path

cloud_config = {
   'secure_connect_bundle': SECURE_CONNECT_BUNDLE_PATH
}
auth_provider = PlainTextAuthProvider(ASTRA_CLIENT_ID, ASTRA_CLIENT_SECRET)
cluster = Cluster(cloud=cloud_config, auth_provider=auth_provider, protocol_version=4)
session = cluster.connect()
```

Mounting the Google Drive and defining the function for preprocessing the documents

```
drive.mount('/content/drive')

data_folder = '/content/drive/My Drive/Uni_Modules_Documents'


def read_and_tokenize(data_folder):
    documents = []
    for filename in os.listdir(data_folder):
        file_path = os.path.join(data_folder, filename)
        with open(file_path, 'rb') as file:
            raw_data = file.read()
            result = chardet.detect(raw_data)
            encoding = result['encoding']
        with open(file_path, 'r', encoding=encoding) as file:
            try:
                text = file.read()
                tokens = text.lower().split()
                documents.append(TaggedDocument(words=tokens, tags=[filename]))
            except UnicodeDecodeError as e:
                print(f"UnicodeDecodeError in file {file_path}: {e}")

    return documents
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=Tr
```

Defining the functions for the semantic search

```
KEYSPACE_NAME = 'tables_home'

TABLE_NAME = 'Modules'
NORMAL_TABLE_NAME = 'ModulesNonVectorized'
TABLE_NAME_SUMMARISED = 'SummarisedDocuments'

# Function to get the summarised versions of the modules from Astra DB
def get_document_text_summarised(session, filename):
    query = f"SELECT summary FROM {KEYSPACE_NAME}.{TABLE_NAME_SUMMARISED} WHERE filename = '{filename}'"
    result = session.execute(query)

    if result:
        return result.one().summary
    else:
        return None


# Function to get document text from the non-vectorised table
def get_document_text_non_vectorized(session, filename):
    query = f"SELECT text FROM {KEYSPACE_NAME}.{NORMAL_TABLE_NAME} WHERE filename = '{filename}'"
    result = session.execute(query)

    if result:
        return result.one().text
    else:
        return None

# Function to use the Doc2Vec model to vectorise the user's query
def vectorize_user_query(model, query_text):

    random.seed(42)  # Re-seed here to ensure determinism right before inference
    np.random.seed(42)
    # Tokenise the query text
    query_tokens = query_text.lower().split()

    # Infer the vector for the query using the Doc2Vec model
    query_vector = model.infer_vector(query_tokens)

    return query_vector


# Function to perform semantic search and display results
def semantic_search_and_display(session, query_vector, threshold, top_results):
    search_query = f"SELECT filename, doc_vector FROM {KEYSPACE_NAME}.{TABLE_NAME}"
    result = session.execute(search_query)

    # Normalise query vector
    query_vector_normalized = normalize([query_vector], norm='l2')[0]

    # Collect results and distances using cosine similarity
    results = []
    top_filenames = []

    for i, row in enumerate(result):
        filename = row.filename
        doc_vector = row.doc_vector
        doc_vector_normalized = normalize([doc_vector], norm='l2')[0]

        sim = cosine_similarity([query_vector_normalized], [doc_vector_normalized])[0][0]

        # Only consider results with similarity above the threshold
        if sim >= threshold:
            results.append({'filename': filename, 'similarity': sim})

    # Sort results by similarity
    sorted_results = sorted(results, key=lambda x: x['similarity'], reverse=True)

    # Display top results
    count = 0
    for res in sorted_results[:top_results]:
        filename = res['filename']
        similarity = res['similarity']
        document_text = get_document_text_non_vectorized(session, filename)
        summarised_text = get_document_text_summarised(session, filename)

        if document_text is not None and len(document_text) > 0:
            # Display the original document text
            print(f"Filename: {filename}, Cosine Similarity: {similarity}")
            print(f"Original Document Text:\n{document_text}")
            print(" ")
            print(f"Summarised Document Text:\n{summarised_text}")
            print(" ")
```

```
            print('='*50)

            top_filenames.append(filename)
            count += 1

    if count == 0:
        print("No matching documents found.")

    return top_filenames
```

Summarisation Step (don't need to run now)

Splitting up the document files into segments to process the summarise step (no need to run again since split will stay the same)

```
# The path to save the segments in Google Drive
drive_segments_path = '/content/drive/My Drive/document_segments.pkl'

# Function to save the segments into Google Drive
def save_segments_to_drive(segments, drive_path):
    with open(drive_path, 'wb') as f:
        pickle.dump(segments, f)

# Function to split the documents into segments
def split_documents(documents):
    total_documents = len(documents)
    num_segments = 4  # Divide documents into 4 segments
    segment_size = total_documents // num_segments
    remaining_documents = total_documents % num_segments

    print("Total Documents:", total_documents)
    print("Segment Size:", segment_size)
    print("Remaining Documents:", remaining_documents)

    segments = [[] for _ in range(num_segments)]
    current_segment = 0
    documents_remaining = total_documents

    for doc in documents:
        if current_segment < num_segments:
            segments[current_segment].append(doc)
            documents_remaining -= 1

            print("Segment:", current_segment, "Documents Remaining:", documents_remaining)

            if documents_remaining == 0:
                break

            # If remaining documents, increase segment size
            if remaining_documents > 0:
                if len(segments[current_segment]) >= segment_size + 1:
                    remaining_documents -= 1
                    current_segment += 1
            else:
                if len(segments[current_segment]) >= segment_size:
                    current_segment += 1
        else:
            break

    return segments

# Read and tokenise documents
documents = read_and_tokenize(data_folder)

# Split the documents into segments
document_segments = split_documents(documents)

# Save the segments to Google Drive
save_segments_to_drive(document_segments, drive_segments_path)

# Print segments to check if they have been split correctly
for i, segment in enumerate(document_segments):
    print(f"Segment {i+1} - Number of documents: {len(segment)}")
```

Loading the split of documents back in from google drive

```
# Define the path to the saved segments file in Google Drive
drive_segments_path = '/content/drive/My Drive/document_segments.pkl'

# Function to load the segments
def load_segments_from_drive(drive_path):
    with open(drive_path, 'rb') as f:
        segments = pickle.load(f)
    return segments

# Load the segments
document_segments = load_segments_from_drive(drive_segments_path)

# Print segments to check if they have been loaded correctly
for i, segment in enumerate(document_segments):
    print(f"Segment {i+1}:")
    for doc in segment:
        print(doc.tags[0])
    print(f"Number of documents: {len(segment)}")
```

Summarising a chosen segment

```
# Define the path to the saved segments and summaries files in Google Drive
drive_segments_path = '/content/drive/My Drive/document_segments.pkl'
drive_summaries_path = '/content/drive/My Drive/document_summaries.pkl'

# Load the pre-trained BART model and tokeniser
model_name = "facebook/bart-large-cnn"
tokenizer = BartTokenizer.from_pretrained(model_name)
model = BartForConditionalGeneration.from_pretrained(model_name)

# Function to capitalise the first letter of each sentence
def capitalize_sentences(text):
    sentences = re.split(r'(?<!\w\.\w.)(?<![A-Z][a-z]\.)(?<=\.|\?)\s', text)
    capitalized_sentences = [sentence.capitalize() for sentence in sentences]
    return ' '.join(capitalized_sentences)

# Function to summarise text using BART
def summarize_text(document_text):
    inputs = tokenizer([document_text], max_length=1024, return_tensors='pt', truncation=True)
    summary_ids = model.generate(inputs['input_ids'], num_beams=4, min_length=100, max_length=1000, early_stopping=True)
    summary = tokenizer.decode(summary_ids[0], skip_special_tokens=True, clean_up_tokenization_spaces=False)
    return capitalize_sentences(summary)

# Function to load the segments
def load_segments_from_drive(drive_path):
    with open(drive_path, 'rb') as f:
        segments = pickle.load(f)
    return segments

# Function to save the segments
def save_segments_to_drive(segments, drive_path):
    with open(drive_path, 'wb') as f:
        pickle.dump(segments, f)

# Function to load the summaries
def load_summaries_from_drive(drive_path):
    try:
        with open(drive_path, 'rb') as f:
            summaries = pickle.load(f)
    except FileNotFoundError:
        summaries = []
    return summaries

# Function to save the summaries
def save_summaries_to_drive(summaries, drive_path):
    with open(drive_path, 'wb') as f:
        pickle.dump(summaries, f)

# Load the segments
document_segments = load_segments_from_drive(drive_segments_path)

# Function to summarise a segment
def summarize_segment(segment_number):

    # Load the documents from the selected segment
```

```
    segment_documents = document_segments[segment_number]

    # Load the previously summarised documents
    previous_summaries = load_summaries_from_drive(drive_summaries_path)

    # Keep track of the number of files processed
    num_file = 0

    # Load the filenames associated with each document
    segment_filenames = [doc.tags[0] for doc in segment_documents]

    # Summarise each document in the segment
    segment_summaries = []
    for doc in segment_documents:
        document_text = ' '.join(doc.words)
        summary = summarize_text(document_text)
        segment_summaries.append((doc.tags[0], summary))  # Store the filename along with the summary
        print("Appending file", num_file)
        num_file += 1

    # Append the new summaries to the existing summaries
    combined_summaries = previous_summaries + segment_summaries

    # Save the combined summaries back to storage
    save_summaries_to_drive(combined_summaries, drive_summaries_path)

# To summarise the first segment:
summarize_segment(3)
```

Loading summaries back in to see how many files have been processed

```
# Define the path to the saved summaries file in Google Drive
drive_summaries_path = '/content/drive/My Drive/document_summaries.pkl'

# Function to load the summaries from the pickle file
def load_summaries_from_drive(drive_path):
    try:
        with open(drive_path, 'rb') as f:
            summaries = pickle.load(f)
    except FileNotFoundError:
        summaries = []
    return summaries

# Load the summaries from the pickle file
summarised_documents = load_summaries_from_drive(drive_summaries_path)
print ("Length of summarised files so far:", len(summarised_documents))

# Print the summarised documents
for i, summary in enumerate(summarised_documents):
    print(f"Document {i+1} summary:")
    print(summary)
    print()
```

Create the table for the summarised documents

```
KEYSPACE_NAME = 'tables_home'
TABLE_NAME_SUMMARISED = 'SummarisedDocuments'

# Function for executing a query with retry attempts to avoid operation timeout error occurring stopping the code
def execute_query_with_retry(query, session, max_retries=10, retry_interval=5):
    for attempt in range(max_retries):
        try:
            session.execute(query)
            print(f"Successfully executed query: {query}")
            break  # Successful, exit loop
        except Exception as e:
            print(f"Error: {e}")
            if attempt < max_retries - 1:
                print(f"Retrying in {retry_interval} seconds...")
                time.sleep(retry_interval)

# Drop the 'SummarisedDocuments' table if it exists
drop_table_query = f"DROP TABLE IF EXISTS {KEYSPACE_NAME}.{TABLE_NAME_SUMMARISED}"
execute_query_with_retry(drop_table_query, session)
print("Dropped existing table if it exists.")

# Create the 'SummarisedDocuments' table with retry attempts
create_table_query = f"""
```

```
CREATE TABLE IF NOT EXISTS {KEYSPACE_NAME}.{TABLE_NAME_SUMMARISED} (
    filename TEXT PRIMARY KEY,
    summary TEXT
)
"""
execute_query_with_retry(create_table_query, session)
print("Created new 'SummarisedDocuments' table.")
```

Defining the function for inserting the summarised documents into Astra DB

```
# Function to insert summarised documents into the 'SummarisedDocuments' table using batch statements
def insert_summarised_docs(session, summarised_documents, batch_size):

    # Prepare the insert query
    insert_query = f"INSERT INTO {KEYSPACE_NAME}.{TABLE_NAME_SUMMARISED} (filename, summary) VALUES (?, ?)"
    prepared_insert = session.prepare(insert_query)

    # Create a batch statement
    batch = BatchStatement()

    # Iterate over summarised_documents and add batch statements
    for filename, summary in summarised_documents:
        # Bind parameters to the prepared statement
        bound_statement = prepared_insert.bind((filename, summary))

        # Add the bound statement to the batch
        batch.add(bound_statement)

        # Execute the batch when it reaches the specified batch size
        if len(batch) >= batch_size:
            session.execute(batch)
            batch = BatchStatement()

    # Execute any remaining statements in the batch
    if batch:
        session.execute(batch)

    # Message to indicate the insertion was successful
    print(f"Inserted summarised documents into the '{TABLE_NAME_SUMMARISED}' table.")
```

Calling the function to insert the summarised documents

```
insert_summarised_docs(session, summarised_documents, batch_size = 15)
```

```
Inserted summarised documents into the 'SummarisedDocuments' table.
```

The Data Model

Defining the function for training a model (don't need to run now)

```
def train_doc2vec_model(documents, vector_size, window, min_count, epochs):
    model = Doc2Vec(vector_size=vector_size, window=window, min_count=min_count, workers=4)

    train_docs, test_docs = train_test_split(documents, test_size=0.2, random_state=42)
    model.build_vocab(documents)
    model.train(train_docs, total_examples=model.corpus_count, epochs=epochs)
    return model
```

Load in a previous model with best parameters

```
data_folder = '/content/drive/My Drive/Uni_Modules_Documents'
documents = read_and_tokenize(data_folder)
final_model_path_segment2_test2 = '/content/drive/My Drive/module_info_doc2vec_model_segment2_test2'
loaded_model = Doc2Vec.load(final_model_path_segment2_test2)
```

Vectorising the data

Vectorising the documents and displaying them as file name and vector

```
# Vectorise existing documents in the specified folder
existing_data_folder = '/content/drive/My Drive/Uni_Modules_Documents'

def vectorize_existing_documents(model, folder_path):
    vectorized_documents = []

    for filename in os.listdir(folder_path):
        file_path = os.path.join(folder_path, filename)
        with open(file_path, 'r', encoding='utf-8', errors='ignore') as file:
            text = file.read()
            tokens = text.lower().split()
            inferred_vector = model.infer_vector(tokens)
            vectorized_documents.append({'filename': filename, 'vector': inferred_vector})

    return vectorized_documents

# Vectorise existing documents
vectorized_docs = vectorize_existing_documents(loaded_model, existing_data_folder)

# Display the vectors for each document
for doc in vectorized_docs:
    print(f"Filename: {doc['filename']}, Vector: {doc['vector']}")
```

```
 -2.10865997e-02 -1.33363530e-02  5.28852940e-02  2.55095139e-02
  8.80088750e-03 -1.52832111e-02 -4.76789773e-02 -2.14105677e-02
 -2.47239210e-02  1.89222712e-02 -7.83792231e-03 -1.01460598e-03
  3.80066559e-02  1.38961580e-02  3.54348458e-02  4.74451557e-02
 -2.38220450e-02  4.77641337e-02 -5.27275540e-02  5.68021648e-03]
Filename: DJ22016.txt, Vector: [ 0.03229819  0.05551609  0.01388632  0.02346907 -0.01464185 -0.07660795
  0.10077694  0.10315573  0.00588618  0.0242901  -0.00488756 -0.08895537
  0.01622519 -0.01390985 -0.03999426 -0.01454482  0.03842136 -0.04238027
 -0.0039442   0.00044223 -0.03722843 -0.03665702  0.06114624 -0.04993546
 -0.02016744 -0.06671459 -0.01036156 -0.0639685   0.04636813 -0.061484
 -0.02409177 -0.04529737  0.00459839 -0.03310296 -0.00468583 -0.00599546
 -0.03843213 -0.10012157 -0.05990045 -0.01120523 -0.06529704 -0.00294882
  0.09930375 -0.15940702  0.04085983  0.06229771  0.068062   -0.10010646
  0.01728276  0.08538623  0.05879722  0.01457037 -0.00802217  0.03879355
 -0.02883586 -0.01488332  0.07373788 -0.00561411 -0.03187     0.01896785
 -0.07131771 -0.00068272 -0.04719733 -0.02752096 -0.01503558  0.02191342
 -0.04485433 -0.00879825 -0.04376647 -0.02895143 -0.0519517   0.04115737
  0.07124419 -0.07001346  0.02027167  0.04976889 -0.08510328  0.04679753
  0.03065637  0.0552936  -0.03861371 -0.08707608 -0.04945257  0.14228529
 -0.12238751 -0.02581869 -0.06238278  0.04835483  0.06822317 -0.02409001
  0.04211699 -0.12018166  0.04298631  0.02720178  0.10088565  0.06420567
  0.06681708 -0.02559848 -0.09771743  0.02754017  0.03772573 -0.01642229
  0.08556653  0.08107675  0.08413216 -0.03545362  0.03831553  0.03300079
 -0.09427031 -0.00226532 -0.04076549 -0.06241128  0.04616522  0.07250863
  0.00474493  0.00489814 -0.03798876  0.04642612  0.04267562 -0.0796471
  0.00786544  0.02147936  0.06339144  0.0127092   0.02196686  0.04565349
 -0.00164085 -0.00769805  0.00590391  0.07443827  0.08687673  0.14042379
 -0.00670294 -0.03035867  0.03607009  0.04320849 -0.04447912 -0.06354392
 -0.0270732  -0.10532483  0.0023159  -0.02955859  0.02173279 -0.02370756
  0.05042806 -0.03229658 -0.09108291 -0.05320815  0.04677127 -0.0992081
  0.01129264 -0.1725746  -0.0610307  -0.12154226  0.04575629  0.00961709
 -0.07683381 -0.00851387  0.03542272  0.00402855  0.06791611 -0.01040899
  0.02148926  0.0717921   0.06275725  0.03222845  0.01151085 -0.00244748
 -0.04757484  0.11529601 -0.06103299 -0.06637907  0.0279579   0.00932029
 -0.02452809  0.00873311 -0.05240782  0.00347077  0.01195958  0.03231275
 -0.03603109  0.02161928 -0.06548066  0.01179012 -0.04044856 -0.06273178
  0.0515712   0.06399056 -0.00084573 -0.03664634  0.0139816   0.01243141
 -0.09881507 -0.03756318  0.07041488 -0.0752561   0.01106136 -0.01430494
  0.00745543  0.03694102 -0.05626602  0.00598596  0.09822121 -0.13195378
  0.0167415  -0.02012476  0.02060083  0.0849026  -0.02986077 -0.13513798
 -0.01897467 -0.02171068 -0.00149988 -0.04329507 -0.03517412 -0.02546331
 -0.00922254 -0.14205565 -0.04114148 -0.02996879  0.04006293  0.03210131
 -0.01530487 -0.09926429  0.00738111 -0.03796382 -0.03345878  0.01990163
 -0.04855678 -0.01961541  0.03946258  0.00064009  0.00757436  0.08745168
  0.07355636  0.09155549  0.00371954  0.06045258  0.0102683  -0.02404953
  0.04830659 -0.03110317 -0.06652015  0.03359571  0.0615833  -0.04295037
 -0.06766534  0.01093874  0.00912789  0.05959755 -0.00085802 -0.01244935
  0.09521981 -0.02851679 -0.10325965 -0.0589906   0.11440451 -0.01597785
 -0.09871921 -0.0660387   0.06836921 -0.00918825  0.01539375 -0.12556925
 -0.10568145 -0.00812255  0.11556338 -0.01700043 -0.0215163   0.01931568
 -0.07223965 -0.05489545 -0.03113347 -0.00733089  0.0467754   0.03594637
  0.05876877  0.00380846  0.02903819  0.02677953 -0.04671011 -0.02660345
  0.02075911  0.0795929  -0.03428128 -0.054781   -0.08469412 -0.03743371
  0.05664594  0.06410658  0.06643732  0.03903269  0.0886631   0.04835624
  0.04634263  0.05515927 -0.01771895  0.06082178  0.0480215  -0.0036196 ]
Filename: DJ31035.txt, Vector: [-0.01382222 -0.00455265  0.01095082  0.01390787 -0.00660568 -0.05832383
  0.06869937  0.05164632  0.0605716   0.02494079 -0.01470928 -0.01434308
  0.01620888 -0.00016232 -0.03549657 -0.00584762  0.00916557  0.00275012
```

Storing the Vectorised documents into Astra DB

Create the table for Vectorised documents

```
KEYSPACE_NAME = 'tables_home'
TABLE_NAME = 'Modules'

# Function for executing a query with retry attempts to avoid operation timeout error occuring stopping the code
def execute_query_with_retry_vec(query, max_retries=10, retry_interval=5):
    for attempt in range(max_retries):
        try:
            session.execute(query)
            print(f"Successfully executed query")
            break  # Successful, exit loop
        except Exception as e:
            print(f"Error: {e}")
            if attempt < max_retries - 1:
                print(f"Retrying in {retry_interval} seconds...")
                time.sleep(retry_interval)

# Drop the table if it exists with retry attempts
drop_table_query = f"DROP TABLE IF EXISTS {KEYSPACE_NAME}.{TABLE_NAME}"
execute_query_with_retry_vec(drop_table_query)

# Create the 'Modules' table with filename as the primary key and doc_vector column with retry attempts
create_table_query = f"""
CREATE TABLE IF NOT EXISTS {KEYSPACE_NAME}.{TABLE_NAME} (
    filename TEXT PRIMARY KEY,
    doc_vector LIST<FLOAT>
)
"""
execute_query_with_retry_vec(create_table_query)

# Create an index on the 'doc_vector' column for semantic search with retry attempts
index_creation_query = f"""
CREATE CUSTOM INDEX IF NOT EXISTS semantic_search_index ON {KEYSPACE_NAME}.{TABLE_NAME} (doc_vector) USING 'StorageAttached
"""
execute_query_with_retry_vec(index_creation_query)
```

```
Successfully executed query
Successfully executed query
Successfully executed query
```

Defining the function for inserting the vectorised documents into Astra DB

```
def insert_vectorized_docs(session, vectorized_docs, batch_size):
    KEYSPACE_NAME = 'tables_home'
    TABLE_NAME = 'Modules'

    # Prepare the insert query
    insert_query = f"INSERT INTO {KEYSPACE_NAME}.{TABLE_NAME} (filename, doc_vector) VALUES (?, ?)"
    prepared_insert = session.prepare(insert_query)

    # Create a batch statement
    batch = BatchStatement()

    # Iterate over vectorised_docs and add batch statements
    for doc in vectorized_docs:
        filename = doc['filename']
        vector = doc['vector']

        # Bind parameters to the prepared statement
        bound_statement = prepared_insert.bind((filename, vector))

        # Add the bound statement to the batch
        batch.add(bound_statement)

        # Execute the batch when it reaches the specified batch size
        if len(batch) >= batch_size:
            session.execute(batch)
            batch = BatchStatement()

    # Execute any remaining statements in the batch
    if batch:
        session.execute(batch)
```

```
    # Message to say the insertion was successful
    print(f"Inserted documents into the 'Modules' table.")
```

Calling the function to insert the vectorised documents

```
insert_vectorized_docs(session, vectorized_docs, batch_size=15)
```

```
WARNING:cassandra.protocol:Server warning: Detected collection doc_vector with 300 items, greater than the maximum recomme
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Detected collection doc_vector with 300 items, greater than the maximum recomme
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Detected collection doc_vector with 300 items, greater than the maximum recomme
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
WARNING:cassandra.protocol:Server warning: Batch for ["36646662386635302d303630662d343635622d396137662d3365316562363836663
```

Storing the Non Vectorised documents into Astra DB (don't need to run again unless more modules are added or existing modules need to be changed)

Create the table for non-vectorised documents

```
KEYSPACE_NAME = 'tables_home'
TABLE_NAME_NON_VECTORIZED = 'ModulesNonVectorized'

# Function for executing a query with retry attempts to avoid operation timeout error occuring stopping the code
def execute_query_with_retry_non_vec(query, max_retries=10, retry_interval=5):
    for attempt in range(max_retries):
        try:
            session.execute(query)
            print(f"Successfully executed query")
```

```
            break  # Successful, exit loop
        except Exception as e:
            print(f"Error: {e}")
            if attempt < max_retries - 1:
                print(f"Retrying in {retry_interval} seconds...")
                time.sleep(retry_interval)

# Drop the table if it exists with retry attempts
drop_table_query = f"DROP TABLE IF EXISTS {KEYSPACE_NAME}.{TABLE_NAME_NON_VECTORIZED}"
execute_query_with_retry_non_vec(drop_table_query)

# Create the 'ModulesNonVectorized' table with filename as the primary key and text column with retry attempts
create_table_query = f"""
CREATE TABLE IF NOT EXISTS {KEYSPACE_NAME}.{TABLE_NAME_NON_VECTORIZED} (
    filename TEXT PRIMARY KEY,
    text TEXT
)
"""
execute_query_with_retry_non_vec(create_table_query)
```

```
Successfully executed query
Successfully executed query
```

Defining the function for inserting the non-vectorised documents into Astra DB

```
def insert_non_vectorized_docs(session, documents):

    # Insert non-vectorised documents into the 'ModulesNonVectorized' table
    for doc in documents:
        filename = doc.tags[0]  # Assuming tags contain the filename
        text = ' '.join(doc.words)

        # Use query for insertion
        insert_query = f"INSERT INTO {KEYSPACE_NAME}.{TABLE_NAME_NON_VECTORIZED} (filename, text) VALUES (?, ?)"
        prepared_insert = session.prepare(insert_query)
        bound_statement = BoundStatement(prepared_insert)

        # Explicitly set the primary key column and text
        bound_statement.bind((filename, text))

        session.execute(bound_statement)

        print(f"Inserted document {filename} into the 'ModulesNonVectorized' table.")
```

Calling the function to insert the non-vectorised documents

```
insert_non_vectorized_docs(session, documents)
```

Finding the best hyperparameters (don't need to run now)

Splitting up the combinations of hyperparameters into 6 segments to use them individually to prevent memory issues

```
# The path to save the segments in Google Drive
drive_segments_path = '/content/drive/My Drive/segments.pkl'

# Function to save the segments into google drive
def save_segments_to_drive(segments, drive_path):
    with open(drive_path, 'wb') as f:
        pickle.dump(segments, f)

# Function to split the permutations of hyperparameters into 6 segments
def split_hyperparams_grid(hyperparams_grid):

    # Calculate the total number of combinations
    total_combinations = 1
    for param_list in hyperparams_grid.values():
        total_combinations *= len(param_list)

    # Split the combinations into 6 segments
    segment_size = total_combinations // 6
    segments = [dict() for _ in range(6)]
    current_segment = 0
```

```
    current_combinations = 0

    for params in islice(ParameterGrid(hyperparams_grid), total_combinations):
        if current_combinations >= segment_size:
            current_segment += 1
            current_combinations = 0

        if current_segment >= 6:
            break

        segments[current_segment][current_combinations] = params
        current_combinations += 1

    return segments

# Define the hyperparameters grid
hyperparams_grid = {
    'vector_size': [50, 100, 125, 150, 175, 200, 250, 275, 300],
    'window': [1, 2, 3, 4, 5, 6],
    'min_count': [1, 2, 3, 4, 5],
    'epochs': [3, 5, 8, 10, 15, 20]
}

# Get the segments
segments = split_hyperparams_grid(hyperparams_grid)

# Saves the segments to google drive
save_segments_to_drive(segments, drive_segments_path)

# Prints each segment with the permutations of hyperparameters
for i, segment in enumerate(segments):
    print(f"Segment {i+1} - Combinations: {len(segment)}")
    print(segment)
```

Loads the segments split into colabs from google drive

```
# Define the path to the saved segments file in Google Drive
drive_segments_path = '/content/drive/My Drive/segments.pkl'

# Function to load the segments
def load_segments_from_drive(drive_path):
    with open(drive_path, 'rb') as f:
        segments = pickle.load(f)
    return segments

# Load the segments
segments = load_segments_from_drive(drive_segments_path)


# Print segments to check if it has loaded back in correctly
for i, segment in enumerate(segments):
    print(f"Segment {i+1}")
    print(segment)
```

Test 1 for hyperparameters (only expected filename)

```
# Not all the urls will be used for the final model (this is just to keep models that have been trained before stored so th

###################
#TEST 1 RUNTHROUGHS
###################
# Creating the url's for each model segment for runthrough 1 (65% threshold) for test 1
final_model_path_segment1 = '/content/drive/My Drive/module_info_doc2vec_model_segment1' ## Best of segment: 33% success, p
final_model_path_segment2 = '/content/drive/My Drive/module_info_doc2vec_model_segment2' ## Best of segment: 53% success, p
final_model_path_segment3 = '/content/drive/My Drive/module_info_doc2vec_model_segment3' ## Best of segment: 53% success, p
final_model_path_segment4 = '/content/drive/My Drive/module_info_doc2vec_model_segment4' ## Best of segment: 53% success, p
final_model_path_segment5 = '/content/drive/My Drive/module_info_doc2vec_model_segment5' ## Best of segment: 40% success, p
final_model_path_segment6 = '/content/drive/My Drive/module_info_doc2vec_model_segment6' ## Best of segment: 40% success, p

# Creating the url's for each model segment for runthrough 2 (75% threshold) for test 1
final_model_path_segment1_run2 = '/content/drive/My Drive/module_info_doc2vec_model_segment1_run2' ## Best of segment: 40%
final_model_path_segment2_run2 = '/content/drive/My Drive/module_info_doc2vec_model_segment2_run2' ## Best of segment: 47%
final_model_path_segment3_run2 = '/content/drive/My Drive/module_info_doc2vec_model_segment3_run2' ## Best of segment: 47%
final_model_path_segment4_run2 = '/content/drive/My Drive/module_info_doc2vec_model_segment4_run2' ## Best of segment: 47%
final_model_path_segment5_run2 = '/content/drive/My Drive/module_info_doc2vec_model_segment5_run2' ## Best of segment: 40%
final_model_path_segment6_run2 = '/content/drive/My Drive/module_info_doc2vec_model_segment6_run2' ## Best of segment: 40%
```

```
# Creating the url's for each model segment for runthrough 3 (85% threshold) for test 1
final_model_path_segment1_run3 = '/content/drive/My Drive/module_info_doc2vec_model_segment1_run3' ## Best of segment: 27%
final_model_path_segment2_run3 = '/content/drive/My Drive/module_info_doc2vec_model_segment2_run3' ## Best of segment: 47%
final_model_path_segment3_run3 = '/content/drive/My Drive/module_info_doc2vec_model_segment3_run3' ## Best of segment: 47%
final_model_path_segment4_run3 = '/content/drive/My Drive/module_info_doc2vec_model_segment4_run3' ## Best of segment: 47%
final_model_path_segment5_run3 = '/content/drive/My Drive/module_info_doc2vec_model_segment5_run3' ## Best of segment: 40%
final_model_path_segment6_run3 = '/content/drive/My Drive/module_info_doc2vec_model_segment6_run3' ## Best of segment: 33%
```

```
def run_segment(segment):
    # Define user queries and expected filename letters
    user_queries = ["Mathematics", "Physics", "Show me law modules", "Spanish modules", "Art and Design", "Geography", "Cor
    expected_filename_letters = [["MA"], ["PH"], ["LW"], ["PS"], ["DJ"], ["GE"], ["CS"], ["ME"], ["PO"], ["EN"], ["CA"], ['

    # Initialise MultiLabelBinarizer
    mlb = MultiLabelBinarizer()

    best_accuracy = 0
    best_params = None
    number_of_iterations = 0

    for params in segment.values():

        number_of_iterations += 1
        # Clears the output every iteration to avoid memory issues
        display.clear_output(wait=True)

        print("=" * 50)
        print("Iteration:", number_of_iterations)
        print("Parameters:", params )
        print("")

        # Extract the current hyperparameters
        vector_size = params['vector_size']
        window = params['window']
        min_count = params['min_count']
        epochs = params['epochs']

        # Initialise and train Doc2Vec model with the current hyperparameters
        model = train_doc2vec_model(documents, vector_size=vector_size, window=window, min_count=min_count, epochs=epochs)

        # Call all functions to reinsert vectorised documents into tables.
        KEYSPACE_NAME = 'tables_home'
        TABLE_NAME = 'Modules'

        vectorized_docs = vectorize_existing_documents(model, existing_data_folder)

        # Drops previous table to avoid duplication issues
        drop_table_query = f"DROP TABLE IF EXISTS {KEYSPACE_NAME}.{TABLE_NAME}"
        execute_query_with_retry_vec(drop_table_query)

        create_table_query = f"""
        CREATE TABLE IF NOT EXISTS {KEYSPACE_NAME}.{TABLE_NAME} (
            filename TEXT PRIMARY KEY,
            doc_vector LIST<FLOAT>
        )
        """
        execute_query_with_retry_vec(create_table_query)

        index_creation_query = f"""
        CREATE CUSTOM INDEX IF NOT EXISTS semantic_search_index ON {KEYSPACE_NAME}.{TABLE_NAME} (doc_vector) USING 'Storage
        """
        execute_query_with_retry_vec(index_creation_query)

        insert_vectorized_docs(session, vectorized_docs, batch_size=15)

        # Retrieve relevant documents using semantic search for each user query
        all_predictions = []
        for query in user_queries:
            query_vector = vectorize_user_query(model, query)
            relevant_filenames = semantic_search_and_display(session, query_vector, threshold=0.85, top_results=5)
            filename_letters = [filename[:2] for filename in relevant_filenames]
            all_predictions.append(filename_letters)

        # Encode expected filename letters and predictions using MultiLabelBinarizer
        expected_labels = mlb.fit_transform(expected_filename_letters)
        predicted_labels = mlb.transform(all_predictions)

        # Evaluate model based on accuracy score for all user queries
        avg_accuracy = accuracy_score(expected_labels, predicted_labels)
```

```
        print("average accuracy", avg_accuracy)
        print("")

        # Update best parameters if necessary and save the current best model for that segment
        if avg_accuracy > best_accuracy:
            best_accuracy = avg_accuracy
            best_params = params
            model.save(final_model_path_segment6_run3)

        del model

    return best_params, best_accuracy


# Run the chosen segment
best_params_segment, best_accuracy_segment = run_segment(segments[5])

print("Best hyperparameters found in the segment:")
print(best_params_segment)
print("Accuracy:", best_accuracy_segment)
```

Test 2 for hyperparameters (with key words included along with expected filenames)

```
# Not all the urls will be used for the final model (this is just to keep models that have been trained before stored so th

###################
#TEST 2 RUNTHROUGHS
###################
# Creating the url's for each model segment for runthrough 1 (65% threshold) for test 2
final_model_path_segment1_test2 = '/content/drive/My Drive/module_info_doc2vec_model_segment1_test2' ## Best of segment: 67
final_model_path_segment2_test2 = '/content/drive/My Drive/module_info_doc2vec_model_segment2_test2' ## Best of segment: 83
final_model_path_segment3_test2 = '/content/drive/My Drive/module_info_doc2vec_model_segment3_test2' ## Best of segment: 81
final_model_path_segment4_test2 = '/content/drive/My Drive/module_info_doc2vec_model_segment4_test2' ## Best of segment: 77
final_model_path_segment5_test2 = '/content/drive/My Drive/module_info_doc2vec_model_segment5_test2' ## Best of segment: 65
final_model_path_segment6_test2 = '/content/drive/My Drive/module_info_doc2vec_model_segment6_test2' ## Best of segment: 59

# Creating the url's for each model segment for runthrough 2 (75% threshold) for test 2
final_model_path_segment1_test2_run2 = '/content/drive/My Drive/module_info_doc2vec_model_segment1_test2_run2' ## Best of s
final_model_path_segment2_test2_run2 = '/content/drive/My Drive/module_info_doc2vec_model_segment2_test2_run2' ## Best of s
final_model_path_segment3_test2_run2 = '/content/drive/My Drive/module_info_doc2vec_model_segment3_test2_run2' ## Best of s
final_model_path_segment4_test2_run2 = '/content/drive/My Drive/module_info_doc2vec_model_segment4_test2_run2' ## Best of s
final_model_path_segment5_test2_run2 = '/content/drive/My Drive/module_info_doc2vec_model_segment5_test2_run2' ## Best of s
final_model_path_segment6_test2_run2 = '/content/drive/My Drive/module_info_doc2vec_model_segment6_test2_run2' ## Best of s

# Creating the url's for each model segment for runthrough 3 (85% threshold) for test 2
final_model_path_segment1_test2_run3 = '/content/drive/My Drive/module_info_doc2vec_model_segment1_test2_run3' ## Best of s
final_model_path_segment2_test2_run3 = '/content/drive/My Drive/module_info_doc2vec_model_segment2_test2_run3' ## Best of s
final_model_path_segment3_test2_run3 = '/content/drive/My Drive/module_info_doc2vec_model_segment3_test2_run3' ## Best of s
final_model_path_segment4_test2_run3 = '/content/drive/My Drive/module_info_doc2vec_model_segment4_test2_run3' ## Best of s
final_model_path_segment5_test2_run3 = '/content/drive/My Drive/module_info_doc2vec_model_segment5_test2_run3' ## Best of s
final_model_path_segment6_test2_run3 = '/content/drive/My Drive/module_info_doc2vec_model_segment6_test2_run3' ## Best of s


def run_segment(segment):

    # Define user queries and expected filename letters
    user_queries = ["Mathematics", "Physics", "Show me law modules", "Spanish modules", "Art and Design", "Geography", "Cor
    expected_filename_letters = ["MA"] * 5 + ["PH"] * 5 + ["LW"] * 5 + ["PS"] * 5 + ["DJ"] * 5 + ["GE"] * 5 + ["CS"] * 5 +
    expected_filename_letters_positions = ["MA", "PH", "LW", "PS", "DJ", "GE", "CS", "ME", "PO", "EN", "CA", "NU", "AR", "E

    # Initialize MultiLabelBinarizer
    mlb = MultiLabelBinarizer()

    best_accuracy = 0
    best_params = None
    number_of_iterations = 0

    for params in segment.values():

        number_of_iterations += 1
        # Clears the output every iteration to avoid memory issues
        display.clear_output(wait=True)

        print("=" * 50)
        print("Iteration:", number_of_iterations)
        print("Parameters:", params )
        print("Best Accuracy so far", best_accuracy)
```

```
print("")

# Extract the current hyperparameters
vector_size = params['vector_size']
window = params['window']
min_count = params['min_count']
epochs = params['epochs']

# Initialise and train Doc2Vec model with the current hyperparameters
model = train_doc2vec_model(documents, vector_size=vector_size, window=window, min_count=min_count, epochs=epochs)

# Call all functions to reinsert vectorised documents into tables.
KEYSPACE_NAME = 'tables_home'
TABLE_NAME = 'Modules'

vectorized_docs = vectorize_existing_documents(model, existing_data_folder)

# Drops previous table to avoid duplication issues
drop_table_query = f"DROP TABLE IF EXISTS {KEYSPACE_NAME}.{TABLE_NAME}"
execute_query_with_retry_vec(drop_table_query)

create_table_query = f"""
CREATE TABLE IF NOT EXISTS {KEYSPACE_NAME}.{TABLE_NAME} (
    filename TEXT PRIMARY KEY,
    doc_vector LIST<FLOAT>
)
"""
execute_query_with_retry_vec(create_table_query)

index_creation_query = f"""
CREATE CUSTOM INDEX IF NOT EXISTS semantic_search_index ON {KEYSPACE_NAME}.{TABLE_NAME} (doc_vector) USING 'Storage
"""
execute_query_with_retry_vec(index_creation_query)

insert_vectorized_docs(session, vectorized_docs, batch_size=15)

# Retrieve relevant documents using semantic search for each user query

excluded_words = ["module", "show", "me", "and", "i", "want", "to", "see"]
all_predictions = []  # Initialise with an empty list


for query in user_queries:
    query_vector = vectorize_user_query(model, query)
    relevant_filenames = semantic_search_and_display(session, query_vector, threshold=0.85, top_results=5)
    none_count = 0
    file_count = 0

    if relevant_filenames:

        # If less than 5 files are returned
        if len(relevant_filenames) < 5:
           file_count = len(relevant_filenames)
           while file_count < 5:
                all_predictions.append("None")
                file_count += 1

           for filename in relevant_filenames:
                    document = next((doc for doc in documents if doc.tags[0] == filename), None)
                    if document:
                        document_text = ' '.join(document.words)
                        query_words = [word.lower() for word in query.split() if word.lower() not in excluded_words

                        if any(word.lower() in document_text.lower() for word in query_words):
                            query_position = user_queries.index(query)
                            all_predictions.append(expected_filename_letters_positions[query_position])
                        else:
                            all_predictions.append(filename[:2])


        # If all 5 results are returned
        else:
            for filename in relevant_filenames:
                    document = next((doc for doc in documents if doc.tags[0] == filename), None)
                    if document:
                        document_text = ' '.join(document.words)
                        query_words = [word.lower() for word in query.split() if word.lower() not in excluded_words

                        if any(word.lower() in document_text.lower() for word in query_words):
                            query_position = user_queries.index(query)
                            all_predictions.append(expected_filename_letters_positions[query_position])
                        else:
```

```
                                    all_predictions.append(filename[:2])

            # If no files are returned
            else:
                if none_count < 5:
                    while none_count < 5:
                        all_predictions.append("None")
                        none_count += 1


        print(all_predictions)
        print(expected_filename_letters)

        # Evaluate model based on accuracy score for all user queries
        avg_accuracy = accuracy_score(all_predictions, expected_filename_letters)

        print("average accuracy", avg_accuracy)
        print("")

        # Update best parameters if necessary and save the current best model for that segment
        if avg_accuracy > best_accuracy:
            best_accuracy = avg_accuracy
            best_params = params
            model.save(final_model_path_segment6_test2_run3)

        del model

    return best_params, best_accuracy

# Run the chosen segment
best_params_segment, best_accuracy_segment = run_segment(segments[5])

print("Best hyperparameters found in the segment:")
print(best_params_segment)
print("Accuracy:", best_accuracy_segment)
```

Semantic Search function called (to show model works as intended within Colabs without Anvil interference)

```
# Load the trained model
final_model_path_segment2_test2 = '/content/drive/My Drive/module_info_doc2vec_model_segment2_test2'
loaded_model = Doc2Vec.load(final_model_path_segment2_test2)

KEYSPACE_NAME = 'tables_home'
TABLE_NAME = 'Modules'

NORMAL_TABLE_NAME = 'ModulesNonVectorized'

# Example usage:
query_text = "physics"
query_vector = vectorize_user_query(loaded_model, query_text)

# For the retry method
max_retries=10
retry_interval=5

def execute_query_with_retry():
    for attempt in range(max_retries):
        try:
            semantic_search_and_display(session, query_vector, threshold=0.65, top_results=5)
            print(f"Successfully executed query")
            break  # Successful, exit loop
        except Exception as e:
            print(f"Retrying...")
            if attempt < max_retries - 1:
                print(f"Retrying in {retry_interval} seconds...")
                time.sleep(retry_interval)

execute_query_with_retry()
```

```
ifferential equations via separation of variables and direct integration * determinants and matrices. eigenvalues and eigenv
```

```
g the liquid drop model and the development of the semi-empirical mass formula (semf) * exploitation of the semf for determi
```

```
to condensed matter physics, including energy bands, free-electron model of metals, and semiconductors. * introduction to nu
```

```
mass, conservation of momentum, euler’s equation, energy equation, equation of state. * common approximations incompressible
```

Defining the semantic search function for the *webpage*

```
def semantic_search_and_display_web(session, query_vector, threshold, top_results):
    search_query = f"SELECT filename, doc_vector FROM {KEYSPACE_NAME}.{TABLE_NAME}"
    result = session.execute(search_query)

    # Normalize query vector
    query_vector_normalized = normalize([query_vector], norm='l2')[0]

    # Collect results and distances using cosine similarity
    search_results = []

    for i, row in enumerate(result):
        filename = row.filename
        doc_vector = row.doc_vector
        doc_vector_normalized = normalize([doc_vector], norm='l2')[0]
        sim = cosine_similarity([query_vector_normalized], [doc_vector_normalized])[0][0]

        # Only consider results with similarity above the threshold
        if sim >= threshold:
            search_results.append({'filename': filename, 'similarity': sim})

    # Sort results by similarity
    sorted_results = sorted(search_results, key=lambda x: x['similarity'], reverse=True)

    # Compile top results
    top_results_list = []
    for res in sorted_results[:top_results]:
        filename = res['filename']
        similarity = res['similarity']
        document_text = get_document_text_non_vectorized(session, filename)
        summarised_text = get_document_text_summarised(session, filename)

        if document_text and summarised_text:
            # Append result details to the list
            result_details = {
                'filename': filename,
                'similarity': similarity,
                'original_text': document_text,
                'summarised_text': summarised_text
            }
            top_results_list.append(result_details)

    return top_results_list
```

Cell to interact with Anvil Webpage

```
# Connect to Anvil with the Uplink key
anvil.server.connect("server_WKP7MDP6G4T7TYNVGP4URKGM-4CHFUI6ZBU3AINRJ")

@anvil.server.callable
def semantic_search(query):
    print(f"Received query: '{query}'")

    query_vector = vectorize_user_query(loaded_model, query)

    # Execute the search and retrieve results
    search_results = semantic_search_and_display_web(session, query_vector, threshold=0.65, top_results=5)
    print(f"Search results: {search_results}")

    formatted_results = []
    for result in search_results:
        formatted_result = f"{result['filename']} - Similarity: {result['similarity']}\nOriginal Text: {result['original_te
        formatted_results.append(formatted_result)

    return formatted_results

# Keep the notebook connected to Anvil indefinitely
anvil.server.wait_forever()
```

```
Connecting to wss://anvil.works/uplink
Anvil websocket open
Connected to "Default Environment" as SERVER
Received query: 'mathematics'
Search results: [{'filename': 'PH12004.txt', 'similarity': 0.9097860079487334, 'original_text': 'light and matter module (ph
Received query: 'mathematics'
Search results: [{'filename': 'PY22002.txt', 'similarity': 0.8999065058105454, 'original_text': 'individual development modu
Received query: 'mathematics'
Search results: [{'filename': 'PY32002.txt', 'similarity': 0.9536505065234939, 'original_text': 'psychology of language modu
Received query: 'physics'
Search results: [{'filename': 'PH52012.txt', 'similarity': 0.9380176916410692, 'original_text': 'condensed matter physics ii
Received query: 'law'
Search results: [{'filename': 'LW32028.txt', 'similarity': 0.9475740941096799, 'original_text': "english law of property mod
Received query: 'physics'
Search results: [{'filename': 'PH32009.txt', 'similarity': 0.9425365040638625, 'original_text': 'thermal physics i module (p
Received query: 'law'
Search results: [{'filename': 'LW12010.txt', 'similarity': 0.8994466458306448, 'original_text': 'scots criminal law & evider
Received query: 'english'
Search results: [{'filename': 'PS40002.txt', 'similarity': 0.9181587731109389, 'original_text': 'applied spanish 4: language
Received query: 'english and film'
Search results: [{'filename': 'LV32001.txt', 'similarity': 0.9337207232920306, 'original_text': 'digital evidence module (lv
```

Evaluation step (not used anymore as this version of testing the hyperparameters was scrapped)

```
# Load your pre-trained model
model_path = '/content/drive/My Drive/module_info_doc2vec_model_v1'
loaded_model = Doc2Vec.load(model_path)

train_docs, test_docs = train_test_split(documents, test_size=0.2, random_state=42)

def top_k_accuracy(true_indices, predicted_indices):
    correct_predictions = 0
    for true_index, top_predictions in zip(true_indices, predicted_indices):
        if true_index in top_predictions:
            correct_predictions += 1
    return correct_predictions / len(true_indices)

def mean_reciprocal_rank(true_indices, predicted_indices):
    reciprocal_ranks = []
    for true_index, top_predictions in zip(true_indices, predicted_indices):
        if true_index in top_predictions:
            reciprocal_ranks.append(1 / (top_predictions.index(true_index) + 1))
        else:
            reciprocal_ranks.append(0)
    if not reciprocal_ranks:
        return 0.0
    return sum(reciprocal_ranks) / len(reciprocal_ranks)

def precision_at_k(true_indices, predicted_indices):
    correct_predictions = 0
    total_predictions = 0
    for true_index, top_predictions in zip(true_indices, predicted_indices):
```

```
            if true_index in top_predictions:
                correct_predictions += 1
            if top_predictions:
                total_predictions += 1
        if total_predictions == 0:
            return 0.0
        return correct_predictions / total_predictions


    def get_ground_truth_indices(test_docs):
        return [pair[1][0] for pair in test_docs]

    def get_predicted_indices(model, test_docs):
        predicted_indices = []

        for query, relevant_document_index in test_docs:
            # Always treat the query as a list of tokens
            query_vector = vectorize_user_query(loaded_model, ' '.join(query))

            # Perform semantic search to get similar documents
            similar_documents = semantic_search_and_display(session, query_vector, threshold=0.8, top_results=5)

            print("Query:", query)
            print("Filenames in similar_documents:", similar_documents)

            # Append all filenames obtained from the semantic search
            predicted_indices.append(similar_documents)

        print("Filenames in similar_documents:", predicted_indices)
        return predicted_indices

    def evaluate_model(model, test_docs):
        # Perform semantic search on the test_docs
        # Extract ground truth indices for evaluation
        true_indices = get_ground_truth_indices(test_docs)

        # For each query, get the predicted indices using semantic_search_and_display
        predicted_indices = get_predicted_indices(model, test_docs)

        # Compute evaluation metrics
        top_k_accuracy_value = top_k_accuracy(true_indices, predicted_indices)
        mean_reciprocal_rank_value = mean_reciprocal_rank(true_indices, predicted_indices)
        precision_at_k_value = precision_at_k(true_indices, predicted_indices)

        # Print evaluation results
        print("")
        print("Evaluation Results:")
        print("=" * 50)
        print("Top-k Accuracy: {}".format(top_k_accuracy_value))
        print("Mean Reciprocal Rank: {}".format(mean_reciprocal_rank_value))
        print("Precision: {}".format(precision_at_k_value))
        print("=" * 50)

    # Evaluate the pre-trained model
    evaluate_model(loaded_model, test_docs)
```