Case Study 1 Report

Ethan Fang, Jackson Schuetzle

October 24, 2024

1 Short Summary

Our best model was a stacked ensemble composed of our best RandomForest model and best XGBoost model. Preprocessing included multivariate imputation and hyperparameter search via Tree-Parzen Estimators. More detail about the preprocessing, hyperparameter tuning, and best model parameters are expressed in the rest of the report.

2 Preprocessing

2.0.1 Imputation

Before any feature scaling/extraction was done, we needed to deal with missing values. Univariate imputation (mean, median, mode) mostly imputed zero-values or values extremely close to zero due to high zero rates. We define a *zero rate* as the ratio of samples that have value zero for some given feature. Consequently, we decided to use a multivariate imputation method to more accurately impute data.

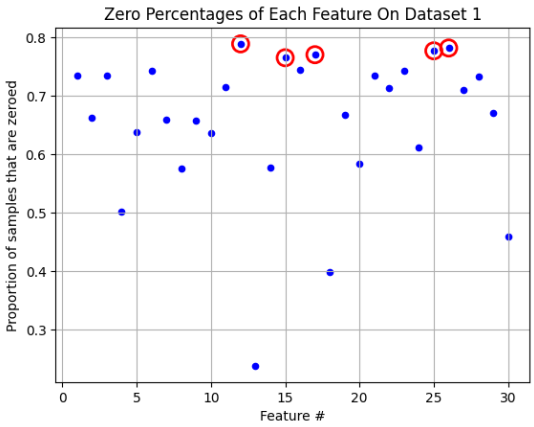

Figure 1: Zero rates for Dataset 1

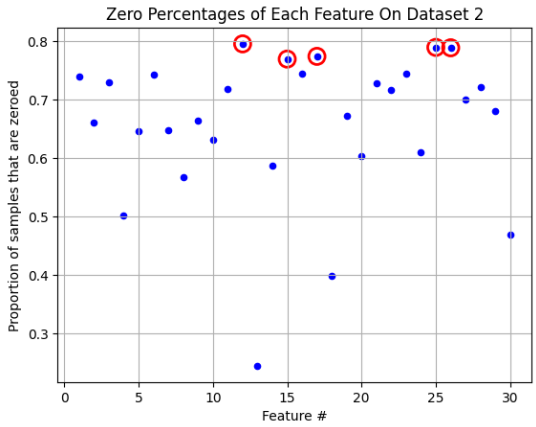

Figure 2: Zero rates for Dataset 2

The `IterativeImputer` sklearn class uses a round robin approach in order to complete the imputation. Each round, it selects some feature, y, and treats the rest of the features as the input, X. Then, it uses a classifier to train on (X, y), and finally uses the samples with missing data as the test set. Our motivation for choosing a classifier came from the sklearn documentation [1] where they experimented with 1) Bayesian Ridge classifier with default hyperparameters and 2) custom pipeline with a degree 2 polynomial kernel followed by Ridge regression. We decided to not move forward with the Bayesian Ridge classifier since it frequently imputed negative values (which have no meaning in the context of the case study).

2.0.2 TF-IDF

DeBarr and Wechsler [2] used *term frequency* and *inverse document frequency* in a random forest spam email classifier. After further research, we learned that tf-idf (a product of tf and idf) is frequently used in machine learning models that deal with language processing. The original data provided was already represented as the term frequency of the feature word, so all that was required was to calculate the IDF of each feature. The idea behind IDF is words that appear more frequently in the dataset are less informative than features that occur less frequently (but appear nonetheless). There are a handful of different equations used to calculate the IDF; we choose the following form

$$\text{IDF}(f) = \log\left(\frac{N}{N_f}\right)$$

where N is the total number of samples in the dataset and N_f is the number of samples that contain the word represented by feature f. Since it's assumed that the set of features provided are useful for classifying emails, we can assume that N_f will be nonzero for all features.

This preprocessing is likely good practice in this context; the data does not give the ordering of words nor does it give the actual words that the features represent, so we may not know how common the words are in regular language. **We choose not to include this preprocessing step in the final submission since the accuracy metrics tended to be lower when using TF-IDF**. However, this strategy would likely be more successful if the set of feature words contained words common to the language of the dataset.

Method	AUC	TPR at 1% FPR
With TF-IDF	0.846	0.216
Without TF-IDF	0.922	0.533

Table 1: Effect of TF-IDF on AUC and TPR at 1% FPR

2.0.3 Variance Analysis

Datasets where PCA can produce effective dimensionality reduction usually contain 1) features that are somewhat correlated with each other, and 2) features that have relatively high variance. Neither of two conditions hold in this dataset, as the covariance matrix contains no single entry with magnitude > 0.001. The graph of explained variance for each principal component are given below.

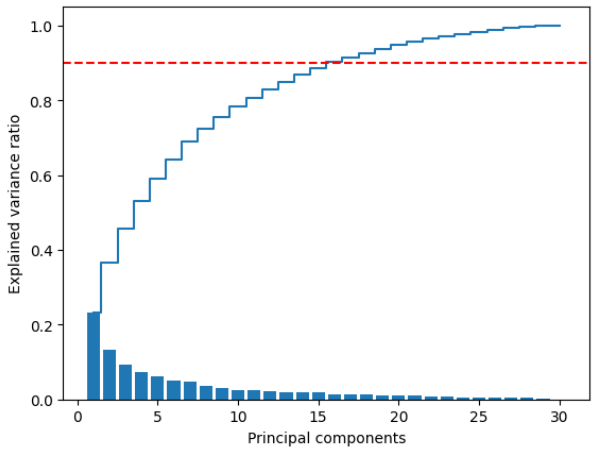

Figure 3: Explained Variances of Principal Components

PCA would not be an effective method of dimensionality reduction for this dataset since it would take 15 principal components in order to explain 90% of the total variance.

2.1 Splitting and Scoring Metrics

Our group started with an 80-20 train-test split and found this method to be quite good. In order to avoid test set leakage, we used cross validation via `cross_val_score` in sklearn with a 5-fold stratified method. By default, this function uses the default scoring method of the classifier being tested, which oftentimes is just a classification accuracy. However, the case study focuses on AUC and TPR at 1% FPR, so we decided to define custom scoring metrics via sklearn's scoring API. AUC is already a supported metric in sklearn, however TPR at 1% FPR is not. All that was required was to take the given `tprAtFPR` function and wrap it in a scorer object via the `make_scorer` function in sklearn.

When performing hyperparameter searches, it may be possible that the tuple with the highest AUC produced a lower TPR, and vice versa. To avoid this situation, we created a custom scoring metric, `auc_tpr_loss` that takes the difference between the output metric and some upper bound on what we believe the metric could be. For example, if we thought the highest AUC and TPR values were 0.93 and 0.6 respectively, then the custom scoring function would look like so

$$\text{custom metric} = \alpha(0.93 - \text{AUC}) + \beta(0.5 - \text{TPR at } 1\% \text{ FPR})$$

We include constant α and β as scaling factors to pay more attention to one metric if it seems too low. We ended up with $\alpha = \beta = 1$, so we did not make use of the constants. Even so, this format makes the scorer easily flexible.

2.2 Hyperparameter Tuning

At first, we used sklearn's `GridSearchCV` class in order to select optimal parameter tuples. However, ensemble methods generally tend to have greater number of hyperparameters, i.e. they include all the original parameters of the underlying classifier plus any new parameters for ensembling. This made it difficult to manually narrow down the hyperparameter ranges. Consequently, we decided to use a Bayesian optimization technique called **Tree-Structured Parzen Estimators**.

Techniques that use Bayesian Optimization are often higher performing that normal brute-force searches since the accuracy of previous hyperparameter tuples affect the decision of the next sample. Usually, this entails trying to maximize the posterior probability, $P(Y \mid X)$, where X is a parameter tuple and Y is some accuracy metric. TPE is slightly different in that it tries to model the reverse conditional $P(X \mid Y)$, the probability that a parameter tuple corresponds to the metric seen. Specifically, TPE separates samples into a "good group", G, and a "bad group", B, and then selects the tuple with a high probability of being in G and a low probability of being in B. After selection, it then adjusts black-box estimating model, and iteratively keeps testing new parameter tuples until some limit is reached.

So how exactly does TPE improve hyperparameter search? Simply put,

1. Speeds up model training by selecting future tuple samples in a smart way
2. Selects parameter values from a distribution instead of a concrete set of values
3. Picks hyperparameter tuples with higher accuracy metrics

Parameter Expression	Optimal Results	Fastest Results
(ordinal parameters) hp.uniform hp.quniform hp.loguniform hp.qloguniform	20 x # parameters	10 x # parameters
(categorical parameters) hp.choice	15 x total categorical breadth*	

Figure 4: General Guide for TPE Iterations [3]

To determine the number of iterations that should be performed for Bayesian optimization, we referred to the best practices highlighted in a blog by Databricks [3]. The article recommends that the number of evaluations should be proportional to the number of hyperparameters being optimized to ensure sufficient exploration of the search space.

3 Classifiers, Testing, and Results

The data provided motivated us to choose a supervised learner since the classes of the data are provided and completely known. Our group went straight to trying out **ensemble methods** since these models frequently outperform singular models. We tried out both bagging and boosting methods, and found decent results with both. To maximize our accuracy, we combined our bagging (RandomForest) and boosting (XGBoost) classifiers with the ensemble stacking method, using Logistic Regression as our final estimator.

3.1 Random Forest

Unlike other probabilistic models, decision trees don't really have many assumptions tied to them. During training, the model may encounter a group of features that are correlated and choose to focus on only one of them–deeming the others not important. However, this situation does not occur with our dataset since the data has been shown to not be correlated (in preprocessing section above).

The hyperparameters we considered during tuning were `critierion`, `max_depth`, `min_samples_split`, `n_estimators`, and `max_features`.requires the hyperparameters to be defined as probability distributions to be sampled from. Since nothing is known about which hyperparameter tuples will produce the best accuracies, the distributions we started with were uniform.

$$\begin{aligned} \text{n\_estimators} &= \text{IntUniform}(100, 300) \\ \text{max\_features} &= \text{Uniform}(0, 0.3) \\ \text{min\_samples\_split} &= \text{IntUniform}(2, 30) \\ \text{max\_depth} &= \text{IntUniform}(4, 16) \end{aligned}$$

We choose starting distributions in a way such that TPE could select a tuple of hyperparameters that (in theory) would produce a diverse forest. In the end, this was the case as our `max_features, max_depth,` and `min_samples_split` parameters encourage individually over-fit trees.

3.2 XGBoost

For defining the hyperparameter search space, we utilized insights from a comprehensive guide on XGBoost hyperparameter tuning [4]. This resource outlines the significance of each hyperparameter and recommends practical ranges based on empirical results.

Parameters like `max_depth, min_child_weight`, and `gamma` directly influence the depth and structure of the trees which affects the model's ability to capture the patterns within the data. This may make the model more prone to over-fitting, which is why parameters like `reg_alpha` and `reg_lambda` help by adding penalties to the loss function. `subsample` and `colsample_bytree` introduce randomness into the training process, which can improve the model's ability to generalize to unseen data. Finally, `learning_rate` and `n_estimators` govern how quickly the model adapts to the training data.

3.3 Ensemble Stacking

Using ensemble stacking to combine our Random Forest and XGBoost models offers significant benefits for enhancing our classification results. Random Forest, excels at reducing variance by bootstrap aggregating, thus providing robustness against overfitting and handling noisy data effectively. On the other hand, XGBoost utilizes boosting methods to reduce bias by sequentially focusing on the misclassified instances and capturing complex patterns and feature interactions within the dataset. By stacking these two models, we get "the best of both worlds" which balances the bias-variance trade-off more effectively.

3.4 Results

- Best RandomForest:
  {criterion: gini, max_depth: 20, max_features: 1, min_samples_split: 6, n_estimators: 205}
  AUC: 0.917
  TPR at 1% FPR: 0.437

- Best XGBoost:
  {colsample_bytree: 0.8877, gamma: 0.2739, learning_rate: 0.0074, max_depth: 7, min_child_weight: 10, n_estimators: 300, reg_alpha: 0.9868, reg_lambda: 0.2015, subsample: 0.7286}
  AUC: 0.909
  TPR at 1% FPR: 0.439

- **Final Model (Stacking) Performance:**
  AUC: 0.922
  TPR at 1% FPR: 0.533

4 Evaluation

The metrics used in this case study cover both Type I and Type II errors. In the context spam classification, both are of importance.

- Type I error occurs when our model classifies an email as spam when it indeed is not spam. Classifying an email as spam usually constitutes placing the email into a spam inbox that users rarely ever check. Therefore, a Type I error would lead the user to believe they didn't receive an email, or it would lead them to look through their spam folder which exposes them to more spam (increasing their vulnerability).

- Type II error occurs when our model doesn't classify an email as spam when it indeed was spam. This would place the email in the user's normal inbox, exposing them to spam directly.

Since we do care about both types of errors, it's good practice to use metrics that emphasize the rates at which the model commits these errors. Therefore, the current metrics the case study requires are likely a good choice moving forward since True Positive Rate takes into account the rate at which we commit Type II errors and False Positive Rate takes into account the rate at which we commit Type I errors.

If our algorithm were deployed as a spam filter, we could receive feedback by implementing buttons that allow the user to report an email from their primary inbox as spam. In future trainings of the model, these sample emails can have increased weights so that the classifier pays more attention to the correctness of their classification. Theoretically, this would allow us to reduce Type II errors. Getting feedback about Type I errors from users would be more difficult. This would require users to go into their spam folder and find emails that they consider to be not spam.

It's very common for businesses to use a spam filtering service that allows employees to report these happenings. You can encourage feedback by creating a point system and rewarding employees for giving feedback.

5 Works Cited

[1] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

[2] DeBarr, Dave and Harry Wechsler. "Spam Detection using Clustering, Random Forests, and Active Learning." (2009). link.

[3] Owen, Sean. "Use Hyperopt Optimally with Spark and Mlflow to Build Your Best Model." Databricks, 15 Apr. 2021, link.

[4] Banerjee, Prashant. "A Guide on XGBoost hyperparameters tuning." Kaggle, 2020, link.

6 Appendix

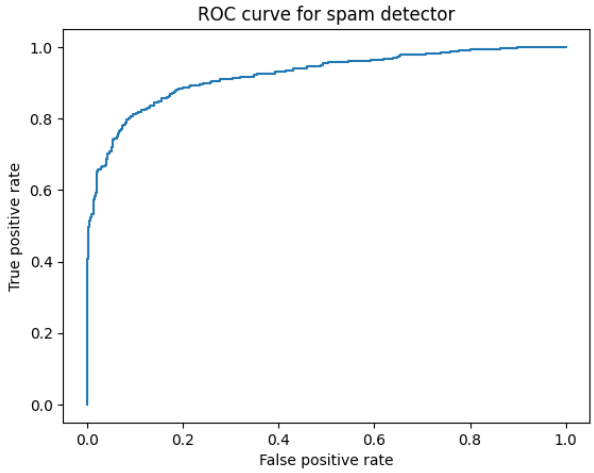

Figure 5: ROC Curve for Best Stacked Model