Deep Learning for SMS Spam Detection: A Transformer-Based Approach

Ben Snyder*, Alejandro Morales*, Sean Davis*
*Georgia Institute of Technology, Atlanta, GA, USA
{bsnyder39, amorales74, sdavis361}@gatech.edu

***Abstract*—In the age of reliance on constant communication between individuals and organizations, it is common for users to receive various forms of spam or scam messages, some of which can lead to stolen information and other privacy breaches. While there have been various naive attempts to remedy this issue and provide a safe messaging platform, such as keyword and area code filtering, these can easily be avoided by malicious attackers. We propose a lightweight Transformer-based classifier to detect spam in SMS messages and output flagged words, addressing the limitations of traditional filtering methods. We evaluate various tokenization strategies, loss functions, and architectural settings to optimize performance, achieving strong results even under adversarial testing.**

I. Introduction

Spam messages, including unsolicited emails and texts, present ongoing challenges for individuals and organizations by impairing productivity, user experience, and security. Effective spam detection is vital to counter phishing attempts, fraud, and privacy breaches [1]. Traditional spam classification methods, such as heuristic rules, keyword filtering, and classical machine learning techniques (e.g., Naive Bayes, logistic regression, and Support Vector Machines), rely heavily on manual feature engineering and struggle to capture complex semantic and sequential relationships in text [2], [3]. Although recent deep learning approaches, like Convolutional Neural Networks (CNNs) and Feedforward Neural Networks (FNNs), improve feature learning from raw data, they often overlook sequential dependencies or positional relationships critical for text analysis.

In recent years, the use of a Transformer for sequential processing has completely evolved the deep learning landscape, allowing for efficient processing over variable length sequences. This architecture leverages self-attention mechanisms to capture long-range dependencies and positional relationships in text, which greatly improve on the clear flaws of the previous deep learning approaches. While the Transformer was originally designed as a component of a sequence to sequence model [4], stacking several Transformer layers has been found to act as a very powerful encoder, summarizing the information the input in a very compressed subspace. The great difficulty of Natural Language Processing tasks involves deriving a succinct representation of the words and the relationship between them, but once we can produce this representation, the classification task is largely trivial. There are many ways to achieve this goal, but for our lightweight binary classification goal, we use a small, simple feedforward neural network, which can be trained in a standard end to end fashion with the Transformer Encoder. However, due to the inconsistent nature of text messages, we must be careful to take note of various forms of noise in our input, including unseen words, and random characters, to ensure our model derives the correct context out of each word. Our work focuses on working through this text inconsistency and analyzing how various design choices, including tokenization strategies, loss functions, and hyperparameters, influence performance. We hope to use this model as a lightweight embedded piece of a messaging framework to allow users to detect spam and learn from flagged words for future texts.

II. Dataset

We utilize a collection of several diverse SMS datasets, originally organized by Almeida et al. [5] for mobile phone research. These datasets include spam and non-spam messages, ranging from only a few words to over 150 words and taken from various sources, including UK forum posts, Singapore student texts, and other wide reaching datasets. The diversity of these examples (which range from casual communication to promotional material) is essential for our purposes, since we want to classify sample messages of many different styles. In total, our dataset comprises of 5,574 English SMS messages labeled as either `ham` (non-spam) or `spam`. Approximately 13% of the messages are labeled spam, presenting a notable class imbalance that we take special note of to ensure our model can learn both categories equally.

The dataset reflects real-world SMS characteristics: informal grammar, misspellings, links, and punctuation artifacts. This makes it a suitable benchmark for evaluating spam classifiers under noisy, user-generated inputs. The diversity of message styles supports generalization, while the imbalance presents a realistic classification challenge.

Note the raw data set consists of our sample text as a single string and a label (`spam` or `ham`). To use this data to train / use our model, we need to convert all the strings to numeric values. We discuss this in detail in section III-B1

III. Methodology

In this section, we will detail the general approach we took to process our data in preparation for model training and testing. The architecture consists of a preprocessing/tokenization process, which is fed into an embedding layer with positional encoding, then followed by K Transformer Encoder layers and a final feedforward neural network layer which outputs its prediction. We also extend this model to return the top K tokens which were relevant for its decision.

A. Motivation

When attempting to design a deep learning model to solve the spam classification problem, we noted a few clear properties of the problem space that we needed to address: we have sequential, non numeric input, a binary output, and we need to capture multiple meanings of the same word in different contexts. The natural step was to explore leveraging the powerful attention-based Transformer architecture to process the input data. However, a vanilla Transformer outputs intermediate encoded representations, not a class distribution, so we knew we needed a separate classifier. We knew a sufficiently large feedforward neural network would be able to learn the classification function, so we decided on a traditonal one layer FFN with ReLU activation. This architecture was immediately successful on testing (after sufficient hyperparameter tuning), so additional performance boosts resulted solely from preprocessing improvements and loss function changes.

B. Pre-Processing

1) Text Processing and Tokenization: Before feeding SMS messages into our Transformer classifier, we normalize and tokenize them into fixed-length sequences of $L = 180$ tokens. We also converted the labels into binary values, with 0 representing `ham` and 1 representing `spam`. We implemented six preprocessing pipelines, each differing in how much text structure and vocabulary detail is preserved:

- **Minimal preprocessing** (`raw_lowercase`): Remove all non-alphabetic characters, lowercase the text, split on whitespace, append `<eos>`, and pad with `<pad>` tokens to length L.
- **Default** (`preprocess_text`): Replace URLs and email addresses with `<url>` and `<email>` tokens via regular expressions, strip remaining punctuation (except periods and angle brackets), lowercase, split on spaces, prepend `<cls>`, append `<eos>`, and pad or truncate to length L.
- **No special tokens** (`no_special_tokens`): Identical to the default pipeline except that `<cls>` and `<eos>` markers are omitted; only padding enforces fixed length.
- **Raw** (`preprocess_raw`): Lowercase and split on whitespace only, then add `<cls>` and `<eos>` markers and pad/truncate to L.
- **Stemmed** (`preprocess_stemmed`): Apply the default pipeline's cleaning steps, then run each token through a Porter stemmer before adding `<cls>`/`<eos>` and padding to L.
- **WordPiece** (`preprocess_wordpiece`): After default cleaning, tokenize using a BERT WordPiece tokenizer (via the HuggingFace library), then insert `<cls>`/`<eos>` and pad/truncate to L.

Each function ensures that numeric characters are preserved (except in the minimal pipeline), that padding and special markers enforce uniform input length, and that common noisy elements, such as URLs, emails, and stray punctuation, are handled consistently. These pipelines allow us to trade off vocabulary size, sequence length, and semantic richness. The detailed quantitative comparison appears in Section IV-D.

2) Vocabulary Construction: In order to make the tokenized words fit for our deep learning model (which requires numeric Tensor inputs), we must convert all of the words to encodings. We choose to develop a comprehensive Vocabulary, which maps all words found in our data set (which is representative of the global set of text words) to unique integers. We also reserve unique indexes for our special tokens, which are treated as another element in the Vocabulary. To ensure we keep this vocabulary representative of the global word set, we only insert words that are found more than a given lower threshold of times in our data set (as to not create embedddings for garbage words). This threshold is configurable, but we found that choosing a two word minimum helped grow the vocabulary to a sufficient size without including signficiant noise. If a given word is not found in our vocabulary during model inference, we use a default special token `<unk>` to signify an unseen token (which can still be used in the model's decision making).

C. Transformer Classifier

1) Embedding Layer + Positional Encoding: We use a configurable sized embedding layer to allow the model to learn good representations for each input token. Additionally, we have a separate positional encoding layer, which is also a learned parameter, which allows our model to take into account the order of the given tokens. We sum the output of these two separate layers together to provide our final embedding result. Additionally, we append a "cls" embedding to the front of each embeddinng output, which acts as a summary token for the whole sequence. We use the output of this cls index as the final encoding.

2) Transformer Encoder Layer: We use a configurable number of Transformer Encoder layers to help transform the input embeddings into intermediate representations which eventually provide a comprehensive summary of the

full sequence. The Transformer Encoder uses self attention and element-wise feed forward networks to efficiently derive how important each element in the sequence is to the others. Although the Transformer Encoder outputs final intermediate results for each element in the sequence, we use the results for the first element, the specially trained cls parameter, to pass to the next layer for classification purposes.

Additionally, we add a forward hook in the first Transformer Encoder layer to capture the attention weights that were the most relevant during each forward pass. We do this by recording the highest attention weights for the source element cls (the summary element). The higher weighted indices had more input on the decision making. Using these weights, we can recover the original input tokens that were the most relevant, allowing us to analyze the model's decision making.

3) Feedforward Neural Network Layer: We use a traditional feedforward neural network as a classifier, with one hidden layer (with a ReLu activation) and an output layer. We do not apply an activation function to the output layer, since we will later test different loss functions to remedy the class imbalance, some of which require the raw logits (as opposed to a normalized distribution).

D. Training our Model

1) Optimizer: We used AdamW as our optimizer due to its strong performance in transformer-based models. AdamW improves upon Adam by decoupling weight decay from gradient updates, which leads to more effective regularization and more stable convergence. This is particularly important for large models that are prone to overfitting. Although alternatives like SGD with momentum are viable, they typically require more careful tuning and longer training. Given AdamW's success in similar tasks, we chose to focus our tuning efforts on loss functions and architectural parameters rather than optimizer selection.

2) Loss Function: Loss functions guide how models learn by shaping their parameter updates. For our binary classification task, we began with Binary Cross Entropy (BCE), a common choice, but its performance was limited by the class imbalance in our dataset, where spam messages were much less frequent than ham.

To address this, we used Weighted BCE (WBCE), which increases the penalty for misclassifying spam. By varying the `pos_weight` parameter across several values (0.5, 1.0, 1.5, 2.0), we aimed to improve the model's sensitivity to the minority class.

We also explored Focal Loss, which emphasizes harder examples and reduces the influence of easy ones. It introduces two tunable parameters: α, which adjusts class weighting, and γ, which controls the focus on misclassified samples. We conducted a grid search over $\alpha \in 0.25, 0.5, 1.0$ and $\gamma \in 1.0, 2.0, 3.0$ to identify the most effective configuration.

IV. Experimental Setup and Results

A. Training Pipeline

To ensure robust evaluation, we used a two-stage splitting strategy. The final training used a stratified 70/15/15 split to maintain the balance of the class between the training, validation, and test sets, which is critical for unbalanced tasks. Random splits risked distorting this balance.

For hyperparameter tuning, we used Stratified K-Fold Cross-Validation, which preserved class ratios across folds and provided more reliable performance estimates. Aggregating results across folds also allowed us to monitor training stability via standard deviation plots.

B. Hyperparameter Search Strategy

Tuning hyperparameters is challenging due to the lack of universal rules for selecting optimal values. Although grid search is common, it often lacks interpretability. Instead, we used a structured sweep approach, varying one hyperparameter at a time while keeping others fixed. This allowed us to isolate each parameter's impact and better visualize its effect on model performance.

We grouped the hyperparameters into two categories, beginning with architectural choices that affect the model's capacity and expressiveness.

- Embedding dimension: size of token embeddings; higher values increase capacity and cost.
- Feedforward dimension: width of the transformer's internal MLP; deeper layers improve expressiveness.
- Attention heads: number of parallel attention mechanisms; more heads capture richer context.
- Dropout rate: regularization technique to reduce overfitting by randomly masking units.
- Classifier hidden dim: size of the final dense layer; controls decision boundary complexity.

The second group of hyperparameters can be classified as optimization hyperparameters. These control how the model learns during training and influence how effectively it converges.

- Learning rate: controls the step size for parameter updates; smaller values promote stable convergence.
- Batch size: number of samples per training step; smaller sizes (e.g., 16 or 32) improve generalization, while larger ones speed up training.

C. Loss Function Comparison

Table 1 shows the best configuration using BCE loss. Although BCE does not directly address class imbalance, the model performed well. The top model used a lightweight architecture, suggesting that larger models did not offer additional benefits given the simplicity of the task and dataset. The smaller model likely generalized better while also reducing overfitting and training time.

Its low learning rate contributed to more stable convergence. These results suggest that the model's success may

Parameter	Value
Model Configuration	
Embedding Dimension (`embed_dim`)	128
Feedforward Dimension (`ff_dim`)	128
Number of Attention Heads (`num_heads`)	4
Dropout Rate (`dropout`)	0.0
Batch Size (`batch_size`)	32
Learning Rate (`learning_rate`)	1e-4
Model Parameters	737,921
Best Performance (Epoch 9)	
F1 Score	0.9296
Precision	0.9851
Recall	0.8800
Validation Loss	0.0850
Training Loss	0.0436

TABLE I: Best configuration and performance using Binary Cross-Entropy (BCE) loss.

Multiplier	Val Loss	Recall	F1	Precis
0.5	0.1695	0.9000	0.9247	0.9507
1.0	0.4379	0.8926	0.9204	0.9500
1.5	0.4920	0.8933	0.9210	0.9504
2.0	0.5757	0.8867	0.9048	0.9236

TABLE II: F1, Precision, Recall, and Validation Loss for different `pos_multiplier` values (WBCE).

Parameter	Value
Model Configuration	
Embedding Dimension (`embed_dim`)	128
Feedforward Dimension (`ff_dim`)	128
Number of Attention Heads (`num_heads`)	4
Dropout Rate (`dropout`)	0.0
Batch Size (`batch_size`)	32
Learning Rate (`learning_rate`)	3e-4
Model Parameters	737,921
Best Performance (Epoch 6)	
F1 Score	0.9048
Precision	0.9236
Recall	0.8867
Validation Loss	0.5757
Training Loss	0.1512

TABLE III: Best configuration and performance using WBCE (`pos_weight` $= 12.92$).

(a, b)	Val Loss	Recall	F1	Prec.
(0.25, 3.0)	0.0050	0.9267	0.9360	0.9456
(0.50, 2.0)	0.0144	0.9067	0.9379	0.9714
(0.50, 3.0)	0.0081	0.9067	0.9315	0.9577
(1.00, 2.0)	0.0288	0.8933	0.9404	0.9926
(0.25, 2.0)	0.0099	0.8926	0.9366	0.9852
(1.00, 3.0)	0.0201	0.8800	0.9263	0.9778
(0.25, 1.0)	0.0093	0.8792	0.9225	0.9704
(0.50, 1.0)	0.0416	0.8725	0.9286	0.9924
(1.00, 1.0)	0.0281	0.8658	0.9214	0.9847

TABLE IV: F1, Precision, Recall, and validation loss for various (α, γ) settings in Focal Loss (sorted by recall).

reflect its ability to optimize effectively within the dataset's constraints, rather than its capacity to learn deeper or more complex patterns. This may have led to a preference for the majority class.

Table 2 shows WBCE did not consistently outperform BCE in terms of F1 score, it did improve recall. This suggests that increasing the pos_weight made the model more sensitive to spam by penalizing misclassifications more heavily. However, this also led to higher validation loss, reflecting a tradeoff between capturing harder examples and minimizing overall loss.

These results highlight that the best model is not always the one with the lowest loss, but the one that aligns with task priorities, such as detecting minority class instances. Interestingly, the highest F1 score came from the smallest multiplier, which may indicate that the model prefers simpler optimization paths. It is also possible that the model lacks the capacity to benefit from stronger weighting. Future work could explore whether a larger model would be better suited to take advantage of more aggressive reweighting.

The optimal configuration for the weighted BCE setup was similar to that of standard BCE, with the best performance coming from a relatively small model. This again raises the possibility of underfitting or limited representational demands. A key difference was the use of a higher learning rate, which may have allowed the model to correct misclassifications more quickly.

Performance peaked at epoch 6 before declining, suggesting that the optimizer may have moved into less favorable regions as training progressed. This highlights the importance of monitoring performance across epochs and suggests that early stopping or adaptive learning rate schedules could improve results.

Focal Loss achieved the highest F1 score overall, although it was not consistently the best across all configurations. Its variability is interesting. The best result came from a setup with high gamma (3.0), which emphasized hard-to-classify examples, and low alpha (0.25), suggesting that focusing on difficult cases was more effective than reweighting classes. In contrast, the worst result used the same gamma but the highest alpha, reinforcing this pattern.

These results suggest that Focal Loss performed best for our dataset when it emphasizes challenging examples rather than attempting to balance class frequencies. This highlights the importance of aligning loss function design with dataset characteristics. In our case, tuning alpha and gamma was essential for strong generalization, underscoring that loss configuration plays a role as critical as model architecture.

Among the models tested, the highlighted configuration offered the best balance. It achieved the highest recall and third-highest F1 score, with strong precision. The slight drop in precision suggests it favored recall by classifying more borderline cases as spam. This model used a higher embedding dimension and a low learning rate, contributing to its ability to capture subtle patterns and converge stably. With more parameters than other configurations, its strong performance may reflect increased representational

Parameter	Value
Model Configuration	
Embedding Dimension (`embed_dim`)	256
Feedforward Dimension (`ff_dim`)	128
Number of Attention Heads (`num_heads`)	4
Dropout Rate (`dropout`)	0.0
Batch Size (`batch_size`)	32
Learning Rate (`learning_rate`)	1e-4
Model Parameters	1,737,601
Best Performance (Epoch 9)	
F1 Score	0.9360
Precision	0.9456
Recall	0.9267
Validation Loss	0.0050
Training Loss	0.0004

TABLE V: Best configuration and performance using Focal Loss ($\alpha = 0.25$, $\gamma = 3.0$).

capacity. It also reached peak performance by epoch 6, indicating relatively fast convergence.

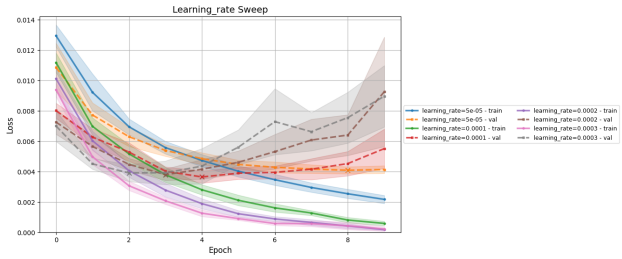

Fig. 1: Learning Rate Validation Curve using Focal Loss with $\alpha = 0.25$, $\gamma = 3.0$.

1) Best Model Analysis: From our loss function comparison, it is apparent that the model with focal loss performed the best with respect to a balance between recall, F1, and precision scores. While several learning rates performed well in terms of raw loss values, the most stable validation curve came from the smallest learning rate, 5e-5. It maintained low variance across folds and a consistent downward trend, suggesting strong generalization. In contrast, 0.0002 reached lower loss values earlier but showed signs of instability in later epochs. 0.0003 dropped quickly but exhibited high variance and rising loss near the end of training, pointing to potential overfitting.

This behavior reflects the core challenge of spam classification, where the minority class is both sparse and noisy. Higher learning rates may skip over subtle patterns in such data, while smaller ones allow the model to adapt more carefully to difficult or ambiguous examples. These trends reinforce that validation stability is just as important as fast convergence, and that conservative learning rates can improve generalization in noisy, imbalanced tasks like ours.

Under Focal Loss, the best performance often came from models with larger embedding dimensions, though this trend was not consistent. While higher-dimensional

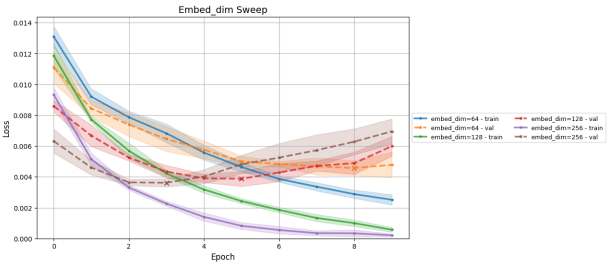

Fig. 2: Embedding Dimension Validation Curve using Focal Loss with $\alpha = 0.25$, $\gamma = 3.0$.

embeddings can provide richer representations, they also led to greater variance in validation loss, especially in later epochs, indicating a higher risk of overfitting in noisy, imbalanced data.

Notably, the model with embed_dim = 64 achieved the most stable validation loss and best generalization by epoch 10, despite its lower capacity. This suggests that a compact embedding space may act as a regularizer, encouraging the model to focus on broader patterns rather than rare noise. The narrow gap between training and validation loss supports this view.

These findings highlight that optimal performance depends on matching model complexity to data characteristics. Observing learning dynamics across configurations offers insight beyond what loss values alone can reveal, challenging the assumption that larger models are always better.

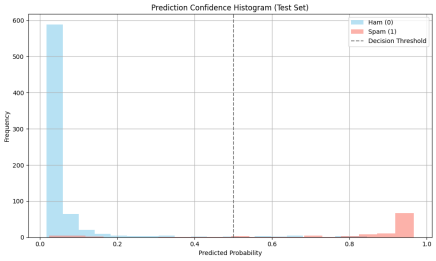

Fig. 3: embed_dim = 256

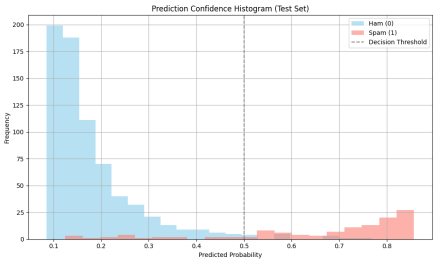

Fig. 4: embed_dim = 64

The prediction confidence histograms show that the

64-dimensional embedding model generalizes better, with spam and ham predictions more cleanly separated around the decision threshold. In contrast, the 256-dimensional model pushes nearly all outputs, including spam, toward low probabilities, showing extreme overconfidence in predicting ham. This suggests the larger architecture overfit to the majority class, causing it to suppress spam signals entirely. As a result, spam messages are misclassified with high confidence, making them harder to recover, even with threshold tuning.

D. Tokenization Analysis

Tokenization is crucial in SMS spam classification, where informal language, misspellings, and noise are common. Different tokenization strategies determine how well the model captures patterns while handling linguistic variation and obfuscation. Testing multiple approaches helps balance preserving information against removing noise.

As such, we analyzed the effect of different tokenization strategies on spam classification performance. The tested strategies included `default`, `minimal`, `no_special`, `raw`, `stemmed`, and `wordpiece` tokenizations. A detailed overview of these methods is provided in Section III-B1.

Among the strategies, `wordpiece` achieved the highest test accuracy (98.69%), with spam precision of 0.99 and recall of 0.90. Its ability to decompose rare or misspelled words into subword units likely improved generalization to unseen spam variations. The `minimal` tokenizer also performed well (97.85% accuracy), preserving more original text while maintaining strong spam detection.

The `default` tokenizer achieved slightly lower performance (94.98%), likely due to over-cleaning, which removed useful punctuation cues, lowering spam precision (0.75). The `raw` and `stemmed` pipelines showed similar accuracies (96.54% and 96.77%), with stemming offering limited benefit and occasionally distorting word meanings.

Overall, these results show that more sophisticated tokenization, such as subword modeling with `wordpiece`, enhances spam detection by improving robustness to noisy and adversarial input common in real-world messaging.

E. Adversarial Robustness Analysis

The justification for conducting this adversarial analysis was to evaluate the model's resilience to common evasion techniques used by spammers, ensuring its practical utility in real-world scenarios where adversaries may deliberately modify messages to evade detection.

As such, to assess the robustness of the spam classification model against adversarial attacks, we constructed adversarial test cases through targeted text perturbations. We categorized these into three groups: **light adversarial spam**, involving minor obfuscations like character substitutions (e.g., changing "Win a free iPhone now!" to "W1n a fr33 iPh0ne n0w!"); **extreme adversarial spam**, featuring aggressive lexical manipulations, noisy formatting, and fake URLs (e.g., altering the original message into "!!! FrEEEEE iPHonE. g3t n0w www.claimwinnings[dot]co"); and **adversarial ham**, legitimate messages rephrased to mimic spam through exaggerated urgency and formatting (e.g., converting "Hey, are you coming to dinner tonight?" into "Dinner Reminder!!! RSVP NOW for Tonight!").

Figure 5 visualizes the model's confidence scores for representative adversarial messages. The model maintained strong precision against both **light** and **extreme adversarial spam**, though recall varied (40% and 80%, respectively), indicating that the model is generally resilient but can still be susceptible to subtler text obfuscations. Notably, it demonstrated exceptional resistance to false positives, confidently classifying adversarial ham as legitimate (confidence scores ranging between 0.01 and 0.02), reflecting robustness in distinguishing aggressively phrased legitimate messages from actual spam.

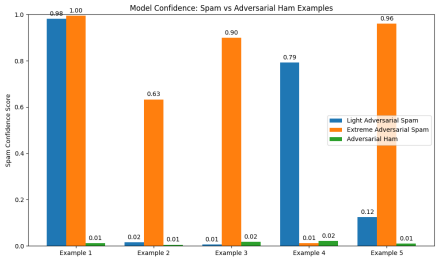

Fig. 5: Model confidence scores for examples of adversarially perturbed spam and ham messages.

V. Conclusion

In conclusion, the Transformer based deep learning approach to solving the spam classification problem has proved effective. However, we note that our model has found various ways to learn the distinction between spam and non spam that was not as nuanced as we would hope. When carefully crafting messages that are not obvious spam, it is very easy to trick our model due to its lack of inherent understanding of natural language. However, if we were to greatly expand the size of our model or use a pre-trained large language model as a starting point, we would likely have much better results when testing these adversarial examples.

However, for many use cases, we do not need to be perfect in our predictions, but instead just provide a warning to users so they may take a second look at their texts. Due to our model's lightweight design, it is much more fit for distribution embedded in the average texting service than a large language model. Our model offers both spam prediction, how confident it is in its decision, and key flagged words, allowing users to make informed decisions.

REFERENCES

[1] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz, "A bayesian approach to filtering junk e-mail," in *AAAI-98 Workshop on Learning for Text Categorization*, vol. 62, 1998, pp. 98–105.

[2] T. S. Guzella and W. M. Caminhas, "A review of machine learning approaches to spam filtering," *Expert Systems with Applications*, vol. 36, no. 7, pp. 10 206–10 222, 2009.

[3] E. Blanzieri and A. Bryl, "A survey of learning-based techniques of email spam filtering," *Artificial Intelligence Review*, vol. 29, no. 1, pp. 63–92, 2008.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.

[5] T. A. Almeida, J. M. G. Hidalgo, and A. Yamakami, "Contributions to the study of SMS spam filtering: new collection and results," in *Proceedings of the 11th ACM symposium on Document engineering*. Mountain View California USA: ACM, Sep. 2011, pp. 259–262. [Online]. Available: https://dl.acm.org/doi/10.1145/2034691.2034742