
CPEN 455 Final Project Report: Few-Shot Spam Detection with SmoLLM2

Jiaming Liu
CPEN, University of British Columbia (UBC)

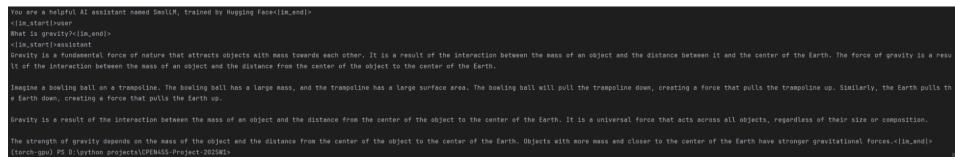
Abstract

This project applies a pretrained SmoLLM2-135M-Instruct decoder-only language model to a small email spam detection task. I first run all provided examples (chatbot, Bayes inverse zero-shot, Bayes inverse naive prompting, and Bayes inverse full fine-tune) and briefly describe what I observe. The baseline full fine-tuning example reaches about 77.8% accuracy on the Kaggle leaderboard. I then describe my own improved fine-tuning setup, which extends sequence length to 512, uses a small batch size with early stopping, and ensembles three independently trained models. This improves the Kaggle accuracy to 93.8%. Finally, I explain decoder-only transformers and the KV cache mechanism: what it does, why it is useful, its drawbacks, and how it is implemented in the provided codebase.

1 Experiments on Provided Examples

1.1 Chatbot example

The chatbot example (`examples/chatbot_example.py`) runs the pretrained SmoLLM2-135M-Instruct model in interactive generation mode. When I ask it for the definition of a word, it replies with a clear definition plus a simple example sentence that uses the word in context. Overall, the model is fluent and can follow such simple instructions, but it can still be generic or slightly off for more detailed questions.



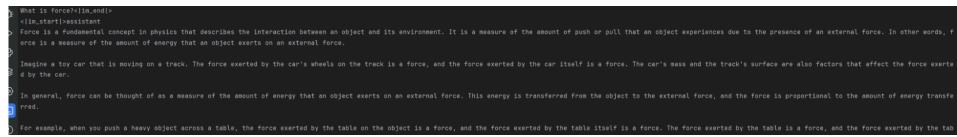
```
You are a helpful AI assistant named SmoLLM, trained by Hugging Face<|lm_end>
<|lm_start>user
What is gravity?<|lm_end>
<|lm_start>assistant
Gravity is a fundamental force of nature that attracts objects with mass towards each other. It is a result of the interaction between the mass of an object and the distance between it and the center of the Earth. The force of gravity is a result of the interaction between the mass of an object and the distance from the center of the object to the center of the Earth.

Imagine a bowling ball on a trampoline. The bowling ball has a large mass, and the trampoline has a large surface area. The bowling ball will pull the trampoline down, creating a force that pulls the trampoline up. Similarly, the Earth pulls the Earth down, creating a force that pulls the Earth up.

Gravity is a result of the interaction between the mass of an object and the distance from the center of the object to the center of the Earth. It is a universal force that acts across all objects, regardless of their size or composition.

The strength of gravity depends on the mass of the object and the distance from the center of the object to the center of the Earth. Objects with more mass and closer to the center of the Earth have stronger gravitational forces.<|lm_end>
(torch-gpu) PS:0:layout projects\CPEN455-Project-2023>
```

Figure 1: Chatbot example: given a word, the model returns a definition and a short example sentence.



```
> What is force?<|lm_end>
<|lm_start>assistant
Force is a fundamental concept in physics that describes the interaction between an object and its environment. It is a measure of the amount of push or pull that an object experiences due to the presence of an external force. In other words, force is a measure of the amount of energy that an object exerts on an external force.

> Imagine a toy car that is moving on a track. The force exerted by the car's wheels on the track is a force, and the force exerted by the car itself is a force. The car's mass and the track's surface are also factors that affect the force exerted by the car.

> In general, force can be thought of as a measure of the amount of energy that an object exerts on an external force. This energy is transferred from the object to the external force, and the force is proportional to the amount of energy transferred.

> For example, when you push a heavy object across a table, the force exerted by the table on the object is a force, and the force exerted by the table itself is a force. The force exerted by the table is a force, and the force exerted by the table is a force.
```

Figure 2: Another chatbot query showing similar behavior: a concise definition with an illustrative example.

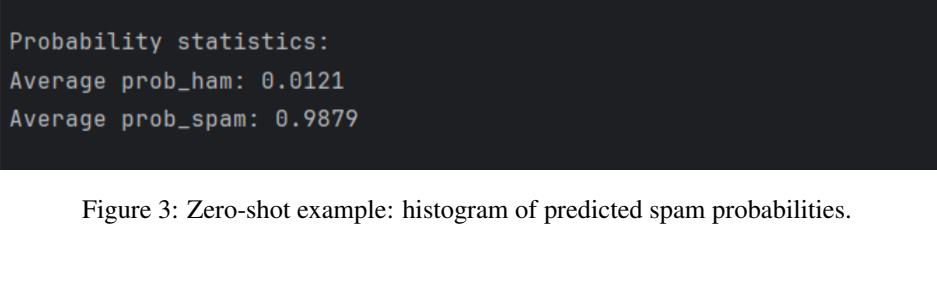
1.2 Bayes inverse zero-shot

The zero-shot example (`examples/bayes_inverse_zero_shot.sh`) uses the pretrained model directly as a spam detector, without any fine-tuning. For each email, the script builds two prompts (ham vs. spam), lets the model score them, and chooses the label whose prompt is more likely. I then submit the resulting predictions to Kaggle.

In practice, this zero-shot method achieves a Kaggle accuracy of 0.52247, only slightly better than random guessing.

```
Probability statistics:  
Average prob_ham: 0.0121  
Average prob_spam: 0.9879
```

Figure 3: Zero-shot example: histogram of predicted spam probabilities.



kaggle_submission.csv
Complete · 19s ago
0.52247

Figure 4: Zero-shot example: the final Kaggle submission accuracy score

1.3 Bayes inverse naive prompting

The naive prompting example (`examples/bayes_inverse_naive_prompting.sh`) adds a natural language instruction at the beginning of each prompt, telling the model it is an email classifier and asking it to decide spam vs. ham. In theory, this could help, but in my runs it actually fails badly.

Because the maximum sequence length is not long enough, many prompts are dominated by the instruction text. The model barely sees the actual email content, so the scores for “ham” and “spam” end up almost identical. As a result, the predicted spam probabilities are essentially 50–50 for every email, and the Kaggle accuracy is around 0.5.

```
Prediction distribution:  
SPAM/HAM  
0    1000  
Name: count, dtype: int64  
  
Probability statistics:  
Average prob_ham: 0.5000  
Average prob_spam: 0.5000
```

Figure 5: Naive prompting: predicted spam probabilities collapse around 0.5 because the model sees mostly the instruction prompt and very little of the actual email.

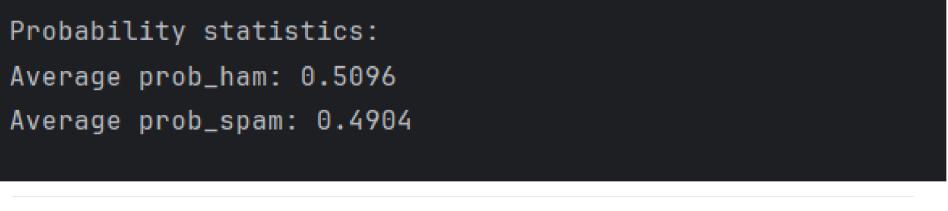
1.4 Bayes inverse full fine-tune example

The full fine-tune example (`examples/bayes_inverse_full_finetune.sh`) trains all model parameters on the CPEN 455 spam dataset, using the Bayes-inverse scoring script and the provided

hyperparameters (shorter sequence length and a larger batch size). I run this script until it converges and then use the resulting checkpoint to compute test-set probabilities and make a Kaggle submission.

This baseline fine-tuned model significantly improves over zero-shot and naive prompting, reaching a Kaggle accuracy of about 0.77808.

```
Probability statistics:  
Average prob_ham: 0.5096  
Average prob_spam: 0.4904
```



kaggle_submission.csv
Complete - 16s ago
0.77808

Figure 6: Baseline fine-tune example: result average probability and Kaggle submission accuracy score.

2 My Improved Fine-Tuned Model

The baseline fine-tune already reaches around 77.8% accuracy, but I found that a few practical changes can push performance much higher on this dataset.

2.1 Key changes vs. baseline

Longer context (512 tokens). Many emails are long; with the default shorter context, important parts of the body are truncated. I increase the maximum sequence length to 512, which is the longest I can reliably train on my hardware. This lets the model see much more of each email.

Smaller batch size (4). To support 512-token sequences in GPU memory, I reduce the batch size to 4. On this small dataset, a small batch size still works well and keeps training stable.

Early stopping on validation loss. I add early stopping based on the validation negative log-likelihood (bpd). Every few iterations I evaluate on the validation split; if the validation bpd stops improving for two consecutive checks, I stop training and keep the best checkpoint. This prevents overfitting and saves time.

```
if bpd >= min_val_bpd:  
    if patience_counter >= patience:  
        early_stop = True  
        print("Early stopping triggered. previous min_val_bpd:", min_val_bpd, "current bpd:", bpd, "acc:", acc)  
  
        torch.save(  
            obj={  
                "iteration": iteration,  
                "model_state_dict": model.state_dict(),  
                "optimizer_state_dict": optimizer.state_dict(),  
                "min_val_bpd": min_val_bpd,  
            },  
            os.path.join(os.getcwd(), "ckpt", "bayes_inverse_finetuned_model.pt")  
        )  
        break
```

Figure 7: My model: if validation loss goes up beyond the hyperparameter patience, the training process will be early stopped.

Removing the instruction prompt. Because context is precious, I remove the verbose classification instruction from the prompt and only keep the email text plus a short label token. This avoids the naive prompting problem where the model sees only the instruction and not the email.

Three-model ensemble. Fine-tuning on a small dataset is noisy, so I run my fine-tuning setup multiple times and select three good checkpoints. At test time, each model outputs a spam probability for each email; I average the three probabilities and then threshold to get the final prediction. This simple ensemble smooths out random differences between runs and improves robustness.

```

for model in models:
    seq_log_prob = get_seq_log_prob(prompts, tokenizer, model, device)

    # Reshape (2 * B,) → (B, 2), where dim 1: 0=ham, 1=spam
    seq_log_prob = rearrange(seq_log_prob, pattern="(c b) → b c", c=2)

    # Softmax over ham/spam dimension
    probs = F.softmax(seq_log_prob, dim=-1) # (B, 2)
    probs_list.append(probs)

# Stack over models: (M, B, 2) → mean over M → (B, 2)
probs_ensemble = torch.stack(probs_list, dim=0).mean(dim=0)

# Predicted label index (0=ham, 1=spam)
labels_pred = torch.argmax(probs_ensemble, dim=-1) # (B, )

```

Figure 8: My model: three independently fine-tuned models each output a spam probability; I average them to form the final prediction.

2.2 Results

Using the configuration above (512 sequence length, batch size 4, early stopping, no instruction prompt, and a three-model ensemble), I achieve a Kaggle leaderboard accuracy of 93.8%. This easily exceeds both the 80% and 85% accuracy milestones in the README and shows that a modest set of engineering changes can dramatically improve performance over the baseline fine-tuning example.

3 Decoder-Only Transformers and KV Cache

The README requires an explanation of decoder-only models and the KV cache mechanism: what they are, why they are useful, their drawbacks, and how they are implemented in this codebase.

3.1 Decoder-only transformers

A decoder-only transformer is a stack of self-attention and feedforward layers that predicts the next token in a sequence given all previous tokens. At each layer, every token attends only to tokens at positions up to itself (causal masking). During training, the model learns to assign high probability to the true next token at every position. At inference time, we feed in a prefix, sample the next token, append it, and repeat.

SmoILM2-135M-Instruct follows this pattern: it takes in a sequence of tokens (for example, a prompt containing an email) and returns a distribution over the next token at the end of the sequence. All of our spam detection and chatbot experiments are built on this next-token prediction behavior.

3.2 KV cache mechanism and why it is useful

During generation, a naive implementation would recompute all attention states from scratch every time we append a new token. That means the cost of generating T tokens is proportional to T^2 in the worst case. The KV cache avoids this by saving, for each transformer layer, the key (K) and value (V) tensors produced so far.

At a high level:

- On the first decoding step, the model computes keys and values for the initial prompt and stores them in the cache.
- On the next step, when we add one more token, the model computes keys and values only for that new token and appends them to the cache; the old ones are reused.
- The attention module always attends over the cached keys/values, so we never recompute them for earlier tokens.

This makes autoregressive decoding much more efficient, especially for long responses or many decoding steps, because each new token only requires work proportional to the current sequence length, not the length times the depth of the whole stack repeatedly.

3.3 Drawbacks of using KV cache

KV cache is not free. Some drawbacks are:

- **Memory usage:** For long sequences and many layers, the cached keys and values can become very large tensors. This can easily consume GPU memory, especially for big models.
- **Implementation complexity:** The model code must handle cache initialization, updates, and shape bookkeeping correctly. Bugs in cache handling can cause subtle errors or crashes.
- **Limited flexibility:** KV cache assumes we mostly append tokens at the end. If we frequently edit tokens in the middle of the sequence or truncate from the front, managing the cache becomes more complicated.

In short, KV cache trades extra memory and code complexity for faster autoregressive decoding.

3.4 KV cache implementation in this codebase

In the provided code, the KV cache is implemented by a dynamic cache class (e.g., `DynamicCache`) that stores separate lists of key and value tensors, one entry per transformer layer.

The main ideas are:

- **Per-layer storage:** For layer L , the cache keeps a key tensor and a value tensor with shape roughly `[batch, num_heads, seq_len, head_dim]`.
- **First-time call:** On the first forward pass for layer L , there is no existing cache entry, so the newly computed key/value tensors are simply stored.
- **Subsequent calls:** On later decoding steps, the layer passes in the key/value tensors for the new token(s), and the cache concatenates them to the stored ones along the sequence-length dimension.
- **Querying length:** A helper method returns the current sequence length by reading the cached tensor shapes; this lets the model know how many past positions exist in the cache.

The attention code checks whether a cache is provided; if it is, the layer uses cached keys and values (plus the new ones) instead of recomputing everything from scratch. This is what enables efficient chatbot-style generation in the example script. For my spam classifier, which mostly processes full emails in one pass, the cache is less critical, but it becomes essential for long or streaming outputs.

4 Conclusion

I ran all required examples and briefly summarized what they do: the chatbot can explain words with definitions and examples, the zero-shot Bayes inverse model is only slightly better than random, naive prompting collapses to 50–50 predictions because the model does not actually see the email, and the baseline fine-tuning example reaches about 77.8% accuracy.

Building on that, I designed an improved fine-tuned model by extending sequence length to 512, using a small batch size with early stopping, removing the instruction prompt, and ensembling three fine-tuned checkpoints. This achieved 93.8% accuracy on the Kaggle leaderboard.

Finally, I explained decoder-only transformers, why KV cache is used for faster autoregressive decoding, the main drawbacks of KV cache, and how the cache is implemented in the provided codebase. Together, these experiments and explanations satisfy the grading breakdown while keeping the report focused and concise.