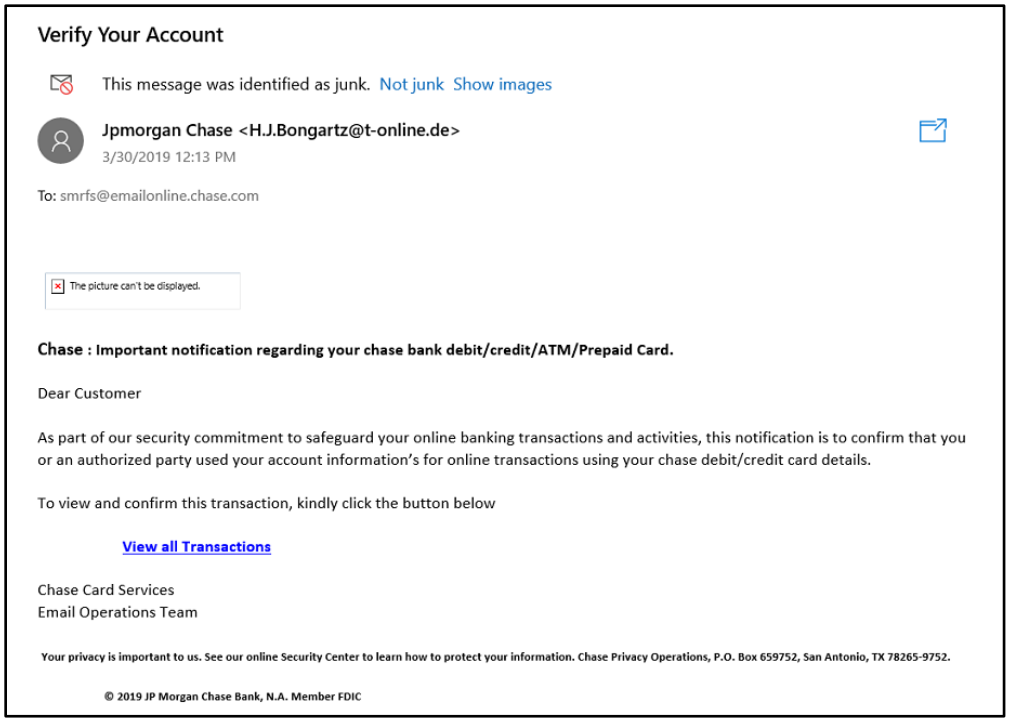
Predicting Spam Emails using Tree based Model

1. Business Understanding

Introduction: Spam emails are unsolicited junk emails which are sent in bulk through the email system. These emails are sent to a group of recipients who did not grant permission to receive such emails. Most of the spammers use spambots to crawl the email address from internet. These spam emails account for billions of emails sent in a day and most of them are commercial in nature. They contain links that look genuine and convincingly familiar. For example, Fig. 1 shows a spam email that looks strikingly familiar to what one might expect to receive from a bank. However, the links lead to phishing websites that hosts malware or ransomware. Furthermore, affected malware computers could become hosts for infecting other computers connected to the same network costing individuals and businesses billions of dollars every year. Though the concept of spam emails has been there for a while, they have obtained more notoriety in the last decade because of the ever evolving and personalized techniques used by spammers such as spear phishing.


Verify Your Account

This message was identified as junk. Not junk Show images

Jpmorgan Chase <H.J.Bongartz@t-online.de>
3/30/2019 12:13 PM

To: smrfs@emailonline.chase.com

The picture can't be displayed.

Chase : Important notification regarding your chase bank debit/credit/ATM/Prepaid Card.

Dear Customer

As part of our security commitment to safeguard your online banking transactions and activities, this notification is to confirm that you or an authorized party used your account information's for online transactions using your chase debit/credit card details.

To view and confirm this transaction, kindly click the button below

View all Transactions

Chase Card Services
Email Operations Team

Your privacy is important to us. See our online Security Center to learn how to protect your information. Chase Privacy Operations, P.O. Box 659752, San Antonio, TX 78265-9752.

© 2019 JP Morgan Chase Bank, N.A. Member FDIC

Fig. 1. Example of a spam email that looks convincingly familiar to what a recipient might expect [1]

Dealing with Spam: One way to arrest spam is by dealing with it at source by blocking its originating IP address. Email service providers (like Google, Yahoo) can block the list of IP addresses that are already reported as trojans/bots that spread spam messages. But this is not an effective way to prevent spam since new computers are affected by malware on a regular basis. The infected IP address list maintained by service providers cannot be updated frequently and at any given time will only contain a fraction of the infected computers.

The above-mentioned problems on the loss incurred due to spam emails and practical issues with blocking potential spam email senders shows the importance to develop an effective email identification system that can filter spam from non-spam messages. One such approach is to use spam filters that detect unsolicited, unwanted, and virus-infested emails and stop them from getting into the target's inbox. Email service providers use spam filters to make sure they aren't distributing spam. They use "heuristics" methods, which means that each email message is subjected to thousands of predefined rules. In our case study, we explored and evaluated one such spam filter based on a "tree-based" heuristic method that can effectively identify spam from a group of emails based on various attributes.

Assumption: Since the spam filter system is based on a set of rules for identifying spam, we assume that the spammers will not learn these rules and change their approach when sending spam in the future. This is however not a fair assumption, because spammers are constantly evolving and learning new techniques to trick the recipients and the spam filters. Hence, we also propose workarounds for overcoming evolving spam characteristics towards the end of this study.

2. Data Evaluation / Engineering

2.1 Data Dictionary

Table 1: Data Dictionary [2]

Variable	Type	Definition
isRe	*logical*	TRUE if Re: appears at the start of the subject.
numLines	*integer*	Number of lines in the body of the message.
bodyCharCt	*integer*	Number of characters in the body of the message.
underscore	*logical*	TRUE if email address in the From field of the header contains an underscore.
subExcCt	*integer*	Number of exclamation marks in the subject.
subQuesCt	*integer*	Number of question marks in the subject.
numAtt	*integer*	Number of attachments in the message.
priority	*logical*	TRUE if a Priority key is present in the header.
numRec	*numeric*	Number of recipients of the message, including CCs.
perCaps	*numeric*	Percentage of capitals among all letters in the message body, excluding attachments.
isInReplyTo	*logical*	TRUE if the In-Reply-To key is present in the header.
sortedRec	*logical*	TRUE if the recipients' email addresses are sorted.
subPunc	*logical*	TRUE if words in the subject have punctuation or numbers embedded in them, e.g., w!se.
hour	*numeric*	Hour of the day in the Date field.
multipartText	*logical*	TRUE if the MIME type is multipart/text.
hasImages	*logical*	TRUE if the message contains images.
isPGPsigned	*logical*	TRUE if the message contains a PGP signature.
perHTML	*numeric*	Percentage of characters in HTML tags in the message body in comparison to all characters.
subSpamWords	*logical*	TRUE if the subject contains one of the words in a spam word vector.
subBlanks	*numeric*	Percentage of blanks in the subject.
noHost	*logical*	TRUE if there is no hostname in the Message-Id key in the header.
numEnd	*logical*	TRUE if the email sender's address (before the @) ends in a number.
isYelling	*logical*	TRUE if the subject is all capital letters.
forwards	*numeric*	Number of forward symbols in a line of the body, e.g., >>> xxx contains 3 forwards.
isOrigMsg	*logical*	TRUE if the message body contains the phrase original message.
isDear	*logical*	TRUE if the message body contains the word dear.
isWrote	*logical*	TRUE if the message contains the phrase wrote:.
avgWordLen	*numeric*	The average length of the words in a message.
numDlr	*numeric*	Number of dollar signs in the message body.

For this study, we used a dataset that had data from 9348 emails and 29 features/attributes as indicated in Table 1. The target variable "`isSpam`" takes Boolean (True/False) values and shows whether an email is spam or not. The data contains several features that are important for detecting spam from non-spam emails. For example, Fig. 2 shows some of the trigger words that are used often by spammers. The data contains a feature called `subSpamWords` which is *True* if the subject contains words from one such trigger word list. Similarly, as indicated in Fig. 3, other features that are indicative of spam emails are the use of all capital words, use of exclamations, attachments, etc. These are also captured in the data using features such as `perCaps`, `isYelling`, `subExcCt`, `numAtt`, etc.

SPAM Trigger words				
% off	Double your income	Great offer	Never	Remove
$$$	Earn $	Guarantee	No gimmiks	Reverses
100% free	Earn extra cash	Help	No hidden costs	Sample
100% satisfied	Extra income	Hidden	No investment	Satisfaction
Acceptance	F r e e	Hidden assets	Now only	Satisfaction guaranteed
Accordingly	Fast cash	Increase sales	Obligation	Save $
Act now	Free	Increase traffic	One hundred percent free	Search engine listings
Affordable	Free gift	Increase your sales	One time	Serious cash
Apply now	Free info	Incredible deal	Opportunity	Solution
Avoid	Free installation	Info you requested	Order now	Special promotion
Billion	Free investment	Lifetime	Order today	Stop
Cash bonus	Free leads	Limited time offer	Please read	Success
Chance	Free membership	Lose	Prices	Test
Cheap	Free offer	Maintained	Problem	Thousands
Click here	Free preview	Make $	Promise you	Urgent
Compare rates	Free trial	Medium	Refinance	Visit our website
Credit	Get started now	Miracle	Reminder	Web traffic

Fig. 2. List of some spam trigger words [3]

Fig. 3. Important characteristics of a spam email [3]

2.2 Missing Data Analysis

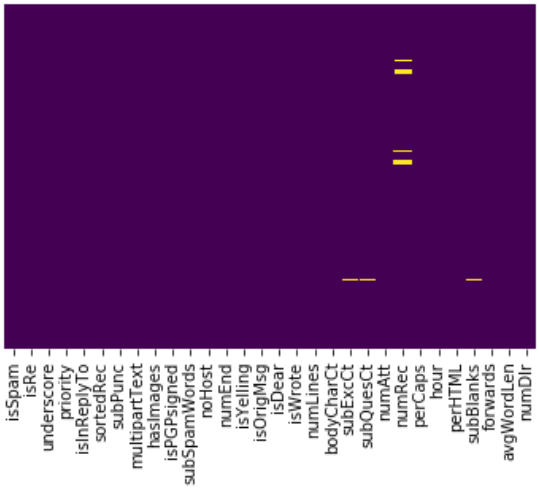


Fig. 4. Missing data heatmap (Yellow color indicates missing values)

This heatmap in Fig. 4 shows that there are missing values (indicated in yellow) across four columns: `subExcCt`, `subQuesCt`, `numRec`, `subBlanks`. Out of these four columns, `numRec` had the greatest number of missing values (282 missing values) across 9348 rows. One option to deal with the missing values is to impute them based on mean/median/mode value of the features. However, this can introduce bias in the data. Since we did not have the right context behind why these values were missing and the missing observations constituted only 3% of the entire dataset, we decided to delete them from the data before developing the model. While this process is also not ideal, we felt that it was more appropriate given the circumstances. An alternate approach could be to impute using the closest known non null observation. This process is called KNN imputation [4]. This gives a more context specific imputation, but we will leave this as a post initial evaluation exercise.

2.3 Exploratory Analysis

In this section, we will look at the plots that show the summary of few interesting attributes in the dataset for both spam and non-spam category.

`numLines`: Fig. 5 shows the boxplot for spam detection using the number of lines as the sole attribute. The box plot on the left shows that the `numLines` attribute is right skewed and no clear inference can be made from it. Hence, we used log transformation and the box plot on the right shows the distribution of log(`numLines`) for both spam and non-spam category. The spam emails seem to have a greater number of lines in them in general compared to non-spam emails. This makes sense since spammers will try to pepper the email with links and repeated claims (such as winning the lottery, etc.) in many places in hopes that the recipients will click on any one of them. Non-spam emails tend to be more direct and shorter.

Although not completely separated, there is still some level of difference between the distribution in spam and non-spam emails. Hence this feature may be useful in differentiating between spam and non-spam, especially when used in conjunction with other important features.

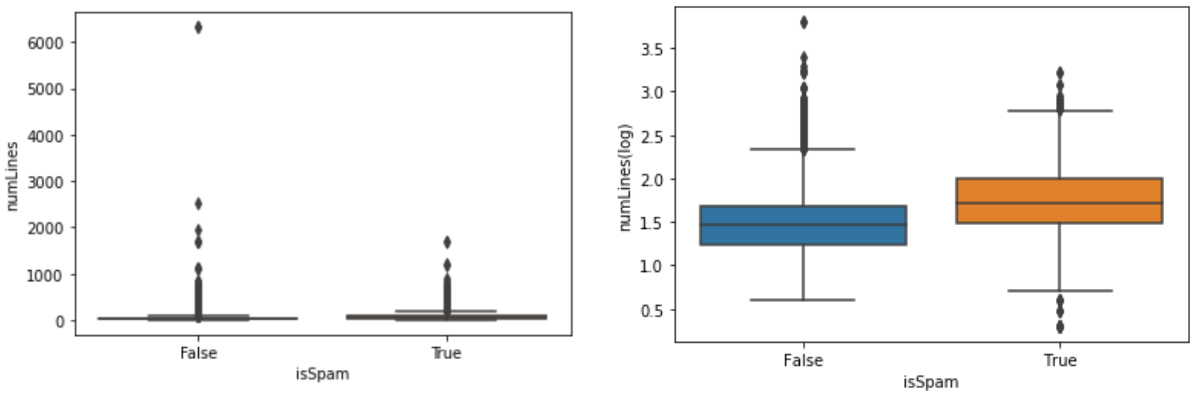


Fig. 5. Distribution of number of lines in the body of spam and non-spam emails (left: Original scale, right: log Y axis)

`perHTML`: Fig. 6 shows that spam emails generally have a very high percentage of HTML links in them compared to non-spam emails. These HTML links may be taking the recipients to phishing websites or pointing them to malware. Hence this attribute may be important in identifying spam from non-spam emails.

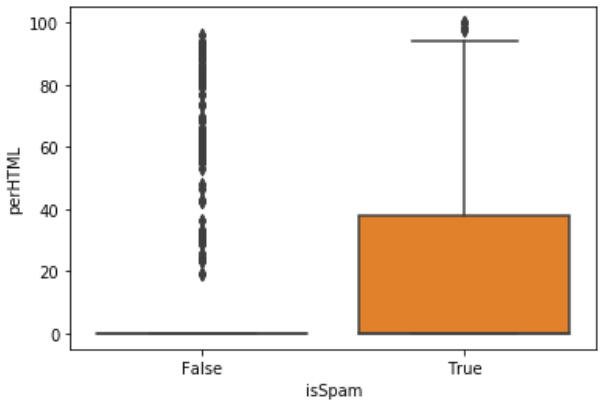


Fig. 6. Distribution of percentage of HTML in spam and non-spam emails

`bodyCharCt`: Like `numLines`, `bodyCharCt` shows some separation between spam and non-spam emails. Again, although not exactly separable, this may be an important feature when used in conjunction with other stronger features in detecting spam.

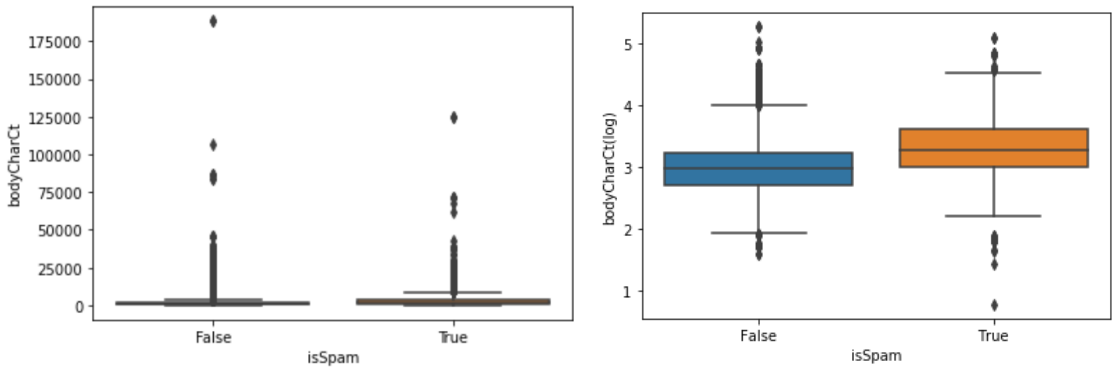


Fig. 7. Distribution of number of characters in the body of spam and non-spam emails (left: Original scale, right: log Y axis)

perCaps: Fig. 8 shows that the percentage of words using capital letters is much higher in spam emails vs. that in non-spam emails. Spam email generally contain a lot of words in capital letters to grab the recipient's attention. Hence this might be an important feature in segregating spam from non-spam.

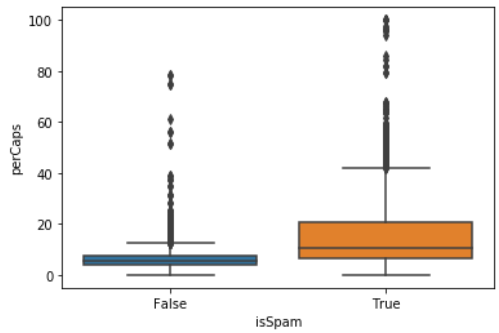


Fig. 8. Distribution of percentage of capital letters in the body of spam and non-spam emails

forwards: Fig. 9 shows that when emails are forwarded, they are most likely to be a non-spam email. This makes sense since most recipients would not forward spam emails knowingly. Hence this could be an important attribute in deciding whether an email is spam or not.

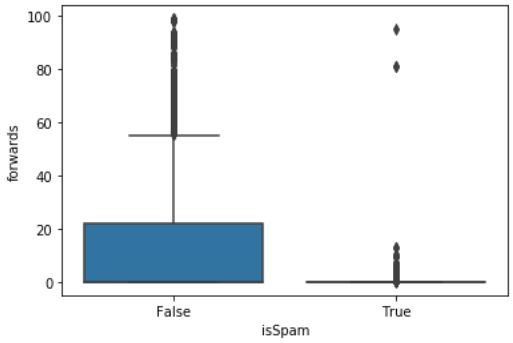


Fig. 9. Distribution of forwards between spam and non-spam emails

underscore: Fig. 10 shows that only a minute percentage of spam emails contain underscore in the "from" email ID whereas there are no underscores in non-spam email IDs. This shows that the presence of underscore can only classify a small percentage of spam emails and may not be an important feature when building the model.

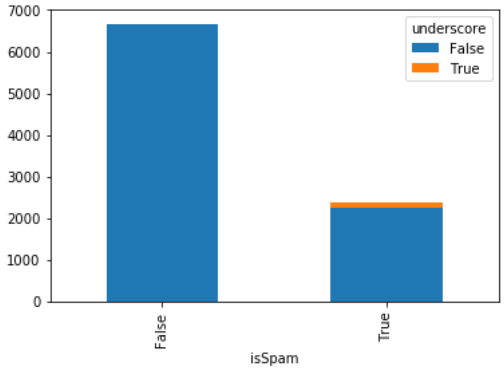


Fig. 10. Count of "from" email address containing an underscore for spam and non-spam emails

`isPGPSigned`: Fig. 11 shows that only some of the non-spam emails are PGP signed whereas none of the spam emails are PGP signed. This attribute alone cannot be used to determine if an email is spam or not spam because if an email is not PGP signed it could either be a spam or not spam.

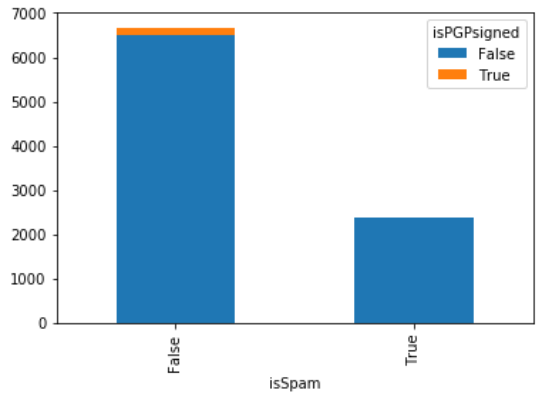


Fig. 11. Count of PGP signatures contained in spam and non-spam emails

`hasImages`: Fig. 12 shows the count of emails containing images. This attribute does not have much importance in determining spam because only a negligible amount of spam emails have images.

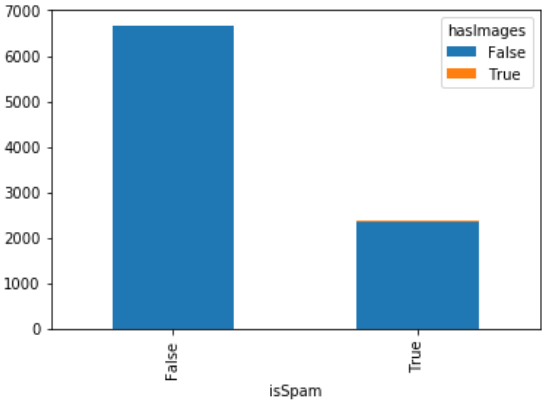


Fig. 12. Count of spam and non-spam emails that contain images

3. Modeling Preparations

3.1 Model Selection and Tuning Overview

One of the objectives of this case is to examine and understand the choices the model is making to identify the spam emails. Since we need the model to be explainable, we cannot use a black-box model, rather we will use a model which can clearly explain why a decision was made (a white-box model). A white-box models allows the user to clearly understand how the model behaves, which variables are used, and how (and when) they are used. We choose to use the *"Decision Tree"* model in the Scikit-learn library available in the Python programming language. This *"Decision Tree"* model can perform regression in addition to binary and multi-class classifications. This case study

requires the use of a binary classification model only as the target variable can only have two values: *"Spam"* (True) or *"Not Spam"* (False).

A *"Decision Tree Classifier"* works by selecting a feature from the set of predictors and then making a comparison to a threshold (if the feature is numeric) or to a fixed value (if the feature is categorical or Boolean). The decision of what feature and threshold to select is based on how pure the resultant splits will be (based on the training data). A pure split is a split that contains observations from only one of the class levels (either "Spam" or "Not Spam"). Of course, it may not be possible to get a pure split in all comparisons. Hence a measure is used to define this "purity" (the most popular being the "Gini" index and "Entropy"). A value of this purity index equal to 0 represents a pure split and a value equal to 1 represents a split which has equal number of observations from each class level (random guess). The feature and threshold selected is one that results in the maximum decrease in the purity index from the previous split. Based on the outcome of the comparison, the decision tree makes more hierarchical comparison using the same or other features until it reaches a terminal condition (leaf node) based on certain hyperparameters. At this point, it makes a prediction based on the majority class present in the leaf node.

The main advantage of the "*Decision Tree Classifier"* is the simplicity of its interpretation and visualization of the results. The comparison of features to fixed thresholds or values is easy to understand and interpret. Moreover, these methods do not require transformation of the predictors or the need to scale them (as required by other algorithms). Nonetheless, there are some limitations that need to be considered while building these models:

1. These models in the Scikit framework cannot support missing predictor values (as can be done in some other languages such as R). As indicated in Section 2.2, we decided to remove records with missing values since they represented a very small fraction of the entire dataset and we did not have the right context to impute them.

2. These models tend to overfit i.e. the model will produce good metrics when training on known observations but will not be able to generalize when predicting unseen observations i.e. it tends to memorize the training observations. To reduce the risk of overfitting we will use cross-validation technique during training as described in 3.3. The final model is refit on the entire training data using the best obtained hyperparameters and then used to make predictions on a completely unseen hold-out (test) dataset.

3. The model contains several hyperparameters that can be used to train the model and make bias (underfitting) vs variance (overfitting) tradeoffs. Since hand tuning multiple hyperparameters is very costly, we prefer to use a grid search approach. However, even grid search can be costly since we can be evaluating the hyperparameters in a suboptimal search region [5]. Hence, in order to make the hyperparameter search more efficient, we will use Bayesian Optimization. This method allows for making good tradeoffs between exploring unknown regions of the search space and exploiting known good regions when deciding what set of hyperparameters to try next in the optimization process. It has been shown to be about 10x more efficient than regular grid search approaches.

3.2 Custom Evaluation Metric

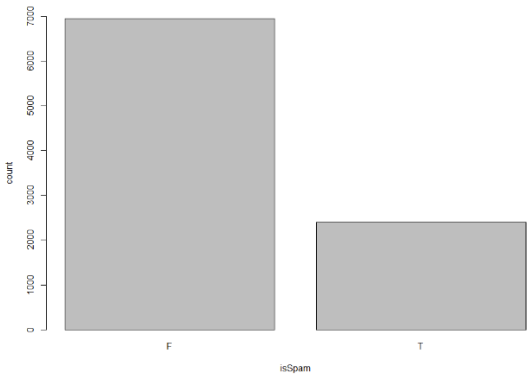


Fig. 13. Number of Spam and Non-Spam emails.

Fig. 13 shows that the dataset used for this study is imbalanced. The number of rows for non-spam email are higher (6951 entries) when compared to number of rows for spam email (2397 entries). Hence the metric used to evaluate our model should be able to handle this imbalance in spam vs non-spam categories. When it comes to model evaluation metrics, Precision, Recall, and F1 score are important metrics. These are described below.

Precision: This metric evaluates how precise the model is. It calculates how many observations are true positives out of the total predicted positives. Total predicted positive is the sum of True positives and False positives. In this case, True Positive means a spam identified as spam and False Positive means a non-spam identified as spam. Precision is a good measure to determine when the cost of False Positive (non-spam detected as spam) is high. In this case, even though our primary objective is to classify spam, we want to make sure that we do not classify non-spam emails as spam as it may lead to important emails going to spam. Hence our final metric will be skewed a little towards the Precision of the spam class level.

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} = \frac{True\ Positive}{Total\ Predicted\ Positive} \qquad \text{Equation 1}$$

Recall: Recall calculates how many actual positives the model can capture by classifying it as True Positive. This measure is negatively affected by False Negatives. In this case, False Negative means a spam identified as non-spam. Recall is a good model metric to use when there is a high cost associated with False Negative. Since our data is skewed towards the non-spam class, our model might easily learn to just predict everything as non-spam in which case the Recall will be low. Hence it is important to consider Recall in our final metric as well.

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} = \frac{True\ Positive}{Total\ Actual\ Positive} \qquad \text{Equation 2}$$

F1: F1 score is a measure that balances both precision and recall score of the model. If any one of these metrics is low, the resultant F1 score will be low as well. In this case, since we need to take both Precision and Recall into consideration, the F1 score is an ideal scoring metric to compare various models and pick the best one.

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \qquad \text{Equation 3}$$

Final Custom Metric: As described above, one of the objectives in this case study was to try not to classify non-spam emails as spam. This implies that the precision score needs to be weighted high as the cost of false positive (non-spam classified as spam) is considered high. We also need to include the F1 score to take Recall into account due to the skewed dataset. The scikit-learn framework is helpful in addressing the need for a custom metric like this [6]. Using this we implemented a customized metric skewed towards the Precision of the "spam" class level while at the same time taking a balanced approach so as not to skew the results too much.

$$Custom\ score = F1_{macro} + Precision_{positive\_class} \qquad \text{Equation 4}$$

3.3 Sampling Technique

Fig. 14 depicts the sampling technique used in this study. To best evaluate the model performance, we used 80% of the records to train the model and 20% was held out for testing the final model. Even while training the model, we needed to make sure that we have an unbiased estimate of the model's performance. One approach to achieve this is to use k-fold cross validation. In this approach, the training data is divided into k different segments. In each iteration, only "k-1" folds are used for training and the "out of fold" (OOF) sample is used as a "validation" set to compute the metric. This process is repeated "k" times and in the end, each fold will be treated as the "validation" set once. The final metric can be obtained by averaging the metric across the "k" folds. In this case, we decided to choose k = 3.

As indicated in 3.1, the "*Decision Tree Classifier*" tends to overfit. During the model tuning step, we evaluated various combinations of hyperparameters. The hyperparameters that were tuned were those that were most likely to prevent overfitting and underfitting as indicated in Table 2. The "best" set of hyperparameters were selected using the

average value of the custom metric across the k-folds. Using this "best" set of hyperparameters, the model was refit over the entire train dataset (80%). This final retrained model was then used to make a prediction on the hold-out (test) dataset containing the remaining 20% of the observations to obtain the final performance of the model.

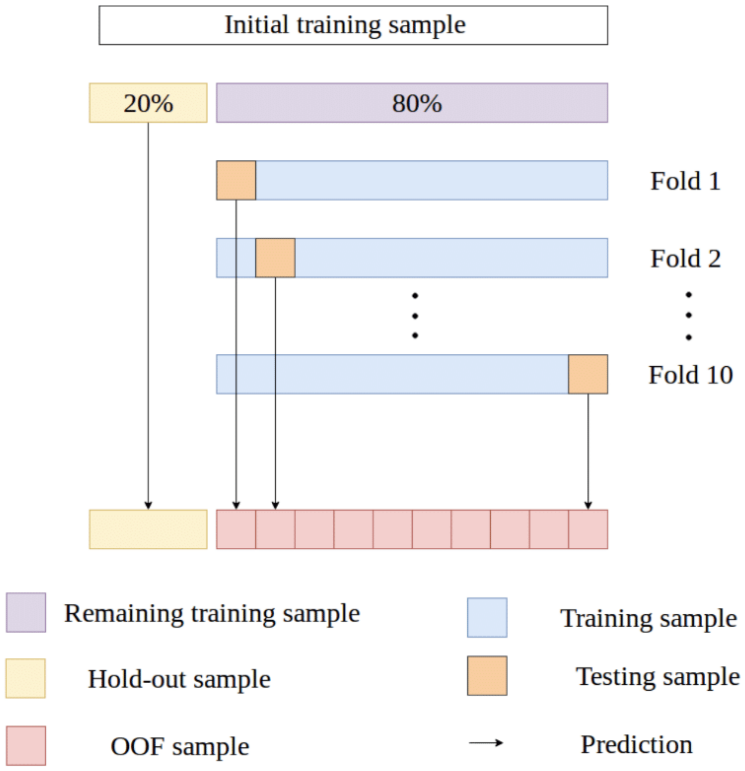


Fig. 14: Sampling technique used for model generation [7]

3.4 Reproducibility

Decision Trees are based on randomization of the feature selection during the splitting of the internal trees. This can create reproducibility issues as different executions of the model may obtain different results. To mitigate this risk, we set a fixed seed value of 42, which will ensure that the results obtained are repeatable.

4. Modeling Building & Evaluation

4.1 Important Hyperparameters

Table 2 shows the list of parameters we choose as the most import for Decision Tree Classifiers, the parameters have been selected based on their impact to the model for this case study. We wanted to pay attention predominantly to the risk of overfitting which is common observed for the decision trees. In addition, we wanted to have options to reduce the complexity of the model to make it more explainable. Unfortunately, there is no unique hyperparameter that can satisfy both these requirements. Hence, we need to find the right balance between all these parameters to maximize our objective and custom evaluation metric.

Table 2: Important Hyperparameters in Decision Tree Classifier [8]

Parameter	Meaning	Impact on Model Performance	Default Value
`max_depth`	The maximum depth of the tree. If *None*, then nodes are expanded until all leaves are pure or until all leaves contain less than *min_samples_split* samples.	Increase this parameter to reduce bias (underfitting). Increasing it too much may lead to high variance (overfitting)	None
`min_samples_split`	The minimum number of samples required to split an internal node.	Increase this parameter to reduce variance (overfitting). Increasing it too much may lead to high bias (underfitting).	2
`min_samples_leaf`	The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least *min_samples_leaf* training samples in each of the left and right branches.	Increase this parameter to reduce variance (overfitting). Increasing it too much may lead to high bias (underfitting).	1
`max_features`	The number of features to consider when looking for the best split. If *None*, then *max_features* = n_features.	Increase this parameter to reduce bias (underfitting). Increasing it too much (including too many redundant features) could lead to increased variance (overfitting).	None
`ccp_alpha`	Complexity parameter used for Minimal Cost-Complexity Pruning. By default (0), no pruning is performed. It is expressed as a decimal number ranging from 0 to 1	Increase this parameter to reduce variance (overfitting). Increasing it too much may lead to increased bias (underfitting).	0
`criterion`	The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.	Difficult to quantify. Gini tends to be faster. One or both could be evaluated and best one selected in the end.	gini

4.2 The baseline model

To evaluate the performance of the final optimized model, we first developed a simple baseline model that can be used as a reference point. The baseline model was created using the Decision Tree Classifier with default setting as shown in Table 2. The resulting confusion matrix for the test dataset is displayed in Table 3, while Fig. 15 shows the key metrics. The model obtained an average (macro) F1-score of 0.96 and a precision for the positive class of 0.95 resulting in a custom score of $0.96 + 0.95 = 1.91$. This is a very good score considering that the maximum achievable score is 2. We can see that even when we didn't specify any optimized parameters, the model has been able to find good separation across the spam and non-spam emails using the available predictors. Given such a large score for the baseline model, we are not expecting any large improvement with the optimized decision tree model.

Table 3 Confusion Matrix for the Baseline Model

	Predicted Non-Spam	Predicted Spam
Actual Non-Spam	1,322	25
Actual Spam	30	433

	precision	recall	f1-score	support
False	0.98	0.98	0.98	1347
True	0.95	0.94	0.94	463
accuracy			0.97	1810
macro avg	0.96	0.96	0.96	1810
weighted avg	0.97	0.97	0.97	1810

Fig. 15 The classification report for the baseline models

4.3 Tuning the model

Tuning the model is an important step to find the best parameters that can lead to a high performance and a generalizable model. The dataset has many predictors (29) and more than nine thousand records which may lead to a complex model that can easily overfit. As indicated in 4.1, we need to find the right set of hyperparameters to prevent overfitting and balance underfitting. The search space used for the hyperparameters optimization is indicated in Table 4. A normal grid search algorithm must parse more than a million combinations to satisfy all the selected parameters (considering all discrete values for hyperparameters such as `max_depth`, `min_samples_split`, `min_samples_leaf`, and `max_features`). However, with parallelized Bayesian optimization, we can find the ideal set of hyperparameters within a very short amount of time (a few minutes in this case).

Table 4: Hyperparameter Search Space for Bayesian Optimization

Parameter	Min Value	Max Value
max_depth	1	100
min_samples_split	2	100
min_samples_leaf	1	100
max_features	1	Maximum number of features available (29)
ccp_alpha	0.00	1.00

Table 5: Optimal Hyperparameter values obtained from the Bayesian Optimization process

Parameter	Optimal Value
max_depth	23
min_samples_split	2
min_samples_leaf	1
max_features	17
ccp_alpha	0.00

After running the optimization process, we found the ideal model has the hypermeters listed in Table 5. We can quickly notice that these parameters are not very different from the baseline model, hence we would not expect a large difference in the results. Fig. 16 shows the key metric for the optimized model. The overall custom score for the optimized model is $F1_{macro} + precision_{positive} = 0.96 + 0.95 = 1.91$ which is the same value obtained with the baseline model.

A more detailed view from of the confusion matrix depicted in Table 6 shows small changes compared to the baseline model. While the baseline model misclassified 25 non-spam emails as spam (Table 3), the optimized model misclassified only 23 of them. This improved performance is compensated for by a worse classification of actual spam email as spam (426 vs. 433). The optimized model misclassified 37 spam emails as non-spam, while the baseline model misclassified only 30 of them. This is acceptable since our goal was to err on this side of the scale (we were ok with more spam not being predicted as spam if we get less non-spam emails going to spam). Another way to look at this is to dissect the precision metric for the positive class (Fig. 16), where for every 100 email classified as spam by the classifier, 95 would have actually been spam. Hence, 5 out of every 100 spam classified emails would have actually been non-spam in reality. These emails may not be immediately visible to the user. In any case, as predicted earlier, the baseline performance was very close to optimal and further improvements were likely going to be minimal if any.

	precision	recall	f1-score	support
False	0.97	0.98	0.98	1347
True	0.95	0.92	0.93	463
accuracy			0.97	1810
macro avg	0.96	0.95	0.96	1810
weighted avg	0.97	0.97	0.97	1810

Fig. 16. The classification report for the optimized model

Table 6 Confusion Matrix for the Optimized Model

	Predicted Non-Spam	**Predicted Spam**
Actual Non-Spam	1,324	23
Actual Spam	37	426

4.4 Impact of the hyperparameters on the model performance

During the hyperparameter optimization process, since all the hyperparameters are varied simultaneously, it is difficult to determine the impact of a single hyperparameter in isolation. However, using the results of the individual runs of the optimizer, we can construct a surface response of the custom metric as the hyperparameters are varied. In addition, this can also be used to construct partial dependence plots. Partial dependence plots are plots that show how varying a single attribute/parameter (hyperparameter) will impact the outcome (custom metric) when all the other attributes/parameters are kept constant. This result can be observed from Fig. 17. From the plots shown on the diagonals, we can see that the performance remains almost constant when most of the hyperparameters are changed. The hyperparameter that seems to have the most impact on the performance of the model is the `ccp_alpha`. The result seems to indicate that reducing this parameter all the way down to 0 is recommended to maximize the custom metric. This would correspond to an untrimmed tree that can be built as deep as possible. `max_depth` also seems to have a marginal impact on the performance and the plot shows that if `max_depth` was varied in isolation, it should be decreased to some extent. This manifests as a not so deep tree in the final optimized model (`max_depth` = 23) compared to the baseline. Finally, `max_features` and `min_samples_leaf` also seems to also have a marginal impact on the performance. In isolation (all other parameters remaining constant), the optimizer recommends that these parameters be increased to get marginal gains in performance. However, in summary all these marginal gains are likely going to be overshadowed by the decrease in `ccp_alpha`.

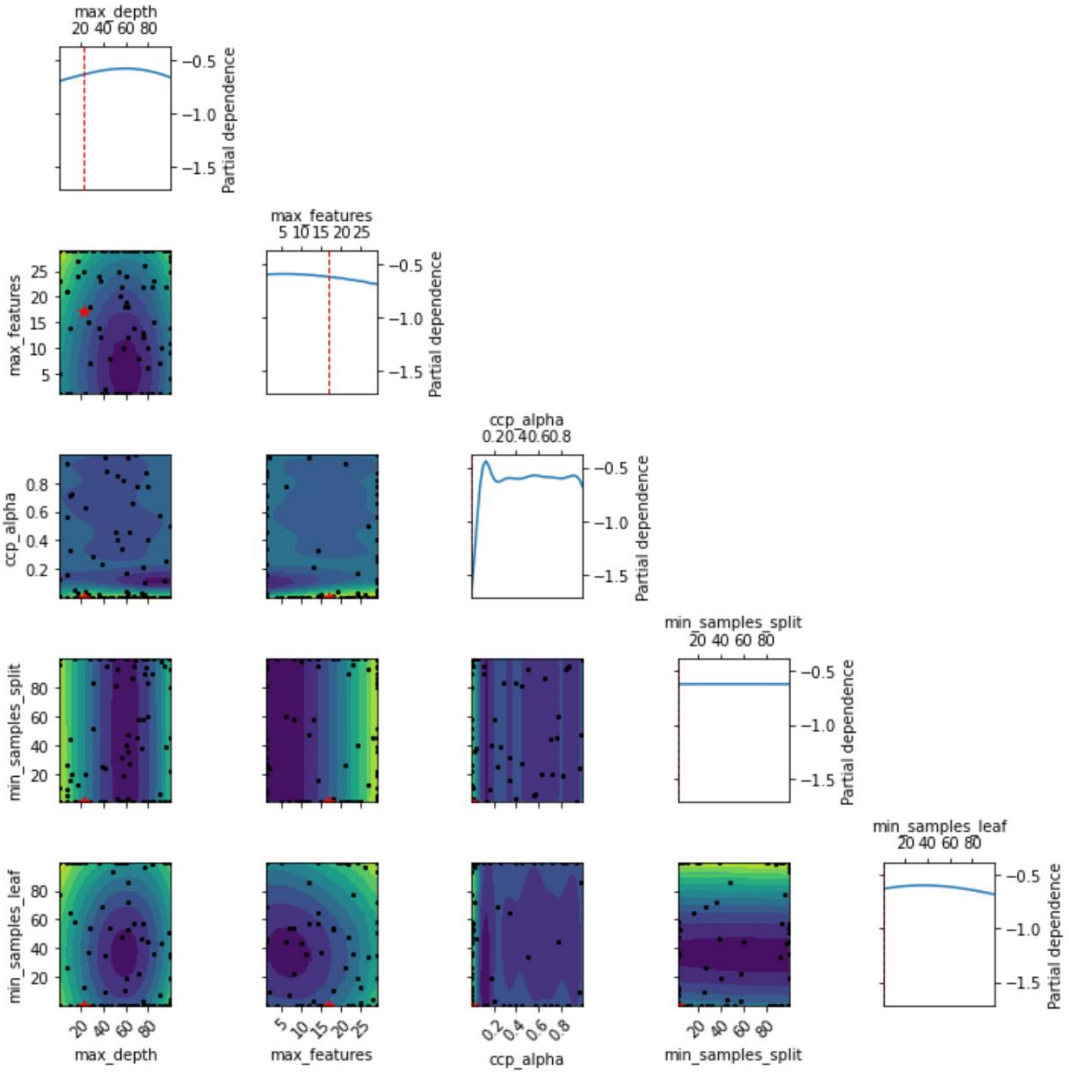


Fig. 17. Partial Dependence Plots for the Hyperparameters used during Bayesian Optimization

5. Model Interpretability & Explainability

5.1 Important Variables

Fig. 18 shows the feature importance obtained from the final model. We use the feature randomizer method to obtain the feature importance. In this method, first a simple model is developed to get a baseline score. Then the model building process is repeated by randomizing one of the features at a time (converting this feature into "white-noise"). If the feature was important in classifying the emails, then randomizing that feature would lead to a reduction in the metric. The more the metric deteriorates, the more important that feature is. The extent of reduction can hence be used a proxy for feature importance and that is shown in Fig. 18.

The results in this figure are mostly in line with our exploratory analysis. The features such as `bodyCharCt`, `perHTML`, `perCaps`, `numLines`, `forwards` were considered somewhat important to decipher between spam and non-spam emails and they show up on the top of the list. Other features such as `isPGPSigned`, `subQuesCt`, `hasImages` were not considered as important features and they show up at the bottom of the list.

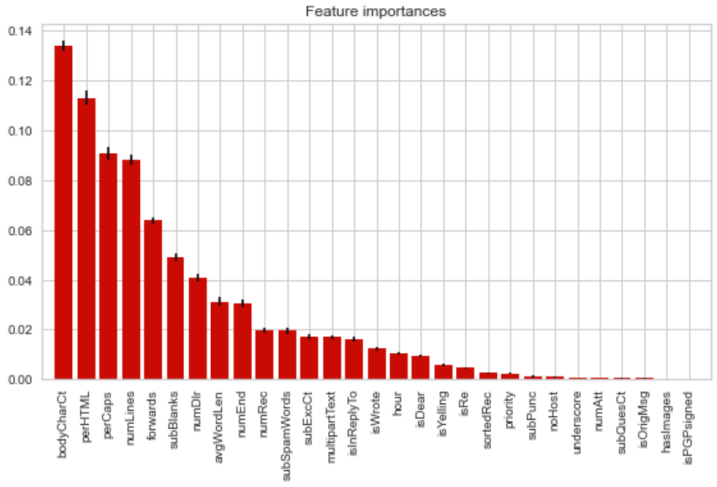


Fig. 18. Feature Importance obtained from the Optimized Model

Another way to visualize the joint importance of features is shown in Fig. 19 which shows how good two features used in conjunction will be at separating the spam and non-spam emails. This visualization creates a hexplot considering some of the top features in pairs and plotting them on the x and y axis. The regions at the intersection of these two features can then be represented by a hexagon. The color of the hexagon is determined by how many spam and non-spam emails are in that feature intersection region. A region containing only spam emails is represented by a pure yellow color and a region containing only non-spam emails is represented by a pure purple color. These regions can be easily separated by a decision tree using threshold comparison-based rules. If a region consists of both spam and non-spam emails, the color of the hexagon will lie in the middle (varying shades of green and blue). These regions are harder to separate for a decision tree and it will not perform well in these regions. Ideally, we would like to see regions with only pure purple and pure yellow hexagons. In such cases, the features can be considered as important in separating the spam from non-spam emails.

Additionally, the background shade shows the decision that a tree-based classifier that is built with just these two features would make. Again, a yellow region represented feature combinations where spam would be detected, and a purple region represented feature combinations where non-spam would be detected. Ideally, we would like to have pure yellow hexagons in a purely yellow shaded region (and same for purple colored regions). If any of the shaded regions has a bluish or greenish colored hexagon, then we know that the model is not doing well in that region.

From Fig. 19, we can see that a feature like `forwards` is an important feature and whenever it is used in conjunction with another top feature, the space consists of predominantly uniformly colored hexagons that are easy to separate for the decision tree. Similarly, `perCaps` and `perHTML` in combination with one other top feature seems to perform reasonably well in separating spam from non-spam (mostly homogeneous colored hexagons). `bodyCharCt` and `numLines` also show a decent amount of separation between the two class levels. This is by no means a perfect way to visualize the effects (since eventually the model will be built with many more features) but can give us a relatively good estimate of which features are important in deciphering between spam and non-spam.

On the contrary, Fig. 20 shows some of the features that show up at the bottom of the feature importance list in Fig. 18. Clearly, `numAtt, subQuesCt` and `hour` are not so good at separating the spam and non-spam emails since their space consists of a lot of bluish and greenish colored hexagons (instead of purple and yellow colored hexagons).

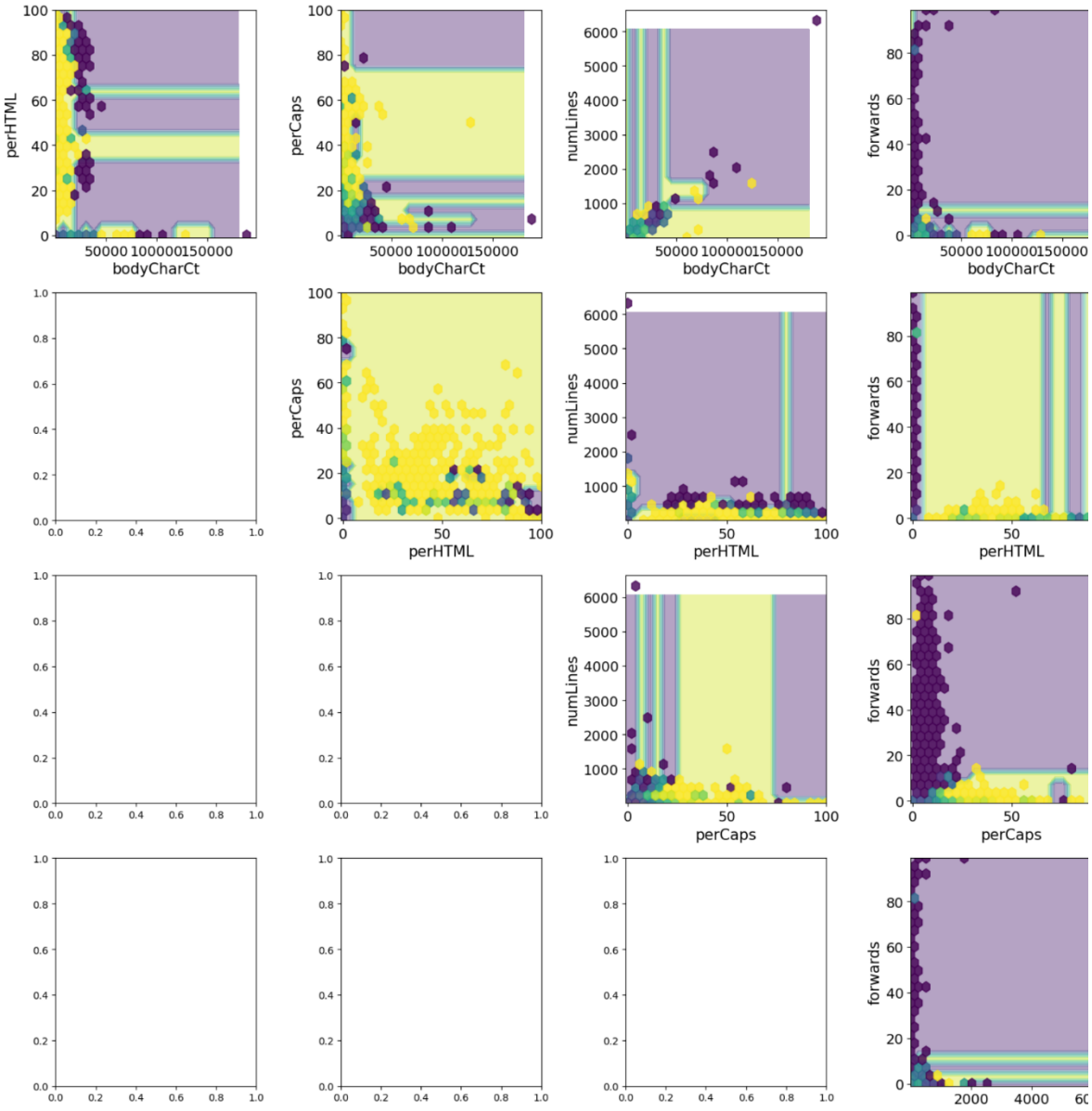


Fig. 19. Visualizing important features in separating the spam from non-spam emails.

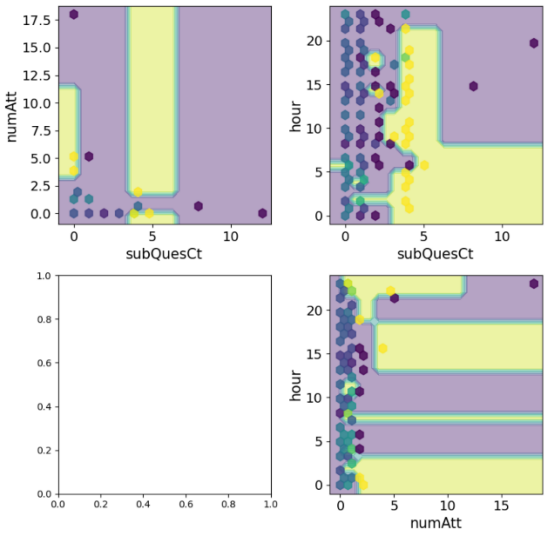


Fig. 20. Visualizing unimportant features in separating the spam from non-spam emails.

5.2 Tree Visualization

The Decision Tree Classifiers provides an easy way to visualize the decision-making process that is used to classify spam and non-spam emails. As the optimized model has 23 decision layers it would be difficult to show it in this document. Hence, we fit a new simpler model with a maximum depth of 3 layers to explain the model's decision-making process. This can then be extended to visualize the decision-making process for the final more complicated model. Fig. 21 shows the decision tree for the new simple model.

The objective of each decision point in the tree is to split the dataset in such way that the Gini index is reduced as much as possible. The first decision point uses the percent of capitalized letters (`perCaps`) to split the dataset in two section. If the percent of capitalized letters in an observation is less than 12.516, then the observation will be split to the left node, otherwise will be split to the right. The left branch has a Gini index of 0.291, while the right branch has a Gini index of 0.364, which are both lower than the starting Gini index of 0.388. The left branch has more non-spam emails compared to spam emails (5,069 vs 1,088) while the right branch has more spam emails than non-spam emails (821 vs. 258). This makes sense since we observed that the non-spam emails have a much lower percentage of words that use capital letters compared to non-spam emails. Using this information, we see that the model has already been able to separate the data in two groups are purer than the original data.

Next decision point on the left branch uses the percent of HTML tags (`perHTML`) in the email to split the records in that node. If this value is less than 3.926 the flows goes to the left, otherwise goes to the right. The left branch still has majority of non-spam emails, but with a lower Gini index of 0.233. The right branch has more spam than non-spam emails. Again, this makes intuitive sense since spam emails had more HTML tags as observed in the exploratory analysis.

Next, we encounter the last decision point on the left branch which is based on number of `forwards`. If there are less than 1.592 forwards, we will end with a non-spam decision. The same is also observed if the number of forwards is more than 1.592. However, we see that the purity of the split with greater than 1.592 forwards is much better (Gini index = 0.014) than the split with less than 1.592 forwards (Gini index = 0.361). Again, this makes intuitive sense from the exploratory analysis where we noticed that the non-spam emails had in general many more forwards compared to non-spam emails. We also note that the split for less than 1.592 forwards risks classifying emails as non-spam when a large portion (745 out of 3146) of them are spam. This indicates the need of more layers in the tree to further improve the performance of the model.

The same process can be repeated to visualize any other branch of this tree. From this analysis, we note another interesting observation that the decision tree has very low Gini index for five of the eight end points (leaf nodes) although these nodes tend to have very few observations. This indicates the decision tree can correctly classify some of the emails with high confidence and that it needs to improve only three of the leaf nodes further.

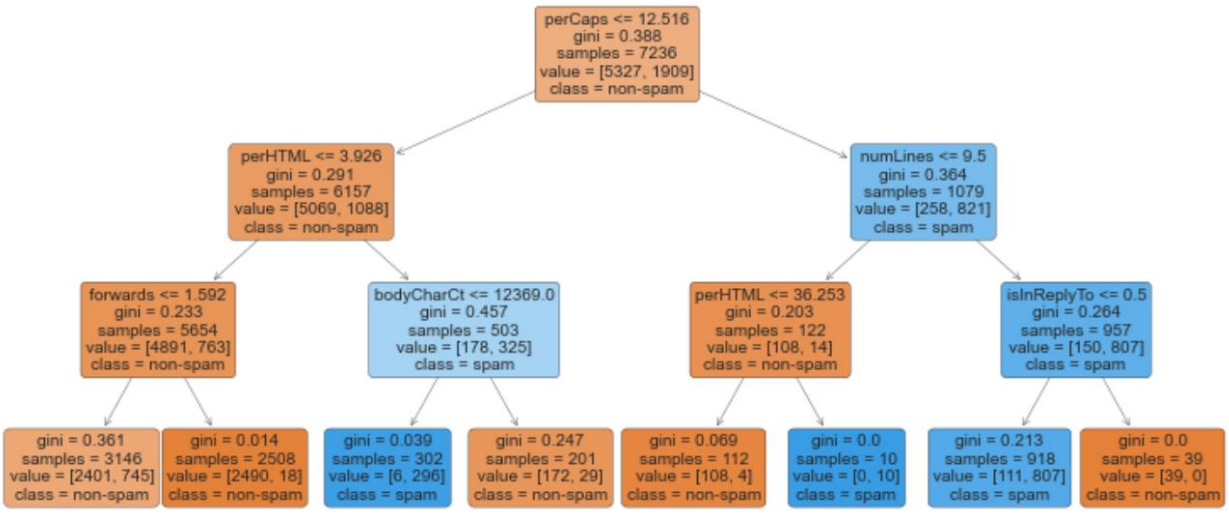


Fig. 21 A simple classification tree for the given dataset

5.3 Decision Making for a Spam and a Non-Spam Email

By using a similar process as the one shown during the tree visualization, we can list all the decision rules used by the full optimized model with a `max_depth` of 23. Fig. 22 shows the decision path for a non-spam email. We notice that the decision tree started by checking if the percent HTML tags (`perHTML`) is below a threshold of 3.93%. The observed email had no HTML tags (0%) in line with what would be expected from a non-spam email. The next decision point checked if the percent of capitalized letters (`perCaps`) was less than 13.05%. This email only had 6.34% of capitalized letters. Again, this makes intuitive sense from the exploratory analysis which indicates that the non-spam emails had a much smaller percentage of words that were capitalized. Subsequently, the classifier checks for the forwards tags in the email (`forwards`), comparing it to threshold of 1.59. This email has 53.84 forwards tags again matching our intuition from the exploratory analysis. A few other comparisons were made, and the classifier was finally able to tag this correctly as a non-spam email. Since a lot of the "important" features were compared first and this email followed many of the norms of a non-spam email, the classifier only needed 10 levels to classify the email (although the tree had a total of 23 levels).

```
plot_path(0, clf1, X_test,y_test)
```

```
Actual Classification: False
Rules used to predict sample 0:
decision id node 0 : (X[0, 24 (perHTML)] (= 0.0) <= 3.9264414310455322)
decision id node 1 : (X[0, 22 (perCaps)] (= 6.340058) <= 13.051740169525146)
decision id node 2 : (X[0, 26 (forwards)] (= 53.84615) > 1.5920854806900024)
decision id node 386 : (X[0, 27 (avgWordLen)] (= 3.9885059999999997) > 3.0593059062957764)
decision id node 388 : (X[0, 14 (isDear)] (= False) <= 0.5)
decision id node 389 : (X[0, 25 (subBlanks)] (= 14.814810000000001) <= 30.8712100982666)
decision id node 390 : (X[0, 6 (multipartText)] (= False) <= 0.5)
decision id node 391 : (X[0, 25 (subBlanks)] (= 14.814810000000001) > 5.202702522277832)
decision id node 395 : (X[0, 0 (isRe)] (= True) > 0.5)
decision id node 411 : (X[0, 22 (perCaps)] (= 6.340058) <= 11.11745023727417)
Predicted  Classification: [False]
```

Fig. 22 Decision Path from the Optimized Model for a non-spam email.

Fig. 23 shows the decision path for a spam email. We notice from the first step that the decision tree is already able to diverge away from the path taken by the non-spam email in the previous example. This email has a percent of HTML tags (`perHTML`) equal to 27.4% which is greater than the threshold of 3.93%, while the previous case had a 0% of HTML tags. This matches the intuition and exploratory analysis that indicates that the spam emails will have a higher percentage of HTML links. We again see some of the top features such as `bodyCharCt` and `perHTML` showing up in the subsequent comparisons. The comparison for `bodyCharCt` is a little non-intuitive. We noted from the exploratory analysis that the spam emails had a slightly higher number of lines and character count than non-spam emails, but also that there was considerable overlap in the distribution. In this case, the split path taken is for the comparison that is less than the threshold. This might have been done by the tree to fine tuning the decision to reduce any false positive or false negative classification.

```
plot_path(4, clf1, X_test,y_test)
```

```
Actual Classification: True
Rules used to predict sample 4:
decision id node 0 : (X[4, 24 (perHTML)] (= 27.41228) > 3.9264414310455322)
decision id node 516 : (X[4, 17 (bodyCharCt)] (= 7695) <= 18434.5)
decision id node 517 : (X[4, 17 (bodyCharCt)] (= 7695) <= 12369.0)
decision id node 518 : (X[4, 24 (perHTML)] (= 27.41228) <= 93.61559295654297)
decision id node 519 : (X[4, 17 (bodyCharCt)] (= 7695) > 249.5)
decision id node 523 : (X[4, 17 (bodyCharCt)] (= 7695) <= 8664.0)
decision id node 524 : (X[4, 25 (subBlanks)] (= 5.128205) <= 12.63297986984253)
decision id node 525 : (X[4, 25 (subBlanks)] (= 5.128205) <= 12.390349864959717)
Predicted  Classification: [True]
```

Fig. 23 Decision Path from the Optimized Model for a spam email.

It is interesting to see how the first few decision points in the optimized model use similar features as the one in the simple tree in section 5.2. In fact, more complex models are mainly adding more decision points to improve the performance measure, but they may share a common core decision steps for the most influential predictors. We also notice the `bodyCharCt` predictor has been used four times and `subBlanks` has been used two times during the decision tree. It is common for decision trees to use a factor multiple times to best tune the final decision.

6. Conclusion

In this case study, we have explored some important parameters that can be used to separate spam from non-spam emails and build a spam classifier that has a very good overall performance. We weighted the spam classifier towards not classifying non-spam emails as spam since that could potentially send important emails to spam. This was done with the help of a customer metric which took a balanced approach between classifying spam as non-spam and non-spam as spam (*F1* score) but weighted the cost of classification of non-spam as spam slightly higher (*precision* of spam class). The spam classifier was optimized using Bayesian Optimization to fine tune its performance.

We also visualized how the model would make decisions in classifying unseen emails in the future. The decisions were made using one feature at a time and comparing the feature to a threshold (for numeric values) or to fixed values (for categorical or Boolean features). Based on a hierarchy of decisions (a tree), the model could eventually classify the email as either spam or not spam. We also explored the path taken by two emails (one spam and one non-spam) down this hierarchy of decisions and confirmed that it correlated well with the exploratory analysis performed earlier.

However, it important to note that pure rule-based methods can be easy to hack and overcome for spammers. The rules discussed in this case study are well known and intelligent spammers can easily avoid these common mistakes in order to trick the spam detection system. Hence, we recommend two follow-up approaches. First, if we continue to use such rule based models, we should retrain the models from time to time using the latest email data. That way, the model can learn from the latest and greatest tricks used by the spammers. These new model rules will also need to be kept a secret so that spammers don't find ways to overcome them easily. Alternately, black box models (such as a neural network) could be used to build a spam detection system as well. Just as it would be difficult for the model developers to explain these models and why they take certain decisions, it will be equally difficult for hackers to come up with alternatives to fool such a spam detection system.

7. References

[1] "Sophicity," [Online]. Available: https://sophicity.com/ResourcesBlog.aspx?CNID=3384.

[2] D. Nolan and D. T. Lang, Data Science in R: A Case Studies Approach to Computational Reasoning and Problem Solving.

[3] "Yesware," [Online]. Available: https://www.yesware.com/blog/email-spam/.

[4] "scikit-learn," [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.impute.KNNImputer.html. [Accessed 04 10 2020].

[5] "Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization," Coursera, [Online]. Available: https://www.coursera.org/lecture/deep-neural-network/tuning-process-dknSn. [Accessed 04 10 2020].

[6] S.-l. developers, "Decision Trees," 2020. [Online]. Available: https://scikit-learn.org/stable/modules/tree.html.

[7] "Researchgate," [Online]. Available: https://www.researchgate.net/figure/A-schematic-view-of-the-learning-via-10-fold-cross-validation-procedure-and-validation_fig2_333640326. [Accessed 04 10 2020].

[8] "scikit-learn," [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html. [Accessed 04 10 2020].

[9] L. Breiman, J. Friedman, C. Stone and R. A. Olshen, Classification and Regression Trees (Wadsworth Statistics/Probability), 1984.

[10] D. Engels and R. Slater, *Quantifying the World,* Southern Methodist University.

8. Appendix

The entire code used for this case study is available in Python Notebook format on the following GitHub repository: https://github.com/ngupta23/ds7333_qtw/tree/master/case_study_3/analysis/master