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 pretraiend 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, providing insights into the potential of LLMs in cybersecurity applications.

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 (e.g., $L = 30$ for SmolLM2-135M). Each layer contains two sub-components connected via residual connections:

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 Our Implementation

Our implementation uses a `DynamicCache` class 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

Our implementation 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.

Architecture For a model with L layers, we introduce learnable prefix parameters:

$$P_K \in \mathbb{R}^{L \times H \times p \times d_k} \quad \text{(prefix keys)} \tag{17}$$

$$P_V \in \mathbb{R}^{L \times H \times p \times d_k} \quad \text{(prefix values)} \tag{18}$$

where H is the number of key-value heads, p is the prefix length (typically 10-50), and d_k is the head dimension. These parameters are initialized using a normal distribution with mean 0 and standard deviation 0.02.

Integration with Attention During forward pass, the prefix is converted into a cache and prepended to the attention computation. For layer ℓ, the modified attention becomes:

$$\text{Attention}_\ell(Q, K, V) = \text{softmax}\left(\frac{Q \cdot [P_K^\ell; K]^T}{\sqrt{d_k}}\right) \cdot [P_V^\ell; V] \tag{19}$$

where $[P_K^\ell; K]$ denotes concatenation of the prefix keys with the computed keys from the input. The model effectively attends over both the prefix (task-specific context) and the actual input tokens.

Efficiency Benefits Prefix tuning dramatically reduces the number of trainable parameters. For SmolLM2-135M with 135 million parameters, using a prefix length of 10 requires training only:

$$|\theta_{\text{prefix}}| = 2 \cdot L \cdot H \cdot p \cdot d_k \tag{20}$$

For $L = 30$ layers, $H = 4$ key-value heads, $p = 10$, and $d_k = 128$, this amounts to approximately 307,200 trainable parameters—less than 0.3% of the full model. This reduction significantly decreases memory requirements, training time, and the risk of catastrophic forgetting while maintaining competitive performance on the downstream task.

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.

3 Experiments

Here you can add the main content of your report.

3.1 Subsection

Additional details can go here.

4 Conclusion

Summary of your findings.