BNP Paribas Cardif Claims Management: An Exercise in Prediction Using Anonymous Variables

Capstone Project *for* Springboard Data Science Intensive

Ritwik Raj

August 24, 2016

Table of Contents

Introduction 3

Provided Data 5

Evaluation Metric 5

Exploratory Data Analysis 6

- Evaluation of Missing Values 6
- Categorical and Numerical Features 11
- Unique Values 12
- Missing Data Revisited: To impute or not to impute? 14
- Choice of Classifier: XGBoost 15
- Examining Possible Feature Correlation 16
- Distribution of Numerical Variables 19

Prediction of Classes Using XGBoost 21

- Parameter Tuning 21
- RandomizedSearchCV and GridSearchCV 22
- XGB.CV and the ModelFit function 22
- Combination of Categorical Labels across Training and Test Data 24
- Successive Reduction of Feature Space 24

Results 26

- Other Unsuccessful Approaches 28
- Challenges 29
- Conclusions 30

Introduction

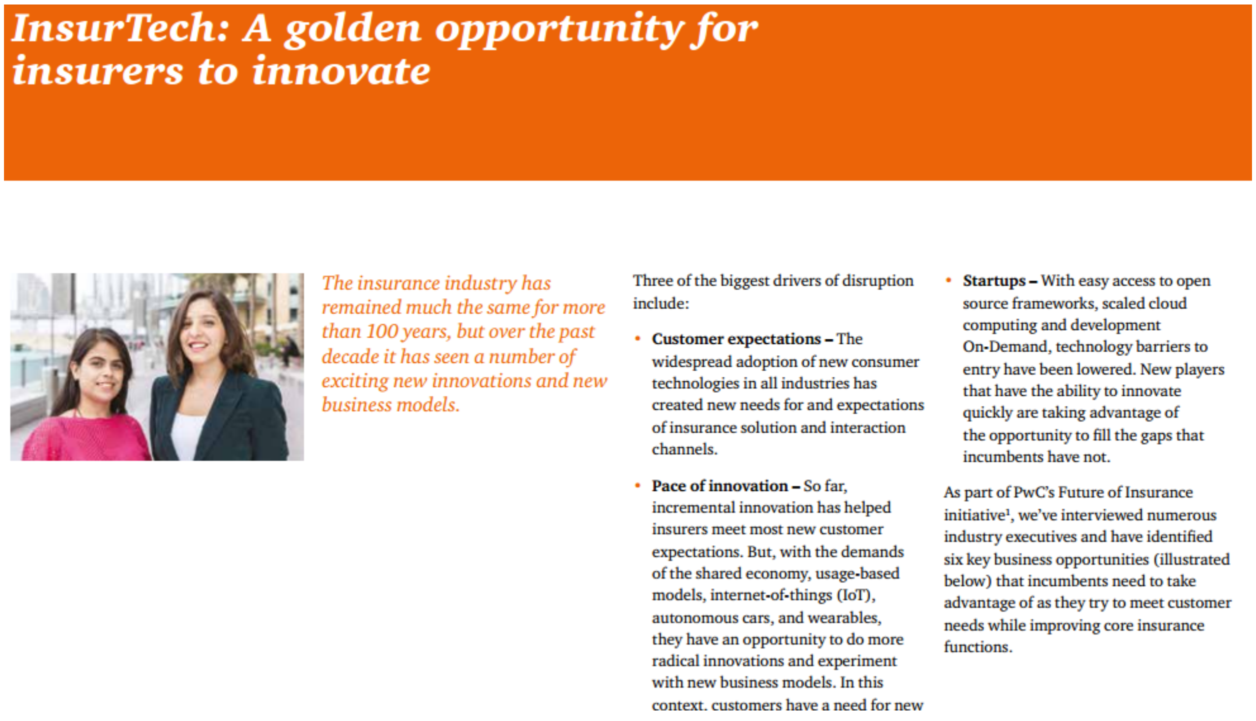
For my Springboard Data Science Intensive Capstone project, I selected a recently concluded Kaggle competition, sponsored by BNP Paribas Cardif, which is the insurance arm of BNP Paribas. This competition, titled BNP Paribas Cardif Claims Management, challenged Kaggle members to accelerate their claims management process by predicting if a claim:

a. Could be accelerated for approval leading to faster payments, or
b. Requires additional information before approval is made

In their own words, this is an important problem to solve because:

"In a world shaped by the emergence of new uses and lifestyles, everything is going faster and faster. When facing unexpected events, customers expect their insurer to support them as soon as possible."

While BNP Paribas Cardif may be in the minority when it comes to expressing their desire to improve and innovate via a Kaggle contest, the belief that this is a pressing concern for the insurance industry is widely held. In fact, Pricewaterhouse Coopers, the largest professional services firm in the world, has issued a loud and clear call to the industry in its 2016 Annual Report :



InsurTech: A golden opportunity for insurers to innovate

The insurance industry has remained much the same for more than 100 years, but over the past decade it has seen a number of exciting new innovations and new business models.

Three of the biggest drivers of disruption include:

- **Customer expectations –** The widespread adoption of new consumer technologies in all industries has created new needs for and expectations of insurance solution and interaction channels.
- **Pace of innovation –** So far, incremental innovation has helped insurers meet most new customer expectations. But, with the demands of the shared economy, usage-based models, internet-of-things (IoT), autonomous cars, and wearables, they have an opportunity to do more radical innovations and experiment with new business models. In this context. customers have a need for new
- **Startups –** With easy access to open source frameworks, scaled cloud computing and development On-Demand, technology barriers to entry have been lowered. New players that have the ability to innovate quickly are taking advantage of the opportunity to fill the gaps that incumbents have not.

As part of PwC's Future of Insurance initiative[1], we've interviewed numerous industry executives and have identified six key business opportunities (illustrated below) that incumbents need to take advantage of as they try to meet customer needs while improving core insurance functions.

PwC calls out Customer Expectations and Pace of Innovation as the biggest drivers of disruption for the insurance industry. Therefore, it makes a lot of sense that a company like BNP Paribas Cardif, which has a very small market share in the US market, would try to distinguish itself by its exceptional customer service.

As can be imagined, one component of good customer service is good training, ethical practices, effective digital presence etc. However, the speed of response is often held back by the current limitations of the technology being used.

Therefore, if BNP Paribas Cardif can use machine learning to gain insights very early in the claims process, they will have made a giant stride towards their goal – not to mention, they would have a significant competitive advantage as well.

Provided Data

The competition organizers provided two datasets:

1. Training data with a binary target (0/1 where 1 indicates that the claim is suitable for accelerated approval)
2. Test data of the same form, obviously without target values

Both datasets have ~114,000 rows, where each row corresponds to a claim, and both have 131 features/predictors. The features are both numerical and categorical, **and they contain the information available to BNP Paribas Cardif when the claims were received**. Also, there are no ordinal variables, meaning that none of the categorical variables have an order embedded in the values taken by that variable.

A “feature” of the datasets is that all of the 131 features are anonymized, i.e. we have no information about what they represent. The features are numbered v1 through v131. At the outset, we have not been told how many variables are numerical and how many are categorical.

Finally, the data has a large number of missing values.

Evaluation Metric

Submissions to the competition were judged based on the Log Loss of the model on the test dataset:

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log(p_i) + (1-y_i)\log(1-p_i))$$

where N is the number of observations, *log* is the natural logarithm, y_i is the binary target and p_i is the predicted probability that y_i is equal to 1.

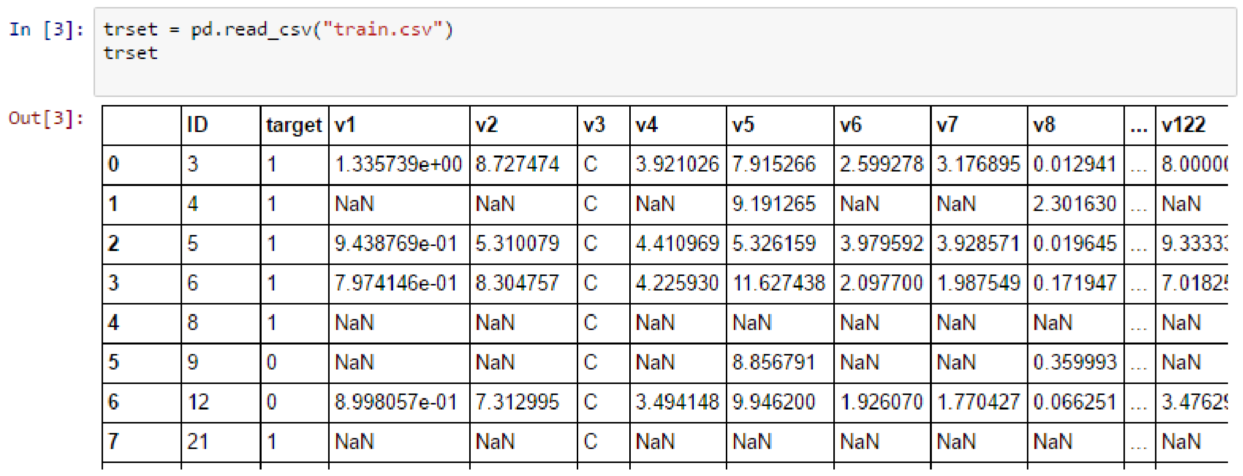
Exploratory Data Analysis

I used IPython notebooks for my analysis. The notebook with the code and comments is available in the Github folder where all the other DSI documents are located (https://github.com/rr2/DSI).

In the following sections, I summarize the steps taken, and the reasoning behind the decisions.

Evaluation of Missing Values

I read in the training data using Pandas' read_csv() function. This is what the head of the data set looks like:



```
In [3]: trset = pd.read_csv("train.csv")
        trset
```

Out[3]:

	ID	target	v1	v2	v3	v4	v5	v6	v7	v8	...	v122
0	3	1	1.335739e+00	8.727474	C	3.921026	7.915266	2.599278	3.176895	0.012941	...	8.00000
1	4	1	NaN	NaN	C	NaN	9.191265	NaN	NaN	2.301630	...	NaN
2	5	1	9.438769e-01	5.310079	C	4.410969	5.326159	3.979592	3.928571	0.019645	...	9.3333
3	6	1	7.974146e-01	8.304757	C	4.225930	11.627438	2.097700	1.987549	0.171947	...	7.0182
4	8	1	NaN	NaN	C	NaN	NaN	NaN	NaN	NaN	...	NaN
5	9	0	NaN	NaN	C	NaN	8.856791	NaN	NaN	0.359993	...	NaN
6	12	0	8.998057e-01	7.312995	C	3.494148	9.946200	1.926070	1.770427	0.066251	...	3.4762
7	21	1	NaN	NaN	C	NaN	NaN	NaN	NaN	NaN	...	NaN

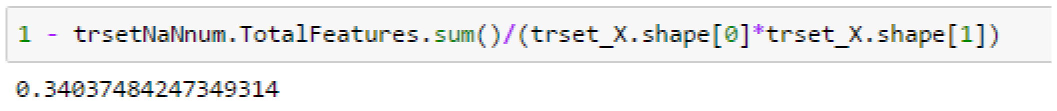
As we can see, there are several NaN's in this screen grab, which stand for missing values.

This dataframe has 114321 rows and 133 columns (features + ID + target).

Given that the majority of classification algorithms do not play well with missing data, the next agenda item was to explore the number and distribution of the NaN's. I decided to approach the number of NaNs on a per-sample basis. The goal was to answer questions such as:

1. How many rows contain NaN's?
2. How many rows are complete, i.e. they have no missing values?
3. Are the missing values evenly distributed, or are they concentrated across the data set?
4. What is the feature-wise distribution? Are there any features that do not have any missing values?
5. What is the overall proportion of NaN's in the data set, i.e. how sparse is it?

To answer these questions, I calculated the number of non-NaN (or available) features in each row. I also calculated that the data set is 34.04% sparse, which tells us that there are a significant number of missing values.

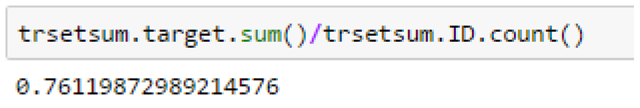


```
1 - trsetNaNnum.TotalFeatures.sum()/(trset_X.shape[0]*trset_X.shape[1])
```

```
0.34037484247349314
```

Some other metrics for the data set were:

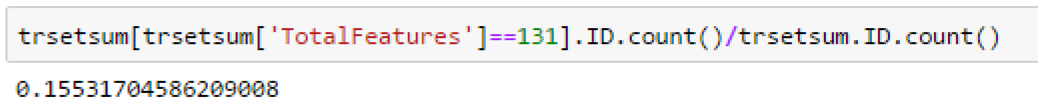
76.12% of the training samples were 1’s, i.e. they were expedited:



```
trsetsum.target.sum()/trsetsum.ID.count()
```

```
0.76119872989214576
```

15.53% of the samples had all 131 features present:

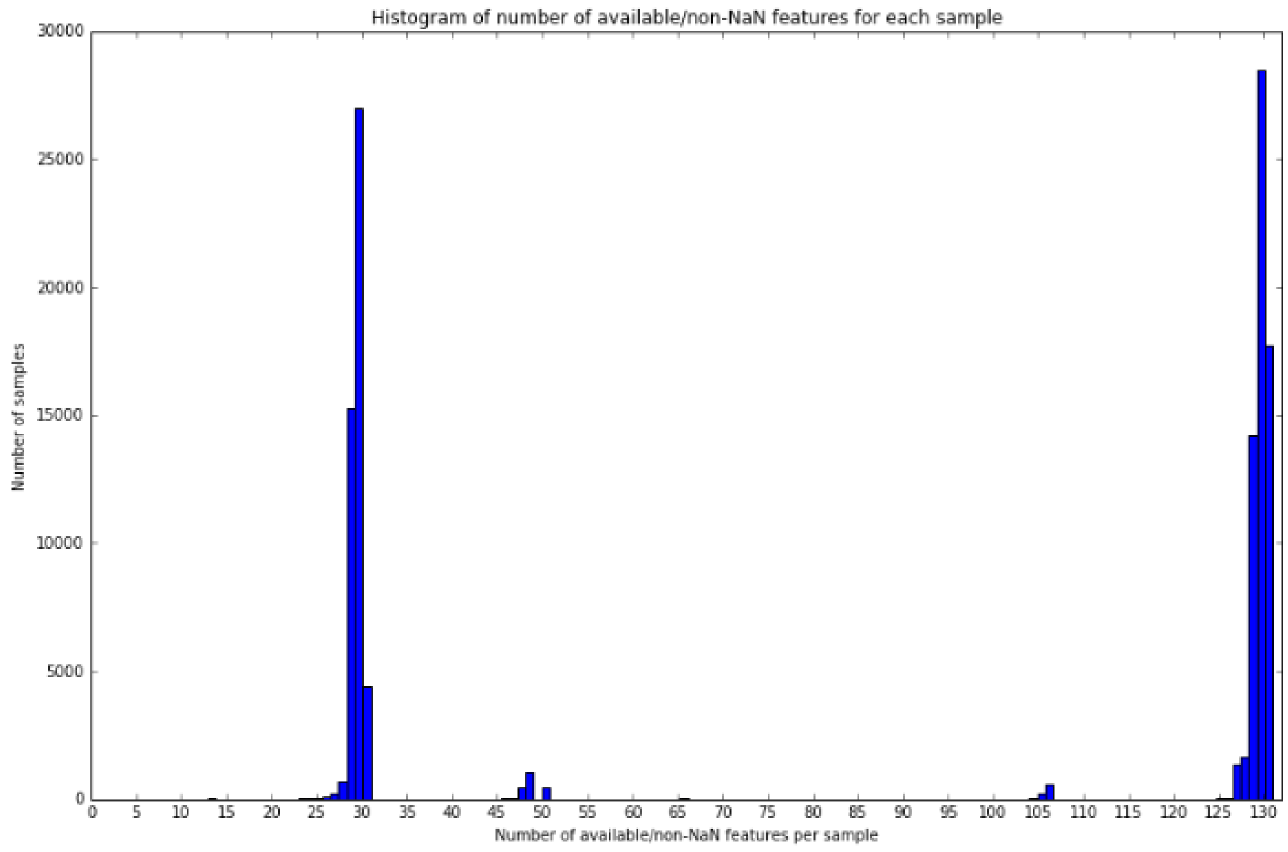


```
trsetsum[trsetsum['TotalFeatures']==131].ID.count()/trsetsum.ID.count()
```

```
0.15531704586209008
```

These numbers tell us that we are dealing with a 3:1 class ratio and that ~85% of samples have missing values.

Continuing with our investigation into missing values, I plotted a histogram of the number of available features per row. The resulting histogram gives us a clear answer to the question: “How many rows contain N number of features?”.



This histogram clearly demonstrates that there is a strong clustering in two buckets:

1. Rows with 25 to 31 available features
2. Rows with 125 to 131 available features

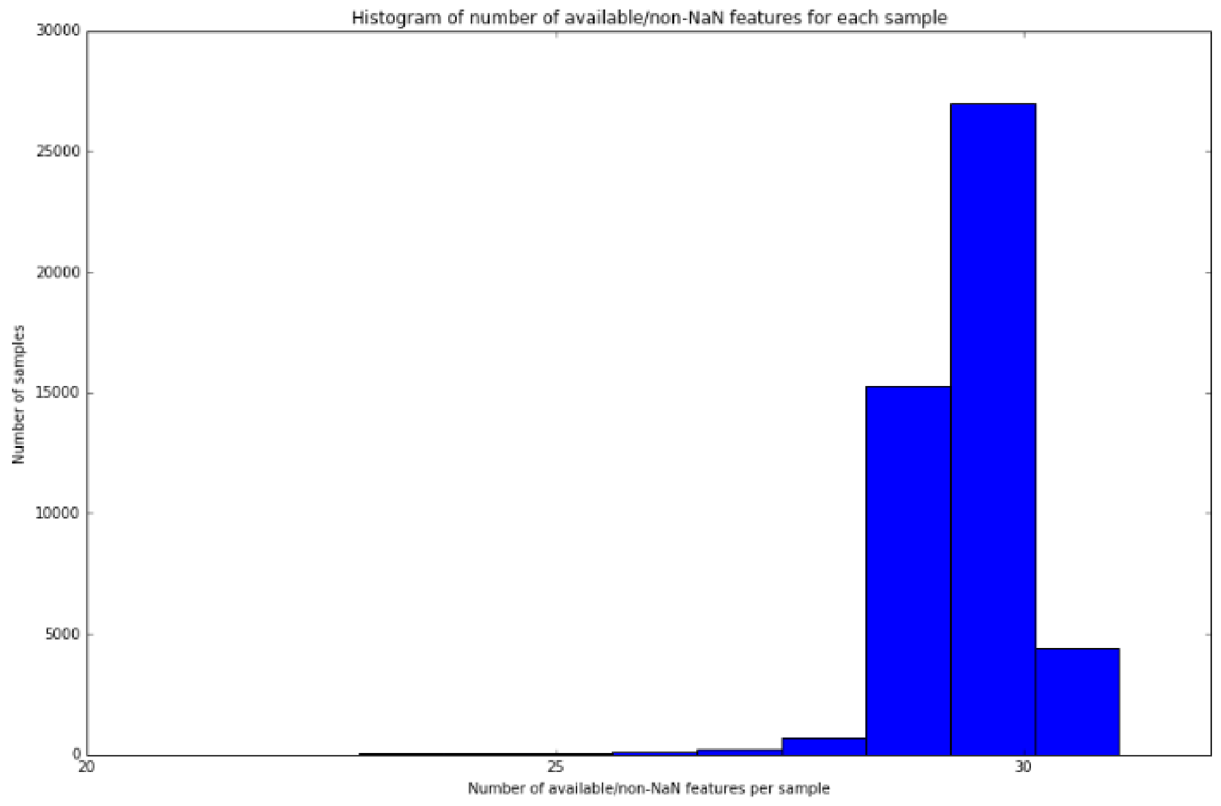
There are a small number of samples that do not fall into either of these buckets. We can calculate their percentage by

```
(trsetsum27_31.ID.count()+trsetsum127_131.ID.count())/trset.ID.count()
```

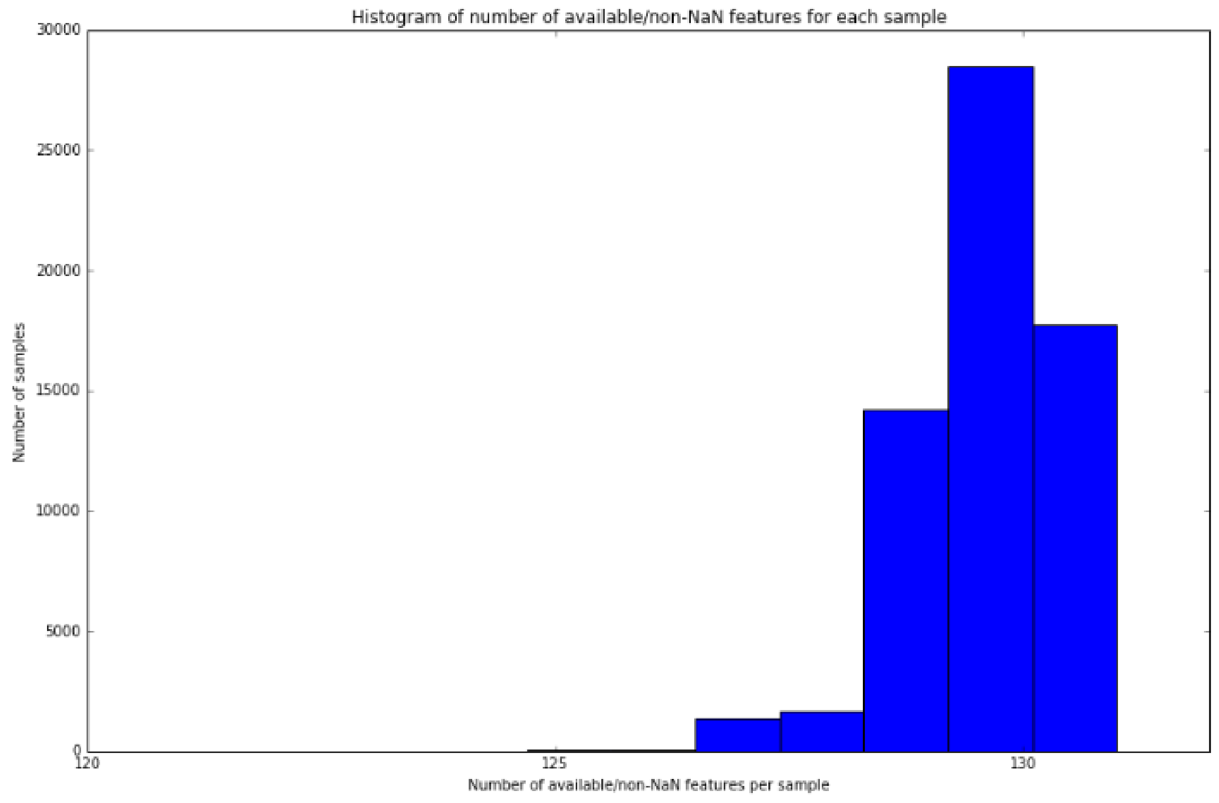
```
0.971475057076127769
```

Which shows us that 97.15% of the samples are in the two buckets described above.

We can observe the zoomed in distributions for these buckets as well:



For the range 25 to 31 features, and

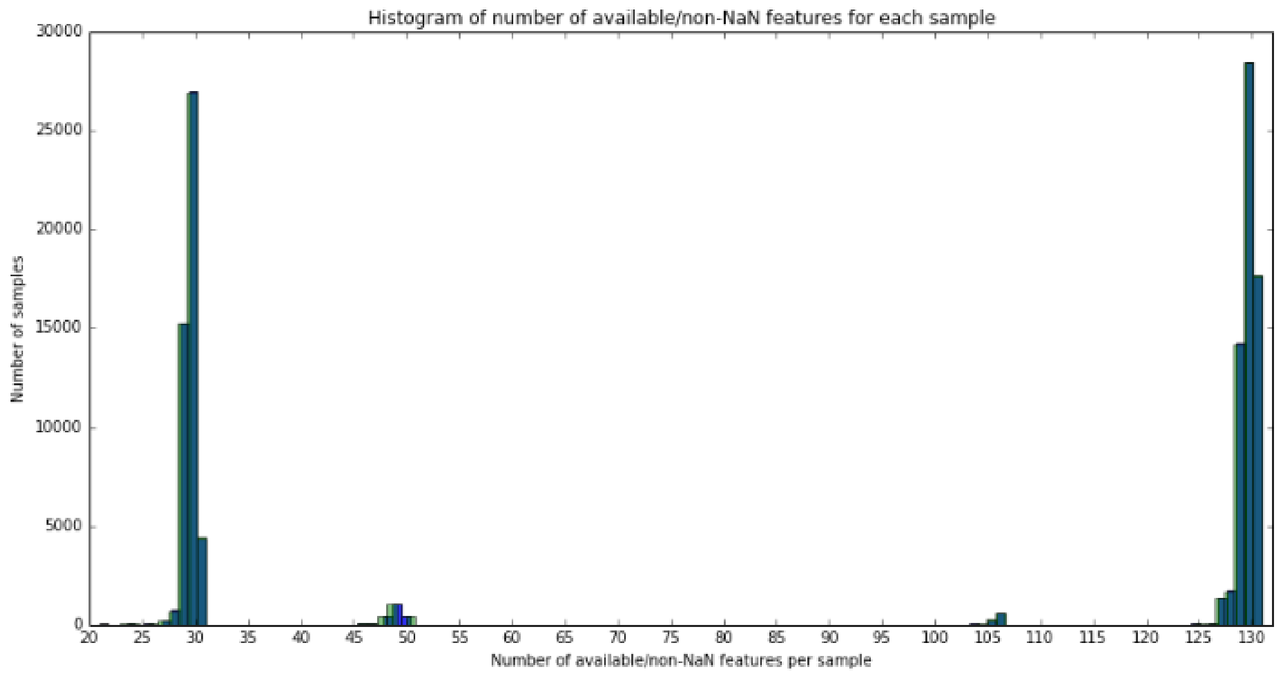


For the 125 to 131 feature range.

Discussion:

At this stage, it makes sense to consider the possibility of treating our training data as two separate data sets. We can then train two classifiers for either bucket of data, and use them to make predictions.

However, this would only be possible if the *test* data also organizes itself into these neat buckets. I created a similar histogram for the test data and compared the two:

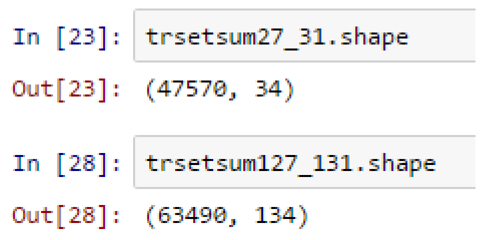


Remarkably, the training and test histograms overlap almost exactly with each other. This is the first sign that we may be able to divide our training and test sets into two and process them separately from each other.

The other consideration is the **distribution** of the missing values. If the training and test sets have the same number of missing values, but they are for different features, then we will not be able to split the data sets.

The procedure for checking that is to essentially compute the features that have zero available values for both the smaller subsets, and ensure that the list for the test set matches that for the training set. I was able to verify this.

Therefore, we go ahead and create the two subsets with the following dimensions:

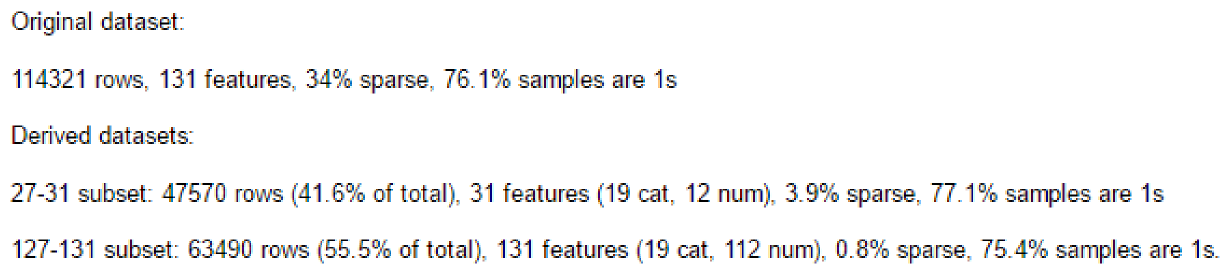


```
In [23]: trsetsum27_31.shape
Out[23]: (47570, 34)

In [28]: trsetsum127_131.shape
Out[28]: (63490, 134)
```

The additional column seen above is the 'TotalFeatures' column added to store the number ofavailable features per row. Also, please note that the subsets created were for 27 to 31 available features, and 127 to 131 available features. The reason? Sparsity!

The biggest win for us by making this split comes with respect to sparsity:



Original dataset:

114321 rows, 131 features, 34% sparse, 76.1% samples are 1s

Derived datasets:

27-31 subset: 47570 rows (41.6% of total), 31 features (19 cat, 12 num), 3.9% sparse, 77.1% samples are 1s

127-131 subset: 63490 rows (55.5% of total), 131 features (19 cat, 112 num), 0.8% sparse, 75.4% samples are 1s.

As we can see from the above summary, our new subsets are only **3.9% and 0.8% sparse**, down from the **34% sparsity** of the original data set.

Categorical and Numerical Features

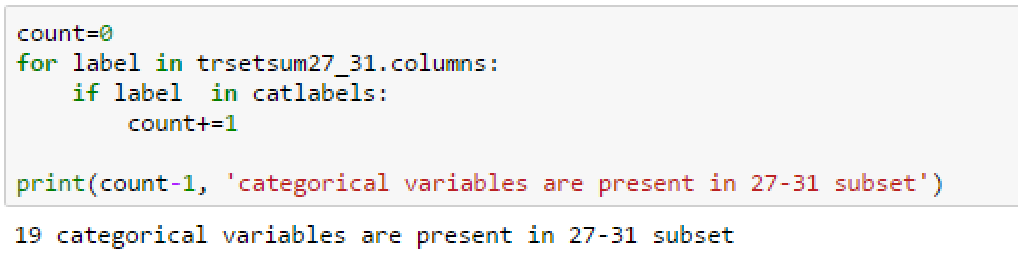
At this point, we still do not know the numbers of categorical and numerical features in our data. That is easy to do, and we find that:

```
['target', 'v3', 'v22', 'v24', 'v30', 'v31', 'v47', 'v52', 'v56', 'v66', '
v71', 'v74', 'v75', 'v79', 'v91', 'v107', 'v110', 'v112', 'v113', 'v125']
```

are our 19 categorical variables, excluding the target. This implies that 131 – 19 = 112 variables are numerical.

Now, after we split our data sets, how many categorical and numerical variables did we end up with in the smaller subset? Recall that our larger subset has between 127 and 131 features, so we are assured of the full representation of features in that subset.

This is easy to do:



```
count=0
for label in trsetsum27_31.columns:
    if label  in catlabels:
        count+=1

print(count-1, 'categorical variables are present in 27-31 subset')
```

```
19 categorical variables are present in 27-31 subset
```

So, we find that all 19 categoricals are represented in the smaller subset, and therefore, 12 numerical variables make up the remainder of the 31 features.

Unique Values

For categorical variables, the number of unique values is of interest to us. The reason is that most classification algorithms only work with numerical values. Therefore, categorical values have to converted to numericals using a step called one-hot encoding.

One-hot encoding simply refers to splitting the set of N unique values for a categorical variable into N vectors of size Nx1, each of which has all values equal to zero, except one which is set to 1. This is similar to writing the vector {a, b, c} as a linear combination of {a, 0, 0}, {0, b, 0} and {0, 0, c}.

This ends up creating N new variables for each categorical variable with N uniques. We normally drop one of the N new variables since the Nth variable is a linear combination of the other N-1 for a finite set of uniques.

Clearly, if N is large, then the number of new features created will also be very large. Therefore, we are interested in seeing if all of our categorical variables may be one-hot encoded. Let’s look at the number of uniques for each of the categorical features:

```
LABEL    27-31    127-131  Total
target   2        2        2
v3       4        4        4
v22      11934    13332    18211
v24      5        5        5
v30      8        8        8
v31      4        4        4
v47      9        9        10
v52      12       12       13
v56      108      115      123
v66      3        3        3
v71      5        6        9
v74      3        3        3
v75      4        3        4
v79      18       17       18
v91      7        7        8
v107     7        7        8
v110     3        3        3
v112     23       23       23
v113     37       35       37
v125     91       90       91
```

This immediately shows a concern for us: v22 has 18000+ unique values in the training set. When we inspect v22, we find what appear to be codes comprising 2 to 4 letters:

```
v22uniques = pd.DataFrame(trset['v22'].unique())
v22uniques.reset_index()
v22uniques.columns=['name']
v22uniques_sorted = v22uniques.sort_values(by='name')
v22uniques_sorted
```

	name
1766	AA
4830	AAA
4843	AAAA
1952	AAAB
10650	AAAC
16595	AAAE
15408	AAAF
8909	AAAG
12225	AAAH
935	AAAK
9471	AAAN
5971	AAAP
13383	AAAR
4429	AAAS

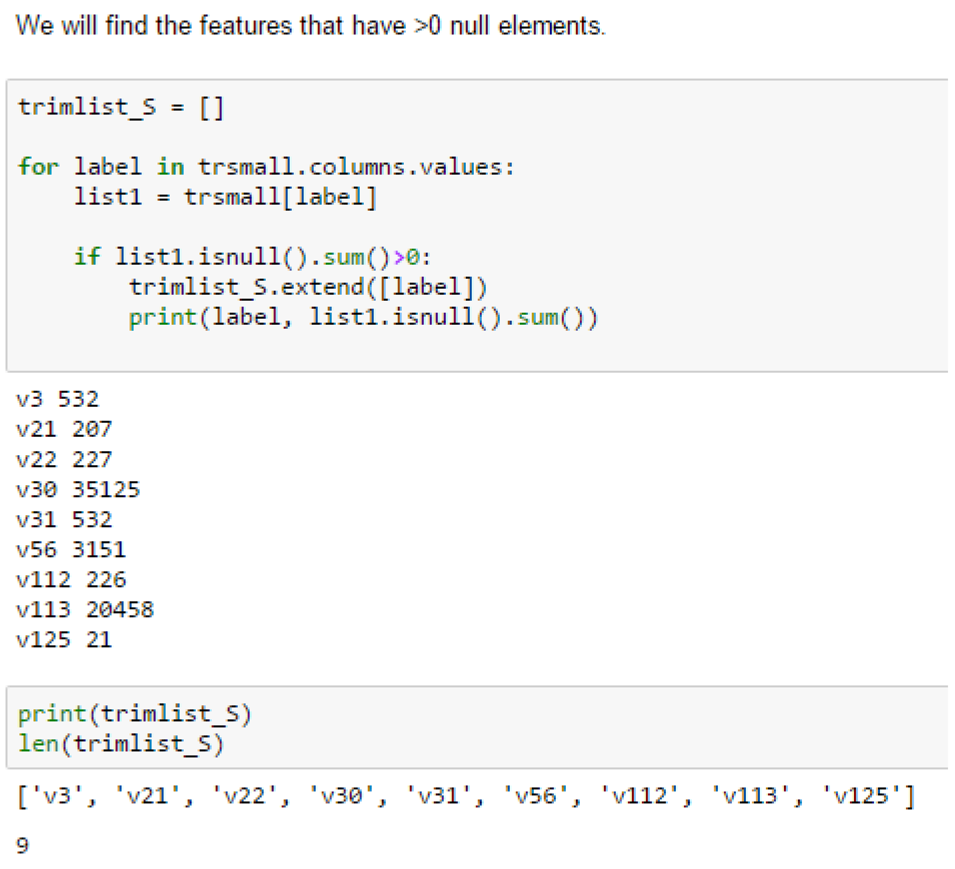
and so on till AHPS (via ZZZ). The first thought that comes to mind is that this may be a sequence of numbers coded as letters. However, recall that the organizers have told us that none of the categorical features are ordinal. Therefore, we cannot assume a sequence in these unique values.

Based on the assumption that a feature with such a high number of uniques would have a lesser impact on the classification, and also based on the fact that v22 would make the problem computationally intractable if one-hot encoded, I decided to drop v22 from the features for a first pass.

Before proceeding further, I computed the number of unique values for each of the numerical values as well. However, beyond the fact that 4 of the 112 numerical variables take a small number of integer values, and the others take float values, there was nothing remarkable about that result.

Missing Data Revisited: To impute or not to impute?

So far, we have encountered two serious difficulties with our data: the (reduced) number of missing values and the feature with 18000 uniques. While we have discarded the v22 feature, it is clear that the rows with missing values cannot simply be discarded since they represent a fairly high number of samples:



We will find the features that have >0 null elements.

```
trimlist_S = []

for label in trsmall.columns.values:
    list1 = trsmall[label]

    if list1.isnull().sum()>0:
        trimlist_S.extend([label])
        print(label, list1.isnull().sum())
```

```
v3 532
v21 207
v22 227
v30 35125
v31 532
v56 3151
v112 226
v113 20458
v125 21
```

```
print(trimlist_S)
len(trimlist_S)
```

```
['v3', 'v21', 'v22', 'v30', 'v31', 'v56', 'v112', 'v113', 'v125']

9
```

The above list shows the features that have missing values and the number of rows in which they are missing. Clearly, we would have to discard at least 35000+ rows (and perhaps more) to get rid of all the missing values. With the subset only comprising 47000 samples, this is not viable. Similar results were found for the larger subset.

Therefore, we consider the idea of imputation. Imputing missing values means coming up with replacement values for the ones that are missing. There are several approaches for imputation, such as filling in the median or mean value for the field, or to replace categorical values with a random distribution of the observed unique values in the same distribution as they are observed in the data.

There are pros and cons of imputation. The main advantage is that we will have a complete data set and we can choose from a variety of classification algorithms. The main drawback is that we have already partitioned the data and therefore, imputation across the two subsets of data could lead to pollution of the relationships in the data.

Based on reading the discussions in the forums, there seemed to be a way forward that would allow classification without imputation. This was to use XGBoost as the classifier.

Choice of Classifier: XGBoost

I will make a slight detour here to speak about XGBoost, since it provided input to the EDA process as well.

XGBoost stands for Extreme Gradient Boosting, which is a version of a gradient boosting method for trees. In particular, it has found widespread applications in ML competitions. It works very well with sparse data, and does not require imputation of missing data.

The advantages of using XGBoost are several:

1. It is based on tree ensembles – which means that it already does ensembling of a large number of trees for us.
2. Trees are naturally well suited for handling missing values. However, XGBoost handles missing values especially well because it places the missing value in both leaf nodes, one at a time, and sees which branch gives the better result.
3. XGBoost has regularization built into it. All we have to do is specify the L1 or L2 regularization parameter.
4. XGBoost has several parameters for tuning, and has a nice Python API, which I have used.

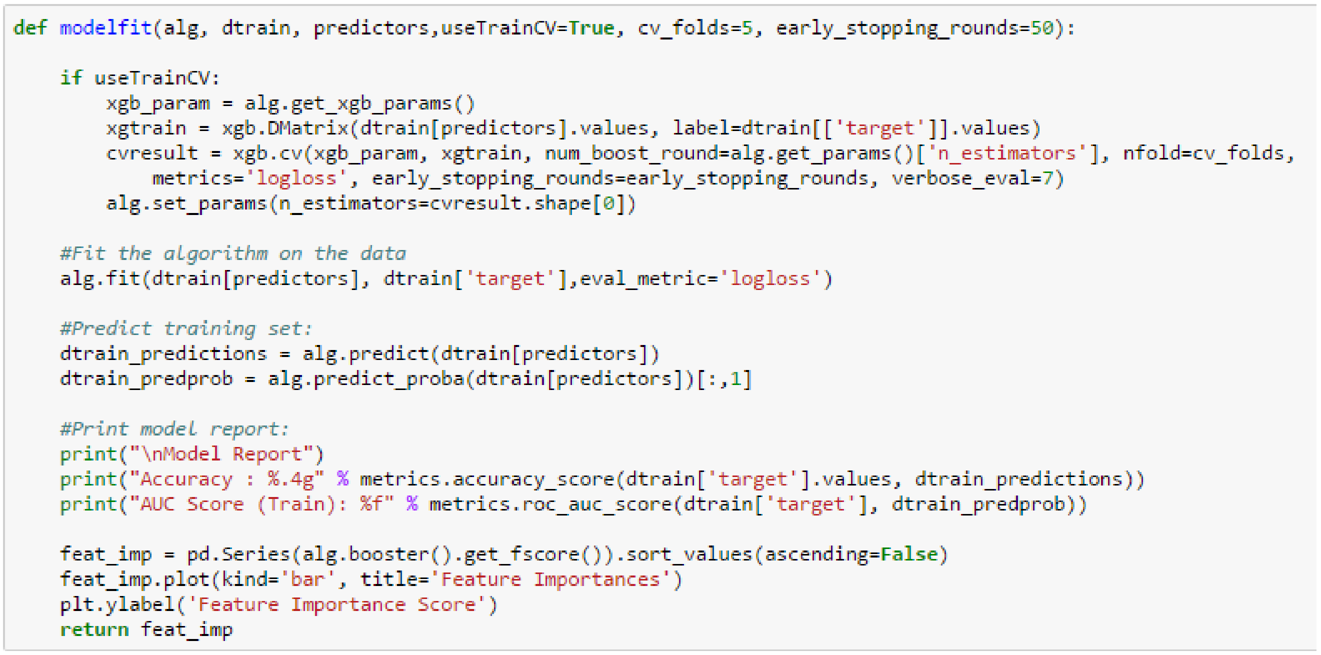
Based on the choice of XGBoost, I decided to not impute the missing data.

Examining Possible Feature Correlation

While we have split our data set into two parts, there are still expected to be some common themes to be found in the data. At the moment, we have no intuition about which features are more important than the others. A common way to discover correlated variables is to make scatter plots of the numerical features. But with 112 numerical features, plotting all the numerical features against each other is large task.

Also, we expect that some features will be less important than others. So, we would like to first get an idea of which features are important, and then examine if they are correlated in any way.

For this, I adapted with minor modifications the following function used in the tutorial at https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/



```
def modelfit(alg, dtrain, predictors,useTrainCV=True, cv_folds=5, early_stopping_rounds=50):

    if useTrainCV:
        xgb_param = alg.get_xgb_params()
        xgtrain = xgb.DMatrix(dtrain[predictors].values, label=dtrain[['target']].values)
        cvresult = xgb.cv(xgb_param, xgtrain, num_boost_round=alg.get_params()['n_estimators'], nfold=cv_folds,
            metrics='logloss', early_stopping_rounds=early_stopping_rounds, verbose_eval=7)
        alg.set_params(n_estimators=cvresult.shape[0])

    #Fit the algorithm on the data
    alg.fit(dtrain[predictors], dtrain['target'],eval_metric='logloss')

    #Predict training set:
    dtrain_predictions = alg.predict(dtrain[predictors])
    dtrain_predprob = alg.predict_proba(dtrain[predictors])[:,1]

    #Print model report:
    print("\nModel Report")
    print("Accuracy : %.4g" % metrics.accuracy_score(dtrain['target'].values, dtrain_predictions))
    print("AUC Score (Train): %f" % metrics.roc_auc_score(dtrain['target'], dtrain_predprob))

    feat_imp = pd.Series(alg.booster().get_fscore()).sort_values(ascending=False)
    feat_imp.plot(kind='bar', title='Feature Importances')
    plt.ylabel('Feature Importance Score')
    return feat_imp
```

This function takes as argument an XGBClassifier object (for binary classification), number of folds for cross validation, and a criterion for stopping further iterations.

It then does a cross validation (in the case below, a 5-fold CV) and returns the model accuracy and area under the ROC curve. For the competition, the metric used to evaluate the model is logloss, which is the criterion selected in this function also.

When the function exits, it returns an ordered array of features in the order of their importance to the classification. It also returns the weights computed by XGBoost in constructing the ensembles of trees, boosting trees that fit the cross-validation data better.

I start off with some default parameters for the XGBClassifier:

xgb1 = XGBClassifier(learning_rate =0.1, n_estimators=1000, max_depth=5,

min_child_weight=1, gamma=0, subsample=0.8, colsample_bytree=0.8,

objective= 'binary:logistic', nthread=4, scale_pos_weight=1, seed=27)

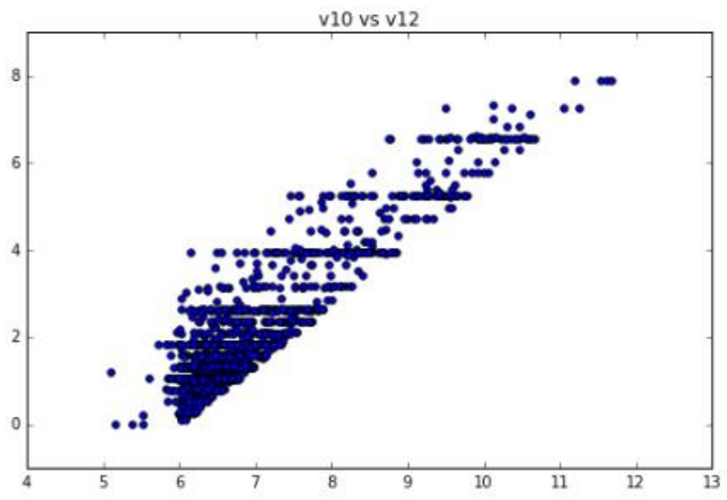
A discussion of these parameters is available here.

Then I ran the classifier against both subsets of the data in order to see if there were any features that were found to be important in both subsets. When I inspected the list of important features, it became readily apparent that a set of 8 features ranked very highly in both classifiers.

```
selectedlabels = ['v50','v12','v21','v40','v10','v114','v14','v34']
```

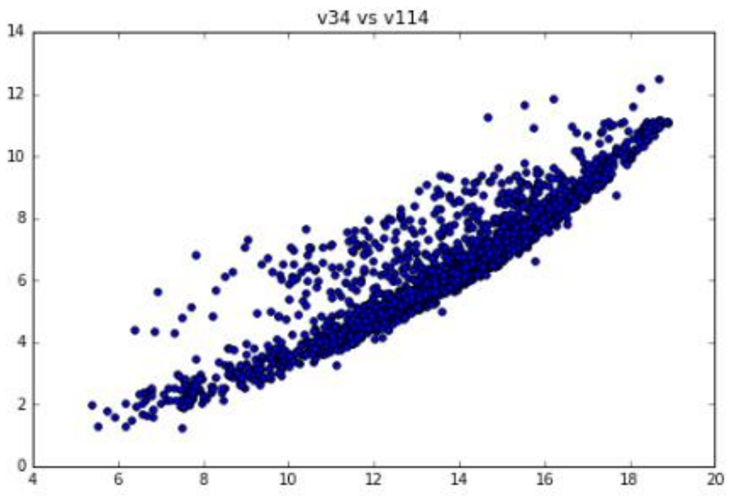
These 8 features were the top 8 in order of importance for the smaller data set, and they were also in the top 14 features for the large set (not in the same order).

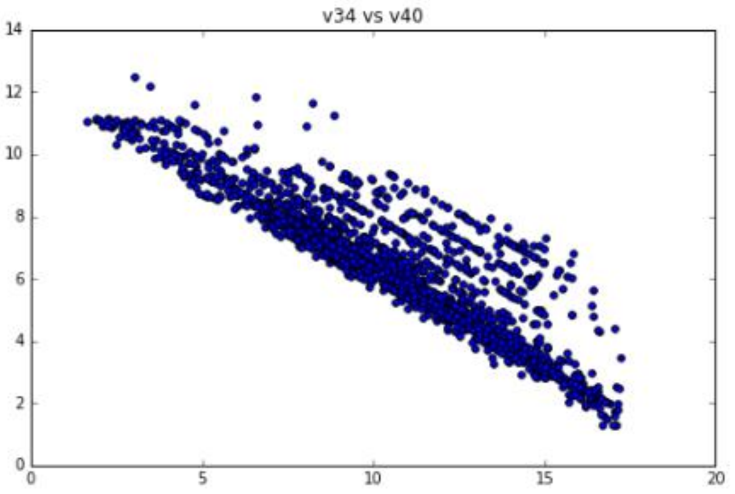
Therefore, I decided to plot the scatter charts for these variables against each other (for the entire dataset), and see if they held any evidence of correlation. Correlations are important to look at because they can cause the model to learn behavior that is more strongly weighted towards the correlated features. The most interesting plots are shown here:

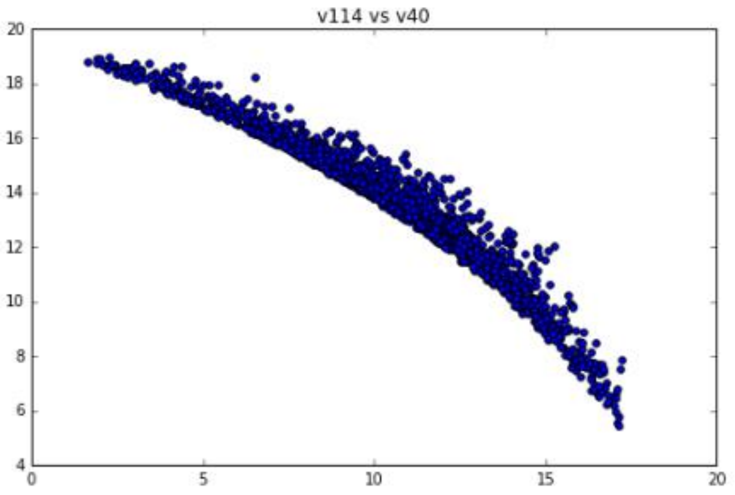


As we can see here, v10 and v12 appear to be correlated.

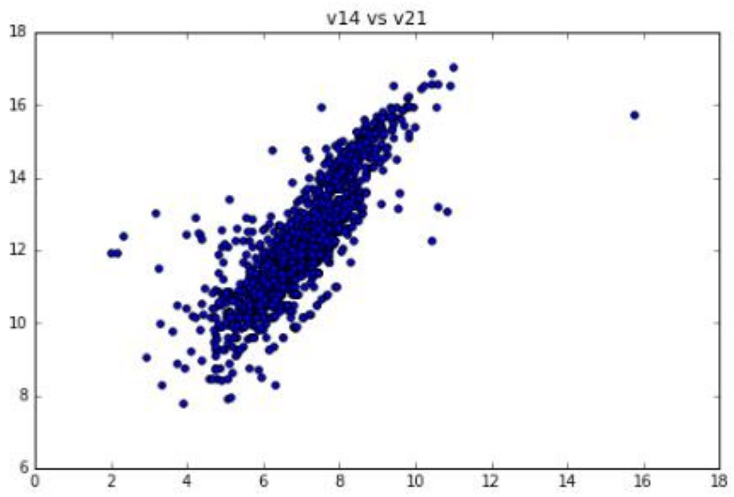
Another group that showed some evidence of correlation was the triad of v34, v40 and v114:







Finally, v14 and v21 also showed some correlation.

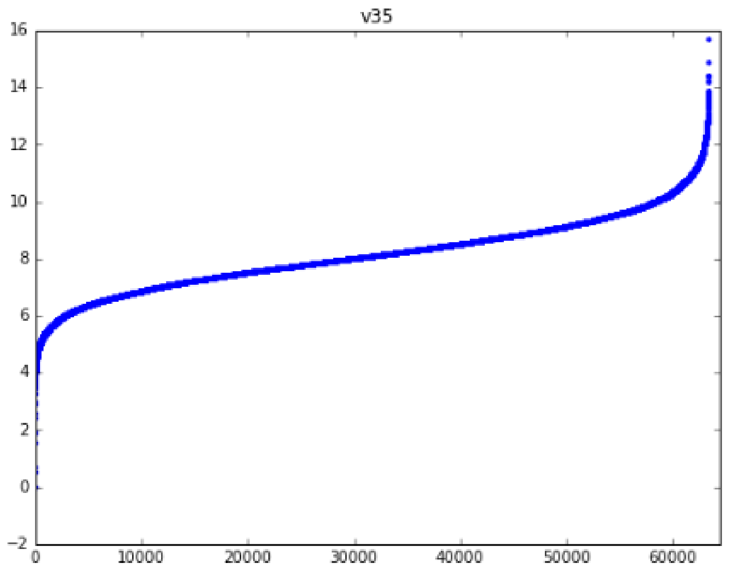


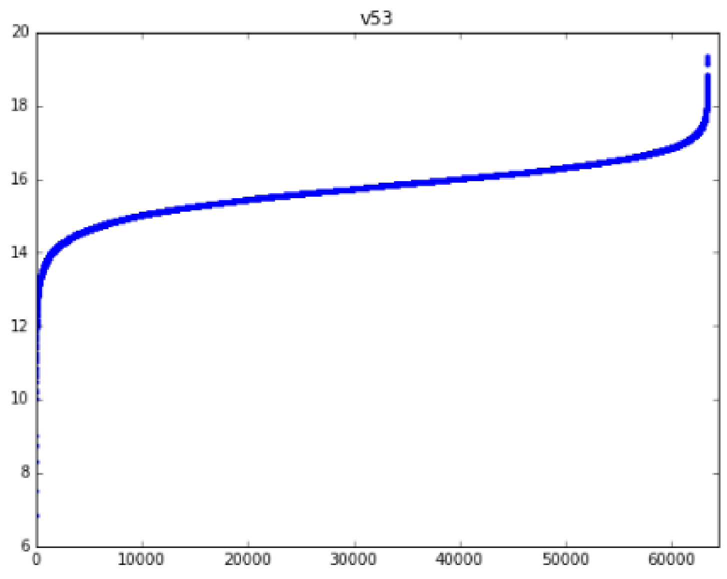
One way to account for these features is to drop at least one of them. This has a much greater impact on methods where correlated columns can cause problems with invertibility of the matrix. However, we will keep these features in since we are using a boosted trees model.

Distribution of Numerical Variables

We noted earlier that out of the 112 numerical variables, 4 take integer values. However, we do not know anything about the other 108. Let’s plot the distribution of these variables.

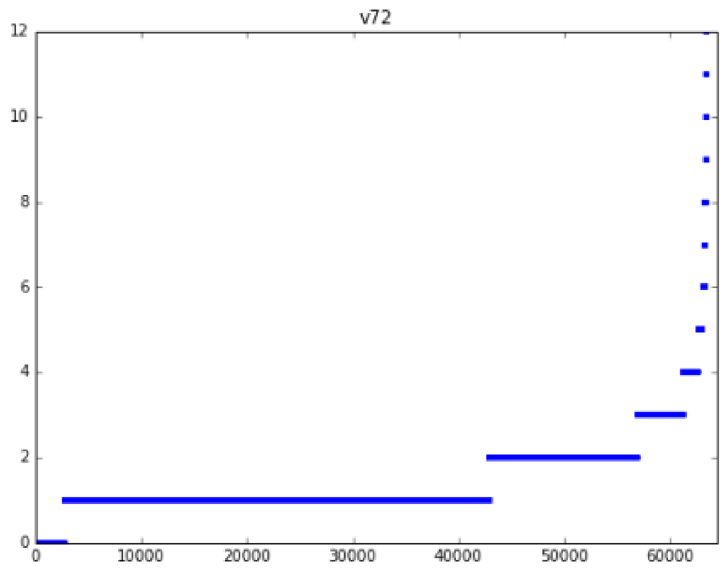
We find that all the numerical variables are scaled to the range [0,20] and have approximately similar distributions. Examples are shown below:





This clearly show that the variables have been transformed (scaled and translated) to the [0,20] range. This makes it difficult to interpret what each of the variables stands for.

Also, as mentioned earlier, there are 4 integer-valued variables. Here is an example of v72:



I also plotted these variables color-coded by their target values, but there were no distinguishing features between the 0- and 1- sets for any of the variables.

Prediction of Classes Using XGBoost

The preceding data analysis resulted in the following artifacts and observations:

1. 97% of the training and test data is grouped into two major buckets. Therefore, we split our data set into subsets, and plan to train two classifiers independently to make predictions.
2. We removed categorical feature v22 due to its high cardinality.
3. As a result, one subset has 30 features, and the other has 130 features.
4. None of the numerical variables shows a clustering effect by itself when grouped by target value.

Now, we dive into the setup of the prediction setup with XGBoost.

Parameter Tuning

As mentioned previously, XGBoost has several parameters that can be tuned. The following descriptions for these parameters have been taken from Ian Howson's page on the R version of XGBoost :

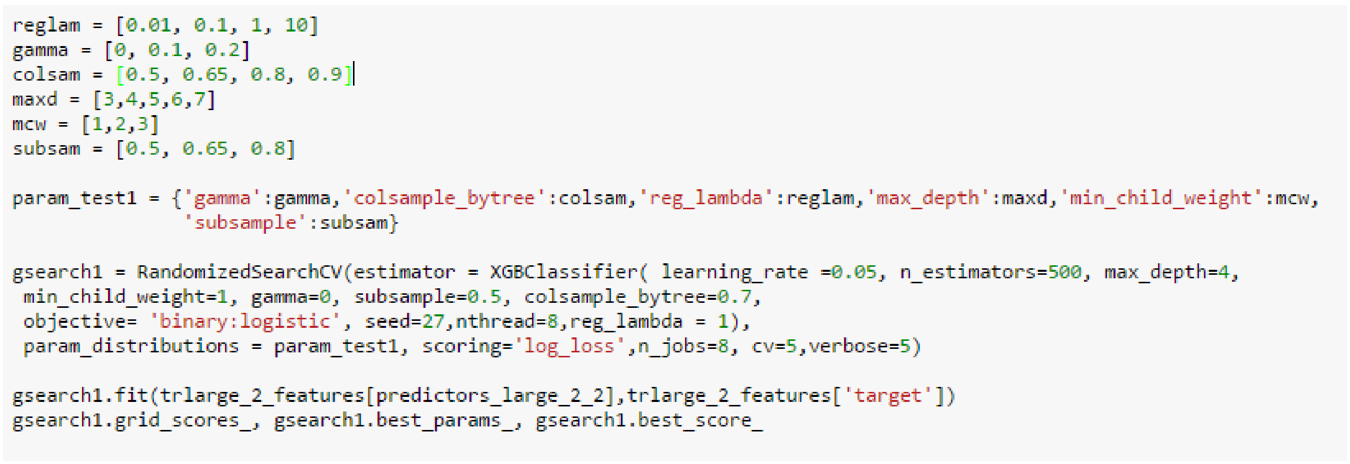
1. Learning rate (learning_rate): Scales the contribution of each tree by a factor of 0 < learning_rate < 1 when it is added to the current approximation. Used to prevent overfitting by making the boosting process more conservative. Low value means model more robust to overfitting but slower to compute.
2. Gamma: minimum loss reduction required to make a further partition on a leaf node of the tree. the larger, the more conservative the algorithm will be.
3. Max_depth: maximum depth of a tree.
4. Minimum child weight: (min_child_weight): Minimum sum of instance weight(hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than min_child_weight, then the building process will give up further partitioning. The larger, the more conservative the algorithm will be.
5. Subsample: subsample ratio of the training instance. Setting it to 0.5 means that xgboost randomly collected half of the data instances to grow trees and this will prevent overfitting.
6. Colsample_bytree: subsample ratio of columns when constructing each tree.
7. Reg_lambda: L2 regularization term on weights
8. Reg_alpha: L1 regularization term on weights

The addition of a regularization term to the model is very helpful in controlling overfitting.

In order to explore the space, I used GridSearchCV and RandomizedSearchCV from Scikit-learn.

RandomizedSearchCV and GridSearchCV

First, RandomizedSearchCV was used to try combinations of a larger number of parameters, and then GridSearchCV was used to fine tune a subset of those parameters:



```
reglam = [0.01, 0.1, 1, 10]
gamma = [0, 0.1, 0.2]
colsam = [0.5, 0.65, 0.8, 0.9]
maxd = [3,4,5,6,7]
mcw = [1,2,3]
subsam = [0.5, 0.65, 0.8]

param_test1 = {'gamma':gamma,'colsample_bytree':colsam,'reg_lambda':reglam,'max_depth':maxd,'min_child_weight':mcw,
               'subsample':subsam}

gsearch1 = RandomizedSearchCV(estimator = XGBClassifier( learning_rate =0.05, n_estimators=500, max_depth=4,
 min_child_weight=1, gamma=0, subsample=0.5, colsample_bytree=0.7,
 objective= 'binary:logistic', seed=27,nthread=8,reg_lambda = 1),
 param_distributions = param_test1, scoring='log_loss',n_jobs=8, cv=5,verbose=5)

gsearch1.fit(trlarge_2_features[predictors_large_2_2],trlarge_2_features['target'])
gsearch1.grid_scores_, gsearch1.best_params_, gsearch1.best_score_
```

The above code sets up a Randomized Search over 6 parameters, and generates the logistic loss for each combination. It then returns the best parameter set found.

RandomizedSearchCV and GridSearchCV cover the same space. However, GridSearchCV does an exhaustive run through all possible combinations, while RandomizedSearchCV creates a smaller subset at random and does a search on those parameters.

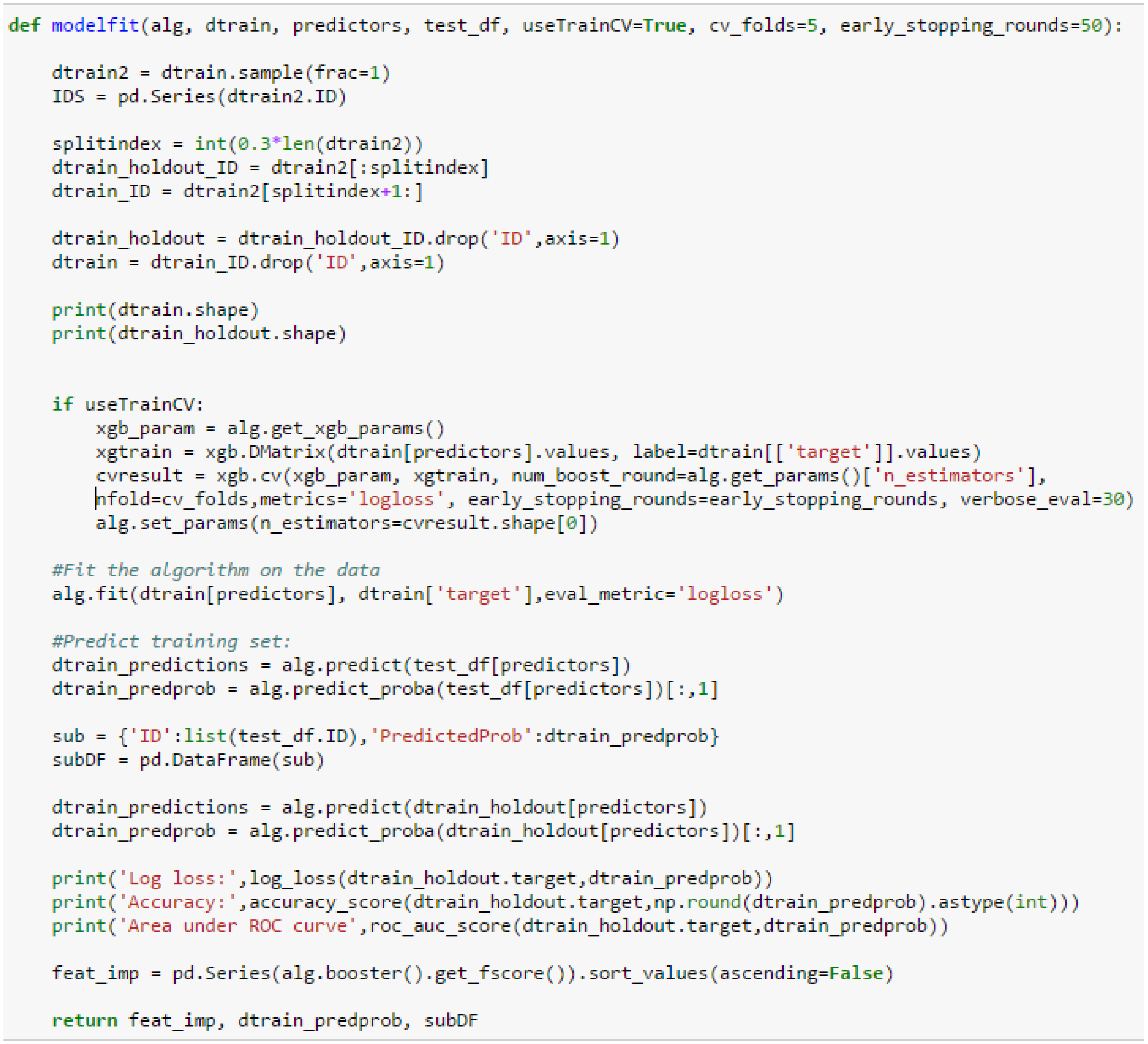
XGB.CV and the ModelFit function

XGBoost also provides its own Cross-validation routine called XGB.CV. This was used in fitting the model to the data.

In the following image, we see an expanded form of the ModelFit function introduced earlier. This function takes as argument an XGBClassifier model and parameters, training data, and test data.

The training data is first split into a holdout set and a set that is actually used for training in a 30-70 split. This was done to ensure that the reported accuracy would be on a dataset that was not used for learning. Another reason for this was that the Kaggle competition only allows submissions with predicted probabilities for all rows. Since we are working with a subset of the total set, we do not have predictions for all the rows.

Therefore, this method allows us to get an estimate of the accuracy on unseen data. The split is done randomly each time, so we do not have test-training data leakage.



```python
def modelfit(alg, dtrain, predictors, test_df, useTrainCV=True, cv_folds=5, early_stopping_rounds=50):

    dtrain2 = dtrain.sample(frac=1)
    IDS = pd.Series(dtrain2.ID)

    splitindex = int(0.3*len(dtrain2))
    dtrain_holdout_ID = dtrain2[:splitindex]
    dtrain_ID = dtrain2[splitindex+1:]

    dtrain_holdout = dtrain_holdout_ID.drop('ID',axis=1)
    dtrain = dtrain_ID.drop('ID',axis=1)

    print(dtrain.shape)
    print(dtrain_holdout.shape)


    if useTrainCV:
        xgb_param = alg.get_xgb_params()
        xgtrain = xgb.DMatrix(dtrain[predictors].values, label=dtrain[['target']].values)
        cvresult = xgb.cv(xgb_param, xgtrain, num_boost_round=alg.get_params()['n_estimators'],
        nfold=cv_folds,metrics='logloss', early_stopping_rounds=early_stopping_rounds, verbose_eval=30)
        alg.set_params(n_estimators=cvresult.shape[0])

    #Fit the algorithm on the data
    alg.fit(dtrain[predictors], dtrain['target'],eval_metric='logloss')

    #Predict training set:
    dtrain_predictions = alg.predict(test_df[predictors])
    dtrain_predprob = alg.predict_proba(test_df[predictors])[:,1]

    sub = {'ID':list(test_df.ID),'PredictedProb':dtrain_predprob}
    subDF = pd.DataFrame(sub)

    dtrain_predictions = alg.predict(dtrain_holdout[predictors])
    dtrain_predprob = alg.predict_proba(dtrain_holdout[predictors])[:,1]

    print('Log loss:',log_loss(dtrain_holdout.target,dtrain_predprob))
    print('Accuracy:',accuracy_score(dtrain_holdout.target,np.round(dtrain_predprob).astype(int)))
    print('Area under ROC curve',roc_auc_score(dtrain_holdout.target,dtrain_predprob))

    feat_imp = pd.Series(alg.booster().get_fscore()).sort_values(ascending=False)

    return feat_imp, dtrain_predprob, subDF
```

However, we do make predictions for test data as well. The idea here was to create the predictions using the two classifiers for 97% of the data, and then figure out a way to also get estimates for the other <3% of samples. However, I was not able to get to that point.

Combination of Categorical Labels across Training and Test Data

An important consideration that came up during making predictions for the test data, was that some of the unique values for categoricals in the test data were absent in the training data. Therefore, I made the decision to use the list of uniques in the test and training data to make a master list. While this is making use of the test data, the assumption is that in a problem with available classes, we would have the list at our disposal. Therefore, all labels were used to create the master list before doing one-hot encoding.

Successive Reduction of Feature Space

After tuning the parameters and setting up the framework to divide the training data to make predictions, now I was ready to start calculating probabilities. In order to prune the list of features further, I also used the 'feature_importance' values returned by XGBoost to remove the least important features from subsequent prediction runs.

```
for i in range(0,20):
    xgb_sm = XGBClassifier(learning_rate =0.02,n_estimators=500,max_depth=5,min_child_weight=1,
     gamma=0.2,subsample=0.8,colsample_bytree=0.5,objective= 'binary:logistic',
     nthread=8,reg_lambda = 1, max_delta_step=1,seed=i)

    xgb_lg = XGBClassifier(learning_rate =0.02,n_estimators=500,max_depth=6,min_child_weight=1,
     gamma=0.1,subsample=0.65,colsample_bytree=0.8,objective= 'binary:logistic',nthread=8,
     reg_lambda =10, max_delta_step=1, seed=i)
```

These were the two classifiers with parameters from the Randomized and Grid Search CV.

```
print(len(predictors_sm), ' features')
feat_imp_sm, predprob, subDF_sm = modelfit(xgb_sm, trset_sm_run, predictors_sm,testv_sm_xgb)

print(len(predictors_lg), ' features')
feat_imp_lg, predprob, subDF_lg = modelfit(xgb_lg, trset_lg_run, predictors_lg,testv_lg_xgb)

fname = 'sub'+str(i+1)+'.csv'
subDF = pd.concat([subDF_sm,subDF_lg])
subDF.to_csv(fname)

featdf = pd.DataFrame(feat_imp_sm)
featdf.reset_index()
featdf['newindex']=range(0,len(featdf))
featdf['vars']=featdf.index
featdf.set_index('newindex')

droplist = list(featdf.vars[-20:])
toplist = list(featdf.vars[:50])
```

Here, I call the modelfit function with the twos sets of training data and then create a submission CSV file for Kaggle. Then I look at the Important features list returned by XGBoost and remove the bottom 20 from the list.

This code then re-runs in a loop with the reduced feature set.

Results

Overall results of the prediction have been good.

The best results for the two subsets are:

30-feature data set (Best results on holdout training data)

```
Log loss: 0.44355929663
Accuracy: 0.800154104791
Area under ROC curve 0.771802793489

Log loss: 0.445006117821
Accuracy: 0.795951246848
Area under ROC curve 0.769495204621

Log loss: 0.445665628794
Accuracy: 0.797702437658
Area under ROC curve 0.767029763335
```

130-feature data set (Best results on holdout training data)

```
Log loss: 0.475450764217
Accuracy: 0.776955380577
Area under ROC curve 0.747324417547

Log loss: 0.475835528177
Accuracy: 0.775695538058
Area under ROC curve 0.753182134266

Log loss: 0.478191525982
Accuracy: 0.775170603675
Area under ROC curve 0.744346197938
```

These results indicate that we have a model which is competent, but not great. In particular, the outcome of several different approaches (Please see the next page) has been to maintain this level of performance, but not better it.

More specifically, it may be argued that a classifier that does not have the benefit of any domain knowledge or correlations has done a good job if it gets between 75% and 80% of the predictions right. This by itself is a good enough result to justify more effort being spent on the problem so that it becomes a valuable decision aid.

The area under the ROC curve is in the region of 0.75 for both classifiers, which indicates that misclassification is a considerable problem.

Other Unsuccessful Approaches

During the course of this project, I tried a number of other strategies which had either an uncertain effect or no effect at all. I am listing them here for the sake of completeness:

1. **Removing correlated features:** As discussed in the section on correlated variables, there were a few features that appeared to be correlated when we looked at the scatter plots. However, removing them did not improve the logistic loss.

2. **Successive reduction of feature space:** Reducing the feature space of ~450 variables gradually based on feature importance did not help significantly in the accuracy. However, it did speed up the model, and certainly suggested that a similar accuracy is possible with a smaller set of data. However, this is possibly due to the nature of XGBoost and boosted trees in general. The least important features are already treated as such and their absence will not help the algorithm significantly.

3. **Removal of features with large number of missing values:** We saw that features v30 and v113 had a significant number of missing values in both data sets. However, for reasons similar to the above, their removal did not help much.

4. **Creating pair wise combinations of categorical variables:** An initial idea was to create pairwise combinations of categorical variables and add them to the feature set. However, this had two disadvantages:

    a. It increased the number of features post one-hot encoding to over 2400
    b. It made the organization of data into two non-sparse buckets impossible, since now we had missing values for all combinations where one of the variables involved was missing.

    As a result, I did not use this strategy.

5. **Reducing numerical values to percentile-based buckets:** The relatively standard form of the numerical variables' distribution suggested that there may be a benefit to labeling outliers separately from the values that are closer to the mean. As a result, I tried to replace all numerical variables by their binned equivalents: bottom 5%, next 45% (till median), 45% just above median, and top 5%. However, while this strategy made the computation faster, it did not help accuracy.

Challenges

This project, like any other investigative effort, came with a set of challenges that were not apparent at the beginning. Some of these are discussed here.

1. Computational resources: It became apparent once I began the simulations that my personal computer would not be able to function effectively under the performance and time constraints. To overcome this, I experimented with Amazon Web Services EC2 machines and picked one for the remainder of the project. Fortunately, Anaconda has its own Linux AMI, which meant that it was possible to get a machine that came with the latest Python and Anaconda installed.

2. Installation of XGBoost wrapper for Python: While there was little doubt that XGBoost would be the best choice for this project, it was a significant challenge to get the Python wrapper installed along with XGBoost. My lack of Linux background was also a factor.

3. Kaggle submission format: This was a relatively low-impact issue. Since I was using two classifiers, which did not cover the entire test set, I did not have a complete submission file. However, I worked around this by presenting results on the holdout data, which is assumed to be representative of the test data.

4. Anonymization of Data: In hindsight, this was the biggest drawback of using this project as my Capstone project. The non-interpretability of feature labels, as well as the relative non-interpretability of results from XGBoost meant that it was very hard to glean business insights from the work that I did. While I learned a lot about the inner workings of boosted trees, feature engineering, and exploratory analysis, the project would have been more rewarding and impactful if the data set had been more “real-world”.

Conclusions

1. Stronger feature engineering is required for this problem. This is evidenced by a variety of feature selection and parameter tuning approaches being unable to improve performance of the model beyond a certain point.
2. Feature engineering in this case implies a level of de-anonymization of the feature labels. It is quite likely that some of the data have strong correlations. However, because of the scaling of the numerical data, this information has been hidden.
3. XGBoost will be my algorithm of choice for a number of reasons:
    a. In-built ensembling : This is a huge benefit for a problem with as many missing values since there are very few algorithms that can be used in an ensemble.
    b. Regularization: Both L1 and L2 regularization are available and I saw significant improvement by tweaking these parameters.
    c. Easy to interpret parameters: The tree model lends itself nicely to parameters that are easy to understand
4. In hindsight, while this was a great opportunity to become comfortable with uncertainty in machine learning, it was not a well-advised choice of project for the Data Science Intensive.