Credit Card Fraud: Handling highly imbalance classes and why Receiver Operating Characteristics Curve (ROC Curve) should not be used, and Precision/Recall curve should be preferred in highly imbalanced situations

Motivation

In this notebook, we will explore the data through initial EDA with some visualizations. Then we will hit the main point of the notebook, which is to explore ways to handle imbalanced data. We will use two methods:

1. Using the weights parameters in Sci-Kit Learn classifiers
2. Over and Undersampling (to be developed in the next version)

Finally, we will quantify and illustrate the effects of the trade off between True Positive Rate and False Positive Rate using ROC and Precision/Recall (PR) curves, and disucss **why the popular ROC curve should not be used on highly imbalanced dataset** (or in general, why I prefer PR curves over ROC).

Imports and reading in the data

In [1]:

import pandas as pd

pd.options.display.max\_colwidth = 200

pd.options.display.max\_columns = 200

import numpy as np

import time

from sklearn.linear\_model import LogisticRegression

from sklearn.ensemble import GradientBoostingClassifier,RandomForestClassifier,AdaBoostClassifier

from sklearn.neural\_network import MLPClassifier

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import cross\_val\_predict,cross\_val\_score,train\_test\_split

from sklearn.metrics import classification\_report,confusion\_matrix,roc\_curve,auc,precision\_recall\_curve,roc\_curve

import os

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

import seaborn as sns

%matplotlib inline

In [2]:

df = pd.read\_csv('../input/creditcard.csv')

The data has a column called 'Time', which are seconds from which the very first data observation took place. Let's convert that to hours of a day. I'm guessing the data starts at midnight and ends at midnight.

In [3]:

df['hour'] = df['Time'].apply(lambda x: np.ceil(float(x)/3600) % 24)

Seems right that we have the least transactions from 1AM to 5AM. Let's see a breakdown of our legit vs fraud transactions via a pivot table.

**Note:**

* Class 0 = Legit transactions
* Class 1 = Fruadulent transactions

In [4]:

df.pivot\_table(values='Amount',index='hour',columns='Class',aggfunc='count')

Out[4]:

| Class | 0 | 1 |
| --- | --- | --- |
| hour |  |  |
| 0.0 | 10919 | 21 |
| 1.0 | 7687 | 6 |
| 2.0 | 4212 | 10 |
| 3.0 | 3269 | 57 |
| 4.0 | 3476 | 17 |
| 5.0 | 2185 | 23 |
| 6.0 | 2979 | 11 |
| 7.0 | 4093 | 9 |
| 8.0 | 7219 | 23 |
| 9.0 | 10266 | 9 |
| 10.0 | 15824 | 16 |
| 11.0 | 16593 | 8 |
| 12.0 | 16804 | 53 |
| 13.0 | 15400 | 17 |
| 14.0 | 15350 | 17 |
| 15.0 | 16545 | 23 |
| 16.0 | 16434 | 26 |
| 17.0 | 16435 | 22 |
| 18.0 | 16135 | 29 |
| 19.0 | 17003 | 33 |
| 20.0 | 15632 | 19 |
| 21.0 | 16739 | 18 |
| 22.0 | 17692 | 16 |
| 23.0 | 15424 | 9 |

Visualizing the data

In [5]:

def PlotHistogram(df,norm):

bins = np.arange(df['hour'].min(),df['hour'].max()+2)

plt.figure(figsize=(15,4))

sns.distplot(df[df['Class']==0.0]['hour'],

norm\_hist=norm,

bins=bins,

kde=False,

color='b',

hist\_kws={'alpha':.5},

label='Legit')

sns.distplot(df[df['Class']==1.0]['hour'],

norm\_hist=norm,

bins=bins,

kde=False,

color='r',

label='Fraud',

hist\_kws={'alpha':.5})

plt.xticks(range(0,24))

plt.legend()

plt.show()

In [6]:

start = time.time()

print('Normalized histogram of Legit/Fraud over hour of the day')

PlotHistogram(df,True)

print('Counts histogram of Legit/Fraud over hour of the day')

print('\*you can barely see the Fraud cases since there are so little of them.')

PlotHistogram(df,False)

print(time.time()-start)

Normalized histogram of Legit/Fraud over hour of the day

Counts histogram of Legit/Fraud over hour of the day

\*you can barely see the Fraud cases since there are so little of them.

0.9953107833862305

In [7]:

print('Fraud is **{}% o**f our data.'.format(df['Class'].value\_counts()[1] / float(df['Class'].value\_counts()[0])\*100))

Fraud is 0.17304750013189596% of our data.

Hour of the day seems to have some impact on the number of Fraud cases. I'll be sure to to add the 'hour' dimension to visualizations later to further investigate its impact.

Before we train our classifers, we need to normalize the Amount since it's on a totally different scale. The distributions are also highly skewed with a lot of statistical outliers. All Fraud cases are in the low dollar values i.e. Amount.

**We also have a HUGE class imbalance.** More on that later when we start to train classifiers.

In [8]:

mask\_true = (df['Class'] == 1.0)

mask\_false = (df['Class'] == 0.0)

df['Amount'] = StandardScaler().fit\_transform(df[['Amount']])

In [9]:

def PlotViolins(minHour,maxHour):

plt.figure(figsize=(15,6))

plt.title('Amount by class throughout the day')

plt.ylim([-1,3.0])

sns.violinplot(data=df[df['hour'].isin(range(minHour,maxHour+1))],x='hour',y='Amount',hue='Class',split=True,palette='Set2',cut=0)

plt.legend(loc='lower right')

plt.show()

PlotViolins(0,11)

PlotViolins(12,23)

Let's see how well the PCA components complement each other by looking their interactions with each other. I'm only including the first 6 components here.

In [10]:

*# Model building*

Let's start with a vanilla Logistic Regression since it seems like for some of the features, a sigmoid curve can sort of separate the classes.sns.pairplot(data=pd.concat([df.loc[:,'hour'],df.loc[:,'V1':'V6'],df.loc[:,'Class']],axis=1),

hue='Class',

diag\_kind='kde',

plot\_kws={'alpha':0.2})

File "<ipython-input-10-0b4abfa933c6>", line 2

Let's start with a vanilla Logistic Regression since it seems like for some of the features, a sigmoid curve can sort of separate the classes.sns.pairplot(data=pd.concat([df.loc[:,'hour'],df.loc[:,'V1':'V6'],df.loc[:,'Class']],axis=1),

^

SyntaxError: invalid syntax

Seems like most features show both Fraud and Legit purchases overlapping each other, with V4 showing a little promise. We can't rely on our eyes to do all the investigation work since we at most can make 4-dimensional charts (3 features + a color) to investigate our data, and our data has more than 30 features.

Let's get to model building and see how well our data can separate the classes.

Model building

Let's start with a vanilla Logistic Regression since it seems like for some of the features, a sigmoid curve can sort of separate the classes.

In [11]:

features = pd.concat([df.loc[:,'V1':'Amount'],df.loc[:,'Time']],axis=1)

target = df['Class']

x\_train,x\_test,y\_train,y\_test = train\_test\_split(features,target, stratify=target,test\_size=0.35, random\_state=1)

print('y\_train class counts')

print(y\_train.value\_counts())

print('')

print('y\_test class counts')

print(y\_test.value\_counts())

*# Let's store our y\_test legit and fraud counts for normalization purposes later on*

y\_test\_legit = y\_test.value\_counts()[0]

y\_test\_fraud = y\_test.value\_counts()[1]

y\_train class counts

0 184804

1 320

Name: Class, dtype: int64

y\_test class counts

0 99511

1 172

Name: Class, dtype: int64

In [12]:

lr\_model = LogisticRegression()

lr\_model.fit(x\_train,y\_train)

pred = lr\_model.predict(x\_test)

In [13]:

def PlotConfusionMatrix(y\_test,pred,y\_test\_legit,y\_test\_fraud):

cfn\_matrix = confusion\_matrix(y\_test,pred)

cfn\_norm\_matrix = np.array([[1.0 / y\_test\_legit,1.0/y\_test\_legit],[1.0/y\_test\_fraud,1.0/y\_test\_fraud]])

norm\_cfn\_matrix = cfn\_matrix \* cfn\_norm\_matrix

fig = plt.figure(figsize=(15,5))

ax = fig.add\_subplot(1,2,1)

sns.heatmap(cfn\_matrix,cmap='coolwarm\_r',linewidths=0.5,annot=True,ax=ax)

plt.title('Confusion Matrix')

plt.ylabel('Real Classes')

plt.xlabel('Predicted Classes')

ax = fig.add\_subplot(1,2,2)

sns.heatmap(norm\_cfn\_matrix,cmap='coolwarm\_r',linewidths=0.5,annot=True,ax=ax)

plt.title('Normalized Confusion Matrix')

plt.ylabel('Real Classes')

plt.xlabel('Predicted Classes')

plt.show()

print('---Classification Report---')

print(classification\_report(y\_test,pred))

PlotConfusionMatrix(y\_test,pred,y\_test\_legit,y\_test\_fraud)

---Classification Report---

precision recall f1-score support

0 1.00 1.00 1.00 99511

1 0.72 0.59 0.65 172

avg / total 1.00 1.00 1.00 99683

I'm not a huge fan of confusion matrix because they can be misleading. I almost ALWAYS go directly to my classification report for precision and recall scores for each class. And Seaborn's heatmap coloring scheme is not helping. Our True Positive rate is 0.59 as also shown in the classification report which is not TOO bad, but it's shown as blood red. The normalized confusion matrix tells a better story. But, actual precision and recall scores are my go-to metrics.

It seems like we aren't very good with catching our frauds, which is expected with a vanilla Logistic Regression without addressing the class imbalance issue.

Addressing class imbalance

Using weights to counteract the class imbalance

Sci-Kit Learn classifiers can give heavier weights to the minority class using a simple parameter during model initiation. Let's see how that will improve our results

In [14]:

lr\_model = LogisticRegression(class\_weight='balanced')

lr\_model.fit(x\_train,y\_train)

pred = lr\_model.predict(x\_test)

PlotConfusionMatrix(y\_test,pred,y\_test\_legit,y\_test\_fraud)

---Classification Report---

precision recall f1-score support

0 1.00 0.98 0.99 99511

1 0.06 0.92 0.12 172

avg / total 1.00 0.98 0.99 99683

Looking at the normalized confusion matrix, it seems like our classifer is doing very well! We have a 98% True Negative rate and a 92% True Positive rate! Seems like a perfect classifer, doesn't it?

However, if we look at the individual precision scores, this classifer is now a lot less precise than before. This is because we have increased our Fraud recall score at the expense of more mis-classified Legit cases. With the "balanced" weight parameter, we have increased our false positive counts from 39 to 2300. 2300 is still only a small fraction of truely negative cases (out of 99511), that's why the percentage shown on the Normalized Confusion Matrix is still relatively small at 0.023%. Let's try to specify our own weights. The weights are somewhat arbitrary but it illustrates the tradeoff between precision and recall.

In [15]:

for w **in** [1,5,10,100,500,1000]:

print('---Weight of **{}** for Fraud class---'.format(w))

lr\_model = LogisticRegression(class\_weight={0:1,1:w})

lr\_model.fit(x\_train,y\_train)

pred = lr\_model.predict(x\_test)

PlotConfusionMatrix(y\_test,pred,y\_test\_legit,y\_test\_fraud)

---Weight of 1 for Fraud class---

---Classification Report---

precision recall f1-score support

0 1.00 1.00 1.00 99511

1 0.72 0.59 0.65 172

avg / total 1.00 1.00 1.00 99683

---Weight of 5 for Fraud class---

---Classification Report---

precision recall f1-score support

0 1.00 1.00 1.00 99511

1 0.77 0.81 0.79 172

avg / total 1.00 1.00 1.00 99683

---Weight of 10 for Fraud class---

---Classification Report---

precision recall f1-score support

0 1.00 1.00 1.00 99511

1 0.67 0.83 0.74 172

avg / total 1.00 1.00 1.00 99683

---Weight of 100 for Fraud class---

---Classification Report---

precision recall f1-score support

0 1.00 1.00 1.00 99511

1 0.26 0.87 0.41 172

avg / total 1.00 1.00 1.00 99683

---Weight of 500 for Fraud class---

---Classification Report---

precision recall f1-score support

0 1.00 0.98 0.99 99511

1 0.07 0.92 0.14 172

avg / total 1.00 0.98 0.99 99683

---Weight of 1000 for Fraud class---

---Classification Report---

precision recall f1-score support

0 1.00 0.96 0.98 99511

1 0.04 0.94 0.07 172

avg / total 1.00 0.96 0.98 99683

ROC versus Precision/Recall Curves

Just by manaually selecting a range of weights to boost the minority class already helped our model have better recall, and in some cases, better precision also. Recall and Precision are usually trade offs of each other, so when you can improve both **at the same time**, your model's overall performance is undeniably improved.

To illustrate the trade off between precision vs recall, and let's also include False Positive Rate vs True Positive Rate (ROC), let's plot the ROC and Precision/Recall curves for different weights for the minority class.

In [16]:

fig = plt.figure(figsize=(15,8))

ax1 = fig.add\_subplot(1,2,1)

ax1.set\_xlim([-0.05,1.05])

ax1.set\_ylim([-0.05,1.05])

ax1.set\_xlabel('Recall')

ax1.set\_ylabel('Precision')

ax1.set\_title('PR Curve')

ax2 = fig.add\_subplot(1,2,2)

ax2.set\_xlim([-0.05,1.05])

ax2.set\_ylim([-0.05,1.05])

ax2.set\_xlabel('False Positive Rate')

ax2.set\_ylabel('True Positive Rate')

ax2.set\_title('ROC Curve')

for w,k **in** zip([1,5,10,20,50,100,10000],'bgrcmykw'):

lr\_model = LogisticRegression(class\_weight={0:1,1:w})

lr\_model.fit(x\_train,y\_train)

pred\_prob = lr\_model.predict\_proba(x\_test)[:,1]

p,r,\_ = precision\_recall\_curve(y\_test,pred\_prob)

tpr,fpr,\_ = roc\_curve(y\_test,pred\_prob)

ax1.plot(r,p,c=k,label=w)

ax2.plot(tpr,fpr,c=k,label=w)

ax1.legend(loc='lower left')

ax2.legend(loc='lower left')

plt.show()

Conclusion

For a PR curve, a good classifer aims for the upper right corner of the chart but upper left for the ROC curve.

While PR and ROC curves use the same data, i.e. the real class labels and predicted probability for the class lables, you can see that the two charts tell very different stories, with some weights seem to perform better in ROC than in the PR curve.

While the blue, w=1, line performed poorly in both charts, the black, w=10000, line performed "well" in the ROC but poorly in the PR curve. This is due to the high class imbalance in our data. ROC curve is not a good visual illustration for highly imbalanced data, because the False Positive Rate ( False Positives / Total Real Negatives ) does not drop drastically when the Total Real Negatives is huge.

Whereas Precision ( True Positives / (True Positives + False Positives) ) is highly sensitive to False Positives and is not impacted by a large total real negative denominator.

The biggest difference among the models are at around 0.8 recall rate. Seems like a lower weight, i.e. 5 and 10, out performs other weights significantly at 0.8 recall. This means that with those specific weights, our model can detect frauds fairly well (catching 80% of fraud) while not annoying a bunch of customers with false positives with an equally high precision of 80%.

Without further tuning our model, and of course we should do cross validation for any real model tuning/validation, it seems like a vanilla Logistic Regression is stuck at around 0.8 Precision and Recall.

So how do we know if we should sacrifice our precision for more recall, i.e. catching fraud? That is where data science meets your core business parameters. If the cost of missing a fraud highly outweighs the cost of canceling a bunch of legit customer transactions, i.e. false positives, then perhaps we can choose a weight that gives us a higher recall rate. Or maybe catching 80% of fraud is good enough for your business if you can minimize also minimize the "user friction" or credit card disruptions by keeping our precision high.