Spam Filtering using a Random Forest Classifier

CSDS 340: Machine Learning – Case Study 1

Clay Preusch & Grant Konkel

Choice of Algorithm:

Our solution was a `RandomForestClassifier()` (from sklearn.ensemble) that was preprocessed with `SelectFromModel()` (from sklearn.feature_selection) and then pipelined and fit on the best hyperparameters that we could find by using a hybrid of grid and random search. Our hyperparameters for Random Forest are shown below:

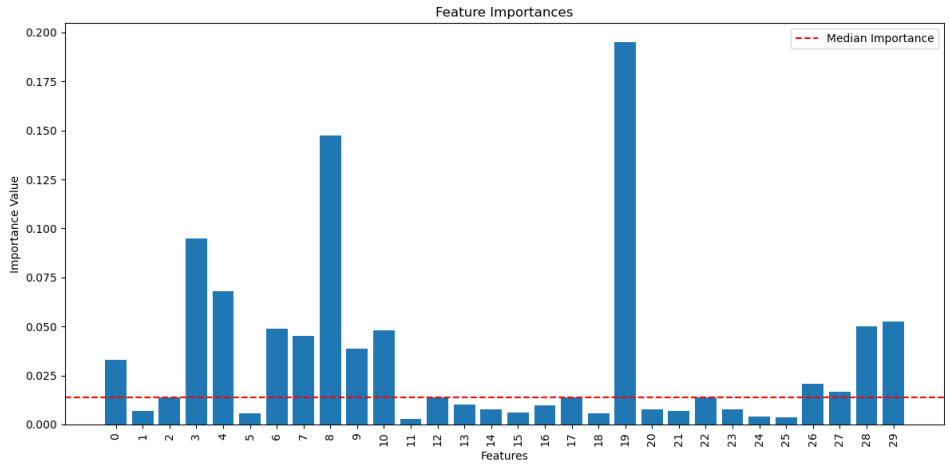
```
rf = RandomForestClassifier(
    n_estimators=1500,
    criterion='log_loss',
    max_depth=10,
    min_samples_split=2,
    min_samples_leaf=1,
    max_features='log2',
    bootstrap=False,
    random_state=seed
)
```

Initial Strategies: To find our solution, we started by following the advice of Professor Xu and focusing our attention on ensemble learning algorithms such as Gradient Boosting and Random Forest. With initial tests and comparisons, we saw more promising results with a Random Forest Classifier, so we continued with that choice before moving on to preprocessing and finding the best hyperparameters. For preprocessing, we tried a variety of algorithms, but ended up only using feature selection with `SelectFromModel()`. After finishing the preprocessing step, we utilized grid search and then randomized search to find hyperparameters that produced the highest test set accuracy (grid search to find an area where we had highest accuracies, and then random search on nearby parameters).

Preprocessing:

Once we decided to focus our attention on a Random Forest Classifier, the next step was preprocessing the data. This consisted of picking strategies for normalizing and standardizing the data, and for performing feature selection, extraction and transformation. After doing some research, we found that with Random Forest Classifiers, the key points lie in manipulating the features, but we still wanted to experiment with normalization and standardization. Comparing the results of normalized and non-normalized data, we agreed that the normalization was insignificant, and not applicable to our classifier. When experimenting with standardization, we tried several functions, including `MaxAbsScaler()`. We tried `MaxAbsScaler()`, as our research indicated that it handles sparsity effectively. However, after comparing it with other algorithms and our Random Forest Classifier without standardization, we found that none were necessary. This is

because tree-based classifiers draw decision boundaries based on feature thresholds, and are therefore not sensitive to the magnitude of features. For feature manipulation, we focused mostly on feature selection and dimension reduction using strategies such as PCA, LDA and `SelectFromModel()`. `SelectFromModel()` gave the best results, which was our expectation. This selection strategy plays well with the Random Forest Classifier, because it selects features that are deemed more important by a decision tree classifier, and a random forest is an ensemble of trees. This was implemented in our code by selecting features above the median "feature importance" (shown below). We experimented with feature extraction using a few strategies, but random forest works well with minimally processed data, so we ended up not finding any strategies that consistently improved our classifier. After finishing the preprocessing step, we concluded that only feature selection consistently and significantly improved our algorithm (which is in line with the research that we had done on Random Forest Classifiers), so we proceeded by only performing feature selection in our preprocessing step.



This graph demonstrates the feature "Importance Values" computed by our Random Forest Classifier. In scikit-learn, `feature_importances_` is a field generated after fitting a tree-based model. The median importance value was computed, and only features above this threshold were selected during the classification of the model. As this visual demonstrates, the Random Forest Classifier utilizes a select few features more than others when classifying examples. It deems these features as much more important. By selecting these important features with `SelectFromModel()`, the Random Forest Classifier can still perform very well, and save significantly in computation complexity.

Selection and Testing of Algorithm:

We initially selected an Ensemble Classifier because of the suggestion by Professor Xu, but we focused on a Random Forest Classifier because of how it performed in comparison to other classifiers, such as Gradient Boosting. In general, ensemble algorithms work really

well for large datasets, because they allow the classifier to try to classify every point well without the risk of too much overfitting. We also knew we wanted a non-linear decision boundary. By considering the classifiers we learned about in class, we had Ensemble Learning, Naive Bayes, K-nn and Kernel SVM to choose from. Naive Bayes makes large assumptions, and without knowing what each feature represents, conditional independence may not be a satisfactory classifier. K-nn is weak when dealing in high dimensional space. Kernel SVM tries to find a linearization of non-linear data, which can be incredibly hard with 30 features. Each of these has a somewhat significant drawback that ensemble doesn't, so we "selected" an ensemble algorithm and then tested these against each other to find which one worked better.

Testing and Hyperparameter Search: We utilized a combination of both Grid Search and Randomized Search when looking for the ideal hyperparameters. This step began by researching the various hyperparameter fields within Random Forest Classification, and deciding which choices were applicable to our data set and use case. We dealt with a lot of trial and error, learning how hyperparameters impact our data by simply trying out lots of iterations of searches. In the end, the procedure that led to our best hyperparameters was as follows:
First, we performed an extensive Grid Search over a large variety of hyperparameters. These fields were chosen based on our preliminary trial and error experiments. We knew we wanted to cover a large enough domain of possible hyperparameter values.

```
grid_search_params = {
    'randomforestclassifier__n_estimators': [500, 1000, 1500, 2000],
    'randomforestclassifier__criterion': ['gini', 'entropy', 'log_loss'],
    'randomforestclassifier__max_depth': [None, 10, 20, 30],
    'randomforestclassifier__min_samples_split': [2, 4, 6, 8, 10],
    'randomforestclassifier__min_samples_leaf': [1, 2, 3, 4],
    'randomforestclassifier__max_features': ['sqrt', 'log2'],
    'randomforestclassifier__bootstrap': [True, False]
}
```

This search was very computationally expensive, but the results led us in a good direction.

```
grid_best_params =  {
'randomforestclassifier__n_estimators': 2000
'randomforestclassifier__criterion': 'log_loss',
'randomforestclassifier__max_depth': 20,
'randomforestclassifier__min_samples_split': 2,
'randomforestclassifier__min_samples_leaf': 2,
'randomforestclassifier__max_features': 'log2',
'randomforestclassifier__bootstrap': False
}
```

The accuracies resulting from these parameters:

```
Test set AUC: 0.8701689286500958
TPR at FPR = 0.01: 0.27225130890052357
```

From here, we performed Randomized Search on a more precise domain to even further refine this search. After a few attempts, this led to our final best hyperparameters, which were listed on the first page of this report.

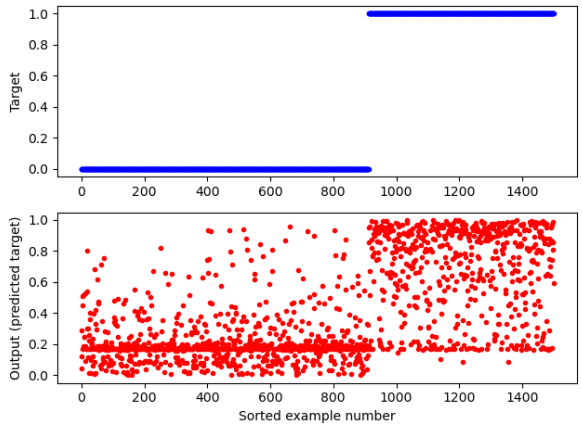
Hyperparameter Analysis: There were several hyperparameters that consistently resulted from our Grid and Randomized Searches. These were criterion '*log_loss*', max_features '*log2'*, and bootstrap '*False'*.
The criterion parameter measures the quality of each split during classification. Logarithmic loss performed well on our dataset because it emphasizes the penalization of incorrect confident predictions. This was critical in making our classifier more accurate and better calibrated.
The max_features parameter determines the number of features to consider when looking for the best split. Because our dataset has such high dimensionality, the choice of log2 restricted the number of features even more, making our process faster and reducing the overall variance of the model by making each tree in the forest more diverse.
Finally, the bootstrap parameter set to False meant that the entire dataset was used to build each tree in the forest. This allowed each tree to extract as much data as possible, and during experimentation, we found that this led to greater performance. A possible effect of not using bootstrapping is overfitting, but we avoided this by training over enough trees and carefully selecting our other hyperparameters accordingly.

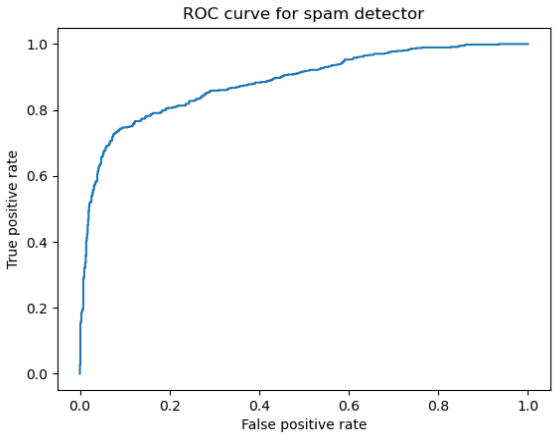
Recommendations:

The natural intuition and implication is that a personalized spam filter should be fit for the specific person. If someone receives a lot of very important emails every day, they may want the allowed false positive rate to be even lower than the standard '.01', because they cannot afford to miss any emails that they do not see. If someone is implementing this for an email address that does not get a lot of important information, they may want to allow a higher false positive rate in order to lower the false negative rate. If someone looks at both their spam and inbox and uses their judgment in partnership with the classification, they may not care at all about a particular TPR and FPR, and they may just care about optimizing the prediction accuracy. This being said, for most people (who will likely not view their inboxes simultaneously), TPR @ FPR = n will likely be one of the best ways to evaluate the quality of the classification. This evaluator allows the user to choose the classification "severity" that works best for them and choose the tradeoff between the false positives and the false

negatives. The feedback that we would ask for would probably be along the lines of collecting more samples. If a user is getting a lot of misclassified emails, and we could add those samples to the dataset, then we could run the classifier again and see the improvement. We could also collect data to see if the same types of emails are getting misclassified, and if so we could experiment with incorporating boosting (of some sort) into our algorithm, so we can prioritize correcting the emails that are being missed.

Results:





These two graphs work together to show the accuracy of our classifier. The first graph shows the ideal classification of emails vs our classification (using confidence levels in our classification). As you can see, the bottom graph is incredibly noisy, however, it is actually quite accurate for classifying emails.
The second graph shown is the TPR vs FPR curve that was set up by the professor and then run on our classifier. The curve is very steep initially, meaning that it will have a competitive TPR value when a low FPR is allowed, which is typically how this classifier would be run (for a real life person).

When evaluating our Random Forest Classifier on the entire set, our performance was evaluated on the following metrics:

```
Test set AUC: 0.8831157951017657
TPR at FPR = 0.01: 0.32809773123909247
```

We find this to be a pretty competitive classifier, based on the problem presented as well as in comparison to some of the top classifiers shown on the pre-submission leaderboard. Further, when evaluating the classifier on both datasets, where the testing and training samples are chosen randomly between the two, our classifier sees a very large improvement, particularly with the TPR @ FPR evaluator. This gives us confidence in our classifier's robustness as it applies to more diverse datasets.