Fine-Tuning BERT for Attribution of Large Language Model Outputs Using Sequence Classification

Krishnasai Paleti
kxp5619@psu.edu

Mega Sri Shyam
mpb6512@psu.edu

Raghavendra Jagirdar
rvj5301@psu.edu

October 2024

1 Introduction

The rapid development of large language models (LLMs) has revolutionized natural language processing (NLP) across various domains. With models like GPT-4o, Llama, and Gemma achieving unprecedented levels of fluency and coherence, there is growing interest in understanding how these models differ in their outputs. Identifying the unique characteristics of LLM completions is crucial for applications that rely on model interpretability, model comparison, and optimization of LLM-driven tasks.

This project aims to build a classifier capable of identifying which LLM generated a given text completion based on a common truncated prompt. By analyzing the outputs of four different LLMs — GPT-4o, llama 3.1, Gemma 2, and Mixtral this project provides insights into how different models approach the same linguistic input and how these differences can be leveraged for classification tasks.

The central challenge of this project is to determine which LLM generated a given text completion (xj) based on a provided prompt (xi). While these models are trained on similar objectives, they exhibit unique behavior when producing text. By constructing a classifier that can accurately distinguish between the outputs of different LLMs, we aim to uncover the distinguishing features of each model's language generation process.

To address this problem, we curated a dataset consisting of 1,000 common prompts (xi) and 200 unique prompts per LLM, which were completed by four different LLMs. These xi-xj pairs were used as input features for training a classifier. We fine-tuned a BERT model as the classifier, which was trained to distinguish between the outputs of the different LLMs. The model was evaluated on key metrics such as accuracy, precision, recall, and F1-score to measure its effectiveness in identifying the correct LLM based on the generated text.

This project contributes to the growing field of model interpretability in NLP by providing a practical approach to differentiate between outputs from various LLMs. By understanding the unique behaviors and patterns that each LLM exhibits, researchers and practitioners can make more informed decisions about which model is best suited for specific tasks. Additionally, this work has implications for model benchmarking, improving the accuracy of LLM-driven applications, and enhancing trust in AI-generated content.

2 Dataset Curation

The dataset for this project was curated through a multi-step process that involved extracting xi prompts from first-person perspective books and articles, using tools like OpenAI Playground's Retrieval-Augmented Generation (RAG) capabilities. The dataset consists of both common and unique prompts, which were then processed through various language models to generate text completions.

2.1 Data Collection Process

The initial step in curating the dataset involved gathering xi prompts, which are truncated sentences extracted from first-person perspective books and articles (Harry Potter, Percy Jackson, blog posts, etc.). To automate and streamline this extraction process, we utilized the Retrieval-Augmented Generation (RAG) capabilities of OpenAI Playground. This allowed us to upload pages of text and efficiently extract relevant truncated sentences. Using this method, we collected an initial set of 1,021 common xi prompts, which would be provided to all four LLMs for generating their respective completions (xj).

In addition to the common prompts, we created 800 unique xi prompts split equally among the LLMs (200 per LLM). These unique xi prompts act as control inputs, enabling the classifier to learn generalizable patterns across different LLMs.

2.2 Custom Scripts and Data Formatting

To organize and prepare the dataset, we wrote custom Python scripts to handle various preprocessing tasks:

- **Duplicate Checking**: we created and used .txt files named xi.txt for the common prompts and controlxi_1.txt, controlxi_2.txt, controlxi_3.txt, and controlxi_4.txt for the unique prompts. A custom Python script was implemented to check for duplicates across these files, ensuring there were no duplicates in the combined set of 1,821 xi prompts (1,021 common and 800 unique).

- **CSV Conversion**: After validating the uniqueness of the xi prompts, we ran another custom script to convert the .txt files into .csv format for further processing. Each *.csv* file followed the format: xi, xj, and LLM.

2.3 Generation of Completions (xj) Using Multiple LLMs

Once the xi prompts were prepared, they were fed into four different LLMs to generate corresponding xj completions. The LLMs used for this task were:

- **GPT4o-mini**: Leveraged through the OpenAI API using the *chat completions* method.
- **Llama3.1, Gemma2, and Mixtral**: These models were accessed via the Groq API service.

For each xi, the corresponding xj completions were generated using the prompt *"You are a helpful assistant, please complete the following sentence. Only give me the completed part. Do not give me input sentence as well"* and stored in separate *.csv* files. Each LLM csv contains 1,221 samples, with 1,021 samples corresponding to the common xi prompts and 200 unique xi prompts per LLM.

2.4 Data Splitting and Class Balance

After generating the xi-xj pairs, the next critical step was splitting the dataset into training, evaluation, and test sets in a class-balanced manner. To avoid feeding the model any bias, we ensured the data was split in the following ratio:

- 80% for training
- 10% for evaluation/validation
- 10% for testing

We created a Jupyter notebook to automate this class-balanced split. The split was performed in two stages:

- **Common xi Split**: The first stage involved splitting the 1,021 common xi samples from all LLMs (across all .csv files) into training, evaluation, and test sets.
- **Unique xi Split**: The second stage involved applying the same split to the 200 unique xi samples per LLM.

This two-stage split ensured that the training, evaluation, and test sets had no class imbalance, which is crucial for fine-grained model tuning. After the split, the datasets were shuffled to avoid any inherent bias in batch selection during model training

2.5 Final Dataset Structure

The final curated dataset consists of:

- 4,084 xi-xj pairs generated from the common xi prompts, distributed equally across the four LLMs.
- 800 xi-xj pairs generated from the unique xi prompts, distributed as 200 per LLM.

This totals 4,884 data points, each with the following structure:

- **xi**: The truncated prompt.
- **xj**: The LLM-generated completion.
- **LLM Label**: The LLM responsible for generating the completion (GPT4o-mini, Llama3.1, Gemma2, Mixtral).

After the dataset was curated and split, it was **shuffled** to ensure that no bias influenced the training process. Extreme care was taken in this curation process to ensure the data was free from bias, well-balanced, and represented a rich dataset for training a classifier capable of distinguishing LLM outputs.

3 Classifier and Training

We have chosen bert-base-uncased from Hugging Face transformers library as our deep learning classifier.

3.1 Classifier: BERT Base Uncased for Sequence Classification

To classify the generated xi-xj pairs and determine which LLM produced the text completion, we utilized the pre-trained BERT base uncased model from Hugging Face. Specifically, we used the BERT for Sequence Classification model, which is well-suited for tasks where the goal is to assign a single label to a given input.

The architecture of BERT base uncased provides robust embeddings and attention mechanisms that enable it to capture complex relationships between the xi prompts and xj completions. By fine-tuning the model on this specific task, we aimed to leverage BERT's ability to differentiate subtle linguistic nuances across different LLM outputs.

3.2 Data Preprocessing

Before training the classifier, we created a custom preprocessing function. This function was responsible for:

- **Tokenizing the xi-xj pairs**: For each record, the truncated input xi and its corresponding xj completion were tokenized using Hugging Face's BertTokenizer.

- **Label Mapping**: The LLMs were mapped to numerical labels, with GPT4o-mini = 0, Gemma2 = 1, and Mixtral = 2, Llama3.1 = 3. This label mapping enabled the classifier to distinguish between the outputs of each LLM.

The preprocessing function was applied to all three datasets: training, evaluation, and test. This ensured that all inputs were standardized and ready for BERT's input format, including truncation and padding.

3.3 Training Setup and Experimentation

Given the limited weekly GPU time available on Google Colab, we decided to integrate two tools into my experiment setup:

- **Hugging Face Hub**: This allowed us to store and share the best models from each run, providing easy access and the ability to deploy these models as APIs later.
- **Weights and Biases (WandB)**: WandB enabled live tracking of training metrics, such as loss and accuracy, for each run. It also provided a dashboard for monitoring different hyperparameter settings and fine-tuning experiments in real-time.

3.4 Hyperparameter Tuning

The primary hyperparameters we focused on were the batch size and learning rate. Since GPU time was limited, we adopted a pruning approach to optimize the training process. The training was broken down into two stages:

- **Batch Size Tuning**: we fixed the learning rate at the default 5e-5 (recommended for BERT models) and ran experiments with two different batch sizes: 16 and 32.
- **Learning Rate Tuning**: After determining the optimal batch size from the initial experiments, we then fixed the batch size and ran experiments to tune the learning rate, testing values of 1e-5, 2e-5, and 5e-5.

This approach significantly reduced the number of runs while still allowing for efficient hyperparameter optimization. In total, we performed 5 runs, each running for 20 epochs. We use the default *adamw* optimizer for all the runs.

Training Arguments:

- num_train_epochs=20,
- per_device_train_batch_size=32,
- per_device_eval_batch_size=64,
- warmup_steps=500,

- weight_decay=0.01,
- eval_strategy="epoch",
- save_strategy="epoch",
- load_best_model_at_end=True,
- metric_for_best_model="accuracy"
- greater_is_better=True

3.5 Model Evaluation

During training, we used accuracy as the primary metric to select the best model for each run. The model that achieved the highest accuracy on the evaluation set was then re-run with additional metrics, such as:

- Precision
- Recall
- F1-score

This final evaluation run ensured that the selected model not only performed well in terms of accuracy but also generalized well across other important metrics.

By incorporating WandB and Hugging Face Hub, we were able to track each of these runs, push the best models to the hub for later use, and generate comprehensive reports for each experiment. This setup allowed us to optimize the model without exceeding Colab's GPU time limits.

3.6 Total Runs

In total, we conducted 5 runs:

- 4 runs for hyperparameter tuning (batch size and learning rate).
- 1 final run for the optimal hyperparameters, where we evaluated additional metrics (precision, recall, F1-score) to provide a deeper understanding of the model's performance.

4 Results and Presentation

4.1 Overview of Results

After running a total of 6 experiments to optimize the model's hyperparameters, we evaluated the model using multiple performance metrics. The metrics focused on the model's ability to accurately classify the LLM that generated the xj completion from the xi prompt. The best batch size and learning rate combination is **32 and 5e-5**.The key metrics used were:

- Accuracy: Measures the percentage of correct predictions made by the model.
- Precision: Indicates how many of the model's positive predictions were correct.
- Recall: Indicates how well the model identifies all positive cases.
- F1-Score: The harmonic mean of precision and recall, providing a balanced evaluation of the model's performance.

4.2 Model Performance

The following results summarize the best model's performance on the unseen test set after tuning the hyperparameters:

- Test Set Accuracy: 70.71%
- Test Set Precision: 72.65%
- Test Set Recall: 70.71%
- Test Set F1-Score: 70.46%

These results indicate that the fine-tuned BERT model was able to correctly classify the LLM responsible for generating the text completion in approximately 70.71% of cases. While the accuracy provides an overall measure of performance, the precision, recall, and F1-score offer deeper insights into the model's ability to handle false positives and false negatives. The slightly higher precision (72.65%) suggests that the model was more selective in making positive predictions, while the recall (70.71%) indicates that it performed well in capturing the correct positive cases.

4.3 Key Observations

- **Balanced Performance**: The F1-score of 70.46% demonstrates a good balance between precision and recall. This is important for classification tasks where both false positives and false negatives are of concern.
- **No Class Imbalance**: The class-balanced train, eval, and test splits ensured that the model did not favor any specific LLM. This was reflected in the overall consistency across the metrics, with no significant discrepancies between precision and recall.
- **Impact of Hyperparameter Tuning**: The process of tuning batch size and learning rate allowed for an efficient optimization of the model's performance. The best results were achieved with a batch size of 32 and a learning rate of 5e-5.

4.4 Visual Representation of Results

The visual representation (Figure 1) of the results was captured through Weights and Biases (WandB), which provided detailed graphs tracking the training and evaluation process across all runs. The following metrics were visualized:

- **Evaluation Accuracy**: The graph showed the progression of evaluation accuracy across all epochs, providing insights into how well the models generalized across different training configurations.
- **Evaluation Steps Per Second**: This graph tracked the efficiency of each run, helping to measure the computational cost and runtime performance of the model during evaluation.
- **Evaluation Runtime**: This graph illustrated the time taken to evaluate the model across all runs, giving a clear understanding of the total time spent during the evaluation phase.
- **Training Loss**: A graph showing the training loss for each run was critical in understanding how quickly the model converged and whether any overfitting occurred during training.

The use of WandB allowed for real-time tracking of these metrics across all 5 runs, ensuring that hyperparameter tuning and model selection were based on comprehensive data. By monitoring these visual representations, we were able to make informed decisions on the optimal batch size and learning rate, leading to the selection of the best-performing model.

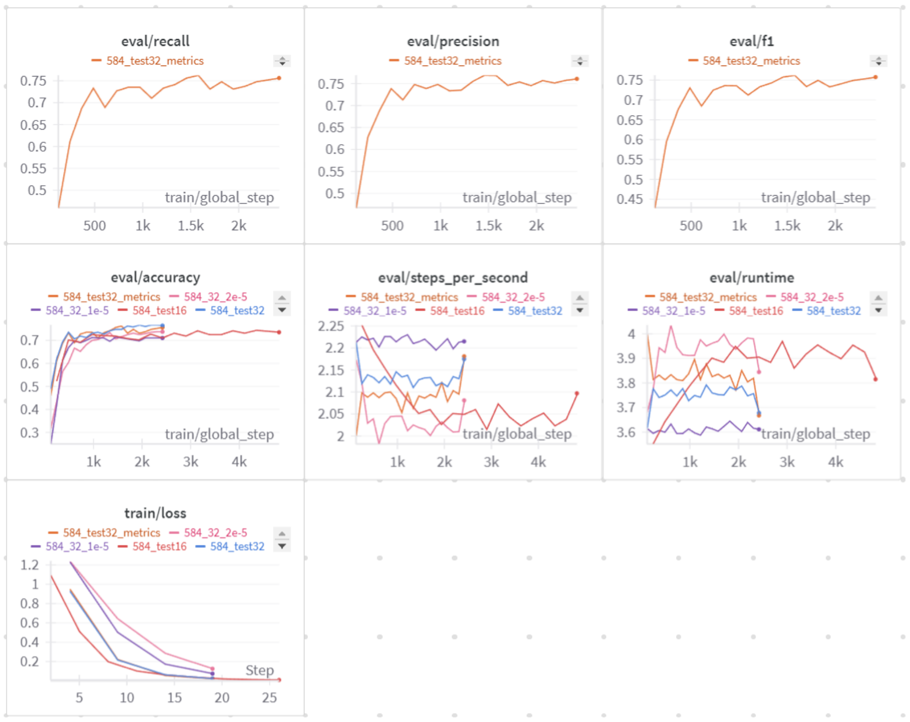

Figure 1: Eval Results

- Best Model: **584test32**
- Best Model Metric run: **584test32metrics**

4.5 Best Model Selection

The model with the best performance was selected based on accuracy as the primary metric during training. However, once the best model was identified, it was reevaluated with precision, recall, and F1-score to ensure that it performed well across all metrics. The best batch size is 32, learning rate is 5e-5 (other hyperparameters remain the same). The best **evaluation accuracy** is **76.56%**.

5 In-Depth Analyses of Experiments

5.1 Batch Size and Memory Constraints

One of the primary factors that influenced the choice of batch size was the GPU memory limitations in Google Colab, which provides 15GB of GPU memory. During experimentation, we incrementally increased the batch size and observed the corresponding GPU memory usage.

- At a batch size of 16, the model comfortably fits within the memory limit. Also the model eval accuracy starts off low and takes longer to reach the peak as shown in **Figure 1.**
- However, when experimenting with a batch size of 32, we observed that the memory usage reached 13.4GB, which was still manageable within the 15GB constraint.

Given this proximity to the memory limit and the need to avoid memory overflow during longer training runs, we chose to stop at a batch size of 32. Further increasing the batch size could have led to memory issues or forced us to reduce other settings like sequence length, which would affect the performance of BERT.

By carefully managing the batch size, we were able to optimize the use of available GPU resources while ensuring that the model could train effectively across all experiments.

5.2 Learning Rate Selection

A significant part of the analysis involved tuning the learning rate. We tested multiple learning rates, including 1e-5, 2e-5, and 5e-5, to understand which setting would provide the best performance for fine-tuning the model. Through these experiments, it became clear that 5e-5 was the optimal learning rate for this task **(Figure 1)**.

The results showed that at 5e-5, the model was able to balance between faster convergence and generalization. Lower learning rates such as 1e-5 and 2e-5 resulted in slower convergence and suboptimal performance, particularly in terms of accuracy and F1-score. This made 5e-5 the best-performing learning rate for my specific classification task.

5.3 Interactive Visualization of Pre-Trained and Fine-Tuned Models

In addition to the experimental results, We conducted a thorough analysis of the attention mechanisms in both the pre-trained and fine-tuned models. Using **BERTviz**, we visualized how attention heads in the BERT model functioned before and after fine-tuning. **(Figure 2, Figure 3, Figure 4, Figure 5)**

- **Pre-Fine-Tuning**: The pre-trained BERT model showed attention patterns that were more generalized, with attention heads often focusing on a broader set of tokens. This is expected behavior in a pre-trained model since it is designed to handle a wide variety of generic NLP tasks without domain-specific optimization.
- **Post-Fine-Tuning**: After fine-tuning the BERT model for the LLM classification task, the attention heads became more focused on specific tokens within the xi-xj pairs that were critical for distinguishing between LLMs.

This fine-tuning allowed the model to better capture nuances in the completions generated by different LLMs. By visualizing these changes, we were able to see how the model's attention became more specialized, helping to improve classification performance.

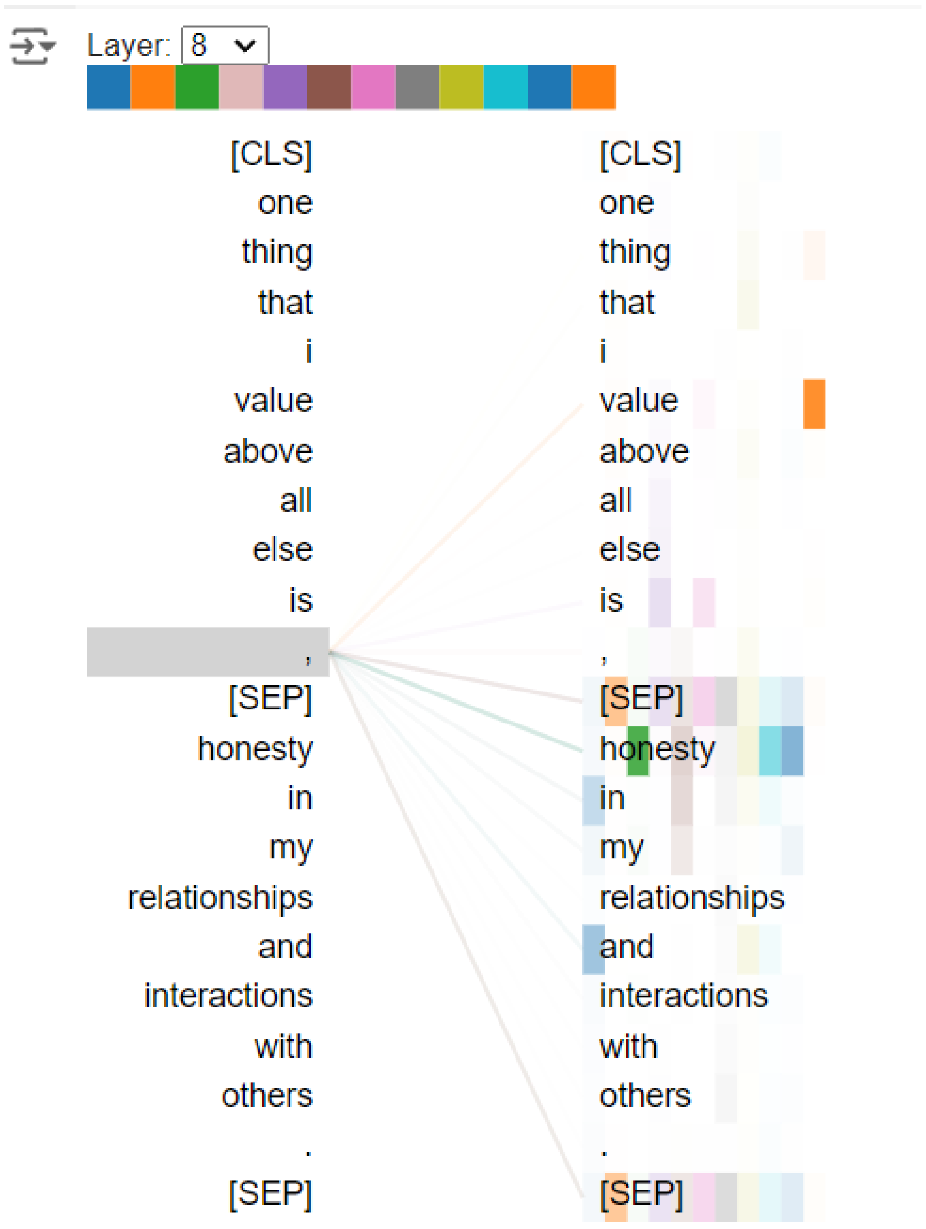

Figure 2: Head view for a test input

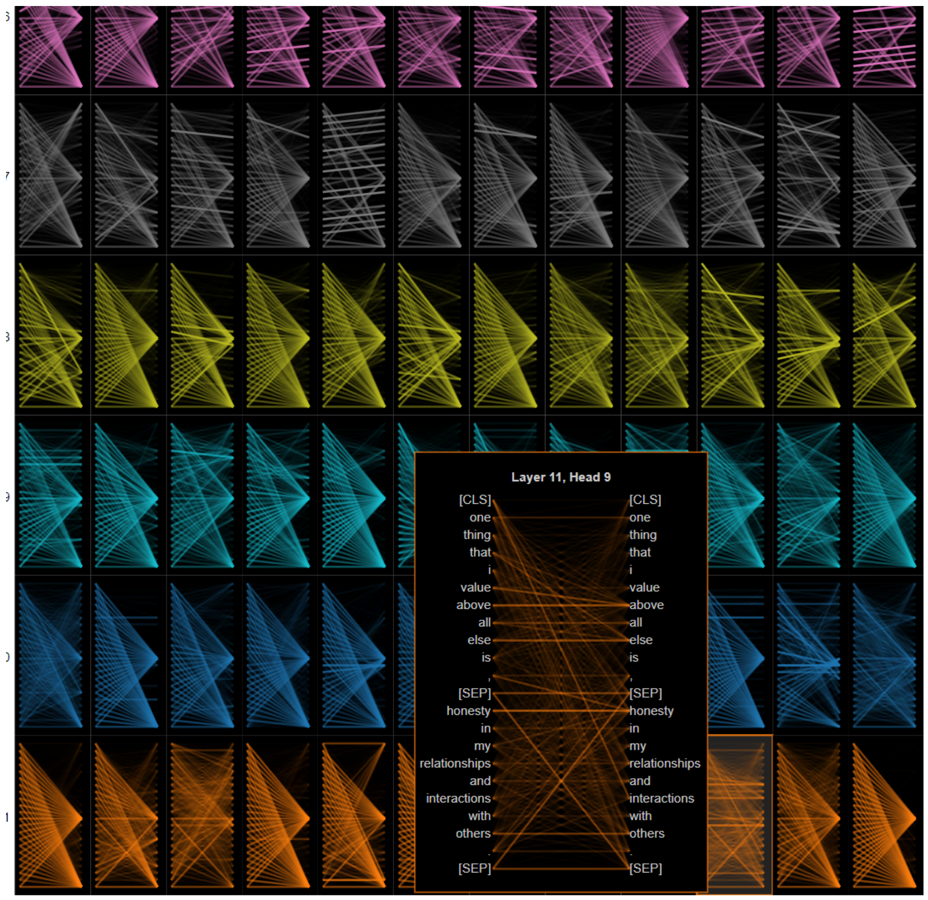

Figure 3: Model view (Truncated)-For full view, check BertViz.ipynb file

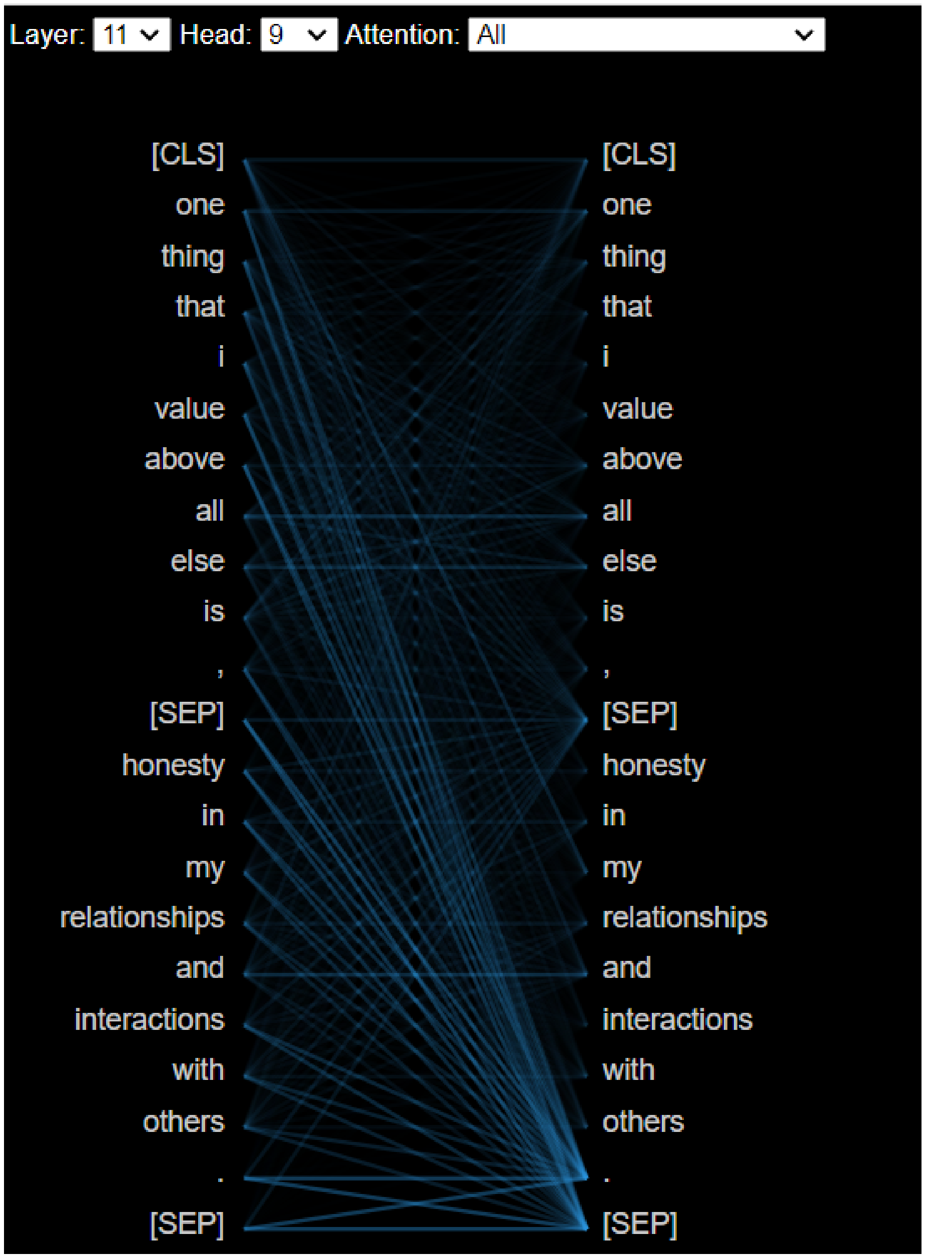
Layer: 11 Head: 9 Attention: All

Figure 4: Pre-Trained BERT model view

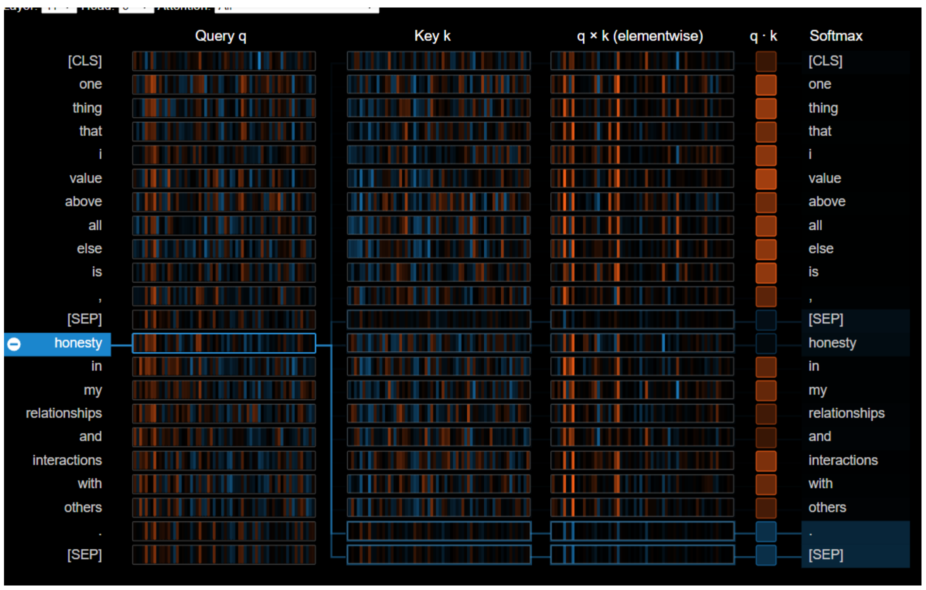

Figure 5: Pre-Trained BERT neuron view

5.4 Insights from Attention Mechanisms

The BERTviz visualizations provided valuable insights into the inner workings of the model:

- Certain attention heads began focusing more on the xj completions, indicating that the fine-tuned model had learned to recognize key differences between LLM outputs.
- The attention heads that initially spread their focus across both xi and xj in the pre-trained model showed more refined attention post-fine-tuning, particularly when handling longer and more complex completions.

These visualizations confirmed that the fine-tuning process had successfully specialized the model for the task at hand, allowing it to distinguish between subtle stylistic and structural differences in the text completions generated by the various LLMs.

5.5 Conclusion of Analyses

Through this series of analyses, we were able to draw meaningful conclusions about the batch size, learning rate, and attention mechanisms within the BERT model:

- Batch Size: Capped at 32 due to GPU memory constraints.

- Learning Rate: 5e-5 provided the best balance between convergence and generalization.
- Attention Mechanisms: Fine-tuning refined the model's focus on important tokens, improving its classification capabilities.

This in-depth analysis complements the experimental results by providing a deeper understanding of the model's behavior and the rationale behind key experimental decisions.

6 Discussion of Related Work

Research in the area of identifying and attributing machine-generated text to specific large language models (LLMs) has gained considerable attention. As the capabilities of LLMs continue to grow, distinguishing between different models' outputs is crucial for applications ranging from content authentication to model optimization. This section discusses three key papers in the field and compares their methodologies and objectives with the approach used in this project.

6.1 LLMDet

6.1.1 Problem Definition

The paper titled LLMDet focuses on the identification and attribution of machine-generated text to specific large language models (LLMs). This task is particularly important in scenarios where distinguishing between outputs from different models is essential for maintaining authenticity, security, and trust in AI-generated content. The challenge lies in creating an attribution method that is both efficient and secure, especially when model access is restricted or when commercial LLMs are involved.

6.1.2 Methodology

LLMDet introduces a proxy perplexity-based approach to tackle the problem of model attribution. The method does not require direct access to the LLMs' architectures or parameters. Instead, it relies on estimating the likelihood of a given text being generated by various LLMs through proxy models. The key steps in their approach include:

- **Proxy Perplexity Computation**: LLMDet uses surrogate models trained to approximate the behavior of the target LLMs. By computing the perplexity scores of a given text across these surrogate models, it estimates the likelihood of that text originating from each target model.
- **Model Agnostic**: The method does not require white-box access to the LLMs themselves. It can work with commercial models and proprietary LLMs where full transparency of the model's structure and weights is not possible.

- **Security and Efficiency**: LLMDet prioritizes the security and efficiency of the detection process, making it suitable for commercial use where access to the inner workings of models might be restricted. It also minimizes computational resources compared to directly using large models for evaluation.

6.1.3 Comparison to Our Project

While LLMDet shares a similar goal with our project—classifying LLM-generated text—it takes a notably different approach, focusing on proxy models and perplexity estimation rather than direct fine-tuning of a transformer model like BERT. Here's how our approach diverges:

- **Direct Classification Using BERT**: Unlike LLMDet, which uses proxy models to infer LLM attribution indirectly, our approach uses BERT base uncased, a transformer model fine-tuned specifically for the sequence classification task. We train BERT to distinguish between different LLMs using xi-xj pairs directly. This method allows BERT to learn the stylistic and structural differences across outputs of different LLMs, providing a more direct and potentially fine-grained attribution.

- **Tokenization and Mapping**: In our approach, a custom preprocessing function tokenizes the xi-xj pairs and maps each LLM to a numerical value. This approach directly integrates the task of LLM classification into the model's learning process, enabling it to make predictions based on text completions generated from specific input prompts.

- **Training Data and Control Inputs**: Unlike LLMDet, which relies on proxy perplexity, our project involves training BERT on a curated dataset where each xi prompt is either common across all LLMs or uniquely assigned to a specific LLM. This distinction allows the model to not only learn the general characteristics of the LLMs but also to differentiate LLM-specific behaviors through control samples. The dataset, balanced across the LLMs, helps the classifier generalize well, making it less reliant on proxy behavior and more capable of identifying intrinsic characteristics of each model.

6.1.4 Additional Insights Gained from LLMDet

While our project uses a different technique, the insights from LLMDet provide valuable takeaways:

- **Perplexity-Based Evaluation as a Supplement**: LLMDet's method suggests that combining our fine-tuned BERT model approach with perplexity-based evaluation might provide a hybrid solution, enhancing classification accuracy and robustness.

6.1.5 Remarks

While LLMDet is closely aligned with our project in terms of task—LLM attribution—the methodology, implementation, and constraints differ significantly. LLMDet's reliance on proxy models and perplexity provides an efficient and secure method suitable for commercial environments with limited access to LLM internals. In contrast, our project adopts a direct sequence classification approach, leveraging BERT's fine-tuning capabilities to classify LLM outputs based on text patterns learned from curated input-output pairs.

6.2 LLM-DetectAIve

6.2.1 Problem Definition

LLM-DetectAIve tackles the problem of identifying not just binary machine-generated versus human-written texts but also offers a more granular classification. It introduces categories like human-written but machine-polished and machine-generated but humanized texts, aiming to capture subtle interventions by LLMs.

6.2.2 Methodology

It employs a multi-way classification approach, leveraging fine-tuned models such as RoBERTa and DeBERTa to discern different levels of LLM involvement in text creation. This approach is particularly suited for educational and academic settings where differentiating the degree of machine intervention is crucial.

6.2.3 Comparison to Our Project

Our project differs as it focuses specifically on classifying completions generated by distinct LLMs (GPT4o, Llama3.1, Gemma2, Mixtral) using a BERT-based model. We don't aim to detect the degree of human or machine intervention but rather to classify the source model based on the output characteristics of xi-xj pairs.

6.3 Machine-Generated Text Localization

6.3.1 Problem Definition

This paper explores a more fine-grained approach by localizing machine-generated text within mixed human-machine documents. It addresses scenarios where only parts of a document are generated by an LLM, which standard binary classification might miss.

6.3.2 Methodology

The approach uses a lightweight localization adaptor called AdaLoc, which provides sentence-level predictions to identify machine-generated text within longer articles. By incorporating multiple sentences as context, it improves the accuracy of detecting short segments generated by LLMs.

6.3.3 Comparison to Our Project

Unlike our work, which classifies text outputs based on predefined input prompts and their completions (xi-xj pairs), this paper focuses on localizing individual sentences within mixed documents. Our project does not aim for localization or handling mixed human-machine content but instead targets overall classification of LLM outputs from predefined prompts.

7 Hugging Face Model Cards

- BatchSize: 32, LR: 5e-5 :- https://huggingface.co/pk248/584_test32
- BatchSize: 32, LR: 2e-5 :- https://huggingface.co/pk248/584_32_2
- BatchSize: 32, LR: 1e-5 :- https://huggingface.co/pk248/584_32_1
- BatchSize: 16, LR: 5e-5 :- https://huggingface.co/pk248/584_test16
- Metric Run (32, 5e-5) :- https://huggingface.co/pk248/584_test32_metrics

8 References

- LLMDet: A Third Party Large Language Models Generated Text Detection Tool
- LLM-DetectAIve: a Tool for Fine-Grained Machine-Generated Text Detection
- Machine-Generated Text Localization
- bertviz
- Weights and Biases(WandB) Documentation
- HuggingFace Documentation
- Groq Documentation
- OpenAI Documentation