Kaggle Competition

Javier Palomares

Instructions:

This is the in class kaggle competition for Data Mining EE 380L. The data needed is in the Data page and you can view your position on the leaderboard. You will only be able to see your position on the public leaderboard. The public leaderboard uses a subset of the testing data that we give you to rank you. You can see your position here. However, there is also a private leaderboard that uses the remaining data in your test set. Please make sure to not overfit. Hopefully, if you do well in the public leaderboard, you will do well in the private leaderboard, but it is still possible to overfit the public leaderboard. Consider using your own validation schemes to get a sense of your generalization error as well.

The kaggle competition is to be done individually. Please don't collaborate in any way as this is a competition amongst you all. You're ranking on both the public and private leaderboards will be part of your grade for this assignment. There is no other information on the features. Please explore the data to understand the features and decided how you want to proceed. Points will be given for creativity as well so try lots of different things (albeit with adequate justification and explanation for your approaches).

As for your report for the competition, everyone should submit a pdf on Canvas detailing what they tried, why they tried it, what they think worked well and why, what they think didn't work well and why, as well as anything you think which makes your solution particularly creative. Please also include your kaggle username, so we know which score belongs to you, along with your public and private leaderboard score. The report will be due after the competition closes so you can see how you ended up.

Reading the data

The first step I took when starting the Kaggle competition was to load the data and take a look at what the input and outputs looked like. I wrote some helper methods to load the data, and to return X_{train}, Y_{train} and X_{test} and the id of the test points. I also wrote a helper method to plot a histogram for each feature in the input, as well as the output,

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib

import matplotlib.pyplot as plt
from scipy.stats import skew
from scipy.stats.stats import pearsonr

%config InlineBackend.figure_format = 'png' #set 'png' here when working on notebook
%matplotlib inline
matplotlib.rcParams['figure.figsize'] = (12.0, 6.0)
```

In [2]:
```
def get_data():
    train = pd.read_csv('./input/train_final.csv')
    test = pd.read_csv('./input/test_final.csv')
    return train,test
def get_train_test():
    train,test = get_data()
    x_train= train.loc[:,'f1':'f24']
    y = train.Y
    x_test = test.loc[:,'f1':'f24']
    ids = test.Id
    return x_train,y,x_test,ids

def plot_histogram_df(x):
    features = x.columns
    # plot the features 3 columns wide
    ncols = 3
    nrows = nrows = int(np.ceil(len(features) / (1.0*ncols)))
    fig, axes = plt.subplots(nrows=nrows, ncols=ncols, figsize=(10, 10))
    counter = 0
    for i in range(nrows):
        for j in range(ncols):
            ax = axes[i][j]
            if counter < len(features):
                values = x[features[counter]]
                ax.hist(values)
                leg = ax.legend(loc='upper right')
                leg.draw_frame(False)
            else:
                ax.set_axis_off()
            counter +=1
    plt.tight_layout()
    plt.show()

def get_numerical_features(x):
    numeric_feats = x.dtypes[x.dtypes != "object"].index
    return numeric_feats

def get_binary_cols(df):
    bin_cols = [col for col in df if
                df[col].dropna().value_counts().index.isin([0,1]).all()]
    return bin_cols
```

In [3]:
```
train,test = get_data()
```

In [4]:
```
train.head()
```

Out[4]:

	Id	Y	f1	f2	f3	f4	f5	f6	f7	f8	...	f15	f16	f17	f18	f19	f20	f21	f22	f23	f24
0	1	1	25884	1	33.63	118596	1	0	118595	125738	...	1945	118450	119184	1	121372	1	1	1	2	1
1	2	1	34346	1	10.62	118041	1	0	117902	130913	...	15385	117945	292795	1	259173	1	1	1	1	1
2	3	1	34923	1	1.77	118327	1	0	117961	124402	...	7547	118933	290919	1	118784	1	1	1	1	1
3	4	1	80926	1	30.09	118300	1	0	117961	301218	...	4933	118458	118331	1	307024	1	1	1	2	1
4	5	1	4674	1	1.77	119921	1	0	119920	302830	...	13836	142145	4673	1	128230	1	1	1	620	1

5 rows × 26 columns

In [5]:
```
train.shape
```

Out[5]:
```
(16383, 26)
```

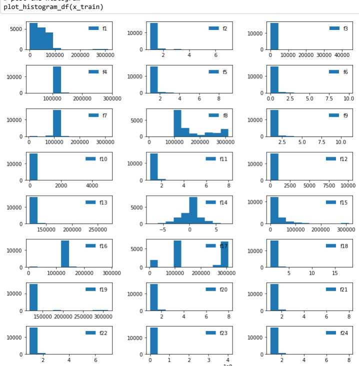
In [3]:
```
x_train,y,x_test,ids = get_train_test()
x_train.shape
```

Out[3]:
```
(16383, 24)
```

I found that the input data has 24 features labeled $f1$ to $f24$, and there are 16383 points in the training data

Now I'll plot the histogram for each of the features

In [10]:
```
# plot the histogram
plot_histogram_df(x_train)
```



In [28]:
```
numeric_feats = get_numerical_features(x_train)
print(numeric_feats)
```

```
Index(['f1', 'f2', 'f3', 'f4', 'f5', 'f6', 'f7', 'f8', 'f9', 'f10', 'f11',
       'f12', 'f13', 'f14', 'f15', 'f16', 'f17', 'f18', 'f19', 'f20', 'f21',
       'f22', 'f23', 'f24'],
      dtype='object')
```

In [37]:
```
bin_cols = get_binary_cols(x_train)
print(bin_cols)
```

```
[]
```

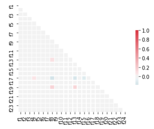
Except for feature f14, none of the features look Gaussian. Several are skewed left, such as $f1$ and $f15$, are skewed left, and it also looks like there are datapoints that are very far from most of the datapoints. Maybe I'll need to remove some outliers. On the positive side, all of the data is numerical, so I don't need to worry about transforming any categorical feature. There are also no other binary columns apart from the output label.

I won't do that immediately next.

I'll continue analyzing the data by visualizing the correlation matrix of the 24 features.

In [11]:
```
# plot the correlation of the features
x_corr = x_train.corr()
# Generate a mask for the upper triangle
mask = np.zeros_like(x_corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
# Set up the matplotlib figure
f, ax = plt.subplots();

# Generate a custom diverging colorma# Generate a custom diverging colormap
cmap = sns.diverging_palette(220, 10, as_cmap=True)
# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(x_corr, mask=mask, cmap=cmap, vmax=1.0, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})
```

Out[11]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f98552c30f0>
```

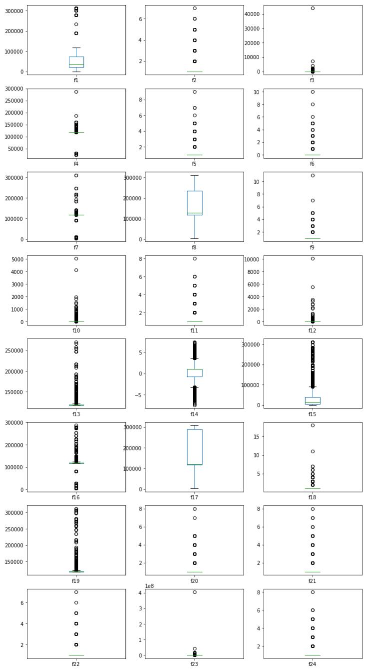


The correlation matrix shows most of the features are independent from one another. $f8$ and $f13$ and $f13$ and $f19$ show some correlation between the 2 features. I'll do some more analysis of the correlation later.

Earlier, the histograms showed there might be outliers in the dataset, so I'll plot a box and whiskers plot for each of the features to get a better visualization.

In [16]:
```
# box and whiskers plot
plt.tight_layout()
x_train.plot(kind='box', subplots=True, layout=(8,3), sharex=False, sharey=False,figsize=(12,24));
```

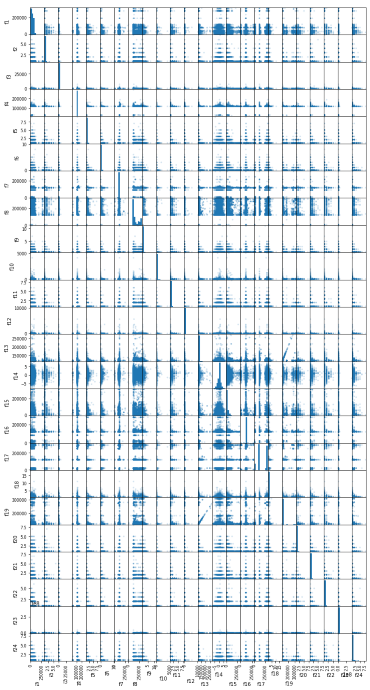
```
<Figure size 432x288 with 0 Axes>
```



The box plots show there are datapoints whose feature values are so far outside of the interquartile range that the plot has had to scale the IQR range to a single line. I don't have a metric for how these outliers affect the performace of my model, but I suspect the accuracy will decrease and that I should either remove the outliers from the dataset, or replace the value of an outlier with another closer to the distribution of the other points. This is something I will do in the preprocessing of the data.

Lastly, I want to graph a scatterplot for each pair of features. The correlation plot showed some correlation between 2 pair of the features, so I want to view it graphically

In [22]:
```
from pandas.plotting import scatter_matrix
scatter_matrix(x_train,figsize=(12,24),alpha=.2)
plt.show()
```

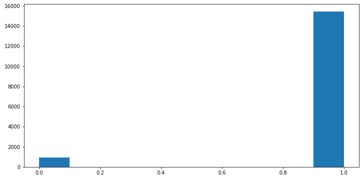


Excluding a few datapoints, the plots show what looks like linear relationships between the $f13, f19$ and $f8, f19$. In preprocessing, I will look into fitting a linear line to replace the 2 features with.

Lastly, I want to plot a histogram of the label.

In [111]:
```
plt.hist(y)
```

Out[111]:
```
(array([  948.,     0.,     0.,     0.,     0.,     0.,     0.,     0.,
            0., 15435.]),
 array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ]),
 <a list of 10 Patch objects>)
```



It looks like the y variable is binary, so I can conclude this is a classification problem with 2 classes.

Data Preprocessing

After visualizing the data, I thought of a few transformations that would be a good idea to try prior to training a model on the data:

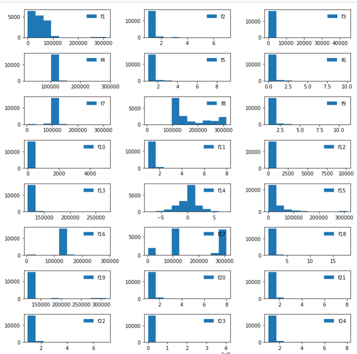
- fill any NA's with the mean of the feature. This should be simple to do especially since all of the data is numerical.
- take the log of features that have a very positive skew (skewed right), and the exponent of features that have a very negative skew (skewed left)
- Fit data to a Gaussian by subtracting the mean and dividing by the standard deviation. $x_i \rightarrow \frac{x_i - \mu}{\sigma}$
- Standardize the data by rescaling values to between 0 and 1. $x_i \rightarrow \frac{x_i - min(x)}{max(x) - min(x)}$
- Replace outliers with a passed in value
- Remove points with outliers from the dataset.

For both of these last two, I'll consider a point an outlier if any of its feature values is more than a threshold number of standard deviations aways from the mean.

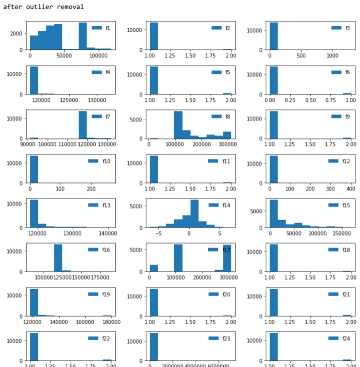
In [69]:
```
def get_numerical_features(x):
    numeric_feats = x.dtypes[x.dtypes != "object"].index
    return numeric_feats
# take the log of skewed features in the testing and training data
def log_norm_skewd_feats(skew_factor,x_train,x_test):
    numeric_feats = get_numerical_features(x_train)
    skewedness_factor = x_train[numeric_feats].apply(lambda x: skew(x.dropna())) #compute skewness
    skewed_feats = skewedness_factor[skewedness_factor > skew_factor].index
    #f1 has a negative value, don't log transform f1
    skewed_feats = skewed_feats.delete(0)
    # take the log of skewed features
    x_train[skewed_feats] = np.log1p(x_train[skewed_feats])
    x_test[skewed_feats] = np.log1p(x_test[skewed_feats])
    return x_train,x_test

# take the log of positive skewed features, and exp of negatively skewed
def log_exp_tranform_features(threshold,x_train,x_test):
    numeric_feats = get_numerical_features(x_train)
    skewedness_factor = x_train[numeric_feats].apply(lambda x: skew(x.dropna())) #compute skewness
    pos_skewed_feats = skewedness_factor[skewedness_factor > threshold].index
    neg_skewed_feats = skewedness_factor[skewedness_factor < -1.0 * threshold].index
    #f1 has a negative value, don't log transform f1
    pos_skewed_feats = pos_skewed_feats.delete(0)
    # take the log of positively_skewed feats
    x_train[pos_skewed_feats] = np.log1p(x_train[pos_skewed_feats])
    x_test[pos_skewed_feats] = np.log1p(x_test[pos_skewed_feats])
    # take the exponent of negatively skewed feats
    x_train[neg_skewed_feats] = np.exp(x_train[neg_skewed_feats])
    x_test[neg_skewed_feats] = np.exp(x_test[neg_skewed_feats])
    return x_train,x_test

def fill_nas_with_mean(x):
    x = x.fillna(x.mean())
    return x

def get_binary_cols(df):
    bin_cols = [col for col in df if
               df[col].dropna().value_counts().index.isin([0,1]).all()]
    return bin_cols

def gauss_norm_data(x_train,x_test):
    mu = np.mean(x_train,axis = 0)
    sigma = np.std(x_train,axis=0)
    x_train = (x_train - mu)/ sigma
    x_test = (x_test - mu) / sigma
    return x_train,x_test

# standardize variables to between 0 and 1
def standardize(x_train,x_test):
    minValue = np.min(x_train,axis = 0)
    maxValue = np.max(x_train,axis = 0)
    x_train = (x_train - minValue) / (maxValue - minValue)
    x_test = (x_test - minValue) / (maxValue - minValue)
    return x_train,x_test


def replace_outliers(x_norm,threshold, toReplaceWith):
    # replace any data more than the threshold number of standard deviations away
    x_norm = x_norm.apply(lambda x:[y if y < threshold else toReplaceWith for y in x])
    x_norm = x_norm.apply(lambda x:[y if y > -1.0 * threshold else (-1.0 *toReplaceWith) for y in x]);
    return x_norm

# remove the row if it contains a value that's too many standard deviations from the mean
def remove_outliers(x_train,y_train,x_train_norm,thresh):
    x_train = x_train[(abs(x_train_norm) < thresh).all(axis=1)]
    y_train = y_train[(abs(x_train_norm) < thresh).all(axis=1)]
    x_train_norm = x_train_norm[(abs(x_train_norm) < thresh).all(axis=1)]
    return x_train,y_train,x_train_norm
```

I'll go ahead and Gaussian normalize the input data, and remove outliers.

In [57]:
```
x_train,y,x_test,ids = get_train_test()
x_train_norm,x_test_norm = gauss_norm_data(x_train,x_test)
plot_histogram_df(x_train)
print("after outlier removal")
x_train_no,y_train_no = remove_outliers(x_train,y_train,x_train_norm,4)
plot_histogram_df(x_train_no)
```



```
after outlier removal
```



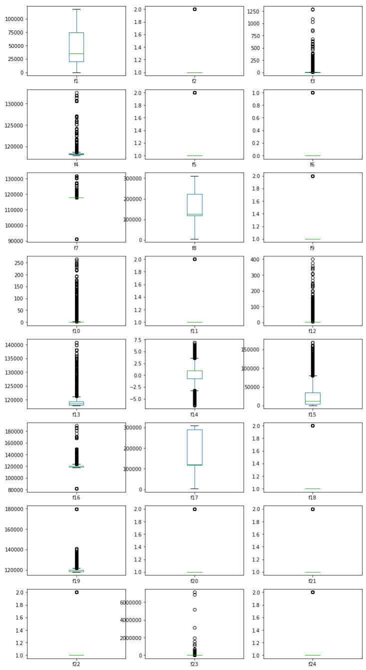
In [59]:
```
print(x_train_no.shape)
print(y_train_no.shape)
```

```
(14050, 24)
(14050,)
```

After Gaussian normalization, some features such as f1 and f17 don't have as long tails. The number of datapoints is also reduced to 14050.

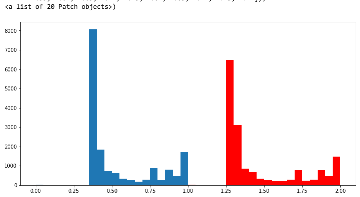
In [54]:
```
plt.tight_layout()
x_train_no.plot(kind='box', subplots=True, layout=(8,3), sharex=False, sharey=False,figsize=(12,24));
```

```
<Figure size 432x288 with 0 Axes>
```

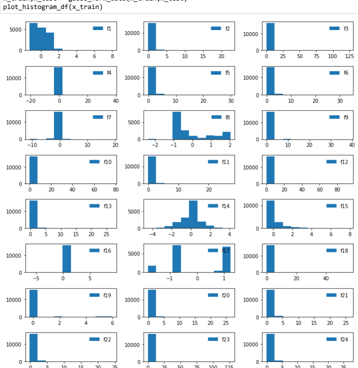


In [226]:
```
x_train,y,x_test,ids = get_train_test()
x_train,x_test = standardize(x_train,x_test)
plt.hist(x_train.f8,bins=20)
plt.hist(np.exp2(x_train.f8),bins=20,color='r')
```

Out[226]:
```
(array([9.000e+00, 0.000e+00, 1.000e+00, 0.000e+00, 0.000e+00, 6.494e+03,
        3.118e+03, 8.450e+02, 6.780e+02, 3.390e+02, 2.600e+02, 1.980e+02,
        2.090e+02, 2.760e+02, 7.680e+02, 2.280e+02, 2.690e+02, 7.780e+02,
        4.520e+02, 1.461e+03]),
 array([1.  , 1.05, 1.1 , 1.15, 1.2 , 1.25, 1.3 , 1.35, 1.4 , 1.45, 1.5 ,
        1.55, 1.6 , 1.65, 1.7 , 1.75, 1.8 , 1.85, 1.9 , 1.95, 2.  ]),
 <a list of 20 Patch objects>)
```



In [195]:
```
x_train,y,x_test,ids = get_train_test()
x_train = fill_nas_with_mean(x_train)
x_train,x_test = gauss_norm_data(x_train,x_test)
plot_histogram_df(x_train)
```



Models

Now that I've spent some time analyzing the data, I will proceed with fitting a model to the data. Since this is a classification problem, I will try to fit a Logistic Regression model.

Logistic Regresssion

I'll first try fitting a logistic regression classifier with no pre-processing on the data. I'll use SkLearn's Cross Validation to optimize the parameter in logistic regression.

In [15]:
```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression,LogisticRegressionCV
from sklearn import metrics
import datetime
```

Just a couple helper functions to compute the auc score from the testing data and to write the predictions into a file for submission.

In [6]:
```
# function to compute the auc score
def auc_score(y_pred_proba,y):
    return metrics.roc_auc_score(y,y_pred_proba)
def write_predictions(filename,header,ids,y_pred):
    f = open(filename,'w')
    numRows = len(ids)
    f.write(header)
    for i in range(numRows):
        idNum = ids[i]
        y = y_pred[i]
        f.write("{},{}\n".format(idNum,y))
    f.close()
def get_prediction_proba(model,x_train,x_test):
    y_train_proba = model.predict_proba(x_train)[:,1]
    y_test_proba = model.predict_proba(x_test)[:,1]
    return y_train_proba,y_test_proba
```

I'll go ahead and fit the logistic regression to the model. I will let the parameter \alpha range from 10^{-5} to 10^{5} and use cross validation to find the optimal value

In [46]:
```
x_train,y,x_test,ids = get_train_test()
n_alphas = 200
alphas = np.logspace(-5, 5, n_alphas).tolist()
model_logistic = LogisticRegressionCV(Cs=alphas,cv=10,solver='liblinear').fit(x_train,y)
```

Now get the predictions for the test data using the model.

In [52]:
```
y_train_proba,y_test_proba = get_prediction_proba(model_logistic,x_train,x_test)
```

I'll find the auc score for the training data to have a metric of how good my model is fitting to the training data

```
In [53]: auc_score(y_train_proba,y)
Out[53]: 0.5325772704098718
```

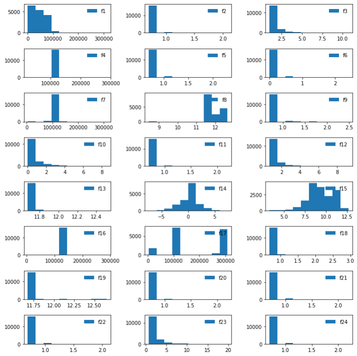
```
In [70]: write_predictions("predictions/logistic_regression_"+datetime.datetime.now().isoformat() + ".csv","Id,Y\n",ids,y_test_proba)
```

With no pre processing, the auc score was 0.5325772704098718.

Next I want to try some preprocessing on the training data. In a previous assignment, we saw an example of log normalizing features that are skewed helped in a regression problem.

I'll go ahead and log normalize features with a skew greater than .75. I will also fill in any NA's in the data with the mean.

```
In [74]: x_train,y,x_test,ids = get_train_test()
         x_train = fill_nas_with_mean(x_train)
         x_train,x_test= log_norm_skewd_feats(.75,x_train,x_test)
         plot_histogram_df(x_train)
```



Now we can train the classifier on the log normalized data

```
In [63]: model_logistic_preproc = LogisticRegressionCV(Cs=alphas,cv=10,solver='liblinear').fit(x_train,y)
```

get the predictions from this logistic regression

```
In [66]: y_train_preproc_proba,y_test_preproc_proba = get_prediction_proba(model_logistic_preproc,x_train,x_test)
```

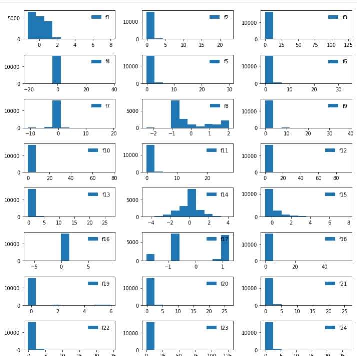
```
In [67]: auc_score(y_train_preproc_proba,y)
Out[67]: 0.5048381056260157
```

```
In [76]: write_predictions("predictions/logistic_regression_preproc_"+datetime.datetime.now().isoformat() + ".csv","Id,Y\n",ids,y_test_preproc_proba)
```

The auc score actually got worse with the log transformation. Since it looks like taking a log is a bad idea for logistic regression, I suspect that the values in the tails of the distributions are statistically significant, and not errors or bad measurements.

I want to try transforming the data by normalizing to a Gaussian to confirm this.

```
In [123]: x_train,y,x_test,ids = get_train_test()
          x_train = fill_nas_with_mean(x_train)
          x_train,x_test = gauss_norm_data(x_train,x_test)
          plot_histogram_df(x_train)
```




```
In [78]: model_logistic_norm = LogisticRegressionCV(Cs=alphas,cv=10,solver='liblinear').fit(x_train,y)
```

```
In [83]: y_train_pred_norm,y_test_pred_norm = get_prediction_proba(model_logistic_norm,x_train,x_test)
```

```
In [84]: auc_score(y_train_pred_norm,y)
Out[84]: 0.5702679946802913
```

The auc score did improve after normalizing to 0.5702679946802913. This is a marginal improvement, and is not much better than randomly guessing the label for the testing data (auc_score of 1/2). I still suspect that transformations to address the long distribution tails of the features will not help for this dataset.

Next I could try removing or replacing outliers prior to training the model, but I don't have confidence that any logistic regression can fit the data without doing some drastic pre processing that would remove many of the training points.

```
In [85]: write_predictions("predictions/logistic_regression_norm_"+datetime.datetime.now().isoformat() + ".csv","Id,Y\n",ids,y_test_pred_norm)
```

XGBoost

Instead I'll try using a different model buy fitting XGBoost Classifier to the data.

```
In [5]: from numpy import loadtxt
        import xgboost as xgb
        from xgboost import XGBClassifier
        from sklearn.model_selection import train_test_split
        from sklearn.metrics import accuracy_score
```

I will first fit a vanilla xgbooster with no parameter tuning.

```
In [38]: x_train,y,x_test,ids = get_train_test()
         model = XGBClassifier()
         model.fit(x_train,y)
Out[38]: XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
                colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
                max_depth=3, min_child_weight=1, missing=None, n_estimators=100,
                n_jobs=1, nthread=None, objective='binary:logistic', random_state=0,
                reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
                silent=True, subsample=1)
```

```
In [42]: y_train_xgboost,y_test_xgboost = get_prediction_proba(model,x_train,x_test)
```

```
In [43]: auc_score(y_train_xgboost,y)
Out[43]: 0.9057251451916913
```

```
In [92]: write_predictions("predictions/xgboost_"+datetime.datetime.now().isoformat() + ".csv","Id,Y\n",ids,y_test_pred_norm)
```

Xgboost scored much better than logistic regression. This gives me confidence that it's the correct path to take rather than trying to optimize a logistic regression model to it.

I'll try xgboost with gaussian normalization preprocessing of the data.

```
In [48]: x_train,y,x_test,ids = get_train_test()
         x_train = fill_nas_with_mean(x_train)
         x_train,x_test = gauss_norm_data(x_train,x_test)
         model_xg_boost_norm = XGBClassifier()
         model_xg_boost_norm.fit(x_train,y)
Out[48]: XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
                colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
                max_depth=3, min_child_weight=1, missing=None, n_estimators=100,
                n_jobs=1, nthread=None, objective='binary:logistic', random_state=0,
                reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
                silent=True, subsample=1)
```

```
In [49]: y_train_norm_xgboost,y_test_norm_xgboost = get_prediction_proba(model_xg_boost_norm,x_train,x_test)
```

```
In [50]: auc_score(y_train_norm_xgboost,y)
Out[50]: 0.9057251451916913
```

Normalizing to a Gaussian had not improvement in the auc score. I'm going to put off any preprocessing other than filling NA's with the mean will try parameter tuning.

```
In [99]: write_predictions("predictions/xgboost_norm_"+datetime.datetime.now().isoformat() + ".csv","Id,Y\n",ids,y_test_norm_xgboost)
```

I'll first try tuning some parameters ad hoc.

In [51]:
```
x_train,y,x_test,ids = get_train_test()
x_train = fill_nas_with_mean(x_train)
clf = xgb.XGBClassifier(
                max_depth = 7,
                n_estimators=700,
                learning_rate=0.1,
                nthread=4,
                subsample=1.0,
                colsample_bytree=0.5,
                min_child_weight = 3,
                seed=1301,
                objective='binary:logistic',
                eval_metric='auc')
clf.fit(x_train,y)
```

Out[51]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=0.5, eval_metric='auc', gamma=0, learning_rate=0.1,
       max_delta_step=0, max_depth=7, min_child_weight=3, missing=None,
       n_estimators=700, n_jobs=1, nthread=4, objective='binary:logistic',
       random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
       seed=1301, silent=True, subsample=1.0)
```

In [52]:
```
y_train,y_test = get_prediction_proba(clf,x_train,x_test)
```

In [53]:
```
auc_score(y_train,y)
```

Out[53]:
```
0.9999991799010141
```

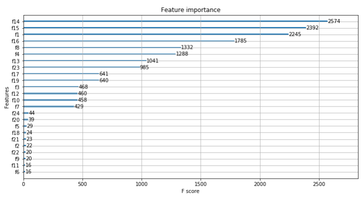
In [239]:
```
write_predictions("predictions/xgboost_"+datetime.datetime.now().isoformat() + ".csv","Id,Y\n",ids,y_test)
```

This prediction had a score of 0.88843 (high enough to get me into second place at the time)! Nice!

After reading through the XgBoost documentation, I noticed that one of the metrics it provides upon training a model is a feature importance. I wonder if some features are more important than others and would allow to throw out some features from the training and test data

In [217]:
```
xgb.plot_importance(clf)
```

Out[217]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8f8704c0f0>
```



In [172]:
```
np.sort(clf.feature_importances_)
```

Out[172]:
```
array([0.00123218, 0.00129085, 0.00134953, 0.0014082 , 0.0014082 ,
       0.0014082 , 0.00158423, 0.00176025, 0.0021123 , 0.00217098,
       0.0249956 , 0.02792935, 0.02845743, 0.02845743, 0.03620255,
       0.03978173, 0.05668016, 0.06084609, 0.07199436, 0.08249721,
       0.10379628, 0.13260576, 0.14117233, 0.14885877], dtype=float32)
```

It does look like some features are much more important than others. The feature importance plot shows 10 are at least an order of magnitude less important than others, so I'll throw them out

In [173]:
```
from sklearn.feature_selection import SelectFromModel
f_score_thresh = 0.01
selection = SelectFromModel(clf,threshold=f_score_thresh,prefit=True)
selection_x_train = selection.transform(x_train)
selection_x_train.shape
```

Out[173]:
```
(16383, 14)
```

The data now only has 14 columns. Now train a model with the selected features

In [174]:
```
clf_selected = xgb.XGBClassifier(
                max_depth = 7,
                n_estimators=700,
                learning_rate=0.1,
                nthread=4,
                subsample=1.0,
                colsample_bytree=0.5,
                min_child_weight = 3,
                seed=1301,
                objective='binary:logistic',
                eval_metric='auc')
```

In [175]:
```
clf_selected.fit(selection_x_train,y)
```

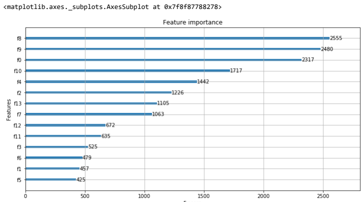
Out[175]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=0.5, eval_metric='auc', gamma=0, learning_rate=0.1,
       max_delta_step=0, max_depth=7, min_child_weight=3, missing=None,
       n_estimators=700, n_jobs=1, nthread=4, objective='binary:logistic',
       random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
       seed=1301, silent=True, subsample=1.0)
```

In [176]:
```
selection_x_test = selection.transform(x_test)
y_train_selected,y_test = get_prediction_proba(clf_selected,selection_x_train,selection_x_test)
auc_score(y_train_selected,y)
```

Out[176]:
```
0.9999977447277887
```

In [177]:
```
xgb.plot_importance(clf_selected)
```

Out[177]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8f87788278>
```



In [87]:
```
write_predictions("predictions/xgboost_selected_"+datetime.datetime.now().isoformat() + ".csv","Id,Y\n",ids,y_test)
```

This had a score of .88643 which is not an improvement over the vanilla xgboost fit. I believe what is going on here is that while some feature are more significant in the predicted label, the xgboost is able to fit the remaining features and they end up improving the accuracy. I won't try throwing out features again.

I'm still suspecting the points in the tails are important, so I want to test removing them (I'll consider them to be outliers) and see how the score changes.

In [79]:
```
x_train,y,x_test,ids = get_train_test()
x_train = fill_nas_with_mean(x_train)
x_train,x_test= log_norm_skewd_feats(.75,x_train,x_test)
x_train_norm,x_test_norm = gauss_norm_data(x_train,x_test)
x_train_no,y_train_no,x_train_norm_no = remove_outliers(x_train,y,x_train_norm,)
clf_no = xgb.XGBClassifier(
                max_depth = 7,
                n_estimators=700,
                learning_rate=0.1,
                nthread=4,
                subsample=1.0,
                colsample_bytree=0.5,
                min_child_weight = 3,
                seed=1301,
                objective='binary:logistic',
                eval_metric='auc')
clf_no.fit(x_train_no,y_train_no)
```

Out[79]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=0.5, eval_metric='auc', gamma=0, learning_rate=0.1,
       max_delta_step=0, max_depth=7, min_child_weight=3, missing=None,
       n_estimators=700, n_jobs=1, nthread=4, objective='binary:logistic',
       random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
       seed=1301, silent=True, subsample=1.0)
```

In [81]:
```
y_train,y_test = get_prediction_proba(clf_no,x_train_no,x_test)
auc_score(y_train,y_train_no)
```

Out[81]:
```
0.9999998254862803
```

In [82]:
```
write_predictions("predictions/xgboost_no_outliers_"+datetime.datetime.now().isoformat() + ".csv","Id,Y\n",ids,y_test)
```

Hmm the auc score (.87) did not increase by removing outliers. I now conclude that the points very far form the mean are actually important points. I will not try to remove them or replace them.

I have tried several pre processing transformations and none of them have produced some significant improvements. I'm going to put that off for now and instead of doing some preprocessing this time, I'll focus on optimizing the parameters of xgboost systematically using GridSearchCV instead of ad hoc as I did previously. I followed the steps from https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/ (https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/).

General Approach for Parameter Tuning
We will use an approach similar to that of GBM here. The various steps to be performed are:

Choose a relatively high learning rate. Generally a learning rate of 0.1 works but somewhere between 0.05 to 0.3 should work for different problems. Determine the optimum number of trees for this learning rate. XGBoost has a very useful function called as "cv" which performs cross-validation at each boosting iteration and thus returns the optimum number of trees required. Tune tree-specific parameters (max_depth, min_child_weight, gamma, subsample, colsample_bytree) for decided learning rate and number of trees. Note that we can choose different parameters to define a tree and I'll take up an example here. Tune regularization parameters (lambda, alpha) for xgboost which can help reduce model complexity and enhance performance. Lower the learning rate and decide the optimal parameters . Let us look at a more detailed step by step approach.

In [11]:
```
from sklearn import metrics
def modelfit(alg, x_train, y_train,useTrainCV=True, cv_folds=5, early_stopping_rounds=50):

    if useTrainCV:
        xgb_param = alg.get_xgb_params()
        xgtrain = xgb.DMatrix(x_train.values, label=y_train)
        cvresult = xgb.cv(xgb_param, xgtrain, num_boost_round=alg.get_params()['n_estimators'], nfold=cv_folds,
            metrics='auc', early_stopping_rounds=early_stopping_rounds)
        alg.set_params(n_estimators=cvresult.shape[0])

    #Fit the algorithm on the data
    alg.fit(x_train, y_train,eval_metric='auc')

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

    #Print model report:
    print("\nModel Report")
    print("Accuracy : {}".format(metrics.accuracy_score(y_train, dtrain_predictions)))
    print("AUC Score (Train): {}".format(metrics.roc_auc_score(y_train, dtrain_predprob)))
```

Step 1: Fix learning rate and number of estimators for tuning tree-based parameters.

In order to decide on boosting parameters, we need to set some initial values of other parameters. Lets take the following values:

- max_depth = 5 : This should be between 3-10. I've started with 5 but you can choose a different number as well. 4-6 can be good starting points.
- min_child_weight = 1 : A smaller value is chosen because it is a highly imbalanced class problem and leaf nodes can have smaller size groups.
- gamma = 0 : A smaller value like 0.1-0.2 can also be chosen for starting. This will anyways be tuned later.
- subsample, colsample_bytree = 0.8 : This is a commonly used used start value. Typical values range between 0.5-0.9.
- scale_pos_weight = 1: Because of high class imbalance.

In [108]:
```
from sklearn.model_selection import GridSearchCV   #Performing grid search
learning_rate=.1
n_estimators=1000
max_depth=5
min_child_weight =1
gamma = 0
subsample = 0.8
colsample_bytree = 0.8
scale_pos_weight = 1
objective = 'binary:logistic'
metric = 'auc'
seed = 1301
xgb1 = XGBClassifier(learning_rate=learning_rate,
                     n_estimators=n_estimators,
                     max_depth=max_depth,
                     min_child_weight=min_child_weight,
                     gamma = gamma,
                     subsample = subsample,
                     colsample_bytree = colsample_bytree,
                     objective = objective,
                     eval_metric = metric,
                     nthread=4,
                     scale_pos_weight = scale_pos_weight)
```

In [109]:
```
x_train,y,x_test,ids = get_train_test()
x_train = fill_nas_with_mean(x_train)
modelfit(xgb1,x_train,y)
```

```
Model Report
Accuracy : 0.9822987242873711
AUC Score (Train): 0.9983787326463638
```

In [113]:
```
n_estimators = xgb1.n_estimators
print("The optimum number of estimators is : {}".format(n_estimators))
```

```
The optimum number of estimators is : 356
```

we got 356 as the optimum number of estimators

Step 2: Tune max_depth and min_child_weight

We tune these first as they will have the highest impact on model outcome. To start with, let's set wider ranges and then we will perform another iteration for smaller ranges.

In [114]:
```
param_test1 = {
 'max_depth':range(3,10,2),
 'min_child_weight':range(1,6,2)
}
```

In [126]:
```
gsearch1 = GridSearchCV(
    estimator = XGBClassifier(
        learning_rate =learning_rate,
        n_estimators=n_estimators,
        max_depth=max_depth,
        min_child_weight=min_child_weight,
        gamma=gamma,
        subsample=subsample,
        colsample_bytree=colsample_bytree,
        objective= objective,
        nthread=4,
        scale_pos_weight=scale_pos_weight,
        seed=1301),
    param_grid = param_test1,
    scoring='roc_auc',
    n_jobs=4,iid=False,
    cv=5)
gsearch1.fit(x_train,y)
gsearch1.cv_results_, gsearch1.best_params_, gsearch1.best_score_
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-126-ceca270e4335> in <module>
     17     cv=5)
     18 gsearch1.fit(x_train,y)
---> 19 gsearch1.grid_scores_, gsearch1.best_params_, gsearch1.best_score_

AttributeError: 'GridSearchCV' object has no attribute 'grid_scores_'
```

In [127]:
```
gsearch1.best_params_
```

Out[127]:
```
{'max_depth': 7, 'min_child_weight': 1}
```

In [128]:
```
gsearch1.best_score_
```

Out[128]:
```
0.8770000136214993
```

In [131]:
```
gsearch1.cv_results_
```

```
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split0_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split1_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split2_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split3_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split4_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('mean_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('std_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
```

Out[131]:
```
{'mean_fit_time': array([200.07258515, 194.57856517, 212.99518743, 333.04188976,
        348.0370245 , 324.64773798, 540.64477139, 520.39580998,
        496.43202696, 702.88177109, 648.7309072 , 411.22267699]),
 'std_fit_time': array([ 28.41301959,  25.49785992,   9.60407223,  51.99864261,
         25.84333151,  18.87359555,  52.17601935,  29.5367285 ,
         40.46794873,  96.57082017,  50.01112376, 141.78229954]),
 'mean_score_time': array([0.10153465, 0.09631677, 0.1053771 , 0.10548792, 0.10340476,
        0.114007  , 0.15683846, 0.15187817, 0.15402689, 0.15734429,
        0.15094571, 0.10790095]),
 'std_score_time': array([0.01225866, 0.00568714, 0.01620792, 0.00851084, 0.00866733,
        0.01731523, 0.02712383, 0.02393402, 0.0077205 , 0.01183384,
        0.00801494, 0.04977948]),
 'param_max_depth': masked_array(data=[3, 3, 3, 5, 5, 5, 7, 7, 7, 9, 9, 9],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False],
        fill_value='?',
             dtype=object),
 'param_min_child_weight': masked_array(data=[1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'max_depth': 3, 'min_child_weight': 1},
  {'max_depth': 3, 'min_child_weight': 3},
  {'max_depth': 3, 'min_child_weight': 5},
  {'max_depth': 5, 'min_child_weight': 1},
  {'max_depth': 5, 'min_child_weight': 3},
  {'max_depth': 5, 'min_child_weight': 5},
  {'max_depth': 7, 'min_child_weight': 1},
  {'max_depth': 7, 'min_child_weight': 3},
  {'max_depth': 7, 'min_child_weight': 5},
  {'max_depth': 9, 'min_child_weight': 1},
  {'max_depth': 9, 'min_child_weight': 3},
  {'max_depth': 9, 'min_child_weight': 5}],
 'split0_test_score': array([0.84393467, 0.84654493, 0.83633062, 0.86903739, 0.86259697,
        0.86061242, 0.86906808, 0.85919987, 0.86110344, 0.85854688,
        0.86279133, 0.86156036]),
 'split1_test_score': array([0.8501219 , 0.85276286, 0.84686887, 0.86435732, 0.86020323,
        0.85044414, 0.86312721, 0.86478185, 0.85231787, 0.86259015,
        0.85475423, 0.85832694]),
 'split2_test_score': array([0.88010843, 0.87328866, 0.87839667, 0.88277496, 0.89168329,
        0.88423269, 0.8806668 , 0.88369222, 0.88160537, 0.88243398,
        0.8814059 , 0.88016044]),
 'split3_test_score': array([0.86910289, 0.86224533, 0.86844216, 0.87495951, 0.87269365,
        0.86693302, 0.87203206, 0.86876524, 0.8698459 , 0.87135761,
        0.87399112, 0.8657221 ]),
 'split4_test_score': array([0.87367918, 0.86958452, 0.87347093, 0.88945964, 0.8887792 ,
        0.88692811, 0.90010592, 0.89563419, 0.88782109, 0.89069969,
        0.89380882, 0.89138956]),
 'mean_test_score': array([0.86338942, 0.86088526, 0.86070185, 0.87611776, 0.87519127,
        0.86983007, 0.87700001, 0.87441468, 0.87053873, 0.87312566,
        0.87335028, 0.87143188]),
 'std_test_score': array([0.01394678, 0.01002985, 0.01625655, 0.0090786 , 0.01300857,
        0.01392086, 0.01286374, 0.01336068, 0.01298195, 0.01201795,
        0.01371977, 0.01246021]),
 'rank_test_score': array([10, 11, 12,  2,  3,  9,  1,  4,  8,  6,  5,  7], dtype=int32),
 'split0_train_score': array([0.97571798, 0.97218397, 0.96780305, 0.99948962, 0.99818767,
        0.99595055, 1.        , 0.99999509, 0.99991816, 1.        ,
        1.        , 0.99999989]),
 'split1_train_score': array([0.9776037 , 0.97372092, 0.9681154 , 0.9994002 , 0.99827688,
        0.99626434, 1.        , 0.99999209, 0.99989231, 1.        ,
        1.        , 0.99999936]),
 'split2_train_score': array([0.97609363, 0.97204524, 0.96796646, 0.99930501, 0.99795807,
        0.99595546, 1.        , 0.99998846, 0.99976538, 1.        ,
        1.        , 0.99999541]),
 'split3_train_score': array([0.97481363, 0.96930352, 0.96552828, 0.99939918, 0.99799597,
        0.99646569, 1.        , 0.99999093, 0.99988626, 1.        ,
        1.        , 0.99999883]),
 'split4_train_score': array([0.97581719, 0.97074193, 0.96548742, 0.99926932, 0.99776508,
        0.99611716, 1.        , 0.99998826, 0.99986492, 1.        ,
        1.        , 0.99999872]),
 'mean_train_score': array([0.97600922, 0.97159912, 0.96698012, 0.99937267, 0.99803674,
        0.99615064, 1.        , 0.99999097, 0.9998654 , 1.        ,
        1.        , 0.99999844]),
 'std_train_score': array([9.05609158e-04, 1.48645165e-03, 1.20622884e-03, 7.79718124e-05,
        1.80056033e-04, 1.95671739e-04, 0.00000000e+00, 2.52276920e-06,
        5.28148301e-05, 0.00000000e+00, 0.00000000e+00, 1.57413308e-06])}
```

Ideal values for max_depth is 7, and min_child_weight is 1. Lets go one step deeper and look for optimum values. We'll search for values 1 above and below the optimum values because we took an interval of two.

In [132]:
```
param_test2 = {
 'max_depth':[6,7,8],
 'min_child_weight':[0,1,2]
}
```

In [134]:
```
gsearch2 = GridSearchCV(
    estimator = XGBClassifier(
        learning_rate =learning_rate,
        n_estimators=n_estimators,
        max_depth=max_depth,
        min_child_weight=min_child_weight,
        gamma=gamma,
        subsample=subsample,
        colsample_bytree=colsample_bytree,
        objective= objective,
        nthread=4,
        scale_pos_weight=scale_pos_weight,
        seed=1301),
    param_grid = param_test2,
    scoring='roc_auc',
    n_jobs=4,iid=False,
    cv=5)
gsearch2.fit(x_train,y)
gsearch2.cv_results_, gsearch2.best_params_, gsearch2.best_score_
```

```
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split0_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split1_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split2_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split3_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split4_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('mean_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('std_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
```

Out[134]:
```
({'mean_fit_time': array([531.48034301, 435.86001711, 415.94839344, 668.36858125,
         543.00041637, 519.27428684, 766.00890255, 682.34646034,
         483.72568927]),
  'std_fit_time': array([ 57.21660473,  74.00972404,  25.43939957,  80.01859947,
          67.78905437,  16.53557169,  89.53520843,  29.54519558,
         105.07059095]),
  'mean_score_time': array([0.13058863, 0.13652391, 0.12357507, 0.1475749 , 0.13912387,
         0.13284802, 0.16092081, 0.16474466, 0.11024508]),
  'std_score_time': array([0.01185207, 0.01603448, 0.02001617, 0.00937002, 0.01967156,
         0.00975611, 0.00953604, 0.02832361, 0.05000259]),
  'param_max_depth': masked_array(data=[6, 6, 6, 7, 7, 7, 8, 8, 8],
               mask=[False, False, False, False, False, False, False, False,
                     False],
         fill_value='?',
              dtype=object),
  'param_min_child_weight': masked_array(data=[0, 1, 2, 0, 1, 2, 0, 1, 2],
               mask=[False, False, False, False, False, False, False, False,
                     False],
         fill_value='?',
              dtype=object),
  'params': [{'max_depth': 6, 'min_child_weight': 0},
   {'max_depth': 6, 'min_child_weight': 1},
   {'max_depth': 6, 'min_child_weight': 2},
   {'max_depth': 7, 'min_child_weight': 0},
   {'max_depth': 7, 'min_child_weight': 1},
   {'max_depth': 7, 'min_child_weight': 2},
   {'max_depth': 8, 'min_child_weight': 0},
   {'max_depth': 8, 'min_child_weight': 1},
   {'max_depth': 8, 'min_child_weight': 2}],
  'split0_test_score': array([0.86907064, 0.87129644, 0.86352701, 0.87431333, 0.86906808,
         0.87182753, 0.87113447, 0.86927523, 0.8637819 ]),
  'split1_test_score': array([0.86587898, 0.85904984, 0.8645389 , 0.8673171 , 0.86312721,
         0.85752732, 0.8642968 , 0.86595997, 0.86221336]),
  'split2_test_score': array([0.8885138 , 0.88399911, 0.88697168, 0.88261811, 0.8806668 ,
         0.88431453, 0.88300939, 0.88943959, 0.88215351]),
  'split3_test_score': array([0.87625783, 0.86993845, 0.87279134, 0.86618916, 0.87203206,
         0.8665448 , 0.86924601, 0.87388742, 0.86919973]),
  'split4_test_score': array([0.89399136, 0.89784778, 0.89567104, 0.89261676, 0.90010592,
         0.89485948, 0.89167751, 0.89604383, 0.8959847 ]),
  'mean_test_score': array([0.87874252, 0.87642632, 0.8767    , 0.87661089, 0.87700001,
         0.87501473, 0.87587284, 0.87892121, 0.87466664]),
  'std_test_score': array([0.01089215, 0.01331598, 0.01266574, 0.00992512, 0.01286374,
         0.01317579, 0.01001089, 0.01174903, 0.01275901]),
  'rank_test_score': array([2, 6, 4, 5, 3, 8, 7, 1, 9], dtype=int32),
  'split0_train_score': array([0.99999957, 0.99999338, 0.99993237, 1.        , 1.        ,
         1.        , 1.        , 1.        , 1.        ]),
  'split1_train_score': array([0.99999893, 0.99998344, 0.99990043, 1.        , 1.        ,
         0.99999936, 1.        , 1.        , 1.        ]),
  'split2_train_score': array([0.99999947, 0.99998835, 0.99989829, 1.        , 1.        ,
         0.99999947, 1.        , 1.        , 1.        ]),
  'split3_train_score': array([1.        , 0.99999349, 0.99992435, 1.        , 1.        ,
         0.99999979, 1.        , 1.        , 1.        ]),
  'split4_train_score': array([0.99999968, 0.99998613, 0.99993075, 1.        , 1.        ,
         1.        , 1.        , 1.        , 1.        ]),
  'mean_train_score': array([0.99999953, 0.99998896, 0.99991724, 1.        , 1.        ,
         0.99999972, 1.        , 1.        , 1.        ]),
  'std_train_score': array([3.48538058e-07, 3.97207377e-06, 1.48591263e-05, 0.00000000e+00,
         0.00000000e+00, 2.66899963e-07, 0.00000000e+00, 0.00000000e+00,
         0.00000000e+00])},
 {'max_depth': 8, 'min_child_weight': 1},
 0.8789212070236058)
```

Now I see the optimal max_depth=8, and min_child_weight=1

Step 3: Tune gamma

Now lets tune gamma value using the parameters already tuned above. Gamma can take various values but I'll check for 5 values here. You can go into more precise values as.

In [136]:
```
param_test3 = {
 'gamma':[i/10.0 for i in range(0,5)]
}
```

In [138]:
```
gsearch3 = GridSearchCV(
    estimator = XGBClassifier(
        learning_rate =learning_rate,
        n_estimators=n_estimators,
        max_depth=8,
        min_child_weight=1,
        gamma=0,
        subsample=subsample,
        colsample_bytree=colsample_bytree,
        objective= 'binary:logistic',
        nthread=4, scale_pos_weight=1,
        seed=1301),
param_grid = param_test3, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch3.fit(x_train,y)
gsearch3.cv_results_, gsearch3.best_params_, gsearch3.best_score_
```

```
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split0_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split1_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split2_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split3_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split4_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('mean_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('std_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
```

Out[138]:
```
({'mean_fit_time': array([606.10097127, 642.53603745, 673.11463914, 643.2649085 ,
         571.24515219]),
  'std_fit_time': array([75.25973992, 63.9321816 , 33.26564528, 77.51674357, 96.56353498]),
  'mean_score_time': array([0.15979686, 0.15158815, 0.14495478, 0.14629498, 0.09905863]),
  'std_score_time': array([0.01789434, 0.01344344, 0.01435273, 0.00751126, 0.04508675]),
  'param_gamma': masked_array(data=[0.0, 0.1, 0.2, 0.3, 0.4],
               mask=[False, False, False, False, False],
         fill_value='?',
              dtype=object),
  'params': [{'gamma': 0.0},
   {'gamma': 0.1},
   {'gamma': 0.2},
   {'gamma': 0.3},
   {'gamma': 0.4}],
  'split0_test_score': array([0.86927523, 0.87412579, 0.87224439, 0.87300564, 0.86726084]),
  'split1_test_score': array([0.86595997, 0.86480913, 0.86524815, 0.86228667, 0.86499071]),
  'split2_test_score': array([0.88943959, 0.87935911, 0.88401275, 0.88545599, 0.88380219]),
  'split3_test_score': array([0.87388742, 0.87683287, 0.87351806, 0.87267137, 0.8692263 ]),
  'split4_test_score': array([0.89604383, 0.89567704, 0.89277187, 0.89560677, 0.89359115]),
  'mean_test_score': array([0.87892121, 0.87816079, 0.87755905, 0.87780529, 0.87577424]),
  'std_test_score': array([0.01174903, 0.01004556, 0.00968725, 0.01153817, 0.01107726]),
  'rank_test_score': array([1, 2, 4, 3, 5], dtype=int32),
  'split0_train_score': array([1., 1., 1., 1., 1.]),
  'split1_train_score': array([1., 1., 1., 1., 1.]),
  'split2_train_score': array([1., 1., 1., 1., 1.]),
  'split3_train_score': array([1., 1., 1., 1., 1.]),
  'split4_train_score': array([1., 1., 1., 1., 1.]),
  'mean_train_score': array([1., 1., 1., 1., 1.]),
  'std_train_score': array([0., 0., 0., 0., 0.])},
 {'gamma': 0.0},
 0.8789212070236058)
```

a gamma value of 0 had the best result. Before proceeding, a good idea would be to re-calibrate the number of boosting rounds for the updated parameters.

In [140]:
```
xgb2=XGBClassifier(
        learning_rate =learning_rate,
        n_estimators=1000,
        max_depth=8,
        min_child_weight=1,
        gamma=0,
        subsample=subsample,
        colsample_bytree=colsample_bytree,
        objective= 'binary:logistic',
        nthread=4, scale_pos_weight=1,
        seed=1301)
modelfit(xgb2,x_train,y)
```

```
Model Report
Accuracy : 0.9999389611182323
AUC Score (Train): 1.0
```

In [142]:
```
xgb2.get_params()
```

Out[142]:
```
{'base_score': 0.5,
 'booster': 'gbtree',
 'colsample_bylevel': 1,
 'colsample_bytree': 0.8,
 'gamma': 0,
 'learning_rate': 0.1,
 'max_delta_step': 0,
 'max_depth': 8,
 'min_child_weight': 1,
 'missing': None,
 'n_estimators': 351,
 'n_jobs': 1,
 'nthread': 4,
 'objective': 'binary:logistic',
 'random_state': 0,
 'reg_alpha': 0,
 'reg_lambda': 1,
 'scale_pos_weight': 1,
 'seed': 1301,
 'silent': True,
 'subsample': 0.8}
```

Step 4: Tune subsample and colsample_bytree

In [141]:
```
param_test4 = {
 'subsample':[i/10.0 for i in range(6,10)],
 'colsample_bytree':[i/10.0 for i in range(6,10)]
}
```

In [143]:
```
gsearch4=GridSearchCV(
    estimator = XGBClassifier(
        learning_rate =learning_rate,
        n_estimators=351,
        max_depth=8,
        min_child_weight=1,
        gamma=0,
        subsample=subsample,
        colsample_bytree=colsample_bytree,
        objective= 'binary:logistic',
        nthread=4, scale_pos_weight=1,
        seed=1301),
    param_grid = param_test4, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch4.fit(x_train,y)
gsearch4.cv_results_, gsearch4.best_params_, gsearch4.best_score_
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-143-6d6fd953a44f> in <module>
     13     param_grid = param_test4, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
     14 gsearch4.fit(x_train,y)
---> 15 gsearch4.cv_results_, gsearch4.best_params_, gsearch4.best_score

AttributeError: 'GridSearchCV' object has no attribute 'best_score'
```

In [144]:
```
gsearch4.cv_results_, gsearch4.best_params_, gsearch4.best_score_
```

```
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split0_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split1_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split2_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split3_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split4_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('mean_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('std_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
```

Out[144]:
```
({'mean_fit_time': array([585.42947907, 587.99896846, 596.53379641, 560.23508177,
         610.31188145, 584.2916574 , 621.98956957, 572.93624964,
         662.18081346, 696.33862581, 654.16811805, 595.48113708,
         694.08773088, 678.11965561, 639.58612914, 478.44051347]),
  'std_fit_time': array([ 71.26748632,  28.57166587,  42.42682695,  79.17638328,
          61.59651907,  56.73174844,  51.05052521,  60.49256264,
          55.68454882,  36.04326289,  60.21487085,  65.5908128 ,
          45.59597506,  49.33872983,  70.29440059, 163.07350864]),
  'mean_score_time': array([0.14747543, 0.16175866, 0.15697408, 0.13776808, 0.15167861,
         0.17306476, 0.13433337, 0.14864049, 0.14410348, 0.15723443,
         0.14645414, 0.14615645, 0.14403286, 0.14965954, 0.14714146,
         0.10577707]),
  'std_score_time': array([0.0114348 , 0.01278391, 0.0162995 , 0.01376354, 0.01167483,
         0.01661709, 0.00781908, 0.0156378 , 0.00865551, 0.00613101,
         0.01504126, 0.01407745, 0.01456839, 0.00552036, 0.01843315,
         0.04791131]),
  'param_colsample_bytree': masked_array(data=[0.6, 0.6, 0.6, 0.6, 0.7, 0.7, 0.7, 0.7, 0.8, 0.8, 0.8,
                    0.8, 0.9, 0.9, 0.9, 0.9],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False],
         fill_value='?',
              dtype=object),
  'param_subsample': masked_array(data=[0.6, 0.7, 0.8, 0.9, 0.6, 0.7, 0.8, 0.9, 0.6, 0.7, 0.8,
                    0.9, 0.6, 0.7, 0.8, 0.9],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False],
         fill_value='?',
              dtype=object),
  'params': [{'colsample_bytree': 0.6, 'subsample': 0.6},
   {'colsample_bytree': 0.6, 'subsample': 0.7},
   {'colsample_bytree': 0.6, 'subsample': 0.8},
   {'colsample_bytree': 0.6, 'subsample': 0.9},
   {'colsample_bytree': 0.7, 'subsample': 0.6},
   {'colsample_bytree': 0.7, 'subsample': 0.7},
   {'colsample_bytree': 0.7, 'subsample': 0.8},
   {'colsample_bytree': 0.7, 'subsample': 0.9},
   {'colsample_bytree': 0.8, 'subsample': 0.6},
   {'colsample_bytree': 0.8, 'subsample': 0.7},
   {'colsample_bytree': 0.8, 'subsample': 0.8},
   {'colsample_bytree': 0.8, 'subsample': 0.9},
   {'colsample_bytree': 0.9, 'subsample': 0.6},
   {'colsample_bytree': 0.9, 'subsample': 0.7},
   {'colsample_bytree': 0.9, 'subsample': 0.8},
   {'colsample_bytree': 0.9, 'subsample': 0.9}],
  'split0_test_score': array([0.8680417 , 0.86751317, 0.87199291, 0.87574549, 0.87051813,
         0.86047005, 0.87340971, 0.87302696, 0.85865003, 0.8638339 ,
         0.86884814, 0.87151808, 0.85948545, 0.86152626, 0.86418853,
         0.86789934]),
  'split1_test_score': array([0.86579544, 0.86411352, 0.87016436, 0.87449491, 0.86529675,
         0.86402742, 0.86128928, 0.86206758, 0.85659813, 0.86349888,
         0.86603584, 0.86113413, 0.85355481, 0.85434334, 0.85650265,
         0.86026461]),
  'split2_test_score': array([0.89152985, 0.89352207, 0.88986156, 0.88609278, 0.89140453,
         0.8839761 , 0.88307248, 0.88469047, 0.88392836, 0.88558812,
         0.88991953, 0.88219102, 0.87824664, 0.88555658, 0.8905819 ,
         0.88063782]),
  'split3_test_score': array([0.86572039, 0.87143217, 0.87851341, 0.87351978, 0.87031981,
         0.87695113, 0.87823061, 0.87956236, 0.87089399, 0.87590647,
         0.87401854, 0.87503835, 0.87718937, 0.87017669, 0.867041  ,
         0.87235257]),
  'split4_test_score': array([0.89653488, 0.89419789, 0.8973413 , 0.89080167, 0.89121388,
         0.89562819, 0.89502145, 0.89037232, 0.89054886, 0.89477892,
         0.89599327, 0.88873977, 0.88628709, 0.89188147, 0.89722989,
         0.89208286]),
  'mean_test_score': array([0.87752445, 0.87815576, 0.88157471, 0.88013093, 0.87775062,
         0.87621058, 0.87820471, 0.87794394, 0.87212387, 0.87672126,
         0.87896306, 0.87572427, 0.87095267, 0.87269687, 0.87510879,
         0.87464744]),
  'std_test_score': array([0.01359688, 0.01303173, 0.01047507, 0.00698731, 0.01122776,
         0.0129158 , 0.01109615, 0.00978606, 0.01343824, 0.01221703,
         0.01186545, 0.00940486, 0.01234039, 0.01415208, 0.01587003,
         0.01093204]),
  'rank_test_score': array([ 8,  5,  1,  2,  7, 10,  4,  6, 15,  9,  3, 11, 16, 14, 12, 13],
        dtype=int32),
  'split0_train_score': array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]),
  'split1_train_score': array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]),
  'split2_train_score': array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]),
  'split3_train_score': array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]),
  'split4_train_score': array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]),
  'mean_train_score': array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]),
  'std_train_score': array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])},
 {'colsample_bytree': 0.6, 'subsample': 0.8},
 0.8815747066619146)
```

best values for colsample_bytree and subsample is .6

Step 5: Tuning Regularization Parameters

In [145]:
```
param_test5 = {
 'reg_alpha':[1e-5, 1e-2, 0.1, 1, 100],
 'reg_lambda':[1e-5, 1e-2, 0.1, 1, 100]
}
gsearch5=GridSearchCV(
    estimator = XGBClassifier(
        learning_rate =learning_rate,
        n_estimators=351,
        max_depth=8,
        min_child_weight=1,
        gamma=0,
        subsample=.6,
        colsample=.6,
        objective= 'binary:logistic',
        nthread=4,
        scale_pos_weight=1,
        seed=1301),
    param_grid = param_test5, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch5.fit(x_train,y)
gsearch5.cv_results_, gsearch5.best_params_, gsearch5.best_score_
```

```
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split0_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split1_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split2_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split3_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split4_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('mean_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('std_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
```

Out[145]:
```
({'mean_fit_time': array([613.32489223, 597.04397445, 617.36770396, 670.6086051 ,
         809.29219165, 655.96184397, 638.26329179, 651.12268691,
         659.28381166, 675.18172693, 618.52337799, 586.23481021,
         619.5687561 , 670.47982998, 685.00779595, 665.32087135,
         713.20123882, 645.57410202, 716.57514105, 699.58594437,
         137.75241084, 138.37652321, 128.10776258, 125.28999438,
         116.32660394]),
  'std_fit_time': array([ 67.75170565,  77.99389043,  72.79295533,  75.73776067,
          38.64910607, 137.18812092, 100.31297799,  68.13214409,
          69.78319228, 134.59346441,  74.45916994, 115.8222221 ,
          41.60988469, 114.61481558,  87.17122848, 129.41783454,
          52.1528167 ,  72.63453217,  59.46233373,  75.08578462,
          10.82956082,  17.68523022,  17.63717503,  20.0968131 ,
          32.47607902]),
  'mean_score_time': array([0.16238723, 0.1486681 , 0.15846596, 0.14258766, 0.15205188,
         0.15453854, 0.15356555, 0.15170436, 0.13798852, 0.15300884,
         0.15345407, 0.15297799, 0.16912622, 0.14655952, 0.1529397 ,
         0.13787222, 0.15909162, 0.15076842, 0.14881001, 0.13768387,
         0.06439171, 0.08082733, 0.07304564, 0.07679133, 0.05479207]),
  'std_score_time': array([0.01321815, 0.00707261, 0.00791572, 0.01955738, 0.01256013,
         0.00513642, 0.01384699, 0.01775573, 0.01638456, 0.0149503 ,
         0.01001195, 0.01113546, 0.01226836, 0.00929172, 0.01472137,
         0.01257205, 0.01930258, 0.01429774, 0.01668555, 0.01850323,
         0.02107542, 0.01114065, 0.00586372, 0.01593616, 0.02572647]),
  'param_reg_alpha': masked_array(data=[1e-05, 1e-05, 1e-05, 1e-05, 1e-05, 0.01, 0.01, 0.01,
                     0.01, 0.01, 0.1, 0.1, 0.1, 0.1, 0.1, 1, 1, 1, 1, 1,
                     100, 100, 100, 100, 100],
               mask=[False, False, False, False, False, False, False, False,
                     False, False, False, False, False, False, False, False,
                     False, False, False, False, False, False, False, False,
                     False],
         fill_value='?',
              dtype=object),
  'param_reg_lambda': masked_array(data=[1e-05, 0.01, 0.1, 1, 100, 1e-05, 0.01, 0.1, 1, 100,
                     1e-05, 0.01, 0.1, 1, 100, 1e-05, 0.01, 0.1, 1, 100,
                     1e-05, 0.01, 0.1, 1, 100],
               mask=[False, False, False, False, False, False, False, False,
                     False, False, False, False, False, False, False, False,
                     False, False, False, False, False, False, False, False,
                     False],
         fill_value='?',
              dtype=object),
  'params': [{'reg_alpha': 1e-05, 'reg_lambda': 1e-05},
   {'reg_alpha': 1e-05, 'reg_lambda': 0.01},
   {'reg_alpha': 1e-05, 'reg_lambda': 0.1},
   {'reg_alpha': 1e-05, 'reg_lambda': 1},
   {'reg_alpha': 1e-05, 'reg_lambda': 100},
   {'reg_alpha': 0.01, 'reg_lambda': 1e-05},
   {'reg_alpha': 0.01, 'reg_lambda': 0.01},
   {'reg_alpha': 0.01, 'reg_lambda': 0.1},
   {'reg_alpha': 0.01, 'reg_lambda': 1},
   {'reg_alpha': 0.01, 'reg_lambda': 100},
   {'reg_alpha': 0.1, 'reg_lambda': 1e-05},
   {'reg_alpha': 0.1, 'reg_lambda': 0.01},
   {'reg_alpha': 0.1, 'reg_lambda': 0.1},
   {'reg_alpha': 0.1, 'reg_lambda': 1},
   {'reg_alpha': 0.1, 'reg_lambda': 100},
   {'reg_alpha': 1, 'reg_lambda': 1e-05},
   {'reg_alpha': 1, 'reg_lambda': 0.01},
   {'reg_alpha': 1, 'reg_lambda': 0.1},
   {'reg_alpha': 1, 'reg_lambda': 1},
   {'reg_alpha': 1, 'reg_lambda': 100},
   {'reg_alpha': 100, 'reg_lambda': 1e-05},
   {'reg_alpha': 100, 'reg_lambda': 0.01},
   {'reg_alpha': 100, 'reg_lambda': 0.1},
   {'reg_alpha': 100, 'reg_lambda': 1},
   {'reg_alpha': 100, 'reg_lambda': 100}],
  'split0_test_score': array([0.85843606, 0.85201098, 0.84766764, 0.85656914, 0.83856751,
         0.85359231, 0.8479319 , 0.8547619 , 0.86166522, 0.83874653,
         0.85580107, 0.85430669, 0.85207918, 0.86067124, 0.83757182,
         0.85869435, 0.85346871, 0.85010059, 0.84927284, 0.83836291,
         0.75437062, 0.75437062, 0.75436721, 0.7543638 , 0.75179616]),
  'split1_test_score': array([0.85614035, 0.85485568, 0.85092834, 0.85899187, 0.83175626,
         0.85292568, 0.85690502, 0.85343802, 0.85516171, 0.83379282,
         0.85178763, 0.85471502, 0.85528276, 0.85573798, 0.83551651,
         0.86134298, 0.84783728, 0.85294955, 0.85018669, 0.83519257,
         0.74252553, 0.74252553, 0.74252638, 0.74314784, 0.74368574]),
  'split2_test_score': array([0.86729409, 0.87577873, 0.87478646, 0.87343017, 0.88097625,
         0.86925903, 0.87392972, 0.87547781, 0.87736092, 0.88126609,
         0.87201763, 0.87067413, 0.88560687, 0.87747686, 0.88141442,
         0.87867117, 0.87950062, 0.88338363, 0.88638433, 0.88128825,
         0.77813667, 0.77813667, 0.77814349, 0.77813582, 0.7745077 ]),
  'split3_test_score': array([0.86738122, 0.86724324, 0.86533303, 0.87418394, 0.85844718,
         0.87110909, 0.87257281, 0.86445805, 0.87093341, 0.85836834,
         0.86903691, 0.88213844, 0.87497322, 0.87339552, 0.85955355,
         0.86719183, 0.87059404, 0.8701167 , 0.86264297, 0.85636643,
         0.77676397, 0.77676397, 0.77676054, 0.77640404, 0.77342688]),
  'split4_test_score': array([0.87659977, 0.88445229, 0.88599058, 0.88354218, 0.87644037,
         0.88607713, 0.88724263, 0.88709266, 0.89121388, 0.8762107 ,
         0.89303582, 0.89017779, 0.88844583, 0.88681671, 0.87309472,
         0.89103906, 0.88928996, 0.88515416, 0.88795821, 0.87374259,
         0.78040957, 0.78041128, 0.78041471, 0.78100431, 0.78056297]),
  'mean_test_score': array([0.8651703 , 0.86686819, 0.86494121, 0.86934346, 0.85723751,
         0.86659265, 0.86771642, 0.86704569, 0.87126703, 0.85767689,
         0.86833581, 0.87040241, 0.87127757, 0.87081966, 0.8574302 ,
         0.87138788, 0.86813812, 0.86834093, 0.86728901, 0.85699055,
         0.76644127, 0.76644161, 0.76644247, 0.76661116, 0.76479589]),
  'std_test_score': array([0.00730646, 0.01227824, 0.01438676, 0.01012002, 0.01965505,
         0.01235271, 0.01379693, 0.01277292, 0.01254352, 0.01912714,
         0.01452653, 0.01438082, 0.0150874 , 0.01128987, 0.01843743,
         0.01199039, 0.01555541, 0.01470867, 0.01691452, 0.01840402,
         0.01520603, 0.01520635, 0.01520783, 0.01507585, 0.01436694]),
  'rank_test_score': array([15, 13, 16,  6, 19, 14, 10, 12,  3, 17,  8,  5,  2,  4, 18,  1,  9,
          7, 11, 20, 24, 23, 22, 21, 25], dtype=int32),
  'split0_train_score': array([1.        , 1.        , 1.        , 1.        , 0.98127441,
         1.        , 1.        , 1.        , 1.        , 0.98122991,
         1.        , 1.        , 1.        , 1.        , 0.98127622,
         1.        , 1.        , 1.        , 1.        , 0.97801803,
         0.79305393, 0.79305041, 0.79305073, 0.79311253, 0.79229702]),
  'split1_train_score': array([1.        , 1.        , 1.        , 1.        , 0.97966235,
         1.        , 1.        , 1.        , 1.        , 0.98003186,
         1.        , 1.        , 1.        , 1.        , 0.97961929,
         1.        , 1.        , 1.        , 1.        , 0.97681143,
         0.79231305, 0.7923131 , 0.79231257, 0.7919624 , 0.79238079]),
  'split2_train_score': array([1.        , 1.        , 1.        , 1.        , 0.97884631,
         1.        , 1.        , 1.        , 1.        , 0.97931063,
         1.        , 1.        , 1.        , 1.        , 0.97903638,
         1.        , 1.        , 1.        , 1.        , 0.9760643 ,
         0.78702612, 0.78702601, 0.78703472, 0.78725358, 0.78345264]),
  'split3_train_score': array([1.        , 1.        , 1.        , 1.        , 0.97958431,
         1.        , 1.        , 1.        , 1.        , 0.97982135,
         1.        , 1.        , 1.        , 1.        , 0.97944961,
         1.        , 1.        , 1.        , 1.        , 0.97551944,
         0.79896559, 0.79896554, 0.79897941, 0.79846005, 0.79013228]),
  'split4_train_score': array([1.        , 1.        , 1.        , 1.        , 0.98099755,
         1.        , 1.        , 1.        , 1.        , 0.98039085,
         1.        , 1.        , 1.        , 1.        , 0.98021576,
         1.        , 1.        , 1.        , 1.        , 0.97718641,
         0.7884946 , 0.78849476, 0.78849572, 0.78869258, 0.78612823]),
  'mean_train_score': array([1.        , 1.        , 1.        , 1.        , 0.98007299,
         1.        , 1.        , 1.        , 1.        , 0.98015692,
         1.        , 1.        , 1.        , 1.        , 0.97991945,
         1.        , 1.        , 1.        , 1.        , 0.97671992,
         0.79197066, 0.79196997, 0.79197463, 0.79189623, 0.78887819]),
  'std_train_score': array([0.        , 0.        , 0.        , 0.        , 0.00091765,
         0.        , 0.        , 0.        , 0.        , 0.00064039,
         0.        , 0.        , 0.        , 0.        , 0.00077708,
         0.        , 0.        , 0.        , 0.        , 0.00086972,
         0.00416536, 0.00416516, 0.0041676 , 0.00390858, 0.00353633])},
 {'reg_alpha': 1, 'reg_lambda': 1e-05},
 0.8713878774249088)
```

optimum value for reg_alpha is 1, reg_lambda is 1e-05. I'll try a few more values

In [146]:
```
param_test6 = {
 'reg_alpha':[0.5, 1, 10],
}
```

In [147]:
```
gsearch6=GridSearchCV(
    estimator = XGBClassifier(
        learning_rate =learning_rate,
        n_estimators=351,
        max_depth=8,
        min_child_weight=1,
        gamma=0,
        subsample=.6,
        colsample=.6,
        objective= 'binary:logistic',
        nthread=4,
        scale_pos_weight=1,
        reg_lambda=1e-05,
        seed=1301),
    param_grid = param_test6, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch6.fit(x_train,y)
gsearch6.cv_results_, gsearch6.best_params_, gsearch6.best_score_
```

```
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split0_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split1_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split2_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split3_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('split4_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('mean_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/home/jpalomares/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are accessing a training score ('std_train_score'), which will not be available by default any more in 0.21. If you need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
```

Out[147]:
```
({'mean_fit_time': array([679.0231245 , 671.59132581, 447.08077736]),
  'std_fit_time': array([51.48427705, 38.43697281, 90.3449704 ]),
  'mean_score_time': array([0.14733696, 0.1621686 , 0.08870173]),
  'std_score_time': array([0.02269533, 0.0211357 , 0.03820289]),
  'param_reg_alpha': masked_array(data=[0.5, 1, 10],
               mask=[False, False, False],
         fill_value='?',
              dtype=object),
  'params': [{'reg_alpha': 0.5}, {'reg_alpha': 1}, {'reg_alpha': 10}],
  'split0_test_score': array([0.85377304, 0.85869435, 0.83605272]),
  'split1_test_score': array([0.85568769, 0.86134298, 0.84096125]),
  'split2_test_score': array([0.87280702, 0.87867117, 0.88490273]),
  'split3_test_score': array([0.86705728, 0.86719183, 0.85985863]),
  'split4_test_score': array([0.88898658, 0.89103906, 0.8821393 ]),
  'mean_test_score': array([0.86766232, 0.87138788, 0.86078293]),
  'std_test_score': array([0.01278954, 0.01199039, 0.02021483]),
  'rank_test_score': array([2, 1, 3], dtype=int32),
  'split0_train_score': array([1.        , 1.        , 0.98649958]),
  'split1_train_score': array([1.        , 1.        , 0.98636251]),
  'split2_train_score': array([1.        , 1.        , 0.98618269]),
  'split3_train_score': array([1.        , 1.        , 0.98641841]),
  'split4_train_score': array([1.        , 1.        , 0.98650083]),
  'mean_train_score': array([1.       , 1.       , 0.9863928]),
  'std_train_score': array([0.       , 0.       , 0.0001173])},
 {'reg_alpha': 1},
 0.8713878774249088)
```

Best value for alpha is still 1.

Step 6: Reducing Learning Rate

Lastly, we should lower the learning rate and add more trees

In [12]:
```
xgb4 = XGBClassifier(
    learning_rate = 0.05,
    n_estimators= 5000,
    max_depth=8,
    min_child_weight=1,
    gamma=0,
    subsample=.6,
    colsample=.6,
    objective= 'binary:logistic',
    nthread=4,
    scale_pos_weight=1,
    reg_lambda=1e-05,
    reg_alpha=1,
    seed=1301,
    eval_metric = 'auc')
modelfit(xgb4,x_train,y)
```

```
Model Report
Accuracy : 0.9921870231337362
AUC Score (Train): 0.9999610452981674
```

In [13]:
```
y_test = xgb4.predict_proba(x_test)[:,1]
```

In [164]:
```
write_predictions("predictions/xgboost_opt_"+datetime.datetime.now().isoformat() + ".csv","Id,Y\n",ids,y_test)
```

I didn't see significant improvements from in the auc score from my ad hoc tuning. I want to try 2 more things:

1. stacking the output of the optimized xgboost to another xgboost model.
2. Increase the max depth to 8 from 7. I want to allow the fit to try using trees that are 1 layer deeper than waht grid search found. I'm concerned that my parameter space wasn't large enough in the Grid Search and I might have missed a better set of parameters. I'm worired if this will overfit, so I'll decrease the col by sample to .5 so that only half of the columns are fit per tree.

In [19]:
```
y_pred = xgb4.predict_proba(x_train)[:,1]
```

Out[19]:
```
(16383, 25)
```

In [ ]:
```
x_train['f25'] = y_pred
x_train.shape
```

In [25]:
```
y_pred_test = xgb4.predict_proba(x_test)[:,1]
```

In [26]:
```
x_test['f25'] = y_pred_test
```

In [ ]:
```
clf = xgb.XGBClassifier(
                max_depth = 7,
                n_estimators=700,
                learning_rate=0.1,
                nthread=4,
                subsample=1.0,
                colsample_bytree=0.5,
                min_child_weight = 3,
                seed=1301,
                objective='binary:logistic',
                eval_metric='auc')
modelfit(clf,x_train,y)
```

In [ ]:
```
clf.fit(x_train,y)
y_train,y_test =get_prediction_proba(clf,x_train,x_test)
write_predictions("predictions/xgboost_stacking_"+datetime.datetime.now().isoformat() + ".csv","Id,Y\n",ids,y_test)
```

Stacking had a score of .82964.

Since stacking did not have a higher auc score. I will try option 2 next.

In [21]:
```
clf = xgb.XGBClassifier(
                max_depth = 8,
                n_estimators=700,
                learning_rate=0.1,
                nthread=4,
                subsample=1.0,
                colsample_bytree=0.5,
                min_child_weight = 1,
                seed=1301,
                objective='binary:logistic',
                eval_metric='auc')
```

In [32]:
```
x_train,y,x_test,ids = get_train_test()
```

In [23]:
```
clf.fit(x_train,y)
```

Out[23]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=0.5, eval_metric='auc', gamma=0, learning_rate=0.1,
       max_delta_step=0, max_depth=8, min_child_weight=1, missing=None,
       n_estimators=700, n_jobs=1, nthread=4, objective='binary:logistic',
       random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
       seed=1301, silent=True, subsample=1.0)
```

In [27]:
```
y_train,y_test =get_prediction_proba(clf,x_train,x_test)
```

In [31]:
```
write_predictions("predictions/xgboost_"+datetime.datetime.now().isoformat() + ".csv","Id,Y\n",ids,y_test)
```

This prediction had a score of 0.89081! This model ended up being my best fit, and I got me in thirds place in the competition with a private leaderboard score of about .90.

Keras

After getting as far as I could with Xgboost, I decided to try using Keras.

In [37]:
```
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from keras.utils import np_utils
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
import numpy
```

In [39]:
```
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
x_train,y,x_test,ids = get_train_test()
num_features = x_train.shape[1]
```

In [46]:
```
# create model
def baseline_model():
    model = Sequential()
    model.add(Dense(36, input_dim=num_features, activation='relu'))
    model.add(Dense(1, activation='softmax'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam',metrics=['accuracy'])
    return model
```

In [47]:
```
estimator = KerasClassifier(build_fn=baseline_model, epochs=50, batch_size=5, verbose=0)
kfold = KFold(n_splits=10, shuffle=True, random_state=seed)
```

In [48]:
```
results = cross_val_score(estimator,x_train,y,cv=kfold)
```

In [24]:
```
model.evaluate(x_train,y)
```

```
16383/16383 [==============================] - 0s 12us/step
```

Out[24]:
```
0.9569809407353391
```

In [25]:
```
y_test = model.predict_proba(x_test)
```

In [26]:
```
print(y_test)
```

```
[[1.]
 [1.]
 [1.]
 ...
 [1.]
 [1.]
 [1.]]
```

I was not able to get keras working prior to the end of the competition. It predicts all labels to be 1, even when using predict_proba() instead of predict().

Conclusion

I ended up doing well on the competition - finishing in 3rd place with pretty much nothing but XgBoost parameter tuning. I'm disappointed that none of my pre processing ideas produced any significant improvements in a model accuracy. Given more time, I would continue to try modeling with keras or other neural networks.