**PRECOG RECRUITMENT TASK**

**Task: Representations for Words, Phrases, Sentences (NLP)**

SUBMITTED BY: ANJANA

**OVERVIEW OF THE REPORT:**

In this report, I finished the following tasks:

1. Word similarity (ii) Unconstrained
2. (ii) Sentence similarity
3. Bonus Task (ii) Prompting LLMs and (iii) Comparing the approaches
4. Paper Reading task

Skipped tasks:

1. I couldn't complete the word similarity with constraints on data sources (a.(i)) as I was stuck on how to convert words into vectors by not using pre- trained models and iwas unsure of how to apply unsupervised learning methods on the vector embeddings. I did give it a try but due to time constraints I had to drop it off.
2. Phrase Similarity (b.(i)) : I tried converting the words present in phrases into vectors using Word2vec , but since the dataset had sentences too, i got distracted into applying it for sentences since a sentence is a huge phrase which has meaning on its own. My bad!
3. Usage of transformers in word/sentence similarity (c(i)) : I tried fine tuning BERT for similarity score generation but i left it midway purely cause of lack of time as it was my first time working with transformers, so I had to figure out a lot of stuff.

P.S: The last two weeks have been pretty jam packed because the semester is ending and we were bombarded with assignments and projects. So i tried my level best to figure things out in the task and im pretty sure i’d have done more if not for the situation.

Thank you for this opportunity!

1. **PROBLEM STATEMENT: Word Similarity Scores**

**METHOD:**

1.**Data Preprocessing**- Load the **Brown corpus** and extract the tokenized text. Remove punctuation from each sentence in the corpus. **Preprocess the text** for Word2Vec model training.

2.**Word2Vec Model Training**: Train a Word2Vec model on the preprocessed text. Save the trained model to disk.

3. Load the saved Word2Vec model.

4. Calculate Similarity Scores: Load the **SimLex-999 dataset**. Calculate the **similarity scores between the words** in the dataset using the **Word2Vec model.**

5. Scale the Similarity Scores: Scale the calculated similarity scores and the SimLex-999 scores to a **range of 0 to 1**.

6. **Evaluate the Model**: Calculate the mean squared error (MSE) and mean absolute error (MAE) between the scaled similarity scores and the SimLex-999 scores.

**INFERENCES:**

This model was initially trained using the Brown corpus dataset and then the simlex-999 dataset was used to test it. After testing, the mean square error (MSE) was found to be 0.191. As lower MSE values indicate better performance, this model produces results with decent accuracy. On average the absolute difference between the actual and predicted values is approximately 0.365, this is the MAE.

As this model doesn’t consider the context in which the word is used and only depends on vector embedding, it is a rigid model. Models which consider the context of the word may be more flexible than this word-2-vec model.

Dataset used: brown corpus and simlex999

Link: <https://colab.research.google.com/github/anjana-psvel/Precog_Task/blob/main/word_similarity.ipynb#scrollTo=RL7rKN4zs2z7>

1. **PROBLEM STATEMENT: Sentence Similarity**

**METHOD:**

1.**Import Libraries**: The code starts by importing the necessary libraries for data manipulation, sentence embedding, and machine learning tasks.

2. Load Data: The code then reads a CSV file containing **sentence pairs and their corresponding labels** using pandas.

3. Sentence Embedding: The code uses the SentenceTransformer library to create vector representations for each sentence. These **embeddings capture the semantic meaning of the sentences.**

4. Similarity Calculation: The code calculates the **cosine similarity between the sentence embeddings of each pair**. This similarity score indicates how similar the sentences are in terms of their meaning.

5. The calculated similarity scores are inserted into the DataFrame as a new column.

6. **Logistic Regression**: The code then trains a logistic regression model using the similarity scores as input features and the labels as output labels. This model is used to predict whether a given sentence pair is similar or not.

7. **Decision Tree and SVM**: The code also trains and evaluates decision tree and support vector machine models using the same data.

8. The code evaluates the performance of the logistic regression model by **calculating the accuracy on the test set**.

9. Normalization: The code normalizes the similarity scores using **z-score** normalization and then scales them to a **range between 0 and 1**.

10. Mean Squared Error: The code calculates the mean squared error (MSE) of the predictions made by the logistic regression model.

**INFERENCES:**

To test for sentence similarity, first the sentence transformers framework was used to create the vector embeddings of each word. A pretrained model named ‘all-MiniLM-L6-v2’ was used as a baseline. This was further trained using the ‘train.csv’ model, which contains two sentences and a label - 0 or 1 - indicating whether they are similar sentences or not. Using this model vector embeddings of the sentence pairs were found and the cosine similarity between the two vectors were calculated. This similarity score was scaled between 0 to 1. Keeping ‘label’ as the dependent variable and ‘sim\_score’ as the independent variable, the model was trained using logistic regression, decision tree and SVM providing accuracies of 0.56, 0.54 and 062 respectively. The area under the curve (AUC) came out to be 0.63. The label scores were either 0 or 1 whereas the similarity scores were within the range 0-1. Binary classification couldn’t be done for the similarity score, this is because the sentences in the dataset were very similar and only varied by one word, but this change altered the meaning of the entire sentence. This was not reflected in the cosine similarity which was calculated. This was a major drawback of this model.

Datasets used: train.csv

Link:<https://colab.research.google.com/drive/1ySEBdLkAXUB6Uchi5uVymj6fWwZUMQme#scrollTo=udX6yO3oFNe7>

1. **PROBLEM STATEMENT: Can you prompt LLMs (ChatGPT, LLAMA) to solve the phrase and sentence similarity scores?**

**METHOD:**

1. Import libraries: The code imports necessary libraries like `pathlib`, `textwrap`, `google.generativeai`, `IPython.display`, `pandas`, and `os`. It also installs the `google-generativeai` library if it's not already installed.

2. Set up API key and GenerativeAI model: The code either uses the provided API key or fetches it from an environment variable. It then configures the GenerativeAI model with the API key. The code lists all available models and selects the `gemini-pro` model for content generation.

3. Read data from the CSV file: The code reads data from a CSV file named "validation.csv" and stores it in a pandas DataFrame.

4. One-shot learning examples: The code uses the model to compare the similarity of two phrases and provides a similarity score between 0 and 1.

Two examples are provided:

- "the quick brown fox" and "a fast red fox"

- "the big bad wolf" and "the small good wolf"

5.Generate content for an empty prompt: The code calls the model's `generate\_content` method with an empty prompt and displays the generated text.

**INFERENCE:**

The Gemini-Pro Model was used to calculate the sentence similarity scores. The ‘model.generate’ function was used to try zero-shot and one-shot learning. One-shot learning resulted in a better similarity score. This model would have provided better results if it were provided with a dataset which gave more insights on how the similarity scores were expected to be.

Datasets used: validation.csv

Link: <https://github.com/anjana-psvel/Precog_Task/blob/main/geminiiiiii.ipynb>

1. **PROBLEM STATEMENT : FINE TUNING GPT 3.5 FOR IDENTIFYING SENTENCE SIMILARITY SCORES**

**METHOD:**

1. Import libraries, Load and Preprocess Data: The code loads the training data from a CSV file (`train\_data.csv`) and selects the first 2000 rows.

- It defines a function `convert\_to\_gpt35\_format` to convert the data into the format expected by the OpenAI API.

- The user message contains the two sentences being compared, and the assistant message contains the label.

2. Split Data: The code uses `train\_test\_split` from scikit-learn to split the converted data into training and validation sets.

- Stratified splitting is used to ensure that the distribution of labels is similar in both sets.

3. Write Data to JSONL Files: The code defines a function `write\_to\_jsonl` to write the training and validation data to JSONL files.

- JSONL (JSON Lines) format is a simple text format where each line is a valid JSON object.

- This format is required by the OpenAI API for fine-tuning.

4. Fine-Tuning with OpenAI: The code initializes an OpenAI client using the API key.

- It creates two files on OpenAI using the `files.create` method, one for training and one for validation.

- A fine-tuning job is then created using the `fine\_tuning.jobs.create` method, specifying the training and validation files, the model to use (`gpt-3.5-turbo`), and a suffix to identify the job.

5. The code retrieves the fine-tuned model ID from the fine-tuning job.

- It defines two functions: `format\_test` to format a test example and `predict` to use the fine-tuned model to predict the label for a given test example.

- The `store\_predictions` function iterates through the test data, formats each example, makes a prediction using the fine-tuned model, and stores the prediction in the `Prediction` column of the test DataFrame.

6. The code calculates the accuracy score by comparing the predicted labels with the actual labels in the test DataFrame.

**INFERENCE:**

The first 2000 records on the ‘train\_data.csv’ dataset was used to fine-tune GPT 3.5. This dataset contains a pair of sentences and a label - 0 or 1. 0 indicates dissimilarity and 1 indicates similarity. First, the csv was converted into the jsonl format. A test dataset taken from the remaining ‘train\_data.csv’ dataset and resulted with an accuracy of 0.91. As the test set was taken from the same dataset, the accuracy was very large, indicating an overfitting model.

Link: <https://github.com/anjana-psvel/Precog_Task/blob/main/fine_tuning_gpt3_5_fin.ipynb>

**COMPARING ALL THE THREE METHODS**: Static word embeddings, fine-tuned transformers, LLMs, I would say that pre-trained models like BERT would’ve given better accuracy when fine tuned properly with our dataset. Next comes static word embeddings and then LLMs. Incorporating a Retrieval-Augmented Generation (RAG) framework into large language models (LLMs) can significantly enhance their capabilities, particularly in tasks that require generating text based on specific contexts or prompts. Fine-tune the model using task-specific data and objectives to further improve its performance on specific applications.