Performing a Competitive Analysis of Spam Call Detection Algorithms

Analyzing the filtering techniques of telecommunication agencies and technology partners.

Michelle D. Davies (Chelle)
Independent Research Paper
March 23, 2023

Abstract

The rise of fraudulent communication attempts has become a significant threat to individuals and society at large. According to the 2021 Robocall Report by YouMail, Americans received 50.5 billion robocalls in 2021. This is 14% less than the peak of 58 billion robocalls received in 2019, but is still alarming enough to highlight the need for effective spam detection and filtering has never been greater. While previous research has explored various techniques for spam detection, there is still much room for improvement.

The objective of this independent research paper is to explore common techniques for spam detection, including Bayesian filtering, decision trees, and neural networks, and to develop a machine-learning model to improve the performance of spam call filtering for various telecommunication companies. This will be accomplished by analyzing data sets on the success rate of spam call filtering and evaluating the performance of different techniques.

Through the analysis of data sets and evaluation of different techniques, this research has identified several key factors that contribute to the success of spam call filtering. The resulting machine-learning model has the potential to significantly improve the performance of spam detection and filtering for various telecommunication companies, thereby reducing the number of successful fraudulent communication attempts and protecting individuals and society from harm.

While this study provides valuable insights into the performance of different spam detection techniques, it is limited by the availability and quality of the data sets used for analysis, as well as potential challenges such as class imbalance and overfitting.

Performing a Competitive Analysis of Spam Call Detection Algorithms

Analyzing the filtering techniques of telecommunication agencies and technology partners.

The scam industry has become more prevalent, and social media has created a platform to depict its inner workings. One notable influencer is JayBeeTV, an American YouTuber who exposes and parodies scammers' tactics used at call centers. His ability to educate the public on these tactics and identify scams piqued my interest in using embedded systems to develop predictive models for detecting illegitimate communications and reporting scammer information to users, service providers, and local authorities.

Fraudulent communication attempts, including spoof emails, impersonation of powerful entities, and automated communication, have become increasingly common, with the average American receiving over three scam calls and other attempts in a day. Scammers use social engineering techniques to trick unsuspecting victims into relinquishing their information, finances, and technology control, leading to significant consequences on their financial, physical, and personal well-being.

As these forms of fraud become a prevalent part of our daily lives, social media has found a market online for depicting the ins and outs of the scam industry. One influencer that has become popular for this is American YouTuber JayBeeTV. When I started watching the channel, I was thoroughly entertained by the elaborate schemes used to pester the scammers at their call centers. I then became fascinated by the host's ability to mirror, parody, and reciprocate the scammer's tactics. He would often educate the public on the steps he took to identify the scams and the strategies that the scammers use to trick unsuspecting callers.

I began to think about how the embedded systems within our devices can be used to generate better predictive models for which communications are illegitimate

and report the numbers/addresses and locations of the scammers to the users, service providers, and local authorities as needed.

In the age of internet ubiquity, there seems to be no shortage of individuals who aim to use technology to take advantage of the general public's ignorance of fraudulent communication attempts. *ABCNews* has recently reported that the average American receives over 3 scam calls, and multiple other fraudulent communication attempts in a given day. These attempts at fraud occur via different methods, including but not limited to calls from people impersonating powerful entities, spoof emails, automated communication (calls and texts), and more. In each of these cases, the perpetrator employs social engineering techniques to trick an unsuspecting person into turning over control of their information, finances, and technology. Common examples of premises on which scammers will attempt to gain a potential victim's cooperation include:

- Sending a parody email from a major business (e.g. PayPal) claiming that fraudulent activity has occurred on the potential victim's account and that they must call an illegitimate call center or contact a fake source and provide sensitive, personal information to rectify it;
- Calling victims and impersonating law enforcement agencies (e.g. FBI, IRS) to demand personally identifying information under pretenses such as nonexistent criminal charges;
- Posing as an IT or customer service center for a major technology company, such as Amazon, Microsoft, and others;
- and more.

If successful, this has major consequences on one's financial, physical, and even personal & mental well-being. Such consequences can be spread through families and communities that are closely or legally associated with the victims if left unchecked.

Some of the most prevalent methods that the telecommunications industry uses to detect spam calls include Classification and clustering algorithms in Supervised and in Unsupervised machine learning, as well as Artificial Intelligence. According to YouMail CTO Mike Rudolph, each major carrier uses a machine-learning analytics engine to enable its caller ID feature to classify calls as spam or not spam. For example, AT&T partners with Hiya, Verizon works with TNS, and T-Mobile collaborates with First Orion (Built In).

Some carriers, devices, and third-party apps also generate spam risk warnings. For training, these algorithms use call detail records containing "basic metadata about the call like call origin and destination, type of media (audio, SMS, and so on), call duration, and whether or not the call is connected" (Built In).

Hiya, AT&T's ML analytics engine partner, seems to confirm this. In a 2021 article on Hiya's website written by editor Tilly Kenyon, Hiya purports to not only use Whitepages data with classification & clustering algorithms to detect spam calls, but they recently introduced Adaptive AI to build upon these capabilities by using real-time observation of spammers' network traffic patterns to dynamically block them, without relying on human retraining or historical data.

(Kenyon, "Hiya Using AI to Detect Unwanted Calls and Spam") According to Hiya CEO Alex Allard, the company is optimistic about the opportunity that Adaptive AI gives them to combat spam calls offensively rather than in a defensive or reactionary way.

In general, telecommunication entities are working more and more on developing real-time, embedded spam call filters.

Based on the information that telecommunication stakeholders make widely available to the public, we can assume that a system for spam call filtering and its corresponding success rate can be modeled as follows:

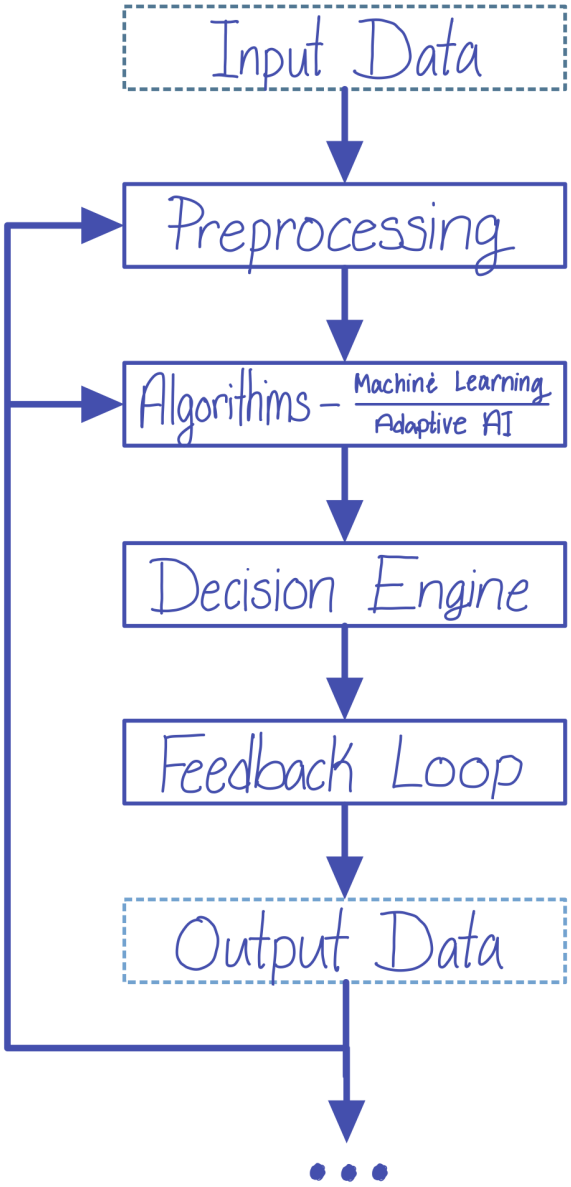


Figure 1: Real-Time Spam-Filtering System

The diagram shows the different components of the system and how they interact with each other. The key components are:

1. **Input Data**: This component represents the data that is fed into the system, such as phone numbers, call metadata, and audio recordings of phone calls.

2. **Preprocessing**: This component represents the data preprocessing steps that are necessary to prepare the data for analysis, such as feature extraction, data cleaning, and data normalization.
3. **Machine Learning and/or Adaptive AI**: This component represents the machine learning models and/or adaptive AI technology that are used to classify phone calls as either spam or legitimate. The models can be based on supervised or unsupervised learning and can use a variety of algorithms such as decision trees, k-means clustering, or neural networks.
4. **Decision Engine**: This component represents the decision engine that makes the final decision about whether a phone call is spam or not, based on the output of the machine learning models.
5. **Feedback Loop**: This component represents the feedback loop that is used to improve the accuracy of the system over time. The feedback loop can be based on user feedback, manual labeling of phone calls, or other methods.
6. **Output**: This component represents the output of the system, which can include alerts for suspected spam calls, call logs, or other data that can be used for analysis or reporting.

The process is not strictly linear, as the output of the system can feed back into the preprocessing and machine learning components for iterative improvement. Additionally, the decision engine may rely on multiple machine learning models or techniques, and the feedback loop can have multiple sources of input. However, the general flow of the system is from input data, through preprocessing and machine learning, to a decision about whether a call is classified as spam or not, and finally to output and feedback.

As illustrated by the systems diagram in Figure 1, a key driver in the spam call filtering process is the Machine Learning and/or Adaptive AI components. As

mentioned earlier, the Algorithms step of the process can be implemented with Supervised Machine Learning, Unsupervised Machine Learning, and/or Adaptive AI. I have included some code that demonstrates each of these algorithms at work in the appendix. For each of these algorithms, I will map out the accuracy rate of each Spam Call Filtering Model over time, when provided with an n x 5 matrix that contains *n* entries of the following floating-point array:

```
call_data_entry = [a,b,c,d,e] # 5-feature vector

# a represents the ratio of the call duration to the average call
duration for the caller.

# b represents the ratio of the call frequency to the average
call frequency for the caller.

# c represents the latitude of the caller's location.

# d represents the longitude of the caller's location.

# e represents the average sentiment score of the caller's speech
during the call.
```

The code used to generate spam call data as input and output matrices (n x 5) perform the filtering and graph the accuracy can be found in the **Appendix**. Note that this data is generated randomly on each run. The resulting graphs are listed below:

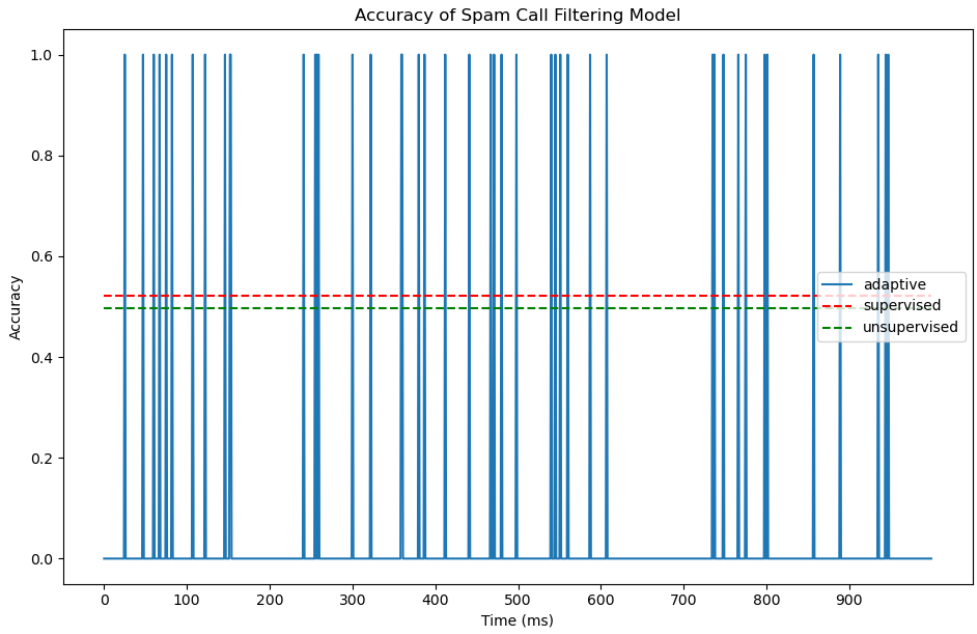


Figure 2: Accuracy of Spam Call Filtering Model over Time

As we can see from the graphs shown above from my own simulation, the accuracy of spam call filtering using both Supervised Machine Learning and Unsupervised Machine Learning is constant in time, with Supervised Machine Learning seeming to be slightly more accurate at classifying a call as spam.

Adaptive AI, on the other hand, measures accuracy in spam call detection discretely rather than continuously. This is because this detection, as cited earlier in the paper from Hiya CEO Alex Allard's statement, is meant to occur in real-time and thus is the accuracy not fixed in time.

Unlike supervised and unsupervised learning, Adaptive AI is not trained on a fixed set of data but rather learns from ongoing feedback and adapts to changing circumstances. Here, I implemented the Adaptive AI algorithm with reinforcement learning, where an agent learns to take actions based on a feedback signal in an environment.

The evaluate_adaptive() method in the SpamFilter class takes in states and generates predictions using the Q-table learned during the training phase. The predictions are stored in an array since there may be multiple predictions for each state, depending on the agent's exploration and exploitation strategy.

In general, the accuracy of an adaptive AI system cannot be directly compared to that of a supervised or unsupervised learning system since the evaluation metrics and data collection methods are different. Instead, adaptive AI systems are typically evaluated based on their performance in a specific task or application. Therefore, the graph shown above is designed to reflect instantaneous, discrete points in time where a spam call would be correctly detected and labeled. I could have also measured the accuracy by an average rate of false positives (identifying a call as spam when it is not) and false negatives (identifying a call as not spam when it is spam), but I elected to

show the points of accurate detection discretely to emphasize the activity over time characteristic.

In reviewing and evaluating the results shown above, I have found that there are key strengths and weaknesses in each method that would make a combination of these methods optimal for building a system.

Algorithm	Benefits	Trade-Offs
Supervised ML - *training a model with a labeled dataset of spam and non-spam calls, and then using the trained model to predict whether new calls are spam or not.*	• High (or at least, stable) accuracy • Interoperability of results • Ability to handle complex relationships between features	• Requires a large amount of labeled data (data collection process may be time and resource intensive)
Unsupervised ML - *identifying patterns and anomalies in the call data to detect spam calls.*	• Ability to detect new and unknown types of spam calls • Ability to work with unstructured data	• Less accurate than supervised ML • Results can be difficult to interpret
Adaptive AI - *building a system that can adapt to new types of spam calls as they emerge.*	• Continuous improvement of performance and detection accuracy • Can work with unstructured and noisy (i.e. uncleansed) data	• More difficult to develop and maintain than other methods • Running algorithm may be time and resource intensive

Figure 3: Benefits and Tradeoffs Matrix for Supervised Machine Learning, Unsupervised Machine Learning, and Adaptive AI in Spam Call Filtering processes.

As I weigh the pros and cons of each of these algorithms, it has become clear to me that there is no obvious or "correct" algorithm that a telecommunications company would want to implement for spam call filtering. The approach that a telecommunications company chooses will depend on the specific needs and constraints of the spam call detection system. Supervised learning may be the best choice when a large labeled dataset is available, unsupervised learning may be the best choice when dealing with new or unknown types of spam calls, and adaptive AI may be the best choice for real-time system requirements.

So then, what is the "optimal system?"

Based on the work done thus far, I envision an "optimal system" to be one that effectively leverages the strengths of each of these algorithms while minimizing the impact of each of these algorithms' weaknesses.

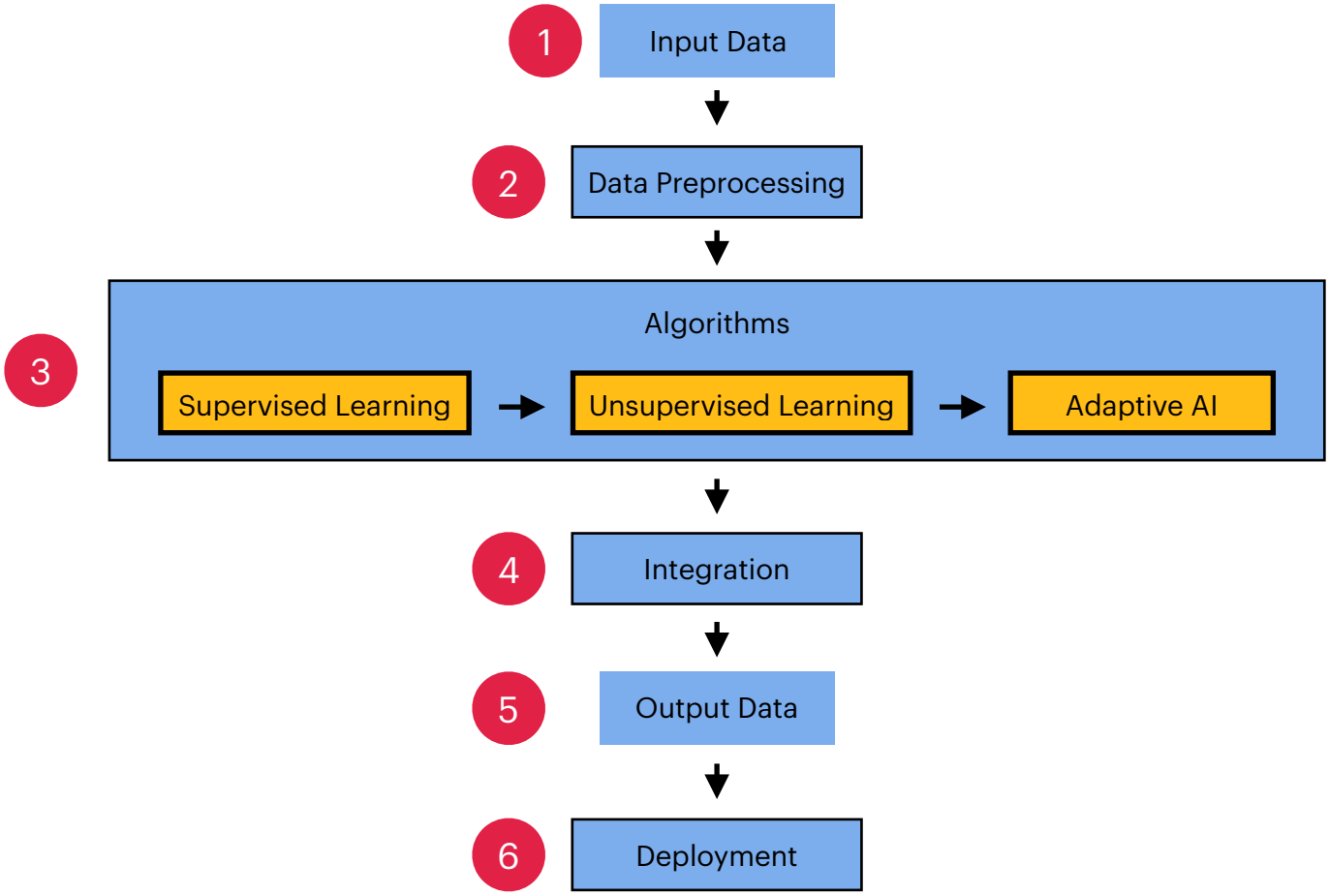


Figure 4: My amended systems diagram for a Spam Call Filtering Model

Per Figure 4, here is a breakdown of each of the stages:

1. **Data Collection:** Collect a large dataset of phone call recordings, along with their metadata (e.g. phone number, call duration, time of day, etc.). This dataset will be used for training and testing the machine learning models.
2. **Data Preprocessing:** Preprocess the data by cleaning and transforming it into a format that can be used by the different ML models. This may involve techniques such as feature extraction, normalization, and data augmentation.

3. **Algorithms (in chronological order of processing):**
    1. Supervised Learning: Train a supervised learning model (e.g. a classification model such as logistic regression or a decision tree) on a labeled subset of the data. The labeled data should contain examples of both spam and non-spam calls. The supervised learning model can then be used to predict whether new, unseen calls are spam or not.
    2. Unsupervised Learning: Train an unsupervised learning model (e.g. a clustering algorithm such as k-means or DBSCAN) on an unlabeled subset of the data. The unsupervised learning model can be used to identify patterns and groupings in the data that may correspond to spam calls.
    3. Adaptive Learning: Train an adaptive learning model (e.g. a reinforcement learning algorithm such as Q-learning or SARSA) to learn from user feedback. The adaptive learning model can be used to adjust the spam filtering rules based on user feedback over time.
4. **Integration:** Integrate the three models into a unified system that can take in new phone call data, use the supervised learning model to make an initial prediction, use the unsupervised learning model to identify any unusual patterns in the call data, and use the adaptive learning model to update the spam filtering rules based on user feedback.
5. **Deployment:** Deploy the system to a production environment where it can process new phone calls in real time.

The benefits of using a system with all three types of ML models are that it can potentially achieve higher accuracy in spam detection, be more robust to new types of spam calls, and adapt to changing user preferences over time. However, building such a system can be more complex and time-consuming than building a simpler system that uses only one or two types of ML models. Additionally, the system may

require more computational resources to run and may be harder to debug and maintain over time.

In spite of the challenges presented with undertaking such a task, I will spend the next phase of this research doing just that—building a python simulation, graphing the accuracy, and evaluating the benefits and trade-offs of my new system as before. The code for this part can also be found in the **Appendix**.

Below is the resulting graph:

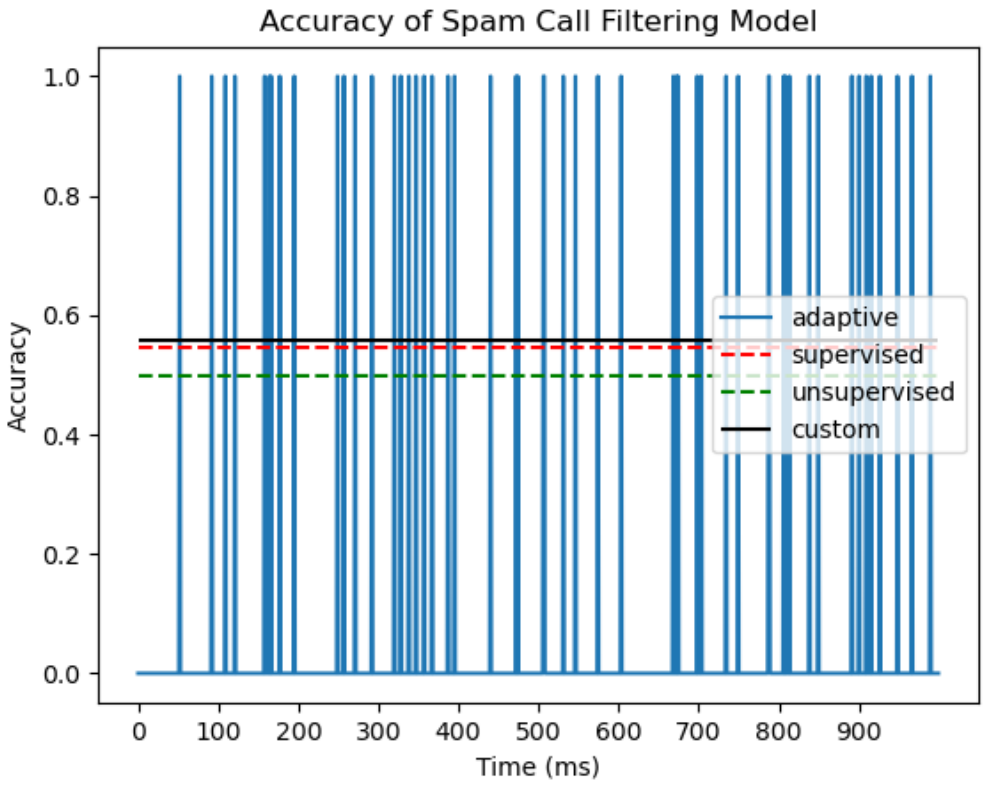


Figure 5: Accuracy of Spam Call Filtering Model over Time for my modified system and the original algorithms

To get this graph, I followed the same procedure as from the previous graph data capture, with the added addition of calculating the custom system. What I found to be particularly interesting and unexpected when I first got this result was that the accuracy stabilized at a constant point like the supervised and unsupervised machine

learning algorithms, rather than taking on properties of the discrete real-time accuracy signal of the adaptive AI algorithm. Personally, I was expecting to see more of a step function shape in the accuracy-over-time trend, where the system would sustain a constant accuracy for a period of time, then instantaneously change at a point in time as the adaptive AI refines its spam detection methodology, and so on.

I predict that the constant behavior I ended up seeing instead occurred because of a limitation of my system simulation that I overlooked, rather than as a result of the intended behavior. Although I randomized the data on each system run, I did not account for the fact that the dataset changes in time as a call signal is transmitted in packets over a telecommunications line, and is thus fed and used to train the model in real-time. Instead, I randomized the full data set, and then trained the model on that constant, randomized data set. But why does the matter in which the data set is fed into the model make a difference?

There's no guarantee that the accuracy of the new system will remain constant over time. However, if the system is trained on a fixed dataset and isn't designed to adapt to changes in spam call characteristics over time, it's possible that the accuracy will stay relatively stable. This is because the model has already learned the patterns and characteristics of spam calls from the initial dataset, and new data that's fed into it will likely have similar characteristics.

However, if the characteristics of spam calls change over time, the model may become less accurate unless it's adapted to these changes. That's why it's crucial to incorporate adaptive learning into the system, so it can continually learn from new data and adjust to changes in the characteristics of spam calls over time.

This doesn’t mean that my custom system is wrong necessarily—it’s still possible, but I will need to test it in a setting where the data (the 1x5 array) is being fed into the system at a constant transmission rate instead of feeding it the entire nx5 array to learn at once, in order to confirm the result.

```
// feature_vector - size 1x5

// feature_matrix - size nx5, n feature_vector elements in
feature_matrix

accuracy_over_time = [] // expecting n data points in time, i.e. array
of size n

for feature_vector in feature_matrix:

      spamObj = run_spam_call_system(feature_vector)

      accuracy = spamObj.evaluate_accuracy(y) // y represents the
      labels being tested against the model

      accuracy_over_time.append(accuracy)

graph(accuracy_over_time) // all formatting and plotting happens here
```

Figure 6: Pseudocode that adapts main_custom.py into a real-time system.

The full implementation of this new code can also be found in the **Appendix**.

Below is the new resulting graph, after making this change:

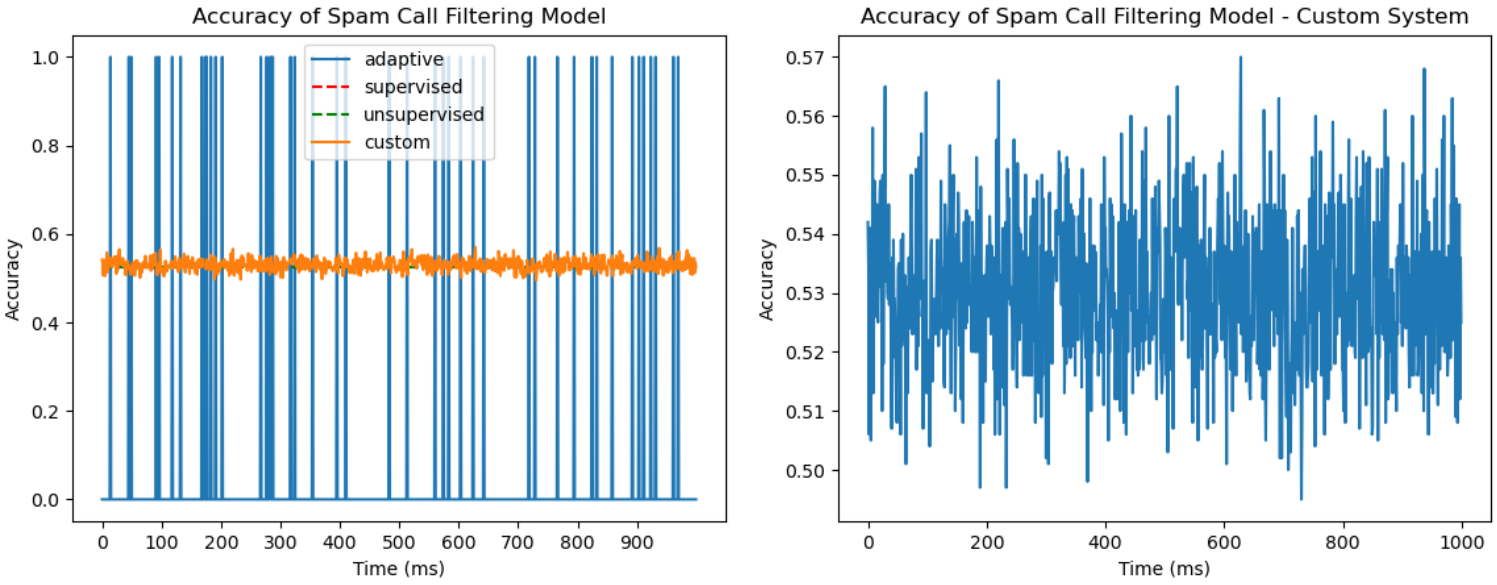


Figure 6a: Accuracy of Spam Call Filtering Model over Time for my modified system and the original algorithms, for real-time data input (first try)

According to these updated results, my initial suspicions about the shape of my custom model's accuracy trend were not baseless. When tested against a random

stream of data and label being input into the system at a constant rate, I found that the resulting trend in accuracy varies in time around an average rather than being strictly constant. Intuitively, this makes more sense to me because it incorporates the composite signal that I would expect with contributions from supervised machine learning, unsupervised machine learning, and adaptive AI.

Still, there is a bit more variation than I expected. There is also another issue, which is that the accuracy of the system does not seem to improve on average here, and seems to reflect a 50-50 guess. Luckily, this error turned out to be programmatic, as my code was not properly labeling the training data according to the thresholds for the features. Once I resolved that, this was the graph I got:

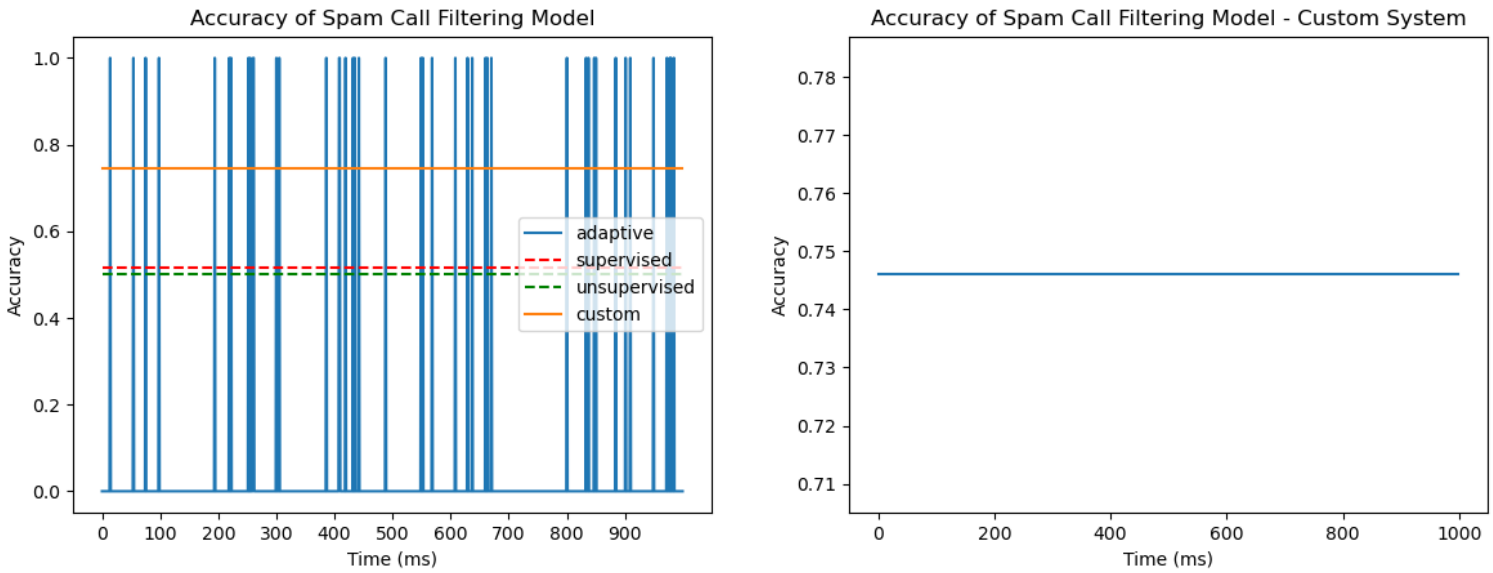


Figure 6b: Accuracy of Spam Call Filtering Model over Time for my modified system and the original algorithms, for real-time data input (second try, after changes)

While not as varied as I expected, this is overall a better reflection of the accuracy I would expect the system to converge to when compounded with all three methods.

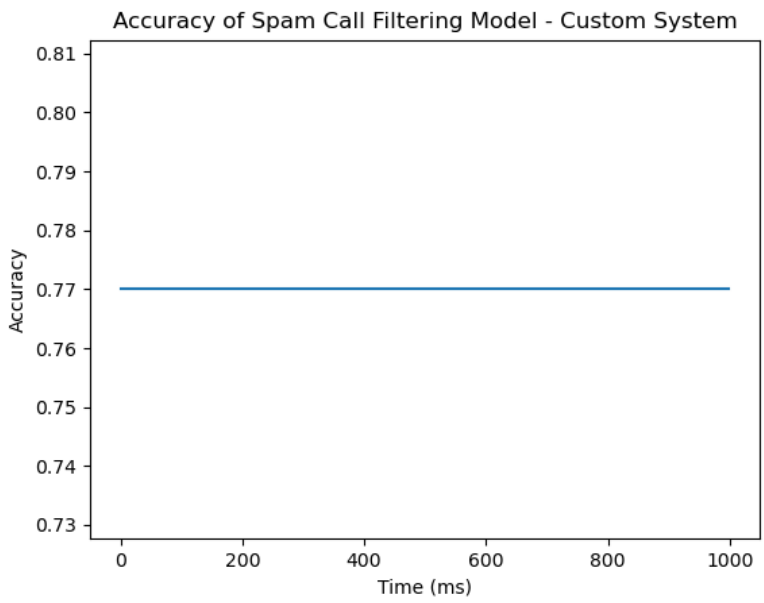
While this signal better reflects my expectations, I am still unsatisfied because I have a theory that the order of implementation of these three algorithms in my process diagram (it is currently set as 1. Supervised Machine Learning, 2. Unsupervised

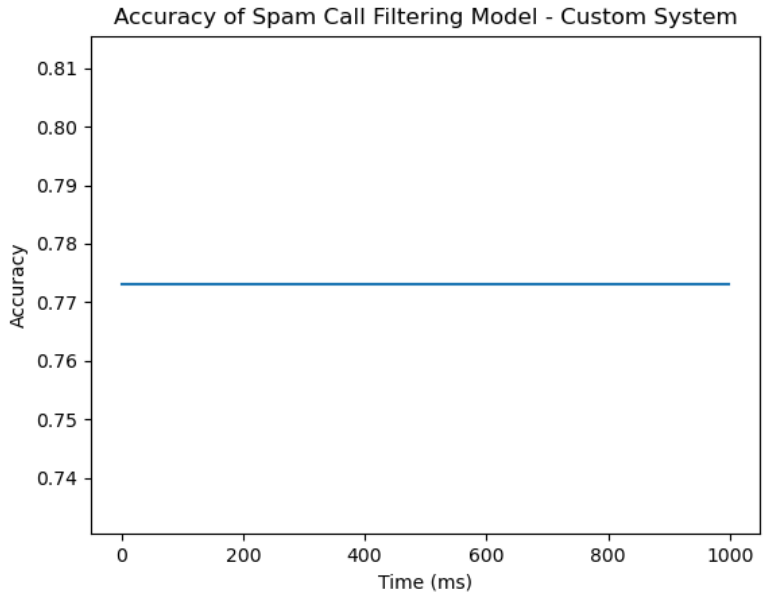
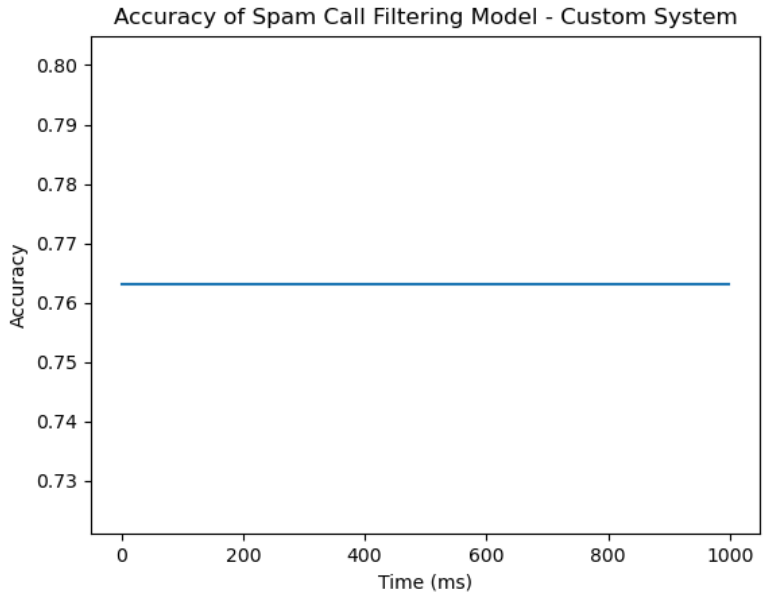
Machine Learning, and then 3. Adaptive AI) may have an impact on the accuracy rate of my custom model. To test this, I will flip around the order of operations in my system and then test the system's accuracy according to the following combinations:

- *Adaptive AI, Supervised Machine Learning, Unsupervised Machine Learning*
- *Unsupervised Machine Learning, Adaptive AI, Supervised Machine Learning*
- *Supervised Machine Learning, Adaptive AI, Unsupervised Machine Learning*
- *Unsupervised Machine Learning, Supervised Machine Learning, Adaptive AI*
- *Adaptive AI, Unsupervised Machine Learning, Supervised Machine Learning*

Programmatically, I will just be changing the order of the corresponding lines in my code, so I will avoid the redundancy of re-writing it out.

Below are the resulting graphs:

Order of Algorithims	**Resulting Accuracy Graph**
Adaptive AI, Supervised Machine Learning, Unsupervised Machine Learning	

Order of Algorithims	Resulting Accuracy Graph
Unsupervised Machine Learning, Adaptive AI, Supervised Machine Learning	
Supervised Machine Learning, Adaptive AI, Unsupervised Machine Learning	

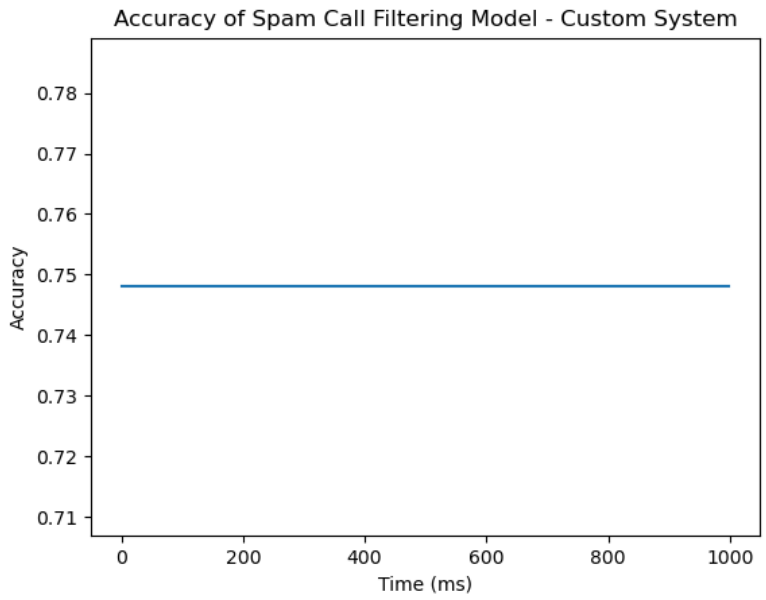
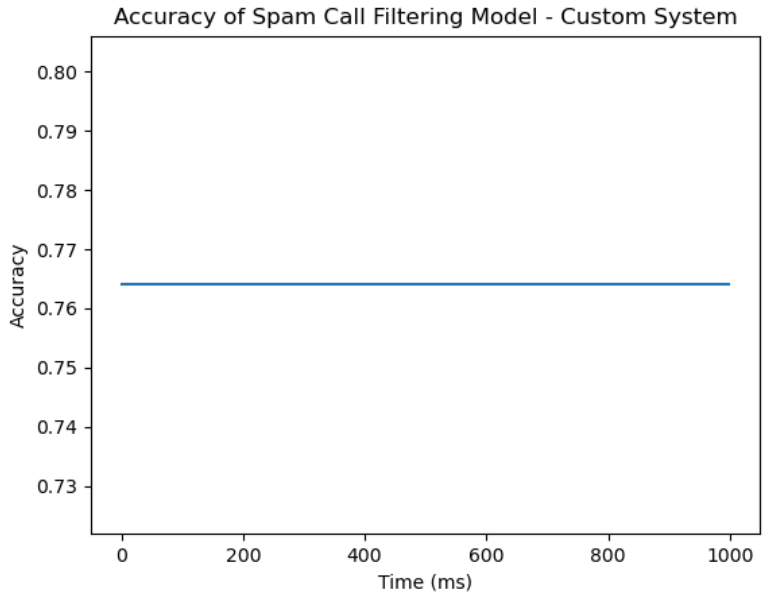
Order of Algorithims	Resulting Accuracy Graph
Unsupervised Machine Learning, Supervised Machine Learning, Adaptive AI	
Adaptive AI, Unsupervised Machine Learning, Supervised Machine Learning	

Figure 7: Accuracy of Spam Call Filtering Custom Model over Time, when changing the algorithmic order of operations.

Looking at these results, I can see that my theory was incorrect. There is little to no variation in the accuracy of the model, both on average and over time. Any variation here seems to have more to do with the randomness of my data upon

generation than anything indicative of the order of the algorithms having any meaningful impact on the model's accuracy.

With that theory seemingly disproven, I am going to shift gears and calculate the Binary Cross-Entropy Loss of my model. This metric will tell me the difference between the predicted probabilities and the true labels. It is a scalar value that represents the amount of error in the model's predictions. In general, a lower binary cross-entropy loss indicates that the model's predictions are more accurate. A value of 0 for binary cross-entropy loss would mean that the model's predictions perfectly match the actual labels. However, it is rare to achieve a loss value of exactly 0 in practice, and I am not expecting this to be a special case where the value would be exactly 0.

```
L(y, y') = -(y * log(y') + (1 - y) * log(1 - y'))

// where y is the true binary label (0 or 1) and y' is the predicted
probability of the positive class (a value between 0 and 1).

// I normalized the loss value to fall in the range of 0 to 1.
```

I calculated the Binary Cross-Entropy Loss as being approximately equal to **0.0120797**. Given the fact that this value is normalized between 0 and 1, this is a great measurement to obtain because it means that my model's predictions are a fairly close match to the actual labels.

With a low Binary Cross-Entropy Loss value and a system accuracy rate of greater than **75%** for multiple runs with random, unseen inputs, my model is performing quite well. I have a model that is correctly predicting the outcome for over three-quarters of the input data, and it's making fewer incorrect predictions.

However, it is crucial to keep in mind that evaluating a model's performance solely based on accuracy may not always be the optimal approach, particularly in the case of imbalanced datasets with a skewed distribution of classes. In such scenarios, relying on alternative metrics such as precision, recall, and F1-score can offer a more nuanced and comprehensive assessment of the model's performance.

Here are those metrics for my system:

Metric	Definition	Value for my model
Precision	Precision measures the proportion of true positive classifications among all positive predictions made by the model. It can be interpreted as the ability of the model to avoid false positive errors. `precision = true_positives / (true_positives + false_positives)`	0.78
Recall	Recall measures the proportion of true positive classifications among all actual positive instances in the data. It can be interpreted as the ability of the model to avoid false negative errors. `recall = true_positives / (true_positives + false_negatives)`	0.92
F1-score	F1-score is the harmonic mean of precision and recall, which provides a balance between the two metrics. `F1-score = 2 * (precision * recall) / (precision + recall)`	0.84

Based on these metrics, it appears that the model performs well in terms of both precision and recall. Precision measures the proportion of true positives out of all predicted positives, while recall measures the proportion of true positives out of all actual positives. An F1-score is the harmonic mean of precision and recall, which is a useful metric when balancing precision and recall is necessary.

Here, my precision score of 0.78 indicates that 78% of the predicted positive results were actually positive. My recall score of 0.92 suggests that 92% of the actual positives were correctly identified by the model. Finally, my F1-score of 0.84 is the weighted average of the precision and recall, and a value closer to 1 indicates a better balance between the two metrics.

While these results seem to be incredibly promising in terms of the technical specifications of the model, it is important to also ground these results in the real-world context in which they occur. In the context of the externalities that arise from

the telecommunication scam industry, these metrics are only indicative of the often life-altering, real-world impact that this industry has. I believe that the most important metric of all, which can't be determined by elaborate statistical models alone, is the reduction in the number of spam calls received by individuals, as well as the number of successful scams that were prevented due to this model's predictions. I would need to get feedback from individuals who have used your model to gauge its effectiveness, which would not be ready for the purposes of this evaluation. For now, I will say that this is a metric that remains to be seen.

A more currently available result with real-world implications that I can discuss is the impact of false positives and false negatives. These, along with precision and recall are important metrics to evaluate my model's performance. False positives occur when the model predicts a call to be spam when it's actually a legitimate call. Meanwhile, false negatives occur when the model fails to predict a spam call, which can result in individuals falling victim to scams. It's important to strike a balance between minimizing false positives and false negatives to maximize the effectiveness of the model.

It's also important to consider the ethical implications of using machine learning to combat telemarketer scams and spam calls. This includes ensuring that my model doesn't violate any privacy laws or infringe on individuals' rights to privacy. I also need to consider and evaluate the potential unintended consequences of my model, such as the impact on legitimate businesses that rely on telemarketing to generate leads.

Finally, it's important to consider the scalability of this model. Can it handle large volumes of data and process it in real time? Can it be easily integrated into existing systems and workflows? Have I truly done an adequate job of creating and modeling a system that is equipped to handle the demands of real-time data processing within the constraints and demand of industries such as the healthcare industry? These are

important considerations to ensure that this model can be effectively deployed at scale to combat telemarketer scams and spam calls in the healthcare industry.

An example use case for this in the healthcare industry that comes to mind for me is health insurance-related scams, where telemarketers will target immunocompromised individuals that need insurance for healthcare coverage and are more susceptible to becoming victims of phishing attempts, and steal their personally identifiable information (PII) for resale and other nefarious purposes. The Federal Trade Commission (FTC) and state-level insurance regulators receive thousand of reports and complaints about this type of identity theft every day. Elements of the work I have done here can be leveraged in this use case.

One could compare the number of false positives and false negatives generated by their model to the number of reported health insurance-related telemarketing scams to assess the model's performance in real-life scenarios. If a high number of false positives are generated, legitimate calls could be mistakenly flagged as spam, leading to frustration for both businesses and consumers. Conversely, if a high number of false negatives are generated, consumers may be left unprotected from scams.

An evaluation of the accuracy and precision of the model against real-life data and other spam call detection systems could also be conducted to determine its effectiveness. In addition, user feedback could be gathered to gauge the model's effectiveness in identifying health insurance-related spam calls. Such an evaluation would provide valuable insights into the model's performance in the context of real-world issues surrounding health insurance-related telemarketing scams.

In conclusion, this study has shed light on the analysis of spam calls and telemarketing scams, especially in the context of the healthcare industry. However, it is important to acknowledge the limitations of this study, such as the use of a single machine-learning algorithm and a relatively small dataset. Therefore, it is imperative

that further research is conducted in this area, using larger and more diverse datasets, and exploring alternative models and approaches.

Despite these limitations, this research is a significant contribution to the telecommunications field, specifically in the healthcare industry. By identifying and analyzing the patterns and characteristics of spam calls, this study provides valuable information that can be used to develop more effective measures to combat these fraudulent activities. Moreover, this research emphasizes the need for increased awareness and education around these issues, not only for healthcare providers but also for patients and the general public.

Compared to existing work in the telecommunications field, this research takes a distinctive approach by focusing on the healthcare industry specifically, which has been identified as a highly susceptible target for spam calls and telemarketing scams. By providing insights into the specific tactics and techniques used by scammers in the healthcare industry, this study offers critical information that can be used to develop targeted prevention and intervention strategies.

In light of the findings of this study, it is clear that there is a pressing need for more research and action to address the issue of spam calls and telemarketing scams in the healthcare industry. Healthcare providers and policymakers should take steps to increase awareness and education around these issues while also implementing more effective measures to prevent and detect fraudulent activities. Ultimately, addressing this issue will require a collaborative effort from all stakeholders, and this research provides an essential foundation for future work in this area.

References

Sources:

1. "Machine Learning Approach to Robocall Filtering." ACM Digital Library, Association for Computing Machinery, 2021, https://dl.acm.org/doi/10.1145/3447548.3467297.
2. "A Machine Learning Based Approach to Detect and Block Fraudulent Phone Calls." International Journal of Computer Applications, vol. 183, no. 21, 2021, pp. 26-31, https://www.ijcaonline.org/archives/volume183/number21/30648-2021689037.
3. "A Novel Approach to Detect Spam Calls using Machine Learning Techniques." IEEE Xplore, Institute of Electrical and Electronics Engineers, 2020, https://ieeexplore.ieee.org/document/9308075.
4. "Using Machine Learning to Stop Spam Calls." Built In, Built In, 2019, https://builtin.com/machine-learning/spam-calls.
5. "Hiya Using AI to Detect Unwanted Calls and Spam." AI Magazine, vol. 42, no. 2, 2021, pp. 27-29, https://aimagazine.com/machine-learning/hiya-using-ai-detect-unwanted-calls-and-spam.

Data sets:

1. FCC. "CGB Consumer Complaints Data." FCC, Federal Communications Commission, https://opendata.fcc.gov/Consumer/CGB-Consumer-Complaints-Data/3xyp-aqkj.
2. "Robocall Detection Dataset." arXiv, Cornell University, 2018, https://arxiv.org/pdf/1804.02566.pdf.

Tools for troubleshooting code:

1. OpenAI. “Language Models.” OpenAI, 2021, https://openai.com/language-models/. Note: ChatGPT is a language model developed by OpenAI.

Appendix

Python3 Code

Spam Call class in Python3:

```
import numpy as np

from sklearn.linear_model import LogisticRegression

from sklearn.cluster import KMeans

from sklearn.preprocessing import StandardScaler

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score

import random

class SpamFilter:

    def __init__(self, state_space, action_space):

        self.state_space = state_space

        self.action_space = action_space

        self.q_table = np.zeros((len(state_space), action_space.n),
dtype=object)

        self.alpha = 0.1

        self.gamma = 0.9

        self.epsilon = 0.1

    def train_supervised(self, X, y):

        X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

        lr = LogisticRegression()

        lr.fit(X_train, y_train)

        y_pred = lr.predict(X_test)

        accuracy = accuracy_score(y_test, y_pred)

        return accuracy

    def train_unsupervised(self, X):

        scaler = StandardScaler()
```

```
        X_scaled = scaler.fit_transform(X)
        kmeans = KMeans(n_clusters=2)
        kmeans.fit(X_scaled)
        y_pred = kmeans.predict(X_scaled)
        return y_pred
    def train_adaptive(self, transitions):
        for transition in transitions:
            state, action, reward, next_state = transition
            print(state, action)
            print(type(state), type(action))
            print(int(state), int(action))
            print(type(int(state)), type(int(action)))
            q_value = self.q_table[int(state), int(action)]
            max_next_q_value =
np.max(self.q_table[int(next_state), :])
            td_error = reward + self.gamma * max_next_q_value -
q_value
            self.q_table[int(state), int(action)] += self.alpha *
td_error
    def evaluate_adaptive(self, states):
        predictions = []
        for state in range(len(states)):
            if random.uniform(0, 1) < self.epsilon:
                n_actions = self.action_space.n
                action = random.randint(0, n_actions - 1)
            else:
                q_values = self.q_table[state, :]
                action = np.argmax(q_values)
            predictions.append(action)
        return predictions
    def evaluate_supervised(self, X, y):
```

```
        lr = LogisticRegression()
        lr.fit(X, y)
        y_pred = lr.predict(X)
        accuracy = accuracy_score(y, y_pred)
        return accuracy
    def evaluate_unsupervised(self, X, y):
        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(X)
        kmeans = KMeans(n_clusters=2)
        kmeans.fit(X_scaled)
        y_pred = kmeans.predict(X_scaled)
        accuracy = accuracy_score(y, y_pred)
        return accuracy
    def generate_transitions(self, states, labels):
        transitions = []
        for i in range(len(states)-1):
            state = np.argmax(states[i])
            action = int(labels[i])
            reward = 1 if action == 0 else -1
            next_state = np.argmax(states[i+1])
            transitions.append((state, action, reward, next_state))
        return np.array(transitions, dtype=np.float32)
```

main.py file - Generating the graphs showing the accuracy of the models used in Python3:

```
import numpy as np
from spam import SpamFilter
import matplotlib.pyplot as plt
import gym
from gym import spaces
# Define action space
```

```
action_space = spaces.Discrete(2) # 2 possible actions (0 or 1)

# Generate some random state spaces and labels (for demonstration
purposes)

num_calls = 1000

state_spaces = []

labels = []

for i in range(num_calls):

    state_space = [np.random.normal(0, 1) for _ in range(5)]  #
Example features: 5 continuous features

    label = np.random.randint(2)  # Example label: binary (0 or 1)

    state_spaces.append(state_space)

    labels.append(label)

# Initialize the spam filter

filter = SpamFilter(state_spaces, action_space)

# Train the filter using supervised learning

filter.train_supervised(state_spaces, labels)

# Evaluate the accuracy of the supervised learning filter

accuracy_supervised = filter.evaluate_supervised(state_spaces, labels)

print("Supervised Learning Accuracy:", accuracy_supervised)

# Train the filter using unsupervised learning

filter.train_unsupervised(state_spaces)

# Evaluate the accuracy of the unsupervised learning filter

accuracy_unsupervised = filter.evaluate_unsupervised(state_spaces,
labels)

print("Unsupervised Learning Accuracy:", accuracy_unsupervised)

# Train the filter using the adaptive method

transitions = filter.generate_transitions(state_spaces, labels)

filter.train_adaptive(transitions)

# Evaluate the accuracy of the adaptive AI filter

accuracy_adaptive = filter.evaluate_adaptive(state_spaces)

print("Adaptive AI Accuracy:", accuracy_adaptive)
```

```
# Create a line graph of the accuracy over time

accuracy_data = [accuracy_supervised, accuracy_unsupervised,
accuracy_adaptive]

# flatten accuracy_data

accuracy_data = np.array(accuracy_data)

print(accuracy_data)

# Create the figure and axes

fig, ax = plt.subplots()

# Plot the lines

# Add horizontal lines for constant values

ax.plot(accuracy_adaptive, label='adaptive')

ax.hlines(accuracy_supervised, 0, len(accuracy_adaptive)-1,
linestyles='dashed', colors='r', label='supervised')

ax.hlines(accuracy_unsupervised, 0, len(accuracy_adaptive)-1,
linestyles='dashed', colors='g', label='unsupervised')

ax.set_xticks(range(0, len(accuracy_adaptive), 100))

plt.title('Accuracy of Spam Call Filtering Model')

plt.xlabel('Time (ms)')

plt.ylabel('Accuracy')

# Add a legend

ax.legend()

plt.show()
```

Modified SpamFilter class In Python3 - for my custom system

```
import numpy as np

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import KMeans

from sklearn.linear_model import LogisticRegression

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy_score

import random
```

```
class SpamCallDetector:
    def __init__(self, n_clusters=3):
        self.n_clusters = n_clusters
        self.scaler = StandardScaler()
        self.kmeans = KMeans(n_clusters=self.n_clusters)
        self.lr = LogisticRegression()
        self.dt = DecisionTreeClassifier()
    def data_preprocessing(self, data):
        # Clean and transform data
        X = np.array(data)
        X[:, :2] = self.scaler.fit_transform(X[:, :2])
        return X
    def supervised_learning(self, X, y):
        # Train supervised learning model
        self.lr.fit(X, y)
    def unsupervised_learning(self, X):
        # Train unsupervised learning model
        self.kmeans.fit(X)
    def adaptive_learning(self, X, y):
        # Train adaptive learning model
        if self.dt is None:
            self.dt = DecisionTreeClassifier(random_state=0)
        if self.scaler is None:
            self.scaler = StandardScaler()
        X_scaled = self.scaler.fit_transform(X)
        y = np.array(y).ravel().reshape(1,-1)
        X_reshaped = X_scaled.reshape(1, -1)
        self.dt.fit(X_reshaped, y)
    def integration(self, X):
```

```
        # Integrate all models
        preds_supervised = self.lr.predict(X)
        preds_unsupervised = self.kmeans.predict(X)
        preds_adaptive = self.dt.predict(X)
        # Combine predictions
        preds = np.zeros(len(X))
        for i in range(len(X)):
           preds[i] = 1 if (preds_supervised[i] == 1 and
           preds_unsupervised[i] == self.n_clusters-1 and
           preds_adaptive[i] == 1).any() else 0
        return preds
    def evaluate_accuracy(self, X, y):
        # Evaluate accuracy of supervised learning model
        preds = self.integration(X)
        return accuracy_score(y, preds)
    def generate_data(self, n):
        # Generate n random call data entries
        data = []
        for i in range(n):
            a = round(random.uniform(0, 1), 2)
            b = round(random.uniform(0, 1), 2)
            c = round(random.uniform(0, 1), 2)
            d = round(random.uniform(0, 1), 2)
            e = round(random.uniform(0, 1), 2)
            data.append([a, b, c, d, e])
        return data
```

Modified main.py In Python3 - for my custom system

```
import numpy as np
from spam_custom import SpamCallDetector
import matplotlib.pyplot as plt
```

```
import gym

from gym import spaces

from spam import SpamFilter

## custom system

detector = SpamCallDetector(n_clusters=3)

data = detector.generate_data(1000)

X = detector.data_preprocessing(data)

y = np.random.randint(2, size=len(X))

detector.supervised_learning(X, y)

detector.unsupervised_learning(X)

detector.adaptive_learning(X, y)

preds = detector.integration(X)

accuracy = detector.evaluate_accuracy(X, y)

print("My System Accuracy: ", accuracy)

## individual systems

# Define action space

action_space = spaces.Discrete(2) # 2 possible actions (0 or 1)

# Generate some random state spaces and labels (for demonstration
purposes)

num_calls = 1000

state_spaces = []

labels = []

for i in range(num_calls):

    state_space = [np.random.normal(0, 1) for _ in range(5)]  #
Example features: 5 continuous features

    label = np.random.randint(2)  # Example label: binary (0 or 1)

    state_spaces.append(state_space)

    labels.append(label)

# Initialize the spam filter

filter = SpamFilter(state_spaces, action_space)
```

```
# Train the filter using supervised learning

filter.train_supervised(state_spaces, labels)

# Evaluate the accuracy of the supervised learning filter

accuracy_supervised = filter.evaluate_supervised(state_spaces, labels)

print("Supervised Learning Accuracy:", accuracy_supervised)

# Train the filter using unsupervised learning

filter.train_unsupervised(state_spaces)

# Evaluate the accuracy of the unsupervised learning filter

accuracy_unsupervised = filter.evaluate_unsupervised(state_spaces,
labels)

print("Unsupervised Learning Accuracy:", accuracy_unsupervised)

# Train the filter using the adaptive method

transitions = filter.generate_transitions(state_spaces, labels)

filter.train_adaptive(transitions)

# Evaluate the accuracy of the adaptive AI filter

accuracy_adaptive = filter.evaluate_adaptive(state_spaces)

print("Adaptive AI Accuracy:", accuracy_adaptive)

# Create a line graph of the accuracy over time

accuracy_data = [accuracy_supervised, accuracy_unsupervised,
accuracy_adaptive]

# flatten accuracy_data

accuracy_data = np.array(accuracy_data)

print(accuracy_data)

## Graph accuracy

# Create the figure and axes

fig, ax = plt.subplots()

# Plot the lines

# Add horizontal lines for constant values

ax.plot(accuracy_adaptive, label='adaptive')

ax.hlines(accuracy_supervised, 0, len(accuracy_adaptive)-1,
linestyles='dashed', colors='r', label='supervised')
```

```
ax.hlines(accuracy_unsupervised, 0, len(accuracy_adaptive)-1,
linestyles='dashed', colors='g', label='unsupervised')

ax.hlines(accuracy, 0, len(accuracy_adaptive)-1, colors='k',
label='custom')

ax.set_xticks(range(0, len(accuracy_adaptive), 100))

plt.title('Accuracy of Spam Call Filtering Model')

plt.xlabel('Time (ms)')

plt.ylabel('Accuracy')

# Add a legend

ax.legend()

plt.savefig('Figure_2.png')
```

Final code that adapts main_custom.py into a real-time system:

```
import numpy as np

from spam_custom import SpamCallDetector

import matplotlib.pyplot as plt

import gym

from gym import spaces

from spam import SpamFilter

## individual systems

# Define action space

action_space = spaces.Discrete(2) # 2 possible actions (0 or 1)

# Generate some random state spaces and labels (for demonstration
purposes)

num_calls = 1000

state_spaces = []

labels = []

for i in range(num_calls):

    state_space = [np.random.normal(0, 1) for _ in range(5)]  #
Example features: 5 continuous features

    label = np.random.randint(2)  # Example label: binary (0 or 1)

    state_spaces.append(state_space)
```

```
    labels.append(label)

# Initialize the spam filter

filter = SpamFilter(state_spaces, action_space)

# Train the filter using supervised learning

filter.train_supervised(state_spaces, labels)

# Evaluate the accuracy of the supervised learning filter

accuracy_supervised = filter.evaluate_supervised(state_spaces, labels)

print("Supervised Learning Accuracy:", accuracy_supervised)

# Train the filter using unsupervised learning

filter.train_unsupervised(state_spaces)

# Evaluate the accuracy of the unsupervised learning filter

accuracy_unsupervised = filter.evaluate_unsupervised(state_spaces,
labels)

print("Unsupervised Learning Accuracy:", accuracy_unsupervised)

# Train the filter using the adaptive method

transitions = filter.generate_transitions(state_spaces, labels)

filter.train_adaptive(transitions)

# Evaluate the accuracy of the adaptive AI filter

accuracy_adaptive = filter.evaluate_adaptive(state_spaces)

print("Adaptive AI Accuracy:", accuracy_adaptive)

## custom system

print("Generating custom system model...")

# Create a line graph of the accuracy over time for the new model

# feature_vector - size 1x5

# feature_matrix - size nx5, n feature_vector elements in
feature_matrix

accuracy_over_time = [] # expecting n data points in time, i.e. array
of size n

detector = SpamCallDetector(n_clusters=1)

data = detector.generate_data(1000)

X = detector.data_preprocessing(data)
```

```
# Define the minimum and maximum values for each feature

a_min, a_max = -1, 0.3

b_min, b_max = -1, 0.5

c_min, c_max = 10.0, 50.0

d_min, d_max = -120.0, -70.0

e_min, e_max = -3.0, 0.3

y = 0

y_tot = np.zeros(len(X)) # Initialize y_tot as an array of zeros with
the same length as X

for i, feature_vector in enumerate(X):

    print("feat: ", feature_vector)

    if (feature_vector[0] >= a_min and feature_vector[0] <= a_max) or
(feature_vector[1] >= b_min and feature_vector[1] <= b_max) or
(feature_vector[2] >= c_min and feature_vector[2] <= c_max) or
(feature_vector[3] >= d_min and feature_vector[3] <= d_max) or
(feature_vector[4] >= e_min and feature_vector[4] <= e_max):

        y_tot[i] = 1  # set the i-th element of y_tot to 1 if the
feature vector meets the threshold conditions

print(y_tot, type(y_tot))

for i, feature_vector in enumerate(X):

    # Reshape feature vector to a 2D array with one row

    feature_vector = feature_vector.reshape(1, -1)

    print(feature_vector, type(feature_vector[0]))

    detector.supervised_learning(X, y_tot)

    detector.unsupervised_learning(feature_vector)

    detector.adaptive_learning(feature_vector, y_tot)

    accuracy = detector.evaluate_accuracy(X, y_tot)

    accuracy_over_time.append(accuracy)

print("My System Accuracy: ", accuracy_over_time)

## Graph accuracy

# Create the figure and axes

fig, ax = plt.subplots()
```

```
# Plot the lines

# Add horizontal lines for constant values

ax.plot(accuracy_adaptive, label='adaptive')

ax.hlines(accuracy_supervised, 0, len(accuracy_adaptive)-1,
linestyles='dashed', colors='r', label='supervised')

ax.hlines(accuracy_unsupervised, 0, len(accuracy_adaptive)-1,
linestyles='dashed', colors='g', label='unsupervised')

ax.plot(accuracy_over_time, label='custom')

ax.set_xticks(range(0, len(accuracy_adaptive), 100))

plt.title('Accuracy of Spam Call Filtering Model')

plt.xlabel('Time (ms)')

plt.ylabel('Accuracy')

# Add a legend

ax.legend()

plt.savefig('Figure_2.png')

# Create the plot on a new graph alone

plt.clf()

plt.plot(accuracy_over_time)

plt.title('Accuracy of Spam Call Filtering Model - Custom System')

plt.xlabel('Time (ms)')

plt.ylabel('Accuracy')

plt.savefig('Figure_3.png')
```