Sentence Embedding & API Deployment Report

# Overview

In this exercise, I worked with the HuggingFace `transformers` library to generate sentence embeddings using the pre-trained model `sentence-transformers/paraphrase-mpnet-base-v2`. Sentence embeddings are numerical representations of text that capture semantic information and can be used for tasks like semantic search, clustering, or sentence similarity.

# Step-by-Step Breakdown

1. Tokenization: I used `AutoTokenizer` to tokenize a list of sentences.

2. Model Loading: I loaded the pre-trained model from HuggingFace.

3. Model Inference: Used `torch.no\_grad()` to prevent gradient computation.

4. Mean Pooling: Averaged token embeddings using the attention mask.

5. Output: Printed 768-dimensional sentence embeddings.

# Pooling Strategy: Mean Pooling

The `mean\_pooling` function ensures that only non-padding tokens contribute to the final embedding, improving semantic accuracy.

# If I Had to Build This from Scratch

# 1. Data Collection

# I would first gather a large dataset of sentence pairs with semantic similarity scores, such as SNLI, STS-B, or Quora Question Pairs.

# 2. Model Architecture

# I would implement a transformer-based architecture (similar to BERT or MPNet) using PyTorch or TensorFlow, including:

# Token & positional embeddings

# Multi-head self-attention layers

# Feed-forward layers

# Layer normalization and residual connections

# 3. Training Objective

# I would train using a contrastive learning objective (e.g., triplet loss or cosine similarity loss), ensuring semantically similar sentences are closer in vector space.

# 4. Pooling Layer

# After obtaining final hidden states, I’d use mean pooling, max pooling, or [CLS] token embedding for fixed-size vectors.

# 5. Evaluation

# Evaluation would be on downstream tasks like semantic similarity or clustering using benchmarks such as the STS Benchmark.

# Reflection

Using pre-trained models like paraphrase-mpnet-base-v2 significantly accelerates NLP experimentation and deployment by providing high-quality semantic representations out of the box. However, understanding internals like attention, pooling, and training objectives gives me the confidence to customize or build models for specialized applications when needed.

# 🚀 Part B: Deployment of Sentence Similarity Model as an API

# Core Approach

I deployed the sentence similarity logic from Part A as a RESTful API using FastAPI. Clients can submit two input sentences and receive a real-time similarity score.

# Steps Taken

1. API Development: Created a POST `/similarity` endpoint.  
2. Model Loading and Caching: Model is cached for performance.  
3. Hosting: Deployed on a self-hosted Linux server using `uvicorn`.  
4. Request Format:  
```  
{  
 "text1": "nuclear body seeks new tech",  
 "text2": "terror suspects face arrest"  
}  
```  
5. Response Format:  
```  
{  
 "similarity score": 0.2174  
}  
```  
6. Similarity Computation: Used cosine similarity with score normalization.  
7. Error Handling: Input validation and status codes.  
8. Testing: Validated using Postman, curl, browser.

# Submission Contents

- Live API: http://207.148.78.17:8005/similarity  
- Source Code: Includes model and deployment scripts.  
- This Report: Methodology and reflection.

# 📌 Reflection

Deploying this model as a real-time API made it immediately useful for downstream tasks. Key benefits include:

Performance: Model caching and CPU-friendly inference provide low-latency responses.

Scalability: FastAPI’s async support and uvicorn allow concurrent request handling.

Portability: The setup is easy to deploy across cloud or local environments.

# Challenges and Limitations:

* Pre-trained models are not domain-adaptive.
* Only one endpoint is supported.
* No authentication or rate-limiting is implemented.
* Time constraints (2 days) prevented more advanced optimization.