# How does credit card fraud occur?

**Credit card fraud** happens when consumers give their credit card number to unfamiliar individuals, when cards are lost or stolen, when mail is diverted from the intended recipient and taken by criminals, or when employees of a business copy the cards or card numbers of a cardholder. In this notebook we will develop a few ML models using anonymized credit card transaction data. The challenge behind fraud detection is that frauds are far less common as compared to legal transactions

# EDA

Before creating a model it is important to get a general understanding of the data

import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn as sns  
import warnings  
warnings.filterwarnings("ignore")  
plt.style.use('bmh')

df = pd.read\_csv(r'C:\Users\rahul\Documents\TCR ML Internship\3. Credit Card Fraud\creditcard.csv\creditcard.csv')  
df.head()

Time V1 V2 V3 V4 V5 V6 V7 \  
0 0.0 -1.359807 -0.072781 2.536347 1.378155 -0.338321 0.462388 0.239599   
1 0.0 1.191857 0.266151 0.166480 0.448154 0.060018 -0.082361 -0.078803   
2 1.0 -1.358354 -1.340163 1.773209 0.379780 -0.503198 1.800499 0.791461   
3 1.0 -0.966272 -0.185226 1.792993 -0.863291 -0.010309 1.247203 0.237609   
4 2.0 -1.158233 0.877737 1.548718 0.403034 -0.407193 0.095921 0.592941   
  
 V8 V9 ... V21 V22 V23 V24 V25 \  
0 0.098698 0.363787 ... -0.018307 0.277838 -0.110474 0.066928 0.128539   
1 0.085102 -0.255425 ... -0.225775 -0.638672 0.101288 -0.339846 0.167170   
2 0.247676 -1.514654 ... 0.247998 0.771679 0.909412 -0.689281 -0.327642   
3 0.377436 -1.387024 ... -0.108300 0.005274 -0.190321 -1.175575 0.647376   
4 -0.270533 0.817739 ... -0.009431 0.798278 -0.137458 0.141267 -0.206010   
  
 V26 V27 V28 Amount Class   
0 -0.189115 0.133558 -0.021053 149.62 0   
1 0.125895 -0.008983 0.014724 2.69 0   
2 -0.139097 -0.055353 -0.059752 378.66 0   
3 -0.221929 0.062723 0.061458 123.50 0   
4 0.502292 0.219422 0.215153 69.99 0   
  
[5 rows x 31 columns]

df.describe()

Time V1 V2 V3 V4 \  
count 284807.000000 2.848070e+05 2.848070e+05 2.848070e+05 2.848070e+05   
mean 94813.859575 3.918649e-15 5.682686e-16 -8.761736e-15 2.811118e-15   
std 47488.145955 1.958696e+00 1.651309e+00 1.516255e+00 1.415869e+00   
min 0.000000 -5.640751e+01 -7.271573e+01 -4.832559e+01 -5.683171e+00   
25% 54201.500000 -9.203734e-01 -5.985499e-01 -8.903648e-01 -8.486401e-01   
50% 84692.000000 1.810880e-02 6.548556e-02 1.798463e-01 -1.984653e-02   
75% 139320.500000 1.315642e+00 8.037239e-01 1.027196e+00 7.433413e-01   
max 172792.000000 2.454930e+00 2.205773e+01 9.382558e+00 1.687534e+01   
  
 V5 V6 V7 V8 V9 \  
count 2.848070e+05 2.848070e+05 2.848070e+05 2.848070e+05 2.848070e+05   
mean -1.552103e-15 2.040130e-15 -1.698953e-15 -1.893285e-16 -3.147640e-15   
std 1.380247e+00 1.332271e+00 1.237094e+00 1.194353e+00 1.098632e+00   
min -1.137433e+02 -2.616051e+01 -4.355724e+01 -7.321672e+01 -1.343407e+01   
25% -6.915971e-01 -7.682956e-01 -5.540759e-01 -2.086297e-01 -6.430976e-01   
50% -5.433583e-02 -2.741871e-01 4.010308e-02 2.235804e-02 -5.142873e-02   
75% 6.119264e-01 3.985649e-01 5.704361e-01 3.273459e-01 5.971390e-01   
max 3.480167e+01 7.330163e+01 1.205895e+02 2.000721e+01 1.559499e+01   
  
 ... V21 V22 V23 V24 \  
count ... 2.848070e+05 2.848070e+05 2.848070e+05 2.848070e+05   
mean ... 1.473120e-16 8.042109e-16 5.282512e-16 4.456271e-15   
std ... 7.345240e-01 7.257016e-01 6.244603e-01 6.056471e-01   
min ... -3.483038e+01 -1.093314e+01 -4.480774e+01 -2.836627e+00   
25% ... -2.283949e-01 -5.423504e-01 -1.618463e-01 -3.545861e-01   
50% ... -2.945017e-02 6.781943e-03 -1.119293e-02 4.097606e-02   
75% ... 1.863772e-01 5.285536e-01 1.476421e-01 4.395266e-01   
max ... 2.720284e+01 1.050309e+01 2.252841e+01 4.584549e+00   
  
 V25 V26 V27 V28 Amount \  
count 2.848070e+05 2.848070e+05 2.848070e+05 2.848070e+05 284807.000000   
mean 1.426896e-15 1.701640e-15 -3.662252e-16 -1.217809e-16 88.349619   
std 5.212781e-01 4.822270e-01 4.036325e-01 3.300833e-01 250.120109   
min -1.029540e+01 -2.604551e+00 -2.256568e+01 -1.543008e+01 0.000000   
25% -3.171451e-01 -3.269839e-01 -7.083953e-02 -5.295979e-02 5.600000   
50% 1.659350e-02 -5.213911e-02 1.342146e-03 1.124383e-02 22.000000   
75% 3.507156e-01 2.409522e-01 9.104512e-02 7.827995e-02 77.165000   
max 7.519589e+00 3.517346e+00 3.161220e+01 3.384781e+01 25691.160000   
  
 Class   
count 284807.000000   
mean 0.001727   
std 0.041527   
min 0.000000   
25% 0.000000   
50% 0.000000   
75% 0.000000   
max 1.000000   
  
[8 rows x 31 columns]

df.isnull().sum()

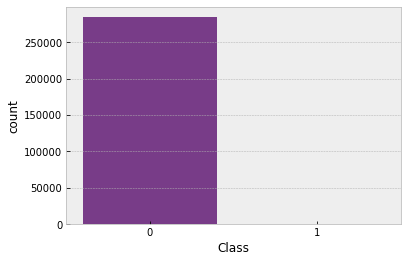
Time 0  
V1 0  
V2 0  
V3 0  
V4 0  
V5 0  
V6 0  
V7 0  
V8 0  
V9 0  
V10 0  
V11 0  
V12 0  
V13 0  
V14 0  
V15 0  
V16 0  
V17 0  
V18 0  
V19 0  
V20 0  
V21 0  
V22 0  
V23 0  
V24 0  
V25 0  
V26 0  
V27 0  
V28 0  
Amount 0  
Class 0  
dtype: int64

df.dtypes

Time float64  
V1 float64  
V2 float64  
V3 float64  
V4 float64  
V5 float64  
V6 float64  
V7 float64  
V8 float64  
V9 float64  
V10 float64  
V11 float64  
V12 float64  
V13 float64  
V14 float64  
V15 float64  
V16 float64  
V17 float64  
V18 float64  
V19 float64  
V20 float64  
V21 float64  
V22 float64  
V23 float64  
V24 float64  
V25 float64  
V26 float64  
V27 float64  
V28 float64  
Amount float64  
Class int64  
dtype: object

sns.countplot(x='Class', data=df, palette='CMRmap')  
print('Non-fraud transactions: {}%'.format(round(df.Class.value\_counts()[0]/len(df)\*100.0,2)))  
print('Fraud transactions: {}%'.format(round(df.Class.value\_counts()[1]/len(df)\*100.0,2)))

Non-fraud transactions: 99.83%  
Fraud transactions: 0.17%



This dataset is severely imbalanced (most of the transactions are non-fraud). So the algorithms are much more likely to classify new observations to the majority class and high accuracy won't tell us anything. To address the problem of imbalanced dataset we can use **undersampling** and **oversampling** data approach techniques. **Oversampling** increases the number of minority class members in the training set. The advantage of **oversampling** is that no information from the original training set is lost unlike in **undersampling**, as all observations from the minority and majority classes are kept. On the other hand, it is prone to overfitting. There is a type of **oversampling** called **SMOTE (Synthetic Minority Oversampling Technique)**, which we are going to use to make our dataset balanced. It creates synthetic points from the minority class

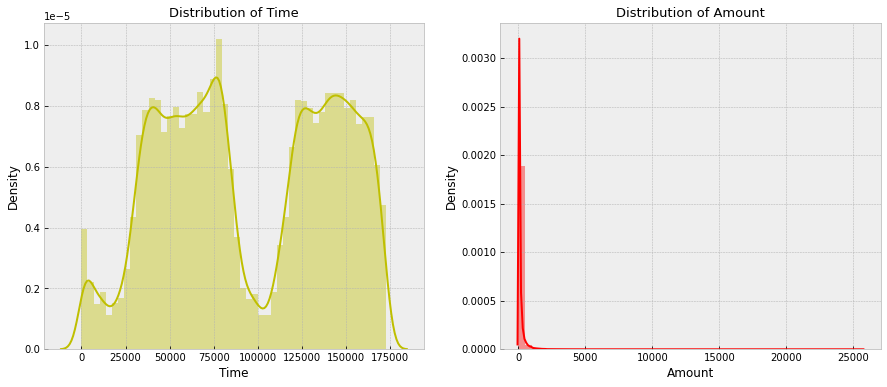
Also we shouldn't use accuracy score as a metric with imbalanced datasets (will be usually high and misleading), instead we should use f1-score, precision/recall score and confusion matrix

* **Recall of fraud cases (sensitivity)** summarizes true positive rate (True positive/True positive + False Negative) - how many cases we got correct out of all the positive ones
* **Recall of non-fraud (specificity)** summarizes true negative rate (True negative/True negative + False positive) - how many cases we got correct out of all the negative ones
* **Precision of fraud cases** (True positive/True positive + False positive) summarizes the accuracy of fraud cases detected - out of all predicted as fraud, how many are correct
* **Precision of non-fraud cases** (True negative/True negative + False negative) summarizes the accuracy of non-fraud cases detected - out of all predicted as non-fraud, how many are correct
* **F1-score** is the harmonic mean of recall and precision

But first let's see the distributions of transaction time and transaction amount to have an idea how skewed these features are. Due to privacy reasons we don't know the names of the other features. All we know is that all of them (except time and amount) went through PCA transformation, which means that they were previously scaled

f, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))  
ax1 = sns.distplot(df['Time'], ax=ax1, color='y')  
ax2 = sns.distplot(df['Amount'], ax=ax2, color='r')  
ax1.set\_title('Distribution of Time', fontsize=13)  
ax2.set\_title('Distribution of Amount', fontsize=13)

Text(0.5, 1.0, 'Distribution of Amount')



To normalize the distribution we are going to use a method called Feature Scaling. In our case it is better to use the Robust Scaler algorithm because it's robust to outliers

from sklearn.preprocessing import RobustScaler  
rs = RobustScaler()  
df['scaled\_amount'] = rs.fit\_transform(df['Amount'].values.reshape(-1,1))  
df['scaled\_time'] = rs.fit\_transform(df['Time'].values.reshape(-1,1))  
df.drop(['Time', 'Amount'], axis=1, inplace=True)

scaled\_amount = df['scaled\_amount']  
scaled\_time = df['scaled\_time']  
df.drop(['scaled\_amount', 'scaled\_time'], axis=1, inplace=True)  
df.insert(0, 'scaled\_amount', scaled\_amount)  
df.insert(0, 'scaled\_time', scaled\_time)  
df.head()

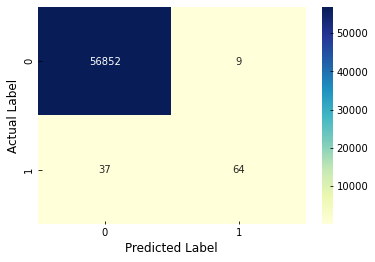
scaled\_time scaled\_amount V1 V2 V3 V4 \  
0 -0.994983 1.783274 -1.359807 -0.072781 2.536347 1.378155   
1 -0.994983 -0.269825 1.191857 0.266151 0.166480 0.448154   
2 -0.994972 4.983721 -1.358354 -1.340163 1.773209 0.379780   
3 -0.994972 1.418291 -0.966272 -0.185226 1.792993 -0.863291   
4 -0.994960 0.670579 -1.158233 0.877737 1.548718 0.403034   
  
 V5 V6 V7 V8 ... V20 V21 V22 \  
0 -0.338321 0.462388 0.239599 0.098698 ... 0.251412 -0.018307 0.277838   
1 0.060018 -0.082361 -0.078803 0.085102 ... -0.069083 -0.225775 -0.638672   
2 -0.503198 1.800499 0.791461 0.247676 ... 0.524980 0.247998 0.771679   
3 -0.010309 1.247203 0.237609 0.377436 ... -0.208038 -0.108300 0.005274   
4 -0.407193 0.095921 0.592941 -0.270533 ... 0.408542 -0.009431 0.798278   
  
 V23 V24 V25 V26 V27 V28 Class   
0 -0.110474 0.066928 0.128539 -0.189115 0.133558 -0.021053 0   
1 0.101288 -0.339846 0.167170 0.125895 -0.008983 0.014724 0   
2 0.909412 -0.689281 -0.327642 -0.139097 -0.055353 -0.059752 0   
3 -0.190321 -1.175575 0.647376 -0.221929 0.062723 0.061458 0   
4 -0.137458 0.141267 -0.206010 0.502292 0.219422 0.215153 0   
  
[5 rows x 31 columns]

So time and amount are now scaled as well

# Logistic Regression without SMOTE

from sklearn.model\_selection import train\_test\_split as holdout  
x = np.array(df.iloc[:, df.columns != 'Class'])  
y = np.array(df.iloc[:, df.columns == 'Class'])  
x\_train, x\_test, y\_train, y\_test = holdout(x, y, test\_size=0.2, random\_state=0)  
  
from sklearn.linear\_model import LogisticRegression  
from sklearn.metrics import confusion\_matrix, precision\_recall\_curve, classification\_report, precision\_score, recall\_score, accuracy\_score  
logreg = LogisticRegression()  
logreg.fit(x\_train, y\_train)  
y\_pred = logreg.predict(x\_test)  
cnf\_matrix = confusion\_matrix(y\_test, y\_pred)  
  
sns.heatmap(pd.DataFrame(cnf\_matrix), annot=True, cmap="YlGnBu", fmt='g')  
plt.ylabel('Actual Label')  
plt.xlabel('Predicted Label')  
  
labels = ['Non-fraud', 'Fraud']  
print(classification\_report(y\_test, y\_pred, target\_names=labels))

precision recall f1-score support  
  
 Non-fraud 1.00 1.00 1.00 56861  
 Fraud 0.88 0.63 0.74 101  
  
 accuracy 1.00 56962  
 macro avg 0.94 0.82 0.87 56962  
weighted avg 1.00 1.00 1.00 56962

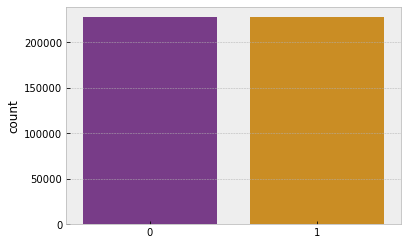


As we can see this is not a good model, because it is biased towards majority class and the recall in minority class is not as high as disired

from imblearn.over\_sampling import SMOTE  
  
print("Transaction Number x\_train dataset: ", x\_train.shape)  
print("Transaction Number y\_train dataset: ", y\_train.shape)  
print("Transaction Number x\_test dataset: ", x\_test.shape)  
print("Transaction Number y\_test dataset: ", y\_test.shape)  
  
print("Before OverSampling, counts of label '1': {}".format(sum(y\_train==1)))  
print("Before OverSampling, counts of label '0': {} \n".format(sum(y\_train==0)))  
  
sm = SMOTE(random\_state=2)  
x\_train\_s, y\_train\_s = sm.fit\_resample(x\_train, y\_train.ravel())  
  
print('After OverSampling, the shape of train\_x: {}'.format(x\_train\_s.shape))  
print('After OverSampling, the shape of train\_y: {} \n'.format(y\_train\_s.shape))  
  
print("After OverSampling, counts of label '1', %: {}".format(sum(y\_train\_s==1)/len(y\_train\_s)\*100.0,2))  
print("After OverSampling, counts of label '0', %: {}".format(sum(y\_train\_s==0)/len(y\_train\_s)\*100.0,2))  
  
sns.countplot(x=y\_train\_s, data=df, palette='CMRmap')

Transaction Number x\_train dataset: (227845, 30)  
Transaction Number y\_train dataset: (227845, 1)  
Transaction Number x\_test dataset: (56962, 30)  
Transaction Number y\_test dataset: (56962, 1)  
Before OverSampling, counts of label '1': [391]  
Before OverSampling, counts of label '0': [227454]   
  
After OverSampling, the shape of train\_x: (454908, 30)  
After OverSampling, the shape of train\_y: (454908,)   
  
After OverSampling, counts of label '1', %: 50.0  
After OverSampling, counts of label '0', %: 50.0

<AxesSubplot:ylabel='count'>

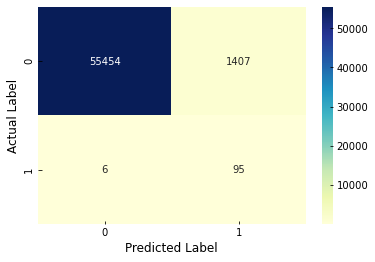


Now the dataset is balanced, so we can build a Logistic Regression model with SMOTE. One important thing to point out here is that we used SMOTE after cross validation in order to avoid data leakage problem and hence overfitting

# Logistic Regression with SMOTE

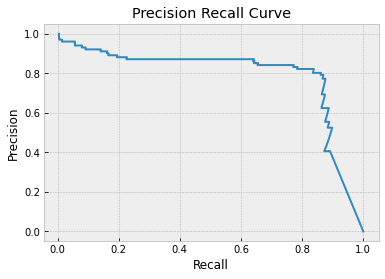
logreg = LogisticRegression()  
logreg.fit(x\_train\_s, y\_train\_s)  
y\_pred = logreg.predict(x\_test)  
cnf\_matrix = confusion\_matrix(y\_test, y\_pred)  
  
sns.heatmap(pd.DataFrame(cnf\_matrix), annot=True, cmap="YlGnBu", fmt='g')  
plt.ylabel('Actual Label')  
plt.xlabel('Predicted Label')  
  
print(classification\_report(y\_test, y\_pred))

precision recall f1-score support  
  
 0 1.00 0.98 0.99 56861  
 1 0.06 0.94 0.12 101  
  
 accuracy 0.98 56962  
 macro avg 0.53 0.96 0.55 56962  
weighted avg 1.00 0.98 0.99 56962



y\_pred\_prob = logreg.predict\_proba(x\_test)[:,1]  
precision, recall, thresholds = precision\_recall\_curve(y\_test, y\_pred\_prob)  
plt.plot(precision, recall)  
plt.xlabel('Recall')  
plt.ylabel('Precision')  
plt.title('Precision Recall Curve')

Text(0.5, 1.0, 'Precision Recall Curve')



We got a high recall which means our model is able to detect the highest number of fraud transactions, while the precision is very low which is not good because it means that the model classifies a lot of non-fraud transactions as fraud. The customers of a financial institution are not going to be satisfied with that fact and may even stop using the service of that financial institution. So in this case it's also important to have a high precision, which we are going to try to achieve with Random Forest

# Displaying the predictions

y\_pred

array([0, 0, 0, ..., 0, 0, 0], dtype=int64)

# Predicitng an arbitrary input data

logreg.predict([[-0.63699057, 6.74100468, -1.53032404, -0.23597764, 1.85501924,  
 1.25543555, -1.684011 , 2.0743704 , 1.77938361, -0.20089071,  
 -0.10246641, -0.07156013, 1.01670848, 0.2247269 , -0.13841633,  
 -0.38018538, 0.77750365, -0.05996581, -0.28821343, 0.64818512,  
 0.78002443, -0.27823953, 0.03182868, 0.85273687, -0.42126432,  
 -0.22027806, 0.19826107, -0.05929797, 0.10316612, -0.2738302 ]])

array([0], dtype=int64)

# The prediction obtained is 0

# The End