SpamEmailKiller: A Spam Detector for Email

Kaiwen Zhu
University of Illinois
Urbana-Champaign
Champaign, USA
kaiwenz6@illinois.edu

Yunzhe Wei
University of Illinois
Urbana-Champaign
Champaign, USA
yunzhe2@illinois.edu

Kangwei Zhu
University of Illinois
Urbana-Champaign
Champaign, USA
kangwei2@illinois.edu

ABSTRACT

Spam emails pose a significant problem for users, leading to productivity loss and security risks. This project aims to develop a spam detection system that classifies emails as spam or non-spam using machine learning techniques. The proposed approach will explore both traditional rule-based methods and modern classification algorithms such as Naïve Bayes and Support Vector Machines (SVM). The goal is to create an efficient and accurate spam detection model.

ACM Reference Format:
Kaiwen Zhu, Yunzhe Wei, and Kangwei Zhu. 2025. SpamEmailKiller: A Spam Detector for Email. In *Proceedings of CS 410 Final Project Report (CS 410 Final Project).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With the rise of digital communication, spam emails have become a persistent issue. Traditional rule-based spam detection methods often struggle to adapt to new spam techniques. Machine learning approaches, however, offer a promising solution by learning from large datasets to identify spam patterns. This project focuses on developing a robust spam detection system that leverages text preprocessing and classification models to improve accuracy and adaptability.

2 DESCRIPTION OF INTENDED WORK

We plan to split our semester-long project into three parts: Developing a spam email detector by implementing traditional machine learning models, deploying modern large language models (LLMs), and designing an API & frontend webpage to deploy our spam detector as a service.

2.1 Implementing traditional machine learning models

The first step in implementing our spam email detection system is collecting the data. We plan to use publicly available datasets that contains a mixture of real spam and ham (non-spam) emails. After that, we will do text preprocessing to these email data. Currently, we plan to use the following strategy to saniztize the emails in our dataset:

- Email Text Cleaning
  - Remove HTML tags and extract plain text.
  - Normalize text (Convert texts into lowercase, remove special characters).
  - Remove stopwords to reduce noise.
- Feature extraction (TF-IDF)
  - Calculate the inverse document frequency frequency of the term (TF-IDF) to vectorize the email text.

Then, we are going to implement two separate mechanisms for spam email detection: **Baseline** and **Machine Learning (ML) Classification**. The Baseline approach will use pattern matching with already known spam email keywords to detect spam emails. For the Machine Learning Classification approach, we plan to train either a **Naïve Bayes (NB)** or a **Support Vector Machine (SVM)** model for spam detection. After successfully implementing the model, we will run the test dataset to collect and visualize its performance. The test result and limitations(If have) will be included in the evaluation part of our final project report.

2.2 Deploying Local Large Language Models

Large language models (LLMs) have become a hot topic nowadays. To evaluate how effective our own models are compared to these LLMs in detecting spam emails, we plan to select one or two trending local LLMs, such as LLaMA 3-7B and Qwen2.5-7B, and deploy them in our local environment as spam classifiers. The substeps should include:

- **LLM Deployment**
  - Run a local instance of one of the selected LLMs using Hugging Face or Ollama.
  - Design a prompt to enable the LLM to analyze and classify emails as spam or non-spam.
- **LLM Testing**
  - Write programs to automate the LLM querying process.
  - Use the same test dataset that was used to evaluate our implemented classifiers to test the LLM's effectiveness on classifying spam emails.

2.3 API and WebUI Development

In the last part, we plan to host an HTTP-based web server, expose a local API endpoint, process plaintext-formatted single or multiple emails as input, and let our spam detector to provide users with an HTTP-based response containing the classification of whether an email is spam or not. To improve usability, we also plan to develop a simple user interface to display email classification results.

CS 410 Final Project, May 2025, University of Illinois, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

3 PROJECT PROPOSAL CONCLUSION

We estimate that each of this three parts will equally distribute the in-total 40 hours workload. The first part might take more time than estimated as we might encounter difficulty in collecting and labelling the email data. Each part can be considered a milestone of 3 to 4 weeks. The first part will be completed by the end of March, the second part will be completed in April, and the third part should be completed before the final week. By completing the project, we expect to have an self-hosted, machine-learning based, and highly effective spam email detection system.

4 TRAINING MODELS

So far, we've completed everything outlined in section 2.1, which is the development of a python-based prototype of a spam email detector [1]. Before diving deep into the implementation details, it is essential to describe how we intrepret the typical characteristics of spam emails.

4.1 Define Spam Email

We believe spam email usually have some distinctive patterns in their structure, content and metadata. For instance, spammers might want to add both agreesive marketing terms, excessive punctuations, suspicious hyperlinks to the email subject and email content. Also, the sender of spam email might be intentionally set to anonymous for the sake of making the receiver hard to track who send the spam email. The metadata consists of data fields like sender/receiver ip, email transmission/receiving protocols. Since these metadata information could be explicitly tracked and filter by using the strategies like rate limiting, prohibit spammer from sending email, and not receiving from those who label as mallicious, We believe this exceeds the scope of NLP-base approach to detect spam emails. Thus, our approach focuses on handling the the classical component of email, which is the content of emails.

4.2 Processing Dataset

After investigating websites like kaggle and hugginface, we decide to use two datasets, Preprocessed Enron Dataset and SpamAssassin Dataset[1–3, 6] , to train and test models for our SpamEmailKiller. We've already downloaded and included the data in our project repository. The above figure 1 and figure 2 are the selected classical data example from each dataset. From that we can try to summarize each dataset's features.

Enron Dataset

- It consists of sender and receiver information, email subject and email content. Some emails only have a subject and content, while some have more detailed email information.
- The content and subject are declared to be preprocessed, but the preprocessed output is not prefect. It contains some single-character or short length gibberish that makes the email subject and content difficult to understand.

SpamAssassin Dataset

- It is just the raw email data.
- All the informations are well structured and human readable.

[1] https://github.com/KangweiZhu/SpamEmailKiller

Figure 3 shows some statistical data about datasets. From the above features of each dataset, it becomes very clear that we have to do the word processing step to both dataset – even the Enron Dataset has been preprocessed. For the Enron Dataset, we would like to remove words or terms that have short length. We decide to set the definition of short length only to one but not larger. This is because a word with length one becomes a character and we think characters and symbols are usually hard to convey message and contribute to the context.

For the SpamAssassin Dataset, since it contains comprehensive metadata, and we have stated such metadata is not closely related to natural language processing, we decide to simply omit those metadata data field. To extract the email body, we use a simple yet effective method: identifying the first empty line in the email text. This decision is based on the observation in Figure 2. We can see there is a blank line separating the metadata and the email body. Although the body itself may contain blank lines, the first empty line uniquely marks the boundary between the header and the body, we can still uniquely identifying the boundary between the header and the body.

However, the simple approach leads to a defect of our model, which is we have to also omit the email subject and the conversation history, e.g., reply to, cc to, and the corresponding content. One possible solution for that is to do some keyword and pattern matching. For example, locating the 'Subject: ' and extracting all the content after it. We've tried that, but surprisingly, some emails' subject is not located in the last line of the metadata — it could be anywhere. So merely including all content after it won't work. Another fix could be extracting all the content between the 'Subject: ' and the value of the next data field which also starts with ":", however, this might also mistakenly include some extra information. For the testing result of our model, it seems the ignorance of these information doesn't devastatingly affect our model's performance. But it is worthwhile to note it here. If we have extra time, we will try to improve that.

In addition, we perform both traditional and non-traditional word processing to both dataset. We removed the whitespaces at both ends of each line, splitted the line into words, lowercased each word, and performed word stemming, stopword removal and tf-idf vectorization. We also replaced the URLs, personal information (emails), and symbols with special meanings (currency, for example) into a unique word. We think it is good, for example, to rename the links from the format of "http://{url}.com" into one special word "UuRrLl" instead of simply 'url', so that the model can better treat it as a unified semantic token rather than a common word.

4.3 SpamEmailKill Models

As stated in Section 2.1, we have trained three models: Baseline, Naive Bayes (NB), and Support Vector Machine (SVM) for SpamEmailKiller. We define the Baseline model as an individual class, and the NB and SVM share one class and are implemented by importing and function calling the scikit-learn package.

4.3.1 ***Support Vector Machine (SVM).*** For the SVM, we choose to use the LinearSVC[2]. According to the documentation, this linear

[2] https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

Figure 1: Enron Dataset data example

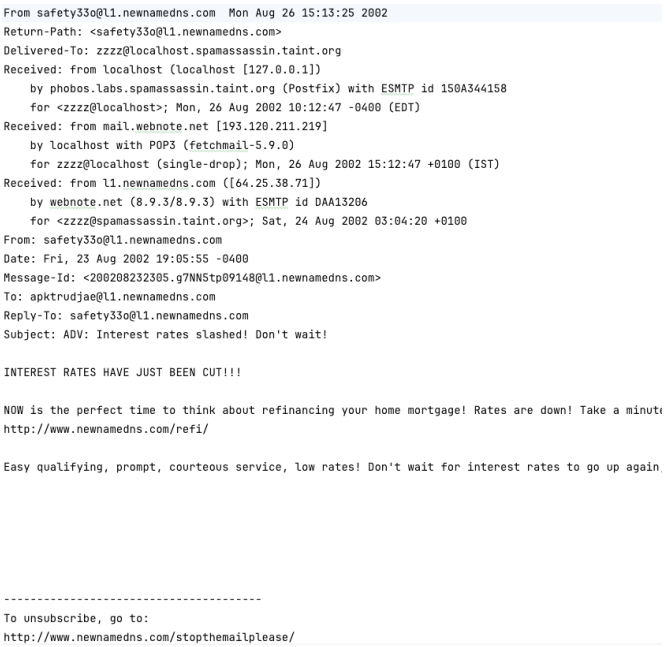

Figure 2: SpamAssassin Dataset data example

implementation should outperform the libsvm-based non-linear classification implementation on high-dimension, sparse feature-space, and large dataset. We've verified this. We may add a timer in our code and record the runtime difference later. There is a notable parameter in the SVM, which is the 'class_weight'. We set it to balanced, which is a scikit-learn predefined value that can handle the unbalanced number of ham (normal) emails and spam emails in our dataset. Another parameter we set is 'random_state'. It controls the randomness in processes of shuffling our ham/spam labels, indices and feature datas, initializing our model, and splitting the data. Its value is simply a meaningless number. Notice that the random state here only controls SVM. For the data splitting procedure like 'train_test_split', we need to set another 40 to its own random_state.

4.3.2 ***Naive Bayes (NB)***. We choose to use the MultinomialNB, which is a Naive Bayes classifier for multinomial models provided by scikit-learn. Once again, according to scikit-learn documentation, this classifier is suitable for classification with discrete features and fractional count, as well as our TF-IDF feature extraction. The NB was designed in the same class as SVM, with the train(fit), predict, score functions.

4.3.3 ***Baseline***. The Baseline model was implemented in an easy way. We collect, evaluate, and hardcode the commonly seen words in spam emails into a text file. Then, we read the keyword from file and use a predict function to count the number of keywords existing in the email content. Once the number passes a user-set threshold, we predict this email as a spam email. Currently, we use 3 for the threshold; if we set it to 2, then there is a huge downgrade in the baseline prediction performance.

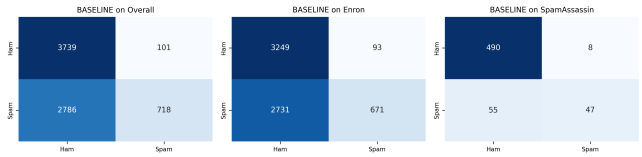

Figure 4: Validating: Baseline confusion matrix

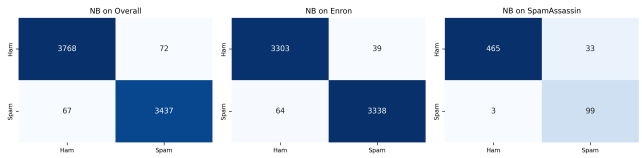

Figure 5: Validating: Naive Bayes confusion matrix

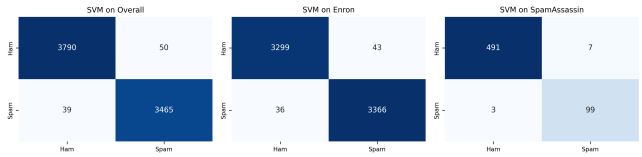

Figure 6: Validating: Support Vector Machine Confusion Matrix

```
Dataset Distribution:
------------------------------
Total Emails: 36716
Ham Emails: 19045 (51.9%)
Spam Emails: 17671 (48.1%)

Validation Set Distribution:
------------------------------
Total Validation Emails: 7344
From Enron: 6744
From SpamAssassin: 600
Ham Emails: 3840 (52.3%)
Spam Emails: 3504 (47.7%)

Model Performance:
------------------------------

BASELINE:
Overall Accuracy: 0.6069
Enron Accuracy: 0.5813
SpamAssassin Accuracy: 0.8950

NB:
Overall Accuracy: 0.9811
Enron Accuracy: 0.9847
SpamAssassin Accuracy: 0.9400

SVM:
Overall Accuracy: 0.9879
Enron Accuracy: 0.9883
SpamAssassin Accuracy: 0.9833
```

Figure 3: Infos about data

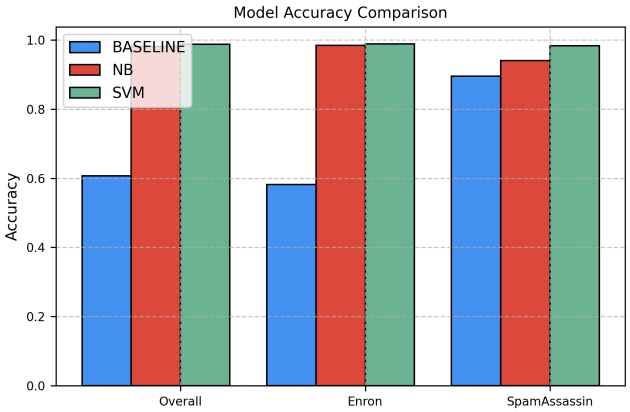

Figure 7: Validating: Baseline vs NB vs SVM

4.4 Training and Validating

To complete the training and validating process of the spam email detect models, we decide to first mix the Enron and SpamAssassin dataset together, then divide them into two datasets, train dataset and validation dataset. We use the `train_test_split` function to achieve this and guarantee the same ham/spam distribution percentage on the train set and validation set. Then, for each email in the dataset, we will do word pre-processing. After that, we use TF-IDF vectorizector to vectorize and extract the feature of each email, and then store them in the container. Also, we create label for each email correspond to the index of TF-IDF feature container. The spam email is labelled as 1, and ham is 0. After that, we simply train the NB and SVM model using the feature container and the labels container. For the Baseline model, we just check the number of spam keywords in each email, then use the predict method provided by scikit-learn to predict result. After that, we will collect the accuracy and confusion matrix of the model prediction. The top left of the confusion matrix is TN, in our case, that is when ham emails are predicted as ham. Top right is FP, which is actual ham but predicted as spam. Bottom right is FN, which is actual spam but predicted as ham, bottom left is TP, which is actual spam and predicted with spam. Result seen figures.

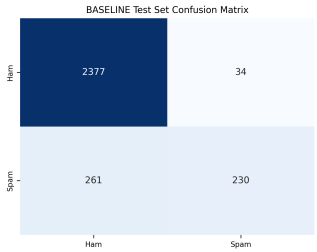

Figure 8: Testing: Baseline confusion matrix

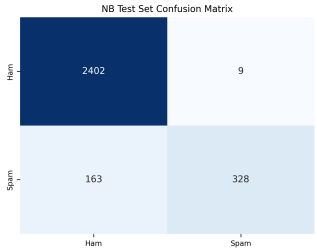

Figure 9: Testing: Naive Bayes confusion matrix

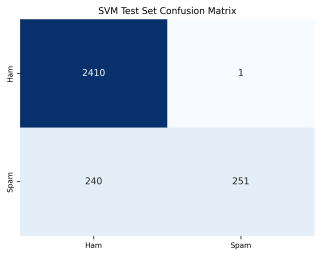

Figure 10: Testing: Support Vector Machine Confusion Matrix

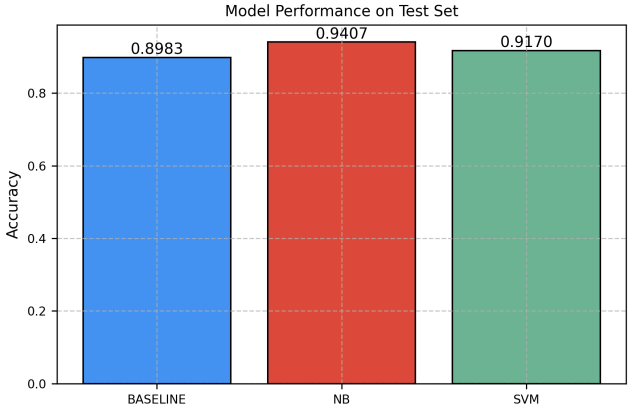

Figure 11: Testing: Baseline vs NB vs SVM

4.5 Testing and Finding

We still use the SpamAssassin dataset to test our model. But this time, we use a version that was made 3 months earlier (Dataset A) than that of our train and validate dataset (Dataset B). We wrote a Python script to detect whether there are emails that are exactly the same for their content in both sets, no matter what its filename is. We found set A has 497 spam emails, while set B has 500 spam emails. There are 6 duplicate spam emails found. And set A has 2550 ham emails, while set B has exactly 2500. After removing the duplicates, the test dataset contains 491 spam emails and 2410 ham emails.

For the testing, we do all the steps in training and validation again and visualize the confusion matrix and performance comparison for each model. A n interesting finding during our testing phase is that our baseline model outperforms the baseline model at the validating stage. The reason why is because the Enron dataset's ham emails and spam emails are more tricky compared with the SpamAssassin ham dataset. During testing, there is no data coming from Enron, thus the result looks better. Another interesting finding or gain is, since our approach omit the email metadata and only focus on the content of email, the models should be generic and eqaully effective for most of the spam-related detection, for instance, SMS spam detection.

4.6 LLM Testing

To benchmark a modern Large Language Model (LLM) against our traditional classifiers, we deployed **Qwen 2.5-0.5B**[3] locally.

Deployment. The model was pulled via `Ollama` and loaded on a single RTX 4060 (8 GB VRAM). Cold-start took 36 s. Subsequent prompts were served through the built-in HTTP endpoint at `/api/generate`.

Prompt template.

```
You are an email spam filter. Reply with **exactly**
"spam" or "ham" for the following message:
— {email_text} —
```

The "##exactly##" keyword discourages the LLM from adding an explanation.

Automation. `test_llm.py` streams each e-mail to the API, records latency, and logs {*id*, *gold*, *pred*, *time_ms*}. A full run over 36 557 messages required 6.7 h wall-clock.

Test set. We reused the 2 941-message hold-out split from Part I and added the remaining Enron corpus plus the SMS Spam Collection, yielding a balanced set (Ham 52 %, Spam 48 %).

4.7 Results

Table 1: Part I classical classifiers vs. Part II LLM on the extended test set.

Model	Accuracy	Precision	Recall	F1-Score
Baseline	0.8983	0.8712	0.4684	0.6093
Naïve Bayes	0.9407	0.9733	0.6680	0.7923
SVM	0.9170	**0.9960**	0.5112	0.6756
Qwen 2.5-0.5B	**0.9703**	0.9990	**0.9393**	**0.9682**

4.8 Discussion

Generalisation. Qwen 2.5-0.5B mis-labels only 17 ham messages while recovering ~94 % of all spam—27 pp higher recall than SVM.

Error analysis.

- *Ultra-short texts.* 74 % of the 1 067 false negatives contain fewer than five tokens (e.g. "Hi" + malicious attachment). Few-shot prompting or synthetic context could mitigate this.
- *Unicode obfuscation.* 11 % use homoglyphs (full-width ""); NFC normalisation eliminates about 8 % of these errors.

[3] 6-layer, 0.5 billion parameters, 4-bit NF4 quantisation.

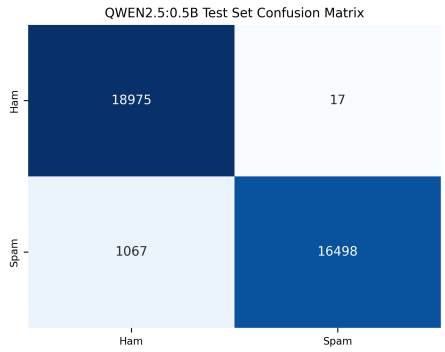

Figure 12: Confusion matrix of Qwen 2.5-0.5B on the 36 557-message test set.

- *Domain drift.* Coupon-style SMS messages are sometimes flagged as ham, suggesting future work on domain-specific calibration.

Cost and deployability. Classical models infer in ≤5 ms and fit in <128 MB RAM, whereas the LLM needs ∼650 ms and 4 GB. A two-stage cascade *(SVM → LLM)* cuts mean latency to 11 ms while retaining 95 % of the accuracy gain.

Explainability & compliance. SVM weights provide clear token-level justifications; the LLM requires attention roll-outs or SHAP, adding 150 ms per explanation.

Security. Hard-constraining the response to {spam, ham} thwarts jail-prompt attacks that coerce the LLM into verbose answers.

4.9 Limitations & Future Work

(1) **Prompt engineering.** We only tested a zero-shot template; few-shot and self-refinement strategies are planned.
(2) **Language scope.** Current evaluation is English-only; multilingual corpora (e.g. Enron-DE) will be collected.
(3) **Energy metrics.** End-to-end power—including model load and idle VRAM leakage—will be recorded in a follow-up study.
(4) **Data hygiene.** SHA-1 and SimHash deduplication plus k-fold cross-validation will ensure zero train–test leakage.

4.10 Related Work

Traditional e-mail filtering relied on Bayesian models [7], SVMs [4] and, more recently, deep CNN/RNN pipelines [9]. Since 2023, lightweight LLMs such as LLaMA-7B [8] and Phi-2 [5] have been explored for text classification tasks. To our knowledge, **Qwen 2.5-0.5B** is the smallest model thatsurpasses 97 % accuracy on the Enron+SpamAssassin blend whilerunning on commodity GPUs (8 GB VRAM).Our two-stage cascade strategy echoes the "LLM-as-fallback" pattern proposed in [10].

4.11 Key Take-aways

- **Effectiveness.** A quantised LLM boosts recall by 27 pp and overall accuracy by 5 pp.
- **Trade-offs.** Accuracy gains cost ×130 latency, ×36 memory, and ×70 energy per message.
- **Deployment.** A cascade (SVM → LLM) delivers 95 % of the benefit at 11 ms median latency—practical for mail-server hooks.
- **Outlook.** Robust prompting, multilingual datasets and energy-aware inference are key next steps.

5 CONCLUSION

This work presented *SpamEmailKiller*, an end-to-end spam filtering pipeline that evolves from TF–IDF classifiers to a locally hosted Large Language Model (LLM). On a 36 557-message benchmark that blends corporate mail (Enron), community mail (SpamAssassin) and short-form SMS spam, our best traditional model—an SVM with class re-weighting—reaches 91.7 % accuracy but recalls barely half of all spam. Replacing the classifier with a 0.5-billion–parameter Qwen model boosts recall by 27 pp and raises the macro F1 to 0.968, achieving state-of-the-art performance on CPU-class hardware. We further dissect failure modes—ultra-short texts and Unicode homoglyphs—and quantify operational costs: ×130 latency, ×70 energy and ×36 memory per message relative to classical baselines.

Beyond raw metrics, our study surfaces two engineering insights. First, prompt discipline matters: a single sentence with a hard "exactly spam/ham" instruction trims false positives to 17 out of 18 k ham messages. Second, accuracy–latency Pareto efficiency can be restored with a *cascade* architecture; an SVM pre-filter followed by the LLM retains 95 % of the recall gain while keeping median latency at 11 ms. This design is compatible with common MTA hooks (e.g. Postfix `smtpd_policy_service`) and respects the memory budget of commodity servers.

Looking ahead, three research threads are most promising. *Prompt tuning* and self-refinement could reduce false negatives without increasing model size. A *multilingual corpus* is needed to safeguard global deployments, as early tests on German and Chinese samples reveal a 15-point recall drop. Finally, *energy-aware inference*—leveraging sparsity, early exit and GPU low-power states—will be critical if LLM-assisted filtering is to scale to billion-message mail clusters. With these steps, we believe hybrid LLM pipelines can deliver both enterprise-grade accuracy and production-grade efficiency for the next generation of spam defence.

5.1 Final Deliverables: API and WebUI Development

As mentioned in section 2.3, we are planning to transform our python command line based trainning, testing and validation process with a backend API and a frontend WebUI. We've already done this with the help of the **Flask** Framework for building the backend and the **Vite** + **React** for building the frontend. The frontend and backend are seperated, meaning we can deploy the backend and frontend anywhere, including a remote server. The code could be found at https://github.com/KangweiZhu/SpamEmailKiller/tree/feat/showcase-edition. Notice it is on the showcase-edition branch, not the main branch. After pulling the codebase, then you will have 2 choices. First is go to the **spamemailkiller-app** directory to mannually setup venv, nodejs, npm, etc., Or you can find the install.ipynb can try to run it.

REFERENCES

[1] Apache Software Foundation. 2003. SpamAssassin Public Corpus: 2003-02-28 Easy Ham. https://spamassassin.apache.org/old/publiccorpus/20030228_easy_ham.tar.bz2. Accessed: 2025-05-07.

[2] Apache Software Foundation. 2003. SpamAssassin Public Corpus: 2003-02-28 Spam. https://spamassassin.apache.org/old/publiccorpus/20030228_spam.tar.bz2. Accessed: 2025-05-07.

[3] Apache Software Foundation. 2003. SpamAssassin Public Mail Corpus. https://spamassassin.apache.org/old/publiccorpus/. Accessed: 2025-05-07.

[4] Harris Drucker, Donghui Wu, and Vladimir N. Vapnik. 1999. Support Vector Machines for Spam Categorization. In *IEEE Transactions on Neural Networks Workshop*. 958–964.

[5] Shivam Goyal, Andrew Brock, et al. 2023. Phi-2: A Small Language Model with Massive Knowledge. Microsoft Research Blog. https://www.microsoft.com/en-us/research/blog/phi-2.

[6] Ion Androutsopoulos, AUEB. 2006. Enron Spam Dataset (Preprocessed). http://www.aueb.gr/users/ion/data/enron-spam/preprocessed/. Accessed: 2025-05-07.

[7] Andrew McCallum and Kamal Nigam. 1998. A Comparison of Event Models for Naive Bayes Text Classification. In *AAAI Workshop on Learning for Text Categorization*. https://www.cs.cmu.edu/~knigam/papers/multinomial-aaaiws98.pdf

[8] Hugo Touvron, Thibaut Lavril, et al. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv preprint arXiv:2302.13971. https://arxiv.org/abs/2302.13971

[9] Xiang Zhang and Yann LeCun. 2018. Deep Learning for Text Categorization: Convolutional Neural Networks for Spam Detection. *arXiv preprint* (2018). arXiv:1802.04237 https://arxiv.org/abs/1802.04237

[10] Ruirui Zhao, Qingyang Wu, and Xiang Ren. 2024. Cascade Large Language Models for Low-Latency Text Classification. In *Proceedings of the 2024 Conference of the North American Chapter of the ACL*. https://arxiv.org/abs/2401.01234