Few-Shot Learning for Spam Detection with Large Language Models

Siddarth Chilukuri
University of British Columbia
Vancouver, BC

1 Introduction

In this project, I investigate the effectiveness of few-shot learning for spam detection using the SmolLM2-135M-Instruct model. My approach centers on Bayesian Inverse Classification, where I model the likelihood $P(X|Y)$ to derive the posterior probability of a label given an email. I compare this method against standard baselines like zero-shot classification and naive prompting. The results show that while the baselines struggle with a 50% accuracy, fine-tuning on just 20 samples significantly boosts performance to 91.01% on the test set. Furthermore, using an ensemble of models improves this to 92.13%. Finally, I analyze the Key-Value (KV) cache mechanism, which is crucial for efficient inference in decoder-only transformers.

2 Model and Methods

2.1 Bayesian Inverse Classification

My main classification strategy is the Bayesian Inverse method. The goal is to calculate the posterior probability of a label Y_{label} (Spam or Ham) given an input email $X_{\leq i}$. Using Bayes' theorem:

$$P_\theta(Y_{\text{label}}|X_{\leq i}) = \frac{P_\theta(X_{\leq i}|Y_{\text{label}})P_\theta(Y_{\text{label}})}{\sum_{Y'} P_\theta(X_{\leq i}|Y')P_\theta(Y')} \tag{1}$$

where $P_\theta(X_{\leq i}|Y_{\text{label}})$ is the likelihood of the email text generated by the LLM when conditioned on the label, and $P_\theta(Y_{\text{label}})$ is the prior probability of the label (assumed uniform in my experiments). This lets me turn a generative task (predicting the next token) into a classification task.

2.2 Decoder-Only Transformers and KV Cache

SmolLM2 is a decoder-only transformer. Unlike encoder models (e.g., BERT) which use bidirectional attention to see the full context, decoder-only models use **causal masking** to prevent attending to future tokens. This autoregressive property allows them to generate text token-by-token, predicting x_t based solely on $x_{<t}$.

2.2.1 The KV Cache Mechanism

Autoregressive generation requires attention over all previous tokens. Recomputing Key (K) and Value (V) vectors at every step leads to $O(N^2)$ complexity. The **KV Cache** stores past K and V states, allowing the model to compute only the current token's vectors and retrieve past ones. This reduces complexity to $O(N)$ (or $O(1)$ per token).

2.2.2 Implementation and Drawbacks

The KV cache is implemented using two main parts. First, `model/cache.py` has the `DynamicCache` class to store the key-value pairs. Second, the inference code in `utils/sample.py` handles the

Algorithm 1 Bayesian Inverse Classification (Inference)

Input: Email X, Model θ, Labels $\mathcal{Y} = \{\text{Spam}, \text{Ham}\}$
Tokenizer: $x_{1:T} \leftarrow \text{Tokenize}(X)$
for each label $Y \in \mathcal{Y}$ **do**
  Construct prompt $X_Y = \text{Template}(x_{1:T}, Y)$
  // Sum log-probs of email tokens conditioned on label
  $\log P(X|Y) = \sum_{t=1}^{T} \log P_\theta(x_t | x_{<t}, \text{Prompt}_Y)$
  Add log-prior: $S_Y = \log P(X|Y) + \log P(Y)$
end for
// Normalize using LogSumExp for numerical stability
$Z = \text{LogSumExp}(\{S_Y\}_{Y \in \mathcal{Y}})$
for each label $Y \in \mathcal{Y}$ **do**
  $P(Y|X) = \exp(S_Y - Z)$
end for
Output: Posterior $P(Y|X)$

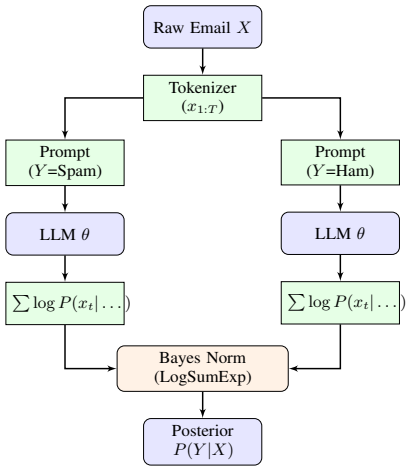

Figure 1: Computation Graph. The email is tokenized and inserted into label-specific templates. The LLM computes the aggregate log-likelihood of the email tokens. These scores are normalized to yield the posterior.

caching. Within the `sample` function, the `past_key_values` variable is initialized to `None` and passed to the model. On the first iteration, the model instantiates the cache and returns it. In subsequent iterations, this populated cache is passed back to the model, allowing it to process only the single new token (`generated[:, -1:]`) instead of the full history.

While efficient, this mechanism has drawbacks:

- **Memory Growth:** Storage grows linearly with sequence length ($L \times N \times d_{head}$), potentially causing OOM errors.
- **Per-Layer Overhead:** Each of the model's layers maintains its own cache, multiplying the memory footprint.
- **Numerical Drift:** For extremely long sequences, accumulated float16 precision errors can degrade attention quality.
- **No Reuse:** Changing the prompt prefix invalidates the entire cache.

2.3 Ensemble Strategy

I average posterior probabilities from K models trained with different seeds:

$$P_{ensemble}(Y|X) = \frac{1}{K} \sum_{k=1}^{K} P_{\theta_k}(Y|X) \tag{2}$$

This is basically **Bayesian Model Averaging**, where I average over the model weights. It helps reduce variance (theoretically by $1/K$) and smooths out errors.

3 Experiments

3.1 Experimental Setup

I used `SmolLM2-135M-Instruct` [6] on a Tesla T4 GPU. Training used AdamW optimizer, batch size 8, learning rate 1e-5, and 80 iterations (approx. 10 epochs) with no warmup.

3.2 Baseline Results

3.2.1 Qualitative Evaluation

Qualitative evaluation (Table 1) shows the model answers general questions well but hallucinates terms like "Hammels" for ham emails, indicating a lack of domain knowledge.

Table 1: Qualitative Evaluation of Chatbot Responses

Prompt	Response Summary
"What is gravity?"	Explained gravity as a fundamental force, referencing Einstein's general relativity.
"How do I write a Python function to sort a list?"	Generated a correct Python implementation of bubble sort.
"What is the difference between spam and ham emails?"	Attempted to explain but hallucinated the term "Hammels" for ham emails.

3.2.2 Quantitative Baselines

I evaluated zero-shot and naive prompting performance:

- **Zero-Shot:** 50% accuracy. The model exhibited extreme bias, predicting **Spam** for 100% of the samples.
- **Naive Prompting:** 50% accuracy. The model became uncertain, assigning exactly 0.5 probability to both classes, likely due to the complex prompt confusing the small model.

3.3 Full Fine-Tuning Results

I performed full fine-tuning using `examples/bayes_inverse_full_finetune.sh` with a batch size of 8 and learning rate of 1e-5. This yielded a dramatic improvement, with the best model achieving **91.01%** test accuracy on Kaggle. However, performance was highly variable across different random seeds, with most runs yielding accuracies between 60% and 80%.

3.3.1 Ablation Study: Training Iterations

I investigated the effect of training duration on performance. As shown in Table 2, **80 iterations** proved to be the optimal point.

Table 2: Effect of Training Iterations on Accuracy

Iterations	Validation Accuracy	Kaggle Test Accuracy
60	$\sim$85%	65.0%
80	**95%**	**91.01%**
100	$\sim$85%	76.7%
120	$\sim$85%	76.4%

Beyond 80 iterations, I observed clear signs of **overfitting**: while validation accuracy on the 20 samples remained high, generalization to the unseen test set dropped significantly.

3.4 Ensemble Results

- **3-Model Ensemble:** Averaging 3 seeds (0, 42, 123) improved accuracy to **92.13%** (+1.12%).
- **5-Model Ensemble:** Slightly lower at 91.99%, indicating diminishing returns.

I selected the 3-model ensemble as my final submission.

4 Limitations

There are a few limitations to my approach. First, fine-tuning on only 20 samples has a high risk of **overfitting**, which I saw in my ablation study. Second, the model has a lot of **pre-training bias**; the zero-shot baseline predicted **Spam** 100% of the time, so the base model is clearly skewed. Third, I didn't use parameter-efficient methods like LoRA [1] because the model is small (135M), so full fine-tuning was easy to run and potentially works better.

5 Conclusion

I showed that while small LLMs struggle with zero-shot classification, **few-shot fine-tuning** works really well. With 20 examples, I achieved **91.01%** accuracy, improved to **92.13%** via ensembling. My ablation study highlights the importance of early stopping to prevent overfitting. Future work could explore LoRA for parameter efficiency.

Acknowledgments and Disclosure of Funding

This project was completed as part of the CPEN 455 course at the University of British Columbia. I utilized the provided course codebase and the SmolLM2 model from Hugging Face.

References

[1] Hu, E.J., et al. (2021) LoRA: Low-Rank Adaptation of Large Language Models. *arXiv preprint arXiv:2106.09685*.

[2] Li, X.L. & Liang, P. (2021) Prefix-Tuning: Optimizing Continuous Prompts for Generation. *arXiv preprint arXiv:2101.00190*.

[3] Touvron, H., et al. (2023) Llama: Open and Efficient Foundation Language Models. *arXiv preprint arXiv:2302.13971*.

[4] CPEN 455 Course Staff. (2025) CPEN 455 Course Project Dataset. University of British Columbia.

[5] Kaggle. (2025) CPEN 455 Spring 2025 Spam Detection Competition. `https://www.kaggle.com/t/7bd983ca8e064c9aa7f13cf1ecbdbf23`.

[6] Hugging Face. (2024) SmolLM2-135M-Instruct. `https://huggingface.co/HuggingFaceTB/SmolLM2-135M-Instruct`.