

Overview

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'. 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-sensitive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

Inspiration Identify fraudulent credit card transactions.

Given the class imbalance ratio, we are measuring the accuracy using the Area Under the Precision-Recall Curve (AUPRC). Confusion matrix accuracy is not meaningful for unbalanced classification.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
%matplotlib inline
```

```
# Data can be downloaded from https://www.kaggle.com/mlg-ulb/creditcardfraud/downloads/creditcardfraud.zip/3
df= pd.read_csv('/content/Credit card fraud detection.zip')
```

```
df.head()
```

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

5 rows x 31 columns

```
#Null Value Check
```

```
df.isnull().values.any()
```

```
False
```

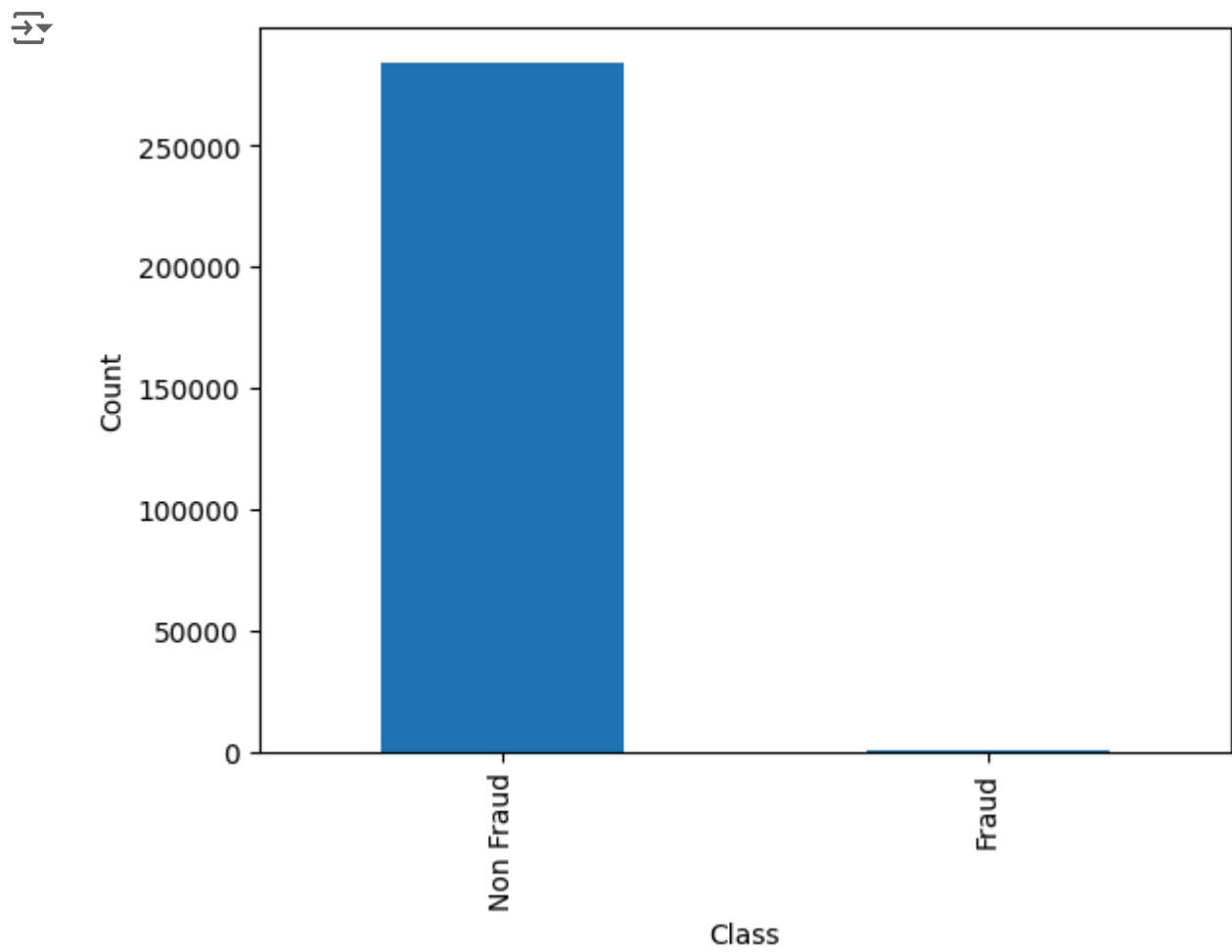
```
#Data Class Balance Check
```

```
print('Fraud Percentage: {}'.format(round((df['Class'].value_counts()[1]/len(df))*100,2)))
```

```
print('Non Fraud Percentage: {}'.format(round((df['Class'].value_counts()[0]/len(df))*100,2)))
```

```
Fraud Percentage: 0.17
Non Fraud Percentage: 99.83
```

```
count= df['Class'].value_counts()
count.plot(kind='bar')
plt.xticks(range(2),['Non Fraud','Fraud'])
plt.xlabel('Class')
plt.ylabel('Count')
plt.show()
```



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!

#

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.

```
fig, ax= plt.subplots(2,1, figsize=(20,10))
```

```
amount= df['Amount'].values
time= df['Time'].values
```

```
sns.distplot(amount,ax=ax[0], color='r')
sns.distplot(time,ax=ax[1],color='b')
```

```
<ipython-input-15-e412a53d57ab>:6: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

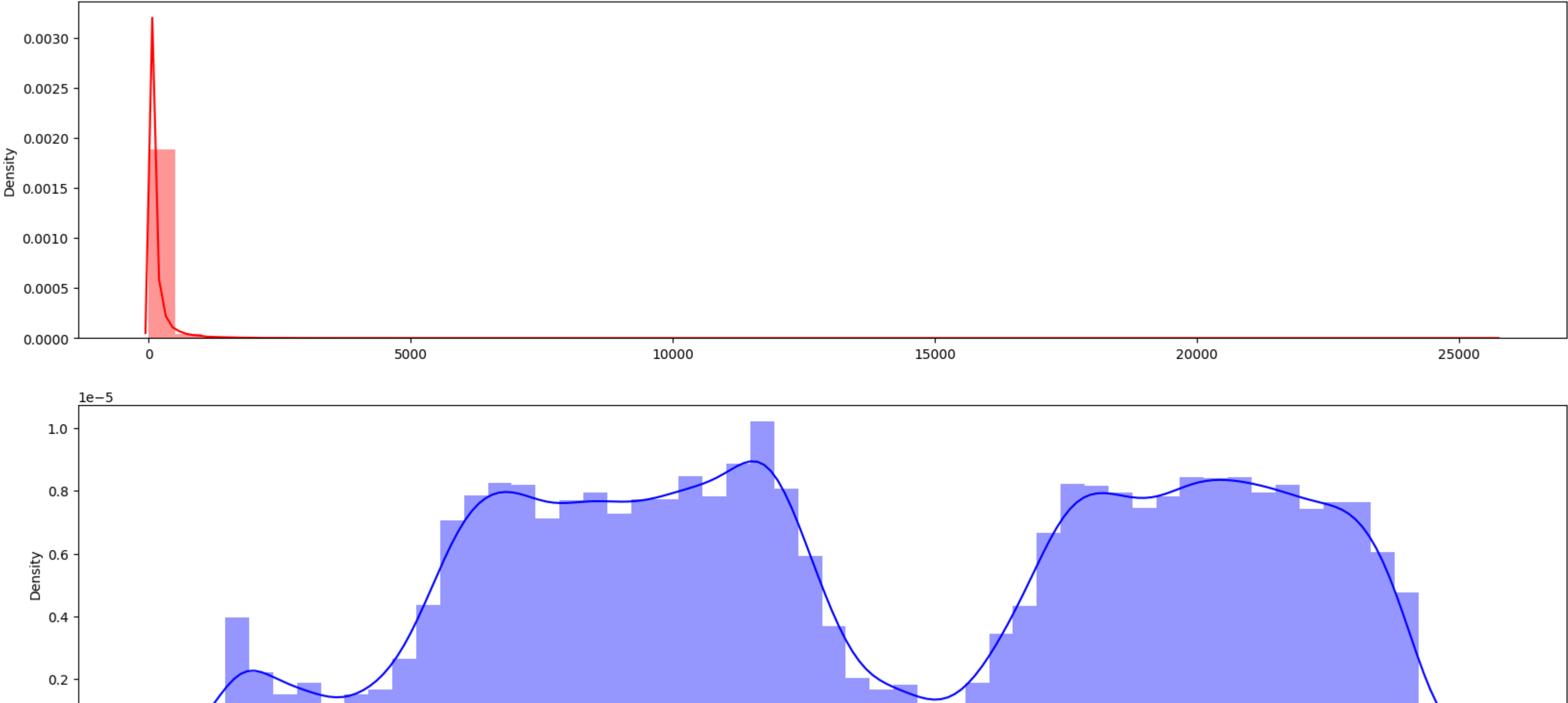
sns.distplot(amount,ax=ax[0], color='r')
<ipython-input-15-e412a53d57ab>:7: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

sns.distplot(time,ax=ax[1],color='b')
<Axes: ylabel='Density'>
```



By distribution we can see transaction amounts are very small, where as time is distributed.

Scaling

As data is given after PCA to hide original data so scalling was done on the variables except time and amount which we will scale

```
from sklearn.preprocessing import RobustScaler # it is prone to outliers
ss1= RobustScaler()
df['Amount']= ss1.fit_transform(df['Amount'].values.reshape(-1, 1))
```

```
ss2= RobustScaler()
df['Time']= ss2.fit_transform(df['Time'].values.reshape(-1, 1))
```

df.head()

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
0	-0.994983	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-0.021053	1.783274	0
1	-0.994983	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983	0.014724	-0.269825	0
2	-0.994972	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-0.059752	4.983721	0
3	-0.994972	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	0.061458	1.418291	0
4	-0.994960	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	0.670579	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.

```
xorg=df.drop('Class',axis=1)
yorg= df.loc[:,'Class']
```

```
from sklearn.model_selection import train_test_split
xorgtrain,xorgtest,yorgtrain,yorgtest= train_test_split(xorg,yorg,test_size=0.2,random_state=9)
```

```
print(xorgtrain.shape,xorgtest.shape,yorgtrain.shape,yorgtest.shape)
```

(227845, 30) (56962, 30) (227845,) (56962,)

Random Sampling

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.

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)

```
#Using imblearn library
# from imblearn.under_sampling import NearMiss

# nm=NearMiss(random_state=9)
# xr,yr= nm.fit_sample(xorg,yorg)

# from collections import Counter
# print('Original Count: {}'.format(Counter(yorg)))
# print('Sampled Count: {}'.format(Counter(yr)))

# # Now we have equal fraud and non fraud data.

# new_df= pd.concat([pd.DataFrame(xr,columns=xorg.columns),pd.DataFrame(yr)],axis=1)

# new_df= new_df.rename({0:'Class'},axis=1)

# new_df.head()
```

Using shuffling and selecting first 492 non fraud

```
# 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] #Taking top 492 row for 0
```

```
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()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class		
	263331	0.895300	-1.464293	-0.830415	0.640442	-3.158653	-0.264022	0.128316	-0.362657	0.607648	-2.418095	...	-0.585805	-1.794969	0.025075	-1.501282	0.515391	-0.546809	0.116766	-0.053893	1.261371	0	
	263274	0.894959	-0.644278	5.002352	-8.252739	7.756915	-0.216267	-2.751496	-3.358857	1.406268	-4.403852	...	0.587728	-0.605759	0.033746	-0.756170	-0.008172	0.532772	0.663970	0.192067	-0.296653	1	
	33429	-0.557619	1.133693	-0.331651	-0.393754	-0.265962	0.195842	0.284746	-0.025223	0.086040	-0.019296	...	0.022245	-0.114212	-0.231298	-1.081747	0.396008	1.116253	-0.098167	-0.012269	0.931181	0	
	10204	-0.809161	-4.641893	2.902086	-1.572939	2.507299	-0.871783	-1.040903	-1.593901	-3.254905	1.908963	...	1.963597	-0.217414	-0.549340	0.645545	-0.354558	-0.611764	-3.908080	-0.671248	-0.148257	1	
	73857	-0.345176	-6.159607	1.468713	-6.850888	5.174706	-2.986704	-1.795054	-6.545072	2.621236	-3.605870	...	1.061314	0.125737	0.589592	-0.568731	0.582825	-0.042583	0.951130	0.158996	-0.295815	1	

5 rows x 31 columns

⌵ Oversampling using RandomOverSampler

```
# from imblearn.over_sampling import RandomOverSampler
# nm= RandomOverSampler(ratio=1,random_state=42)
# xr,yr= nm.fit_sample(xorg,yorg)

# from collections import Counter
# print('Original Count: {}'.format(Counter(yorg)))
# print('Sampled Count: {}'.format(Counter(yr)))

# # Now we have equal fraud and non fraud data.

# new_df= pd.concat([pd.DataFrame(xr,columns=xorg.columns),pd.DataFrame(yr)],axis=1)

# new_df= new_df.rename({0:'Class'},axis=1)

# new_df.head()
```

```
new_df.shape
```

```
(984, 31)
```

```
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')
```

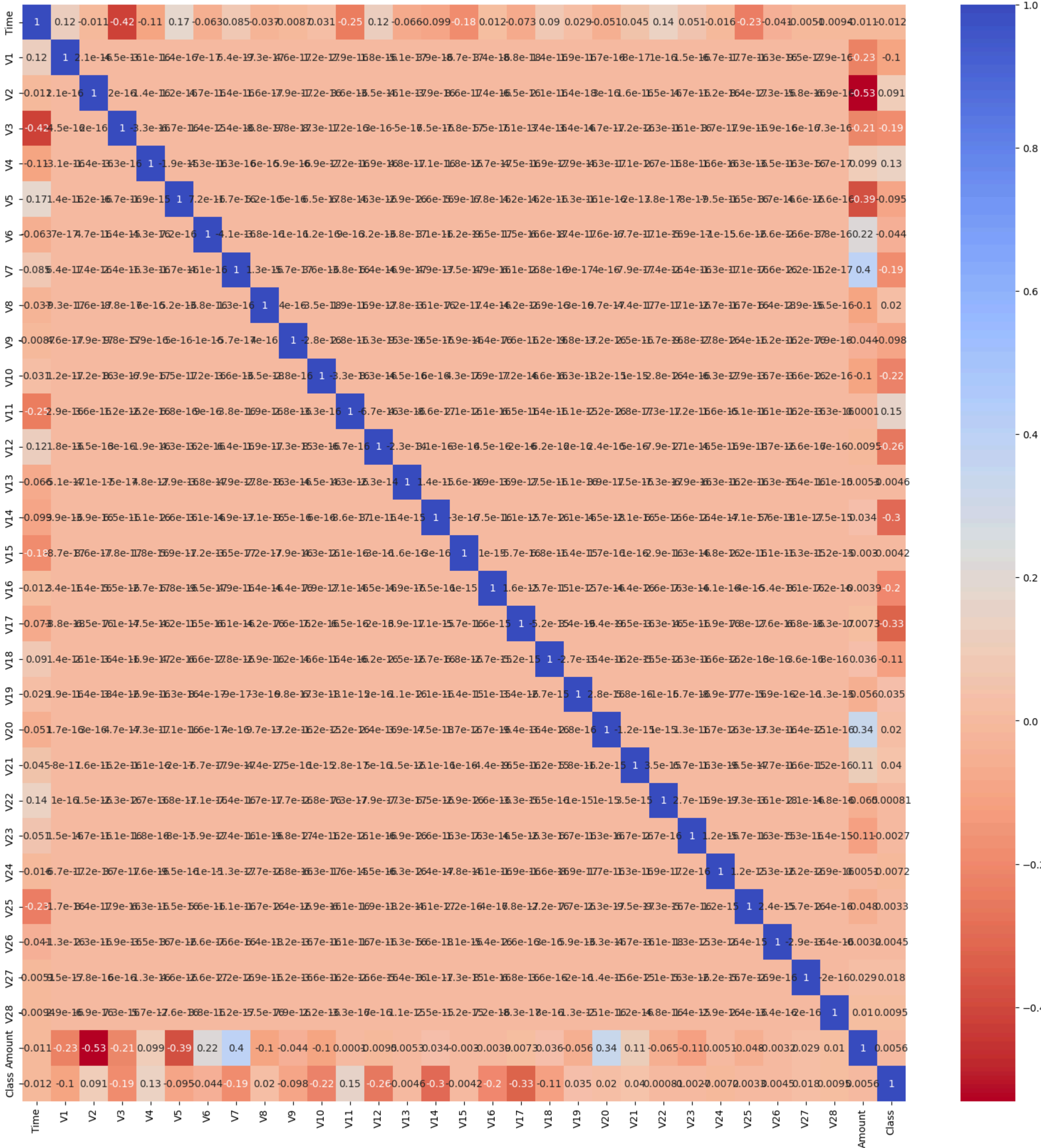
⌵ Correlation Matrices

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.

```
#for Original Data frame
plt.figure(figsize=(20,20))

sns.heatmap(df.corr(),annot=True,cmap='coolwarm_r')
```

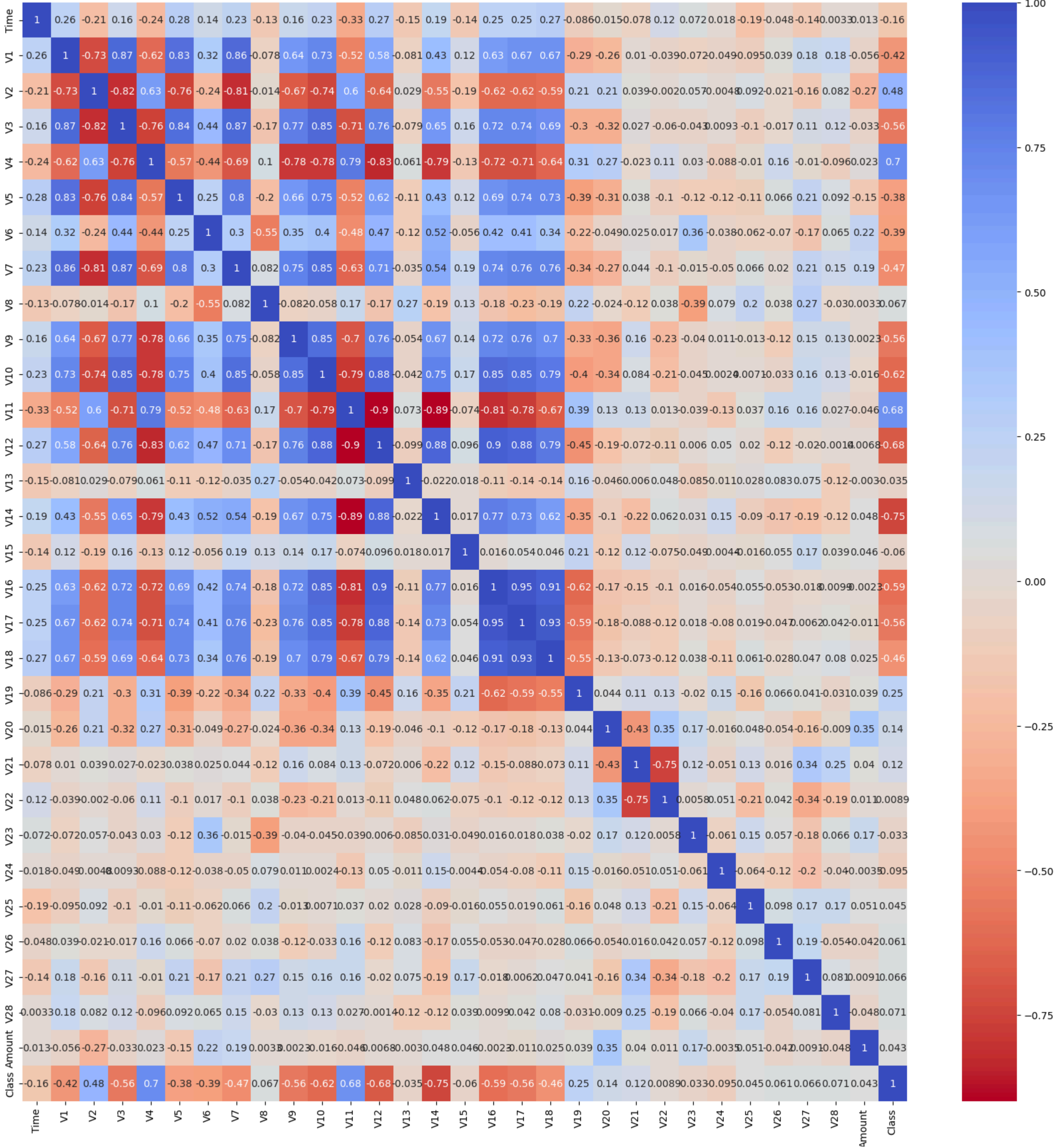




```
#For new sampled df
#for Original Data frame
plt.figure(figsize=(20,20))

sns.heatmap(new_df.corr(),annot=True,cmap='coolwarm_r')
```





Negative Correlations: V17,V16, V14, V12 and V10 are negatively correlated. Notice how the lower these values are, the more likely the end result will be a fraud transaction. Positive Correlations: V2, V4, V11, and V19 are positively correlated. Notice how the higher these values are, the more likely the end result will be a fraud transaction. BoxPlots: We will use boxplots to have a better understanding of the distribution of these features in fradulent and non fradulent transactions.

✖ Negative Correlation

```
f, axes = plt.subplots(ncols=5, 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, ax=axes[0])
axes[0].set_title('V17 vs Class Negative Correlation')

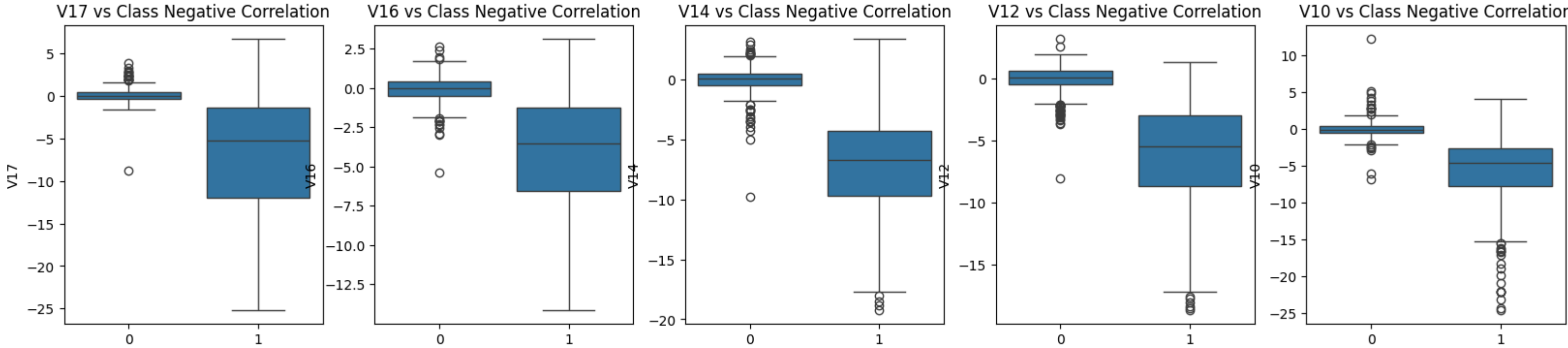
sns.boxplot(x="Class", y="V16", data=new_df, ax=axes[1])
axes[1].set_title('V16 vs Class Negative Correlation')

sns.boxplot(x="Class", y="V14", data=new_df, ax=axes[2])
axes[2].set_title('V14 vs Class Negative Correlation')

sns.boxplot(x="Class", y="V12", data=new_df, ax=axes[3])
axes[3].set_title('V12 vs Class Negative Correlation')

sns.boxplot(x="Class", y="V10", data=new_df, ax=axes[4])
axes[4].set_title('V10 vs Class Negative Correlation')

plt.show()
```



✖ Positive Correlation



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

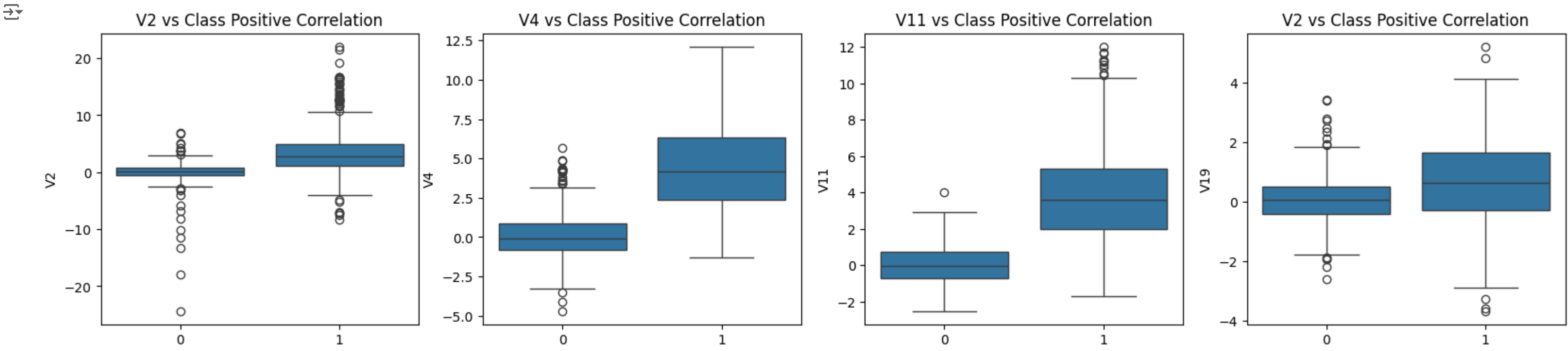
# Postive Correlations with our Class (The higher our feature value the more likely it will be a fraud transaction)
sns.boxplot(x="Class", y="V2", data=new_df, ax=axes[0])
axes[0].set_title('V2 vs Class Positive Correlation')

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

sns.boxplot(x="Class", y="V11", data=new_df, ax=axes[2])
axes[2].set_title('V11 vs Class Positive Correlation')

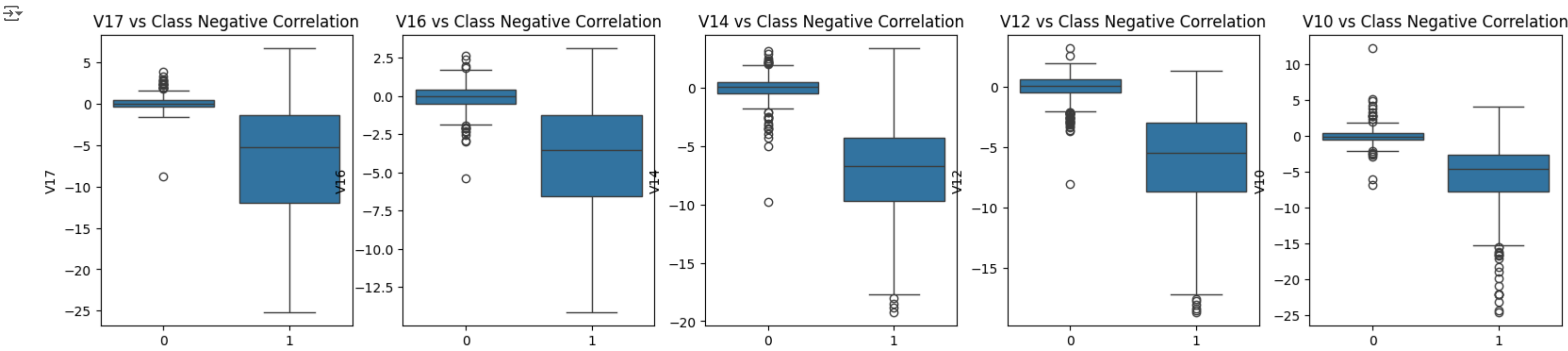
sns.boxplot(x="Class", y="V19", data=new_df, ax=axes[3])
axes[3].set_title('V2 vs Class Positive Correlation')

plt.show()
```



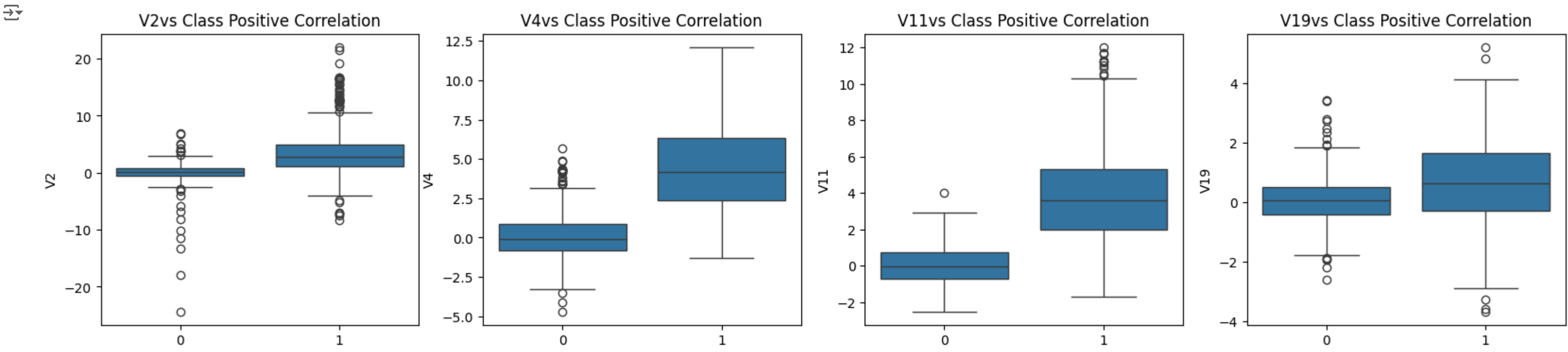
```
neg= ['V17','V16','V14','V12','V10']

f, axes = plt.subplots(ncols=len(neg), figsize=(20,4))
for i,j in enumerate(neg):
# Negative Correlations with our Class (The lower our feature value the more likely it will be a fraud transaction)
    sns.boxplot(x="Class", y=j, data=new_df, ax=axes[i])
    axes[i].set_title(j + ' vs Class Negative Correlation')
```



```
pos= ['V2','V4','V11','V19']

f, axes = plt.subplots(ncols=len(pos), figsize=(20,4))
for i,j in enumerate(pos):
# Postive Correlations with our Class (The higher our feature value the more likely it will be a fraud transaction)
    sns.boxplot(x="Class", y=j, data=new_df, ax=axes[i])
    axes[i].set_title(j+'vs Class Positive Correlation')
```



Boxplots are a standardized way of displaying the distribution of data based on a five number summary ("minimum", first quartile (Q1), median, third quartile (Q3), and "maximum").

median (Q2/50th Percentile): the middle value of the dataset.

first quartile (Q1/25th Percentile): the middle number between the smallest number (not the "minimum") and the median of the dataset.

third quartile (Q3/75th Percentile): the middle value between the median and the highest value (not the "maximum") of the dataset.

interquartile range (IQR): 25th to the 75th percentile.

whiskers (shown in blue)

outliers (shown as green circles)

"maximum": Q3 + 1.5\*IQR

"minimum": Q1 -1.5\*IQR

'outliers= 3\* IQR or more than that

## ✧ Anomly Detection

visualize Distributions: We first start by visualizing the distribution of the feature we are going to use to eliminate some of the outliers. V14 is the only feature that has a Gaussian distribution compared to features V12 and V10. Determining the threshold: After we decide which number we will use to multiply with the iqr (the lower more outliers removed), we will proceed in determining the upper and lower thresholds by substrating q25 - threshold (lower extreme threshold) and adding q75 + threshold (upper extreme threshold). Conditional Dropping: Lastly, we create a conditional dropping stating that if the "threshold" is exceeded in both extremes, the instances will be removed. Boxplot Representation: Visualize through the boxplot that the number of "extreme outliers" have been reduced to a considerable amount.

```
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()
```

```
<ipython-input-33-59787bdd53f6>:6: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

sns.distplot(v14_fraud_dist,ax=ax1, fit=norm, color='#FB8861')
<ipython-input-33-59787bdd53f6>:10: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

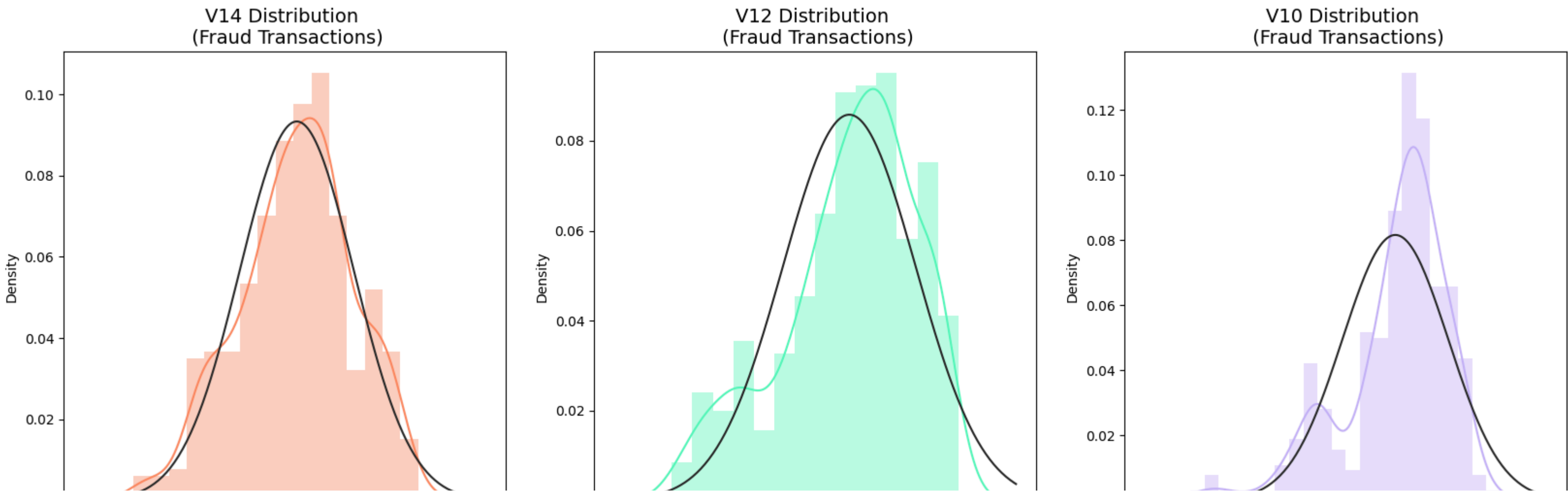
sns.distplot(v12_fraud_dist,ax=ax2, fit=norm, color='#56F9BB')
<ipython-input-33-59787bdd53f6>:15: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

sns.distplot(v10_fraud_dist,ax=ax3, fit=norm, color='#C5B3F9')
```



Outliers Treatment

as we say in box plots all the variables with coorelation have outliers so now we will treat them in with iqr , lb and ub.

```
df2= new_df # I am creating the copy of new_df to preserve the original data
treat= ['V14','V12','V10']
for j in treat:
    q25,q75= new_df[j].quantile(q=0.25),new_df[j].quantile(q=0.75)
    iqr= q75-q25
    cut_off= iqr*1.5
    lb,ub= q25-cut_off,q75+cut_off
    outliers= [x for x in new_df[j] if x<=lb or x>=ub]
    print(j,'Q25: {} , Q75: {}, IQR: {}, Cutoff: {}, LB: {}, UB: {}'.format(q25,q75,iqr,cut_off,lb,ub))
    print(len(outliers), outliers)
    df2= df2.drop(df2[(df2['V14'] > ub) | (df2['V14']< lb)].index, axis=0)
    print(df2.shape)
    print('----' * 44)

# Tried imputing outliers.
# lb,ub= new_df[j].quantile(q=0.05),new_df[j].quantile(q=0.95)
# outliers= [x for x in new_df[j] if x>ub or x<lb]
# print(len(outliers),outliers)
# func= (lambda x: x if x>=ub or x<=lb else new_df[j].mean())
# df2[j]= df2[j].apply(func)

V14 Q25: -6.750004584799415 , Q75: 0.163257785319399, IQR: 6.913262370118813, Cutoff: 10.36989355517822, LB: -17.119898139977636, UB: 10.533151340497618,
8 [-19.2143254902614, -18.8220867423816, -18.4937733551053, -17.7216383537133, -17.230202160711, -17.6206343516773, -17.4759212828566, -18.0499976898594]
(976, 31)

V12 Q25: -5.533784951690725 , Q75: 0.2083650728716845, IQR: 5.7421500245624095, Cutoff: 8.613225036843614, LB: -14.14700998853434, UB: 8.821590109715299,
43 [-18.5536970096458, -17.1829184301947, -15.0226996343749, -14.1750301634055, -15.0941631493865, -15.5923232225286, -15.4790524832016, -16.0603057628826, -14.2254557039818, -15.969207520809, -17.7691434633638, -14.2960914258331, -17.6316063138707,
(958, 31)

V10 Q25: -4.599840755426754 , Q75: -0.005881901635780749, IQR: 4.593958853790974, Cutoff: 6.890938280686461, LB: -11.490779036113215, UB: 6.88505637905068,
79 [-13.2151722995049, -12.9654812039222, -16.3035376590131, -12.7447607871859, -13.6705451263516, -11.589748311433, -12.6959474039839, -14.5571590528859, -15.1237521803455, -11.7868116556041, -15.5637913387301, -12.2653238444006, -11.5619497720699,
(905, 31)
```

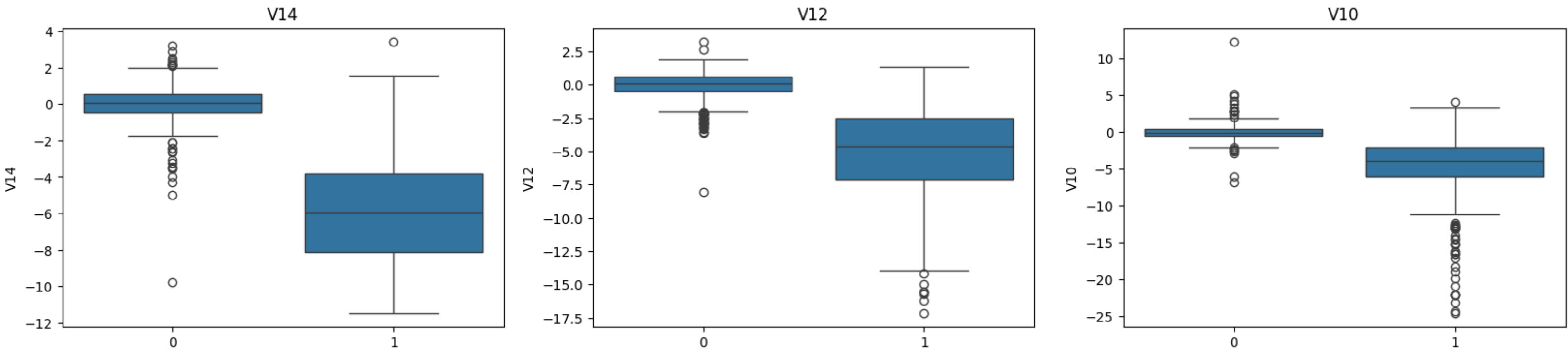
df2.shape

(905, 31)

```
from collections import Counter
Counter(df2['Class'])

Counter({0: 492, 1: 413})

f, axes = plt.subplots(ncols=len(treat), figsize=(20,4))
for i,j in enumerate(treat):
# Postive Correlations with our Class (The higher our feature value the more likely it will be a fraud transaction)
sns.boxplot(x="Class", y=j, data=df2, ax=axes[i])
axes[i].set_title(j)
```



```
from sklearn.model_selection import train_test_split,cross_val_score,cross_val_predict
from sklearn.metrics import roc_auc_score
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier

x=df2.drop('Class',axis=1).values
y= df2.loc[:, 'Class'].values

# SPlitting the test and train after removing outliers


xtrain,xtest,ytrain,ytest=train_test_split(x,y,test_size=0.2,random_state=42)
```

Fitting the models and calculating test and training score

```
classifier= {
    'Logistic Regression':LogisticRegression(),
    'KNN':KNeighborsClassifier(),
    'SVC':SVC(),
    'DecisionTree':DecisionTreeClassifier()
}

import warnings
warnings.filterwarnings('ignore')
```

```
for key,values in classifier.items():
    values.fit(xtrain,ytrain)
    training_score= cross_val_score(values,xtrain,ytrain,cv=5)
    print('Training accuracy score of {} is {}'.format(key,round(training_score.mean()*100,2)))
    train_pred = cross_val_predict(values, xtrain, ytrain, cv=5)
    print('Roc_Auc training score for {} is {}: '.format(key, round(roc_auc_score(ytrain,train_pred)*100,2)))
    test_score= cross_val_score(values,xtest,ytest,cv=5)
    print('Test accuracy score of {} is {}'.format(key,round(test_score.mean()*100,2)))
    test_pred = cross_val_predict(values, xtest, ytest, cv=5)
    print('Roc_Auc test score for {} is {}: '.format(key, round(roc_auc_score(ytest,test_pred)*100,2)))
    print('---'*30)
```

 Training accuracy score of Logistic Regression is 91.57  
Roc\_Auc training score for Logistic Regression is 91.22:  
Test accuracy score of Logistic Regression is 89.5  
Roc\_Auc test score for Logistic Regression is 89.36:

-----

Training accuracy score of KNN is 91.85  
Roc\_Auc training score for KNN is 91.4:  
Test accuracy score of KNN is 89.5  
Roc\_Auc test score for KNN is 89.23:

-----

Training accuracy score of SVC is 91.71  
Roc\_Auc training score for SVC is 91.22:  
Test accuracy score of SVC is 90.05  
Roc\_Auc test score for SVC is 89.55:


-----

Training accuracy score of DecisionTree is 89.09  
Roc\_Auc training score for DecisionTree is 90.07:  
Test accuracy score of DecisionTree is 87.28  
Roc\_Auc test score for DecisionTree is 85.58:

-----

↳ Calculating for original data

```
for key,values in classifier.items():
    values.fit(xtrain,ytrain)
    test_score= cross_val_score(values,xorgtest,yorgtest,cv=5)
    print('Test accuracy score of {} is {}'.format(key,round(test_score.mean()*100,2)))
    test_pred = cross_val_predict(values, xorgtest,yorgtest, cv=5)
    print('Roc_Auc test score for {} is {}: '.format(key, round(roc_auc_score(yorgtest,test_pred)*100,2)))
    print('---'*30)
```

 Test accuracy score of Logistic Regression is 99.93  
Roc\_Auc test score for Logistic Regression is 83.16:

-----

Test accuracy score of KNN is 99.92  
Roc\_Auc test score for KNN is 84.15:

-----

Test accuracy score of SVC is 99.91  
Roc\_Auc test score for SVC is 75.74:

-----

Test accuracy score of DecisionTree is 99.9  
Roc\_Auc test score for DecisionTree is 85.62:

-----

↳ Hyper Parameter Tuning using GridSearchCv


```
# 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]}
knears_params = {"n_neighbors": list(range(2,5,1)), 'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute']}
svc_params = {'C': [0.5, 0.7, 0.9, 1], 'kernel': ['rbf', 'poly', 'sigmoid', 'linear']}
tree_params = {"criterion": ["gini", "entropy"], "max_depth": list(range(2,4,1)),
               "min_samples_leaf": list(range(5,7,1))}

classifier= {
    'Logistic Regression':LogisticRegression(),
    'KNN':KNeighborsClassifier(),
    'SVC':SVC(),
    'DecisionTree':DecisionTreeClassifier()
}


def grid_search(classifier,Param):
    grid_log_reg = GridSearchCV(classifier,param_grid=Param)
    grid_log_reg.fit(xtrain, ytrain)
    best_param = grid_log_reg.best_estimator_
    print('{} algorithm best parameter are : {}'.format(classifier.__class__.__name__,best_param))
```

```
grid_search(LogisticRegression(),log_reg_params)
```

 LogisticRegression algorithm best parameter are : LogisticRegression(C=0.1)

```
log_reg = LogisticRegression(C=1, class_weight=None, dual=False, fit_intercept=True,
                             intercept_scaling=1, l1_ratio=None, max_iter=100,
                             multi_class='ovr', n_jobs=None, penalty='l1',
                             random_state=None, solver='liblinear', tol=0.0001,
                             verbose=0, warm_start=False)

log_reg.fit(xtrain,ytrain)
```



LogisticRegression

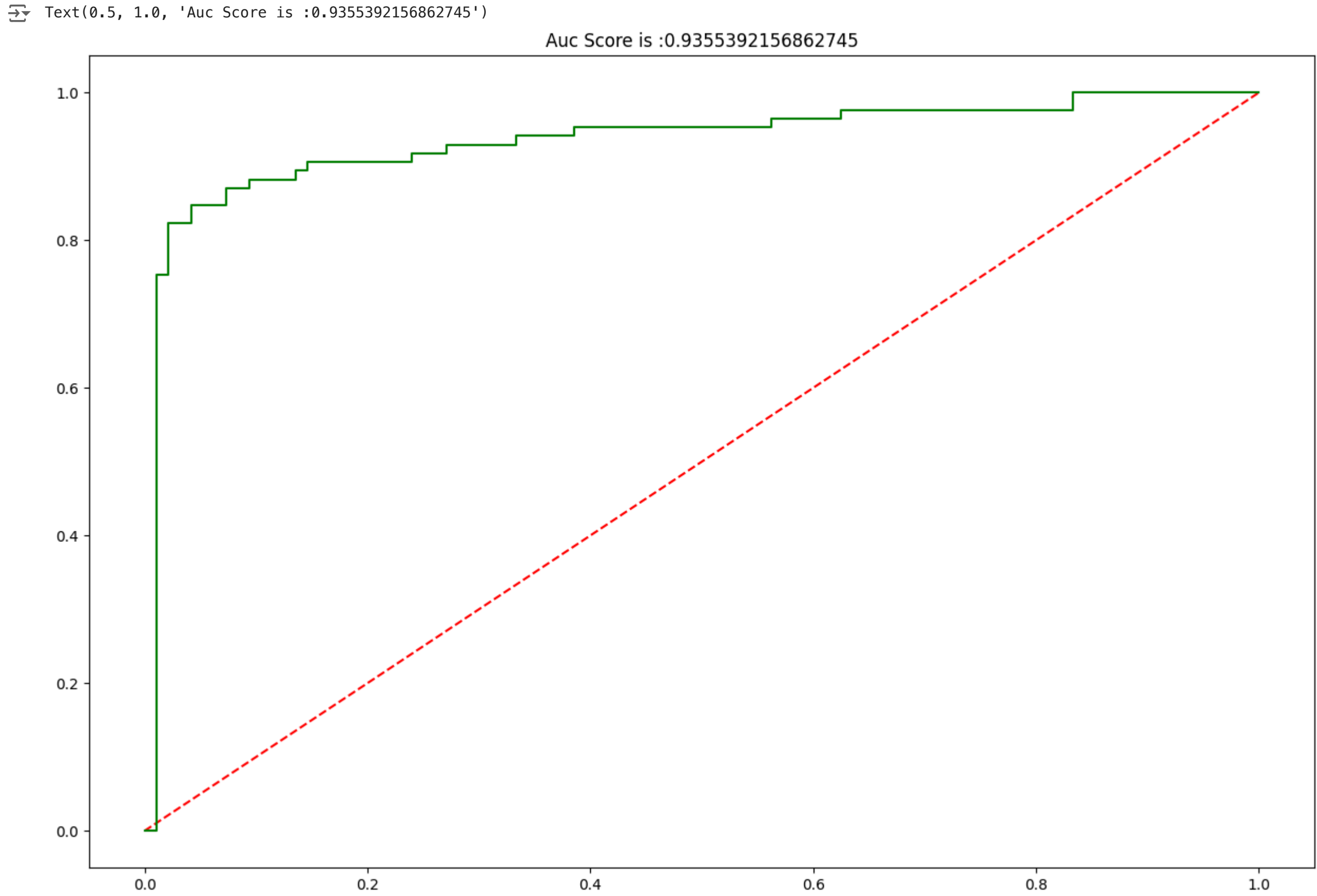
LogisticRegression(C=1, multi\_class='ovr', penalty='l1', solver='liblinear')

```
log_reg_pred = cross_val_predict(log_reg, xtest, ytest, cv=5,method="decision_function")
```

```
from sklearn.metrics import roc_curve, auc
fpr,tpr,threshold= roc_curve(ytest,log_reg_pred)
```

```
plt.figure(figsize=(15,10))
plt.plot([0,1],[0,1], 'r--')
plt.plot(fpr,tpr, 'g')
plt.title('Auc Score is :'+str(auc(fpr,tpr)))
```





Start coding or [generate](#) with AI.

✖ Deeper look into the logistic regression classifier.

True Positives: Correctly Classified Fraud Transactions  
False Positives: Incorrectly Classified Fraud Transactions  
Negative: Correctly Classified Non-Fraud Transactions  
False Negative: Incorrectly Classified Non-Fraud Transactions  
Precision (1-Specificity): True Positives/(True Positives + False Positives) i.e ratio of correct predicted positive to the total predicted positive.  
Recall or Sensitivity: True Positives/(True Positives + False Negatives) i.e what percentage of fraud is correctly identified  
F1 Score = 2\*(Recall \* Precision) / (Recall + Precision) is the weighted average of Precision and Recall.  
Accuracy = TP+TN/TP+FP+FN+TN = TP/TP+FN  
Specificity= TN/TN+FP i.e what percent is of non fraud is correctly identified  
Precision as the name says, says how precise (how sure) is our model in detecting fraud transactions while recall is the amount of fraud cases our model is able to detect.

Precision/Recall Tradeoff: The more precise (selective) our model is, the less cases it will detect. Example: Assuming that our model has a precision of 95%, Let's say there are only 5 fraud cases in which the model is 95% precise or more that these are fraud cases. Then let's say there are 5 more cases that our model considers 90% to be a fraud case, if we lower the precision there are more cases that our model will be able to detect.

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import recall_score, precision_score, f1_score, accuracy_score

# Define and configure the Logistic Regression model
log_reg = LogisticRegression(C=1, class_weight=None, dual=False, fit_intercept=True,
                             intercept_scaling=1, l1_ratio=None, max_iter=100,
                             multi_class='ovr', n_jobs=None, penalty='l1',
                             random_state=None, solver='liblinear', tol=0.0001,
                             verbose=0, warm_start=False)

# Fit the model with the training data
log_reg.fit(xtrain, ytrain)

# Predict on the test set
y_pred = log_reg.predict(xtest)

# Calculate and print evaluation metrics
print('Recall Score: {:.2f}'.format(recall_score(ytest, y_pred)))
print('Precision Score: {:.2f}'.format(precision_score(ytest, y_pred)))
print('F1 Score: {:.2f}'.format(f1_score(ytest, y_pred)))
print('Accuracy Score: {:.2f}'.format(accuracy_score(ytest, y_pred)))
```

↗ Recall Score: 0.86  
Precision Score: 0.94  
F1 Score: 0.90  
Accuracy Score: 0.91

✖ Testing on original dataset

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

print('Recall Score: {:.2f}'.format(recall_score(yorgtest, y_pred)))
print('Precision Score: {:.2f}'.format(precision_score(yorgtest, y_pred)))
print('F1 Score: {:.2f}'.format(f1_score(yorgtest, y_pred)))
print('Accuracy Score: {:.2f}'.format(accuracy_score(yorgtest, y_pred)))
```

↗ Recall Score: 0.90  
Precision Score: 0.06  
F1 Score: 0.12  
Accuracy Score: 0.98

so while predicting on original test values it has an unaccepted accuracy which shows our model is over fitted. This over fitting occurred because we did the sampling before cross validating.

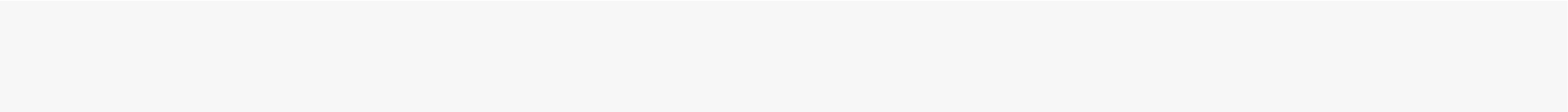
So we will do undersampling during cross validation, which will be correct value for down sampling

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.you will see amazing precision and recall scores but in reality our data is overfitting!

Below I am doing undersampling during cross validation but still accuracy wont be good, as in undersampling we loose the information.

- 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:



```
import numpy as np
from collections import Counter
from sklearn.model_selection import StratifiedKFold, RandomizedSearchCV
from imblearn.pipeline import make_pipeline as imbalanced_make_pipeline
from imblearn.under_sampling import NearMiss
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import recall_score, precision_score, f1_score, accuracy_score, roc_auc_score

# Assuming df is your DataFrame containing the data
undersample_X = df.drop('Class', axis=1)
undersample_y = df['Class']

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

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_resample(undersample_X.values, undersample_y.values)
print('NearMiss Label Distribution: {}'.format(Counter(y_nearmiss)))


# Define parameter grid for Logistic Regression
log_reg_params = {
    "penalty": ['l1', 'l2'],
    'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],
    'solver': ['liblinear', 'saga'], # Only solvers that support 'l1' penalty
    'multi_class': ['ovr', 'auto'] # Valid values for multi_class
}


rand_log_reg1 = RandomizedSearchCV(LogisticRegression(), log_reg_params, n_iter=4)


# Cross Validating the right way
for train, test in sss.split(undersample_Xtrain, undersample_ytrain):
    undersample_pipeline = imbalanced_make_pipeline(NearMiss(sampling_strategy='majority'), rand_log_reg1)
    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(undersample_Xtrain[test], undersample_ytrain[test]))
    undersample_precision.append(precision_score(undersample_ytrain[test], undersample_prediction))
    undersample_recall.append(recall_score(undersample_ytrain[test], undersample_prediction))
    undersample_f1.append(f1_score(undersample_ytrain[test], undersample_prediction))
    undersample_auc.append(roc_auc_score(undersample_ytrain[test], undersample_prediction))


print('--' * 45)
print("accuracy: {}".format(np.mean(undersample_accuracy)))
print("precision: {}".format(np.mean(undersample_precision)))
print("recall: {}".format(np.mean(undersample_recall)))
print("f1: {}".format(np.mean(undersample_f1)))
print("AUC: {}".format(np.mean(undersample_auc)))
print('--' * 45)
```


 Train: [ 56960 56961 56962 ... 284804 284805 284806] Test: [ 0 1 2 ... 56959 57120 57133]


 Train: [ 0 1 2 ... 284804 284805 284806] Test: [ 56960 56961 56962 ... 118991 119721 120262]


 Train: [ 0 1 2 ... 284804 284805 284806] Test: [113911 113912 113913 ... 174022 175541 175592]


 Train: [ 0 1 2 ... 284804 284805 284806] Test: [170874 170875 170876 ... 227848 227849 227850]


 Train: [ 0 1 2 ... 227848 227849 227850] Test: [226747 226866 226870 ... 284804 284805 284806]


 NearMiss Label Distribution: Counter({0: 492, 1: 492})


 -----


 accuracy: 0.5914701603044465

 precision: 0.004065441180996422


 recall: 0.9491723466407009

 f1: 0.008095754796972412

 AUC: 0.7700113288613244

 -----

```
from sklearn.metrics import classification_report
labels = ['No Fraud', 'Fraud']
nm_prediction = rand_log_reg1.best_estimator_.predict(undersample_Xtest)
print(classification_report(undersample_ytest, nm_prediction, target_names=labels))
```



	precision	recall	f1-score	support
No Fraud	1.00	0.63	0.77	56863
Fraud	0.00	0.91	0.01	98
accuracy			0.63	56961
macro avg	0.50	0.77	0.39	56961
weighted avg	1.00	0.63	0.77	56961

```
best= rand_log_reg1.best_estimator_
y_score = best.decision_function(xorgtest)

from sklearn.metrics import average_precision_score
average_precision = average_precision_score(yorgtest, y_score)

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

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


from sklearn.metrics import precision_recall_curve

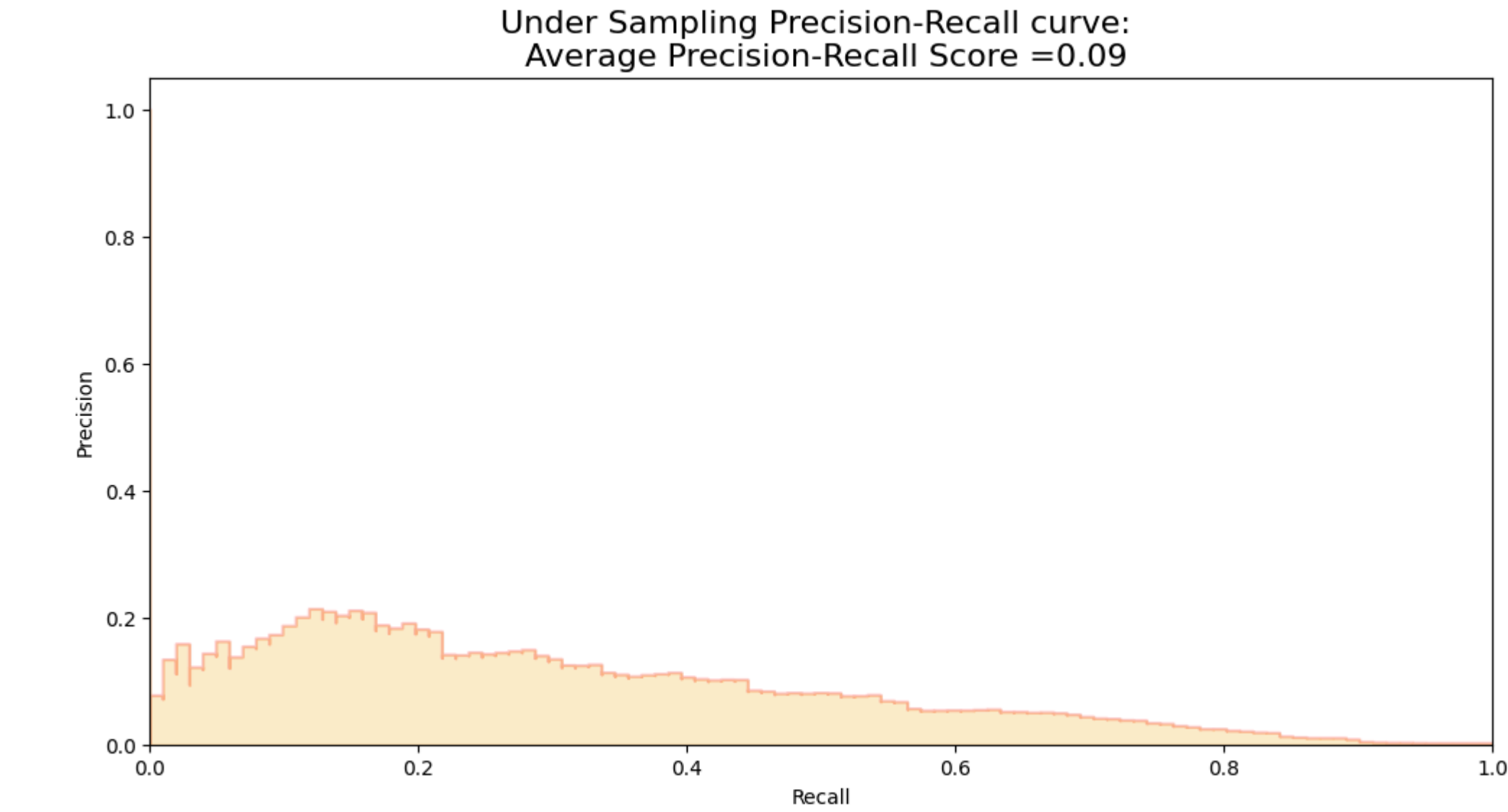
precision, recall, threshold = precision_recall_curve(yorgtest, y_score)

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('Under Sampling Precision-Recall curve: \n Average Precision-Recall Score = {0:0.2f}'.format(
    average_precision), fontsize=16)
```



 Average precision-recall score: 0.09  
Text(0.5, 1.0, 'Under Sampling Precision-Recall curve: \n Average Precision-Recall Score =0.09')



⌵ Now we will increase our data points so that we dont losse any informations

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. SMOTE will be done "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.

```
from sklearn.model_selection import train_test_split, RandomizedSearchCV, StratifiedKFold
from imblearn.pipeline import make_pipeline as imbalanced_make_pipeline
from imblearn.over_sampling import SMOTE

print('Length of X (train): {} | Length of y (train): {}'.format(len(xorgtrain), len(yorgtrain)))
print('Length of X (test): {} | Length of y (test): {}'.format(len(xorgtest), len(yorgtest)))


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

sss = StratifiedKFold(n_splits=5, random_state=None, shuffle=False)
# 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(xorgtrain, yorgtrain):
    pipeline = imbalanced_make_pipeline(SMOTE(sampling_strategy='minority'), rand_log_reg)
    # SMOTE happens during Cross Validation not before..
    model = pipeline.fit(xorgtrain.iloc[train], yorgtrain.iloc[train])
    best_est = rand_log_reg.best_estimator_
    prediction = best_est.predict(xorgtrain.iloc[test])

    accuracy_lst.append(pipeline.score(xorgtrain.iloc[test], yorgtrain.iloc[test]))
    precision_lst.append(precision_score(yorgtrain.iloc[test], prediction))
    recall_lst.append(recall_score(yorgtrain.iloc[test], prediction))
    f1_lst.append(f1_score(yorgtrain.iloc[test], prediction))
    auc_lst.append(roc_auc_score(yorgtrain.iloc[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): 227845 | Length of y (train): 227845  
Length of X (test): 56962 | Length of y (test): 56962

-----

accuracy: 0.9738243103864468  
precision: 0.05634405287211528  
recall: 0.9027588445309963  
f1: 0.10603537097238376

-----

```
from sklearn.metrics import classification_report
labels = ['No Fraud', 'Fraud']
smote_prediction = best_est.predict(xorgtest)
print(classification_report(yorgtest, smote_prediction, target_names=labels))
```



	precision	recall	f1-score	support
No Fraud	1.00	0.97	0.99	56861
Fraud	0.05	0.91	0.10	101
accuracy			0.97	56962
macro avg	0.53	0.94	0.54	56962
weighted avg	1.00	0.97	0.98	56962

```
y_score = best_est.decision_function(xorgtest)

from sklearn.metrics import average_precision_score
average_precision = average_precision_score(yorgtest, y_score)

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

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

from sklearn.metrics import precision_recall_curve

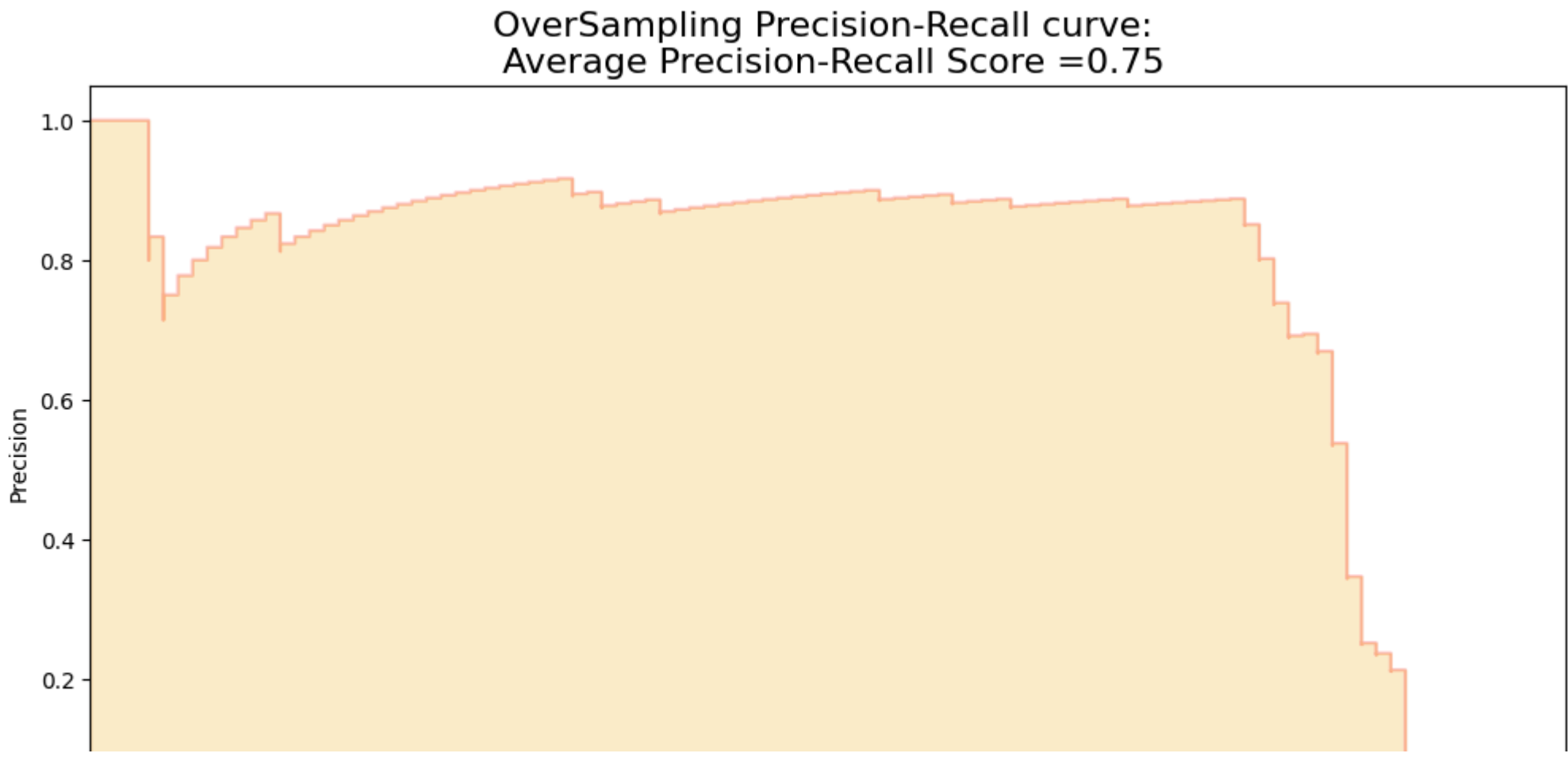
precision, recall, threshold = precision_recall_curve(yorgtest, y_score)

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)
```

↗

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



Neural network testing for Under and Over Sample data.

I am making simple two hidden layer network and will use it

```
import keras
from keras.models import Sequential
from keras.layers import Dense

import warnings
warnings.filterwarnings('ignore')

classifier= Sequential()
classifier.add(Dense(15, activation='relu',kernel_initializer='uniform',input_shape=(30,)))
classifier.add(Dense(15, activation='relu',kernel_initializer='uniform'))
classifier.add(Dense(1, activation='sigmoid', kernel_initializer='uniform' ))

classifier.summary()
```

↗ Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 15)	465
dense_1 (Dense)	(None, 15)	240
dense_2 (Dense)	(None, 1)	16
Total params: 721 (2.82 KB)		
Trainable params: 721 (2.82 KB)		
Non-trainable params: 0 (0.00 Byte)		

```
classifier.compile(optimizer='adam',loss= ['binary_crossentropy'],metrics=['accuracy'])
```

```
from imblearn.under_sampling import NearMiss

# Assuming df is your DataFrame containing the data
X = df.drop('Class', axis=1)
y = df['Class']

# Applying NearMiss technique to undersample the data
X_nearmiss, y_nearmiss = NearMiss().fit_resample(X, y)

# Assuming classifier is a neural network model (e.g., from Keras)
```