Case Study 3, Unit 6 - Spam Classifier
Daniel Davieau, Lei Jiang, Kristen Rollins
18 February 2020

Introduction

As we have all likely experienced, unwanted emails, also known as spam, can be an irritation to our inboxes and thus our increasingly electronic lives. Spam filters have become standard email extension and can perform quite well, often relieving our inboxes of that burden. Even more frustrating, however, may be when an email that is actually important is mistakenly sent to a spam filter and neglected. Clearly spam filters are not perfect, so it is useful to investigate how they make their decisions and tradeoffs. In this study we build our own spam classifiers to explore two different approaches of detecting spam. We were able to explore and tweak a tree-based classifier to achieve an F1 score of 87%, as well as modify a Naive Bayes classifier to achieve a Type I error rate under 1%.

Background

Our spam filter relies on data from SpamAssassin (https://spamassassin.apache.org/), consisting of over 9,000 emails that have been hand-labeled as spam or "ham" (i.e. not spam, wanted emails). The Nolan and Lang textbook, *Data Science in R*, can be referenced for a detailed description of how the SpamAssassin data was extracted, cleaned and loaded for analysis. Features were also derived from the content of the messages, such as how many capitals were in the subject line, or how many attachments were included. A baseline Naive Bayes classifier was created on this data with a ⅔ training, ⅓ testing split, and subsequently a tree-based recursive partitioning model was created using these new features. Our first goal was to understand the parameters used for the tree-based model, and tweak them to possibly improve the spam filter's performance. Our second goal was to use cross-validation to select a τ threshold for the Naive Bayes model to attempt to achieve a Type I error rate of 1%.

Methodology

We first wanted to explore the classification tree model created in the textbook. Several features were derived from the headers and content of all the emails, such as `isYelling` (subject line is in all caps) and `subExcCt` (number of exclamation marks in the subject line), which are things that we subconsciously use to identify spam ourselves. With these characteristics we can create a tree-based model to classify spam programmatically. The method we will use here is called recursive partitioning. In this method, the messages are split into two groups based on some value of one of the features; then these separate groups are again split based on other features. These splits continue recursively until we have partitioned the messages into subgroups that are nearly all "ham" or spam. This algorithm results in a decision tree, or classification tree, which is very valuable in terms of interpretation since we can trace exactly how a message came to be classified as ham or spam. The R package `rpart` was used in the textbook to implement decision trees, and we employed this package as well.

The second type of model we examined was a Naive Bayes classifier. This model takes the approach of using probabilities to classify our test set of messages based on probabilities of words occurring in the training set. Using Bayes' Rule, we can express our goal with the following formula, which denotes the probability that a message is spam given the content of the message.

$$\mathbb{P}(\text{message is spam} \mid \text{message content}) = \frac{\mathbb{P}(\text{message content} \mid \text{spam})\mathbb{P}(\text{spam})}{\mathbb{P}(\text{message content})}$$

To simplify this equation, the model (naively) assumes that the chance for a particular word to occur in a spam message is independent of all other words in the message. This is definitely not the case, as words are not randomly selected, but the simplification has proven to actually work well in identifying spam. The textbook describes the full process of simplifying the equation in section 3.4., and what we are left with is calculating the following.

$$\begin{aligned} &\sum_{\text{words in message}} \log \mathbb{P}(\text{word present} \mid \text{spam}) - \log \mathbb{P}(\text{word present} \mid \text{ham}) \\ &\quad + \sum_{\text{words not in message}} \log \mathbb{P}(\text{word absent} \mid \text{spam}) - \log \mathbb{P}(\text{word absent} \mid \text{ham}) \\ &\quad\quad + \quad \log \mathbb{P}(\text{spam}) - \log \mathbb{P}(\text{ham}) \end{aligned}$$

The text also noted that logs were introduced because taking logs typically yields better statistical properties. So this quantity above that we will compute for each email is its log likelihood ratio (LLR). If the LLR is above a certain threshold τ (tau), then we will classify the message as spam, otherwise we will identify it as ham.

Lastly, the metrics we will use to evaluate our models are important to understand as well. The textbook largely uses Type I and Type II error, but we would like to introduce precision, recall, and F1 scores where possible. Type I error is the percent of false positives while Type II error is the percent false negatives (where we consider spam to be the "positive" class). Precision is how many correct spam predictions we made divided by how many total we predicted to be spam. Conversely, recall is how many correct spam predictions we made divided by the total actual spam messages. F1 is a balanced score of precision and recall for the positive class. We assume that it is preferable to classify spam as ham, rather than ham as spam. Therefore we will tend towards keeping Type I error low and precision high, but to make the spam filters most effective we will most strongly emphasize high F1 scores.

Results

Question 19: Decision Tree Tuning

Question 19 in the Nolan and Lang text asked us to consider other parameters that can control the recursive partitioning process, specifically in the `rpart.control()` documentation. The primary parameters we explored were `maxdepth` and `cp`, as we felt that these were the most influential factors for the resulting tree structure. Based on the documentation, the `maxdepth` parameter defaults to 30 and sets the maximum depth of any node of the final tree. In addition, `cp` is a complexity parameter that is used to reduce overfitting or underfitting, as a certain split must improve the tree's performance by at least `cp` to be included. If `cp` is high (maximum of 1), then fewer splits will be made and it has the risk of underfitting the data. If `cp` is very low (minimum of 0), then the tree will continue to grow without restrictions and it will likely overfit the data.

We first experimented with changing the complexity parameter while holding others constant. Figure 3 in the Appendix demonstrates the results of tweaking `cp` by itself, with the values 0.0001, 0.002, and 0.01. As seen, the smallest `cp` value resulted in the deepest, most complex tree, while the progressively larger `cp` values resulted in progressively shallower trees. This aligns with our understanding of how the complexity factor affects the model, i.e. a higher complexity value adds more restriction to how much the tree is allowed to grow. Next we experimented with changing both `maxdepth` and `cp`. Here we allowed the max depth to take the values of 4, 6, and 10, while letting complexity again shift between 0.0001, 0.002, and 0.01. These results can be found in Table 2 in the Appendix. From these combinations we saw the trend that lower max depth and larger cp lead to simpler trees but decreased performance. From these results we decided it was not worth considering trees with `maxdepth` less than 10.

Table 1 demonstrates the results of some of the additional experimentation with tree pruning, using the following guidance from Ben Gorman (referenced in Appendix): "As a rule of thumb, it's best to prune a decision tree using the cp of smallest tree that is within one standard deviation of the tree with the smallest xerror." With this direction we first fit trees with a very small complexity parameter, then pruned them back to an optimal `cp` based on model summary information. Additionally, the final entry in the table was found through a grid search across cp values using the caret package, from which we selected the `cp` yielding the highest F1 score.

Table 1. Selection of rpart() Parameter Tuning Results

Model Name	**CP**	**Max Depth**	**F1-Score**	**Precision**	**Recall**
rpartFitExperiement	0.00229453	30	0.8679722046	0.87627551020	0.8598247810
rpartFit0	0.0001	10	0.86821705426	0.89719626168	0.8410513141
Pruned rpartFit0	0.0059449	10	0.86095238095	0.87371134021	0.8485607009
rpartFit10	0.0001	14	0.86900958466	0.88772845953	0.8510638298
Pruned rpartFit10	0.0021902	14	0.86763774541	0.87820512820	0.8573216521
rpartFit12	0.00075	14	0.87078294080	0.88601036269	0.8560700876

From these results we found that after pruning, the F1 score and precision slightly decreased while recall increased compared to the corresponding non-pruned model, since the cp values were increased. Though the scores are slightly better for the non-pruned trees, in some cases we might prefer the pruned trees because they are easier to interpret. Figure 4 in the Appendix shows the tree structure we would recommend if performance is the most important factor, which is the rpartFit12 model from Table 1. This decision tree has the best F1 score of 87%, meaning it is most balanced and high performing in precision and recall, but it is quite unwieldy and the tree visual has too low of resolution to read and interpret. Therefore for interpretation we would recommend the pruned rpartFit0 model, found in Figure 1.

In this decision tree, the first feature that is considered is `perCaps`, which is the percentage of capital letters in the message body. At this node, the data is split based on whether there were less than 13% or more than (or exactly) 13% capital letters. If we were to stop at this node, the model would predict the majority class which is false (i.e. ham) and thus misclassify all 1598 spam messages. Continuing through the tree, however, the messages are partitioned on various features (described on page 151 of the text) to give further information from which to make predictions. The tree was constructed so that the final (leaf) nodes would be as homogenous as possible within the constraints of the max depth, cp and other parameters specified. While this tree model performs slightly worse than a more complex model, it is very interpretable and also reduces the risk of overfitting for future predictions.

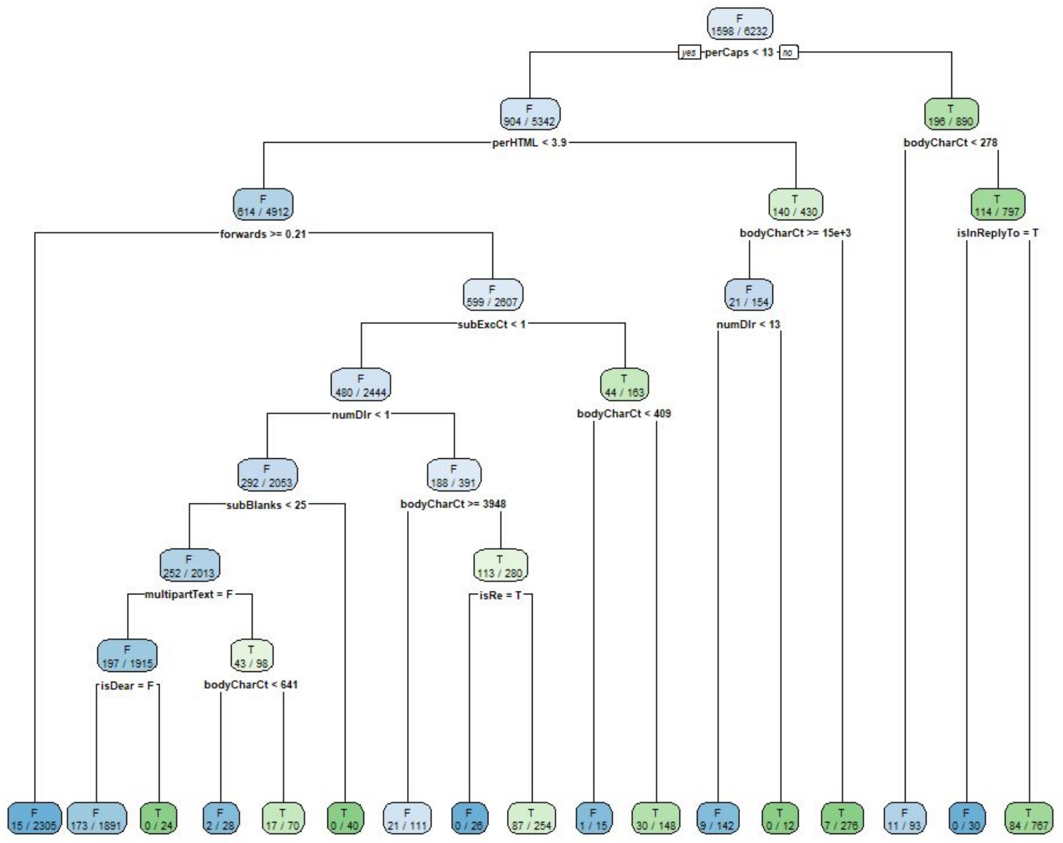

Figure 1. Pruned Decision Tree (Pruned rpartFit0)

Question 20: Naive Bayes Tuning

In Question 20 from the Nolan and Lang text, we were tasked with adjusting their Naive Bayes model to achieve a Type I error rate of 1%. Having such a low Type I (or false positive) error rate may be desirable to ensure that important emails do not get sent to a spam folder. The text also asked us to perform cross-fold validation when achieving this error, which is a valuable technique to make sure a model is performing consistently across a dataset. However, they gave instructions to perform cross validation on only the training set, but we felt that this would waste too much data (as their training set was only ⅔ of the whole dataset). Thus we chose to perform 5-fold cross validation on the full dataset, and evaluated our model by averaging the errors of the five left out test sets.

The result we found from this exercise was that a τ threshold of 30 yielded a Type I error rate of 0.01, or 0.009495 to be exact. This log odds threshold was required to obtain the 1% false positive rate for the pooled log likelihood ratios from each of our five cross-validated test sets. While this low false positive rate appears impressive, meaning that hardly any legitimate emails will be classified as spam, the major tradeoff we found was that this τ threshold inflated the Type II error rate to 0.43 across our test sets. In other words, with this threshold the classifier would have mistakenly sent 43% of actual spam emails to the normal inbox. This tradeoff is visualized in Figure 2. We are unsure why the Type II error increased so much, but if our calculation is correct, we would not recommend making this adjustment because of the imbalance it causes. While misclassifying legitimate emails is undesirable, we do not believe it is so unacceptable that we would make this adjustment, as the benefit of the spam filter drops so significantly.

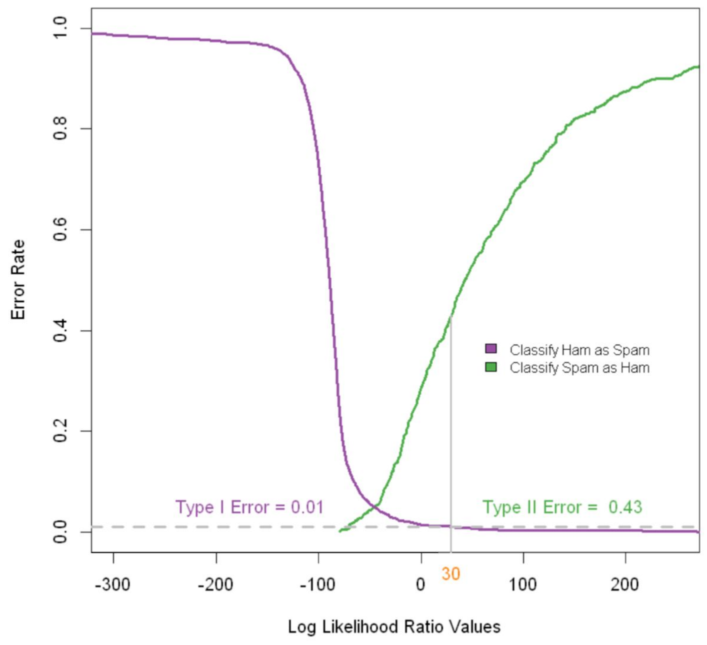

Figure 2. Determining Threshold for Log Likelihood Ratios

Conclusions

In this analysis we explored tradeoffs that can occur in both tree-based models and Naive Bayes classifiers, in the context of creating spam filters. First of all, it is important to note that the decisions made in the Nolan and Lang text, such as how they handled capitalization and punctuation, as well as which derived features they chose to create, certainly affected the characteristics and performance of our models. Further analysis with more advanced NLP (natural language processing) techniques and additional derived features could be explored to improve the performance of the models.

With the textbook's process in place, we first explored tuning the parameters of the recursive partitioning model. We found that tweaking the parameters can have a large impact on the performance as well as interpretability of the classifier. Specifically, we found that lower max depth and higher cp resulted in simpler models but decreased performance, with the reverse being true as well up to a certain point. We also adjusted the textbook's Naive Bayes classifier to achieve a 1% Type I error rate across five cross-validation testing folds. The tradeoff here, however, was that the Type II error expanded to 43% over the same testing data. We would not recommend driving down the false positive rate so much that the spam filter is nearly useless. Nevertheless, these types of tradeoffs are important to thoughtfully consider for any type of model based on the application and stakeholders' needs.

References

- Nolan, D., and Temple Lang, D. (2015), Data Science in R: A Case Studies Approach to Computational Reasoning and Problem Solving. Boca Raton, FL: CRC Press (NTL)
- R code provided by Dr. Robert Slater, adapted from http://rdatasciencecases.org/
- https://www.gormanalysis.com/blog/decision-trees-in-r-using-rpart/

Appendix

Supplemental Figures/Tables

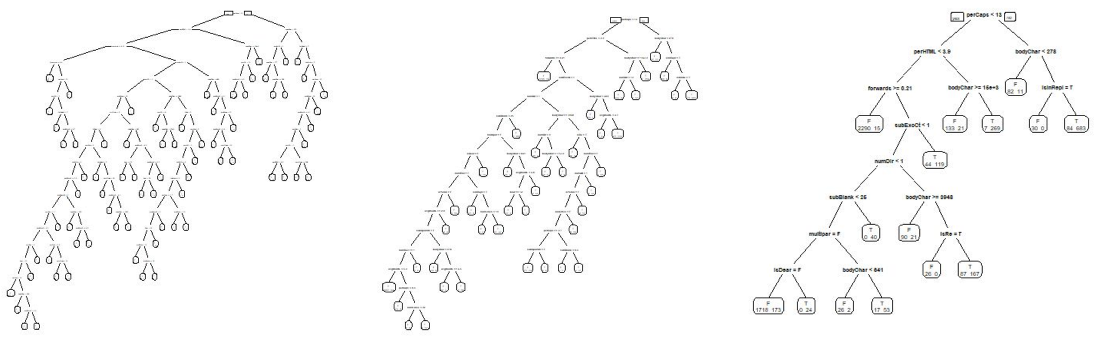

Figure 3. rpart() tuning: cp = 0.0001 (left), cp = 0.002 (middle), and cp = 0.01 (right)

Table 2. Results of Tuning MaxDepth and CP rpart() Parameters

Model Name	CP	Max Depth	F1-Score	Precision	Recall
rpartFit1	0.0001	10	0.86821705426	0.89719626168	0.84105131414
rpartFit2	0.002	10	0.86615186615	0.89139072848	0.84230287860
rpartFit3	0.01	10	0.84355444305	0.84355444305	0.84355444305
rpartFit4	0.0001	6	0.81680892974	0.85911602210	0.77847309136
rpartFit5	0.002	6	0.81710526316	0.86130374480	0.77722152691
rpartFit6	0.01	6	0.80517799353	0.83378016086	0.77847309136
rpartFit7	0.0001	4	0.75299085151	0.86012861736	0.66958698373
rpartFit8	0.002	4	0.75299085151	0.86012861736	0.66958698373
rpartFit9	0.01	4	0.74824191280	0.85393258427	0.66583229036

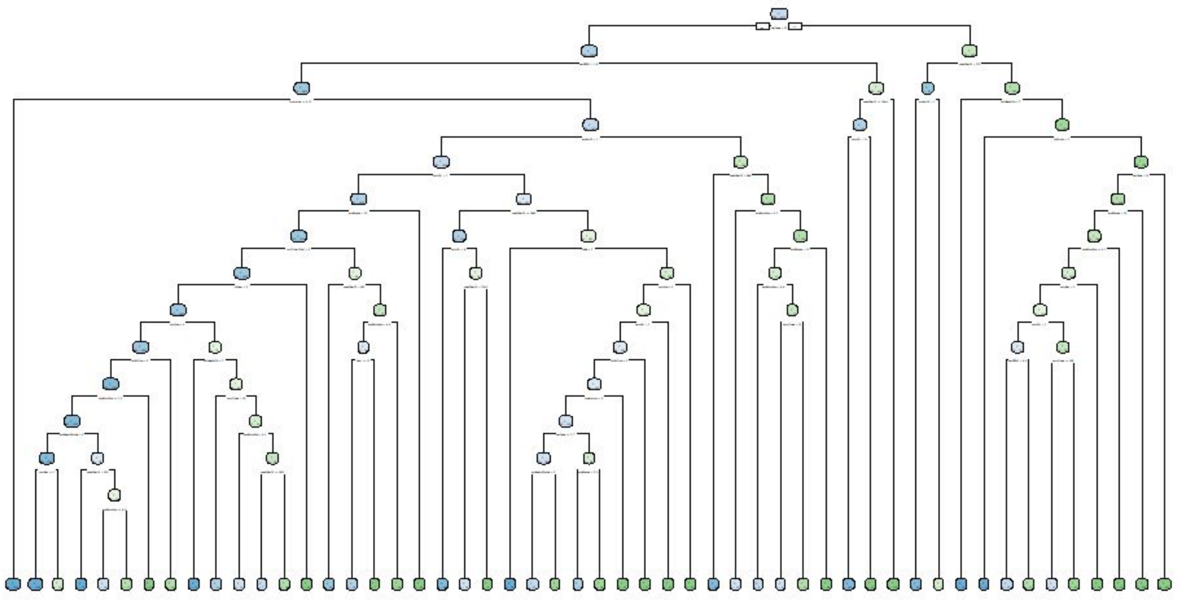

Figure 4. Unpruned Decision Tree (rpartFit12)

Codebase

Our code has been included in a zipped folder called lJiang_kRollins_dDavieauCase3_Code.zip. The files in this folder include:

- lJiang_kRollins_dDavieauCaseStudy3_Final.ipynb - our primary code base, where we processed the data and explored the tree based model for question 19
- Question20_ByTheBook.ipynb - where we performed question 20 analysis based on the textbook

Assignment

- **Q.19** Consider the other parameters that can be used to control the recursive partitioning process. Read the documentation for them in the rpart.control() documentation. Also, carry out an Internet search for more information on how to tweak the rpart() tuning parameters. Experiment with values for these parameters. Do the trees that result make sense with your understanding of how the parameters are used? Can you improve the prediction using them?
- **Q.20** In Section 3.6.3 we used the test set that we had put aside to both select τ, the threshold for the log odds, and to evaluate the Type I and II errors incurred when we use this threshold. Ideally, we choose τ from another set of messages that is both independent of our training data and our test data. The method of cross-validation is designed to use the training set for training and validating the model. Implement 5-fold cross-validation to choose τ and assess the error rate with our training data. To do this, follow the steps:
    a. Use the sample() function to permute the indices of the training set, and organize these permuted indices into 5 equal-size sets, called folds.
    b. For each fold, take the corresponding subset from the training data to use as a ‘test’ set. Use the remaining messages in the training data as the training set. Apply the functions developed in Section 3.6 to estimate the probabilities that a word occurs in a message given it is spam or ham, and use these probabilities to compute the log likelihood ratio for the messages in the training set.
    c. Pool all of the LLR values from the messages in all of the folds, i.e., from all of the training data, and use these values and the typeIErrorRate() function to select a threshold that achieves a 1% Type I error.
    d. Apply this threshold to our original/real test set and find its Type I and Type II errors.