Capstone Report

1. Define

Main Submission File: fraud-detection.ipynb

Overview: The goal of this project is to classify real-word e-commerce transactions as fraudulent and non-fraudulent. This is a Kaggle competition using data provided by the Vesta Corporation. The competition is also sponsored by the IEEE Computational Intelligence Society (IEEE-CIS) and their researchers who want to improve the accuracy of fraud detection models.

This project would be about detecting fraudulent transactions, which is a part of a larger field of research known as anomaly detection. This field is being transformed as computers get better and better at analyzing large datasets. See [Chandola et al 2009] for a general overview of anomaly detection techniques and applications.

Problem Statement: As defined by Kaggle, "In this competition you are predicting the probability that an online transaction is fraudulent, as denoted by the binary target `isFraud`." The submission file contains the probability of fraud for a given transaction instead of a rounded number such as 0 or 1.

Metrics: The submission file will contain the ID for each *test* transaction and the corresponding probability that the transaction is fraudulent. Kaggle then calculates the area under the ROC curve. The y-axis of the ROC curve is the True Positive Rate (TPR) = TP / (TP + FN) and the x-axis is the False Positive Rate (FPR) = FP / (TN + FP). By examining different cutoff thresholds (when to predict 0 or 1), other than just 0.5, the ROC curve can be created. The curve represents the tradeoff between false positives and false negatives. The optimal AUC score would be 1, implying that the model perfectly classifies transactions.

This way of measuring classification tasks is better than simply calculating the accuracy of the predictions. In this data set, only 3.5% of the transactions are fraudulent. Thus, a model that predicts every transaction as NOT fraudulent would have an accuracy of 96.5%, giving the illusion that this is an effective model.

2. Analysis

Data Exploration: "The data is broken into two files *identity* and *transaction*, which are joined by TransactionID. Not all transactions have corresponding identity information." Thus, both the train and test set have two files (identity and transaction) that need to be merged to create one train table and one test table. The result is a training set of 590,540 examples and a test set of 506,691 examples, both with 432 features.

VESTA provided the names of the categorical features and I wrote a function that converts the data types of those features to 'object' types, since some of them were 'int64' or 'float64'.

Exploratory Visualization: With the overwhelming amount of feature variables, it seemed more prudent to take a general approach to visualizing the data. First, I examined the amount of missing values to see that some features had almost as many missing values as there were columns. Additionally, there were categorical variables (as defined by the competition sponsor) that had thousands of unique categories.

Subsequently, I plotted the class imbalance of the training data set. Of 590,540 training examples, only 20,663 were labelled fraudulent (1). I thought it might be interesting to compare the distributions of fraudulent transactions vs. non-fraud transactions. This might glean some insight into what makes a transaction fraudulent. A feature with a very different distribution when fraudulent provides lots of information. This early analysis may help in evaluating a model after the fact to see if the model decided a feature with different distributions to be important. Here are some interesting insights about the difference between fraudulent and non-fraud transactions in the **train** set:

- **TransactionAmt** – fraudulent transactions averaged $149 vs. $135 for non-fraud
- For the **ProductCD** variable, 39% of fraud transactions were labelled 'C', vs. 11% for non-fraud transactions
- **Card6** – 76% of non-fraud were debit while fraud transactions were almost evenly credit and debit
- **id_30** – fraud seemed to be less common on MAC OS as opposed to Windows and other operating systems

Since I wrote a function to compare the means of numerical features and the categorical distributions of features, I decided to also use this function to compare the train and test sets to see if there were major differences. Here are some interesting insights about the differences between the **train** and **test** set:

- **id_31** – all the browser versions seemed to be later, implying that the test set was collected at a later point in time than the training set. This was also given by the **TransactionDT** variable and the plot showing that the test set (in orange) was collected later
- otherwise the test set distribution did not seem too dissimilar from the train set

Feature Importance: Though the code no longer appears in the submission file, I viewed the feature importance output given by the Random Forest model in the first fit. The most important feature seemed to be one of the features starting with 'V' with an importance of around .45. Unfortunately, the most important features were those that started with 'V', 'C', and 'M' – these features that had little to no explanation about them.

Algorithms & Techniques

- **Random Forest:** This was the very first model I tried in order to establish a baseline model and score. There was no preprocessing done and I fit the model with default parameters. It achieved a score of .87.
- **XGBoost:** I knew little about XGBoost until I saw the best performing public kernels on Kaggle was using this algorithm. After doing some research, it seems like XGBoost has become a very popular algorithm on Kaggle and usually the winners of the competitions used XGBoost or some form of gradient boosting. Not only was I unfamiliar with XGBoost (aside from using it as a black-box algorithm for the sentiment analysis project), I was also unfamiliar with gradient boosting algorithms.

  I had to do some reading and video watching to get up to speed on how gradient boosting works on a simple level since I did not want to use the algorithm just as a black-box method.

  I tried many iterations with XGBoost, and it seemed like the more preprocessing I tried, the worse the model performed. I will go into this more in the **Implement** and **Results** section.

Benchmark: The public leaderboard on Kaggle establishes a reference for evaluating the performance of my model. The baseline **Random Forest** model I tried at the beginning was in the 85th percentile. The best XGBoost score I got was roughly around the 50th percentile. Ensembling led to a score in the 45th percentile.

3. Preprocessing & Implementation

Removing NaNs: I tried several strategies to deal with the missing values. Many other kernels simply replace NaNs with -999 and told the XGBoost model that missing values were replaced with -999. Prior to trying this method, I tried imputing the mean for numerical features and the most common category for categorical features. This seemed to negatively affect the performance of XGBoost. I learned that the XGBoost algorithm out of the box handles missing values well and does require imputation. I also tried removing features with more than 80% missing values, but that turned out to be unnecessary for XGBoost.

Normalization: normalizing numerical features using the formula (x-mu)/sigma, where mu is the mean of the feature and sigma is the standard deviation of the feature, seemed to hurt the performance of XGBoost. In fact, XGBoost scored around .93 without normalization and about .88 with normalization. Lesson learned. Normalization may only be useful for neural networks and similar convergence algorithms.

Label Encoding: I had initially considered labelling categorical features with strings using the 1-hot encoding scheme. Since there were many categorical features and some with thousands of unique categories, I decided to use label encoding instead to avoid the dimensionality explosion that would be cause by 1-hot encoding with this data set. In the kernel, I mentioned the drawback of label encoding that the model may interpret the categories to have some level of ordering since the labels are numerical (such as 1,2,3). The tradeoff seemed worthwhile to keep the dimensionality manageable.

Memory Reduction: Since the data set was very large, I found it helpful to utilize functions written by other Kagglers to compress the data frames. The function **reduce_mem_usage** reduced the RAM used by the train and test set by about 73% each. This was helpful since fitting the model ate up a large amount of RAM. Having the train and test data use less than .5 GB each before fitting was very helpful.

Dealing w/ Class Imbalance: After reading the XGBoost docs, I found a parameter called **scale_pos_weight** that is recommended to use when there is class imbalance. The docs recommended setting the parameter to sum(# negative examples) / sum(# positive examples), which would be about 30 in this dataset. I tried a few different numbers near 30 and it did not seem to improve performance. This leads me to believe XGBoost is fairly robust to class imbalance as compared to other algorithms.

Implementation

Hyperparameter Tuning with Cross Validation: In order to find a good set of hyperparameters for the XGBoost model, I tried using SKLearn's GridSearchCV and RandomizedSearchCV functions to find well performing combinations. Unfortunately, both of these algorithms used so much RAM that the kernel would quit and restart during the loop.

I decided to write my own grid search loop. Unfortunately, midway through the loop, the RAM limit would be exceeded, and the kernel would restart. I was able to see the first few scores, but since I did only train/test split, the AUC estimate may not have been as reliable as maybe a K-fold cross validation estimate would have been. When I chose parameters that did well (before the kernel quit), the performance seemed to be no better than the initial baseline XGBoost score of .938. It seemed that the AUC estimate from the validation split was unreliable or that the distribution of the test set differed enough from the train set to show the overfitting of the train set.

4. Results

- **Public Leaderboard Results:**
- Random Forest filled NaNs with -999: `.872`
- XGBoost filled NaNs with -999: `.938`, submission file: `baseline_xgboost.csv`
- XGBoost impute mean for numerical NaNs and most common cat for categorical NaNs, also normalized numerical vars: `.878`

- XGBoost impute mean for numerical NaNs and most common cat for categorical NaNs, no normalization: .932, submission file: `preprocessed_xgboost`
- XGBoost, impute mean for numerical NaNs, do not impute most common category for categorical NaNs, no normalization: .934, file: `preprocessed2_xgboost`. **NOTE**: imputing mean for numerical NaNs and most common category for categorical NaNs did not seem to help for XGBoost.
- Version 21: hyperparameter tuning with XGBoost (Grid Search or Random Search), .9284, `xgboost_with_tuning`
- .9226, `xgboost_with_tuning2`
- Ensemble: .9392

Ensembling (RF + XGBoost): In order to improve my score and get some practice with ensembling in machine learning, I decided to save the outputs of my previous models and try submitting a weighted average of those outputs, where the weights are chosen by me based on how well each model performed. For example, when I was ensembling with the random forest and the different xgboost models, the **baseline_xgboost** model performed the best with a score of .938, so the outputs of that model received the largest weight of .8. The random forest received the smallest weight of .05.

Picking the weights was just a tinkering process for me. The code was easy to run several times and submit so I settled on the set of weights that worked best out of the ones I tried. See the **Ensemble.jpg** file for the simple implementation. Combining the previous submissions with the weights (.05, .05, .1, .8) as shown in the picture led to the best score of **.9392**.

5. Conclusions

I found this project quite challenging, likely due to the fact that it was my first exposure to "real-world" data. Initially, there were almost 600,000 training examples with over 400 features. Many of the features had no description (confidential) and made it difficult to apply what little domain knowledge I had.

Another highlight of this project was my discovery of the XGBoost algorithm, which was significantly different than any algorithm I had learned previously. Some of the best scores in the competition were from kernels that did very random feature engineering, no preprocessing (other than replacing missing values with -999 and then setting the hyperparameter 'missing' to -999 when calling the XGBoost constructor), and seemed to pick random parameter settings. XGBoost seemed to work very well despite a large amount of missing values and significant class imbalance. In the future, I would likely start with XGBoost (or another gradient boosting algorithm) when I'm given a problem with structured, tabular data.

The data set was very large, which made it difficult to do hyperparameter tuning without exceeding the RAM limit that Kaggle provides.

Overfitting Disclaimer: Trying to continuously improve performance on the public test set helped me learn a lot about core machine learning fundamentals such as hyperparameter tuning, cross validation, and ensembling. That said, I may have been overfitting the public test set and it is hard to say how well my models would generalize to the private test set that is used at the end of the competition.

Kaggle Project Link – this is the link to the most recent version of the kernel. The most recent version only runs the ensemble code at the beginning, so it is not the best resource for reviewing the project. See the **fraud-detection.ipynb** for the main submission file.