Credit card Fraud Detection

Data Set Information:

- Source: https://www.kaggle.com/mlg-ulb/creditcardfraud (https://www.kaggle.com/mlg-ulb/creditcardfraud)

Context

- It is important that credit card companies are able to recognize fraudulent credit card transactions so that customers are not charged for items that they did not purchase.

Dataset

- The datasets contains transactions made by credit cards in September 2013 by european cardholders.
- This dataset presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions. The dataset is highly unbalanced, the positive class (frauds) account for 0.172% of all transactions.
- It contains only numerical input variables which are the result of a **PCA transformation**.
- Unfortunately, due to confidentiality issues, we cannot provide the original features and more background information about the data.
- Features V1, V2, … V28 are the principal components obtained with PCA, the only features which have not been transformed with PCA are 'Time' and 'Amount'.
- The feature **'Time'** contains the seconds elapsed between each transaction and the first transaction in the dataset.
- The feature **'Amount'** is the transaction Amount, this feature can be used for example-dependant cost-senstive learning.
- The feature **'Class'** is the response variable and it takes value 1 in case of fraud and 0 otherwise.

Aim/Purpose

- The idea is create a machine learning model to:
  - Data Exploration/Visualization
    - Understand structure of the data.
  - Predict the **extreme rare fraud cases** with dimension reduced information due to PCA.
  - Determine factors which have the higher probability to predict the quality.
    - Reduce the dimension of variables. (Feature engineering)
  - Determine machine learning model for best classifying the wines. (Metric = recall in this case).

The flaw of analysis:

- We do not consider **"amount"** in the default XGBoost **cost function**.
  - The accuracy should be lower by doing, but at least paying good result.
- Worth spending time investigating to penaltize higher weight on FP in **cost function** for four different scenarios as such:
  - True Positive (TP):
    - Model predicted **fraud** and it's a **fraud**.
    - Administrative cost that investigated the reason determining fraud.
  - False Positive (FP):
    - Model predicted **non-fraud** and it's a **fraud**:
    - Highest cost as credit default case were missed out.
  - True Negative (TN):
    - Model predicted **fraud** and it's **not a fraud**:
    - Administrative cost that investigated the reason determining fraud.
  - False Negative (FN):
    - Model predicted **non-fraud** and it's **not a fraud**:
    - No cost incurred.
      - Recall: True Positives/(True Positives + False Negatives):
      - **the amount of fraud cases our model is able to detect**
- Include all the data in data cleaning process; **does not remove outlier** as we might exclude rare cases which were fraud actually.
- Do not include randomized grid search for hypertuning due to high conputational cost, and assumed the model was nicely tuned.

Model:

- We used **XGboost algorithm** with **SMOTE** to oversample the **fraud cases**.
- **The advantages of model has been explained in /kaggle_winequality.**
- We further evaluate the model using **Area Under the Precision-Recall Curve (AUPRC)**.
- Recall will be priorised as to minimize possible cost by having the highest TP.

Accuracy:

- The recall is the ratio $\frac{tp+fn}{(tp+tn+fp+fn)}$
- Where tp is the number of true positives and fn the number of false negatives, and vice versas for others as labelled above.
- The accuracy is intuitively the ability of the classifier to correctly classify the samples.

Recall:

- The recall is the ratio $\frac{tp}{(tp+fn)}$
- Where tp is the number of true positives and fn the number of false negatives.
- The recall is intuitively the ability of the classifier to find all the positive samples.

Precision:

- The precision is the ratio $\frac{tp}{(tp+fp)}$
- Where tp is the number of true positives and fp the number of false positives.

- The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

F1 score:

- F1 score, also known as balanced F-score or F-measure
- The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0.
- The relative contribution of precision and recall to the F1 score are equal.

$$F1 \ = \ 2 \times \ \frac{(precision \times recall)}{(precision + recall)}$$

- In the multi-class and multi-label case, this is the average of the F1 score of each class with weighting depending on the average parameter.

In [1]:
```
import matplotlib.pyplot as plt
from datetime import timedelta
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline
from matplotlib import pyplot as plt
from plotly.subplots import make_subplots
from sklearn import preprocessing
from sklearn.cluster import KMeans
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import AdaBoostClassifier, RandomForestClassifier, RandomF
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import LogisticRegressionCV
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_sc
from sklearn.model_selection import GridSearchCV, KFold, StratifiedKFold, cross_
from sklearn.multiclass import OneVsRestClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import MinMaxScaler, MultiLabelBinarizer
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from xgboost import XGBClassifier
from yellowbrick.classifier import ClassPredictionError, ROCAUC

import numpy as np
import pandas as pd
import matplotlib
import plotly.figure_factory as ff
import plotly.graph_objects as go
import scipy
import shap
import squarify
import seaborn as sns
import xgboost as xgb

shap.initjs()
np.random.seed(0)
```

```
Using TensorFlow backend.
C:\Users\moses\anaconda3\lib\site-packages\sklearn\utils\deprecation.py:144: Fu
tureWarning: The sklearn.metrics.classification module is  deprecated in versio
n 0.22 and will be removed in version 0.24. The corresponding classes / functio
ns should instead be imported from sklearn.metrics. Anything that cannot be imp
orted from sklearn.metrics is now part of the private API.
  warnings.warn(message, FutureWarning)
```

In [2]:
```
def ABS_SHAP(df_shap,df):
    #import matplotlib as plt
    # Make a copy of the input data
    shap_v = pd.DataFrame(df_shap)
    feature_list = df.columns
    shap_v.columns = feature_list
    df_v = df.copy().reset_index().drop('index',axis=1)

    # Determine the correlation in order to plot with different colors
    corr_list = list()
    for i in feature_list:
        b = np.corrcoef(shap_v[i],df_v[i])[1][0]
        corr_list.append(b)
    corr_df = pd.concat([pd.Series(feature_list),pd.Series(corr_list)],axis=1).f
    # Make a data frame. Column 1 is the feature, and Column 2 is the correlatio
    corr_df.columns  = ['Variable','Corr']
    corr_df['Sign'] = np.where(corr_df['Corr']>0,'red','blue')

    # Plot it
    shap_abs = np.abs(shap_v)
    k=pd.DataFrame(shap_abs.mean()).reset_index()
    k.columns = ['Variable','SHAP_abs']
    k2 = k.merge(corr_df,left_on = 'Variable',right_on='Variable',how='inner')
    k2 = k2.sort_values(by='SHAP_abs',ascending = True)
    colorlist = k2['Sign']
    ax = k2.plot.barh(x='Variable',y='SHAP_abs',color = colorlist, figsize=(5,6)
    ax.set_xlabel("SHAP Value (Red = Positive Impact)")
```

In [3]:
```
dr_workplace = 'C:\\Users\\moses\\OneDrive\\Documents\\machine learning\\kaggle_
df = pd.read_csv(dr_workplace + '\\creditcard.csv') # Load the data
```

In [4]:
```
df.describe()

# The fraud rate in the data is 0.17%, proves that the data itself is quite imba
```

Out[4]:

	Time	V1	V2	V3	V4	V5
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05
mean	94813.859575	3.919560e-15	5.688174e-16	-8.769071e-15	2.782312e-15	-1.552563e-15
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01

8 rows × 31 columns

In [5]:
```
#Preview first 10 rows.
df.head(10)
```

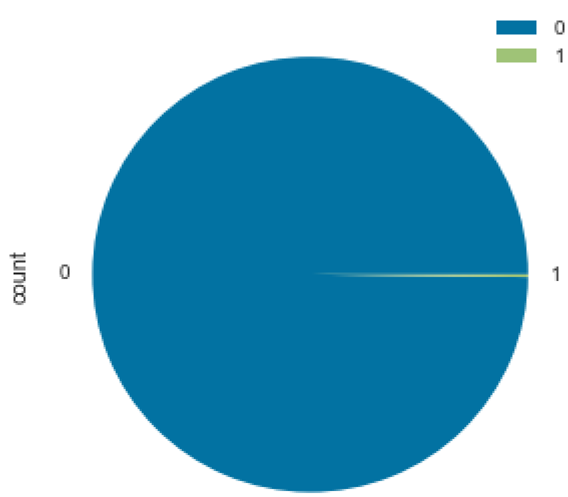
Out[5]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.3
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.2
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.5
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.3
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.8
5	2.0	-0.425966	0.960523	1.141109	-0.168252	0.420987	-0.029728	0.476201	0.260314	-0.5
6	4.0	1.229658	0.141004	0.045371	1.202613	0.191881	0.272708	-0.005159	0.081213	0.4
7	7.0	-0.644269	1.417964	1.074380	-0.492199	0.948934	0.428118	1.120631	-3.807864	0.6
8	7.0	-0.894286	0.286157	-0.113192	-0.271526	2.669599	3.721818	0.370145	0.851084	-0.3
9	9.0	-0.338262	1.119593	1.044367	-0.222187	0.499361	-0.246761	0.651583	0.069539	-0.7

10 rows × 31 columns

In [6]:
```
df_group_class = df.groupby('Class').agg({'Amount': ['count']})
df_group_class.plot.pie(y='Amount', figsize=(5, 5))
```

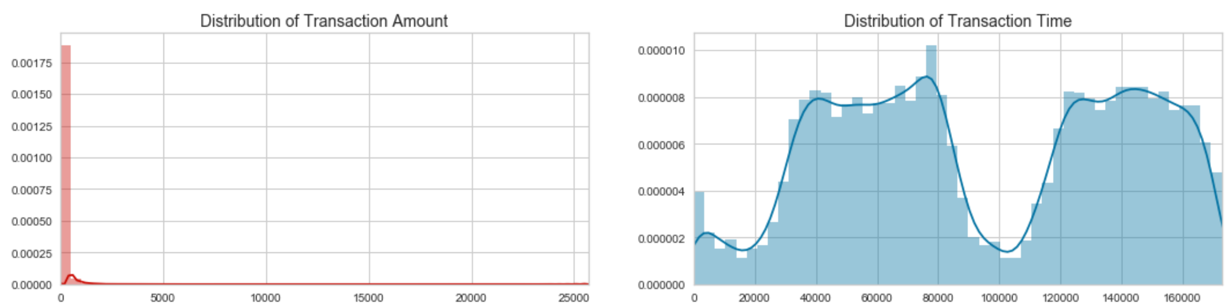
Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0x177c35e3808>



In [7]:
```
# The classes are heavily skewed we need to solve this issue later.
print('No Frauds', round(df['Class'].value_counts()[0]/len(df) * 100,3), '% of tl
print('Frauds', round(df['Class'].value_counts()[1]/len(df) * 100,3), '% of the d
```

```
No Frauds 99.827 % of the dataset
Frauds 0.173 % of the dataset
```

In [8]:
```
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()
```



Correlation matrix

- 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.
- **The subsample is for creation of correlation analysis only, it wouldn't be used for traning of ML model.**

Necessary of removal of outlier

- Removal of outlier might not be necessary as we might be missing out actual fraud cases.
- Moreover, We will be introducing SMOTE technique to oversample the minority rows.

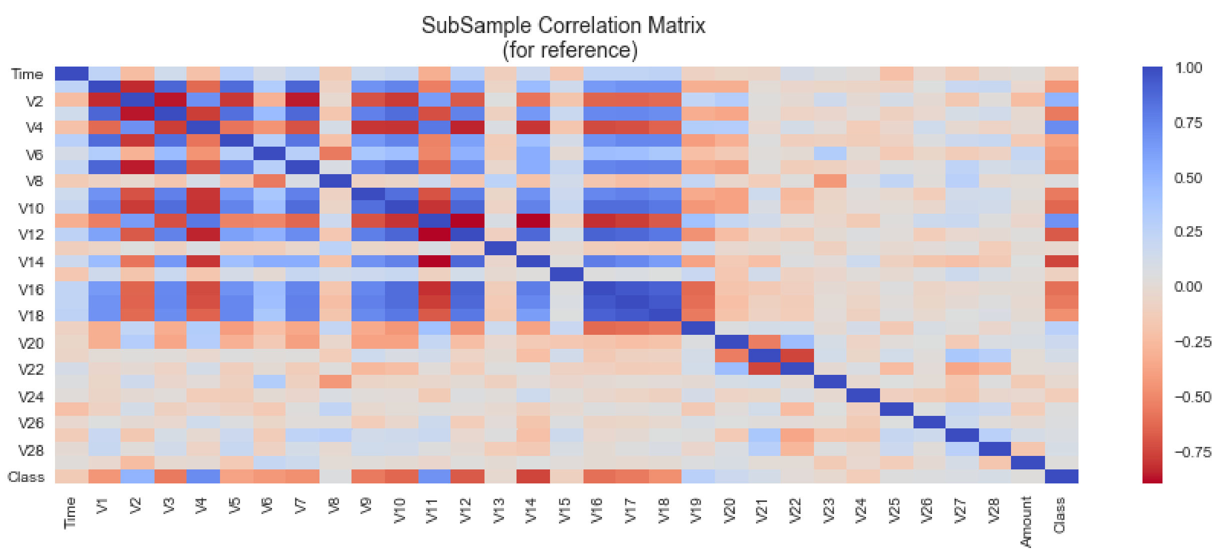
In [9]:
```
df = df.sample(frac=1)

# 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()
```

Out[9]:

	Time	V1	V2	V3	V4	V5	V6	V7	
178066	123462.0	-2.929579	-2.630494	1.775473	-2.310854	-0.490392	-0.042834	-1.674939	0.39
181966	125200.0	-0.769172	1.342212	-2.171454	-0.151513	-0.648374	-0.973504	-1.706658	0.31
265205	161783.0	-0.161936	0.474341	-0.758422	-0.953533	0.751166	-1.570174	2.117030	-0.60
144754	86376.0	-0.670238	0.945206	0.610051	2.640065	-2.707775	1.952611	-1.624608	-5.22
238222	149582.0	-4.280584	1.421100	-3.908229	2.942946	-0.076205	-2.002526	-2.874155	-0.85

5 rows × 31 columns

In [10]:
```
f, ax1 = plt.subplots(1, 1, figsize=(15,5))

sub_sample_corr = new_df.corr()
sns.heatmap(sub_sample_corr, cmap='coolwarm_r', annot_kws={'size':20}, ax=ax1)
ax1.set_title('SubSample Correlation Matrix \n (for reference)', fontsize=14)
plt.show()
```



SMOTE Technique (Over-Sampling):

- 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 and Applying 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.

Outcomes:

- 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.

Note in Applying:

- If we want to implement cross validation, remember to oversample or undersample your training data **during cross-validation**, not before!
- Don't use accuracy score as a metric with imbalanced datasets (will be usually high and misleading), instead use **f1-score, precision/recall score or confusion matrix**.
- We printed out accuracy for **reference only**.

In [11]:
```
# The target variable is 'Class'.
ignore_col_list = ['Time', 'Class']
metric_col_list = ['Class']
feature_names = pd.DataFrame(list(df), columns = ['a'])
feature_names = feature_names[~feature_names['a'].isin(ignore_col_list)]
feature_names = feature_names['a'].values.tolist()

X = df[feature_names]
y = df[metric_col_list]
X = X.values
y = y.values
#Model training

kf = StratifiedKFold(n_splits = 5)
for fold, (train_index, test_index) in enumerate(kf.split(X, y), 1):
    X_train, y_train = X[train_index], y[train_index]
    X_test, y_test = X[test_index], y[test_index]
    X_train_oversampled, y_train_oversampled = SMOTE().fit_sample(X_train, y_tra
    model = XGBClassifier(learning_rate =0.1, n_estimators=140, max_depth=11,
                  min_child_weight=1, gamma=0, subsample=0.9, colsample_bytr
                  objective= 'binary:logistic', nthread=4, scale_pos_weight=
    model.fit(X_train_oversampled, y_train_oversampled)
    y_pred = model.predict(X_test)
    print(f'For fold {fold}:')
    print(f'Accuracy: {model.score(X_test, y_test)}')
```

```
For fold 1:
Accuracy: 0.9994382219725431
For fold 2:
Accuracy: 0.9995962220427653
For fold 3:
Accuracy: 0.9994733238531627
For fold 4:
Accuracy: 0.9994031003669177
For fold 5:
Accuracy: 0.999420656238479
```

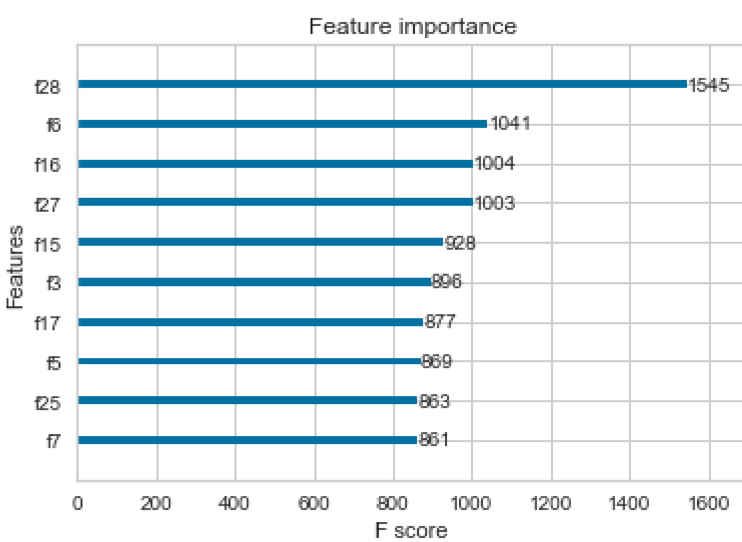
Result:

- Let's look into **fraud cases only**.
- 83% precision, 84% recall and 83% recalls, which I personally think it's pretty good enough!

In [12]:
```
# Evaluate predictions
print('Accuracy of XBG Classifier Model on test set: {:.2%}'
     .format(accuracy_score(y_test, model.predict(X_test))))
print('*' * 60)
print('Confusion Matrix')
print(confusion_matrix(y_test, model.predict(X_test)))
print('*' * 60)
print('Classification Report')
print(classification_report(y_test, model.predict(X_test)))
```

```
Accuracy of XBG Classifier Model on test set: 99.94%
************************************************************
Confusion Matrix
[[56846    17]
 [   16    82]]
************************************************************
Classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56863
           1       0.83      0.84      0.83        98

    accuracy                           1.00     56961
   macro avg       0.91      0.92      0.92     56961
weighted avg       1.00      1.00      1.00     56961
```

In [13]:
```
# The model has been defined as xgboost model in previous tab.
# plot feature importance
from xgboost import plot_importance
from matplotlib import pyplot
plot_importance(model, max_num_features=10) # top 10 most important features
plt.show()
```



Finally, save the model

- Use **Pickle** package to save the trained model.
- To re-load the model, simply loaded_model = pickle.load(open(filename, 'rb'))

In [14]:
```
# save the model to disk
import pickle
filename = 'credit_card_fraud_detection.pkl'
pickle.dump(model, open(filename, 'wb'))
```