CPEN 455 Final Project Report

Anirudh Devanand
University of British Columbia
adevanan@ece.ubc.ca

Abstract

This project explores spam vs. ham email classification using a Bayes inverse classifier on top of the SmolLM2 model. The method is evaluated in three methods (zero shot, naive prompting, and full finetuning) and improved with data synthesis, weight decay and gradient clipping. Qualitatively, the base model behaves like a generic chatbot and is not a reliable spam detector. Quantitatively, the baseline zero shot and naive prompting variants are close to random guessing, while full finetuning substantially improves accuracy on the labelled train/validation split. Data synthesis along with an increased learning rate and a small weight decay and gradient clipping achieve the best generalization on the hidden test set.

1 Reports

I followed the project instructions and evaluated three main variants of the provided Bayes inverse classifier:

- **Zero shot:** directly apply the Bayes inverse procedure with the pre-trained model.
- **Naive prompting:** prepend a short natural language instruction to each email but still use the pre-trained weights.
- **Full finetune:** run supervised finetuning on the labelled train/validation subset, then recompute probabilities.

On the labelled train/validation split, I obtained the following accuracies:

Method	Accuracy (train/val)
Zero shot	$= 37.500\%$
Naive prompting	$= 53.125\%$
Full finetune	$= 78.472\%$

Thus, without task specific training, the model behaves close to random for this binary classification problem. The full finetune setup is the one I treat as the main method/model in the rest of the report. All additional experiments (training size sweeps, data synthesis, ensembling, weight decay, and test set inference) are built on top of this finetuned model and its training pipeline.

2 Model

The core classifier and Bayes inverse logic were pre-existing starter code in `examples/bayes_inverse.py`.

We use SmolLM2-135M, a decoder only transformer used as a spam classifier. For each email, the Bayes inverse classifier constructs two prompts that contain the subject, body, and an explicit label

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

template. It then computes the log likelihood of each full prompt under the model and normalizes these two log probabilities into a binary distributions over ham vs. spam. Training is done to maximizing the log likelihood of the prompt with the correct label.

2.1 Label-aware data synthesis

To increase the effective amount and diversity of labelled data, I implemented a data augmentation procedure in a separate script (`generate_synthetic_emails.py`). The input is the labelled `train_val_subset.csv` file; the output is an augmented file `train_val_augmented.csv` with both original and synthetic rows.

Spam augmentation. For rows labelled as spam, the script modifies the subject and body using simple transformations that mimic real-world spam:

- **Phrase injection:** insert a spammy phrase (e.g., "Limited time offer just for you", "Earn extra money from home", "Your account will be closed soon") at the start, middle, or end of the subject or message.
- **Random uppercase:** randomly uppercase a subset of words to create phrases similar to spam subject lines.
- **Punctuation noise:** perturb exclamation marks and periods to change the overall tone without changing core semantics.

Ham augmentation. For ham emails, the goal is to preserve the semantic content while adding realistic noise:

- **Follow-up markers:** append tags such as "(follow up)" to the subject.
- **Polite closings:** append realistic closing phrases such as "Please let me know if you have any questions.", "Looking forward to your reply.", or "Thanks and best regards.".

Hyperparameters `-spam_factor` and `-ham_factor` control how many synthetic variants are created per original message. The augmented dataset keeps the same schema, so it can be used as a drop-in replacement for training.

2.2 Weight decay and gradient clipping

Initially, the full finetune model used AdamW optimizer with a very small learning rate and no weight decay or gradient clipping. This led to very high training/validation accuracy but noticeably lower accuracy on the hidden Kaggle test set, i.e., strong signs of overfitting.

To address this, I added a small weight decay term to the optimizer: `AdamW(model.parameters(), lr, weight_decay=...)`. In addition, I applied simple gradient clipping before each optimizer step (clipping the global norm of all gradients to a fixed threshold).This limits the impact of rare very large gradients and keeps the update magnitudes more stable. I further doubled the default learning rate. These simple changes gave the best generalization out of all methods I tried.

3 Decoder only transformers and KV cache

A decoder only transformer is a stack of self attention and feed forward blocks that predict the next token given all previous tokens. Each block applies *masked* self-attention so that token t can only attend to positions $1, \ldots, t$, followed by a position wise MLP. The SmolLM2 model used in this project is a decoder only model.

3.1 KV cache mechanism and why it is useful

In a naive implementation, computing $p_\theta(x_t \mid x_{1:t-1})$ for each new token would recompute all attention keys and values for all previous positions. The KV cache avoids this by storing, for each layer and previous time step, the attention keys and values and only computing the new query, key, and value for the current position t.

This reduces the cost of autoregressive generation from $O(T^2)$ per step to roughly $O(T)$ overall, because each new token reuses the cached computation for earlier positions instead of recomputing it from scratch. In practice this is critical for fast decoding and interactive chat style models.

3.2 Drawbacks of KV cache

The KV cache has the following drawbacks:

- **Memory overhead:** for long contexts, storing all keys and values for every layer and head can consume a large amount of GPU memory.
- **Complexity of implementation:** the forward pass has to handle both the initial "no cache" case and the incremental "reuse cache" case, which makes the code more complex and error prone.
- **Limited flexibility:** the cache assumes a simple left to right generation order. Inserting or editing tokens in the middle of the context usually requires recomputing the cache from that point on.

3.3 KV cache in the provided codebase

In the provided codebase, the KV cache is implemented inside the `LlamaModel` used by `examples/bayes_inverse.py` and `examples/chatbot_example.py`. The model forward method supports passing in a batch of tokenized prompts and, optionally, a set of cached key/value tensors. When the cache is present, the model only computes the attention projections for the new tokens and concatenates them with the cached projections from previous time steps.

The Bayes inverse script encodes the entire prompt at once and computes log probabilities over the full sequence, so it does not manually manage the cache. However, the same `LlamaModel` is also used by the chatbot example, where the interactive generation loop can call the model repeatedly with an updated KV cache to efficiently generate one token at a time. This is a scenario where the KV cache provides a clear reduction in latency.

4 Experiments

4.1 Chatbot prompts and qualitative behavior

Prompt 1: What is gravity?

- The model produced a fluent paragraph describing gravity as a force that pulls objects with mass together and mentioning its role in keeping planets in orbit. The answer was broadly correct and well structured.

This shows that, for simple factual questions, the model can behave like a reasonable general purpose assistant.

Prompt 2: What is a transformer in terms of machine learning?

- The model gave a long, confident explanation and correctly stated that transformers are widely used in NLP and can scale well. However, it incorrectly described a transformer as a combination of CNNs, RNNs, and LSTMs, rather than a stack of self-attention and feed forward layers.

Prompt 3: Are you sentient?

- The model claimed to be "a sentient being" and then degenerated into a repetitive loop about "a realm of the spirit," repeating nearly the same sentence many times.

Prompt 4: Is the following email spam? *<followed by a GitHub security notification about a third-party OAuth application being authorized, with links to GitHub settings and security logs.>*

- The model correctly answered that the email was not spam, but justified this by saying it was a legitimate email from "an AI assistant named SmolLM," confusing the system prompt with the sender of the email.

This simple regularization step (weight decay plus clipping) turned out to be the most effective way to improve generalization in my experiments, outperforming both synthetic data and ensembles, even though all three methods were layered on top of the same Bayes inverse architecture.

4.2 Ensemble methods

I also experimented with simple ensembles on top of the Bayes inverse classifier. In this setting, I trained five full finetune models with different random seeds, saved their probability outputs on the test subset, and then ensembled them by averaging the spam/ham probabilities for each email before thresholding. This reduced variance across individual runs and produced more stable predictions, but the best ensembled Kaggle submission still plateaued at around 75% accuracy.

Ensembling multiple models did not close the generalization gap as effectively as the single model with weight decay and gradient clipping trained on synthesized data.

5 Conclusion

In this project, I treated the provided Bayes inverse classifier as a starting point and focused on improving its performance through data syntesis, regularization, and ensembling models workflows rather than architectural changes. On the labelled train/validation subset, zero shot and naive prompting behave close to random guessing, while full finetuning with the same underlying classifier achieves much higher accuracy.

I implemented a label aware data augmentation model for spam and ham emails, explored how performance scales with the amount of labelled data. Empirically, both data synthesis and ensembling by themselves plateaued at around 75% test accuracy, while a simple addition of weight decay and gradient clippingto the optimizer and removing ensembling gave the best test error.

Acknowledgements and AI assistance

I used AI assistance (e.g., ChatGPT) to help draft LaTeX boilerplate, and to generate realistic spam/ham synthesis phrases. All code execution, experiments, and numerical results reported here were run by me on the provided project repository.