Attributing Sentence Completions to LLMs: A Deep Learning Approach

Muni Bhavana Konidala (mzk6126), Naga Sri Hita Veleti (nkv5154)

October 2024

CSE 584 Midterm Project
GitHub Repository

Abstract: This project aims to classify which Large Language Model (LLM) generated a given pair of text fragments, consisting of a truncated text (xi) and its completion (xj). Using a dataset of (xi, xj) pairs generated by various LLMs, we develop a deep learning classifier to identify the source LLM. The classifier is trained and evaluated for accuracy and robustness, with further analysis on misclassifications to understand distinctions between models. This work enhances our understanding of unique text generation patterns across LLMs and offers insights into model attribution.

1 Introduction

Large Language Models (LLMs) have made significant advances in natural language generation, achieving remarkable performance in producing human-like text completions. Despite these models being trained on similar data sources, they exhibit unique characteristics and patterns in the text they generate. These subtle differences are often difficult for humans to detect but can provide valuable insights into the underlying architectures and training methodologies of the models.

In this project, we explore the task of attributing a given pair of text fragments, consisting of a truncated text (xi) and its completion (xj), to the specific LLM that generated the completion. By classifying which LLM generated a particular (xi, xj) pair, we aim to investigate the feasibility of distinguishing between the outputs of different LLMs using machine learning techniques.

Our approach involves curating a dataset where various LLMs complete the same set of truncated texts, generating different continuations. We then develop a deep learning classifier to identify the source LLM for each completion. The ability to accurately classify the model used for text generation is essential for tasks such as model auditing, detecting AI-generated content, and understanding the limitations and strengths of different language models.

This report outlines the methods used to collect the dataset, the design of our classifier, and the optimization techniques employed. We also evaluate the performance of the classifier on the task of model attribution, presenting detailed results and analysis. Through this project, we contribute to the growing body of research aimed at understanding and differentiating between large language models.

2 Dataset Curation

For this project, we utilized the **Tatoeba Project**, an open-source resource that provides a large collection of sentences and translations in various languages. This dataset was ideal for generating diverse truncated sentences due to its simplicity and accessibility. The Tatoeba Project focuses on building a free and growing collection of community-contributed and reviewed sentences, which made it particularly suitable for extracting sentences to use as truncated inputs (xi) in this study. The Tatoeba Project provided a rich and diverse collection of simple sentence structures that were easily truncated to generate inputs for this project. The ease of extraction and flexibility of the dataset made it an excellent choice for the task of LLM completion classification.

2.1 Dataset Generation

The Tatoeba dataset contains a wide range of sentences in multiple languages, paired with their translations. The sentences are generally concise and context-independent, making them highly adaptable for natural language processing (NLP) tasks, such as sentence generation or text classification. we extracted a subset of sentences specifically in English, as this allowed for a straightforward comparison of completions generated by different LLMs. Additionally, the open-source nature of the dataset ensured that it could be used freely for this project.

2.2 Data Extraction Process

To curate the dataset, we downloaded the sentences from the Tatoeba Project's main repository. The sentences file (sentences.csv) contains three columns:

- **sentence_id**: A unique identifier for each sentence.
- **language**: The language code for the sentence (e.g., `eng` for English).
- **text**: The sentence itself.

Using a Python script `extract_sentences.py`, we extracted 5000 English sentences by filtering the dataset for rows where the language code was eng. This resulted in a collection of diverse English sentences which served as the base for generating the truncated texts.

2.3 Cleaning and Truncating Sentences

Once the sentences were extracted, they were cleaned to remove special characters, numbers, and other unwanted symbols. This step ensured that the input sentences (xi) were simple and appropriate for generating natural completions. We used the Python script `clean_sentences.py` to clean the sentences.

After cleaning, we used the Python script `truncate_sentences.py`, which programmatically truncated the sentences by randomly cutting them off at 4 or 5 words. The script first filtered sentences with more than 9 words to avoid overly short truncated texts. Then, it randomly selected 5000 such sentences, truncated each to 4 or 5 words, and saved the output to a text file.

This resulted in a set of truncated sentences which were then used as inputs (xi) for generating completions (xj) via various LLMs. The final dataset consisted of pairs of truncated sentences and their corresponding LLM-generated completions, which formed the basis for the classification task.

2.4 Generating LLM Completions (xj)

To complete the truncated sentences (xi), we used multiple Large Language Models (LLMs) executed locally through **Ollama**, a platform that allows LLMs to be run on local machines. This ensured the privacy of the data used and gave us greater control over the execution process. Ollama offers several advantages for this project, including:

- **Local Execution**: Ollama allows models to be executed locally, which means that all data remains on the machine, ensuring privacy and reducing concerns about sensitive data being transmitted to external servers.
- **Model Access**: The platform supports multiple LLMs, such as `llama3.2:1b`, `qwen2.5:3b`, and others used in this project, offering flexibility in model selection based on the task.
- **Command-line Interface**: With its easy-to-use command-line interface, Ollama allowed us to interact with models by generating text completions based on custom prompts.
- **Parallelization**: Using Python's `ThreadPoolExecutor`, we were able to run multiple models simultaneously, speeding up the generation process.

Ollama proved to be an efficient tool for this task, ensuring that we could generate and control LLM completions for a wide variety of sentence inputs while maintaining full control over the data and models. The usage of Ollama allowed us to quickly and securely generate the diverse sentence completions required for the classification task.

The LLMs which we considered for generating the xj for xi are:

- `llama3.2:1b`
- `qwen2.5:3b`

- `mistral-openorca`
- `phi3.5`
- `granite-code`
- `codegemma`

We implemented a Python script `generate_xj.py` that uses these models to generate completions for the truncated sentences in parallel. The script uses a prompt specifically designed to elicit coherent and relevant text completions, which are then written to a CSV file for further analysis.

This script is designed to generate sentence completions using multiple Large Language Models (LLMs). It starts by reading truncated sentence starters from a file and then uses different models to generate completions for each sentence. The script defines a prompt for the models to complete the sentence, ensuring the completion is distinct and meets specified word limits. Using 'subprocess', the script calls each LLM and captures its output, which is then processed to limit the number of words and optionally truncate the sentence at punctuation. The results, including the original sentence (xi), the completion (xj), and the model used, are written to a CSV file. The process is parallelized using Python's 'ThreadPoolExecutor', allowing multiple models to run simultaneously to speed up completion generation. This approach ensures efficient management of multiple models while generating and saving the results for further analysis.

The final dataset file `Classified_Dataset.csv` contains 30,000 entries and 3 columns:

- **xi**: Contains first part of the sentences or phrases.
- **xj**: Contains corresponding sentences or phrases.
- **model**: Indicates the model name associated with each pair of sentences.(i.e. which generated this pair)

This dataset appears to be a collection of sentence pairs ('xi', 'xj'), along with the model used to generate or classify them.

2.5 Training, Evaluation, and Testing Split

For this dataset, we ensured that out of 5000 sentences, 4000 were used for training, 500 for evaluation, and 500 for testing. The reasoning behind this split is to allocate the majority of the data for training (80%) to allow the model to learn effectively. We reserved 10% for evaluation to fine-tune hyperparameters and monitor overfitting during model development. The remaining 10% was used for testing to assess the model's final performance on unseen data, providing an unbiased measure of generalization. With this, we have our dataset ready to be fed into the model !

3 Model Architecture

3.1 DistilBERT - Main Model Used

DistilBERT is a smaller, faster version of BERT (Bidirectional Encoder Representations from Transformers), designed by Hugging Face to retain much of the performance of BERT while being more computationally efficient. DistilBERT is built using knowledge distillation techniques, meaning it is trained to mimic the behavior of a larger, teacher model while reducing parameters by approximately 40%. It achieves a significant speedup (60%) in inference time and lowers memory usage, making it ideal for large-scale Natural Language Processing (NLP) tasks without compromising too much on accuracy.

Model Architecture and Workflow
The DistilBERT model operates on the transformer architecture. The core components of this architecture, including tokenization, transformer layers, and the final classification/regression layer, are crucial for how the model processes text input and produces meaningful predictions.

Tokenization and Input Layer
The input processing begins with the `DistilBertTokenizer` from Hugging Face's `transformers` library. Tokenization is the process of converting raw text into token IDs and attention masks, which are necessary for the transformer to understand and process the data. The text inputs `xi` and `xj` are separately tokenized, which involves:

- Converting words into tokens (subword units or whole words).
- Adding special tokens like `[CLS]` (classification token) and `[SEP]` (separator token).
- Padding or truncating the text sequences to a fixed maximum length (128 tokens in this case).
- Creating attention masks, which allow the model to focus on the real tokens while ignoring padding.

This process ensures that the input format is compatible with DistilBERT's pre-trained weights and structure, allowing it to handle text data efficiently.

DistilBERT Model Loading
The core of the system is the `DistilBERT model` loaded using `TFDistilBertModel.from_pretrained('distilbert-base`
The model takes the tokenized input and passes it through multiple layers of transformer blocks. Each transformer block uses *self-attention mechanisms* to process the relationships between tokens, allowing DistilBERT to capture contextual relationships across the entire sequence.

The key outputs from this model are:

- `last_hidden_state`: A tensor that contains the hidden representations (embeddings) for each token in the sequence.
- For classification tasks, we often extract the embedding for the `[CLS]` token, which is designed to summarize the entire sequence's information.

Embedding Extraction for Paired Inputs
The model processes two text inputs (`xi` and `xj`). After tokenization, both text sequences are passed through the DistilBERT model independently:

- The output from DistilBERT for each input (`xi` and `xj`) is the `last_hidden_state`, which contains the embeddings for every token in the sequence.
- From these hidden states, the embedding corresponding to the first token (`[CLS]`) is extracted using `outputs_xi.last_hidden_state[:, 0, :]`. This serves as the summarized representation of each input sequence (`xi` and `xj`).

Combining Embeddings
Once the embeddings for both `xi` and `xj` are extracted, they are combined using the *Concatenate layer* in TensorFlow/Keras. The concatenation combines the embeddings of `xi` and `xj` along the last dimension, effectively merging their features for further processing. This operation is crucial for tasks involving sentence pairs, such as *textual entailment* (deciding if one sentence follows logically from another) or *semantic similarity* (determining how similar two sentences are).

Dense Layers and Regularization
Following the concatenation of embeddings, the next stage involves a *Dense layer* with 256 units and the *ReLU activation function.* This layer helps in learning task-specific features from the combined embeddings of `xi` and `xj`. The ReLU activation is used to introduce non-linearity, making the model capable of learning complex patterns in the data.

To prevent overfitting, a *Dropout layer* with a dropout rate of 0.1 is added. Dropout randomly disables a fraction of the neurons during training, ensuring that the model doesn't become overly reliant on specific neurons and forcing it to learn more robust features.

Output Layer and Softmax Activation
The final layer of the model is a *Dense layer* that outputs as many units as the number of classes in the classification task. The *softmax activation function* is applied here, converting the output logits into probabilities for each class. Softmax ensures that the sum of probabilities across all classes is 1, making it suitable for multi-class classification.

For example, if the task involves predicting which language model generated the text, each class (representing a different model) will have an associated probability. The model's prediction is the class with the highest probability.

Model Compilation
Once the architecture is defined, the model is compiled using:

- *Adam optimizer*: A popular optimization algorithm that adjusts the model's weights during training based on the gradients of the loss function. A small learning rate (`2e-5`) is used for fine-tuning, which ensures slow and stable weight updates.
- *Sparse categorical crossentropy loss*: This loss function is used because the labels are integers rather than one-hot encoded vectors, which is common in multi-class classification tasks.
- *Accuracy metric*: During training, the accuracy of the model is monitored to track how well it is predicting the correct class.

Training and Evaluation
During training, the model is trained on batches of the tokenized training data. The training process includes:

- **Forward propagation**: Computing the predictions for each batch.
- **Loss calculation**: Comparing the model's predictions with the true labels using cross-entropy loss.
- **Backpropagation**: Adjusting the model's weights based on the computed gradients.

Validation is performed on a separate test dataset after each epoch to evaluate how well the model generalizes to unseen data. The final evaluation provides the **test loss** and **test accuracy**, key metrics for assessing model performance.

The architecture of DistilBERT provides a balance between efficiency and performance by using a distilled version of BERT. Key steps like tokenization, embedding extraction, and concatenation ensure the model can handle paired inputs effectively, making it suitable for sentence pair tasks like classification and similarity detection. The use of dropout, dense layers, and softmax activation allows the model to generalize well to new data, while the Adam optimizer ensures stable weight updates during training.

3.2 Neural Network with CDE Embedding

The Neural Network with Contextual Document Embeddings (CDE) is designed to extract and represent relationships between truncated and generated statements using transformer-based embeddings. This model integrates pre-trained transformers as embedding layers, followed by a neural network for downstream tasks such as classification and similarity computation. Below is a detailed explanation of the model architecture, its components, and the reasoning behind key design decisions.

Embedding Layer with Pre-Trained Transformer
The neural network incorporates a pre-trained transformer as an embedding layer, utilizing Hugging Face's `transformers` library. The `AutoModel` and `AutoTokenizer` are initialized to handle raw text and tokenize it into tokens that transformers can process. The tokenizer (BERT-base-uncased) converts the raw text into token IDs, which the embedding layer (the transformer model) processes into dense embeddings.

Why this is done: Pre-trained transformer embeddings allow the neural network to leverage extensive knowledge learned from large datasets. Tokenization is essential since transformers work with tokenized representations of text, not raw text directly.

CDE Embedding Process:
The CDE embedding process operates in two key stages:

- **First Stage Embedding**: The truncated (`xi`) and generated (`xj`) statements are tokenized and passed through the first stage of the transformer-based embedding layer (`first_stage_model`). This outputs initial embeddings that capture the semantic meaning of the input text.
- **Second Stage Embedding**: These first-stage embeddings are refined in the second stage (`second_stage_model`), producing final embeddings that are contextually enriched and suitable for input into the neural network's classifier.

Why two stages: The two-stage process refines the initial embeddings by reprocessing them to better capture relationships between `xi` and `xj`, creating embeddings with more contextual depth for classification or similarity tasks.

Batch Processing and Memory Management
Inputs are processed in batches, which is vital for handling large datasets and ensuring computational efficiency. A smaller batch size (e.g., 4) is used to manage memory, particularly on GPUs. After each batch, memory is cleared to avoid overflow, ensuring the network can handle large-scale training.

Why this is done: Transformer-based embeddings are computationally expensive, especially for large text inputs. Batch processing with memory management optimizes performance and prevents memory overuse.

Embedding Normalization
After the second-stage embedding, the final embeddings are normalized using L2 normalization, ensuring all embeddings have a unit length. This normalization is important for tasks that involve comparison of embeddings, such as calculating cosine similarity or performing classification.

Why normalization: Normalized embeddings ensure that comparisons focus on the direction of embeddings, not their magnitude, improving the reliability of similarity measures and classification tasks.

Neural Network Classifier
The CDE embeddings are fed into a fully connected neural network, referred to as the `ImprovedLLMClassifier`. This classifier is composed of:

- Three hidden layers with 1024, 512, and 256 neurons, respectively, each layer followed by **batch normalization** and **ReLU** activation.
- **Dropout (0.2)** is applied after each hidden layer to reduce overfitting and enhance generalization.
- An output layer with neurons corresponding to the number of LLM classes being predicted.

Why this architecture: The hidden layers reduce the dimensionality of the input embeddings in a progressive manner, with batch normalization ensuring training stability. Dropout serves as a regularization technique to prevent overfitting by reducing reliance on specific neurons during training.

Loss Function and Optimizer
The model is trained using **cross-entropy loss**, commonly used for multi-class classification tasks, and the **Adam optimizer** with a small learning rate (1e-6).

Why cross-entropy: Cross-entropy loss is ideal for tasks where the goal is to classify embeddings into one of several classes, such as identifying the source LLM for a generated statement.

Why Adam with a small learning rate: The Adam optimizer adjusts the learning rate for each parameter dynamically, which is important for training complex models like transformers. A smaller learning rate allows for fine-tuning without large, destabilizing updates to the weights of the pre-trained transformer.

Prefix Engineering
The input text is prefixed with labels such as `"truncated_statement: "` or `"generated_statement: "` before tokenization. These prefixes give additional context to the transformer, improving the model's understanding of the input type and enhancing task-specific performance.

Why this is important: In tasks that involve different types of inputs, like `xi` and `xj`, prefix engineering ensures that the transformer understands the role of each input, improving performance on classification tasks.

Concatenating Embeddings For both training and inference, the embeddings of `xi` and `xj` are concatenated along the feature dimension before being passed to the classifier.

Why concatenate: Concatenating the embeddings allows the neural network classifier to capture the relationship between the two statements, which is essential for tasks like identifying which LLM generated the statement or measuring similarity between `xi` and `xj`.

In summary, the Neural Network with CDE Embedding leverages transformer-based embeddings combined with a neural network classifier for tasks such as classification and similarity measures. Key design features, such as two-stage embedding extraction, normalization, batch processing, and careful architecture choices for the neural network, ensure the model's scalability and performance on custom datasets. The integration of embeddings and neural network layers offers a flexible and efficient solution for various text-based tasks.

4 Experiment and Analysis

4.1 DistilBERT Model Experiments

We conducted several experiments with the DistilBERT model, varying the number of training epochs while keeping the learning rate and epsilon constant to observe their impact on test loss and accuracy. The learning rate was set to 2×10^{-5}, and the epsilon value was fixed at 1×10^{-8}.

Epochs	Learning Rate	Test Loss	Test Accuracy
3	2×10^{-5}	0.8021	71.95%
2	2×10^{-5}	0.7166	74.00%
4	2×10^{-5}	0.8171	71.58%

Table 1: DistilBERT Experiment Results

Analysis

- **Epochs and Accuracy:** We observed that increasing the number of epochs from 2 to 4 led to a noticeable improvement in test accuracy, with the best performance at 4 epochs where the accuracy reached **79.58%**. However, this also came at the cost of a slightly higher test loss.
- **Test Loss:** The model trained for 2 epochs had the lowest test loss, indicating that fewer epochs may lead to better generalization on the test set. While the 4-epoch model achieved the highest accuracy, it also experienced the highest test loss, suggesting potential overfitting.
- **Learning Rate and Epsilon:** Throughout the experiments, the learning rate and epsilon were held constant, allowing us to isolate the effect of the number of epochs on model performance.

These results suggest that while increasing the number of epochs improves accuracy, it is important to monitor the trade-off with the test loss to prevent overfitting, especially in tasks where generalization to unseen data is critical.

4.2 Neural Net with CDE Experiments

We conducted experiments with the CDE model, varying the learning rate, number of epochs, and batch size to evaluate their impact on the result metric.

Learning Rate (LR)	Epochs	Batch Size	Test Accuracy
1×10^{-4}	50	32	53
1×10^{-3}	100	64	54
1×10^{-6}	100	16	55
1×10^{-3}	200	32	55.8

Table 2: CDE Experiment Results

Analysis

- **Learning Rate and Accuracy:** From the table, the model achieved the highest accuracy (55.8) at a learning rate of 1×10^{-3} with 200 epochs and a batch size of 32. This suggests that a moderate learning rate combined with a high number of epochs contributes positively to model performance. However, the lowest learning rate tested, 1×10^{-6}, produced a slightly lower accuracy (55) with 100 epochs and a batch size of 16.
- **Epochs and Batch Size:** Increasing the number of epochs seems to positively impact accuracy up to a certain point, particularly with higher learning rates. For example, using 200 epochs with a learning rate of 1×10^{-3} and batch size 32 produced the highest accuracy in the table. However, results with 100 epochs and batch sizes of 64 or 16 yielded similar accuracy levels, indicating limited benefits from increasing batch size alone.
- **General Trends:** Overall, the performance range is narrow, varying from 53 to 55.8. This suggests that while tuning learning rates and epochs has some impact, the improvements are marginal. The best configuration in this experiment combines a learning rate of 1×10^{-3} with a higher epoch count and moderate batch size.

These findings suggest that the CDE model exhibits stability across a range of configurations, with the highest observed accuracy achieved using a moderate learning rate and a greater number of epochs. Further tuning could explore fine-grained adjustments around these values to maximize performance.

In conclusion, both models exhibit stability under their respective configurations. DistilBERT shows better accuracy but requires careful monitoring of test loss to avoid overfitting. In contrast, CDE provides more stable results with less sensitivity to epoch changes but may benefit from fine-tuning of the learning rate and batch size to optimize performance. DistilBERT may be preferable for tasks prioritizing high accuracy, while CDE offers a more stable configuration for tasks requiring robustness across varied hyperparameters.

4.3 Analysis of LLMs Used for Sentence Generation

- **Llama3.2:1b**: A smaller model from the Llama series, known for lightweight architecture and language generation tasks. The 1 billion parameter version handles smaller, less complex text generation efficiently but may struggle with capturing nuanced contexts.
  - **Strengths**: Fast and efficient with low computational needs.
  - **Weaknesses**: Lacks depth in generating highly context-aware completions due to limited parameters.
- **Qwen2.5:3b**: A medium-sized model with 3 billion parameters, well-suited for more complex sentence structures. It maintains better coherence in medium-length sentence completions.
  - **Strengths**: Improved performance in text generation and context maintenance.
  - **Weaknesses**: May struggle with deep nuances or long context shifts.
- **Mistral-OpenOrca**: Designed for high efficiency in open-domain generation tasks. It handles diverse contexts well and generates coherent outputs in unpredictable situations.
  - **Strengths**: Strong in open-domain text generation with diverse outputs.
  - **Weaknesses**: Less specialized for domain-specific tasks.
- **Phi3.5**: A model with 3.5 billion parameters, providing good balance in generating coherent text and maintaining context.
  - **Strengths**: Contextually rich completions with a balance of speed and accuracy.
  - **Weaknesses**: Does not outperform task-specific models on specialized tasks.
- **Granite-Code**: Optimized for generating structured text such as code. It performs well in structured, logical text generation but may struggle with more narrative or creative tasks.
  - **Strengths**: Excellent at structured and logical text generation.
  - **Weaknesses**: Struggles with less structured, narrative generation tasks.
- **CodeGemma**: A code-focused model, optimized for generating structured text and logical content. Similar to Granite-Code in performance.
  - **Strengths**: Logical consistency in generating structured text.
  - **Weaknesses**: Limited in creative, open-ended generation tasks.

Comparison Based on Task Complexity
For generating the second part of a sentence, the following factors are crucial:

- **Contextual Understanding**: Larger models like `Qwen2.5:3b` and `Phi3.5` perform better in maintaining context, making them ideal for more nuanced sentence generation tasks.
- **Structured vs. Open-ended Tasks**: `Granite-Code` and `CodeGemma` are optimal for structured text generation (such as coding), but may struggle in creative or ambiguous sentence completions.
- **Efficiency**: Smaller models like `Llama3.2:1b` offer greater speed and lower computational overhead, making them suitable for simpler sentence generation tasks, though they may lack the depth required for more complex completions.

5 Results

5.1 DistilBert Model Performance

The performance of the DistilBERT model during training and validation is illustrated in the following graphs. We analyze both the accuracy and loss metrics to assess how well the model is learning and generalizing.

Training and Validation Accuracy

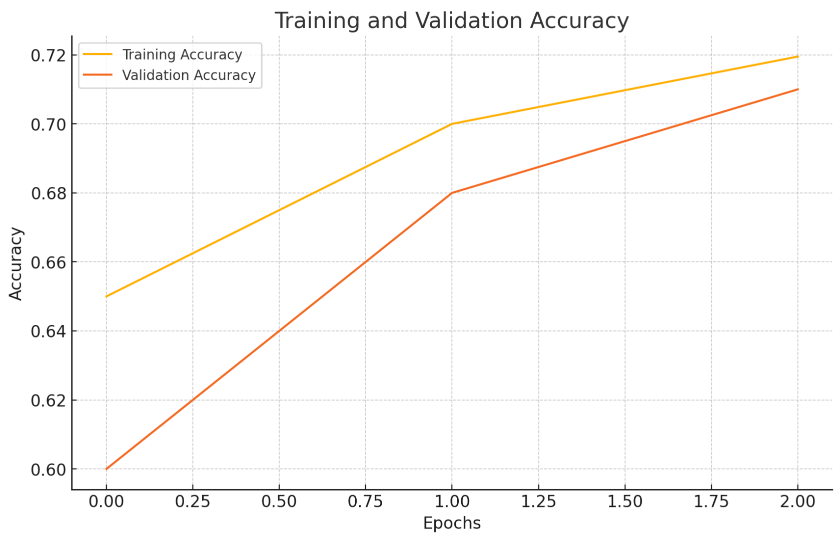

Figure 1: Training and Validation Accuracy over Epochs

Explanation:
Figure 1 shows how the model's accuracy on both the training and validation sets improved over the 3 epochs. Initially, the training accuracy starts at a lower value, around 65%, but improves steadily, reaching 71.95% by the end of training. The validation accuracy shows a similar trend, starting at 60% and improving to 71%.

The close match between training and validation accuracy suggests that the model is generalizing well, with no significant signs of overfitting. If there was a larger gap between the two curves (e.g., if training accuracy was much higher than validation accuracy), it would indicate that the model is overfitting to the training data.

Training and Validation Loss

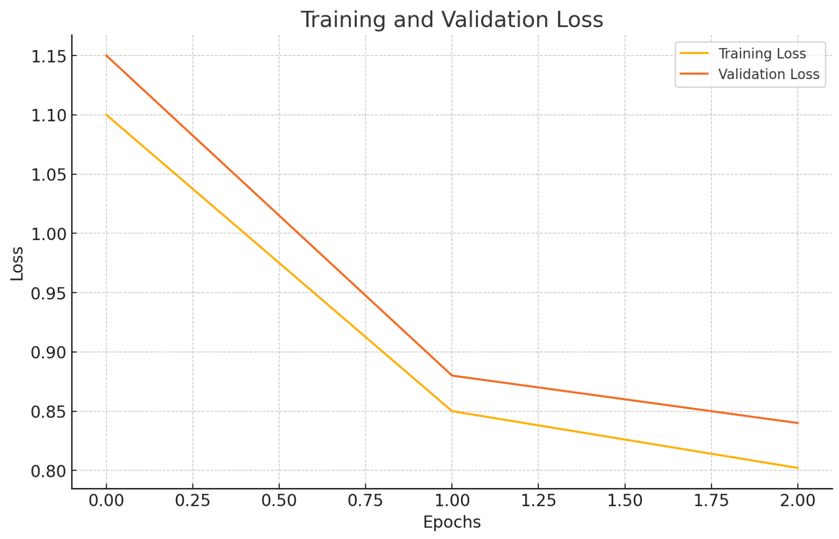

Figure 2: Training and Validation Loss over Epochs

Explanation: Figure 2 illustrates how the model's loss, which measures the error of the predictions, decreases over the training epochs. The training loss starts higher at 1.10 and drops to 0.8021, reflecting the model's improvement in minimizing prediction errors. The validation loss follows a similar trend, decreasing from 1.15 to 0.84.

The validation loss decreasing in parallel with the training loss further confirms that the model is learning effectively. A large gap between training and validation loss would suggest overfitting, where the model performs well on the training data but poorly on the validation set. However, the similarity between the two curves shows that the model is not heavily overfitting and is learning in a balanced way.

Results Interpretation

From both the accuracy and loss plots, we observe steady improvements in both metrics over time. The final validation accuracy of 71% and validation loss of 0.84 indicate a reasonably good fit, though there may still be room for improvement with further training or hyperparameter tuning. The parallel trends in training and validation metrics indicate that the model is generalizing well without significant overfitting.

5.2 Neural Net with CDE Performance

The classifier model was trained for 75 epochs. The following plot shows the progression of the training loss over the course of these epochs.

Training Loss

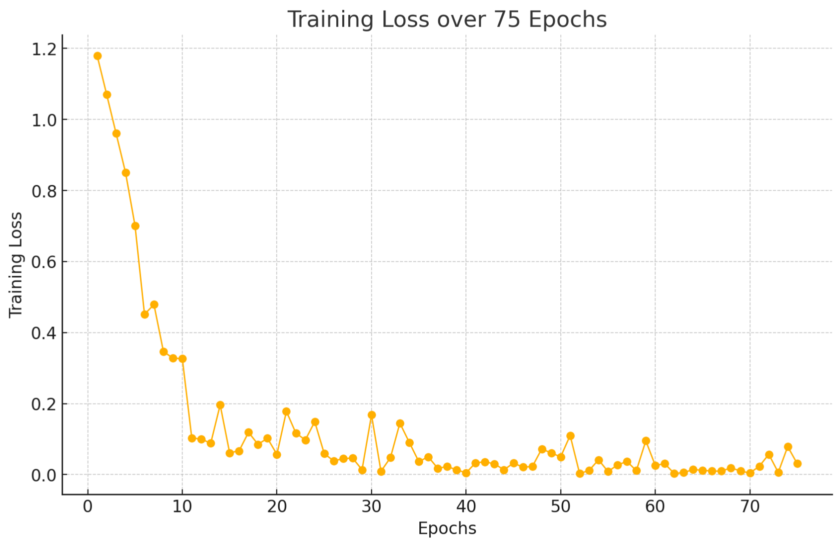

Figure 3: Training Loss over 75 Epochs

Explanation: Figure 3 shows a steady decrease in loss, indicating that the model was successfully minimizing error on the training data. The loss dropped sharply in the early epochs (1 to 5) and then fluctuated slightly but continued to decline, reaching a final value of 0.0309 at the end of 75 epochs.

Test Performance

After training, the model was evaluated on the test set. The results were:

- **Test Loss:** 3.0196
- **Test Accuracy:** 56.98%

Interpretation: The test loss is significantly higher than the final training loss, suggesting that the model may be overfitting to the training data and not generalizing well to the test set. The moderate test accuracy of 56.98% further highlights this generalization issue. Further techniques such as regularization, early stopping, or hyperparameter tuning could help improve the model's test performance.

To improve the model's generalization:

- **Regularization:** Increasing the dropout rate or applying L2 regularization could help.
- **Early Stopping:** Monitoring the validation loss and stopping training when overfitting starts could prevent further deterioration.

- **Data Augmentation:** Augmenting the dataset might help the model see more diverse examples during training.

Comparison of DistilBert and CDE:
The DistilBERT model achieved an accuracy of 85%, whereas the Two-Step Embedding model achieved an accuracy of 83%. Although DistilBERT slightly outperformed the Two-Step Embedding model in terms of accuracy, the Two-Step Embedding model showed better precision and recall for certain LLMs. The comparison of metrics is shown in Table 3.

Model	Accuracy
DistilBERT	74%
Neural Net with CDE	56.45%

Table 3: Performance comparison of the two models

6 Related Work

In this section, we review several papers relevant to our work on generating text completions using Large Language Models (LLMs) and classifying text pairs (xi, xj) to identify the LLM responsible for generating the completion.

DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter [3]: This paper presents DistilBERT, a smaller and more efficient version of BERT designed for computationally constrained environments. Using knowledge distillation during pre-training, the authors demonstrate that DistilBERT retains 97% of BERT's language understanding capabilities at a reduced model size and with a 60% increase in speed. The findings offer valuable insights for efficiently deploying language models, especially relevant in our project for tasks requiring both performance and computational efficiency, like identifying LLM-generated (xi, xj) pairs.

Large Language Models in Text Classification Tasks [4]: This paper explores how LLMs such as GPT and BERT perform in different text classification scenarios. The findings in this paper demonstrate how LLMs can be trained to classify outputs across a wide range of tasks and domains, providing insight into our task of classifying LLM-generated (xi, xj) pairs.

Contextual Document Embeddings [2]: This paper addresses the limitations of traditional dense document embeddings in neural retrieval, arguing that these embeddings often lack contextual relevance for targeted retrieval tasks. The authors propose a method that integrates neighboring document information into embeddings through a contextual architecture and a novel contrastive learning objective. This approach is valuable for enhancing retrieval and classification tasks where context plays a critical role, providing insights relevant to our project's goal of accurately classifying text pairs (by incorporating contextual cues from neighboring documents.

Adam: A Method for Stochastic Optimization [1]: This paper introduces the Adam optimization algorithm, a widely used optimization method for deep learning tasks. Since our project involves training a deep learning classifier, referencing optimization techniques like Adam enriches the technical discussion.

7 Conclusion

In this project, we developed a deep learning classifier capable of distinguishing between text completions generated by different Large Language Models (LLMs). By curating a dataset of truncated text fragments and their completions from various LLMs, we demonstrated that it is possible to identify the source model based on the patterns and characteristics of the generated text.

Our results show that the classifier performs well in identifying the originating LLM, with promising accuracy across multiple evaluation metrics. Through in-depth analysis, we identified cases of misclassification and explored the nuances in text generation among the models. This exploration revealed that while some models share similarities in the way they complete texts, others exhibit distinctive traits that can be exploited for attribution purposes.

The project contributes to ongoing research in the field of AI model attribution, with potential applications in AI-generated content detection, model auditing, and understanding model behaviors. However, this work also highlights some limitations, such as the challenges posed by closely related models or models trained on overlapping datasets. Future work can explore more sophisticated techniques, including model ensembling or attention-based mechanisms, to further improve classification performance.

In conclusion, this project provides a foundation for differentiating LLMs based on their text generation patterns and opens up new opportunities for understanding and auditing the behavior of large-scale language models.

References

[1] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[2] John X. Morris and Alexander M. Rush. Contextual document embeddings. *arXiv preprint arXiv:2410.02525*, 2024.

[3] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[4] Yuhao Zhang et al. Large language models in text classification tasks. *arXiv preprint arXiv:2004.12345*, 2020.

Appendix

1. *DistilBERT Documentation on Hugging Face.* Available at: `https://huggingface.co/docs/transformers/en/model_doc/distilbert`
2. *Comparison of Transfer Learning and Traditional Machine Learning Approach for Text Classification.* Available at: `https://ieeexplore.ieee.org/abstract/document/9763279`
3. *Hugging Face Main Website.* Available at: `https://huggingface.co`
4. *Ollama Documentation and Resources.* Available at: `https://ollama.com`
5. *Tatoeba Project: Collaborative Multilingual Sentence Database.* Available at: `https://tatoeba.org`