Ryan Herrin, Luke Stodgel
DS7333
2/10/2023

Email Spam Detection using Naive Bayes

Introduction

Spam emails can be very disruptive to productivity at a company. If an employee isn't clear on what email in their inbox is spam, this can cause major problems such as getting hacked, compromising the company's security or even their *customer's* security.

In this project, we performed classification using Multinomial Naive Bayes on a data set that we created using emails that were pre-labeled as spam and not_spam.

Some questions we will answer in this project are: is it more important to make a stronger spam filter and risk accidentally marking legitimate emails as spam? Or would you rather have a less restrictive filter and let some spam get into your inbox?

Methods

Our dataset contained a file name, word counts for every word found in our email bank, and the target variable, is_spam. The dataset had 9348 rows and 49508 columns.

Creating the dataset

To build our dataset, instead of using sklearn's countvectorizer Python package, we created our own count-vectorizer function. This allowed us to create more customizable data features. We also implemented Python's multiprocessing library to make creating the initial dataframe faster. We stored all of the email data as dictionaries and then transformed it into a pandas dataframe.

To begin building the data set we read in all of the emails' *paths*. We found four files in each directory called 'cmds', which we discovered were not emails, but were computer scripts used to move all the emails into their corresponding directories. We told our script to ignore the 'cmds' files and then to read and process all of the emails. To create our target feature, 'is_spam', we checked if "_ham" was present in each email's file path. If "_ham" was present, we assigned a 0 (not spam), otherwise we assigned a 1 (is spam).

Next, we used the built-in python 'email' package to read in the emails as binary. We extracted the email body by walking through the email using the email.get_payload() function. If the payload was HTML, then we used Python's HTMLParser package to obtain the stripped payload text.

Finally, we processed the payload data. Here are a few example columns we made: *total_words*: Total number of words in the email, *total_symbols*: Total number of symbols, *first_letter_cap*: Number of words that had the first letter capitalized, *all_cap_words*: Number of words where all letter were capitalized, *avg_line_len*: Average length of the lines (excluding blanks), *line_count*: Number of non blank lines. We also made a column for total occurrences of non-alphanumeric symbols per email and several columns for each different symbol and their counts per email. Lastly, we made columns named after each word, containing their word counts per each email.

To avoid problems or word duplication in this sparse matrix of word counts, we removed all symbols from words before getting word counts (this fixed the problem of duplicates such as *there* and *there's*). Also, we converted all words to lowercase to avoid duplicate words benign counted because of capitalizations. After all of the features (file name, symbols and word counts) for each email were created and stored in a dictionary, we combined and flattened them into a single dataframe.

Why did we pick Multinomial Naive Bayes?

After reading the descriptions of the different Naive Bayes classifiers on sklearn's website, we picked *multinomial* because we thought it would work best with our data set compared to the others. In the description of multinomial Naive Bayes from the website, it says it is "suitable for classification with discrete features (e.g., word counts for text classification)". This is exactly what our data set is made of.

GridsearchCV

We used gridsearchcv to find the best parameters for our model. The parameter grid we used looked like this: alpha = 0.01, 0.1, 1.0, and fit_prior = True, False. We would have tested more alpha values, but the gridsearch took so long to finish (because the data set was so wide and sparse) that we had to reduce the parameter grid to what was mentioned in the previous sentence. In our gridsearch we also used: scoring=precision and cv=5. We picked precision as our scoring metric because we thought it was the most important metric in this spam email classification problem. Precision is the proportion of true positives among all predicted positives. By using precision, we tried to maximize the true positive rate, while minimizing the false positive rate. In other words, it is ok for some spam emails to get through into our inbox but we certainly don't want any legitimate emails to be classified as spam. The best parameters returned by gridsearchcv were alpha=0.01, fit_prior=False.

We also tested using scoring=f1_score in our gridsearchcv function, and the best_params returned were the same as when we used scoring=precision.

In the end we stuck with scoring=precision because of what we said previously about precision being the most important metric for our objective.

Finding the Best Threshold for the Predicted Probabilities Returned by our Model

In our script we created a function to find the best threshold to use when classifying the predicted probabilities returned by our model. In the function, we created a train, test split, found predicted probabilities and then tested thresholds from 0.01 to 1.0, in increments of 0.01, in order to find the best threshold. We picked the best threshold based on the one that returned the highest precision score.

Using similar reasoning as before, we picked precision over all other metrics to find the best threshold so that we could maximize the true positive rate, while minimizing the false positive rate.

In our final model, we also tested using a best_threshold based on f1_score. The metrics for both the f1_score-based-threshold-model and the precision-based-threshold-model resulted in similar but slightly different results. The accuracies and f1_scores of both of the models were almost identical, but the recall of the f1_score-based-threshold-model was about 0.01 higher and the precision was about 0.01 less than the precision-based-threshold-model. The thresholds that were returned when using f1_score as the basis averaged between .7 and .8 while the thresholds returned based on precision scoring averaged between .01 and .2.

Finally, we used the precision-based threshold for our final model because it had the highest scoring precision and also all of the other metrics were on par with the model that was based on f1_score.

Results

Here is a line graph that shows accuracy, precision, recall, and f1_score at each threshold:

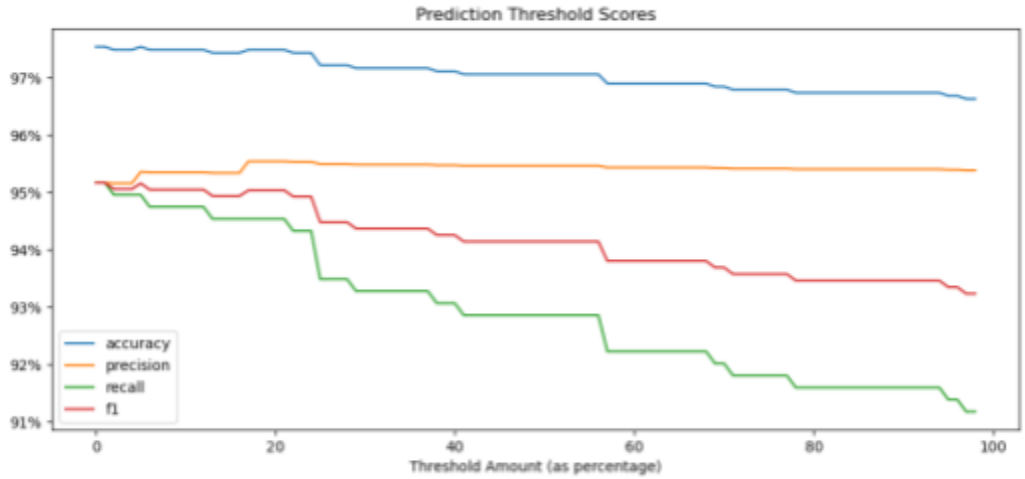

Figure 1 - Finding the Optimal Threshold

To find the best threshold for our final model, we ran our find_best_threshold function 5 times and used the average.

As shown in Figure 1, lower thresholds returned overall the highest metrics. And since we picked our final threshold based on the model's highest precision score, here we would have picked a threshold of around 0.2.

Here is a box plot showing the accuracy, precision, recall and f1_score from our final model:

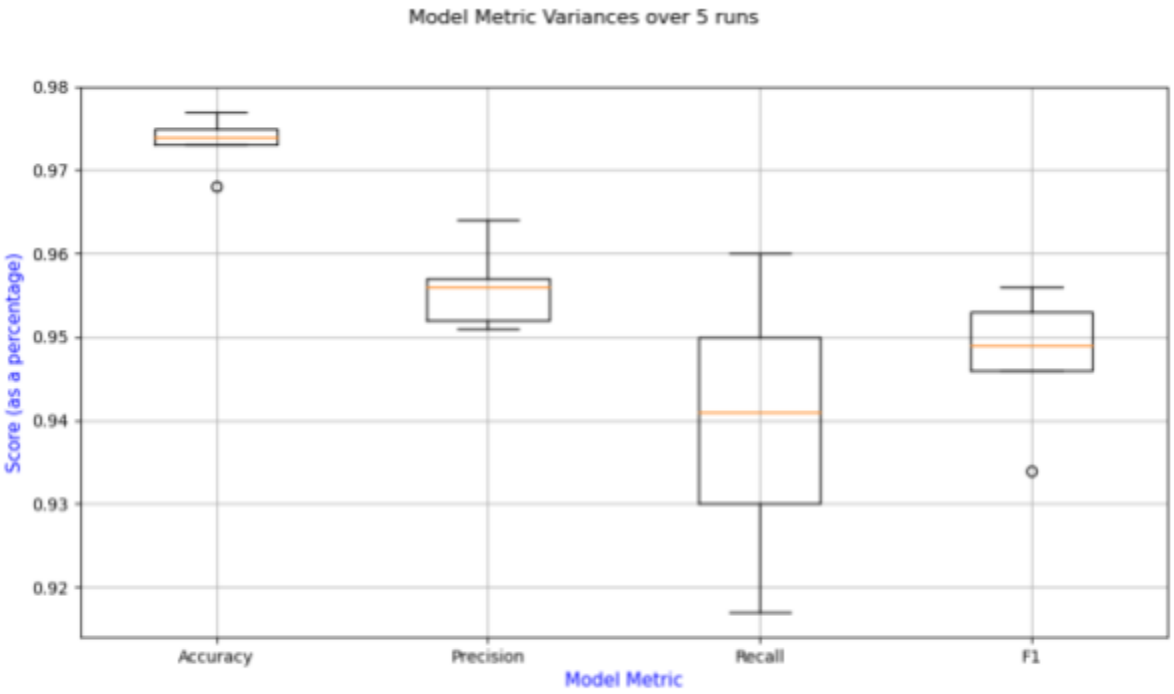

Figure 2 - Performance of Final Model

Figure 2 (above) shows boxplots describing model results from fitting 5 models each with different train/test splits. This allows us to generate mean and variance metrics and to assess our models' performance. The orange lines within the boxes of the chart represent each metric's mean, while the box and whiskers represent each metric's variance. The smaller the variance, the more confident we can be in our model's consistency. All of the means of the models are above 94%, while the lower and upper quartiles for variance is 2% for recall and less than 1% for accuracy, precision and f1_score. This shows that our models consistently return high performance metrics.

Below are descriptions of the results from Figure 1 and their context within this study:

Accuracy: Mean = 97.3% | Variance = 0.001%

Accuracy is determined by the number of correctly predicted instances of the total number of instances. Our model had very high accuracy, and made very accurate predictions. But, while accuracy is important, we found maximizing precision to be more important in this problem context (please see the Gridsearch subsection of the Methods section for an in-depth explanation of why).

Precision: Mean = 95.6% | Variance = 0.002%

In the context of this spam email classification problem, precision is our most important metric because it tells us how many false positives were predicted. The equation for precision is: (true positives / (true positives / false positives)). The more false positives present means the lower the precision score. During testing we tuned the hyperparameters in our gridsearch function and tuned our predicted

probabilities thresholds to provide maximum precision. As you can see this resulted in a very high performing model.

Recall: Mean = 94% | Variance = 0.023%

Recall is similar to precision except it measures false negatives rates instead of false positive rates. The equation for recall is: (true positives / (true positives + false negatives)). In this context, a lower recall would mean that some spam would be labeled as non-spam. While this is an important metric, we pushed our model to focus more on maximizing precision and preventing false positives. Making our model favor precision did not cause our recall to suffer by more than 1%, so we decided we did what was appropriate given this problem context.

F1_Score: Mean = 94.8% | Variance = 0.006%

F1_score is the harmonic mean of precision and recall. The equation for f1_score is this: 2 * (precision * recall) / (precision + recall). We considered using f1_score to find our best threshold, but we instead opted to use precision. Our reasoning behind this can be found in the "Methods->Finding the Best Threshold for the Predicted Probabilities Returned by our Model" section of this paper.

Conclusion

In conclusion we used Multinomial Naive Bayes with parameters: alpha = 0.01, fit_prior = false. Our final model returned an average accuracy of 97.3%, average recall of 94%, average precision of 95.6%, and an average f1_score of 94.8%. Our best threshold for our model's predicted probabilities averaged between .01 and .2.

As an aside, if a company were to continue using this spam filter into the future, our advice would be to, every week, month or quarter, after adding more emails to the data set, update the best threshold and use that in their spam filter.

Appendix A

Link to main script:
https://github.com/Abillelatus/QTW-Case-Studies/blob/main/CaseStudy_3/MSDS-7333-CaseStudy-3.py

Link to full CaseStudy_3 full contents:
https://github.com/Abillelatus/QTW-Case-Studies/tree/main/CaseStudy_3