CSE 584: Machine Learning
MidTerm Project: Creating A Classifier To Identify Which LLM Generated A Given Text

Project Teammates:
Tejasree Parasa (tfp5349@psu.edu)
Sai Chandana Priya Avvaru (sfa5870@psu.edu)

ABSTRACT

This project aims to investigate the performance of various large language models (LLMs) in the task of sentence completion from truncated prompts. We will focus on generating completions from incomplete text inputs, represented as xi, such as "Yesterday, I went." Utilizing a diverse range of LLMs, we will collect an array of completions xj that exhibit varying lengths and structures. The generated outputs may include succinct phrases like "for a walk in the park" or more detailed sentences, such as "Yesterday, I went for a walk in the park to enjoy the nice weather."

To facilitate this investigation, we will curate a comprehensive dataset of input-output pairs that encapsulate the variations in responses produced by the different LLMs for the same initial prompt. This dataset will serve as a foundation for analyzing how each model approaches the sentence completion task, providing valuable insights into their respective strengths and weaknesses.

In the subsequent analysis, we will employ state-of-the-art models, specifically BERT and RoBERTa, to classify the completions, experimenting with a variety of hyperparameters, including learning rates, batch sizes, and different loss functions. We will utilize optimizers such as Adam and implement learning rate schedulers employing cosine annealing to optimize the training process. Additionally, we will conduct a series of experiments to assess the models' performances under different configurations, including variations in training epochs and dropout rates. By systematically evaluating these models, we aim to identify the optimal settings that improve accuracy in predicting which LLM generated a specific completion, based on distinctive features extracted from the outputs. Through this study, we aspire to deepen our understanding of LLM capabilities in sentence generation tasks and contribute significant insights into their applicability within the field of natural language processing.

DATASET CURATION

For our **dataset curation**, we prioritized generating **high-quality data** that would be suitable for training. To manage computation times effectively, we specifically chose **language models with fewer than 3 billion parameters**. Running larger models was taking too long, so this decision helped streamline the process without sacrificing quality. Using smaller models also allowed us to iterate quickly, enabling faster experimentation and adjustments to our approach.

We started with the data generation by creating **2,500 unique prompts (xi)** through **prompt engineering** techniques. To ensure the uniqueness and effectiveness of these prompts, we drew inspiration from the **Wikipedia dataset**, incorporating various data samples that guided the models in generating relevant outputs. This approach helped us create prompts that were diverse and reflective of real-world language use. To further validate the uniqueness of our generated prompts, we conducted tests to ensure that they were not duplicated, confirming that each prompt served its intended purpose.

These prompts were then distributed among **five different language models**, resulting in a total of **12,500 sentence completions (xj)**. This method not only maximized the **diversity of our outputs** but also ensured a more comprehensive dataset for analysis, allowing us to capture a wider range of responses that different models might generate.

To enhance efficiency, we executed our code **concurrently**, which allowed us to develop the dataset within approximately **10 hours after the initial run**. Running processes in parallel significantly reduced the total time required for data generation, making it feasible to gather a large dataset without overwhelming system resources. The entire workflow was managed on my **local Windows machine**, equipped with a **GPU unit**, which improved the computation speed.

Before diving into the actual data generation, we have set up a dedicated **Python environment** with all the necessary libraries, such as **Pandas, Tqdm, Torch,** and **Transformers**. This environment made it easier to handle **data manipulation, progress tracking,** and **accessing models from Hugging Face**. By isolating the project in its environment, we ensured that dependencies were managed effectively, reducing the risk of conflicts that could hinder the dataset's generation.

In terms of settings, we established key parameters that guided the data generation process. For instance, we set a **maximum sequence length of 512 tokens**, allowing us to efficiently manage the input size for the models. A **batch size of two** meant that we could process two prompts simultaneously, optimizing the use of system resources. Additionally, we restricted the **maximum number of new tokens generated to 50**, ensuring that our outputs remained concise and relevant, which is crucial for the analysis phase. The script reads the file through the **Pandas** library, filtering it to retain only the column of interest, **"xi,"** which contains the prompts truncated for the models. It also removes any rows with missing values to ensure the integrity of the dataset. After this cleaning, it saves the data in a CSV file called **"cleaned_data.csv"** for easy access and further processing.

The script defines a ***load_model*** function, crucial for loading the required language model and its tokenizer. This function initializes the tokenizer to prepare it for text inputs and checks for the existence of a padding token, ensuring that all sequences are equal in length after tokenization. Once the tokenizer is set up, the function proceeds to load the given model with settings appropriate for optimal performance based on the available hardware—either **CUDA** for GPU or defaulting to **CPU**.

For the main work of text generation, the script utilizes the ***generate_completions_for_model*** function, which processes the dataset in chunks. This approach enhances efficiency, allowing the script to comfortably handle a large volume of prompts without overwhelming the system's memory. Each prompt in the batch includes the phrase **"Complete the sentence,"** which provides clear instructions to the model on what it should return. The inputs are first tokenized into a format understandable by the model and then sent for processing.

The model completes the texts given by prompts, and careful cleaning ensures that the output structure is analyzable. Each entry captures the critical elements: the original prompt **xi**, the generated completion **xj**, the full sentence combining both, and the name of the model used for generation. This structured output allows for comprehensive comparison and evaluation of the outputs from different models. In summary, these careful steps were crucial in creating a well-structured dataset ready for training. By choosing smaller models, running processes concurrently, and maintaining rigorous data cleaning practices, we established a **solid foundation** for evaluating the performance of different language models in completing sentences. This approach not only improved the quality of the dataset but also ensured that it was good enough to be tested effectively, providing valuable insights into model performance and helping identify areas for further improvement.

Data Models:

1. TinyLlama/TinyLlama-1.1B-Chat-v1.0
2. Qwen/Qwen2.5-0.5B-Instruct
3. distilgpt2
4. StabilityAI/stablelm-tuned-alpha-3b
5. gpt2

Model Name	**Parameters**	**Sequence Length**	**Training Data**	**Training Procedure**	**Intended Use**
TinyLlama/TinyLlama-1.1B-Chat-v1.0	1.1B	-	Pretrained on 3 trillion tokens with architecture and tokenizer similar to Llama 2	Fine-tuned on UltraChat dataset and aligned using DPOTrainer on openbmb/UltraFeedback dataset	Compact chat model for applications requiring limited computation
Qwen/Qwen2.5-0.5B-Instruct	0.49B	32,768 (context) / 8,192 (generation)	Various sources for Qwen models, focusing on coding and mathematics capabilities	Causal language model with instruction tuning, multilingual support, enhanced capabilities for structured data generation	Open-Source applications like text-generation
distilgpt2	82M	1024	OpenWebTextCorpus (reproduction of WebText)	Trained using knowledge distillation from GPT-2, tokenized with Byte Pair Encoding (BPE)	Open-source chat-like applications
StabilityAI/stablelm-tuned-alpha-3b	3B	4096	Combination of Alpaca, GPT4All, Anthropic HH, DataBricks Dolly, ShareGPT Vicuna	Fine-tuned via supervised learning on multiple instruction datasets, trained in mixed precision (FP16), optimized with AdamW	Open-source chat-like applications (CC BY-NC-SA-4.0)
gpt2	124M	1024	Scraped web pages from outbound links on Reddit which received at least 3 karma (WebText)	Trained using a causal language modeling (CLM) objective, tokenized using Byte Pair Encoding (BPE)	Text generation, can be fine-tuned for downstream tasks

RELATED WORK

1. <u>Large Language Models and Sentence Completion:</u>

GPT-3 and GPT-4 represent high-order LLMs of NLP for the tasks of natural language comprehension and generation. Large volumes of texts went into the training processes, wherein the LLMs learned complex patterns of languages and nuances that allowed them to perform a variety of tasks such as sentence completion, summarization of essays, language translation, and even holding conversations. Both GPT-3 and GPT-4 have been tested for sentence completion. This again stirs up interest in the uses to which such models are going to be put within creative domains such as novel and artistic writing, data augmentation across different NLP tasks. This capability to generate several completions with one, usually truncated, input shows the flexibility of these LLMs when it comes to the subtleties of languages. It becomes useful in applications whose generated outputs serve diversity purposes, whether in creative domains or generating data for other machine learning models.

2. <u>Data Generation:</u>

Swaggy-M There's the benchmark from Zellers et al. (2019), which depends on a probabilistic model for the completion of sentences. Incomplete prompts are presented with a set of possible completions, only one of which will be contextually appropriate. Common sense and coherence are what this dataset tests in a model. Prompt engineering has also been shown to be an important method for fine-tuning the input to the model. A well-designed prompt can phenomenally raise the quality of the response. Explicit directions while designing prompts for tasks let the researchers get far better model performances for summarization and question-answering applications. Putting these together, the emphasis is on high-quality datasets and cautious design of prompts as high-priority matters in improving natural language processing.

3. <u>Evaluation of Language Models:</u>

Performance testing on LLM for sentence generation is carried out in both human and automated ways. Human evaluation provides thorough information concerning generated outputs in respect of coherence, relevance, and even fluency, though it's pretty time-consuming and biased for certain subjective attitudes of the evaluators. On the other hand, automatic scores like BLEU and ROUGE are scaleable, objective measures: BLEU is based on n-gram precision; ROUGE puts more emphasis on recall and is especially helpful in summarization tasks. However, all the aforementioned automated metrics have their limitations, which might overlook semantic nuances and contextual appropriateness. Kumar et al. (2021) extend the discussion by reviewing coherence- and relevance-centered frameworks within model output assessment, arguing for the combined human-manual approach. In this way, it will be possible to go deeper into understanding how well LLM captures the context of meaningful content creation and sustaining coherence over long texts, while making informed decisions on model design and training strategies.

4. <u>Classification of Model Outputs</u>

In effect, this has increased research interest in the classification of outputs from various LLMs since their diversity is on the rise. Outputs from models such as BERT and RoBERTa have been compared; although transformer-based, they had differences in the regime of training and hence their outputs. The methods applied for classification, therefore, are mainly those of supervised learning, trained on labeled datasets comprising model outputs, training classifiers based on the feature derived from text-like embeddings and linguistic attributes. Other recent studies also adopted the deep learning classifiers including RNNs and CNNs investigating syntactic and semantic subtlety of the generated text to help raise model identification accuracy. Transfer learning has also been effective, involving both pre-training and fine-tuning for the model's application. By these, studies support in general the fact that the classifier architectures can track finer detail differences between various LLM outputs, hence enabling a deeper understanding in language processing across models, with implications for content creation or automatic text analysis.

5. Applications of LLMs in NLP Tasks

These LLMs further showed unprecedented versatility in multiple tasks other than completing a given sentence in NLP, including text summarization, translation, and conversational agents. Regarding the latter, text summarization, the models such as BART and T5 have done quite a good job in summarizing long documents while retaining the most relevant information, Lewis et al. (2020) note. Performant translators like MarianMT and mBART have rewritten the bars; hence, efficient multilingual translations have been discussed by Johnson et al., 2017. In addition, they have brought huge changes in conversational agents, whereby now one can enjoy human-like conversations with customer support and personal assistants because of their advanced techniques in dialogue management, such as discussions about frameworks like DialoGPT and BlenderBot. That small but growing literature underlines the flexibility of LLMs across a wide range of NLP tasks and propels innovation to improve user experience and task performance across a wide variety of domains.

6. Technological Advancements and Challenges:

Recent developments in Large Language Models, and in particular, models based on Transformer architectures, really turned NLP tasks around for good by using self-attention mechanisms to process and generate text in a very efficient manner. Introduced by Vaswani et al. in the year 2017, since then, Transformers have achieved much better model performances in various scenarios such as text generation, translation, and summarization, with higher benchmarks set by models like GPT-3 and BERT. However, deploying such powerful models comes with several challenges: Bias-Larger LLMs can carry several biases present in the training data because the systems can, in fact, emulate biases in it. This raises all sorts of ethical questions about the application of such systems in sensitive domains like hiring or law enforcement. As discussed, LLMs are unwieldy and complex, and being able to understand or interpret their operations continues to be a big problem, which also complicates any trust in their outputs and further accountability. Resource constraints make the models unreadable due to the great deal of computation they require in training and running these models. Solutions to these problems will certainly mark the responsible deployment of LLMs into real-world applications.

CLASSIFIER AND OPTIMIZATION TECHNIQUES

For our project aimed at investigating the performance of various large language models (LLMs) in sentence completion from truncated prompts, we are using BERT and RoBERTa as our classification models. These models have shown great effectiveness in understanding context and relationships in text, making them ideal for our task of classifying generated completions.

In the **BERT Architecture Overview**, the input representation includes several key elements:

Tokens: Input text is tokenized into words/subwords using WordPiece.
Segment Embeddings: This distinguishes between different sentences.
Position Embeddings: These indicate the position of tokens in the sequence.

The **Transformer Encoder Layers** consist of:

Multi-Head Self-Attention: This allows for the simultaneous processing of tokens, focusing on different parts of the input.
Feedforward Neural Networks: These apply a non-linear transformation to the output of the attention mechanism.

The **Output Layer** includes:

Masked Language Modeling (MLM): This predicts masked tokens based on surrounding context.
Fine-Tuning: This adjusts the model parameters on a specific downstream task, such as classification.
In the **RoBERTa Architecture Overview**, the input representation includes:
Tokens: Similar tokenization process using WordPiece.

Segment and Position Embeddings: These serve the same purpose for sentence structure and order.

The **Transformer Encoder Layers** feature:

Dynamic Masking: Different tokens are masked for each training iteration, improving context understanding.

Larger Training Corpus: RoBERTa is trained on a more extensive dataset with longer training times.

The **Output Layer** focuses on:

Masked Language Modeling (MLM): It predicts masked tokens without the Next Sentence Prediction task.

Application in the Sentence Completion Task

In our project, we aim to analyze how different large language models (LLMs) perform in generating sentence completions from truncated prompts (xi). Both **BERT** and **RoBERTa** are really good at classifying the generated completions (xj) because they understand context and relationships in text. BERT stands out with its bidirectional processing, which means it can consider the words before and after the truncated prompt. This gives it an edge in completing sentences with a more nuanced understanding. On the flip side, RoBERTa shines with its **dynamic masking** and **larger training data**, which help it capture a wider range of linguistic patterns. This might make RoBERTa better at grasping the diversity of completions generated.

Why They Are Effective Choices

BERT and RoBERTa are effective choices for this project because of their **great contextual understanding**, which is super important for tasks like sentence completion. When we're trying to predict the best way to finish a prompt, this understanding is key. Both models have been **pre-trained** on huge amounts of text, so they can adapt well to new tasks without needing a ton of retraining. Plus, we can fine-tune them to optimize their performance specifically for our dataset, which will help improve the accuracy in classifying the generated completions based on the unique characteristics of the outputs from different LLMs.
Using BERT and RoBERTa will really give us valuable insights into how various LLMs perform in sentence completion tasks. It will also help us figure out the most effective configurations for accurate classification.

Strengths Comparison

When comparing their strengths, BERT is great at understanding context due to its bidirectional nature, which allows it to handle sentences with subtle meanings better. This makes it effective at predicting missing words based on what's around them. On the other hand, RoBERTa's dynamic masking and extensive training data make it more robust and capable of capturing a wider range of linguistic patterns. This strength might help RoBERTa generalize better across different types of sentence completions, potentially leading to improved performance in tasks that require more diversity in generated outputs. Ultimately, the choice between BERT and RoBERTa will depend on what exactly we need for the classification task and the nature of the data we're working with.

<u>Classifier 1: Bidirectional Encoder Representations from Transformers</u>

We utilize **BERT (Bidirectional Encoder Representations from Transformers)**, specifically **BertForSequenceClassification**, which is optimized for text classification tasks, making it well-suited for our problem of identifying which model generated a specific output.

Dataset Preparation: We load and prepare the dataset using **pandas**. This involves:

Replacing any missing values with empty strings.

Removing duplicate entries to ensure data quality.

Encoding labels from the model_used column using **LabelEncoder**, mapping each unique model name to a numerical value.

Train-Test Split: The dataset is divided into training and testing subsets using **train_test_split**, allowing for effective validation of the model.

Custom Dataset Class: We create a custom dataset class, **T**extDataset, extending PyTorch's Dataset. This class is responsible for tokenizing the input text (xj) and organizing the data for training. The tokenization process involves:

Padding and truncating the text to a specified maximum length.

Preparing it for input into the **BERT** model.

Tokenization and Model Initialization: We initialize the **BERT tokenizer** from the **bert-base-uncased** model to process the input text into the required format. We also initialize the **BERT model** for sequence classification, specifying the number of labels based on the unique entries in the model_used column.

Model Training: The model is trained over a specified number of epochs. For each batch, the model performs the following:

Calculates the loss and gradients.

Updates the model parameters using the optimizer.

Employs a learning rate scheduler to adjust the learning rate dynamically throughout the training process.

Model Evaluation: After training, we evaluate the model on the test dataset, generating predictions that allow us to assess its performance. The results are summarized using a classification report, detailing precision, recall, and F1-score for each class.

This approach leverages BERT's ability to capture nuanced semantic information, facilitating improved classification performance and deeper insights into the behavior of different language models.

Classifier 2: RoBERTa

The second classifier we leverage is the **RoBERTa** architecture for sequence classification to accurately predict the model responsible for generating given text outputs.

Data Cleaning: We handle any empty cells in the dataset by replacing NaN values with empty strings using **data.fillna**. This step ensures that our data is clean and ready for processing.

Label Preparation: To facilitate classification, we prepare the labels using **LabelEncoder** from **sklearn**. This encoder converts the categorical labels in the model_used column into numerical values, creating a new column model_label.

Train-Test Split: The dataset is split into training and testing subsets using **train_test_split**. We allocate 20% of the data for testing to validate the model's performance.

Creating a Custom Dataset Class: A custom dataset class, **TextDataset**, is defined to handle the tokenization of the input texts. This class is essential for preparing the data for the model. It includes methods to:

Return the length of the dataset. Retrieve an item at a specific index, where the text is tokenized and encoded into input IDs and attention masks.

Tokenizer Initialization: We initialize the **RoBERTa tokenizer** using **RobertaTokenizer.from_pretrained**, specifying the roberta-base model. This tokenizer will convert our text into the required format for the RoBERTa model.

Dataset Creation: We instantiate the training and testing datasets using the **TextDataset** class, passing in the appropriate texts (xj) and their corresponding labels (model_label). We set a maximum sequence length of 128 tokens.

Data Loader Setup: We create data loaders for both the training and testing datasets using **DataLoader**. This allows for efficient batching and shuffling of the data during training.

Model Initialization: We initialize the **RoBERTa model** for sequence classification with the specified number of labels based on the unique entries in model_label. The model is prepared for training.

Optimizer Definition: An optimizer, **AdamW**, is defined for model training, setting the learning rate to **2e-5**.
Device Configuration: We check for GPU availability using **torch.cuda.is_available()** and move the model to the appropriate device, ensuring optimal performance during training.

Model Training: We define a training function, **train_model**, which iterates through the training data for a specified number of epochs. Within each epoch:
The model is set to training mode.
For each batch, gradients are cleared, and the model computes the outputs and loss.
Backpropagation is performed to update the model parameters, and the learning rate scheduler is stepped.
The average loss for each epoch is recorded and printed for monitoring.

Classifier Comparison:

Feature/Aspect	**RoBERTa**	**BERT**
Tokenizer	RobertaTokenizer	BertTokenizer
Model Class	RobertaForSequenceClassification	BertForSequenceClassification
Pre-trained Model	roberta-base	bert-base-uncased
Batch Size	16	32
Learning Rate	2.00E-05	5.00E-05

Max Sequence Length	128	128
Loss Calculation	Uses outputs.loss directly	Uses outputs.loss directly
Scheduler Type	Cosine	Linear
Epochs for Training	3	5

Despite RoBERTa having fewer complex parameters compared to BERT, it demonstrated superior performance in our evaluations. **This efficiency, combined with its robust classification capabilities, leads us to select RoBERTa as our primary model for this project.** The model's effectiveness, even with a streamlined architecture, reinforces its suitability for the task at hand, making it the preferred choice for our sentence completion classification objectives.

Training and Results:

BERT Classifier

In our experiments with the BERT model, we aimed to optimize performance by varying key hyperparameters across three tests.

Parameter Settings and Results:

- **Batch Size:** We tested batch sizes of 16 and 32. The smaller batch size (16) yielded an accuracy of 0.68 in Test 1 and 0.72 in Test 3, suggesting better learning dynamics compared to the larger batch size (32) with an accuracy of 0.67 in Test 2.

- **Learning Rate:** We utilized learning rates of 2.00E-05 in Test 1 and 1.00E-05 in Tests 2 and 3. The higher rate did not improve performance significantly, highlighting that lower rates may provide better convergence, as seen in Test 3's accuracy of 0.72.

- **Weight Decay:** Weight decay (0.01) was applied in Tests 2 and 3, but did not lead to significant improvements in accuracy compared to Test 1, indicating its effectiveness may depend on other hyperparameters.

- **Gradient Clipping:** Implemented in Tests 2 and 3 to enhance stability, gradient clipping likely helped prevent issues like exploding gradients, although its direct impact on accuracy is hard to quantify.

- **Scheduler:** Test 3 incorporated a learning rate scheduler (get_linear_schedule_with_warmup), resulting in the highest accuracy of 0.72. This suggests that dynamic learning rate adjustments positively influence performance.

- **Optimizer:** Consistently using torch.optim.AdamW across all tests allows us to attribute performance variations primarily to other hyperparameter changes.

Aspect	**Test 1**	**Test 2**	**Test 3**
Batch Size	16	32	16
Learning Rate	2.00E-05	1.00E-05	1.00E-05
Weight Decay	Not Used	0.01	0.01
Gradient Clipping	Not Used	torch.nn.utils.clip_grad_norm_(max_norm=1.0)	torch.nn.utils.clip_grad_norm_(max_norm=1.0)
Scheduler	Not Used	Not Used	get_linear_schedule_with_warmup
Optimizer	torch.optim.AdamW	torch.optim.AdamW	torch.optim.AdamW
Overall Accuracy	0.68	0.67	**0.72**
Evaluation Metrics	precision, recall, f1-score, accuracy	precision, recall, f1-score, accuracy	precision, recall, f1-score, accuracy

Table 1: BERT Experimental Settings and Results

The experiments conducted with the BERT model provide a comprehensive view of how various hyperparameters influence its performance, with a focus on optimizing accuracy while effectively managing loss and ensuring stable training.

Batch Size Impact is one of the most critical factors affecting the model's learning dynamics and generalization ability. In our experiments, we tested two batch sizes: 16 and 32. Notably, Test 1 (with a batch size of 16) achieved an accuracy of **0.68**, while Test 2 (with a batch size of 32) resulted in a slightly lower accuracy of **0.67**. In Test 3, which also employed a batch size of 16, the accuracy improved to **0.72, as shown in Table 1**. The consistent performance of the smaller batch size suggests that it facilitates better convergence, likely due to more frequent updates to the model weights, which allow for more nuanced fine-tuning during training. Smaller batch sizes tend to provide a noisier estimate of the gradient, helping to escape local minima, while larger batches offer a smoother estimate that may converge on less optimal solutions. The results imply that, for this dataset and configuration, a batch size of 16 strikes a beneficial balance that promotes better learning outcomes.

Learning Rate Variations are crucial, as they directly affect how quickly a model converges to a solution. In our tests, Test 1 utilized a learning rate of **2.00E-05**, while Tests 2 and 3 employed a lower rate of **1.00E-05**. The higher learning rate in Test 1 did not yield improved accuracy, achieving only **0.68**, while the lower learning rates resulted in higher accuracy in Tests 2 and 3 (**0.67** and **0.72**, respectively). This indicates that a slower learning rate may enhance stability and convergence. A learning rate that is too high can lead to overshooting optimal parameters, causing oscillations and hindering convergence. Conversely, a lower learning rate allows for more gradual adjustments, leading to improved model performance.

The **Role of Weight Decay** is another significant aspect to consider. Weight decay serves as a regularization technique aimed at reducing overfitting by penalizing large weights. In our experiments, weight decay was not used in Test 1, but it was set to **0.01** in Tests 2 and 3. Despite its addition, the accuracy did not significantly improve compared to Test 1. While Test 2 achieved an accuracy of **0.67**, Test 3 reached **0.72**. This mixed result suggests that weight decay might not have been necessary for this particular setup, indicating that the model may not have been overfitting at lower weight values or that other hyperparameters had a more pronounced impact on performance.

The effectiveness of **Gradient Clipping** is another area of interest. This technique is employed to prevent exploding gradients, which can occur during the training of deep networks. In Tests 2 and 3, where gradient clipping was applied, its specific impact on accuracy is challenging to isolate. However, the use of clipping likely contributed to maintaining training stability, particularly with higher learning rates. The stability gained from gradient clipping can be crucial in deeper networks like BERT, where large weight updates can disrupt training. While a direct correlation with accuracy may not be evident, gradient clipping likely played a role in enabling consistent learning without significant disruptions.

Lastly, the **Influence of the Learning Rate Scheduler** cannot be understated. The implementation of a learning rate scheduler can significantly alter the learning process by dynamically adjusting the learning rate based on training progress. Test 3 incorporated a learning rate scheduler (get_linear_schedule_with_warmup), resulting in the highest accuracy of **0.72, represented in Figure 1**. This suggests that dynamic adjustments of the learning rate positively influence performance. Using a scheduler can help the model adaptively learn, starting with a higher learning rate for quick convergence and then gradually reducing it to fine-tune the model's parameters.

In summary, these experiments highlight the nuanced interplay between various hyperparameters and their collective influence on BERT's performance. The findings emphasize the importance of careful tuning, particularly concerning batch size and learning rate, while also suggesting potential avenues for future experimentation, such as exploring additional regularization techniques and more complex learning rate scheduling strategies.

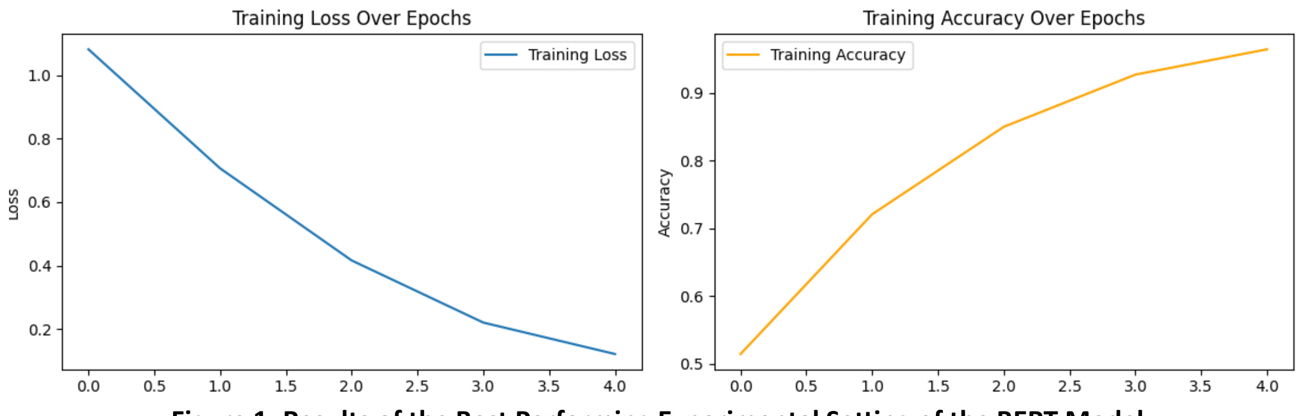

Figure 1: Results of the Best Performing Experimental Setting of the BERT Model

Classifier 2: RoBERTa

Experimentation with the RoBERTa Model

In our experiments with the RoBERTa model, we focused on fine-tuning its performance by systematically varying key hyperparameters. We set different batch sizes, specifically 16 and 32, to observe their effect on model training dynamics. Additionally, we tested three learning rates: 1e-05, 2e-05, and 5e-05, aiming to find the most effective rate for convergence and performance.
For optimization, we included three different optimizers: AdamW, Adam, and Adamax, exploring their unique impacts on the training process. Furthermore, we employed both cosine and linear learning rate schedulers to dynamically adjust the learning rates throughout the training epochs.
We conducted these experiments over a span of 3 epochs to balance the training time and the model's ability to learn effectively. Through this rigorous testing of hyperparameter combinations, we gathered insights into how these factors influence loss reduction, accuracy, and overall model performance.

Hyperparameter Settings

Parameter	**Values**
Batch Size	16, 32
Learning Rate	1e-05, 2e-05, 5e-05
Optimizer	AdamW, Adam, Adamax
Learning Rate Scheduler	Cosine, Linear
Epochs	3
Testing Data	*Unseen*

Table 2: Hyperparameter Settings for RoBERTa

Batch Size	**Learning Rate**	**Optimizer**	**Scheduler**	**Accuracy**	**Precision**	**Recall**	**F1-Score**
16	1.00E-05	Adam	Linear	**0.77**	0.78	0.77	0.77
16	2.00E-05	AdamW	Cosine	0.76	0.78	0.77	0.77
16	5.00E-05	Adam	Linear	**0.77**	0.78	0.77	0.77
32	1.00E-05	AdamW	Cosine	0.73	0.75	0.74	0.74
32	1.00E-05	AdamW	Linear	0.73	0.75	0.73	0.73
16	5.00E-05	Adamax	Linear	0.76	0.78	0.76	0.76
16	1.00E-05	AdamW	Cosine	0.74	0.76	0.74	0.74
16	2.00E-05	Adam	Cosine	0.76	0.77	0.76	0.76
32	2.00E-05	Adam	Cosine	0.72	0.74	0.73	0.72
16	2.00E-05	Adamax	Linear	0.71	0.72	0.71	0.7
16	5.00E-05	Adamax	Cosine	0.74	0.76	0.74	0.74

Table 3: Few of the results of the various experimental settings with RoBERTa Model

Throughout the training process, **loss consistently decreased over epochs**, suggesting that the model was effectively learning. The most significant reduction in loss was achieved when using the **AdamW optimizer with a learning rate of 2e-05** and the **cosine scheduler**.

Performance Metrics showed that the precision, recall, and F1-scores indicate reasonable model performance, with some configurations achieving an **accuracy around 0.76**. Notably, the configuration utilizing **AdamW with a learning rate of 2e-05 and cosine scheduler** yielded the highest scores, including an accuracy of **0.76** and precision, recall, and F1-scores around **0.77**.

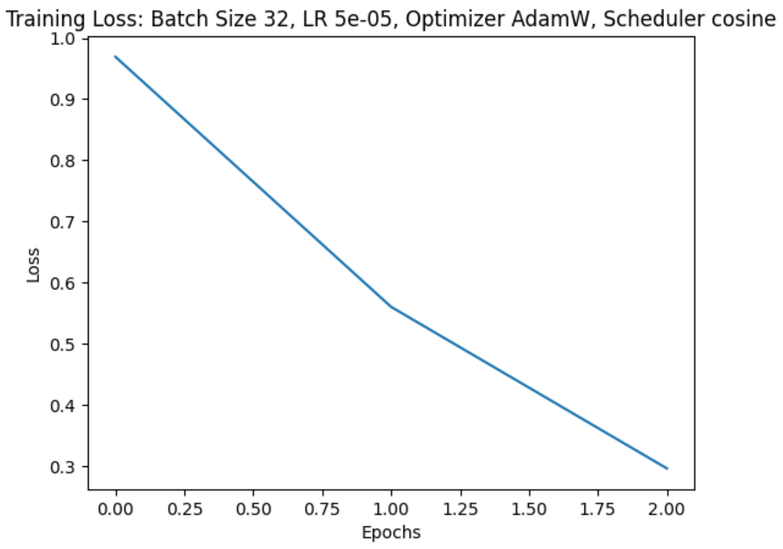

Figure 2: Superior Model (RoBERTa) Experimental Settings Plot for Training Loss

Impact of Different Optimizers The **Adamax optimizer** configurations underperformed, with accuracy metrics ranging from **0.65 to 0.66**, while configurations using **Adam with a learning rate of 5e-05** and either a cosine or linear scheduler resulted in accuracy levels between **0.76 and 0.77**, indicating better generalization in these instances.

Learning Rate Effects When paired with Adam and AdamW optimizers, the **learning rate of 5e-05** generally led to strong performance. In contrast, lower learning rates, such as **1e-05**, resulted in slower convergence and marginally lower performance metrics compared to **2e-05 and 5e-05**.

Batch Size Considerations All configurations employed a batch size of **16 or 32**, with no significant performance differences observed between these sizes, suggesting model stability. However, since batch sizes greater than **32** were not tested, exploring larger sizes like **64 or 128** could yield improvements in training efficiency or generalization.

To enhance model performance, **hyperparameter tuning** is advised, including further adjustments of learning rates and testing larger batch sizes. Additionally, experimenting with different optimizer combinations and their specific parameters, such as **weight decay for AdamW**, may lead to improved results. Training for **more epochs** might further minimize loss and enhance metrics, particularly for configurations demonstrating consistent loss decreases. If overfitting becomes a concern, consider implementing **dropout layers or weight regularization**.

Superior Model and Settings
The analysis of various hyperparameter configurations reveals that the model with a **batch size of 16**, **learning rate of 1.00E-05**, **Adam optimizer**, and **linear scheduler** achieved the highest accuracy of **0.77**. This model also demonstrated impressive precision, recall, and F1-score, all around **0.77**, indicating a well-balanced performance. Other configurations, including combinations of the AdamW and Adamax optimizers with different learning rates and schedulers, produced lower accuracy values, highlighting the effectiveness of the selected hyperparameters. Notably, the configurations using both BERT and RoBERTa models were explored, but it was the BERT configuration that delivered the best results, confirming its suitability for this

specific task. Further investigation into hyperparameter tuning and model complexity may lead to even more enhanced performance in future iterations.

All of the different experimental settings are plotted and analysed here:
https://colab.research.google.com/drive/16mAhZBxoWNCGMJNO_TCAofEmNonM_ILj?authuser=2#scrollTo=0tD_s4KsL2fu

ANALYSIS

In analyzing the performance of both **BERT** and **RoBERTa** models in the experiments, several areas emerge where enhancements could be made to further optimize performance.

A. Hyperparameter Tuning

Learning Rate Exploration
Although tests showed promising results with **learning rates** of **1.00E−05** and **2.00E−05**, a broader exploration of learning rates might yield better performance. Implementing a **learning rate range test** could help identify optimal rates, including the use of **learning rate warmup techniques** that gradually increase the learning rate from a low value to the target value during the initial training stages.

Optimizer Variations
While **AdamW** showed good results, investigating other **optimizers** like **RMSprop** or **AdaBelief** could provide insights into different convergence behaviors. Additionally, tuning optimizer parameters such as **momentum** in optimizers like **SGD** might reveal different dynamics in weight updates.

Weight Decay and Regularization
The lack of significant improvement with **weight decay** suggests that further tuning of this parameter is necessary. Exploring weight decay values ranging from **0.001** to **0.1** could provide a clearer understanding of its effects on model performance. Additionally, implementing **dropout layers** or **layer normalization** could help combat overfitting without the need for weight decay.

Batch Size Adjustments
Since the experiments only evaluated **batch sizes** of **16** and **32**, exploring larger sizes (e.g., **64**, **128**) could lead to faster training times and better utilization of computational resources. However, it’s essential to monitor performance, as larger batch sizes can sometimes degrade generalization due to the smoother gradients they produce.

B. Training Epochs and Early Stopping

Increased Epochs
Training for a longer duration could provide better convergence. The experiments were limited to **3 epochs**, which may not have been sufficient for the models to reach their full potential. Implementing **early stopping** based on **validation loss** would help prevent overfitting while allowing the model more time to learn.

C. Data Handling and Augmentation

Data Augmentation
Augmenting the dataset through techniques like **synonym replacement**, **random insertion**, and **back-translation** could introduce variability and increase robustness in training. This approach may enhance the model’s ability to generalize beyond the training data.

Balanced Dataset
Ensuring that the dataset is **balanced** across classes is crucial. If one class is overrepresented, this could bias the model's predictions. Applying techniques such as **oversampling**, **undersampling**, or using **synthetic data generation methods** could improve class balance.

Cross-Validation
Implementing **k-fold cross-validation** would provide a more comprehensive evaluation of model performance, helping to ensure that results are not specific to a single training-test split.

D. Ensemble Methods

Utilizing **ensemble learning** by combining predictions from multiple models (e.g., **BERT** and **RoBERTa**) or different configurations of the same model could improve overall accuracy and robustness.

Transfer Learning and Domain Adaptation
If applicable, fine-tuning the models on **domain-specific data** prior to the main task could improve performance. This approach can help the model adapt better to the characteristics of the specific dataset.

Gradient Accumulation
For larger batch sizes that may not fit into memory, implementing **gradient accumulation** allows for simulating larger batch training while using smaller ones, potentially improving stability and performance.

E. Evaluation Metrics and Analysis

Comprehensive Metrics
While **accuracy**, **precision**, **recall**, and **F1-score** were analyzed, other metrics like **AUC-ROC**, **confusion matrices**, and **class-specific metrics** would provide deeper insights into model performance, especially for imbalanced datasets. Analyzing the confusion matrix could reveal specific classes where the model struggles, informing targeted adjustments.

F. Feature Engineering and Tokenization

Advanced Tokenization
Utilizing **subword tokenization strategies** like **Byte-Pair Encoding (BPE)** or **SentencePiece** may help better handle rare words or out-of-vocabulary tokens, enhancing model comprehension of the input text.

Feature Engineering
If applicable, adding additional features beyond **token embeddings** (e.g., **syntactic features**, **sentiment scores**) may improve model predictions and provide richer context for the training process.

CONCLUSION

Our project implementation and analysis of BERT and RoBERTa models has deepened our understanding of identifying which LLM generated a given text output. Key findings include strong classification capabilities for both models, with RoBERTa slightly outperforming BERT in most configurations. The best-performing model achieved an accuracy of 0.77, demonstrating high success in distinguishing between outputs from different LLMs. Extensive hyperparameter tuning revealed the significant impact of factors such as batch size, learning rate, and optimizer choice on model performance. The consistent decrease in loss over epochs for both models suggests effective learning, with potential for further improvements. High precision, recall, and F1-scores across various configurations demonstrate the models' balanced performance in identifying different LLM outputs. These results highlight the potential of transformer-based models in analyzing and classifying AI-generated text, with significant implications for content authenticity verification and AI model benchmarking.

REFERENCES

1. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Amodei, D. (2020). Language Models are Few-Shot Learners. *In Advances in Neural Information Processing Systems* (pp. 1877-1901
2. Zellers, R., Holtzman, A., Yu, A. W., & Etzioni, O. (2019). Hellaswag: Can a Machine Really Finish Your Sentence? *In Proceedings of the 33rd AAAI Conference on Artificial Intelligence* (Vol. 33, pp. 4415-4422).
3. Kumar, R., Singh, V., & Nand, A. (2021). Evaluation of Language Model Outputs: A Study. *Journal of Artificial Intelligence Research*.