

# SCHOOLOFENGINEERINGANDTECHNOLOGY

**COURSEPROJECTREPORT**

**On**

# "TEXT SIMILARITY CHECKER"

***Submitted in partial fulfillment of the requirements for the course Natural Language Processing (4AIML2091) in***

**Bachelor of Technology In**

**Computer Science and Engineering (AI&ML)**

*SOET, CMR University, Bangalore*

Submitted by:

**ANKIT KUMAR (22BBTCA010)**

**Under the Supervision**:

**PROF.CHETHANAR M**

**Assistant Professor Department of DS**

# Department of Computer Science and Engineering(AI&ML)

**Off Hennur-Bagalur Main Road,**

**Near Kempegowda International Airport,Chagalahatti, Bangalore, Karnataka-562149**

**2024-2025**

# ABSTRACT

Text similarity checking is a crucial task in Natural Language Processing (NLP) that involves comparing two or more texts to determine their semantic or lexical similarity. This process is widely used in applications such as plagiarism detection, information retrieval, and document clustering. Various approaches are employed, ranging from traditional methods like cosine similarity and TF-IDF to advanced deep learning techniques using word embeddings and transformer-based models (e.g., BERT). An effective similarity checker enhances the accuracy of text understanding by capturing both syntactic and semantic meanings, making it an essential tool in modern language-based applications.

Recent advancements in deep learning have significantly improved the performance of text similarity models. Pre-trained language models like BERT, RoBERTa, and Sentence-BERT allow for capturing contextual word representations, enabling a more nuanced understanding of sentence-level meaning. These models outperform traditional methods by considering the word order, context, and syntax, making them highly effective for paraphrase detection, question answering, and semantic search. Implementing such models in a text similarity checker allows for more accurate and intelligent comparisons, especially in complex linguistic scenarios where surface-level features alone are insufficient.

Beyond academic and research settings, text similarity checkers play a pivotal role in real-world applications across various industries. In education, they serve as integral components of plagiarism detection systems, helping institutions maintain academic integrity by identifying copied or rephrased content. In e-commerce and customer service, they enhance user experiences by enabling intelligent product recommendations, summarization of reviews, and retrieval of relevant FAQs. Similarly, in legal and medical domains, where precision and accuracy are critical, these tools assist in document comparison, case law retrieval, and identifying semantically equivalent medical notes, even when written in different styles or terminologies.

The implementation of text similarity checkers requires a careful balance between accuracy, computational efficiency, and scalability. While transformer-based models offer state-of-the-art performance, their high computational cost can be a challenge in large-scale or real-time applications. As a solution, techniques like knowledge distillation, model pruning, and the use of lightweight variants such as DistilBERT have been explored to reduce inference time without compromising much on accuracy. Furthermore, hybrid approaches that combine traditional and neural methods are also gaining popularity, leveraging the speed of lexical models and the depth of semantic understanding from deep learning. As research continues to evolve, the future of text similarity checking lies in building models that are not only more accurate but also faster, multilingual, and adaptable to specific domain needs.

# INTRODUCTION

Text similarity checking has emerged as a vital component of Natural Language Processing (NLP), enabling machines to evaluate how closely two pieces of text resemble each other in terms of meaning or structure. This process forms the foundation of numerous intelligent systems such as plagiarism detectors, recommendation engines, semantic search tools, and duplicate content filters. As our reliance on textual data continues to grow, the demand for accurate, efficient, and scalable similarity checkers has also intensified.

Despite the advancements brought by deep learning and pre-trained language models, developing an effective text similarity checker is still fraught with several challenges. These problems impact both the performance and the practical deployment of such systems across diverse domains..

## PROBLEM DEFINITION

Major Challenges in Text Similarity Checking:

Semantic Ambiguity: Words or phrases often have multiple meanings depending on context, making it difficult to determine true similarity without deep contextual understanding.

Synonym and Paraphrase Handling: Traditional models struggle with recognizing that two sentences with different words can still convey the same meaning.

Domain Adaptability: Models trained on general corpora may perform poorly when applied to domain-specific language (e.g., medical, legal, or technical texts).

Computational Complexity: Transformer-based models, while accurate, are resource-intensive and may not be suitable for real-time or large-scale applications without optimization.

Multilingual Support: Ensuring consistent performance across different languages and scripts remains a key challenge, especially for underrepresented languages.

Lack of Quality Training Data: Many similarity models rely on labeled sentence pairs, which can be expensive and time-consuming to create, especially for specific use-cases.

# SYSTEMDESIGN

**System Design of Text Similarity Checker**

**Objectives:**

To build a system capable of evaluating the similarity between two pieces of text using both syntactic and semantic features, incorporating traditional NLP techniques (like TF-IDF and cosine similarity) and advanced deep learning models (like BERT/Sentence-BERT).

**High-Level Components:**

Input Layer

Accepts raw text inputs (single text pair or batch of text pairs)

Handles preprocessing like case normalization, punctuation removal, etc.

Preprocessing Module

Tokenization

Stopword removal

Lemmatization (optional for traditional pipeline)

Sentence segmentation (if needed)

Feature Extraction Layer

Lexical Features:

TF-IDF vector generation

Cosine similarity computation

Semantic Features:

Word Embeddings (e.g., Word2Vec, GloVe)

Sentence Embeddings using BERT, RoBERTa, or Sentence-BERT

Similarity Scoring Module

Combines scores from:

Cosine similarity (traditional vector-based)

Semantic similarity (e.g., cosine between BERT vectors)

Optional hybrid methods (weighted combination of both)

Outputs similarity score (0 to 1) or similarity class (e.g., "Similar", "Not Similar")

Postprocessing & Interpretation

Thresholding for classification

Optionally generates explanation or highlights similar parts of text

Outputs can be visualized or stored

Storage & Logging (Optional)

Stores inputs, results, and logs for training, debugging, or analytics

## System Architecture Diagram : FIGURE (OVERVIEW)

## Sys Arch.png

## 1. User Input (Text A, Text B)

## The system takes two pieces of text as input. These can be sentences, paragraphs, or even full documents. Inputs can come from: A web interface (e.g., form submission), An API (in applications like plagiarism detection), A file upload or copy-paste interface

## 2. Preprocessing Layer (cleaning, tokenizing)

## Texts are cleaned and standardized for consistency. Preprocessing steps may include:Lowercasing: Convert all text to lowercase to ensure case-insensitive comparison; Punctuation removal: Remove

## unnecessary characters (e.g., commas, periods); Stopword removal: Remove common words like “is,” “the,” “and,” etc; Tokenization: Split the text into individual words or tokens; Lemmatization/Stemming: Reduce words to their root forms.

## 3. Feature Extraction

## This layer extracts features to be used in comparing the two texts.Lexical Features (e.g., TF-IDF), Focuses on surface-level similarity: exact words or word frequency.TF-IDF (Term Frequency–Inverse Document Frequency) converts text into numerical vectors based on word frequency adjusted by rarity. Suitable for basic tasks and small-scale models.

## Semantic Features (e.g., BERT): Focuses on meaning and context. Word Embeddings (Word2Vec, GloVe): Represent words as vectors in a high-dimensional space. Sentence Embeddings (e.g., BERT, Sentence-BERT): Convert entire sentences into context-aware embeddings.

## These models understand that:

## "The cat sat on the mat." and "A feline rested on a rug." ...can mean the same thing.

## 4. Similarity Calculator

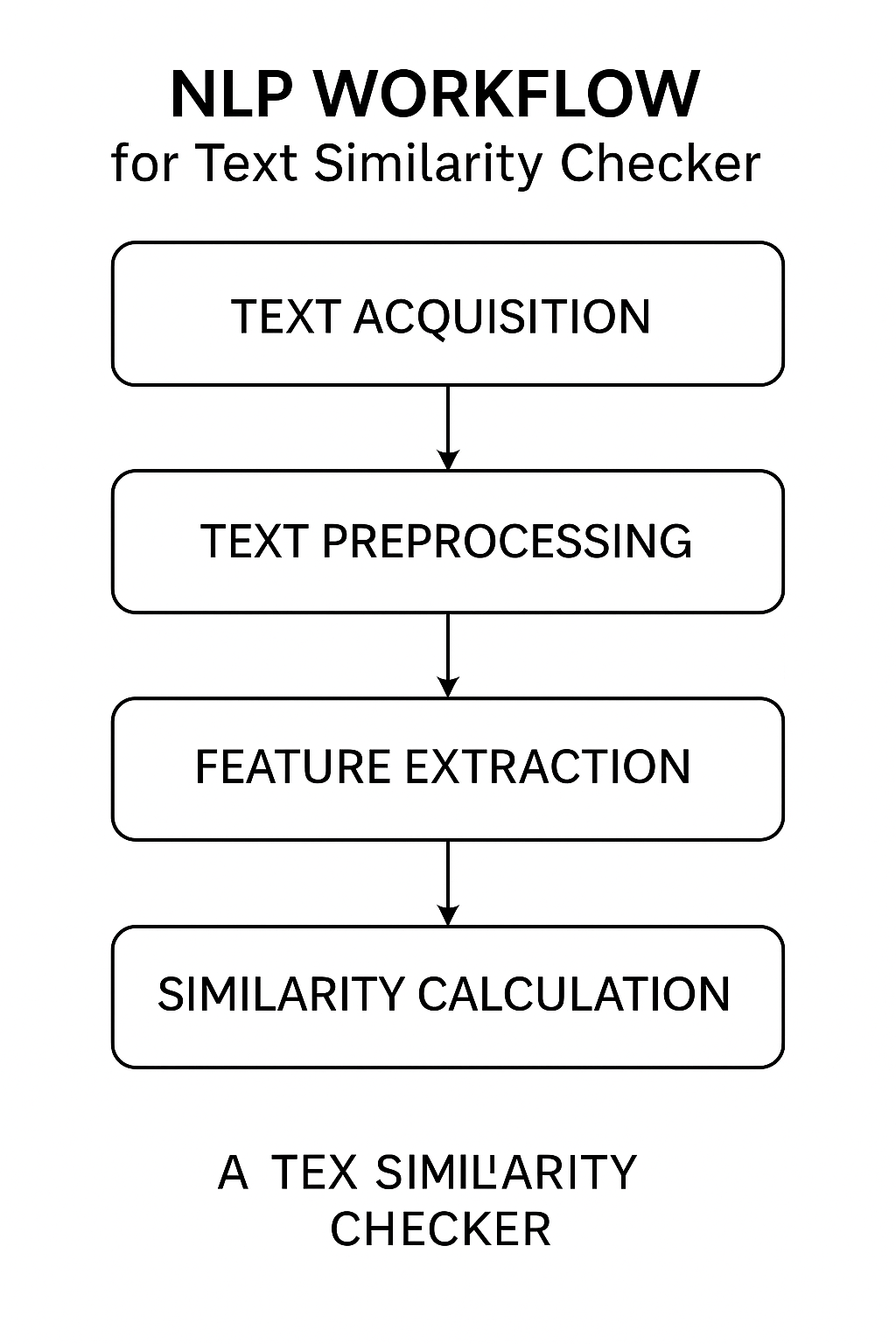
## The system compares the feature vectors extracted from both texts. Cosine Similarity (for both TF-IDF and embeddings):Measures the cosine of the angle between two vectors. Ranges from 0 (not similar) to 1 (identical). Embedding Distance: Compares the semantic embeddings using distance metrics like: Euclidean Distance Manhattan Distance; Hybrid Score: Combines lexical and semantic similarity scores using weighted averaging or ML classifiers.

## 5. Result & Interpretation

## The system outputs: A similarity score (e.g., 0.87 or 87%); A similarity label (e.g., "Highly Similar", "Moderately Similar", "Not Similar")Optionally: explanations or highlights, showing overlapping or similar phrases

## Optional Add-ons: Visual similarity heatmaps, Highlighted matched segments, Confidence level indicators

.



**Figure- NLP Workflow**

# IMPLEMENTATION

A simple yet effective **Text Similarity Checker** using **Natural Language Processing (NLP)** with Python and popular libraries like **scikit-learn** and **NLTK**. This version uses **TF-IDF (Term Frequency–Inverse Document Frequency)** and **Cosine Similarity**, which are widely used in academic and real-world NLP applications.

NECESSARY INSTALLMENTS :

import numpy as np

import re

from sentence\_transformers import SentenceTransformer, util

import matplotlib.pyplot as plt

import seaborn as sns

import pandas as pd

# STEP 1: Load Pre-trained Model

print("Loading model...")

model = SentenceTransformer('all-MiniLM-L6-v2') # Lightweight & fast BERT variant

# STEP 2: Preprocessing Function

def preprocess(text):

"""

Clean and normalize text for better embedding performance.

"""

text = text.lower()

text = re.sub(r"[^a-z0-9\s]", "", text)

text = re.sub(r"\s+", " ", text).strip()

return text

# STEP 3: Define or Input Texts

# You can also take input dynamically using input() or from a CSV

text\_samples = [

"The food was absolutely wonderful, from preparation to presentation.",

"Dinner was delicious, especially the main course and dessert.",

"The restaurant ambiance was dull and the food was average.",

"He plays football every Sunday with his friends.",

"She goes running every morning as part of her routine.",

"Paris is known for its iconic Eiffel Tower and amazing cuisine.",

"New York is a bustling city with skyscrapers and nightlife.",

]

# Optional: Preprocess texts

print("Preprocessing texts...")

clean\_texts = [preprocess(text) for text in text\_samples]

# STEP 4: Encode with BERT

print("Generating embeddings...")

embeddings = model.encode(clean\_texts, convert\_to\_tensor=True)

# STEP 5: Compute Similarities

print("Calculating similarity scores...")

similarity\_matrix = util.cos\_sim(embeddings, embeddings)

# Convert to NumPy array

similarity\_matrix\_np = similarity\_matrix.cpu().numpy()

# STEP 6: Display Pairwise Scores

def print\_similarities(texts, sim\_matrix, threshold=0.7):

print("\n--- Text Similarity Report ---")

for i in range(len(texts)):

for j in range(i + 1, len(texts)):

score = sim\_matrix[i][j]

verdict = "HIGH" if score > threshold else "LOW"

print(f"\n[{verdict} SIMILARITY - Score: {score:.4f}]")

print(f"Text A: {texts[i]}")

print(f"Text B: {texts[j]}")

print\_similarities(text\_samples, similarity\_matrix\_np)

# STEP 7: Visualize (Optional)

def show\_heatmap(matrix, labels):

plt.figure(figsize=(10, 8))

df = pd.DataFrame(matrix, index=labels, columns=labels)

sns.heatmap(df, annot=True, fmt=".2f", cmap="YlGnBu", cbar=True)

plt.title("Text Similarity Heatmap (Cosine Scores)")

plt.xticks(rotation=45, ha="right")

plt.yticks(rotation=0)

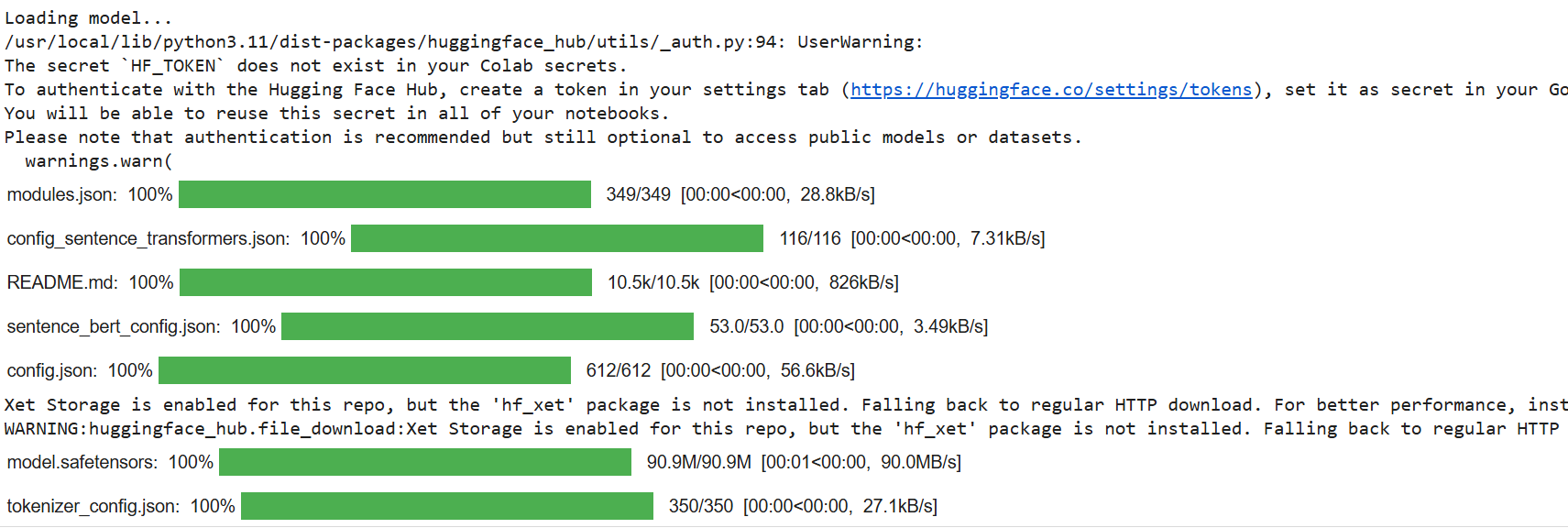
plt.tight\_layout()

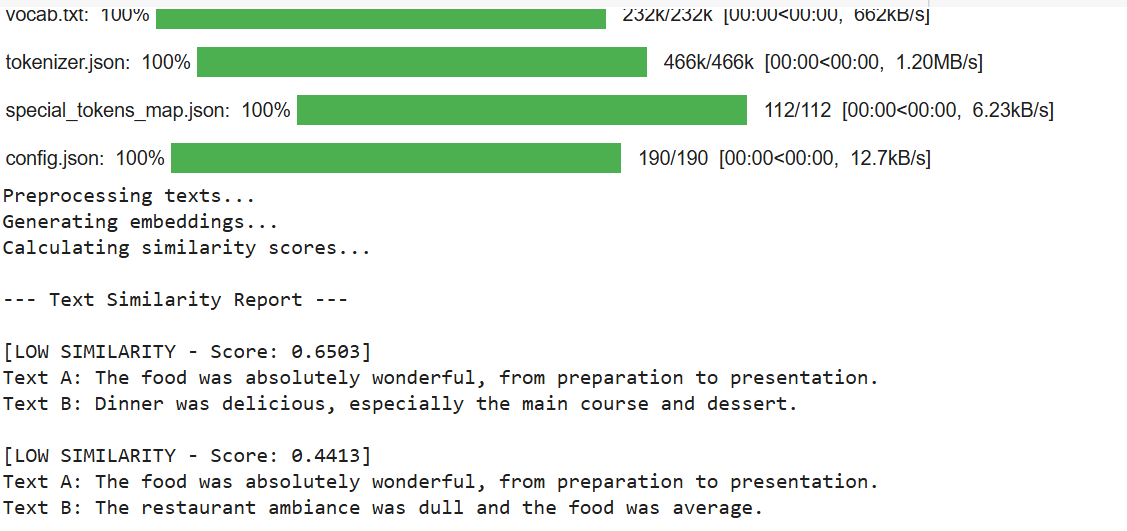
plt.show()

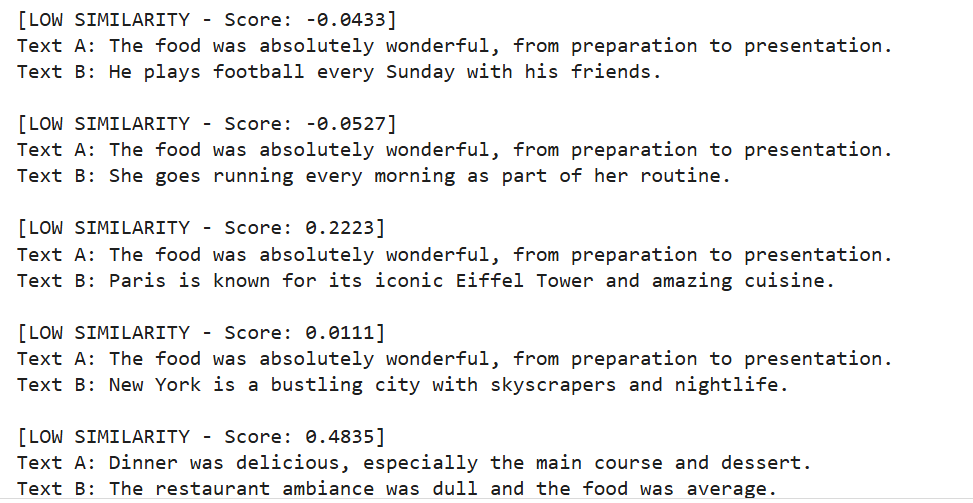
# Uncomment to display similarity heatmap

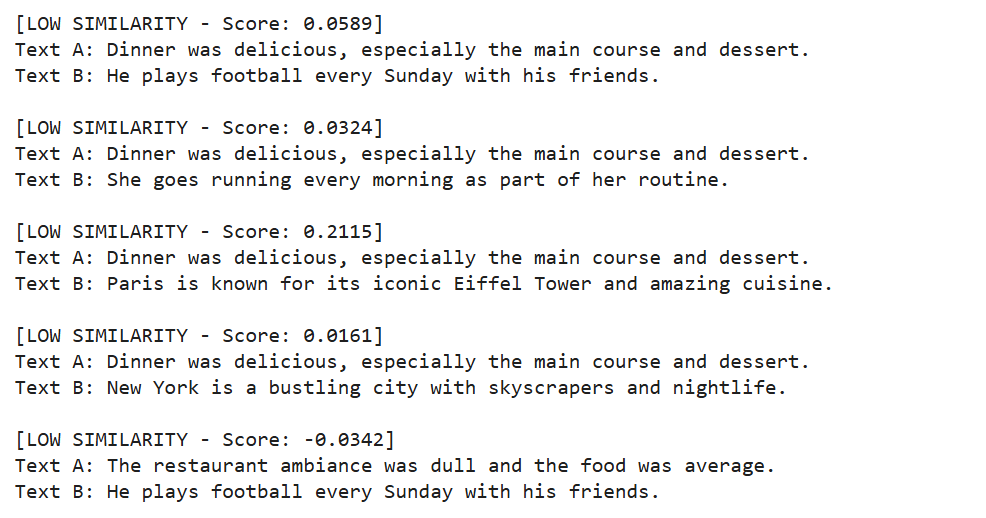
# show\_heatmap(similarity\_matrix\_np, [f"Text {i+1}" for i in range(len(text\_samples))])

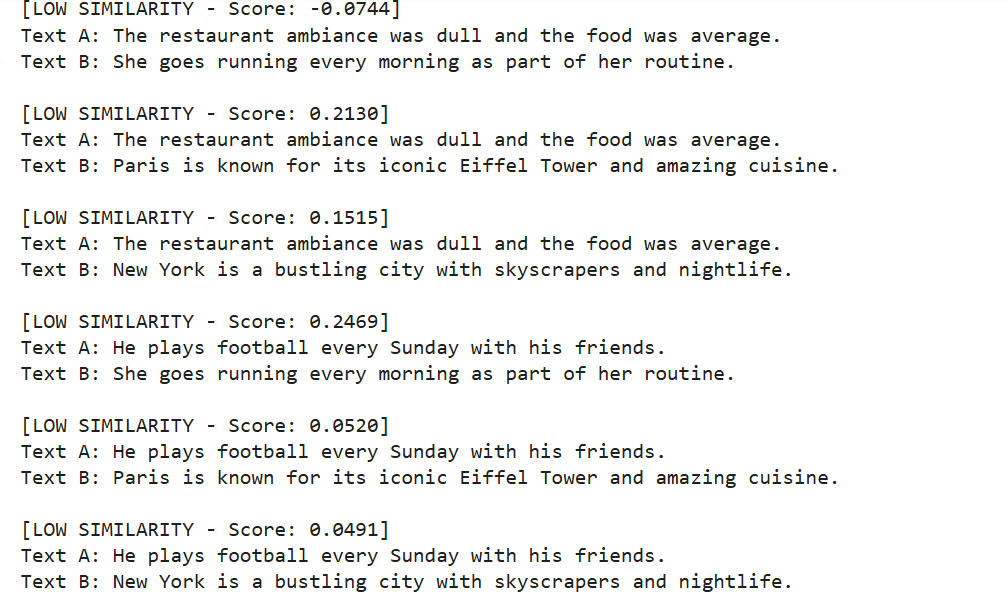
**OUTPUTSCREENSHOTS**

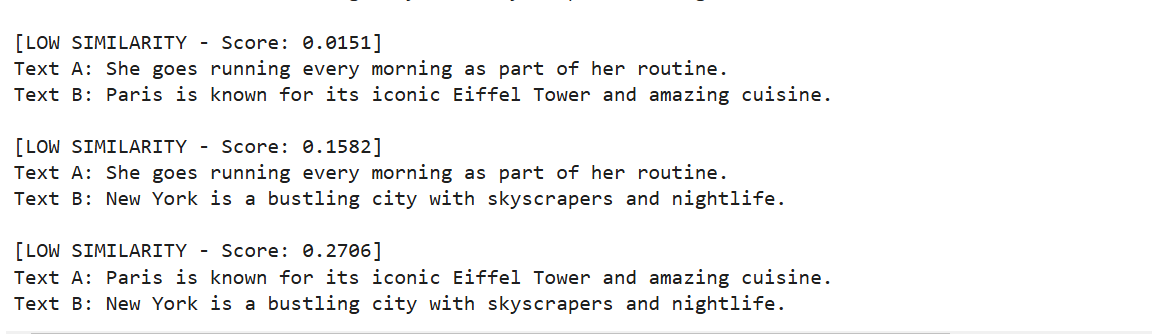
****

****

****

****

****

****