Machine Learning Engineer Nanodegree

Capstone Project

Santhana Sankaramurthy
December 16, 2017

I. Definition

Project Overview

This project is intended to build a machine learning algorithm to effectively detect fraud in credit card transactions. This problem space has been extensively researched and implemented in industry over the years as is evident from the work of Chan and Stolfo – Towards Scalable Learning with Non-Uniform Class and Cost Distributions: A Case Study in Credit Card Fraud Detection available at http://www.aaai.org/Papers/KDD/1998/KDD98-026.pdf

Industry estimates indicate that 0.1% of credit card transactions globally are fraudulent [Source: Wikipedia]. Increasing fraud ultimately results in increasing costs for legitimate card holders, since companies need to recoup these costs in some way. Since credit providers bear the cost of fraud, credit providers that do better at fraud detection are able to charge lesser fees to card holders and be more competitive in the marketplace, both to merchants that use these card providers as well as individual card holders.

More broadly speaking, these algorithms can be used for other imbalanced data sets that occur widely in nature and business, some of which are even more imbalanced e.g. factory production error rates (about 0.1%), server failure rates, cancer detection (0.45%), detecting oil slicks from satellite ocean images, etc.

Problem Statement

We will use the credit card fraud dataset available on Kaggle. This is accessible at the link - https://www.kaggle.com/dalpozz/creditcardfraud.

This consists of real data from European cardholders in 2013. The data consists of 30 total features [i.e. 28 principal components identified through PCA, transaction amount, time [elapsed since first transaction] along with the Class for each transaction - 1 indicating fraud and 0 indicating normal transaction. There are 492 frauds out of 284,807 total transactions in the data provided, hence this is a highly imbalanced dataset with the positive class amounting to only 0.172% of all transactions.

In this project, we will create an algorithm to identify which transactions are fraudulent from the given transaction set. All the transactions in the dataset provided have been labeled as being in one of two potential classes i.e. fraudulent (Class 1) or legitimate (Class 0), we will use supervised learning and specifically, classification algorithms to determine which class each transaction belongs to.

As a first step, we will examine the data set and identify any characteristics that can potentially help us define the algorithm. At this stage, we will also graph the data to examine it visually and identify any patterns. None of the features are categorical, hence none need to be converted to numerical ones before we can feed them to the classifier. We will then review the data to see if we need to do any pre-processing e.g. filling in missing values, scaling, etc.

Examples of classification algorithms that can be used are Logistic Regression, Decision Trees, Naïve Bayes, Random Forest, Simple Vector Machines and Boosting (Gradient Boosting, Ada Boost, XGBoost).

We will train a Logistic Regression Classifier to serve as the baseline. As the data is highly imbalanced, it is important to use algorithms that will work effectively with such datasets – these are Decision Trees, Random Forest and Boosting methods. Once we do the initial evaluation of the algorithms based on relative performance (based on metrics that are defined in the next section), we can identify which one is suitable for further tuning. This can be further tuned using hyper parameter optimization to improve performance further. In addition, depending on algorithm performance, we will also examine if resampling techniques can help by making the data set more balanced.

Metrics

We will evaluate algorithm performance by using the following metrics.

A. **AUPRC (Area Under the Precision-Recall Curve)**
   Classifiers can be evaluated using the Area under the ROC curve or the Area under the Precision Recall curve. We will use the Precision Recall curve as opposed to the area under the ROC curve since the dataset is highly imbalanced and the area under the ROC curve will tend to show a high value even if the classifier is not performing particularly well). The average precision score provided by sklearn is used to compute the AUPRC.
B. **Recall**
   Since we are interested in ensuring that we identify as many of the fraudulent transactions as possible (preferably all), we will use the recall accuracy. This is the number of fraudulent transactions correctly identified divided by the actual number of fraudulent transactions in the test dataset. The higher the recall accuracy, the better the classifier is able to identify all fraudulent transactions.

C. Total Cost of Error

The ultimate objective for the credit card provider is to use this algorithm to minimize the loss due to fraudulent transactions. In this context, a fraudulent transaction for $ 2,000 obviously costs the credit provider way more than a fraudulent transaction that costs $ 20 and we should use an algorithm that catches the larger fraudulent transactions. Measuring performance can be done by minimizing the cost of the error in the algorithm calculated as the sum of the following two numbers

- For False Negative transactions, the loss is the transaction amount since the credit card provider incurs the loss
- For False Positive transactions, the customer will have to prove that the transaction is legitimate and may abandon the transaction or have a negative view of the merchant/ credit provider since the transaction is flagged as fraudulent. Hence the loss can be viewed as 50% of the transaction amount (the exact percentage used i.e. 50% is an arbitrary number)

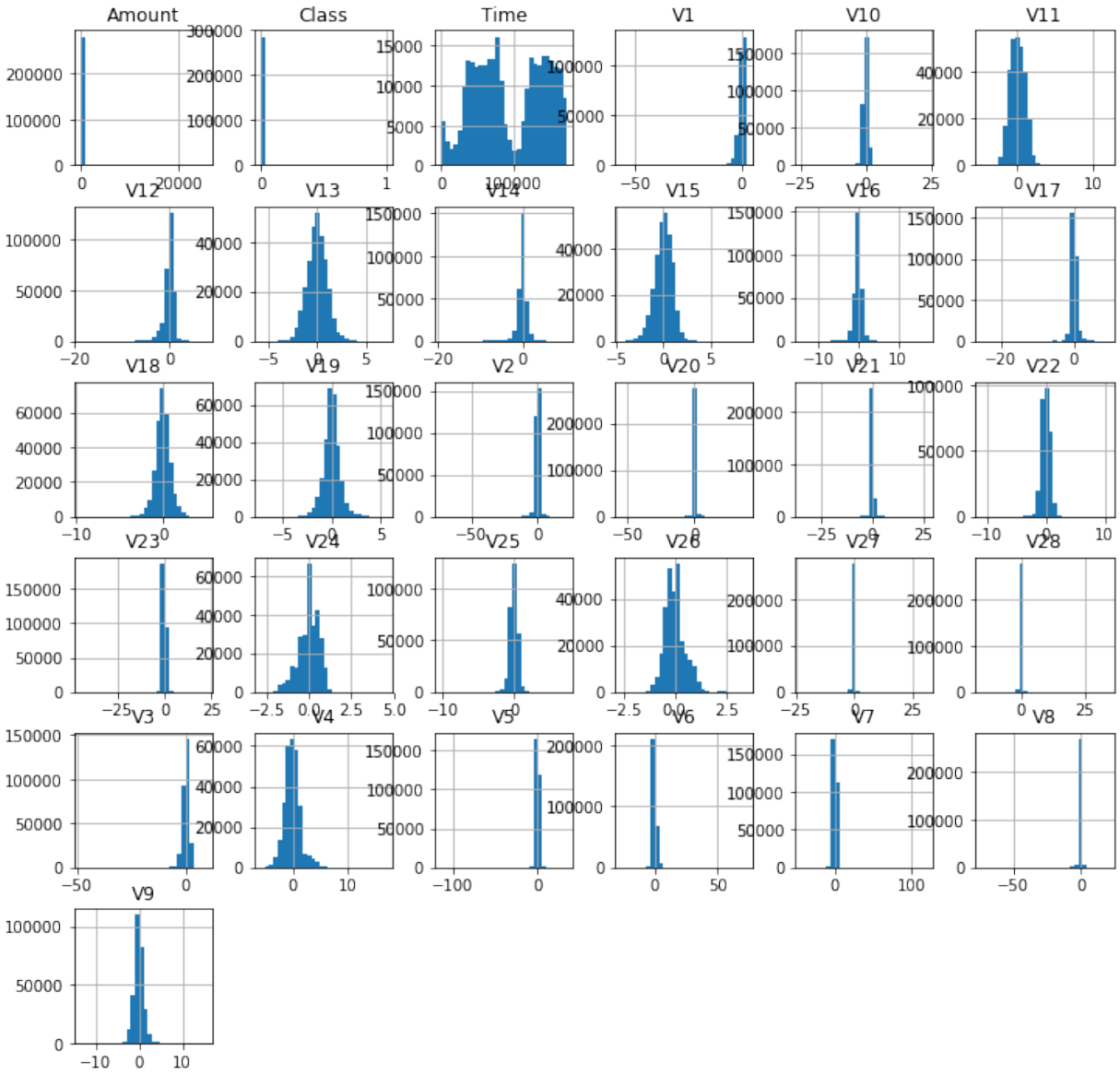
II. Analysis

Data Exploration

There are 284,807 data points with 31 variables each. Of these, 492 transactions have been identified as fraudulent. There are 30 features available in the dataset – 'Amount' indicating transaction amount, 'Time' indicating the time elapsed since the first transaction and V1, V2…V28 – 28 other numerical features that are output by a PCA on the raw data. There are no categorical features and no missing values.

In addition, the dataset also includes the 'Class' feature which has a value of 1 for fraud and 0 for normal transactions – this is the target variable that is to be predicted by the algorithm.

A quick look at the feature distribution provides the following results.

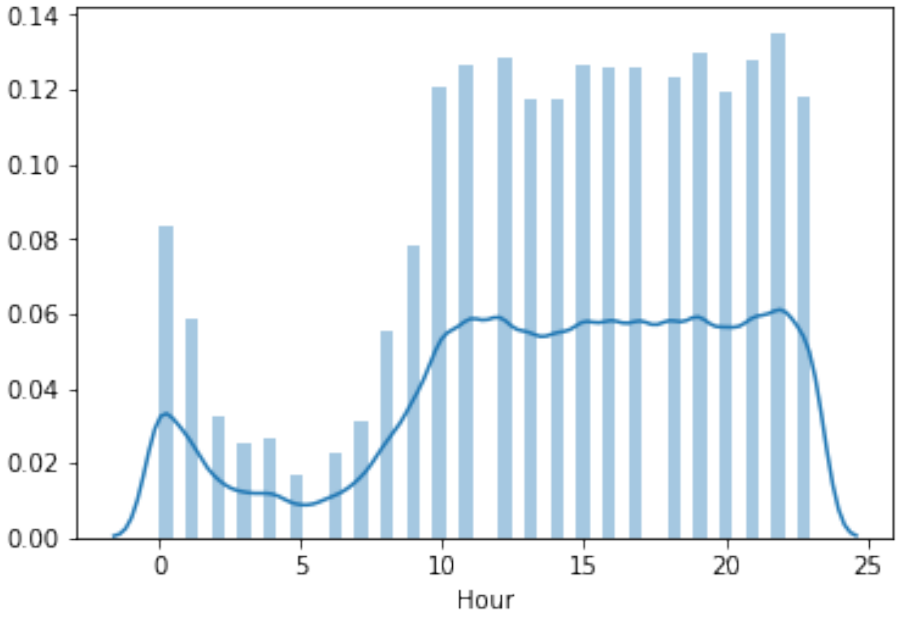


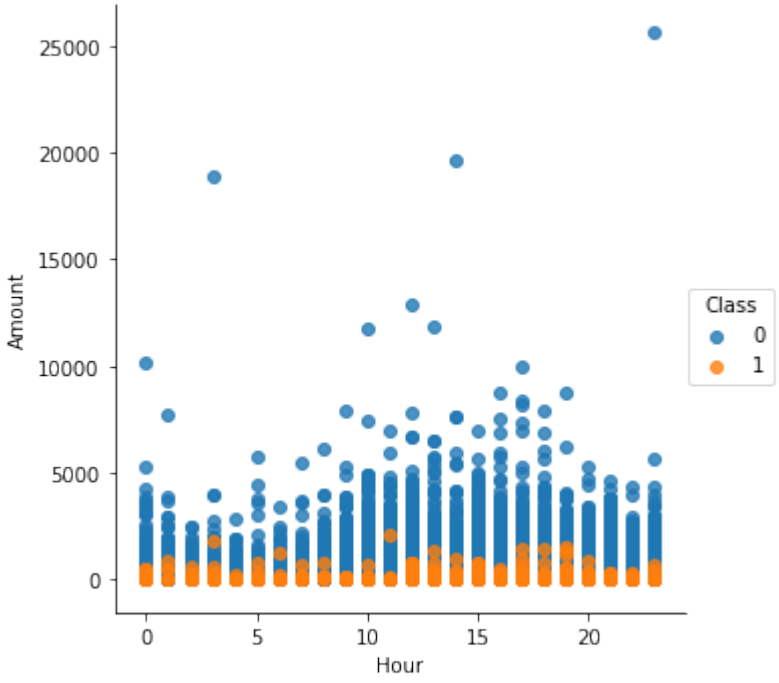
These indicate that virtually all the features are normally distributed, except for the Time variable which is the combination of two normal distributions, centered at two different times.

Exploratory Visualization

To further explore the data, we look at the fraud distribution across the two features that are provided names i.e. the Time and the Amount. The other features have names such as v1,v2… which do not provide any indication of what they refer to. The time refers to the elapsed time since the start of data collection and the 'Amount' refers to the value of the transaction. It will be helpful to examine the classification across these two parameters.

The distribution of fraud doesn't show any noticeable variation across the duration of the dataset, hence the Time feature can be dropped from further analysis.





In contrast, the amount shows that the fraud is concentrated at low levels of fraud. Specifically, he highest value of 'Amount' for fraudulent transactions is $ 2,125.87. If all transactions above this value can probably be safely dropped, it will improve the balance of the dataset and hence improve algorithm performance. However, this rule will not generalize well beyond this particular dataset, we will keep the entire dataset as is. In addition, the cost of an error here i.e. missing a fraudulent transaction worth around $ 2,000+ will be quite high as well.

Algorithms and Techniques

The dataset available is clearly labeled i.e. each transaction is identified as belonging to one of two classes - 'fraudulent' or 'legitimate'. Hence, this is a supervised learning problem and specifically binary classification.

We will evaluate the following algorithms for binary classification i.e. to predict if a particular transaction is legitimate [class 0] or fraudulent [class 1]. A brief description of the algorithm and notes on how they may apply to this project is provided below.

- **Logistic Regression**:
  In this algorithm, the prediction is arrived at as follows.
  - For a given feature vector X (in this case comprising of the training data with features Amount, V1, V2, …V28), the prediction is the value of the sigmoid function h(x) for a given θ. The goal of the algorithm is to find θ, the vector of feature weights. The prediction is defined as follows (this function is also called the logistic function).

    $h_\theta(X) = g(\theta^T X)$ where $g(\theta^T X) = 1/(1+e^{-\theta^T X})$

  - The loss function is a function that shows the prediction error (i.e. the difference between the algorithm prediction and the actual classification as fraudulent or legitimate).
  - We will use the default scikit-learn implementation that in turn uses the lib-linear method to determine θ to minimize the loss function. This implementation uses the coordinate descent method that works as follows – after initializing each feature weight randomly, it finds the value of the weight for feature 1 at which the loss function is minimized. Keeping this value fixed, it then finds weight for feature 2 at which the loss function is minimized. It continues the process until all the feature weights i.e. the complete vector θ is defined. This allows us to find the best prediction.

- **Decision Tree:**
  This is a recursive algorithm that predicts the likely class using feature values to split the dataset. This works as follows
  - It finds the entropy which is a measure for how cleanly the sample splits into Class 0/ 1. The entropy H(s) for a given sample s is defined as follows.
    $H(s) = -p_0 * \log_2 p_0 - p_1 * \log_2 p_1$
    where p_0 and p_1 are the probabilities of Class 0 and Class 1 in the sample s
  - For each feature, the algorithm splits the sample set into discrete intervals based on the value of the feature. It then looks at the information gain i.e. how much the overall entropy has decreased i.e. how much clearer the split between legitimate and fraudulent is based on this feature/attribute. It picks the feature

that has the highest information gain and uses that to split the dataset at this step into subsets, each of which defines a node.
    - At each of the nodes, it then repeats the step above with the remaining features until the tree has classified the sample accurately.

In general, the decision tree algorithm is likely to over fit the training data, which will perform poorly with general datasets. One way to minimize this is to set the min_samples_leaf parameter to 5 or 10 to prevent the sklearn algorithm from building a very large tree with very small sample sizes in the leaf i.e. less than 5 or 10.

- **Ensemble methods:** Random Forest, AdaBoost and XGBoost
  Boosting algorithms work by aggregating predictions across a number of 'weak' learners – i.e. any classification algorithm with predictions slightly better than random. Both AdaBoost and XGBoost tend to over fit much lesser than decision trees.

  Random Forests randomly select a subset of the features in the dataset to construct a set of classifiers (in this case decision trees) and create a prediction by finding the mode of the predictions of the individual classifiers, the one that is most predicted.

  AdaBoost uses a weak learner (in this case, a decision tree stump i.e. a single threshold decision tree) to predict the output for a randomly selected sample. The following steps are then followed

  - The algorithm assigns equal weights to all samples.
  - It then compares the predictions to the correct classification to identify misclassified samples i.e. false positives and false negative transactions.
  - In the next run, weights are assigned to emphasize misclassified samples for the next classifier. The weight is defined as below, where ε is the error rate for each classifier.

$$w_{n+1} = \frac{1}{2} * \frac{1}{1-\varepsilon} w_n \text{ for correctly classified points}$$

$$\text{and } w_{n+1} = \frac{1}{2} * \frac{1}{\varepsilon} w_n \text{ for misclassified points}$$

  - This increases the weight in iteration n+1 for misclassified points from the previous iteration n and reduces the weight for correctly classified points, while keeping the total of all sample weights as 1.
  - There is a 'voting power' α associated with each classifier used to find $H(X)$ i.e. the aggregate of results for all the classifiers $h_t(X)$. α is defined as follows for each classifier t

$$\alpha_t = \log_n \frac{1-\varepsilon}{\varepsilon}$$

$$H(X) = \sum_t \alpha_t \, h_t(X)$$

- Compare predictions from $H(X)$ with the defined level of accuracy. This continues until the classification algorithm (combination of learners) predicts with defined level of accuracy.

XGBoost is similar to AdaBoost in terms of assembling learners, except for a couple of variations.

- XGBoost adds a learner at every step to the ensemble instead of assembling a set of learners upfront, like AdaBoost
- At every step, it takes the errors in predictions of the previous set of learners and fits a new weak learner to them, thus emphasizing the samples that were wrongly predicted.
- The weights of the samples are left untouched for the next iteration, unlike AdaBoost. Instead, the new weak learner is added to the ensemble using gradient descent, thus minimizing the error of the overall ensemble. This process is continued until the desired level of accuracy is achieved.

When fitting all these algorithms to our training set, it is more important to correctly classify higher valued transactions than lower valued ones as the core objective is to minimize the weight. Hence, we will use the transaction amount as the sample weight to fit the algorithms.

In addition to these, re-sampling the dataset to improve the balance can improve algorithm performance. For example, under-sampling will create a more balanced dataset by removing legitimate transactions and keeping fraudulent transactions, so that the classifier will perform better.

This evaluation of the algorithm is done in two steps as follows

Step 1: Draw the ROC and Precision Recall curves

This provides a visual representation of the algorithm performance and indicates which ones are likely to perform better. The ROC curve plots the False Positive Rate (on the x-axis) vs. the True Positive Rate (on the y-axis) for various threshold values i.e. probabilities at which a given sample is classified as belonging to Class 0 or Class 1. The PR curve plots the Precision on the Y-axis vs. the Recall on the X-axis.

The precision-recall curve is likely to be more useful for this dataset since it is highly imbalanced. The reason is as follows - Precision is defined as True Positives divided by (True Positives + False Negatives), hence if the algorithm incorrectly classifies many transactions as legitimate i.e. as False Negatives, the denominator will increase significantly and reduce the Precision of the algorithm i.e. lower Y-value. In contrast, a high number of False Negatives (i.e. fraudulent transactions that the algorithm doesn't classify correctly) does not directly show up in the ROC curve as we are only plotting False Positives vs. True Positives.

Step 2: Generate Confusion Matrix, AUPRC and Recall

We then use the confusion matrix, AUPRC and precision/recall to narrow down and identify which of the algorithms is performing the best based on the metrics identified earlier.

Benchmark

A benchmark model using the same dataset published on Kaggle shows a recall accuracy of 93% i.e. the model correctly identifies 93% of all fraudulent transactions in the dataset. [https://www.kaggle.com/joparga3/in-depth-skewed-data-classif-93-recall-acc-now]. This will be used as the benchmark and our model should perform at or better than this to be considered successful.

III. Methodology

Data Preprocessing

We first check if there are any missing/unknown values in the dataset. A quick check of the dataset using data.keys() provides the output below.

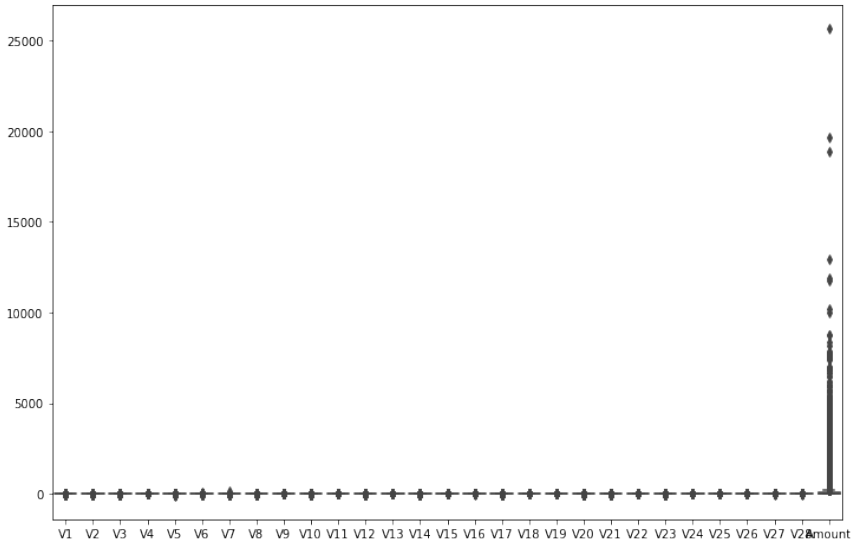
```
Data columns (total 31 columns):
Time      284807 non-null float64
V1        284807 non-null float64
V2        284807 non-null float64
V3        284807 non-null float64
V4        284807 non-null float64
V5        284807 non-null float64
V6        284807 non-null float64
V7        284807 non-null float64
V8        284807 non-null float64
V9        284807 non-null float64
V10       284807 non-null float64
V11       284807 non-null float64
V12       284807 non-null float64
V13       284807 non-null float64
V14       284807 non-null float64
V15       284807 non-null float64
V16       284807 non-null float64
V17       284807 non-null float64
V18       284807 non-null float64
V19       284807 non-null float64
V20       284807 non-null float64
V21       284807 non-null float64
V22       284807 non-null float64
V23       284807 non-null float64
V24       284807 non-null float64
V25       284807 non-null float64
V26       284807 non-null float64
V27       284807 non-null float64
V28       284807 non-null float64
Amount    284807 non-null float64
Class     284807 non-null int64
```

Reviewing the shape of the dataset and the percentage of fraud shows the following output that matches the description provided.

```
0.1727 % of the dataset are fraudulent transactions
Total Value of Transactions = 25162590.01
Total Value of Fraud Transactions = 60127.97
0.2390 % is the % by value of fraudulent transactions
```

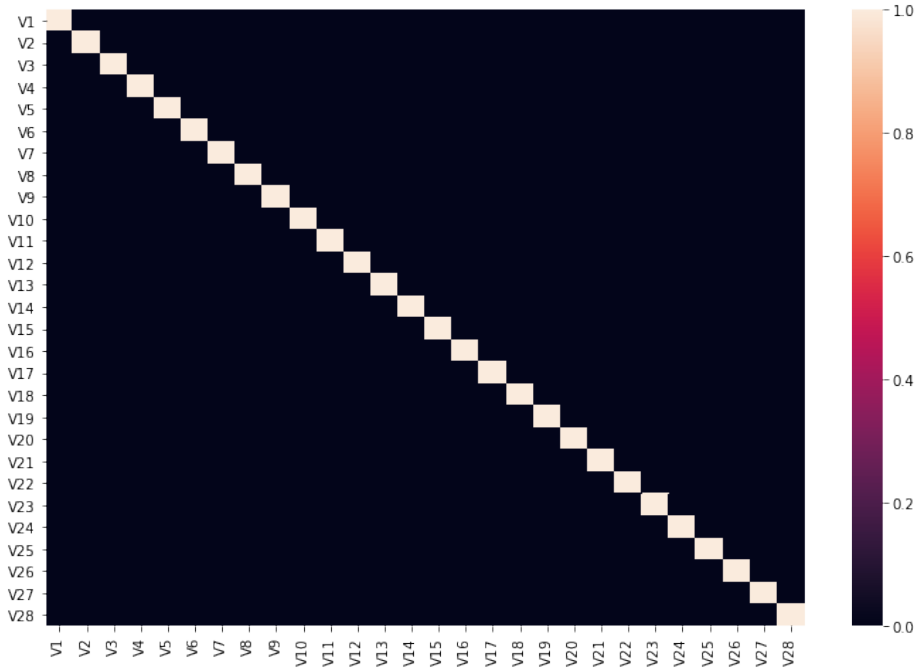
This indicates that all the data has been correctly imported and there that there are no missing values to be addressed during pre-processing. All the features are numerical and there are no categorical features that need to be transformed.

The following is a histogram of all the feature data provided except for Time. This shows that the 'Amount' is clearly on a completely different scale from the other features and must be scaled before being input to a classifier.



Hence, the 'Amount' feature is scaled using a StandardScaler.

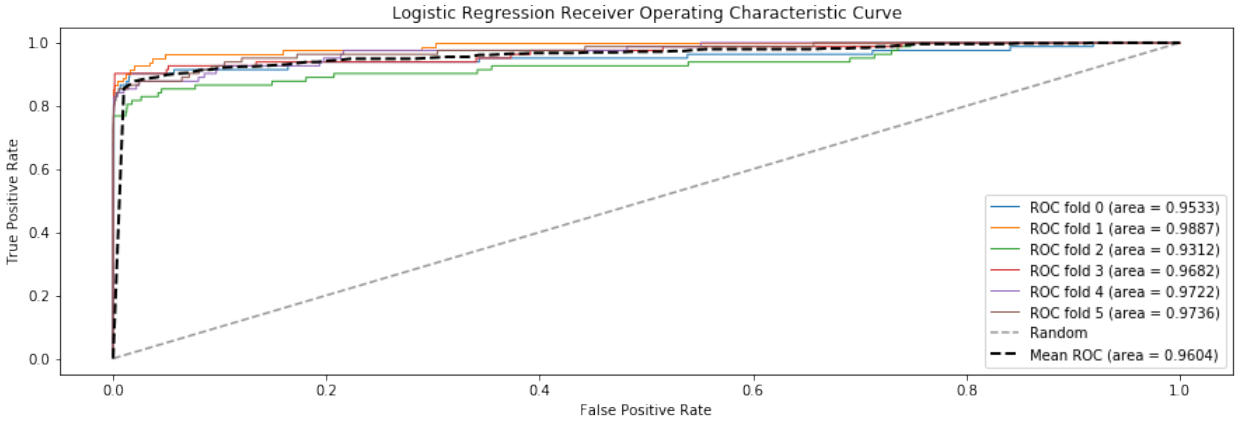
The correlation matrix of the features is shown below. Darker shades indicate low correlation and light red shade indicate higher correlation. It is clear that the features are independent and there is virtually no correlation between features, hence all features will be retained as inputs into the algorithm.

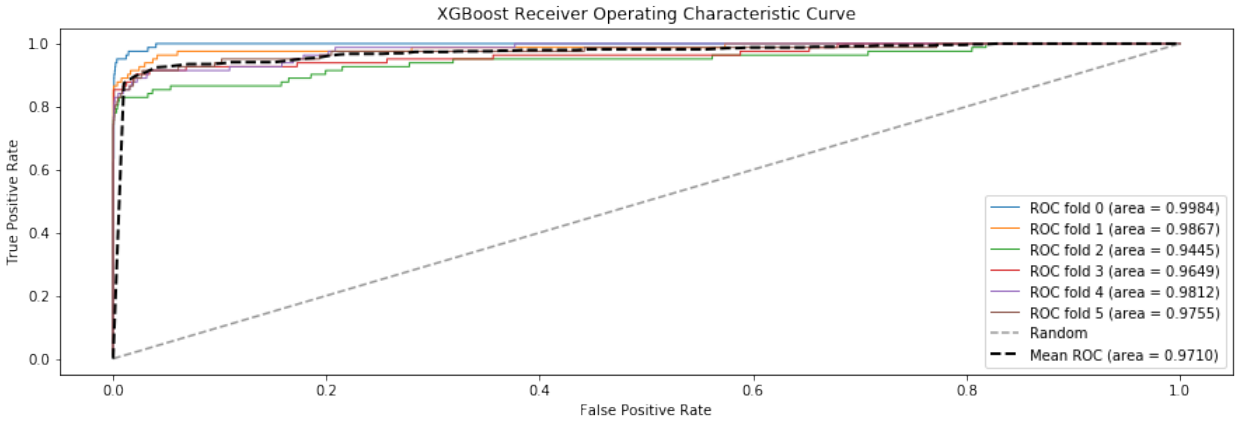


Implementation

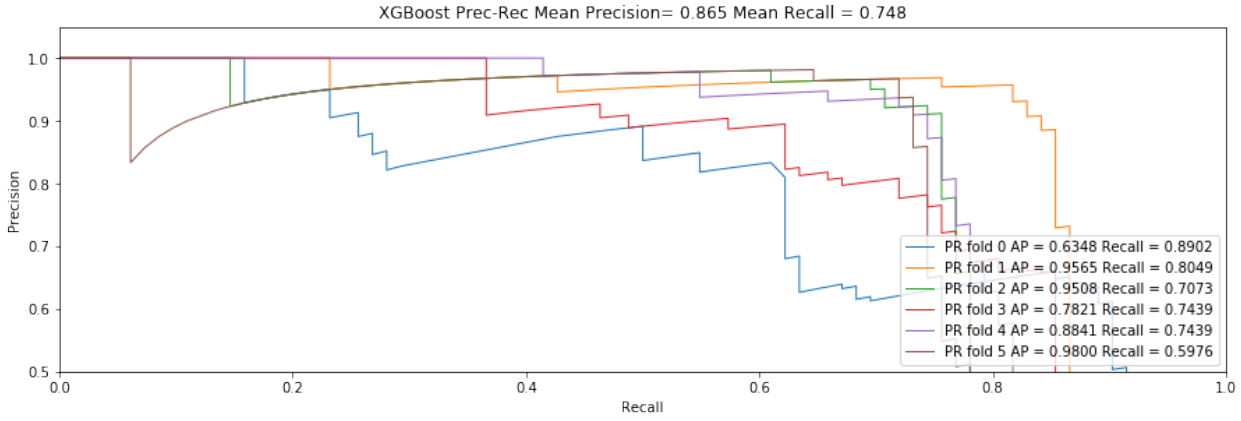
The first analysis is based on an implementation of the five base algorithms and comparing the performance using the helper functions that we have defined. We will implement six-fold cross validation using the scikit-learn StratifiedKfold method. This preserves the imbalance i.e. the class representation in the folds and hence provides results that are more representative of the actual set.

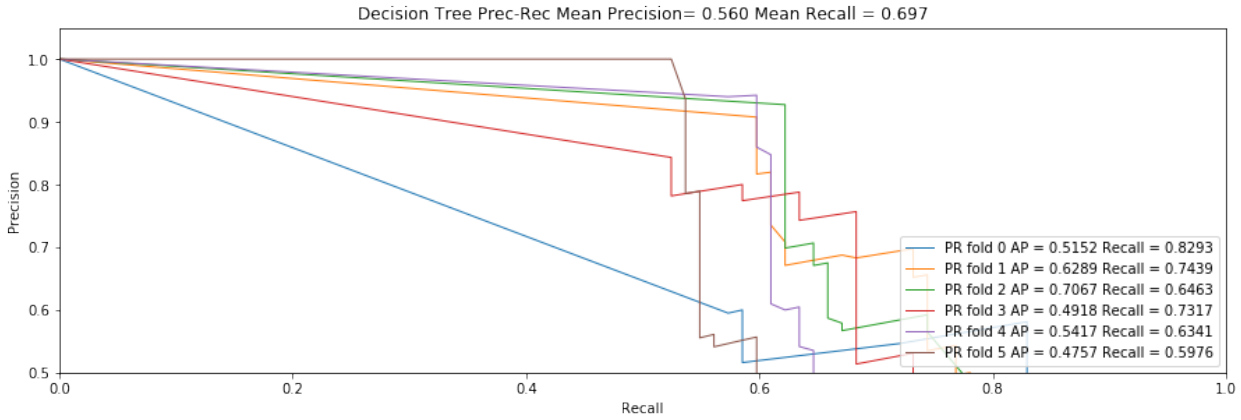
The ROC curves for all the classifiers are heavily weighted towards the top left quadrant of the graph. Two of the curves (for Logistic Regression) and for XGBoost are shown below [all the plotted curves are available in the notebook]. As expected, the curves do not show much of a difference with the Area under the ROC curve (averaged over the 6 folds) being 0.9604 for Logistic Regression and 0.9710 (for XG Boost).

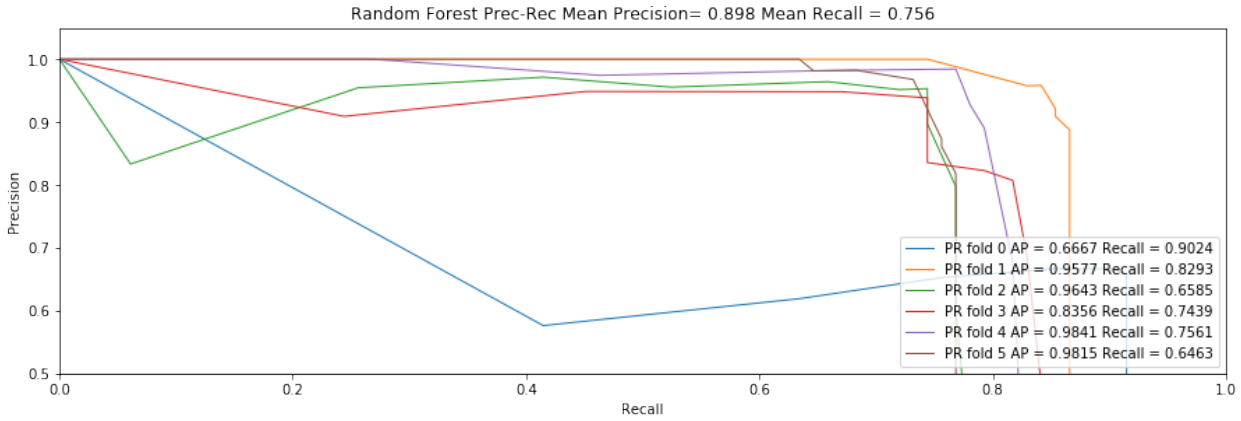


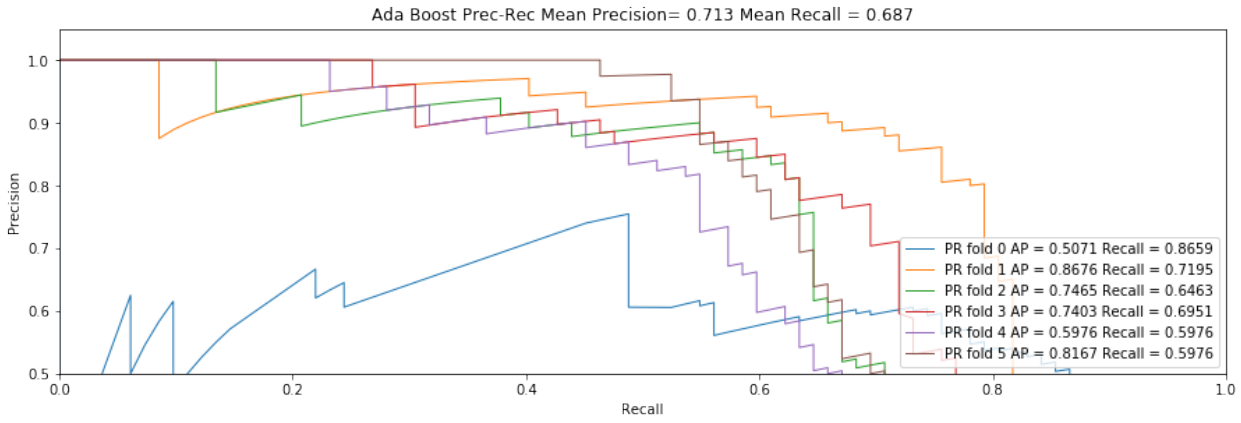


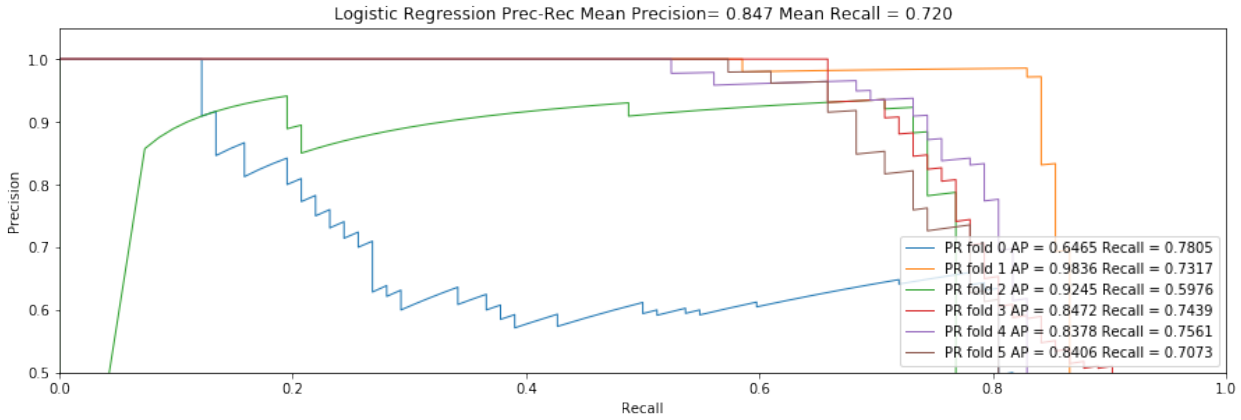
We now turn to the PR curves for the five algorithms which are provided below.


XGBoost Prec-Rec Mean Precision= 0.865 Mean Recall = 0.748
Precision
Recall
PR fold 0 AP = 0.6348 Recall = 0.8902
PR fold 1 AP = 0.9565 Recall = 0.8049
PR fold 2 AP = 0.9508 Recall = 0.7073
PR fold 3 AP = 0.7821 Recall = 0.7439
PR fold 4 AP = 0.8841 Recall = 0.7439
PR fold 5 AP = 0.9800 Recall = 0.5976


Decision Tree Prec-Rec Mean Precision= 0.560 Mean Recall = 0.697
Precision
Recall
PR fold 0 AP = 0.5152 Recall = 0.8293
PR fold 1 AP = 0.6289 Recall = 0.7439
PR fold 2 AP = 0.7067 Recall = 0.6463
PR fold 3 AP = 0.4918 Recall = 0.7317
PR fold 4 AP = 0.5417 Recall = 0.6341
PR fold 5 AP = 0.4757 Recall = 0.5976


Random Forest Prec-Rec Mean Precision= 0.898 Mean Recall = 0.756
Precision
Recall
PR fold 0 AP = 0.6667 Recall = 0.9024
PR fold 1 AP = 0.9577 Recall = 0.8293
PR fold 2 AP = 0.9643 Recall = 0.6585
PR fold 3 AP = 0.8356 Recall = 0.7439
PR fold 4 AP = 0.9841 Recall = 0.7561
PR fold 5 AP = 0.9815 Recall = 0.6463





An examination of the above shows the following

- Decision Tree, AdaBoost and Logistic Regression show generally lower PR curves (and hence lesser AUPRC) compared to XGBoost and Random Forest.
- Mean Precision and Recall are highest for the boosting methods
- All the algorithms (except perhaps for XG Boost) are showing that for the first fold, the PR line indicates low precision and low recall. This means the algorithm is performing poorly on a particular fold of the dataset.

These indicate that XG Boost and Random Forest are likely to perform better for our purposes. The execution time of these algorithms to perform the cross-validation and draw the PR and ROC curves is provided below. This indicates that XGBoost takes significantly longer to execute compared to the other algorithms.

Algorithm	Execution Time (in seconds)
Logistic Regression	107.62
Decision Tree	298.45
Ada Boost	864.78
XG Boost	1936.29
Random Forest	271.44

We next calculate confusion matrices, precision and recall values and the total error cost to make a final determination of the algorithm to be used for further analysis and tuning.

Compared to prior implementations, there is additional code here is to calculate the total error cost which is computed as per metric definitions indicated earlier i.e. cost of all false negative transactions + 50% of cost of false positives.

Average Precision, Recall, F-Score and Error Cost are shown below for the five algorithms.

Algorithm	Average Precision	Recall	Cost of False Negatives ($)	Cost of False Positives ($)	Total Error Cost ($)
Logistic Regression	0.7621	0.9991	8,420.42	481.59	8,902.01
Decision Tree	0.5600	0.8071	4,658.37	995.00	5,653.36
Ada Boost	0.7840	0.9993	3,256.93	1,204.05	4,460.99
XG Boost	0.8754	0.9996	3,738.36	634.63	4,372.99
Random Forest	0.8373	0.9995	6,536.21	123.41	6,659.62

Based on the above data, we can draw the following conclusions

- Logistic Regression and Decision Trees are performing well from an execution time perspective, however average precision, recall accuracy and error costs indicate they are not a good fit for this dataset.
- Random Forest is performing the best from the average precision and recall perspectives. It is also relatively very fast. However, the cost of the error is high, indicating that while it is missing fewer transactions, the ones it is missing are relatively more expensive.
- AdaBoost is doing well with the lowest total error cost, however it has lower average precision and slightly lower recall compared to XGBoost.
- The execution time of XGBoost is significantly higher compared to AdaBoost i.e. 2.2 times execution time.
- XGBoost has better average precision and better recall compared to AdaBoost, though it has higher cost and longer execution time. Hence, there is a tradeoff between using a more complex algorithm to achieve better average precision and accuracy. Hence, we will use XGBoost for further analysis and tuning to improve performance.
- Lastly, the recall is above 0.999 i.e. above 99.9% for 4 of the five algorithms considered, which is better than the benchmark. Hence, the objective of further refinement will be to increase the average precision (which represents the area under the precision-recall curve) and reduce the total error cost to be as close to zero as possible.

Implementation Notes

The first area of interest is in picking the algorithm to tune after the initial analysis, specifically choosing between AdaBoost and XGBoost.

Though XGBoost yields better results in terms of the AUPRC, it takes more than 2 times longer to execute for what appears to be a marginal increase in total error cost (less than $ 100).

Even this difference in error cost is based on our assumption of the cost for false positives that we have arbitrarily attributed to be 50% of the transaction value. If this were say, 10% instead, then the cost of false positives would be as shown in the table below (last two columns added below) and AdaBoost would actually have a lower total error cost.

Algorithm	Average Precision	Recall	Cost of False Negatives ($)	Cost of False Positives ($) at 50% of TV	Total Error Cost ($) at 50% TV for FP	Cost of False Positives ($) at 10% TV	Total Error Cost ($) at 10% TV for FP
Ada Boost	0.7840	0.9993	3,257	1,204	4,461	$ 241	$ 3,498
XG Boost	0.8754	0.9996	3,738	635	4,373	$ 127	$ 3,855

Note: The $ values have been rounded to nearest dollar from the previous comparison.

The implementation of the project was fairly smooth except for determining the AUPRC, which presented some difficulty during code development and debugging.

Determining the AUPRC

The key metric AUPRC is determined using a metric called (somewhat confusingly, in my view) average_precision in scikit-learn. The initial implementation I tried, unsuccessfully, was to compute the value manually by determining precision at various values along the recall axis until I read the scikit-learn documentation closely.

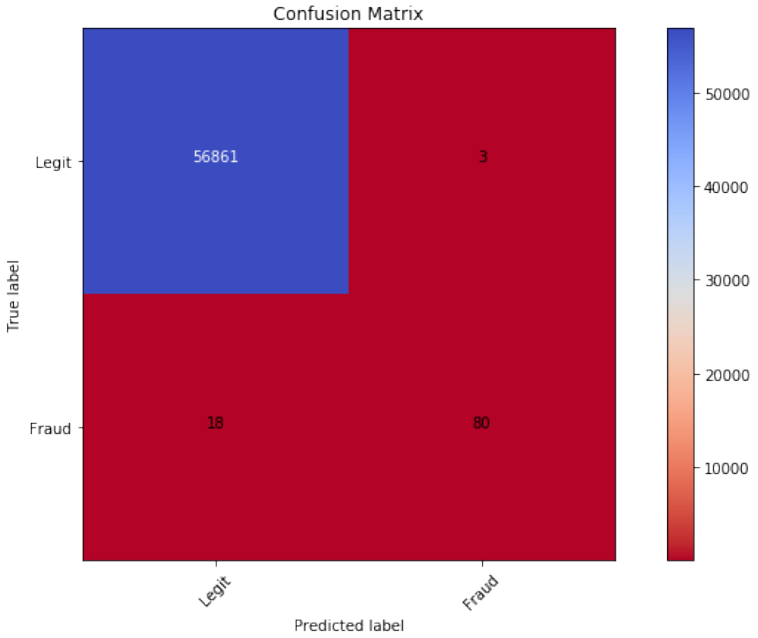
Refinement

Our objectives in tuning the algorithm are to increase the average precision and recall, while decreasing the total cost of error. The initial (untuned XGBoost) performance metrics are as follows.

```
Base XGBoost
Average Precision = 0.8754
              precision    recall  f1-score   support
      Legit     0.9997    0.9999    0.9998     56864
      Fraud     0.9639    0.8163    0.8840        98

avg / total     0.9996    0.9996    0.9996     56962

Total_fn_loss = $ 3738.36
Total_fp_loss = $ 634.63
Total cost of error = $ 4372.99
```



We train the algorithm with an initial set of parameters. The initial parameters used are as follows. The objective is binary: logistic since this is a binary classification problem.

```
ind_params = {'learning_rate': 0.1, 'n_estimators': 300, 'random_state':0,
'subsample': 0.8, 'colsample_bytree': 0.8, 'objective': 'binary:logistic'}
```

The scoring mechanism used for tuning XGBoost is the roc_auc which may not be ideal, for the reasons discussed earlier. A custom scorer that incorporates the error cost will probably yield b etter results.

We will use the RandomizedSearch from sklearn instead of GridSearch since it randomly selects the combinations to search (instead of an exhaustive search of all combinations) and may be much faster. [Reference: http://blog.kaggle.com /2015/07/16/scikit-learn-video-8-efficiently-searching-for-optimal-tuning-parameters/]

```
cv_params = {'max_depth': [3,5,7], 'min_child_weight': [1,3,5]}
```

This search provides the results shown below.

```
params': [{'max_depth': 3, 'min_child_weight': 1},
  {'max_depth': 5, 'min_child_weight': 1},
  {'max_depth': 5, 'min_child_weight': 5},
  {'max_depth': 5, 'min_child_weight': 3},
  {'max_depth': 7, 'min_child_weight': 3},
  {'max_depth': 3, 'min_child_weight': 3},
  {'max_depth': 7, 'min_child_weight': 1},
  {'max_depth': 3, 'min_child_weight': 5}],
 'rank_test_score': array([4, 7, 6, 1, 8, 2, 5, 3], dtype=int32)
```

Hence, the following combination yields best results with a roc_auc score of 0.99999834 - `max_depth': 5, 'min_child_weight': 3`

We then turn to tuning the learning rate and subsample. The possible values fed into the RandomizedSearch are as follows.

cv_params = {'learning_rate': [0.2, 0.1, 0.01], 'subsample': [0.7,0.8,0.9]}

We then get the following results indicating that a learning rate of 0.1 and a subsample of 0.8 performs the best.

```
params': [{'learning_rate': 0.2, 'subsample': 0.8},
  {'learning_rate': 0.1, 'subsample': 0.7},
  {'learning_rate': 0.2, 'subsample': 0.8},
  {'learning_rate': 0.1, 'subsample': 0.8},
  {'learning_rate': 0.01, 'subsample': 0.8},
  {'learning_rate': 0.1, 'subsample': 0.9},
  {'learning_rate': 0.01, 'subsample': 0.7},
  {'learning_rate': 0.01, 'subsample': 0.9}],
 'rank_test_score': array([5, 2, 4, 1, 7, 3, 6, 8], dtype=int32),
```

To summarize, we have found the following parameters through tuning

- Max Depth – 5
- Min Child Weight – 3
- Learning Rate (eta) – 0.1
- Subsample – 0.8

We then use the cv function provided within the xgboost boost API itself. This uses early stopping to determine the best parameters without overfitting. The API requires us to create a DMatrix from the test data, which helps make the algorithm more efficient with a larger training set.

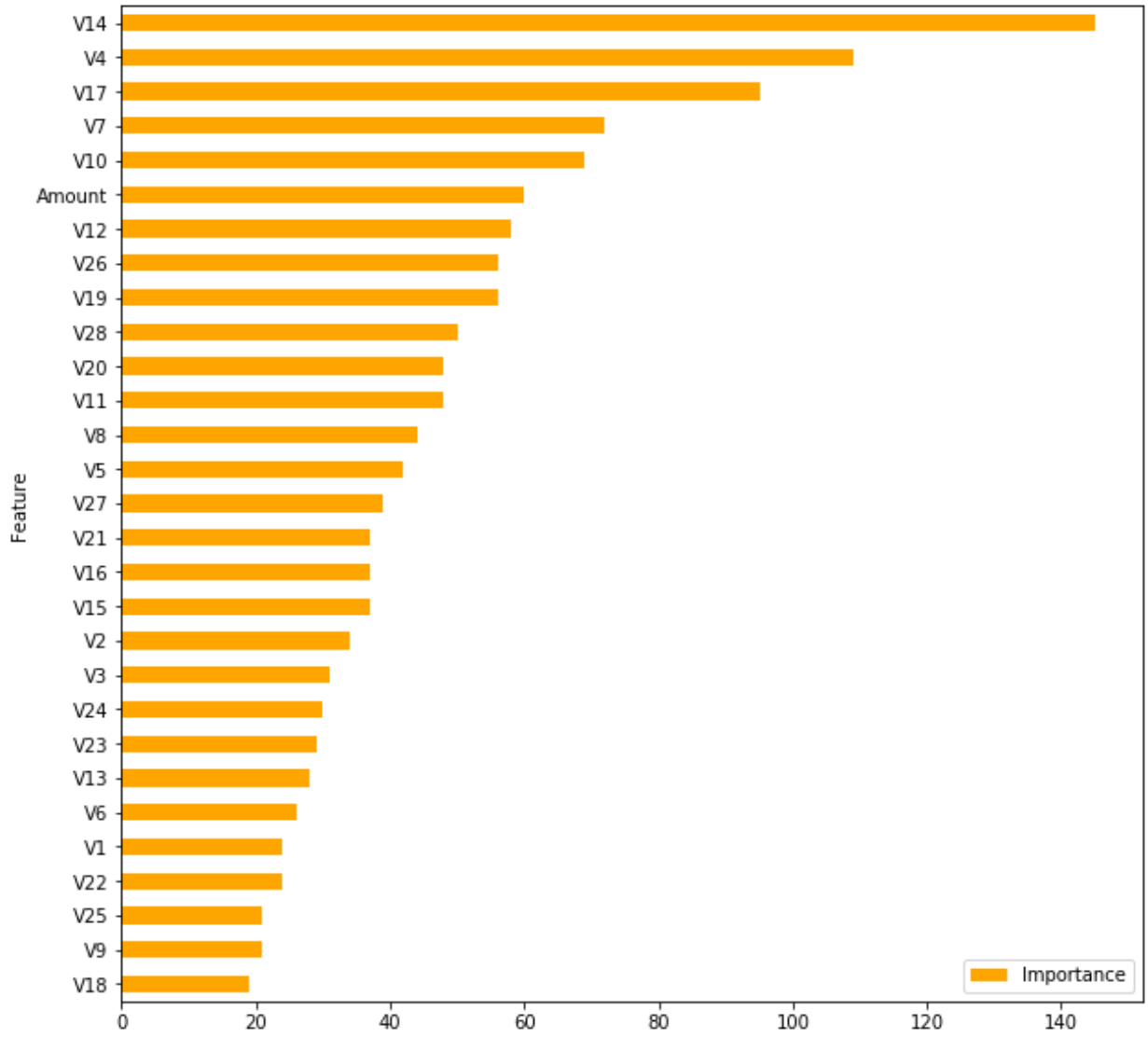
When we look at the results of the cv search below, we see that iteration 123 provides best results.

	test-auc-mean	test-auc-std	train-auc-mean	train-auc-std
114	0.979502	0.009407	0.999553	0.000114
115	0.979580	0.009653	0.999576	0.000119
116	0.979779	0.009119	0.999609	0.000116
117	0.979906	0.008913	0.999624	0.000120
118	0.979634	0.009318	0.999652	0.000101
119	0.979459	0.009361	0.999657	0.000095
120	0.979371	0.009414	0.999666	0.000092
121	0.979821	0.009319	0.999676	0.000087
122	0.979926	0.008866	0.999681	0.000082
123	0.979973	0.008758	0.999692	0.000081

We use this iteration to evaluate the average precision, recall and total cost of error for the tun ed classifier. The results we obtain are as follows.

Algorithm	Average Precision	Recall	Cost of False Negatives ($)	Cost of False Positives ($)	Total Error Cost ($)
Base XG Boost	0.8754	0.9996	3,738.36	634.63	4,372.99
Tuned XG Boost	0.8863	0.9996	4,023.69	60.99	4,084.68

The associated feature importance graph is below, which indicates that V14,V4 and V17 are the features that have the most impact on the model. Since these are PCA features and we don't ha ve access to the underlying 'real-life' features, we are unable to draw further conclusions.



IV. Results

Model Evaluation and Validation

Based on these results, we determine that the tuned XGBoost model performs extremely well against this and achieves a very low total error cost, very high precision and recall accuracy as shown above.

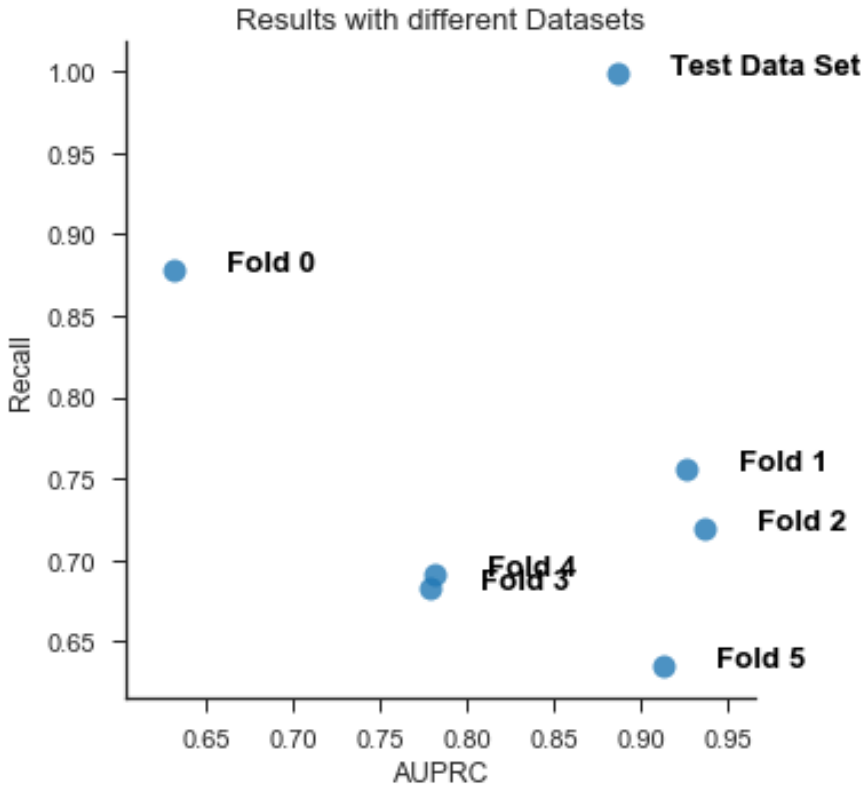
To view the performance in a different light, the total value of transactions in the dataset = $ 25,162,590 and the total value of fraudulent transactions = $ 60,128 i.e. 2.3% by value. **Total error cost of the tuned algorithm = $ 4,084 i.e. 6.7% of the fraudulent transactions and 1.6% of the total transaction value processed.**

Model Robustness

To evaluate how stable the model is, we use the final tuned model to compute AUPRC, Recall and draw performance curves using a Stratified KFold on the dataset for cross-validation. The results in a table and a graphical format are as follows.

Algorithm	Average Precision	Recall
Tuned XG Boost Earlier Test Set	0.8863	0.9996
Tuned XG Boost Fold 0	0.6316	0.8780
Tuned XG Boost Fold 1	0.9254	0.7561
Tuned XG Boost Fold 2	0.9365	0.7195
Tuned XG Boost Fold 3	0.7778	0.6829
Tuned XG Boost Fold 4	0.7808	0.6909
Tuned XG Boost Fold 5	0.9123	0.6341

The results are shown visually below – these show that while the AUPRC has performed well, the recall values are significantly lower for the different folds and the model may have overfitted the dataset and hence is not as robust as we had hoped



V. Conclusion

Reflection I: Hyper-parameter Tuning

Hyper-parameter tuning for XGBoost turned out to be arduous in terms of execution time since each run with GridSearch took a fairly long time. I found that Randomized Search executes faster – while it may still not find the absolute best parameters, it is sufficient for our purposes and executes in a time of 1 hour 15 minutes or so.

Reflection II: Data Quality

Secondly, the dataset available for the project is clean and well-labeled, requiring minimal pre-processing before input. It is unlikely that such well-formed data will be available in the real-world, hence I anticipate that production scenarios will require significantly more work on the dataset before training and evaluating classifiers.

Improvement

There are three areas of improvement identified, one is to use sampling methods, second to consider alternative models/ classifiers that are more robust than the model developed and third, tune XGBoost to execute faster.

Sampling Methods:

One of the key observations throughout the project is that many issues with classifier performance stem from having a huge class imbalance in the dataset. In this context, researchers have developed resampling techniques that can help reduce the imbalance of the dataset and hence improve performance. The family of techniques considered here are the imbalanced-learn methods [API and documentation is available at http://contrib.scikit-learn.org/imbalanced-learn/stable/api.html] Usage of these methods was described at PyData 2016 [accessible at https://www.youtube.com/watch?v=-Z1PaqYKC1w].

We evaluate the performance of the base XGBoost classifier against these sampling methods to determine if there is an improvement in performance. This shows the following results.

```
OverSampler
-------------
Precision Score = 0.9959
Recall Score = 1.0000
F Beta Score = 0.9967
Total_fn_loss = $ 0.00
Total_fp_loss = $ 15327.61
Total cost of error = $ 15327.61

UnderSampler
-------------
Precision Score = 0.9636
Recall Score = 0.8833
F Beta Score = 0.9464
Total_fn_loss = $ 784.90
Total_fp_loss = $ 521.72
Total cost of error = $ 1306.61
```

```
NearMiss
-------------
Precision Score = 1.0000
Recall Score = 0.9917
F Beta Score = 0.9983
Total_fn_loss = $ 0.76
Total_fp_loss = $ 0.00
Total cost of error = $ 0.76

SMOTE ENN
-------------
Precision Score = 0.9926
Recall Score = 0.9848
F Beta Score = 0.9911
Total_fn_loss = $ 84964.72
Total_fp_loss = $ 27396.14
Total cost of error = $ 112360.86
```

These results demonstrate that the NearMiss sampling method vastly improves algoritm performance with a high recall, 100% precision and low error cost. This can be an area for further exploration to improve performance even further. Given that we have already exceeded the benchmark results, this is not explored further in this report.

Execution Time:

During the analysis, it is observed that XGBoost performs well and generates good results. However, it does take significantly longer to execute compared to other algorithms. This can be a serious limitation in the context of real-world datasets with larger sample sizes. Hence reducing execution time should be considered while identifying the algorithm. During our analysis, we found that AdaBoost can be tuned and generate good results in less than half the time for XGBoost. A brief search of the internet also shows that there are newer algorithms such as LightGBM and CatBoost, further variations on XGBoost that have emerged. It will be worthwhile to examine these to see if these execute faster and deliver as good or better results.

Appendix: List of references (not mentioned in the text)

- https://jessesw.com/XG-Boost/
- https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/
- https://www.kaggle.com/lct14558/imbalanced-data-why-you-should-not-use-roc-curve