**Question 2:**

This project involves exploring and comparing different text embeddings for a classification task and, optionally, building a semantic search engine. Below is the explanation and code for each part of the project.

**Part 1: Explore and Compare Text Embeddings**

**1. Investigate Different Types of Text Embeddings:**

**- word2vec:** An embedding method that represents words in vector space by predicting words based on their context (Skip-gram) or predicting context based on words (CBOW). It captures semantic relationships between words.

**- GloVe:** A method that learns word vectors based on global word co-occurrence statistics from a corpus. It creates word embeddings by capturing the frequency with which words co-occur.

**- BERT:** Bidirectional Encoder Representations from Transformers. It generates context-aware embeddings by considering the entire context of a word (both left and right). BERT embeddings are more sophisticated and capture deep contextual meanings.

**2. Apply Embeddings to a Text Classification Task**

I use the IMDB dataset for sentiment analysis. This dataset consists of movie reviews labeled as positive or negative.

**3. Compare Performance:**

I will evaluate the performance of the embeddings using metrics such as accuracy, precision, recall, and F1-score.

Here is the complete code to achieve this:

python

# Install necessary libraries

!pip install transformers torch sklearn gensim faiss-cpu

import numpy as np

import pandas as pd

import torch

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

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

from gensim.models import Word2Vec

from gensim.downloader import load

from transformers import BertTokenizer, BertModel

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import Dataset, DataLoader

# Load the IMDB dataset

from sklearn.datasets import fetch\_openml

data = fetch\_openml('imdb', version=1)

texts = data.data['text']

labels = data.target

labels = (labels == 'positive').astype(int) # Convert to binary labels

# Preprocess the text and split the dataset

X\_train, X\_test, y\_train, y\_test = train\_test\_split(texts, labels, test\_size=0.2, random\_state=42)

# Define functions for embeddings

def get\_word2vec\_embeddings(texts, model):

embeddings = []

for text in texts:

words = text.split()

word\_vecs = [model[word] for word in words if word in model]

if word\_vecs:

embeddings.append(np.mean(word\_vecs, axis=0))

else:

embeddings.append(np.zeros(model.vector\_size))

return np.array(embeddings)

def get\_glove\_embeddings(texts, model):

embeddings = []

for text in texts:

words = text.split()

word\_vecs = [model[word] for word in words if word in model]

if word\_vecs:

embeddings.append(np.mean(word\_vecs, axis=0))

else:

embeddings.append(np.zeros(model.vector\_size))

return np.array(embeddings)

class BERTEmbedder:

def \_\_init\_\_(self):

self.tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')

self.model = BertModel.from\_pretrained('bert-base-uncased')

def embed(self, texts):

embeddings = []

for text in texts:

inputs = self.tokenizer(text, return\_tensors='pt', truncation=True, padding=True, max\_length=512)

with torch.no\_grad():

outputs = self.model(\*\*inputs)

embeddings.append(outputs.last\_hidden\_state.mean(dim=1).squeeze().numpy())

return np.array(embeddings)

# Define the classification model

class SimpleNN(nn.Module):

def \_\_init\_\_(self, input\_dim):

super(SimpleNN, self).\_\_init\_\_()

self.fc = nn.Linear(input\_dim, 1)

self.sigmoid = nn.Sigmoid()

def forward(self, x):

x = self.fc(x)

x = self.sigmoid(x)

return x

def train\_model(train\_data, train\_labels, input\_dim):

model = SimpleNN(input\_dim)

criterion = nn.BCELoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

train\_dataset = torch.utils.data.TensorDataset(torch.tensor(train\_data, dtype=torch.float32),

torch.tensor(train\_labels, dtype=torch.float32))

train\_loader = DataLoader(train\_dataset, batch\_size=32, shuffle=True)

for epoch in range(5):

for batch\_data, batch\_labels in train\_loader:

optimizer.zero\_grad()

outputs = model(batch\_data).squeeze()

loss = criterion(outputs, batch\_labels)

loss.backward()

optimizer.step()

return model

def evaluate\_model(model, test\_data, test\_labels):

with torch.no\_grad():

predictions = model(torch.tensor(test\_data, dtype=torch.float32)).squeeze().numpy()

predictions = (predictions > 0.5).astype(int)

accuracy = accuracy\_score(test\_labels, predictions)

precision = precision\_score(test\_labels, predictions)

recall = recall\_score(test\_labels, predictions)

f1 = f1\_score(test\_labels, predictions)

return accuracy, precision, recall, f1

# Load word2vec and GloVe models

word2vec\_model = load('word2vec-google-news-300')

glove\_model = load('glove-wiki-gigaword-100')

# Create embeddings

word2vec\_train = get\_word2vec\_embeddings(X\_train, word2vec\_model)

word2vec\_test = get\_word2vec\_embeddings(X\_test, word2vec\_model)

glove\_train = get\_glove\_embeddings(X\_train, glove\_model)

glove\_test = get\_glove\_embeddings(X\_test, glove\_model)

bert\_embedder = BERTEmbedder()

bert\_train = bert\_embedder.embed(X\_train)

bert\_test = bert\_embedder.embed(X\_test)

# Train and evaluate models

def run\_experiment(train\_data, test\_data, train\_labels, test\_labels, input\_dim):

model = train\_model(train\_data, train\_labels, input\_dim)

accuracy, precision, recall, f1 = evaluate\_model(model, test\_data, test\_labels)

return accuracy, precision, recall, f1

word2vec\_results = run\_experiment(word2vec\_train, word2vec\_test, y\_train, y\_test, word2vec\_train.shape[1])

glove\_results = run\_experiment(glove\_train, glove\_test, y\_train, y\_test, glove\_train.shape[1])

bert\_results = run\_experiment(bert\_train, bert\_test, y\_train, y\_test, bert\_train.shape[1])

# Print results

print("word2vec Results:")

print(f"Accuracy: {word2vec\_results[0]}")

print(f"Precision: {word2vec\_results[1]}")

print(f"Recall: {word2vec\_results[2]}")

print(f"F1 Score: {word2vec\_results[3]}")

print("\nGloVe Results:")

print(f"Accuracy: {glove\_results[0]}")

print(f"Precision: {glove\_results[1]}")

print(f"Recall: {glove\_results[2]}")

print(f"F1 Score: {glove\_results[3]}")

print("\nBERT Results:")

print(f"Accuracy: {bert\_results[0]}")

print(f"Precision: {bert\_results[1]}")

print(f"Recall: {bert\_results[2]}")

print(f"F1 Score: {bert\_results[3]}")

**Part 2: Build a Semantic Search Engine**

1. Create Document Embeddings:

- Use the embeddings from Part 1 to encode a large corpus of documents.

2. Store Embeddings in a Vector Database:

- Use FAISS (Facebook AI Similarity Search) for efficient similarity search.

3. Implement a Search Interface:

- Allow users to input queries and find relevant documents.

4. Provide Case Studies:

- Demonstrate how the search engine works with examples.

So now here’s the code for building the semantic search engine:

python

import faiss

from sklearn.preprocessing import normalize

# Define function to build and search the vector database

def build\_search\_engine(embeddings, dim):

# Normalize embeddings

embeddings = normalize(embeddings)

# Build FAISS index

index = faiss.IndexFlatL2(dim)

index.add(embeddings)

return index

def search\_query(query, index, embedder):

query\_embedding = embedder.embed([query])

query\_embedding = normalize(query\_embedding)

\_, indices = index.search(query\_embedding, k=5) # Top 5 results

return indices

# Example documents

documents = [

"Global warming is a major issue facing the planet.",

"The development of artificial intelligence is rapid.",

"Education is crucial for economic development.",

"Renewable energy sources are essential for sustainable growth.",

"Healthcare improvements can lead to a better quality of life."

]

# Build the search engine with document embeddings

bert\_embedder = BERTEmbedder()

doc\_embeddings = bert\_embedder.embed(documents)

index = build\_search\_engine(doc\_embeddings, doc\_embeddings.shape[1])

# Search for a query

query = "What are the benefits of renewable energy?"

indices = search\_query(query, index, bert\_embedder)

print("\nSearch Results:")

for i in indices[0]:

print(documents[i])

**Summary of Findings**

- word2vec: Provides word embeddings based on context but lacks deep contextual understanding.

- GloVe: Learns embeddings from word co-occurrence but may not capture context as effectively as BERT.

- BERT: Captures deep contextual meanings and generally provides the best performance for text classification tasks.

In building the semantic search engine, using BERT embeddings enhances search accuracy and relevance. The FAISS library efficiently handles large-scale similarity searches, making it suitable for real-world applications.

This code and approach offer a comprehensive way to evaluate text embeddings and implement a practical search engine, demonstrating the versatility and effectiveness of different NLP techniques.