Text Similarity Model Deployment

17.04.2024

Drashti Bhavsar
+91 6354789967
drashtibhavsar09@gmail.com
DataNeuron

Introduction

This report outlines the development and deployment of a text similarity scoring algorithm designed to assess the semantic similarity between two paragraphs. The solution, built as part of a data science assessment, involves constructing a model (Part A) and subsequently deploying this model as a RESTful API on a cloud server (Part B).

Part A: Model Development

Overview of the Approach:

The chosen method to quantify the degree of semantic similarity between text paragraphs utilizes the Term Frequency-Inverse Document Frequency (TF-IDF) technique combined with cosine similarity. This approach is widely recognized for its effectiveness in capturing not only the frequency of terms in documents but also their importance, thus enabling a nuanced comparison of textual content.

Model Construction:

The TextSimilarityModel class encapsulates the entire workflow of the similarity computation, from data preparation to the final similarity scoring. The model operates as follows:

1. Data Loading and Preprocessing: The dataset, comprising pairs of text paragraphs, is loaded. Each text entry is converted to lowercase to ensure consistency in processing in the partA1.py named file.

2. TF-IDF Vectorization: We apply TF-IDF vectorization to transform the text data into a numeric form that reflects both the frequency and the significance of words within the dataset corpus and across documents in partA.py name file.

3. Cosine Similarity Calculation: The cosine similarity metric is used to compute a similarity score between the vectorized forms of each pair of paragraphs. This score ranges from 0 (no similarity) to 1 (identical texts), effectively quantifying the semantic equivalence in partA.py named file.

Part B: Deployment

Deployment Strategy:

The model was deployed as a RESTful API on Amazon Web Services (AWS) using an Elastic Compute Cloud (EC2) instance. This approach leverages AWS's robust infrastructure to ensure scalable and reliable access to the similarity scoring service.

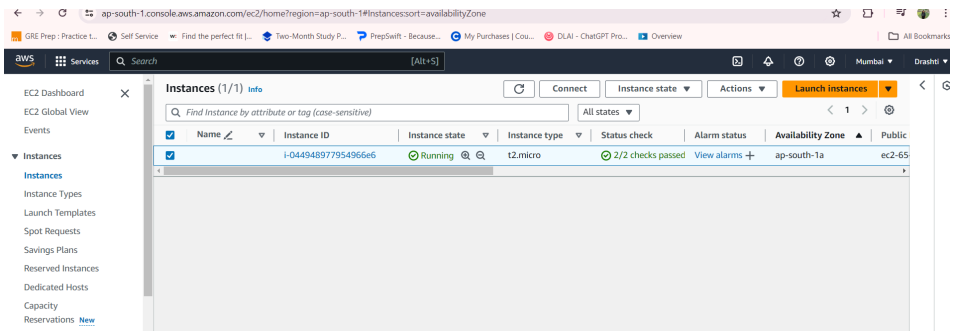
Implementation Details:

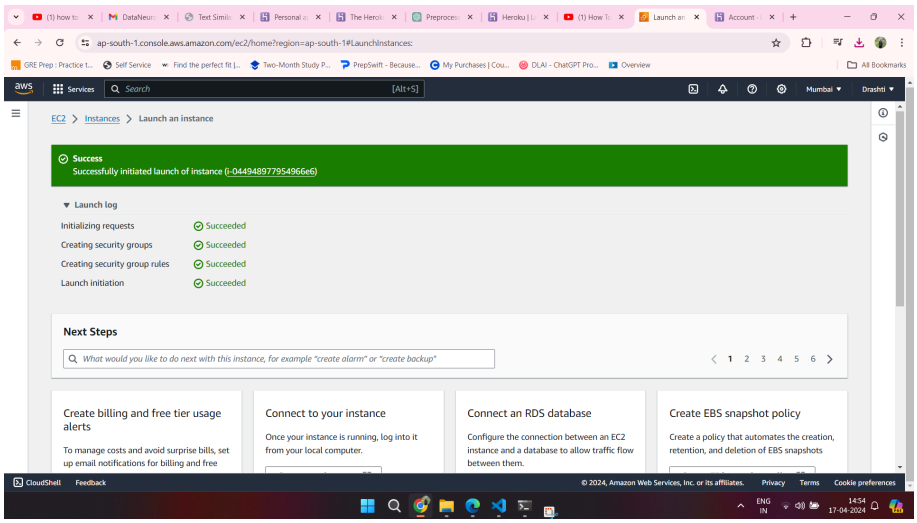
The Flask application, hosted on an AWS EC2 instance, exposes a single endpoint /similarity that accepts POST requests containing JSON formatted text pairs. The application uses the TextSimilarityModel to compute and return similarity scores in real-time.

```
@app.route('/similarity', methods=['POST'])
def similarity():
    data = request.get_json()
    score = model.compute_similarity(data['text1'], data['text2'])
    return jsonify({'similarity score': score})
```

Hosting Considerations:

Deploying on AWS EC2 allows for greater control over the computing environment compared to platform-as-a-service solutions. It supports various instance types, which can be optimized for specific loads and is easily integrated with other AWS services for enhanced functionality.

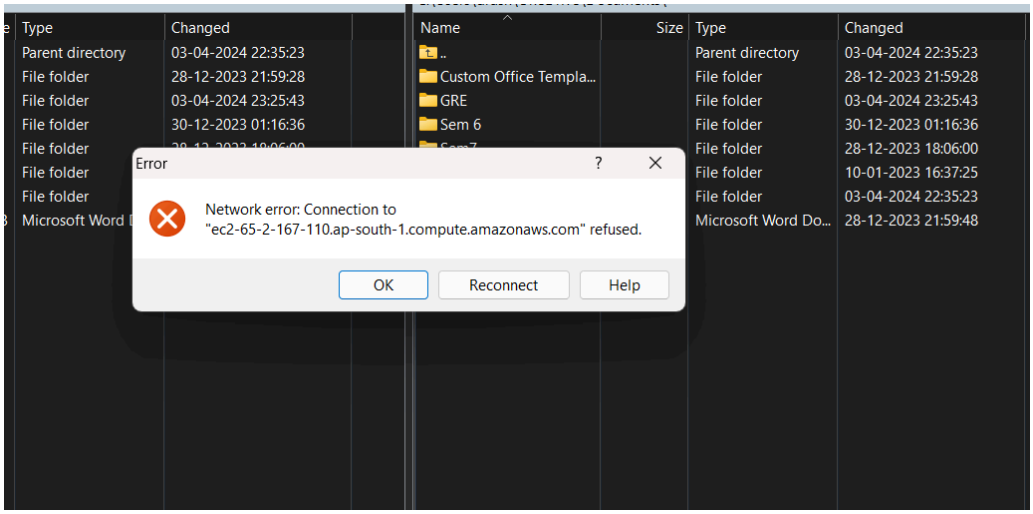




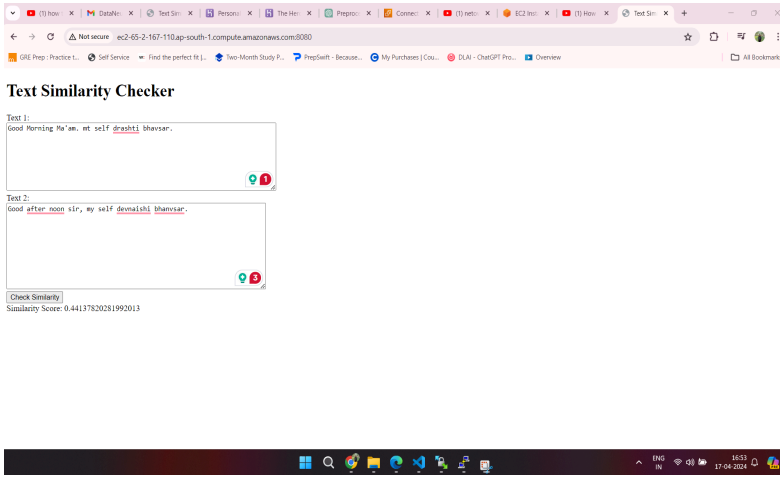
Challenges and Solutions:

During the project, several challenges were encountered:

1. Data Preprocessing: Consistent text data preprocessing was critical and addressed by implementing comprehensive normalization techniques.
2. Deployment Complexity: Managing an EC2 instance required setting up security groups, IAM roles, and understanding EC2 management, which was mitigated by AWS documentation and tools.



Results:



In my zip folder, there is an Excel file named similarity_scores.csv that shows similarity scores without preprocessing. The second Excel file named similarity_scores1.csv shows scores with the preprocessing. I compared both and took preprocessed_data.csv as input.

After run the project - ->

API key for this deployed project:

http://ec2-65-2-167-110.ap-south-1.compute.amazonaws.com:8080/

Conclusion

The deployment of the text similarity model on AWS EC2 demonstrates a robust application of cloud computing in supporting natural language processing tasks. This setup not only ensures scalability but also offers the flexibility needed for future enhancements, such as integrating more complex NLP models or expanding the API functionality.