TEXT SIMILARITY - NLP

Part A

1. Introduction

In this task, we have to compute the similarity between pairs of texts provided in a CSV file named "DataNeuron_Text_Similarity.csv". The goal is to preprocess the text data, transform it using a SentenceTransformer model, and calculate the cosine similarity between the transformed embeddings of each pair of texts.

SentenceTransformer was chosen primarily for its ability to generate high-quality sentence embeddings efficiently. While other transformer models like BERT, RoBERTa, and DistilBERT can also produce embeddings, SentenceTransformer is specifically designed and pre-trained for generating embeddings for entire sentences or paragraphs. This makes it well-suited for tasks like semantic similarity comparison between text pairs.

Additionally, SentenceTransformer offers a variety of pre-trained models fine-tuned on different tasks and datasets, providing flexibility to choose a model that best fits the specific requirements of the task at hand. It also allows for easy integration and usage through a simple API.

I had also tried BERT here but due to the limitation of BERT in handling text with more than 512 tokens. Since the dataset contained rows with text longer than 512 tokens, BERT was not suitable for this task. SentenceTransformer, on the other hand, does not have this limitation and can handle longer text sequences more effectively.

Cosine similarity is commonly used in text similarity tasks because it is scale-invariant, computationally efficient, and provides a clear interpretation of similarity. It is robust to sparse data and effectively captures semantic relationships between text vectors, making it suitable for comparing the similarity between pairs of texts.

2. Data Loading and Preprocessing

We start by loading the dataset from the provided CSV file using the Pandas library. The dataset contains two columns: "text1" and "text2", representing pairs of texts that we need to compare for similarity.

Next, we preprocess the text data using several steps:

- Lowercasing: Convert all text to lowercase to ensure consistency.
- Removing punctuation: Remove any punctuation marks from the text.
- Removing numbers: Remove numerical digits from the text.
- Tokenization: Tokenize the text into individual words using NLTK's word_tokenize function.
- Removing stopwords: Remove common stopwords from the text using NLTK's stopwords corpus.

- Lemmatization: Lemmatize the words to their base form using NLTK's WordNetLemmatizer.

3. Text Transformation using SentenceTransformer

We utilize the SentenceTransformer library to transform the preprocessed text data into numerical embeddings. The SentenceTransformer model used for this task is 'paraphrase-MiniLM-L6-v2'. The embeddings are calculated for each pair of texts in the dataset.

4. Similarity Calculation

Once we have the embeddings for each pair of texts, we calculate the cosine similarity between the embeddings to determine the similarity score between the text pairs.

5. Output

The final output is saved as a CSV file named "OutputPartA.csv", which contains the original text pairs along with their preprocessed versions and the calculated similarity scores.

6. Fine Tuning

In this case, since we're using a pre-trained model (paraphrase-MiniLM-L6-v2) and not doing any additional training on the dataset, there's no need to save the model after passing your dataset to it.

The pre-trained model can be loaded directly in our Flask application (as we have already done in app.py script) and have used to compute similarity scores for any input texts.

Moreover, I have also tried fine-tuning the pre-trained model on this dataset (i.e., further training the pre-trained model on this dataset to better adapt it to the specific task), But since I was using a Goggle Colab for this project for Fine-tuning task I had changed runtime environment from CPU to T4 GPU but it took too much time to process, as time is constraints for this project I have decided to drop the idea of fine-tuning. However Fine tuning often leads to better performance, but it's also more computationally intensive.

In summary, for Part B, you can continue to use the pre-trained model directly to compute similarity scores for the test data. No changes are needed in your current setup for this.

7. Conclusion

Part A of this task successfully preprocesses the text data, transforms it into numerical embeddings using the SentenceTransformer model, and computes the cosine similarity between pairs of texts. The output CSV file provides insights into the similarity between text pairs, which can be further analyzed for various natural language processing tasks. I have uploaded the output file in the Data directory and also uploaded **text_similarity.py** file which belongs to Part A of the project.

Semantic Textual Similarity – Deployment -Part B:

In Part A, we successfully built a model that quantifies the degree of similarity between two pieces of text based on Semantic Textual Similarity (STS). The model was trained on a dataset containing pairs of paragraphs, and it predicts a value between 0-1 indicating the similarity between the pair of text paragraphs.

In Part B, the task was to deploy the model built in Part A on a cloud service provider and expose it as a Server API Endpoint. For this task I have used Flask Api.

1. Flask API

To expose our model as an API, we used Flask, a lightweight web server framework for Python. I had created two files: app.py and test_api.py.

The app.py file contains our Flask application. It loads the SentenceTransformer model, preprocesses the input text, and defines a route /similarity that accepts POST requests. When a request is received, it calculates the semantic textual similarity between the two input texts and returns the similarity score in response. The test_api.py file is used to test our Flask API. It sends a POST request to the /similarity route of our Flask application and prints the request and response bodies.

Both app.py and test_api.py were tested locally and worked as expected, returning the desired output.

2. Deployment Challenges

I had attempted to deploy our Flask application on these cloud platforms - Heroku, PythonAnywhere, Koyeb, Choreo, Render, Railway. However, we faced a significant challenge: the SentenceTransformer model we used is quite large, exceeding 4GB in size. Furthermore, I have already utilized all my free credits of Azure, AWS, and GCP, which could support the deployment of this large file.

Unfortunately, none of the free tiers of the cloud platforms I tried provide this much storage. As a result, we were unable to upload our model to these platforms.

3. Conclusion

While I was successful in building the Flask API and testing it locally, I was unable to deploy it on a cloud platform due to storage limitations. I apologize for this inconvenience. In the future, I will plan to explore other deployment options that can accommodate larger models, or consider using a smaller model that still delivers acceptable performance.