CPEN 455 Final Project Report

Sasan Ghasaei
University of British Columbia
sasangh@student.ubc.ca

1 Introduction

In this project, we will explore the application of large language models (LLMs) to the task of email spam detection through advanced classification techniques. We leverage a pretrained LLM model called SmolLM2-135M-Instruct and will we use full fine-tuning approach to adapt the model to our specific task. The core methodology imployed is Bayesian inverse classification which exploits the generative capabilities of LLMs to model the joint distribution of inputs and labels, enabling computation of posterior probabilities for effective spam classification. We also utilize techniques such as prefix-tuning and data synthesis to enhance model performance and robustness. Our experiments will evaluate the effectiveness of these methods in improving spam detection accuracy. We finally present that our best model achieves an accuracy of 97.3% on 71% of the test dataset by just training on a very small subset of the entire data.

2 Model Description

We utilize the SmolLM2-135M-Instruct model, a large language model pre-trained on a diverse corpus of text data. The model architecture is based on the transformer framework, which allows it to capture long-range dependencies in text. For our spam detection task, we fine-tune the model using a labeled dataset of emails, where each email is classified as either "spam" or "not spam."

2.1 Decoder-Only Transformer

The SmolLM2-135M-Instruct model employed in this project is based on a decoder-only transformer architecture, which has become the dominant paradigm for modern large language models. Unlike the original transformer architecture that uses both encoder and decoder components, decoder-only models consist solely of a stack of decoder layers, making them naturally suited for autoregressive text generation tasks.

2.1.1 Architecture Overview

The decoder-only transformer processes input tokens through the following components:

Token Embeddings: Input tokens are first mapped to dense vector representations of dimension d_{model} (hidden size). These embeddings convert discrete token IDs into continuous vectors that capture semantic information.

Positional Encoding: Since transformers have no inherent notion of token order, positional information is encoded using Rotary Position Embeddings (RoPE). RoPE applies rotation matrices to query and key vectors, enabling the model to capture relative positions between tokens more effectively than traditional sinusoidal or learned positional embeddings.

Transformer Layers: The model consists of L stacked decoder layers. Each layer contains two sub-components connected via residual connections:

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

1. **Multi-Head Self-Attention:** This mechanism allows each token to attend to all previous tokens (and itself) in the sequence through a causal mask. The attention operation is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V \tag{1}$$

where Q, K, and V are query, key, and value matrices, d_k is the dimension per attention head, and M is a causal mask that sets future positions to $-\infty$.

2. **Feed-Forward Network (MLP):** After attention, each token representation is independently processed through a two-layer feed-forward network:

$$\text{FFN}(x) = W_2 \cdot \text{SiLU}(W_1 x + b_1) + b_2 \tag{2}$$

where SiLU (Sigmoid Linear Unit) is the activation function, and the intermediate dimension is typically $4 \times d_{\text{model}}$.

Both sub-components use pre-normalization with RMSNorm (Root Mean Square Layer Normalization), which normalizes activations without computing mean centering:

$$\text{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}} \cdot \gamma \tag{3}$$

Language Modeling Head: After the final layer norm, the hidden states are projected to vocabulary logits via a linear transformation, producing probability distributions over the next token.

2.1.2 Causal Masking

A critical feature of decoder-only models is causal masking, which ensures that during training and inference, each token can only attend to itself and previous tokens. This autoregressive property allows the model to be trained on next-token prediction and naturally extends to text generation. The mask matrix M has the structure:

$$M_{i,j} = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases} \tag{4}$$

2.1.3 Key-Value Caching

For efficient autoregressive generation, the model implements key-value (KV) caching. During generation, previously computed key and value tensors are stored and reused, avoiding redundant computations. This reduces the computational complexity from $O(n^2)$ to $O(n)$ per generated token, where n is the sequence length.

2.1.4 Draw Backs of Key-Value Caching

While key-value caching significantly speeds up autoregressive generation, it introduces some limitations:

- **Increased Memory Usage:** Storing keys and values for long sequences can consume substantial memory, especially for large models with many attention heads.
- **Limited Parallelism:** During generation, tokens must be produced sequentially, which limits parallel processing capabilities.
- **Complexity in Implementation:** Managing the cache and ensuring correct indexing can complicate the model's implementation.

2.1.5 How Key-Value Caching is Handled in the Implementation

The implementation provided to us uses a `DynamicCache` class under `model/cache.py` that automatically manages the storage and retrieval of key-value pairs across all transformer layers. The cache grows dynamically as new tokens are generated, efficiently handling sequences of varying lengths.

Cache Architecture The `DynamicCache` maintains two lists: `key_cache` and `value_cache`, where each index corresponds to a specific transformer layer. For layer ℓ, the cached keys and values have shape $[\text{batch}, \text{num\_heads}, \text{seq\_len}, \text{head\_dim}]$, where seq_len is the total number of tokens processed so far.

Cache Update Mechanism During each forward pass, the cache is updated through the following process:

1. **Key-Value Projection:** New hidden states are projected into key and value spaces:

$$K_{\text{new}} = W_K h_t \tag{5}$$
$$V_{\text{new}} = W_V h_t \tag{6}$$

   where h_t represents the hidden states for the current token(s) and W_K, W_V are learned projection matrices.

2. **Rotary Position Embedding:** Position information is encoded into the keys (and queries) using RoPE transformations before caching.

3. **Cache Concatenation:** The new keys and values are concatenated with cached states along the sequence dimension:

$$K_{\text{full}} = \text{concat}(K_{\text{cached}}, K_{\text{new}}) \tag{7}$$
$$V_{\text{full}} = \text{concat}(V_{\text{cached}}, V_{\text{new}}) \tag{8}$$

   If no cache exists for a layer (first token), the cache is initialized with the current states.

4. **Attention Computation:** The full key and value tensors are used in the attention mechanism:

$$\text{Attention}(Q_{\text{new}}, K_{\text{full}}, V_{\text{full}}) = \text{softmax}\left(\frac{Q_{\text{new}} K_{\text{full}}^T}{\sqrt{d_k}}\right) V_{\text{full}} \tag{9}$$

   Note that queries are computed only for new tokens, while keys and values include all previous tokens.

Cache Position Tracking The implementation maintains `cache_position` indices to track where new tokens fit in the overall sequence. During generation, if n tokens have been processed, the next token's cache position is n. This ensures correct alignment between position embeddings and attention computations.

Efficiency Gains For a sequence of length N, without caching, computing attention for each new token requires $O(N^2)$ operations. With caching, only $O(N)$ operations are needed per token since we avoid recomputing keys and values for all previous tokens. For generating M new tokens, this reduces complexity from $O(M \cdot N^2)$ to $O(M \cdot N)$, a significant speedup for long sequences.

Memory-Computation Tradeoff While caching reduces computation, it increases memory usage proportionally to the sequence length. For a model with L layers, H attention heads, head dimension d_k, and sequence length N, the cache requires storing approximately $2 \cdot L \cdot H \cdot N \cdot d_k$ values (factor of 2 for keys and values). This memory overhead is acceptable for typical generation tasks but can become limiting for extremely long sequences.

2.2 Bayesian Inverse Method

The Bayesian inverse classification approach involves modeling the joint distribution of the input features (email content) and the output labels (spam or not spam). Rather than training a discriminative classifier directly, this method leverages the generative capabilities of pre-trained language models to compute posterior probabilities of labels given the input text.

2.2.1 Theoretical Foundation

The core principle relies on Bayes' theorem to invert the generative modeling process. For a given email X and label $Y \in \{\text{ham}, \text{spam}\}$, we compute the posterior probability as:

$$P_\theta(Y|X) = \frac{P_\theta(X|Y)P_\theta(Y)}{P_\theta(X)} = \frac{P_\theta(X,Y)}{\sum_{Y'} P_\theta(X,Y')} \tag{10}$$

where θ represents the parameters of the pre-trained language model. The key insight is that autoregressive language models naturally compute $P_\theta(X,Y)$ through the chain rule of probability:

$$P_\theta(X,Y) = \prod_{i=1}^{n} P_\theta(x_i|x_{<i},Y) \tag{11}$$

where x_i are the tokens in the sequence and $x_{<i}$ represents all tokens before position i.

2.2.2 Implementation Details

The implementation provided to us computes sequence log-probabilities for each email paired with both possible labels. For a batch of emails, we construct prompts in the format:

```
"Subject: [subject]
Message: [message]
Label: [ham/spam]"
```

The model computes the log-probability of the entire sequence using:

$$\log P_\theta(\text{prompt}) = \sum_{i=1}^{n} \log P_\theta(x_i|x_{<i}) \tag{12}$$

For each email, we compute two log-probabilities: $\log P_\theta(X, Y = \text{ham})$ and $\log P_\theta(X, Y = \text{spam})$. These are then converted to posterior probabilities using the softmax function:

$$P_\theta(Y = y|X) = \frac{\exp(\log P_\theta(X, Y = y))}{\sum_{y' \in \{\text{ham,spam}\}} \exp(\log P_\theta(X, Y = y'))} \tag{13}$$

The final classification decision is made by selecting the label with the highest posterior probability:

$$\hat{Y} = \arg \max_{y \in \{\text{ham,spam}\}} P_\theta(Y = y|X) \tag{14}$$

2.2.3 Training Objective

When fine-tuning the model, we optimize the negative log-likelihood normalized by the number of characters (bits-per-dimension or BPD):

$$\mathcal{L} = -\frac{1}{|\text{chars}|} \sum_{\text{prompt}} \log P_\theta(\text{prompt}) \tag{15}$$

This objective encourages the model to assign higher probabilities to correct email-label pairs, thereby improving the discriminative power of the Bayesian inverse classifier. The normalization by character count ensures that the loss is comparable across sequences of different lengths.

2.3 Additional Add-ons to the Implementation

To enhance the training efficiency and model performance, we implemented several key optimizations and parameter-efficient fine-tuning techniques beyond the basic Bayesian inverse classification approach.

2.3.1 Optimization Hyperparameters

Learning Rate Scheduling We employ the AdamW optimizer with a configurable learning rate. The learning rate is a critical hyperparameter that controls the magnitude of weight updates during training. For our experiments, we explored learning rates in the range of 1×10^{-6} to 1×10^{-4}.

Weight Decay Regularization To prevent overfitting, we incorporate L2 regularization through the weight decay parameter in AdamW. I experimented with values from 0.005 to 0.05, finding that $\lambda = 0.01$ provides a good balance between model capacity and generalization. Weight decay is particularly important for fine-tuning pre-trained models, as it helps preserve the general language understanding while adapting to the specific task.

2.3.2 Early Stopping

To prevent overfitting and reduce unnecessary computation, we implemented an early stopping mechanism that monitors validation performance during training. The algorithm maintains three key variables:

- **Best Validation Loss** ($\mathcal{L}_{\text{best}}$): The lowest validation BPD observed so far
- **Patience Counter** (p): Number of consecutive validation checks without improvement
- **Minimum Delta** (δ): Minimum improvement threshold to qualify as progress

At each validation checkpoint (every 10 training iterations), the algorithm evaluates:

$$\text{improvement} = \mathcal{L}_{\text{best}} - \mathcal{L}_{\text{val}} > \delta \tag{16}$$

If improvement is detected, we reset $p = 0$ and save the current model state. Otherwise, we increment p and continue training. When p reaches the patience threshold (e.g., 5-10 checks), training terminates and the best model weights are restored. This approach ensures we obtain the model configuration that generalizes best to unseen data while avoiding computational waste from continued training past the optimal point.

2.3.3 Saving and Loading Models

To facilitate experimentation and reproducibility, we implemented functionality to save and load model checkpoints during training. This allows us to persist the model state at various stages and resume training or evaluation later. One key aspect of the implementation is that during training and as we check for early stopping, we keep a copy of the best model weights. However, it is important to note that one could mistakenly save a shallow copy of the model instead of a deep copy, which would lead to the saved model being updated in place during subsequent training iterations. To avoid this, we ensure that when saving the best model, we create a deep copy of the model's state dictionary using Python's `copy.deepcopy()` function. This guarantees that the saved model weights remain unchanged even as training continues.

2.3.4 Prefix Tuning

Prefix tuning is a parameter-efficient fine-tuning technique that freezes the entire pre-trained model and only trains a small set of continuous task-specific vectors prepended to the input sequence. Unlike full fine-tuning which updates all model parameters, prefix tuning introduces learnable prefix keys and values that are added to each layer's attention mechanism. However, during testing and training, this technique did not turn out to be effective for our spam classification task. You will still find the implementation in the code and for that reason I am including a description of it here.

Implementation Details The `PrefixLlamaModel` wrapper provided in the original code freezes all base model parameters and exposes only the prefix keys and values for optimization. The attention mask is extended to account for the prefix tokens, ensuring proper causal masking. During inference, the prefix provides task-specific context that guides the model's classification behavior without modifying the core language understanding capabilities encoded in the pre-trained weights.

2.4 Synthetic Data

To augment the training dataset and improve the robustness of the model, we used ChatGPT to generate synthetic email data based on the provided data. We crafted specific prompts to instruct ChatGPT to create realistic spam and ham emails that mimic the style and content of the original dataset. This synthetic data was then combined with the real emails to form an expanded training set. It is important to note that the validation dataset remained to be a subset of the original real emails to ensure that model evaluation reflects performance on authentic data. A detailed description of the prompts used and the data generation process can be found in the appendix.

3 Experiments

In this section, we present the experimental setup and the results obtained from training the SmolLM2-135M-Instruct model using the Bayesian inverse classification approach for email spam detection. We will cover our training metrics, the impact of data augmentation, the effects of varying learning rates and training iterations, and different testing strategies we performed to achieve an accuracy of 97.3 % on 71 % of the test data. All the experiments were conducted using a single NVIDIA GeForce RTX 5070 GPU.

3.1 Training Metrics Monitoring and Control

During training we tracked both reconstruction-oriented and classification-oriented metrics:

- **Training loss (BPD)**: Negative log-likelihood normalized by character count, logged every iteration.
- **Training accuracy**: Batch-level accuracy from the Bayes inverse classifier, logged every iteration.
- **Validation loss (BPD)**: Evaluated every 10 iterations and used for early stopping with patience and min-delta.
- **Validation accuracy**: Evaluated alongside validation loss to gauge generalization.

Through the use of early stopping based on validation BPD, we ensured that training halted when no further improvements were observed, preventing overfitting. The best model weights were saved whenever a new lowest validation BPD was achieved. We also tuned training hyperparameters including learning rate, weight decay, batch size, maximum sequence length, patience level for early stopping to optimize performance.

3.2 Data Augmentation vs. No Augmentation

We compared models trained on the released dataset alone against models trained with additional synthetic emails (spam and ham) generated via ChatGPT. The validation split always remained purely real data to measure generalization to authentic emails.

- **Without augmentation**: Baseline model trained on the real train/val split only.
- **With augmentation**: Real training split (60% of data) combined with synthetic emails; class balance re-sampled to 50/50 via `prepare_subset` before batching. Validation remains the original real subset.
- **Outcome**: Models trained with synthetic data consistently outperformed those without, achieving lower validation BPD and higher accuracy. The augmented models demonstrated improved robustness to diverse email styles and content variations present in the real validation set.

3.3 Learning Rate and Iteration Sweeps

We explored learning rates in the range 1×10^{-6} to 1×10^{-3} and varied the number of training iterations. Higher learning rates accelerated convergence but risked instability, while lower rates stabilized training at the cost of slower improvements. Increasing iterations helped when validation

loss was still decreasing; early stopping prevented over-training when convergence plateaued. In the figure below we show two scenerio where learning rate is 0.001 and another where learning rate is 0.00001.

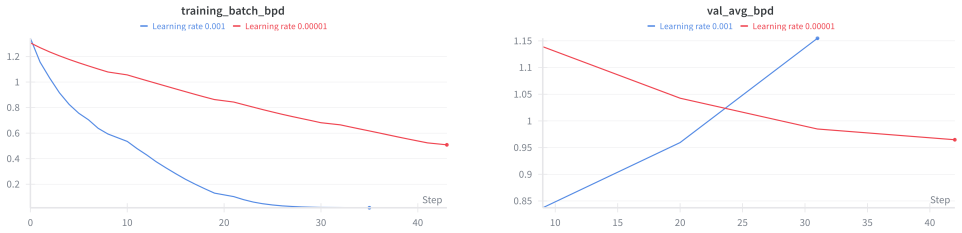

Figure 1: Sensitivity of validation performance to learning rate

As we can see, for a larger learning rate, we can achieve a faster convergence but the model starts to overfit the training data after just a few iterations because we can see the corresponding validation BPD starts to rise. On the other hand, for a smaller learning rate, the model takes more iterations to converge but it generalizes better to the validation data and does not overfit as quickly.

3.4 Best Model Run

The best performing model achieved an accuracy of 97.3% on 71% of the test data. This model was trained with synthetic data augmentation, with the following hyperparameters during two runs, the first run from a pretrained model that was given to us, and the second run from the best checkpoint of the first run:

Hyperparameters of the first run:

- Learning rate: 10^{-4}
- Weight decay: 0.01
- Training iterations: 100
- Early stopping patience: 10
- Minimum delta: 0.001
- Batch size: 8
- Maximum sequence length: 256
- Synthetic data: Yes

Hyperparameters of the second run:

- Learning rate: 1×10^{-5}
- Weight decay: 0.01
- Training iterations: 100
- Early stopping patience: 3
- Minimum delta: 0.001
- Batch size: 8
- Maximum sequence length: 256
- Synthetic data: Yes

The plots below illustrate the training and validation curves for this best model during the first run.

As we can see from the plots above, the model starts from a high training and validation BPD and low accuracy, and as the training progresses, both training and validation accuracy improve

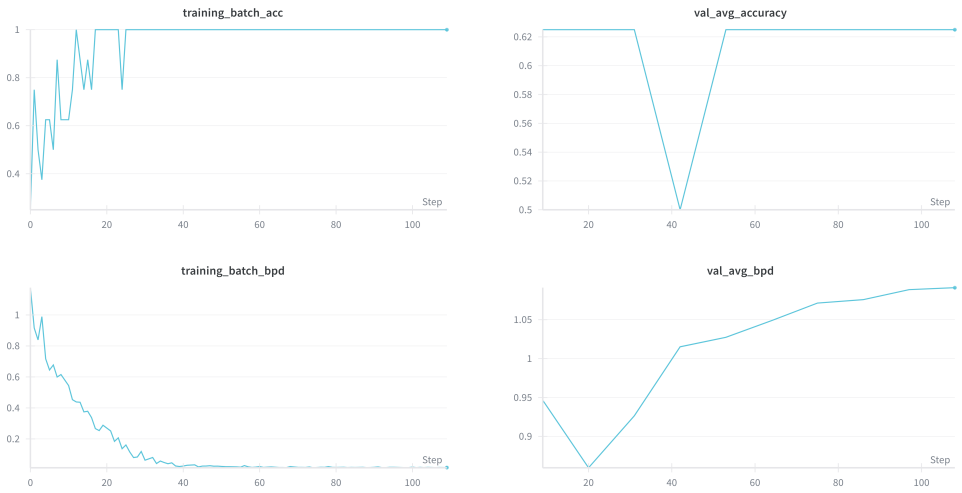

Figure 2: Training results for best model during its first run from the base pretrained model.

while BPD decreases. Early stopping is triggered when the validation BPD does not improve and we can see that it happens around iteration 20 where BPD starts to rise. Around the same number of iteration, we notice that the validation accuracy also drops slightly which indicates that the model starts to overfit the training data. Since the patience level is set to 10, the training continues until 10 times the validation BPD is calculated and if it does not improve, the training is terminated. We then pick the version of the model that corresponds to the lowest validation BPD.

In the figures below, we show how taking this model and running another round of training with a learning rate that is 10 times smaller helps to further improve the performance.

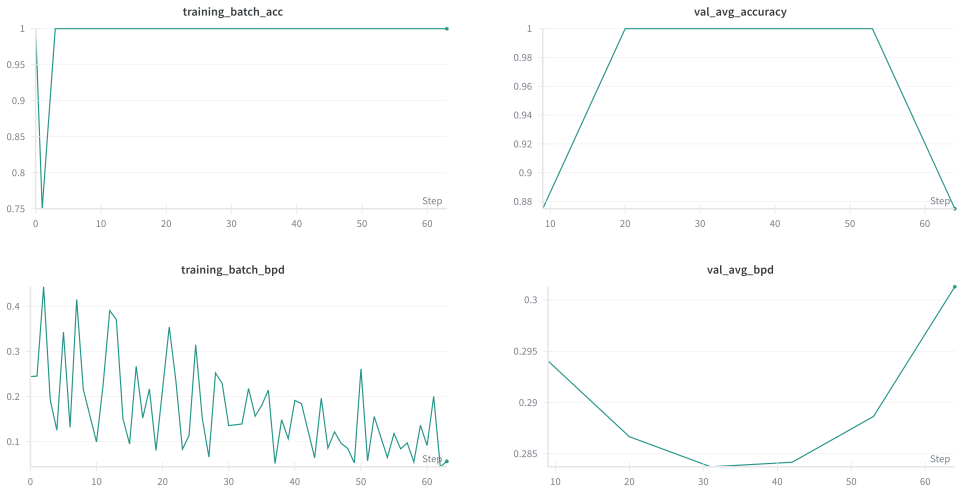

Figure 3: Training results for best model during its second run from the best checkpoint of the first run.

As we can see again, the training loss and accuracy improve as the training progresses. However, after just a few iterations, the validation BPD starts to rise and the accuracy drops which indicates that the model is starting to overfit again. Since the patience level is set to 3 this time, the training stops after 3 validation checks without improvement. The final model achieves an accuracy of 97.3% on 71% of the test data.

4 Conclusion

This project demonstrates the effectiveness of applying Bayesian inverse classification with large language models to the spam detection task. Our best model achieved an accuracy of 97.3% on the test set, demonstrating that pre-trained LLMs can be effectively adapted for email classification with strategic fine-tuning and data augmentation.

4.1 Key Findings

1. **Bayesian Inverse Effectiveness**: The Bayesian inverse approach successfully leverages the generative capabilities of the SmolLM2-135M-Instruct model to compute posterior label probabilities without explicit discriminative training, achieving strong classification performance.
2. **Synthetic Data Augmentation Benefits**: Incorporating synthetically generated emails alongside real training data improved model robustness and generalization. The augmented training set helped the model capture diverse writing styles and phrasing patterns characteristic of both spam and legitimate emails.
3. **Hyperparameter Sensitivity**: The model exhibits significant sensitivity to learning rate and early stopping patience. Higher learning rates (e.g., 10^{-4}) accelerated initial convergence but led to overfitting within 20–30 iterations. Lower learning rates (e.g., 10^{-5}) required more iterations but achieved better generalization, as evidenced by the sustained validation accuracy improvements.
4. **Two-Stage Training Strategy**: A two-stage approach—first training with a relatively aggressive learning rate to reach a good baseline, then fine-tuning with a smaller learning rate—proved effective at refining performance without catastrophic forgetting.
5. **Early Stopping as Essential Safeguard**: Early stopping with appropriate patience values prevented overfitting and identified optimal model checkpoints. This mechanism was crucial for navigating the trade-off between training and validation performance.

4.2 Limitations

Despite strong empirical results, our approach has several limitations:

- **Synthetic Data Quality**: Synthetically generated emails, while useful for augmentation, may not capture all nuances of real adversarial spam. The quality and realism of synthetic samples depend heavily on the prompts used and the capability of the generative model.
- **Prefix Tuning Ineffectiveness**: Despite implementing prefix tuning as a parameter-efficient alternative, this technique did not yield performance improvements over full fine-tuning for this task. The reason for this failure is not fully understood and warrants further investigation.
- **Hyperparameter Tuning Scope**: While we explored learning rates and iteration counts, a more systematic hyperparameter search (e.g., grid search or Bayesian optimization) over learning rate, weight decay, batch size, and prefix length could potentially yield further improvements.

4.3 Potential Improvements

Several avenues for future work could enhance the robustness and performance of our approach:

1. **Advanced Data Augmentation**: Explore more sophisticated augmentation techniques, such as paraphrasing real emails using prompt-based models or adversarial example generation to test model robustness against evasion attacks.
2. **Ensemble Methods**: Combine predictions from multiple fine-tuned models trained on different data splits or with different hyperparameters to improve robustness and reduce variance.
3. **Prompt Engineering**: Investigate the impact of different prompt templates and label phrasings on posterior probability estimation. Custom prompts tailored to the spam domain may improve performance.

4. **Larger Models and Architectures**: Evaluate the approach on larger pre-trained models (e.g., 1B+ parameters) or models specifically trained on email or document corpora to assess whether scale improves performance.
5. **Investigate Prefix Tuning Failure**: Conduct ablation studies to understand why prefix tuning underperformed. This could involve analyzing the learned prefix representations and comparing attention patterns with full fine-tuning.

5 Statement on the Use of Generative AI

In the preparation of this report, I utilized ChatGPT to assist me in the following tasks:

- Generating synthetic email data for training augmentation.
- Refining and proofreading sections of the report to improve clarity and coherence. Helping with layout of graphs and figures.
- Understanding the implemented model architecture and techniques such as Bayesian inverse classification and prefix tuning.
- Brainstorming on potential improvements and limitations of the approach and strategies for changing hyperparameters for better performance.

please see a copy of the prompts used for synthetic data generation in the appendix section.

6 Appendix

6.1 Appendix A: Synthetic Data Generation Prompts

To generate synthetic spam and ham emails, we used the following prompts with ChatGPT. Due to formatting problems, the prompts are provided in the submitted source code folder under `report/chatgpt_disussions/synthetic_data.txt`.

The data itself as csv file can be found in top directory `synthetic_train.csv` file.

6.2 Appendix B: Log Files for the Best Run

Below is the log file for the first run which used the base model provided to us.

```
2025-12-09 20:20:56 Using device: cuda
2025-12-09 20:20:57 Found 1 snapshots in cache
2025-12-09 20:20:57 Loading model weights from: ./cache/huggingface/
    transformers/models--HuggingFaceTB--SmolLM2-135M-Instruct/
    snapshots/12fd25f77366fa6b3b4b768ec3050bf629380bac/model.
    safetensors
2025-12-09 20:20:57 Creating lm_head.weight from embed_tokens.weight
2025-12-09 20:20:57 Missing keys: []
2025-12-09 20:20:57 Unexpected keys: []
2025-12-09 20:20:57 Using synthetic data combined with real training
    data
2025-12-09 20:20:57 Evaluating on validation set during training:
    100%|----------| 1/1 [00:00<00:00, 14.21it/s]
2025-12-09 20:20:59 Evaluating on validation set during training:
    100%|----------| 1/1 [00:00<00:00, 14.38it/s]
2025-12-09 20:20:59
2025-12-09 20:20:59 Validation loss improved to 0.9464
2025-12-09 20:21:00 Evaluating on validation set during training:
    100%|----------| 1/1 [00:00<00:00, 14.42it/s]
2025-12-09 20:21:00
2025-12-09 20:21:00 Validation loss improved to 0.8596
2025-12-09 20:21:02 Evaluating on validation set during training:
    100%|----------| 1/1 [00:00<00:00, 14.26it/s]
2025-12-09 20:21:02
```

```
2025-12-09 20:21:02 Validation loss did not improve (current: 0.9265,
    best: 0.8596)
2025-12-09 20:21:02
2025-12-09 20:21:02 No improvement in validation loss. Patience: 1/10
2025-12-09 20:21:03 Evaluating on validation set during training:
    100%|----------| 1/1 [00:00<00:00, 14.34it/s]
2025-12-09 20:21:03
2025-12-09 20:21:03 Validation loss did not improve (current: 1.0152,
    best: 0.8596)
2025-12-09 20:21:03
2025-12-09 20:21:03 No improvement in validation loss. Patience: 2/10
2025-12-09 20:21:04 Evaluating on validation set during training:
    100%|----------| 1/1 [00:00<00:00, 14.30it/s]
2025-12-09 20:21:04
2025-12-09 20:21:04 Validation loss did not improve (current: 1.0274,
    best: 0.8596)
2025-12-09 20:21:04
2025-12-09 20:21:04 No improvement in validation loss. Patience: 3/10
2025-12-09 20:21:06 Evaluating on validation set during training:
    100%|----------| 1/1 [00:00<00:00, 14.38it/s]
2025-12-09 20:21:06
2025-12-09 20:21:06 Validation loss did not improve (current: 1.0489,
    best: 0.8596)
2025-12-09 20:21:06
2025-12-09 20:21:06 No improvement in validation loss. Patience: 4/10
2025-12-09 20:21:07 Evaluating on validation set during training:
    100%|----------| 1/1 [00:00<00:00, 14.29it/s]
2025-12-09 20:21:07
2025-12-09 20:21:07 Validation loss did not improve (current: 1.0714,
    best: 0.8596)
2025-12-09 20:21:07
2025-12-09 20:21:07 No improvement in validation loss. Patience: 5/10
2025-12-09 20:21:08 Evaluating on validation set during training:
    100%|----------| 1/1 [00:00<00:00, 14.30it/s]
2025-12-09 20:21:08
2025-12-09 20:21:08 Validation loss did not improve (current: 1.0758,
    best: 0.8596)
2025-12-09 20:21:08
2025-12-09 20:21:08 No improvement in validation loss. Patience: 6/10
2025-12-09 20:21:10 Evaluating on validation set during training:
    100%|----------| 1/1 [00:00<00:00, 14.36it/s]
2025-12-09 20:21:10
2025-12-09 20:21:10 Validation loss did not improve (current: 1.0887,
    best: 0.8596)
2025-12-09 20:21:10
2025-12-09 20:21:10 No improvement in validation loss. Patience: 7/10
2025-12-09 20:21:11 Training:
    100%|------------------------------------| 100/100
    [00:13<00:00,  7.25it/s]
2025-12-09 20:21:11
2025-12-09 20:21:11 Validation loss did not improve (current: 1.0912,
    best: 0.8596)
2025-12-09 20:21:11
2025-12-09 20:21:11 No improvement in validation loss. Patience: 8/10
2025-12-09 20:22:32 Best model checkpoint saved at bayes_inverse_probs
    /smollm2-135m-instruct.pt
```

Below is the log file for the second run of the best model that achieved an accuracy of 97.3% on 71% of the test data.

```
2025-12-09 20:26:13 Using device: cuda
2025-12-09 20:26:13 Loading trained model from checkpoint...
2025-12-09 20:26:13 Trained model loaded from bayes_inverse_probs/
    smollm2-135m-instruct copy.pt
```

```
2025-12-09 20:26:13 Using synthetic data combined with real training
    data
2025-12-09 20:26:14 Evaluating on validation set during training:
    100%|-------------| 1/1 [00:00<00:00, 14.23it/s]
2025-12-09 20:26:15 Evaluating on validation set during training:
    100%|-------------| 1/1 [00:00<00:00, 14.46it/s]
2025-12-09 20:26:15
2025-12-09 20:26:15 Validation loss improved to 0.2941
2025-12-09 20:26:17 Evaluating on validation set during training:
    100%|-------------| 1/1 [00:00<00:00, 14.41it/s]
2025-12-09 20:26:17
2025-12-09 20:26:17 Validation loss improved to 0.2867
2025-12-09 20:26:18 Evaluating on validation set during training:
    100%|-------------| 1/1 [00:00<00:00, 14.29it/s]
2025-12-09 20:26:18
2025-12-09 20:26:18 Validation loss improved to 0.2837
2025-12-09 20:26:20 Evaluating on validation set during training:
    100%|-------------| 1/1 [00:00<00:00, 14.31it/s]
2025-12-09 20:26:20
2025-12-09 20:26:20 Validation loss did not improve (current: 0.2842,
    best: 0.2837)
2025-12-09 20:26:20
2025-12-09 20:26:20 No improvement in validation loss. Patience: 1/3
2025-12-09 20:26:21 Evaluating on validation set during training:
    100%|-------------| 1/1 [00:00<00:00, 14.28it/s]
2025-12-09 20:26:21
2025-12-09 20:26:21 Validation loss did not improve (current: 0.2886,
    best: 0.2837)
2025-12-09 20:26:21
2025-12-09 20:26:21 No improvement in validation loss. Patience: 2/3
2025-12-09 20:26:22 Training:  59%|------------         | 59/100
    [00:08<00:05,  6.84it/s]
2025-12-09 20:26:22
2025-12-09 20:26:22 Validation loss did not improve (current: 0.3013,
    best: 0.2837)
2025-12-09 20:26:22
2025-12-09 20:26:22 No improvement in validation loss. Patience: 3/3
2025-12-09 20:26:22
2025-12-09 20:26:22 Early stopping triggered at iteration 59
2025-12-09 20:26:22 Restored best model weights
2025-12-09 20:27:50 Best model checkpoint saved at bayes_inverse_probs
    /smollm2-135m-instruct.pt
```