Note: There are still aspects of this kernel that will be subjected to changes. I've noticed a recent increase of interest towards this kernel so I will focus more on the steps I took and why I took them to make it clear why I took those steps.

If you liked my work, please upvote this kernel since it will keep me motivated to perform more in-depth reserach towards this subject and will look for more efficient ways so that our models are able to detect more accurately both fraud and non-fraud transactions.

In this kernel we will use various predictive models to see how accurate they are in detecting whether a transaction is a normal payment or a fraud. As described in the dataset, the features are scaled and the names of the features are not shown due to privacy reasons. Nevertheless, we can still analyze some important aspects of the dataset. Let's start!

Gather Sense of Our Data:

The first thing we must do is gather a basic sense of our data. Remember, except for the transaction and amount we dont know what the other columns are (due to privacy reasons). The only thing we know, is that those columns that are unknown have been scaled already.

```python
# This Python 3 environment comes with many helpful analytics
libraries installed
# It is defined by the kaggle/python docker image:
https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load in

# Imported Libraries

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA, TruncatedSVD
import matplotlib.patches as mpatches
import time

# Classifier Libraries
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
import collections


# Other Libraries
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from imblearn.pipeline import make_pipeline as
imbalanced_make_pipeline
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import NearMiss
from imblearn.metrics import classification_report_imbalanced
from sklearn.metrics import precision_score, recall_score, f1_score,
roc_auc_score, accuracy_score, classification_report
from collections import Counter
from sklearn.model_selection import KFold, StratifiedKFold
import warnings
warnings.filterwarnings("ignore")


df = pd.read_csv('../input/creditcard.csv')
df.head()
```

```
/opt/conda/lib/python3.6/site-packages/sklearn/externals/six.py:31:
DeprecationWarning: The module is deprecated in version 0.21 and will
be removed in version 0.23 since we've dropped support for Python 2.7.
Please rely on the official version of six
```

```
(https://pypi.org/project/six/).
  "(https://pypi.org/project/six/).", DeprecationWarning)
```

```
   Time        V1        V2        V3  ...         V27       V28
Amount  Class
0   0.0 -1.359807 -0.072781  2.536347  ...    0.133558 -0.021053
149.62      0
1   0.0  1.191857  0.266151  0.166480  ...   -0.008983  0.014724
2.69      0
2   1.0 -1.358354 -1.340163  1.773209  ...   -0.055353 -0.059752
378.66      0
3   1.0 -0.966272 -0.185226  1.792993  ...    0.062723  0.061458
123.50      0
4   2.0 -1.158233  0.877737  1.548718  ...    0.219422  0.215153
69.99      0

[5 rows x 31 columns]
```

```
df.describe()
```

```
                Time            V1      ...              Amount
Class
count  284807.000000  2.848070e+05      ...        284807.000000
284807.000000
mean    94813.859575  3.919560e-15      ...            88.349619
0.001727
std     47488.145955  1.958696e+00      ...           250.120109
0.041527
min         0.000000 -5.640751e+01      ...             0.000000
0.000000
25%     54201.500000 -9.203734e-01      ...             5.600000
0.000000
50%     84692.000000  1.810880e-02      ...            22.000000
0.000000
75%    139320.500000  1.315642e+00      ...            77.165000
0.000000
max    172792.000000  2.454930e+00      ...         25691.160000
1.000000

[8 rows x 31 columns]
```

```
# Good No Null Values!
df.isnull().sum().max()
```

```
0
```

```
df.columns
```

```
Index(['Time', 'V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9',
'V10',
       'V11', 'V12', 'V13', 'V14', 'V15', 'V16', 'V17', 'V18', 'V19',
```

```
'V20',
       'V21', 'V22', 'V23', 'V24', 'V25', 'V26', 'V27', 'V28',
'Amount',
       'Class'],
      dtype='object')
```

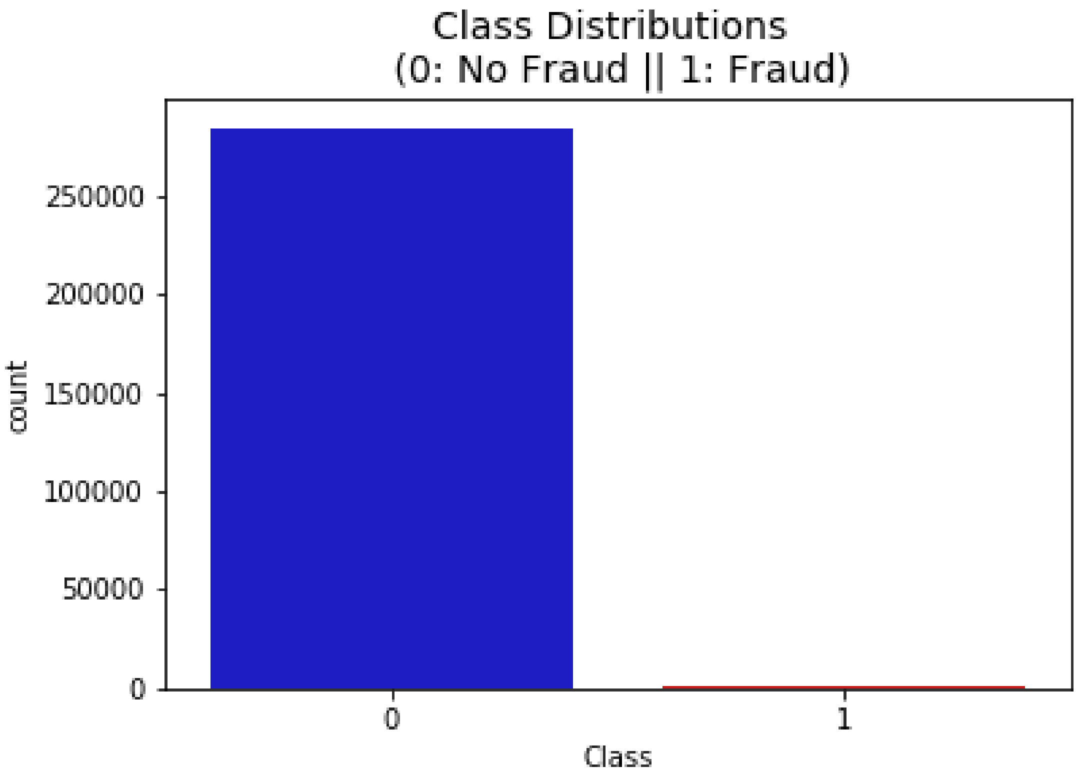
```
# The classes are heavily skewed we need to solve this issue later.
print('No Frauds', round(df['Class'].value_counts()[0]/len(df) *
100,2), '% of the dataset')
print('Frauds', round(df['Class'].value_counts()[1]/len(df) * 100,2),
'% of the dataset')
```

```
No Frauds 99.83 % of the dataset
Frauds 0.17 % of the dataset
```

Note: Notice how imbalanced is our original dataset! Most of the transactions are non-fraud. If we use this dataframe as the base for our predictive models and analysis we might get a lot of errors and our algorithms will probably overfit since it will "assume" that most transactions are not fraud. But we don't want our model to assume, we want our model to detect patterns that give signs of fraud!

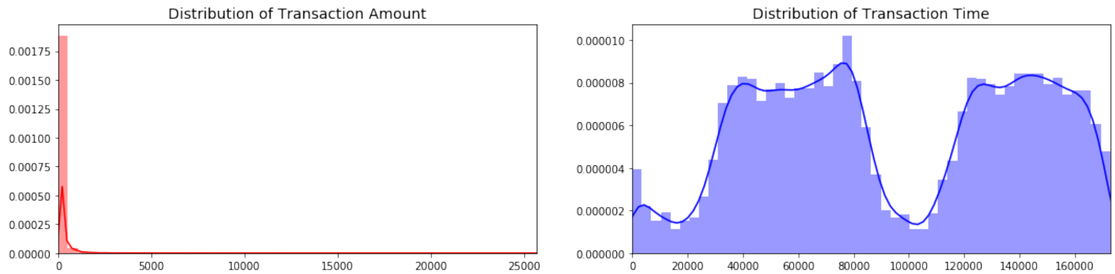
```
colors = ["#0101DF", "#DF0101"]

sns.countplot('Class', data=df, palette=colors)
plt.title('Class Distributions \n (0: No Fraud || 1: Fraud)',
fontsize=14)
```

```
Text(0.5, 1.0, 'Class Distributions \n (0: No Fraud || 1: Fraud)')
```



Distributions: By seeing the distributions we can have an idea how skewed are these features, we can also see further distributions of the other features. There are techniques that can help the distributions be less skewed which will be implemented in this notebook in the future.

```
fig, ax = plt.subplots(1, 2, figsize=(18,4))

amount_val = df['Amount'].values
time_val = df['Time'].values

sns.distplot(amount_val, ax=ax[0], color='r')
ax[0].set_title('Distribution of Transaction Amount', fontsize=14)
ax[0].set_xlim([min(amount_val), max(amount_val)])

sns.distplot(time_val, ax=ax[1], color='b')
ax[1].set_title('Distribution of Transaction Time', fontsize=14)
ax[1].set_xlim([min(time_val), max(time_val)])



plt.show()
```



In this phase of our kernel, we will first scale the columns comprise of Time and Amount . Time and amount should be scaled as the other columns. On the other hand, we need to also create a sub sample of the dataframe in order to have an equal amount of Fraud and Non-Fraud cases, helping our algorithms better understand patterns that determines whether a transaction is a fraud or not.

In this scenario, our subsample will be a dataframe with a 50/50 ratio of fraud and non-fraud transactions. Meaning our sub-sample will have the same amount of fraud and non fraud transactions.

In the beginning of this notebook we saw that the original dataframe was heavily imbalanced! Using the original dataframe will cause the following issues: Overfitting: Our classification models will assume that in most cases there are no frauds! What we want for our model is to be certain when a fraud occurs. Wrong Correlations: Although we don't know what the "V" features stand for, it will be useful to understand how each of this features influence the result (Fraud or No Fraud) by having an imbalance dataframe we are not able to see the true correlations between the class and features.

```
# Since most of our data has already been scaled we should scale the columns that are left to scale (Amount and Time)
from sklearn.preprocessing import StandardScaler, RobustScaler

# RobustScaler is less prone to outliers.

std_scaler = StandardScaler()
rob_scaler = RobustScaler()

df['scaled_amount'] = rob_scaler.fit_transform(df['Amount'].values.reshape(-1,1))
df['scaled_time'] = rob_scaler.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(1, 'scaled_time', scaled_time)
```

```
# Amount and Time are Scaled!

df.head()

   scaled_amount  scaled_time        V1  ...         V27       V28  
Class
0       1.783274    -0.994983 -1.359807  ...    0.133558 -0.021053      
0
1      -0.269825    -0.994983  1.191857  ...   -0.008983  0.014724      
0
2       4.983721    -0.994972 -1.358354  ...   -0.055353 -0.059752      
0
3       1.418291    -0.994972 -0.966272  ...    0.062723  0.061458      
0
4       0.670579    -0.994960 -1.158233  ...    0.219422  0.215153      
0

[5 rows x 31 columns]
```

Splitting the Data (Original DataFrame)

Before proceeding with the Random UnderSampling technique we have to separate the orginal dataframe. Why? for testing purposes, remember although we are splitting the data when implementing Random UnderSampling or OverSampling techniques, we want to test our models on the original testing set not on the testing set created by either of these techniques. The main goal is to fit the model either with the dataframes that were undersample and oversample (in order for our models to detect the patterns), and test it on the original testing set.

```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit

print('No Frauds', round(df['Class'].value_counts()[0]/len(df) * 
100,2), '% of the dataset')
print('Frauds', round(df['Class'].value_counts()[1]/len(df) * 100,2), 
'% of the dataset')

X = df.drop('Class', axis=1)
y = df['Class']

sss = StratifiedKFold(n_splits=5, random_state=None, shuffle=False)

for train_index, test_index in sss.split(X, y):
    print("Train:", train_index, "Test:", test_index)
    original_Xtrain, original_Xtest = X.iloc[train_index], 
X.iloc[test_index]
    original_ytrain, original_ytest = y.iloc[train_index], 
y.iloc[test_index]
```

```
# We already have X_train and y_train for undersample data thats why I
am using original to distinguish and to not overwrite these variables.
# original_Xtrain, original_Xtest, original_ytrain, original_ytest =
train_test_split(X, y, test_size=0.2, random_state=42)

# Check the Distribution of the labels


# Turn into an array
original_Xtrain = original_Xtrain.values
original_Xtest = original_Xtest.values
original_ytrain = original_ytrain.values
original_ytest = original_ytest.values

# See if both the train and test label distribution are similarly
distributed
train_unique_label, train_counts_label = np.unique(original_ytrain,
return_counts=True)
test_unique_label, test_counts_label = np.unique(original_ytest,
return_counts=True)
print('-' * 100)

print('Label Distributions: \n')
print(train_counts_label/ len(original_ytrain))
print(test_counts_label/ len(original_ytest))

No Frauds 99.83 % of the dataset
Frauds 0.17 % of the dataset
Train: [ 30473  30496  31002 ... 284804 284805 284806] Test: [    0
1     2 ... 57017 57018 57019]
Train: [     0      1      2 ... 284804 284805 284806] Test: [ 30473
30496  31002 ... 113964 113965 113966]
Train: [     0      1      2 ... 284804 284805 284806] Test: [ 81609
82400  83053 ... 170946 170947 170948]
Train: [     0      1      2 ... 284804 284805 284806] Test: [150654
150660 150661 ... 227866 227867 227868]
Train: [     0      1      2 ... 227866 227867 227868] Test: [212516
212644 213092 ... 284804 284805 284806]
----------------------------------------------------------------------
------------------------------
Label Distributions: 

[0.99827076 0.00172924]
[0.99827952 0.00172048]
```

Random Under-Sampling:

In this phase of the project we will implement *"Random Under Sampling"* which basically consists of removing data in order to have a more balanced dataset and thus avoiding our models to overfitting.

Steps:

Note: The main issue with "Random Under-Sampling" is that we run the risk that our classification models will not perform as accurate as we would like to since there is a great deal of information loss (bringing 492 non-fraud transaction from 284,315 non-fraud transaction)

```
# Since our classes are highly skewed we should make them equivalent
in order to have a normal distribution of the classes.

# Lets shuffle the data before creating the subsamples

df = df.sample(frac=1)

# amount of fraud classes 492 rows.
fraud_df = df.loc[df['Class'] == 1]
non_fraud_df = df.loc[df['Class'] == 0][:492]

normal_distributed_df = pd.concat([fraud_df, non_fraud_df])

# Shuffle dataframe rows
new_df = normal_distributed_df.sample(frac=1, random_state=42)

new_df.head()

        scaled_amount  scaled_time         V1  ...        V27
V28  Class
35719        0.368616    -0.545789  -0.887048  ...  -0.240767
0.153636      0
154670       1.145812     0.209084  -2.296987  ...   0.969582
0.335041      1
59856       -0.125900    -0.418872   1.205206  ...   0.072760
0.031139      0
14170        1.089779    -0.698951 -15.903635  ...   1.688136
0.527831      1
8842        -0.307413    -0.852912  -4.696795  ...  -1.508458
0.608075      1

[5 rows x 31 columns]
```

Equally Distributing and Correlating:

Now that we have our dataframe correctly balanced, we can go further with our analysis and data preprocessing.

```
print('Distribution of the Classes in the subsample dataset')
print(new_df['Class'].value_counts()/len(new_df))



sns.countplot('Class', data=new_df, palette=colors)
plt.title('Equally Distributed Classes', fontsize=14)
plt.show()
```

```
Distribution of the Classes in the subsample dataset
1    0.5
0    0.5
Name: Class, dtype: float64
```

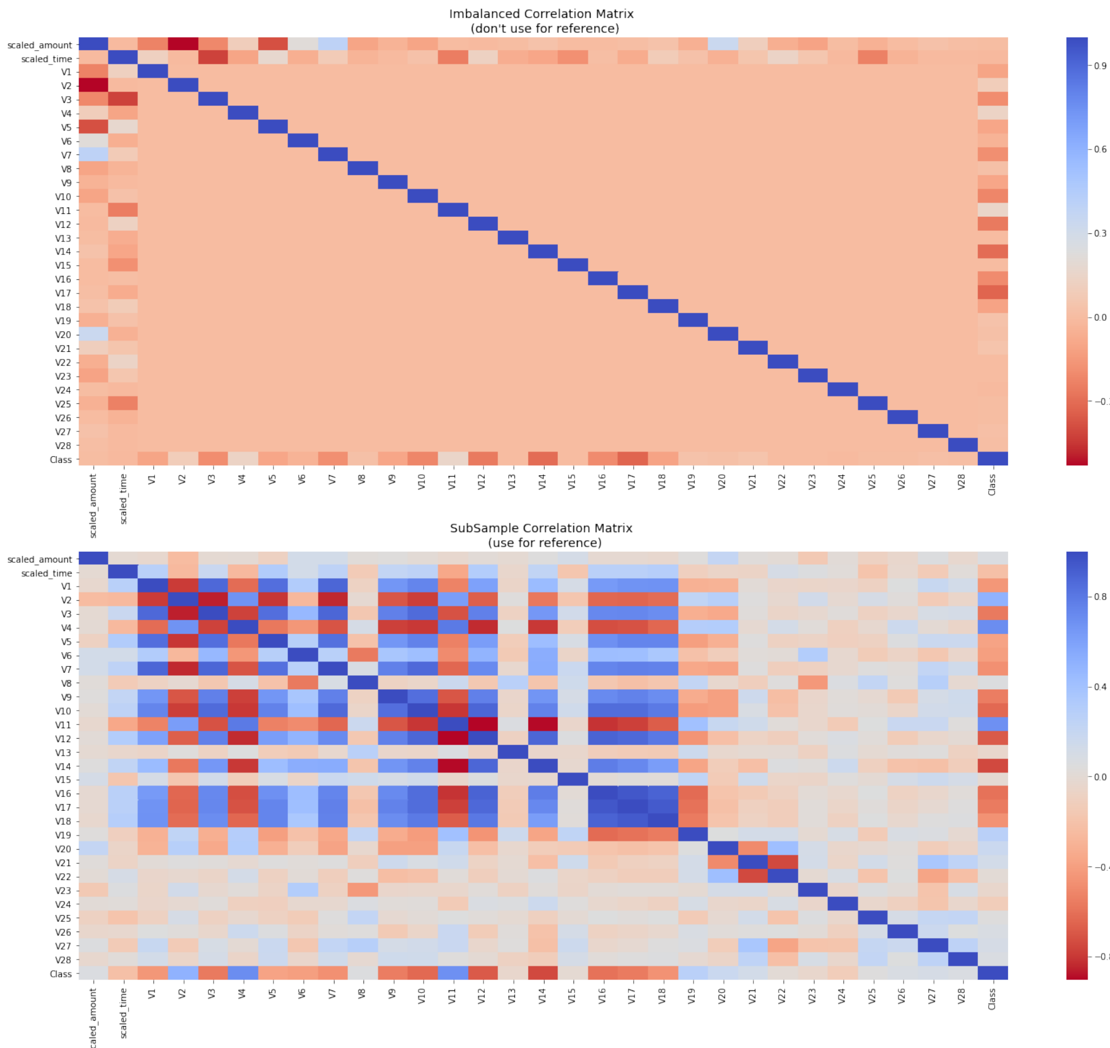
Correlation matrices are the essence of understanding our data. We want to know if there are features that influence heavily in whether a specific transaction is a fraud. However, it is important that we use the correct dataframe (subsample) in order for us to see which features have a high positive or negative correlation with regards to fraud transactions.

Summary and Explanation:

Note: We have to make sure we use the subsample in our correlation matrix or else our correlation matrix will be affected by the high imbalance between our classes. This occurs due to the high class imbalance in the original dataframe.

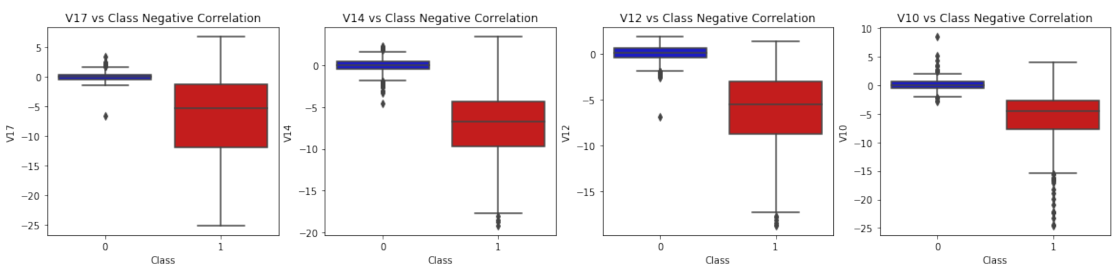
```
# Make sure we use the subsample in our correlation
```

```python
f, (ax1, ax2) = plt.subplots(2, 1, figsize=(24,20))

# Entire DataFrame
corr = df.corr()
sns.heatmap(corr, cmap='coolwarm_r', annot_kws={'size':20}, ax=ax1)
ax1.set_title("Imbalanced Correlation Matrix \n (don't use for reference)", fontsize=14)


sub_sample_corr = new_df.corr()
sns.heatmap(sub_sample_corr, cmap='coolwarm_r', annot_kws={'size':20}, ax=ax2)
ax2.set_title('SubSample Correlation Matrix \n (use for reference)', fontsize=14)
plt.show()
```



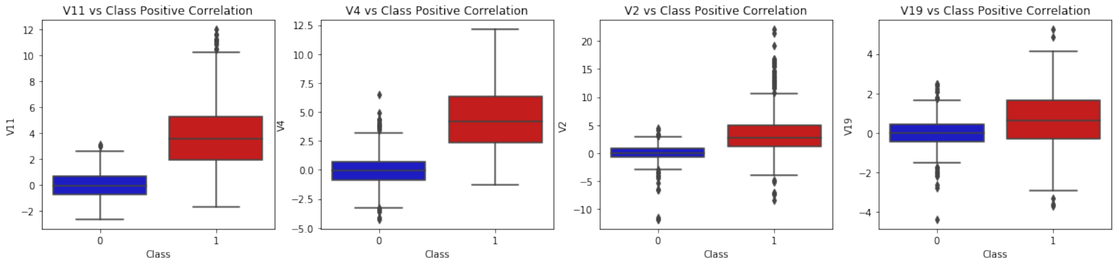

```python
f, axes = plt.subplots(ncols=4, figsize=(20,4))
```

```
# Negative Correlations with our Class (The lower our feature value
the more likely it will be a fraud transaction)
sns.boxplot(x="Class", y="V17", data=new_df, palette=colors,
ax=axes[0])
axes[0].set_title('V17 vs Class Negative Correlation')

sns.boxplot(x="Class", y="V14", data=new_df, palette=colors,
ax=axes[1])
axes[1].set_title('V14 vs Class Negative Correlation')


sns.boxplot(x="Class", y="V12", data=new_df, palette=colors,
ax=axes[2])
axes[2].set_title('V12 vs Class Negative Correlation')


sns.boxplot(x="Class", y="V10", data=new_df, palette=colors,
ax=axes[3])
axes[3].set_title('V10 vs Class Negative Correlation')

plt.show()
```




```
f, axes = plt.subplots(ncols=4, figsize=(20,4))

# Positive correlations (The higher the feature the probability
increases that it will be a fraud transaction)
sns.boxplot(x="Class", y="V11", data=new_df, palette=colors,
ax=axes[0])
axes[0].set_title('V11 vs Class Positive Correlation')

sns.boxplot(x="Class", y="V4", data=new_df, palette=colors,
ax=axes[1])
axes[1].set_title('V4 vs Class Positive Correlation')


sns.boxplot(x="Class", y="V2", data=new_df, palette=colors,
ax=axes[2])
axes[2].set_title('V2 vs Class Positive Correlation')


sns.boxplot(x="Class", y="V19", data=new_df, palette=colors,
```

```
ax=axes[3])
axes[3].set_title('V19 vs Class Positive Correlation')

plt.show()
```



Anomaly Detection:

Our main aim in this section is to remove "extreme outliers" from features that have a high correlation with our classes. This will have a positive impact on the accuracy of our models.

Interquartile Range Method:

Outlier Removal Tradeoff:

We have to be careful as to how far do we want the threshold for removing outliers. We determine the threshold by multiplying a number (ex: 1.5) by the (Interquartile Range). The higher this threshold is, the less outliers will detect (multiplying by a higher number ex: 3), and the lower this threshold is the more outliers it will detect.

The Tradeoff: The lower the threshold the more outliers it will remove however, we want to focus more on "extreme outliers" rather than just outliers. Why? because we might run the risk of information loss which will cause our models to have a lower accuracy. You can play with this threshold and see how it affects the accuracy of our classification models.

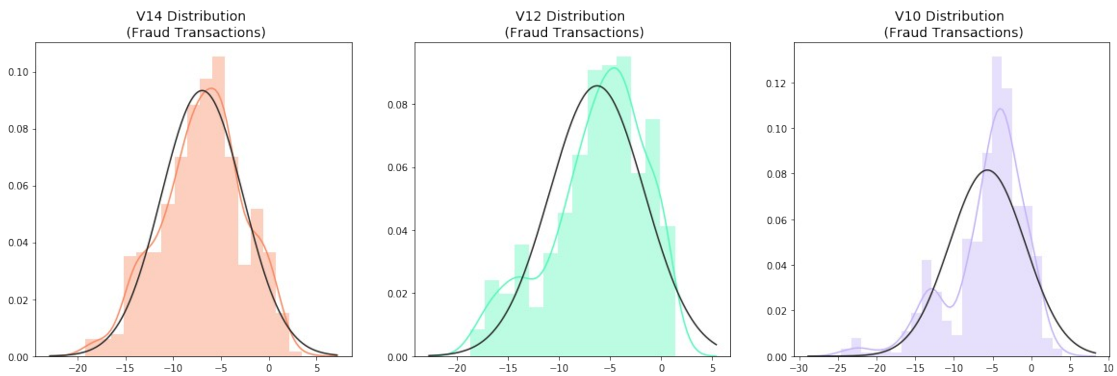
Summary:

Note: After implementing outlier reduction our accuracy has been improved by over 3%! Some outliers can distort the accuracy of our models but remember, we have to avoid an extreme amount of information loss or else our model runs the risk of underfitting.

Reference: More information on Interquartile Range Method: How to Use Statistics to Identify Outliers in Data by Jason Brownless (Machine Learning Mastery blog)

```
from scipy.stats import norm

f, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(20, 6))

v14_fraud_dist = new_df['V14'].loc[new_df['Class'] == 1].values
sns.distplot(v14_fraud_dist,ax=ax1, fit=norm, color='#FB8861')
ax1.set_title('V14 Distribution \n (Fraud Transactions)', fontsize=14)
```

```
v12_fraud_dist = new_df['V12'].loc[new_df['Class'] == 1].values
sns.distplot(v12_fraud_dist,ax=ax2, fit=norm, color='#56F9BB')
ax2.set_title('V12 Distribution \n (Fraud Transactions)', fontsize=14)


v10_fraud_dist = new_df['V10'].loc[new_df['Class'] == 1].values
sns.distplot(v10_fraud_dist,ax=ax3, fit=norm, color='#C5B3F9')
ax3.set_title('V10 Distribution \n (Fraud Transactions)', fontsize=14)

plt.show()
```



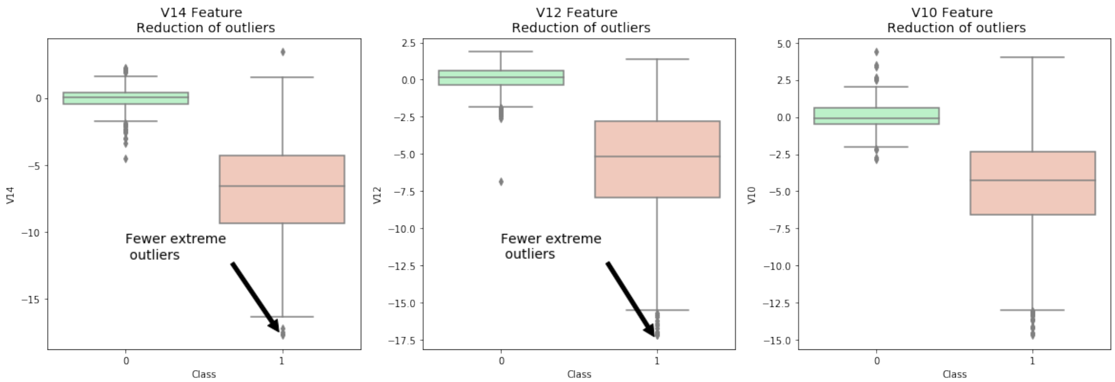

```
# # -----> V14 Removing Outliers (Highest Negative Correlated with
Labels)
v14_fraud = new_df['V14'].loc[new_df['Class'] == 1].values
q25, q75 = np.percentile(v14_fraud, 25), np.percentile(v14_fraud, 75)
print('Quartile 25: {} | Quartile 75: {}'.format(q25, q75))
v14_iqr = q75 - q25
print('iqr: {}'.format(v14_iqr))

v14_cut_off = v14_iqr * 1.5
v14_lower, v14_upper = q25 - v14_cut_off, q75 + v14_cut_off
print('Cut Off: {}'.format(v14_cut_off))
print('V14 Lower: {}'.format(v14_lower))
print('V14 Upper: {}'.format(v14_upper))

outliers = [x for x in v14_fraud if x < v14_lower or x > v14_upper]
print('Feature V14 Outliers for Fraud Cases:
{}'.format(len(outliers)))
print('V10 outliers:{}'.format(outliers))

new_df = new_df.drop(new_df[(new_df['V14'] > v14_upper) |
(new_df['V14'] < v14_lower)].index)
print('----' * 44)

# -----> V12 removing outliers from fraud transactions
v12_fraud = new_df['V12'].loc[new_df['Class'] == 1].values
q25, q75 = np.percentile(v12_fraud, 25), np.percentile(v12_fraud, 75)
```

```
v12_iqr = q75 - q25

v12_cut_off = v12_iqr * 1.5
v12_lower, v12_upper = q25 - v12_cut_off, q75 + v12_cut_off
print('V12 Lower: {}'.format(v12_lower))
print('V12 Upper: {}'.format(v12_upper))
outliers = [x for x in v12_fraud if x < v12_lower or x > v12_upper]
print('V12 outliers: {}'.format(outliers))
print('Feature V12 Outliers for Fraud Cases:
{}'.format(len(outliers)))
new_df = new_df.drop(new_df[(new_df['V12'] > v12_upper) |
(new_df['V12'] < v12_lower)].index)
print('Number of Instances after outliers removal:
{}'.format(len(new_df)))
print('----' * 44)


# Removing outliers V10 Feature
v10_fraud = new_df['V10'].loc[new_df['Class'] == 1].values
q25, q75 = np.percentile(v10_fraud, 25), np.percentile(v10_fraud, 75)
v10_iqr = q75 - q25

v10_cut_off = v10_iqr * 1.5
v10_lower, v10_upper = q25 - v10_cut_off, q75 + v10_cut_off
print('V10 Lower: {}'.format(v10_lower))
print('V10 Upper: {}'.format(v10_upper))
outliers = [x for x in v10_fraud if x < v10_lower or x > v10_upper]
print('V10 outliers: {}'.format(outliers))
print('Feature V10 Outliers for Fraud Cases:
{}'.format(len(outliers)))
new_df = new_df.drop(new_df[(new_df['V10'] > v10_upper) |
(new_df['V10'] < v10_lower)].index)
print('Number of Instances after outliers removal:
{}'.format(len(new_df)))
```

```
Quartile 25: -9.692722964972385 | Quartile 75: -4.282820849486866
iqr: 5.409902115485519
Cut Off: 8.114853173228278
V14 Lower: -17.807576138200663
V14 Upper: 3.8320323237414122
Feature V14 Outliers for Fraud Cases: 4
V10 outliers:[-19.2143254902614, -18.8220867423816, -18.4937733551053,
-18.0499976898559396]
----------------------------------------------------------------------
----------------------------------------------------------------------
------------------------------------
V12 Lower: -17.3430371579634
V12 Upper: 5.776973384895937
V12 outliers: [-18.683714633344298, -18.047596570821604, -
18.4311310279993, -18.553697009645802]
```

```
Feature V12 Outliers for Fraud Cases: 4
Number of Instances after outliers removal: 976
----------------------------------------------------------------------
----------------------------------------------------------------------
-----------------------------------
V10 Lower: -14.89885463232024
V10 Upper: 4.920334958342141
V10 outliers: [-24.403184969972802, -18.9132433348732, -
15.124162814494698, -16.3035376590131, -15.2399619587112, -
15.1237521803455, -14.9246547735487, -16.6496281595399, -
18.2711681738888, -24.5882624372475, -15.346098846877501, -
20.949191554361104, -15.2399619587112, -23.2282548357516, -
15.2318333653018, -22.1870885620007, -17.141513641289198, -
19.836148851696, -22.1870885620007, -16.6011969664137, -
16.7460441053944, -15.563791338730098, -14.9246547735487, -
16.2556117491401, -22.1870885620007, -15.563791338730098, -
22.1870885620007]
Feature V10 Outliers for Fraud Cases: 27
Number of Instances after outliers removal: 947
```

```
f,(ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20,6))

colors = ['#B3F9C5', '#f9c5b3']
# Boxplots with outliers removed
# Feature V14
sns.boxplot(x="Class", y="V14", data=new_df,ax=ax1, palette=colors)
ax1.set_title("V14 Feature \n Reduction of outliers", fontsize=14)
ax1.annotate('Fewer extreme \n outliers', xy=(0.98, -17.5), xytext=(0,
-12),
            arrowprops=dict(facecolor='black'),
            fontsize=14)

# Feature 12
sns.boxplot(x="Class", y="V12", data=new_df, ax=ax2, palette=colors)
ax2.set_title("V12 Feature \n Reduction of outliers", fontsize=14)
ax2.annotate('Fewer extreme \n outliers', xy=(0.98, -17.3), xytext=(0,
-12),
            arrowprops=dict(facecolor='black'),
            fontsize=14)

# Feature V10
sns.boxplot(x="Class", y="V10", data=new_df, ax=ax3, palette=colors)
ax3.set_title("V10 Feature \n Reduction of outliers", fontsize=14)
ax3.annotate('Fewer extreme \n outliers', xy=(0.95, -16.5), xytext=(0,
-12),
            arrowprops=dict(facecolor='black'),
            fontsize=14)


plt.show()
```

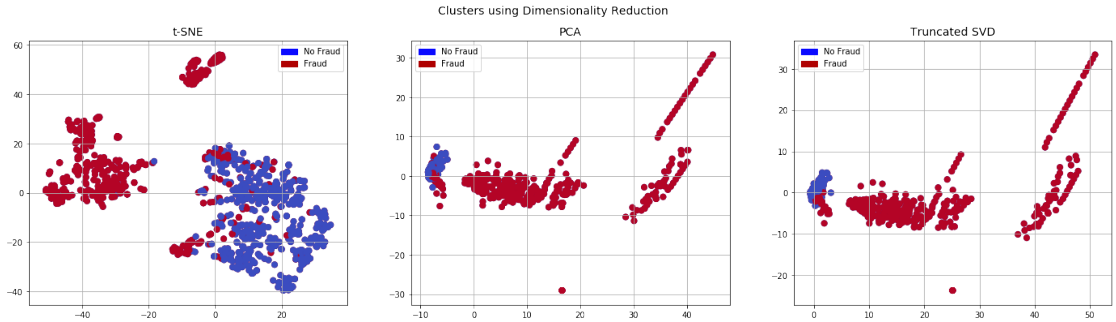


In order to understand this algorithm you have to understand the following terms:
Euclidean Distance   Conditional Probability Normal and T-Distribution Plots

Note: If you want a simple instructive video look at StatQuest: t-SNE, Clearly Explained by Joshua Starmer

```
# New_df is from the random undersample data (fewer instances)
X = new_df.drop('Class', axis=1)
y = new_df['Class']


# T-SNE Implementation
t0 = time.time()
X_reduced_tsne = TSNE(n_components=2,
random_state=42).fit_transform(X.values)
t1 = time.time()
print("T-SNE took {:.2} s".format(t1 - t0))

# PCA Implementation
t0 = time.time()
X_reduced_pca = PCA(n_components=2,
random_state=42).fit_transform(X.values)
t1 = time.time()
print("PCA took {:.2} s".format(t1 - t0))

# TruncatedSVD
t0 = time.time()
X_reduced_svd = TruncatedSVD(n_components=2, algorithm='randomized',
random_state=42).fit_transform(X.values)
t1 = time.time()
print("Truncated SVD took {:.2} s".format(t1 - t0))

T-SNE took 7.0 s
PCA took 0.032 s
Truncated SVD took 0.0051 s
```

```
f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(24,6))
# labels = ['No Fraud', 'Fraud']
f.suptitle('Clusters using Dimensionality Reduction', fontsize=14)


blue_patch = mpatches.Patch(color='#0A0AFF', label='No Fraud')
red_patch = mpatches.Patch(color='#AF0000', label='Fraud')


# t-SNE scatter plot
ax1.scatter(X_reduced_tsne[:,0], X_reduced_tsne[:,1], c=(y == 0),
cmap='coolwarm', label='No Fraud', linewidths=2)
ax1.scatter(X_reduced_tsne[:,0], X_reduced_tsne[:,1], c=(y == 1),
cmap='coolwarm', label='Fraud', linewidths=2)
ax1.set_title('t-SNE', fontsize=14)

ax1.grid(True)

ax1.legend(handles=[blue_patch, red_patch])


# PCA scatter plot
ax2.scatter(X_reduced_pca[:,0], X_reduced_pca[:,1], c=(y == 0),
cmap='coolwarm', label='No Fraud', linewidths=2)
ax2.scatter(X_reduced_pca[:,0], X_reduced_pca[:,1], c=(y == 1),
cmap='coolwarm', label='Fraud', linewidths=2)
ax2.set_title('PCA', fontsize=14)

ax2.grid(True)

ax2.legend(handles=[blue_patch, red_patch])

# TruncatedSVD scatter plot
ax3.scatter(X_reduced_svd[:,0], X_reduced_svd[:,1], c=(y == 0),
cmap='coolwarm', label='No Fraud', linewidths=2)
ax3.scatter(X_reduced_svd[:,0], X_reduced_svd[:,1], c=(y == 1),
cmap='coolwarm', label='Fraud', linewidths=2)
ax3.set_title('Truncated SVD', fontsize=14)

ax3.grid(True)

ax3.legend(handles=[blue_patch, red_patch])

plt.show()
```



In this section we will train four types of classifiers and decide which classifier will be more effective in detecting fraud transactions. Before we have to split our data into training and testing sets and separate the features from the labels.

Summary:

Learning Curves:

```
# Undersampling before cross validating (prone to overfit)
X = new_df.drop('Class', axis=1)
y = new_df['Class']

# Our data is already scaled we should split our training and test
sets
from sklearn.model_selection import train_test_split

# This is explicitly used for undersampling.
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Turn the values into an array for feeding the classification
algorithms.
X_train = X_train.values
X_test = X_test.values
y_train = y_train.values
y_test = y_test.values

# Let's implement simple classifiers

classifiers = {
    "LogisiticRegression": LogisticRegression(),
    "KNearest": KNeighborsClassifier(),
    "Support Vector Classifier": SVC(),
    "DecisionTreeClassifier": DecisionTreeClassifier()
}

# Wow our scores are getting even high scores even when applying cross
validation.
from sklearn.model_selection import cross_val_score
```

```
for key, classifier in classifiers.items():
    classifier.fit(X_train, y_train)
    training_score = cross_val_score(classifier, X_train, y_train,
cv=5)
    print("Classifiers: ", classifier.__class__.__name__, "Has a
training score of", round(training_score.mean(), 2) * 100, "% accuracy
score")
```

```
Classifiers:  LogisticRegression Has a training score of 95.0 %
accuracy score
Classifiers:  KNeighborsClassifier Has a training score of 93.0 %
accuracy score
Classifiers:  SVC Has a training score of 92.0 % accuracy score
Classifiers:  DecisionTreeClassifier Has a training score of 88.0 %
accuracy score
```

```
# Use GridSearchCV to find the best parameters.
from sklearn.model_selection import GridSearchCV


# Logistic Regression
log_reg_params = {"penalty": ['l1', 'l2'], 'C': [0.001, 0.01, 0.1, 1,
10, 100, 1000]}



grid_log_reg = GridSearchCV(LogisticRegression(), log_reg_params)
grid_log_reg.fit(X_train, y_train)
# We automatically get the logistic regression with the best
parameters.
log_reg = grid_log_reg.best_estimator_

knears_params = {"n_neighbors": list(range(2,5,1)), 'algorithm':
['auto', 'ball_tree', 'kd_tree', 'brute']}

grid_knears = GridSearchCV(KNeighborsClassifier(), knears_params)
grid_knears.fit(X_train, y_train)
# KNears best estimator
knears_neighbors = grid_knears.best_estimator_

# Support Vector Classifier
svc_params = {'C': [0.5, 0.7, 0.9, 1], 'kernel': ['rbf', 'poly',
'sigmoid', 'linear']}
grid_svc = GridSearchCV(SVC(), svc_params)
grid_svc.fit(X_train, y_train)

# SVC best estimator
svc = grid_svc.best_estimator_

# DecisionTree Classifier
```

```
tree_params = {"criterion": ["gini", "entropy"], "max_depth":
list(range(2,4,1)),
              "min_samples_leaf": list(range(5,7,1))}
grid_tree = GridSearchCV(DecisionTreeClassifier(), tree_params)
grid_tree.fit(X_train, y_train)

# tree best estimator
tree_clf = grid_tree.best_estimator_

# Overfitting Case

log_reg_score = cross_val_score(log_reg, X_train, y_train, cv=5)
print('Logistic Regression Cross Validation Score: ',
round(log_reg_score.mean() * 100, 2).astype(str) + '%')


knears_score = cross_val_score(knears_neighbors, X_train, y_train,
cv=5)
print('Knears Neighbors Cross Validation Score',
round(knears_score.mean() * 100, 2).astype(str) + '%')

svc_score = cross_val_score(svc, X_train, y_train, cv=5)
print('Support Vector Classifier Cross Validation Score',
round(svc_score.mean() * 100, 2).astype(str) + '%')

tree_score = cross_val_score(tree_clf, X_train, y_train, cv=5)
print('DecisionTree Classifier Cross Validation Score',
round(tree_score.mean() * 100, 2).astype(str) + '%')

Logistic Regression Cross Validation Score:  94.05%
Knears Neighbors Cross Validation Score 92.73%
Support Vector Classifier Cross Validation Score 93.79%
DecisionTree Classifier Cross Validation Score 91.41%

# We will undersample during cross validating
undersample_X = df.drop('Class', axis=1)
undersample_y = df['Class']

for train_index, test_index in sss.split(undersample_X,
undersample_y):
    print("Train:", train_index, "Test:", test_index)
    undersample_Xtrain, undersample_Xtest =
undersample_X.iloc[train_index], undersample_X.iloc[test_index]
    undersample_ytrain, undersample_ytest =
undersample_y.iloc[train_index], undersample_y.iloc[test_index]

undersample_Xtrain = undersample_Xtrain.values
undersample_Xtest = undersample_Xtest.values
undersample_ytrain = undersample_ytrain.values
undersample_ytest = undersample_ytest.values
```

```
undersample_accuracy = []
undersample_precision = []
undersample_recall = []
undersample_f1 = []
undersample_auc = []

# Implementing NearMiss Technique
# Distribution of NearMiss (Just to see how it distributes the labels
we won't use these variables)
X_nearmiss, y_nearmiss = NearMiss().fit_sample(undersample_X.values,
undersample_y.values)
print('NearMiss Label Distribution: {}'.format(Counter(y_nearmiss)))
# Cross Validating the right way

for train, test in sss.split(undersample_Xtrain, undersample_ytrain):
    undersample_pipeline =
imbalanced_make_pipeline(NearMiss(sampling_strategy='majority'),
log_reg) # SMOTE happens during Cross Validation not before..
    undersample_model =
undersample_pipeline.fit(undersample_Xtrain[train],
undersample_ytrain[train])
    undersample_prediction =
undersample_model.predict(undersample_Xtrain[test])


undersample_accuracy.append(undersample_pipeline.score(original_Xtrain
[test], original_ytrain[test]))

undersample_precision.append(precision_score(original_ytrain[test],
undersample_prediction))
    undersample_recall.append(recall_score(original_ytrain[test],
undersample_prediction))
    undersample_f1.append(f1_score(original_ytrain[test],
undersample_prediction))
    undersample_auc.append(roc_auc_score(original_ytrain[test],
undersample_prediction))
```

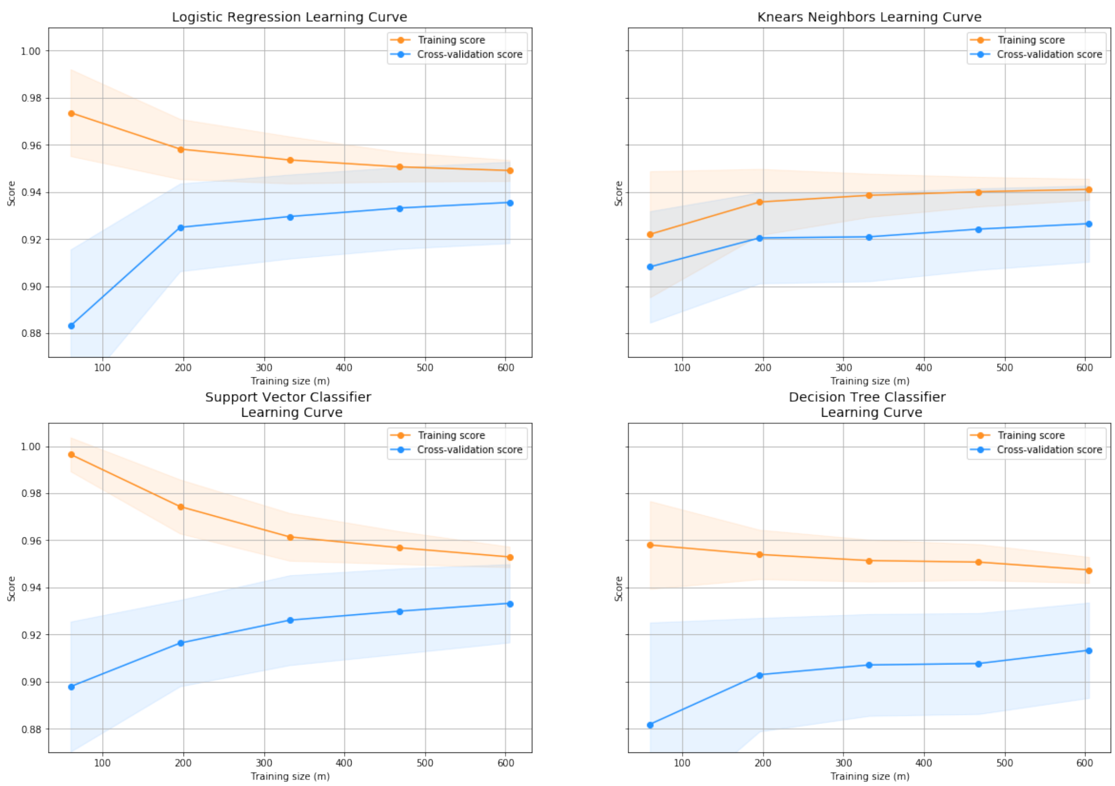
```
Train: [ 56959  56960  56961 ... 284804 284805 284806] Test: [    0
1     2 ... 57174 58268 58463]
Train: [     0      1      2 ... 284804 284805 284806] Test: [ 56959
56960  56961 ... 115109 116514 116648]
Train: [     0      1      2 ... 284804 284805 284806] Test: [113919
113920 113921 ... 170890 170891 170892]
Train: [     0      1      2 ... 284804 284805 284806] Test: [168136
168614 168817 ... 228955 229310 229751]
Train: [     0      1      2 ... 228955 229310 229751] Test: [227842
227843 227844 ... 284804 284805 284806]
NearMiss Label Distribution: Counter({0: 492, 1: 492})
```

```
# Let's Plot LogisticRegression Learning Curve
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import learning_curve

def plot_learning_curve(estimator1, estimator2, estimator3,
estimator4, X, y, ylim=None, cv=None,
                        n_jobs=1, train_sizes=np.linspace(.1, 1.0,
5)):
    f, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2,2, figsize=(20,14),
sharey=True)
    if ylim is not None:
        plt.ylim(*ylim)
    # First Estimator
    train_sizes, train_scores, test_scores = learning_curve(
        estimator1, X, y, cv=cv, n_jobs=n_jobs,
train_sizes=train_sizes)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    ax1.fill_between(train_sizes, train_scores_mean -
train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="#ff9124")
    ax1.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1,
color="#2492ff")
    ax1.plot(train_sizes, train_scores_mean, 'o-', color="#ff9124",
             label="Training score")
    ax1.plot(train_sizes, test_scores_mean, 'o-', color="#2492ff",
             label="Cross-validation score")
    ax1.set_title("Logistic Regression Learning Curve", fontsize=14)
    ax1.set_xlabel('Training size (m)')
    ax1.set_ylabel('Score')
    ax1.grid(True)
    ax1.legend(loc="best")

    # Second Estimator
    train_sizes, train_scores, test_scores = learning_curve(
        estimator2, X, y, cv=cv, n_jobs=n_jobs,
train_sizes=train_sizes)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    ax2.fill_between(train_sizes, train_scores_mean -
train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="#ff9124")
    ax2.fill_between(train_sizes, test_scores_mean - test_scores_std,
```

```
                     test_scores_mean + test_scores_std, alpha=0.1,
color="#2492ff")
    ax2.plot(train_sizes, train_scores_mean, 'o-', color="#ff9124",
             label="Training score")
    ax2.plot(train_sizes, test_scores_mean, 'o-', color="#2492ff",
             label="Cross-validation score")
    ax2.set_title("Knears Neighbors Learning Curve", fontsize=14)
    ax2.set_xlabel('Training size (m)')
    ax2.set_ylabel('Score')
    ax2.grid(True)
    ax2.legend(loc="best")

    # Third Estimator
    train_sizes, train_scores, test_scores = learning_curve(
        estimator3, X, y, cv=cv, n_jobs=n_jobs,
train_sizes=train_sizes)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    ax3.fill_between(train_sizes, train_scores_mean -
train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="#ff9124")
    ax3.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1,
color="#2492ff")
    ax3.plot(train_sizes, train_scores_mean, 'o-', color="#ff9124",
             label="Training score")
    ax3.plot(train_sizes, test_scores_mean, 'o-', color="#2492ff",
             label="Cross-validation score")
    ax3.set_title("Support Vector Classifier \n Learning Curve",
fontsize=14)
    ax3.set_xlabel('Training size (m)')
    ax3.set_ylabel('Score')
    ax3.grid(True)
    ax3.legend(loc="best")

    # Fourth Estimator
    train_sizes, train_scores, test_scores = learning_curve(
        estimator4, X, y, cv=cv, n_jobs=n_jobs,
train_sizes=train_sizes)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    ax4.fill_between(train_sizes, train_scores_mean -
train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="#ff9124")
```

```
    ax4.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1,
color="#2492ff")
    ax4.plot(train_sizes, train_scores_mean, 'o-', color="#ff9124",
             label="Training score")
    ax4.plot(train_sizes, test_scores_mean, 'o-', color="#2492ff",
             label="Cross-validation score")
    ax4.set_title("Decision Tree Classifier \n Learning Curve",
fontsize=14)
    ax4.set_xlabel('Training size (m)')
    ax4.set_ylabel('Score')
    ax4.grid(True)
    ax4.legend(loc="best")
    return plt

cv = ShuffleSplit(n_splits=100, test_size=0.2, random_state=42)
plot_learning_curve(log_reg, knears_neighbors, svc, tree_clf, X_train,
y_train, (0.87, 1.01), cv=cv, n_jobs=4)
```

```
<module 'matplotlib.pyplot' from '/opt/conda/lib/python3.6/site-
packages/matplotlib/pyplot.py'>
```




```
from sklearn.metrics import roc_curve
from sklearn.model_selection import cross_val_predict
# Create a DataFrame with all the scores and the classifiers names.

log_reg_pred = cross_val_predict(log_reg, X_train, y_train, cv=5,
```

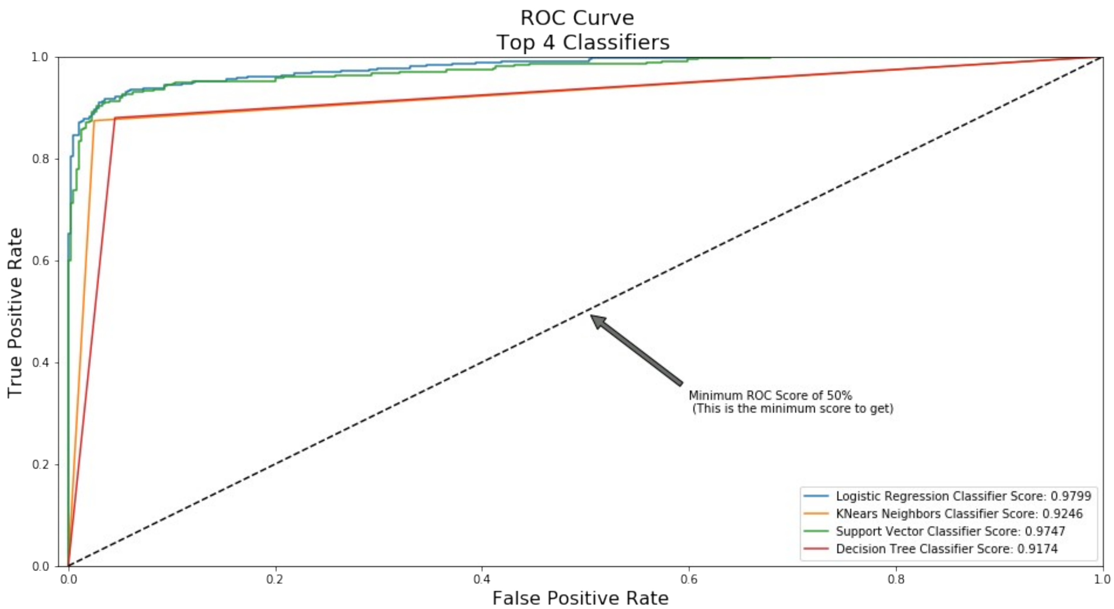
```
                             method="decision_function")

knears_pred = cross_val_predict(knears_neighbors, X_train, y_train,
cv=5)

svc_pred = cross_val_predict(svc, X_train, y_train, cv=5,
                             method="decision_function")

tree_pred = cross_val_predict(tree_clf, X_train, y_train, cv=5)
```

```
from sklearn.metrics import roc_auc_score

print('Logistic Regression: ', roc_auc_score(y_train, log_reg_pred))
print('KNears Neighbors: ', roc_auc_score(y_train, knears_pred))
print('Support Vector Classifier: ', roc_auc_score(y_train, svc_pred))
print('Decision Tree Classifier: ', roc_auc_score(y_train, tree_pred))
```

```
Logistic Regression:  0.9798658657817729
KNears Neighbors:  0.9246195096680248
Support Vector Classifier:  0.9746783159014857
Decision Tree Classifier:  0.9173877431007686
```

```
log_fpr, log_tpr, log_thresold = roc_curve(y_train, log_reg_pred)
knear_fpr, knear_tpr, knear_threshold = roc_curve(y_train,
knears_pred)
svc_fpr, svc_tpr, svc_threshold = roc_curve(y_train, svc_pred)
tree_fpr, tree_tpr, tree_threshold = roc_curve(y_train, tree_pred)


def graph_roc_curve_multiple(log_fpr, log_tpr, knear_fpr, knear_tpr,
svc_fpr, svc_tpr, tree_fpr, tree_tpr):
    plt.figure(figsize=(16,8))
    plt.title('ROC Curve \n Top 4 Classifiers', fontsize=18)
    plt.plot(log_fpr, log_tpr, label='Logistic Regression Classifier
Score: {:.4f}'.format(roc_auc_score(y_train, log_reg_pred)))
    plt.plot(knear_fpr, knear_tpr, label='KNears Neighbors Classifier
Score: {:.4f}'.format(roc_auc_score(y_train, knears_pred)))
    plt.plot(svc_fpr, svc_tpr, label='Support Vector Classifier Score:
{:.4f}'.format(roc_auc_score(y_train, svc_pred)))
    plt.plot(tree_fpr, tree_tpr, label='Decision Tree Classifier
Score: {:.4f}'.format(roc_auc_score(y_train, tree_pred)))
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([-0.01, 1, 0, 1])
    plt.xlabel('False Positive Rate', fontsize=16)
    plt.ylabel('True Positive Rate', fontsize=16)
    plt.annotate('Minimum ROC Score of 50% \n (This is the minimum
score to get)', xy=(0.5, 0.5), xytext=(0.6, 0.3),
                arrowprops=dict(facecolor='#6E726D', shrink=0.05),
                )
    plt.legend()
```

```
graph_roc_curve_multiple(log_fpr, log_tpr, knear_fpr, knear_tpr, svc_fpr, svc_tpr, tree_fpr, tree_tpr)
plt.show()
```



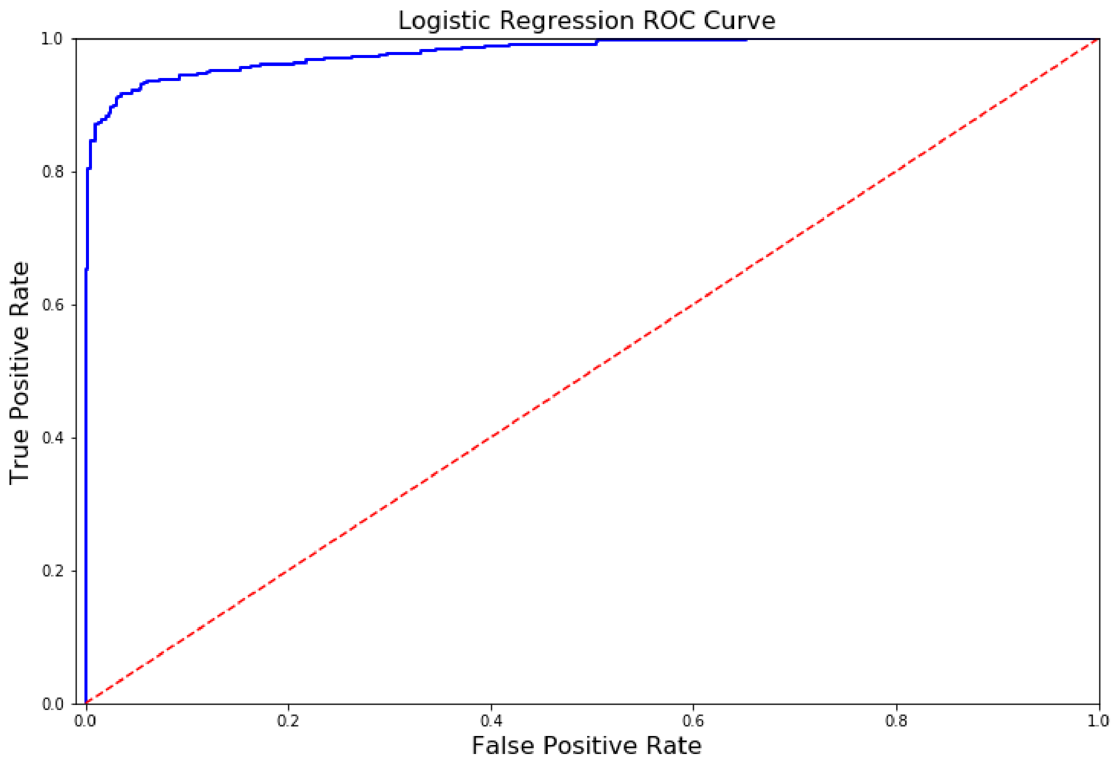
A Deeper Look into LogisticRegression:

In this section we will ive a deeper look into the logistic regression classifier.

Terms:

Summary:

```
def logistic_roc_curve(log_fpr, log_tpr):
    plt.figure(figsize=(12,8))
    plt.title('Logistic Regression ROC Curve', fontsize=16)
    plt.plot(log_fpr, log_tpr, 'b-', linewidth=2)
    plt.plot([0, 1], [0, 1], 'r--')
    plt.xlabel('False Positive Rate', fontsize=16)
    plt.ylabel('True Positive Rate', fontsize=16)
    plt.axis([-0.01,1,0,1])


logistic_roc_curve(log_fpr, log_tpr)
plt.show()
```




```python
from sklearn.metrics import precision_recall_curve

precision, recall, threshold = precision_recall_curve(y_train,
log_reg_pred)

from sklearn.metrics import recall_score, precision_score, f1_score,
accuracy_score
y_pred = log_reg.predict(X_train)

# Overfitting Case
print('---' * 45)
print('Overfitting: \n')
print('Recall Score: {:.2f}'.format(recall_score(y_train, y_pred)))
print('Precision Score: {:.2f}'.format(precision_score(y_train,
y_pred)))
print('F1 Score: {:.2f}'.format(f1_score(y_train, y_pred)))
print('Accuracy Score: {:.2f}'.format(accuracy_score(y_train,
y_pred)))
print('---' * 45)

# How it should look like
print('---' * 45)
print('How it should be:\n')
print("Accuracy Score: {:.2f}".format(np.mean(undersample_accuracy)))
print("Precision Score:
{:.2f}".format(np.mean(undersample_precision)))
print("Recall Score: {:.2f}".format(np.mean(undersample_recall)))
```

```
print("F1 Score: {:.2f}".format(np.mean(undersample_f1)))
print('---' * 45)
```

```
-------------------------------------------------------------------
------------------------------------------------------------------
Overfitting:

Recall Score: 0.90
Precision Score: 0.76
F1 Score: 0.82
Accuracy Score: 0.81
-------------------------------------------------------------------
------------------------------------------------------------------
-------------------------------------------------------------------
------------------------------------------------------------------
How it should be:

Accuracy Score: 0.65
Precision Score: 0.00
Recall Score: 0.29
F1 Score: 0.00
-------------------------------------------------------------------
------------------------------------------------------------------
```

```
undersample_y_score = log_reg.decision_function(original_Xtest)
```

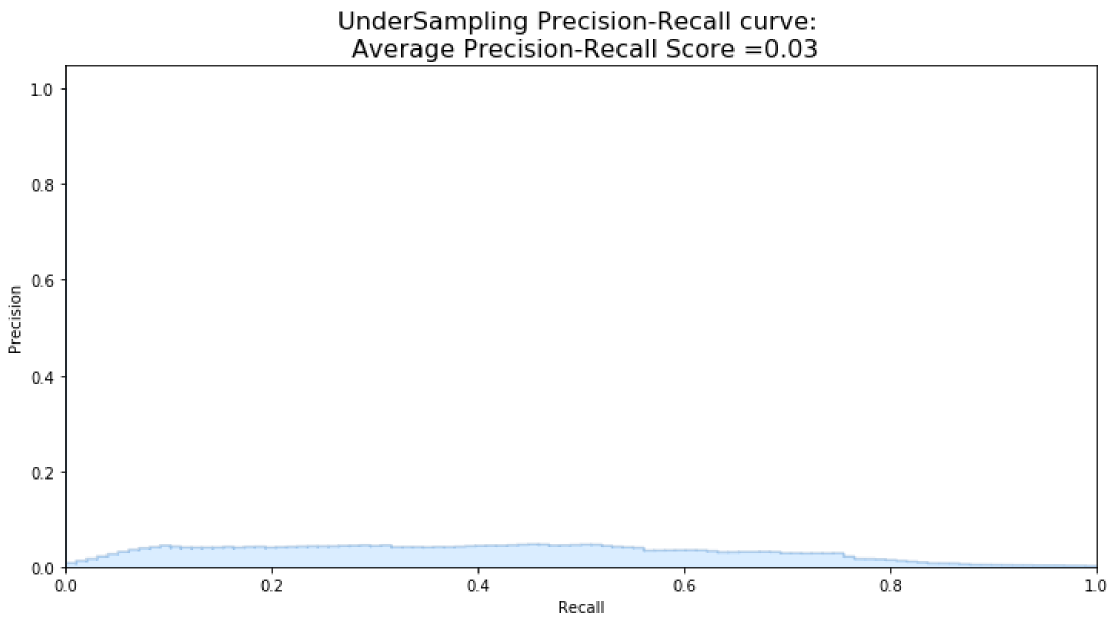
```
from sklearn.metrics import average_precision_score

undersample_average_precision =
average_precision_score(original_ytest, undersample_y_score)

print('Average precision-recall score: {0:0.2f}'.format(
      undersample_average_precision))
```

```
Average precision-recall score: 0.03
```

```
from sklearn.metrics import precision_recall_curve
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(12,6))

precision, recall, _ = precision_recall_curve(original_ytest,
undersample_y_score)

plt.step(recall, precision, color='#004a93', alpha=0.2,
         where='post')
plt.fill_between(recall, precision, step='post', alpha=0.2,
                 color='#48a6ff')

plt.xlabel('Recall')
```

```
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('UnderSampling Precision-Recall curve: \n Average Precision-Recall Score ={0:0.2f}'.format(
          undersample_average_precision), fontsize=16)
```

```
Text(0.5, 1.0, 'UnderSampling Precision-Recall curve: \n Average Precision-Recall Score =0.03')
```



SMOTE Technique (Over-Sampling):

<img src="https://raw.githubusercontent.com/rikunert/SMOTE_visualisation/master/SMOTE_R_visualisation_3.png", width=800> SMOTE stands for Synthetic Minority Over-sampling Technique. Unlike Random UnderSampling, SMOTE creates new synthetic points in order to have an equal balance of the classes. This is another alternative for solving the "class imbalance problems".

Understanding SMOTE: Solving the Class Imbalance: SMOTE creates synthetic points from the minority class in order to reach an equal balance between the minority and majority class. Location of the synthetic points: SMOTE picks the distance between the closest neighbors of the minority class, in between these distances it creates synthetic points.   Final Effect: More information is retained since we didn't have to delete any rows unlike in random undersampling. Accuracy || Time Tradeoff: Although it is likely that SMOTE will be more accurate than random under-sampling, it will take more time to train since no rows are eliminated as previously stated.

Cross Validation Overfitting Mistake:

Overfitting during Cross Validation:

In our undersample analysis I want to show you a common mistake I made that I want to share with all of you. It is simple, if you want to undersample or oversample your data you should not do it before cross validating. Why because you will be directly influencing the validation set before implementing cross-validation causing a "data leakage" problem. In the following section you will see amazing precision and recall scores but in reality our data is overfitting!

The Wrong Way:

As mentioned previously, if we get the minority class ("Fraud) in our case, and create the synthetic points before cross validating we have a certain influence on the "validation set" of the cross validation process. Remember how cross validation works, let's assume we are splitting the data into 5 batches, 4/5 of the dataset will be the training set while 1/5 will be the validation set. The test set should not be touched! For that reason, we have to do the creation of synthetic datapoints "during" cross-validation and not before, just like below:

The Right Way:

As you see above, SMOTE occurs "during" cross validation and not "prior" to the cross validation process. Synthetic data are created only for the training set without affecting the validation set.

References: DEALING WITH IMBALANCED DATA: UNDERSAMPLING, OVERSAMPLING AND PROPER CROSS-VALIDATION

```
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split, 
RandomizedSearchCV


print('Length of X (train): {} | Length of y (train): 
{}'.format(len(original_Xtrain), len(original_ytrain)))
print('Length of X (test): {} | Length of y (test): 
{}'.format(len(original_Xtest), len(original_ytest)))

# List to append the score and then find the average
accuracy_lst = []
precision_lst = []
recall_lst = []
f1_lst = []
auc_lst = []

# Classifier with optimal parameters
# log_reg_sm = grid_log_reg.best_estimator_
```

```
log_reg_sm = LogisticRegression()




rand_log_reg = RandomizedSearchCV(LogisticRegression(),
log_reg_params, n_iter=4)


# Implementing SMOTE Technique
# Cross Validating the right way
# Parameters
log_reg_params = {"penalty": ['l1', 'l2'], 'C': [0.001, 0.01, 0.1, 1,
10, 100, 1000]}
for train, test in sss.split(original_Xtrain, original_ytrain):
    pipeline =
imbalanced_make_pipeline(SMOTE(sampling_strategy='minority'),
rand_log_reg) # SMOTE happens during Cross Validation not before..
    model = pipeline.fit(original_Xtrain[train],
original_ytrain[train])
    best_est = rand_log_reg.best_estimator_
    prediction = best_est.predict(original_Xtrain[test])

    accuracy_lst.append(pipeline.score(original_Xtrain[test],
original_ytrain[test]))
    precision_lst.append(precision_score(original_ytrain[test],
prediction))
    recall_lst.append(recall_score(original_ytrain[test], prediction))
    f1_lst.append(f1_score(original_ytrain[test], prediction))
    auc_lst.append(roc_auc_score(original_ytrain[test], prediction))

print('---' * 45)
print('')
print("accuracy: {}".format(np.mean(accuracy_lst)))
print("precision: {}".format(np.mean(precision_lst)))
print("recall: {}".format(np.mean(recall_lst)))
print("f1: {}".format(np.mean(f1_lst)))
print('---' * 45)
```

```
Length of X (train): 227846 | Length of y (train): 227846
Length of X (test): 56961 | Length of y (test): 56961
----------------------------------------------------------------------
-----------------------------------------------------------------

accuracy: 0.9694005888966659
precision: 0.06547023328181797
recall: 0.9111002921129504
f1: 0.1209666729570652
----------------------------------------------------------------------
-----------------------------------------------------------------
```

```
labels = ['No Fraud', 'Fraud']
smote_prediction = best_est.predict(original_Xtest)
print(classification_report(original_ytest, smote_prediction,
target_names=labels))
```

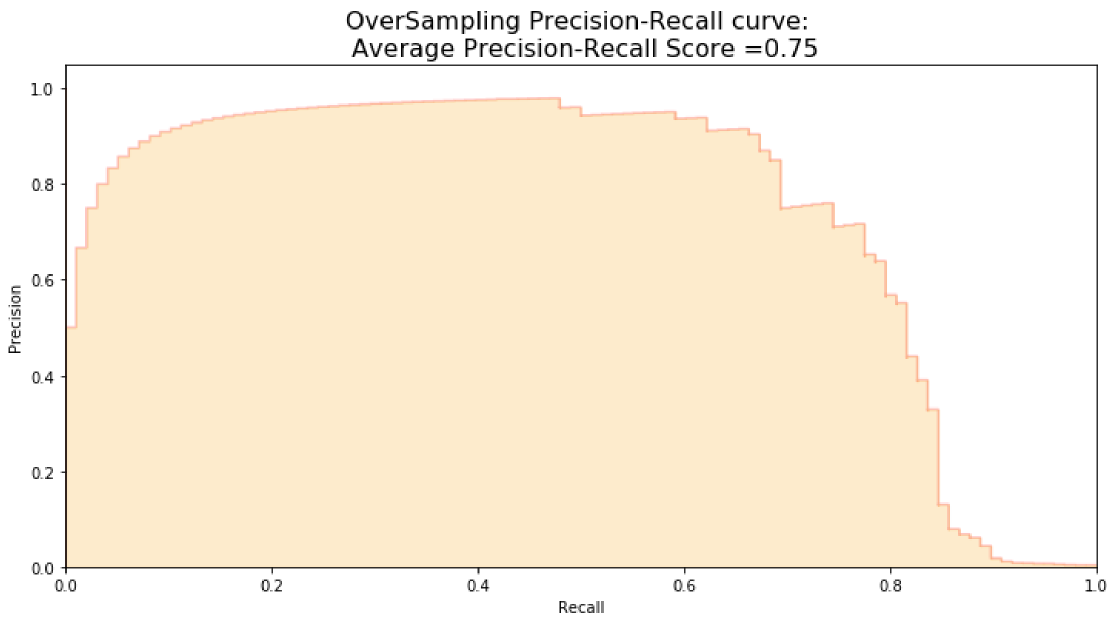
```
              precision    recall  f1-score   support

    No Fraud       1.00      0.99      0.99     56863
       Fraud       0.10      0.86      0.19        98

    accuracy                           0.99     56961
   macro avg       0.55      0.92      0.59     56961
weighted avg       1.00      0.99      0.99     56961
```

```
y_score = best_est.decision_function(original_Xtest)

average_precision = average_precision_score(original_ytest, y_score)

print('Average precision-recall score: {0:0.2f}'.format(
      average_precision))
```

```
Average precision-recall score: 0.75
```

```
fig = plt.figure(figsize=(12,6))

precision, recall, _ = precision_recall_curve(original_ytest, y_score)

plt.step(recall, precision, color='r', alpha=0.2,
         where='post')
plt.fill_between(recall, precision, step='post', alpha=0.2,
                 color='#F59B00')

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('OverSampling Precision-Recall curve: \n Average Precision-
Recall Score ={0:0.2f}'.format(
          average_precision), fontsize=16)
```

```
Text(0.5, 1.0, 'OverSampling Precision-Recall curve: \n Average
Precision-Recall Score =0.75')
```




```
# SMOTE Technique (OverSampling) After splitting and Cross Validating
sm = SMOTE(ratio='minority', random_state=42)
# Xsm_train, ysm_train = sm.fit_sample(X_train, y_train)


# This will be the data were we are going to
Xsm_train, ysm_train = sm.fit_sample(original_Xtrain, original_ytrain)

# We Improve the score by 2% points approximately
# Implement GridSearchCV and the other models.

# Logistic Regression
t0 = time.time()
log_reg_sm = grid_log_reg.best_estimator_
log_reg_sm.fit(Xsm_train, ysm_train)
t1 = time.time()
print("Fitting oversample data took :{} sec".format(t1 - t0))
```

```
Fitting oversample data took :14.371394634246826 sec
```

Test Data with Logistic Regression:

Confusion Matrix:

Positive/Negative: Type of Class (label) ["No", "Yes"] **True/False:** Correctly or Incorrectly classified by the model.

True Negatives (Top-Left Square): This is the number of **correctly** classifications of the "No" (No Fraud Detected) class.

False Negatives (Top-Right Square): This is the number of **incorrectly** classifications of the "No"(No Fraud Detected) class.

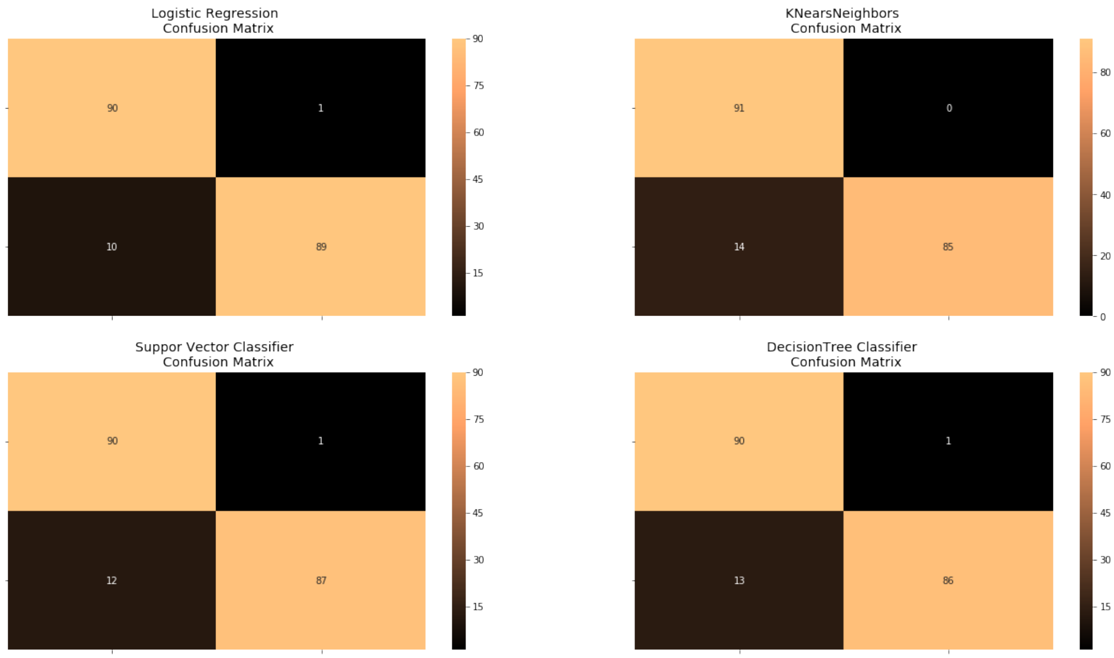
False Positives (Bottom-Left Square): This is the number of **incorrectly** classifications of the "Yes" (Fraud Detected) class

True Positives (Bottom-Right Square): This is the number of **correctly** classifications of the "Yes" (Fraud Detected) class.

Summary:

```
from sklearn.metrics import confusion_matrix

# Logistic Regression fitted using SMOTE technique
y_pred_log_reg = log_reg_sm.predict(X_test)

# Other models fitted with UnderSampling
y_pred_knear = knears_neighbors.predict(X_test)
y_pred_svc = svc.predict(X_test)
y_pred_tree = tree_clf.predict(X_test)


log_reg_cf = confusion_matrix(y_test, y_pred_log_reg)
kneighbors_cf = confusion_matrix(y_test, y_pred_knear)
svc_cf = confusion_matrix(y_test, y_pred_svc)
tree_cf = confusion_matrix(y_test, y_pred_tree)

fig, ax = plt.subplots(2, 2,figsize=(22,12))


sns.heatmap(log_reg_cf, ax=ax[0][0], annot=True, cmap=plt.cm.copper)
ax[0, 0].set_title("Logistic Regression \n Confusion Matrix",
fontsize=14)
ax[0, 0].set_xticklabels(['', ''], fontsize=14, rotation=90)
ax[0, 0].set_yticklabels(['', ''], fontsize=14, rotation=360)

sns.heatmap(kneighbors_cf, ax=ax[0][1], annot=True,
cmap=plt.cm.copper)
ax[0][1].set_title("KNearsNeighbors \n Confusion Matrix", fontsize=14)
ax[0][1].set_xticklabels(['', ''], fontsize=14, rotation=90)
ax[0][1].set_yticklabels(['', ''], fontsize=14, rotation=360)

sns.heatmap(svc_cf, ax=ax[1][0], annot=True, cmap=plt.cm.copper)
ax[1][0].set_title("Suppor Vector Classifier \n Confusion Matrix",
fontsize=14)
ax[1][0].set_xticklabels(['', ''], fontsize=14, rotation=90)
ax[1][0].set_yticklabels(['', ''], fontsize=14, rotation=360)
```

```
sns.heatmap(tree_cf, ax=ax[1][1], annot=True, cmap=plt.cm.copper)
ax[1][1].set_title("DecisionTree Classifier \n Confusion Matrix",
fontsize=14)
ax[1][1].set_xticklabels(['', ''], fontsize=14, rotation=90)
ax[1][1].set_yticklabels(['', ''], fontsize=14, rotation=360)


plt.show()
```




```
from sklearn.metrics import classification_report


print('Logistic Regression:')
print(classification_report(y_test, y_pred_log_reg))

print('KNears Neighbors:')
print(classification_report(y_test, y_pred_knear))

print('Support Vector Classifier:')
print(classification_report(y_test, y_pred_svc))

print('Support Vector Classifier:')
print(classification_report(y_test, y_pred_tree))
```

```
Logistic Regression:
              precision    recall  f1-score   support

           0       0.90      0.99      0.94        91
           1       0.99      0.90      0.94        99
```

```
    accuracy                           0.94       190
   macro avg       0.94      0.94      0.94       190
weighted avg       0.95      0.94      0.94       190

KNears Neighbors:
              precision    recall  f1-score   support

           0       0.87      1.00      0.93        91
           1       1.00      0.86      0.92        99

    accuracy                           0.93       190
   macro avg       0.93      0.93      0.93       190
weighted avg       0.94      0.93      0.93       190

Support Vector Classifier:
              precision    recall  f1-score   support

           0       0.88      0.99      0.93        91
           1       0.99      0.88      0.93        99

    accuracy                           0.93       190
   macro avg       0.94      0.93      0.93       190
weighted avg       0.94      0.93      0.93       190

Support Vector Classifier:
              precision    recall  f1-score   support

           0       0.87      0.99      0.93        91
           1       0.99      0.87      0.92        99

    accuracy                           0.93       190
   macro avg       0.93      0.93      0.93       190
weighted avg       0.93      0.93      0.93       190
```

```
# Final Score in the test set of logistic regression
from sklearn.metrics import accuracy_score

# Logistic Regression with Under-Sampling
y_pred = log_reg.predict(X_test)
undersample_score = accuracy_score(y_test, y_pred)



# Logistic Regression with SMOTE Technique (Better accuracy with SMOTE
t)
y_pred_sm = best_est.predict(original_Xtest)
oversample_score = accuracy_score(original_ytest, y_pred_sm)
```

```python
d = {'Technique': ['Random UnderSampling', 'Oversampling (SMOTE)'],
'Score': [undersample_score, oversample_score]}
final_df = pd.DataFrame(data=d)

# Move column
score = final_df['Score']
final_df.drop('Score', axis=1, inplace=True)
final_df.insert(1, 'Score', score)

# Note how high is accuracy score it can be misleading!
final_df
```

```
             Technique     Score
0  Random UnderSampling  0.942105
1  Oversampling (SMOTE)  0.987079
```

Neural Networks Testing Random UnderSampling Data vs OverSampling (SMOTE):

In this section we will implement a simple Neural Network (with one hidden layer) in order to see which of the two logistic regressions models we implemented in the (undersample or oversample(SMOTE)) has a better accuracy for detecting fraud and non-fraud transactions.

Our Main Goal:

Our main goal is to explore how our simple neural network behaves in both the random undersample and oversample dataframes and see whether they can predict accuractely both non-fraud and fraud cases. Why not only focus on fraud? Imagine you were a cardholder and after you purchased an item your card gets blocked because the bank's algorithm thought your purchase was a fraud. That's why we shouldn't emphasize only in detecting fraud cases but we should also emphasize correctly categorizing non-fraud transactions.

The Confusion Matrix:

Here is again, how the confusion matrix works: Upper Left Square: The amount of correctly classified by our model of no fraud transactions. Upper Right Square: The amount of incorrectly classified transactions as fraud cases, but the actual label is no fraud . Lower Left Square: The amount of incorrectly classified transactions as no fraud cases, but the actual label is fraud . Lower Right Square: The amount of correctly classified by our model of fraud transactions.

Summary (Keras || Random UnderSampling):

```python
import keras
from keras import backend as K
from keras.models import Sequential
```

```python
from keras.layers import Activation
from keras.layers.core import Dense
from keras.optimizers import Adam
from keras.metrics import categorical_crossentropy

n_inputs = X_train.shape[1]

undersample_model = Sequential([
    Dense(n_inputs, input_shape=(n_inputs, ), activation='relu'),
    Dense(32, activation='relu'),
    Dense(2, activation='softmax')
])
```

```
Using TensorFlow backend.
```

```python
undersample_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 30)                930
_________________________________________________________________
dense_2 (Dense)              (None, 32)                992
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 66
=================================================================
Total params: 1,988
Trainable params: 1,988
Non-trainable params: 0
_________________________________________________________________
```

```python
undersample_model.compile(Adam(lr=0.001),
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

undersample_model.fit(X_train, y_train, validation_split=0.2,
batch_size=25, epochs=20, shuffle=True, verbose=2)
```

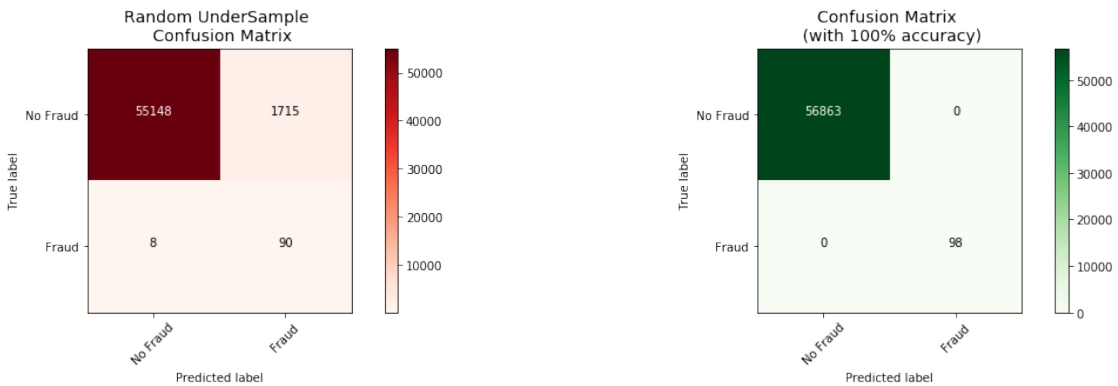
```
Train on 605 samples, validate on 152 samples
Epoch 1/20
 - 0s - loss: 0.4640 - acc: 0.7455 - val_loss: 0.3672 - val_acc:
0.8684
Epoch 2/20
 - 0s - loss: 0.3475 - acc: 0.8579 - val_loss: 0.2970 - val_acc:
0.9342
Epoch 3/20
 - 0s - loss: 0.2830 - acc: 0.9107 - val_loss: 0.2592 - val_acc:
0.9342
Epoch 4/20
 - 0s - loss: 0.2364 - acc: 0.9388 - val_loss: 0.2336 - val_acc:
0.9211
Epoch 5/20
```

```
 - 0s - loss: 0.2038 - acc: 0.9421 - val_loss: 0.2161 - val_acc:
0.9211
Epoch 6/20
 - 0s - loss: 0.1798 - acc: 0.9488 - val_loss: 0.1980 - val_acc:
0.9211
Epoch 7/20
 - 0s - loss: 0.1621 - acc: 0.9504 - val_loss: 0.1890 - val_acc:
0.9276
Epoch 8/20
 - 0s - loss: 0.1470 - acc: 0.9521 - val_loss: 0.1864 - val_acc:
0.9276
Epoch 9/20
 - 0s - loss: 0.1367 - acc: 0.9554 - val_loss: 0.1838 - val_acc:
0.9276
Epoch 10/20
 - 0s - loss: 0.1281 - acc: 0.9603 - val_loss: 0.1826 - val_acc:
0.9211
Epoch 11/20
 - 0s - loss: 0.1218 - acc: 0.9537 - val_loss: 0.1795 - val_acc:
0.9211
Epoch 12/20
 - 0s - loss: 0.1134 - acc: 0.9570 - val_loss: 0.1856 - val_acc:
0.9211
Epoch 13/20
 - 0s - loss: 0.1071 - acc: 0.9587 - val_loss: 0.1852 - val_acc:
0.9276
Epoch 14/20
 - 0s - loss: 0.1015 - acc: 0.9620 - val_loss: 0.1790 - val_acc:
0.9211
Epoch 15/20
 - 0s - loss: 0.0966 - acc: 0.9587 - val_loss: 0.1842 - val_acc:
0.9276
Epoch 16/20
 - 0s - loss: 0.0910 - acc: 0.9636 - val_loss: 0.1813 - val_acc:
0.9276
Epoch 17/20
 - 0s - loss: 0.0871 - acc: 0.9620 - val_loss: 0.1831 - val_acc:
0.9276
Epoch 18/20
 - 0s - loss: 0.0835 - acc: 0.9636 - val_loss: 0.1822 - val_acc:
0.9276
Epoch 19/20
 - 0s - loss: 0.0791 - acc: 0.9702 - val_loss: 0.1822 - val_acc:
0.9276
Epoch 20/20
 - 0s - loss: 0.0751 - acc: 0.9752 - val_loss: 0.1877 - val_acc:
0.9211

<keras.callbacks.History at 0x7f056fd8e278>
```

```python
undersample_predictions = undersample_model.predict(original_Xtest,
batch_size=200, verbose=0)

undersample_fraud_predictions =
undersample_model.predict_classes(original_Xtest, batch_size=200,
verbose=0)

import itertools

# Create a confusion matrix
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title, fontsize=14)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]),
range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

undersample_cm = confusion_matrix(original_ytest,
undersample_fraud_predictions)
actual_cm = confusion_matrix(original_ytest, original_ytest)
labels = ['No Fraud', 'Fraud']
```

```
fig = plt.figure(figsize=(16,8))

fig.add_subplot(221)
plot_confusion_matrix(undersample_cm, labels, title="Random
UnderSample \n Confusion Matrix", cmap=plt.cm.Reds)

fig.add_subplot(222)
plot_confusion_matrix(actual_cm, labels, title="Confusion Matrix \n
(with 100% accuracy)", cmap=plt.cm.Greens)
```

```
Confusion matrix, without normalization
[[55148  1715]
 [    8    90]]
Confusion matrix, without normalization
[[56863     0]
 [    0    98]]
```



Keras || OverSampling (SMOTE):

```
n_inputs = Xsm_train.shape[1]

oversample_model = Sequential([
    Dense(n_inputs, input_shape=(n_inputs, ), activation='relu'),
    Dense(32, activation='relu'),
    Dense(2, activation='softmax')
])

oversample_model.compile(Adam(lr=0.001),
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

oversample_model.fit(Xsm_train, ysm_train, validation_split=0.2,
batch_size=300, epochs=20, shuffle=True, verbose=2)
```

```
Train on 363923 samples, validate on 90981 samples
Epoch 1/20
 - 3s - loss: 0.0641 - acc: 0.9771 - val_loss: 0.0152 - val_acc:
0.9977
Epoch 2/20
 - 3s - loss: 0.0130 - acc: 0.9972 - val_loss: 0.0070 - val_acc:
0.9995
```

```
Epoch 3/20
 - 2s - loss: 0.0078 - acc: 0.9987 - val_loss: 0.0044 - val_acc:
1.0000
Epoch 4/20
 - 2s - loss: 0.0060 - acc: 0.9990 - val_loss: 0.0030 - val_acc:
1.0000
Epoch 5/20
 - 3s - loss: 0.0046 - acc: 0.9992 - val_loss: 0.0036 - val_acc:
0.9999
Epoch 6/20
 - 2s - loss: 0.0035 - acc: 0.9993 - val_loss: 0.0012 - val_acc:
1.0000
Epoch 7/20
 - 2s - loss: 0.0038 - acc: 0.9994 - val_loss: 0.0017 - val_acc:
1.0000
Epoch 8/20
 - 2s - loss: 0.0028 - acc: 0.9995 - val_loss: 0.0027 - val_acc:
0.9999
Epoch 9/20
 - 2s - loss: 0.0022 - acc: 0.9996 - val_loss: 0.0022 - val_acc:
1.0000
Epoch 10/20
 - 2s - loss: 0.0021 - acc: 0.9996 - val_loss: 0.0017 - val_acc:
1.0000
Epoch 11/20
 - 2s - loss: 0.0020 - acc: 0.9996 - val_loss: 0.0013 - val_acc:
1.0000
Epoch 12/20
 - 2s - loss: 0.0018 - acc: 0.9997 - val_loss: 2.8322e-04 - val_acc:
1.0000
Epoch 13/20
 - 3s - loss: 0.0018 - acc: 0.9996 - val_loss: 0.0035 - val_acc:
0.9994
Epoch 14/20
 - 3s - loss: 0.0018 - acc: 0.9997 - val_loss: 9.4907e-04 - val_acc:
1.0000
Epoch 15/20
 - 3s - loss: 0.0014 - acc: 0.9998 - val_loss: 1.5897e-04 - val_acc:
1.0000
Epoch 16/20
 - 2s - loss: 0.0015 - acc: 0.9997 - val_loss: 5.9093e-04 - val_acc:
1.0000
Epoch 17/20
 - 3s - loss: 0.0016 - acc: 0.9997 - val_loss: 3.7523e-04 - val_acc:
1.0000
Epoch 18/20
 - 2s - loss: 0.0014 - acc: 0.9998 - val_loss: 2.7042e-04 - val_acc:
1.0000
Epoch 19/20
 - 3s - loss: 0.0020 - acc: 0.9997 - val_loss: 2.2361e-04 - val_acc:
```

```
1.0000
Epoch 20/20
 - 2s - loss: 0.0012 - acc: 0.9998 - val_loss: 1.8081e-04 - val_acc:
1.0000
```

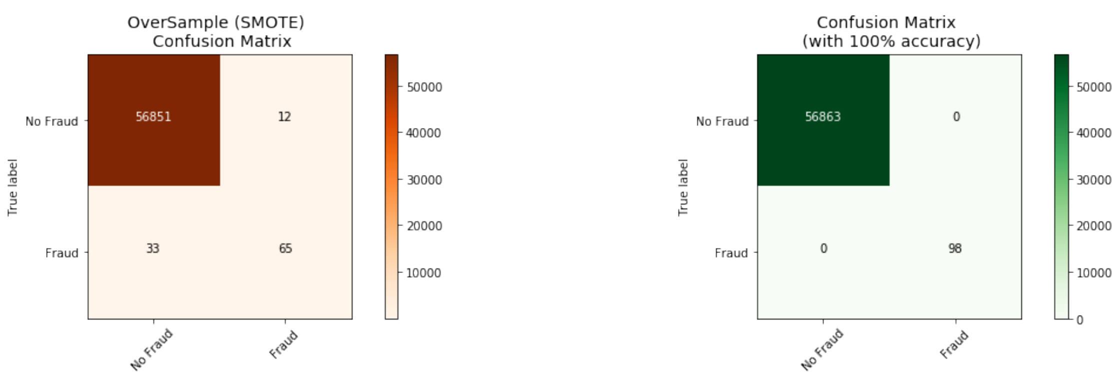
```
<keras.callbacks.History at 0x7f056234b470>
```

```
oversample_predictions = oversample_model.predict(original_Xtest,
batch_size=200, verbose=0)
```

```
oversample_fraud_predictions =
oversample_model.predict_classes(original_Xtest, batch_size=200,
verbose=0)
```

```
oversample_smote = confusion_matrix(original_ytest,
oversample_fraud_predictions)
actual_cm = confusion_matrix(original_ytest, original_ytest)
labels = ['No Fraud', 'Fraud']

fig = plt.figure(figsize=(16,8))

fig.add_subplot(221)
plot_confusion_matrix(oversample_smote, labels, title="OverSample
(SMOTE) \n Confusion Matrix", cmap=plt.cm.Oranges)

fig.add_subplot(222)
plot_confusion_matrix(actual_cm, labels, title="Confusion Matrix \n
(with 100% accuracy)", cmap=plt.cm.Greens)
```

```
Confusion matrix, without normalization
[[56851    12]
 [   33    65]]
Confusion matrix, without normalization
[[56863     0]
 [    0    98]]
```



Conclusion:

Implementing SMOTE on our imbalanced dataset helped us with the imbalance of our labels (more no fraud than fraud transactions). Nevertheless, I still have to state that sometimes the neural network on the oversampled dataset predicts less correct fraud transactions than our model using the undersample dataset. However, remember that the removal of outliers was implemented only on the random undersample dataset and not on the oversampled one. Also, in our undersample data our model is unable to detect for a large number of cases non fraud transactions correctly and instead, misclassifies those non fraud transactions as fraud cases. Imagine that people that were making regular purchases got their card blocked due to the reason that our model classified that transaction as a fraud transaction, this will be a huge disadvantage for the financial institution. The number of customer complaints and customer disatisfaction will increase. The next step of this analysis will be to do an outlier removal on our oversample dataset and see if our accuracy in the test set improves.

Note: One last thing, predictions and accuracies may be subjected to change since I implemented data shuffling on both types of dataframes. The main thing is to see if our models are able to correctly classify no fraud and fraud transactions. I will bring more updates, stay tuned!