Double-click (or enter) to edit

Project Context

This project is about the importance for credit card companies to spot fake transactions, so customers don't pay for things they didn't buy. The information shared comes from a study of credit card purchases made in September 2013 by people in Europe. Over two days, there were 284,807 transactions, and 492 of these were frauds, which is a very small part of all the transactions (0.172%).

Data Overview

The data shared only includes numbers that have been changed for privacy reasons, using a method called PCA, except for two details: 'Time' and 'Amount'. 'Time' shows how many seconds passed between each purchase and the first one in the data. 'Amount' is how much the transaction was. These details can help in learning about fraud in a more detailed way. The data also marks each transaction as fraud or not with a 'Class' feature, where 1 means fraud and 0 means no fraud.

Objective

The goal of the project is to understand complex transaction data and find patterns that show fraud. By applying and comparing a variety of machine learning and deep learning methodologies, we aim to not only detect fraudulent transactions but to open avenues for developing robust, scalable fraud detection systems.

Acknowledgments

This dataset was put together and examined through a joint effort between Worldline and the Machine Learning Group at ULB (Université Libre de Bruxelles), focusing on digging into large sets of data and spotting fraud. You can find more information about what they're currently working on and their past projects related to this subject at their website http://mlg.ulb.ac.be, and for further details, you can visit the Fraud Detection project's page at https://www.researchgate.net/project/Fraud-detection-5, as well as the DefeatFraud project's webpage.

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

```
import sklearn
print(sklearn.__version__)
```

```
1.2.2
```

```
path ="/content/drive/MyDrive/Datasets/creditcard_data.csv"
df = pd.read_csv (path,sep=',')
df.head()
```

	Time	V1	V2	V3	V4	V5	V6	V7	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.0
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.0
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.2
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.3
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.2

5 rows × 31 columns

```
#Checking data types and missing values
print(df.info())

#Summarizing numerical data
print(df.describe())
```

```
memory usage: 67.4 MB
None
                Time             V1            V2             V3  \
count  284806.000000  284806.000000  2.848060e+05  284806.000000
mean    94813.585781       0.000002  6.661837e-07      -0.000002
std     47488.004530       1.958699  1.651311e+00       1.516257
min         0.000000     -56.407510 -7.271573e+01     -48.325589
25%     54201.250000      -0.920374 -5.985522e-01      -0.890368
50%     84691.500000       0.018109  6.549621e-02       0.179846
75%    139320.000000       1.315645  8.037257e-01       1.027198
max    172788.000000       2.454930  2.205773e+01       9.382558

                  V4            V5             V6             V7  \
count  284806.000000  2.848060e+05  284806.000000  284806.000000
mean        0.000002  4.405008e-08       0.000002      -0.000006
std         1.415871  1.380249e+00       1.332273       1.237092
min        -5.683171 -1.137433e+02     -26.160506     -43.557242
25%        -0.848642 -6.915995e-01      -0.768296      -0.554080
50%        -0.019845 -5.433621e-02      -0.274186       0.040097
75%         0.743348  6.119267e-01       0.398567       0.570426
max        16.875344  3.480167e+01      73.301626     120.589494

                  V8             V9  ...           V21            V22  \
count  284806.000000  284806.000000  ...  2.848060e+05  284806.000000
mean        0.000001      -0.000002  ... -9.166149e-07      -0.000002
std         1.194355       1.098634  ...  7.345251e-01       0.725702
min       -73.216718     -13.434066  ... -3.483038e+01     -10.933144
25%        -0.208628      -0.643098  ... -2.283974e-01      -0.542351
50%         0.022358      -0.051429  ... -2.945020e-02       0.006781
75%         0.327346       0.597140  ...  1.863701e-01       0.528548
max        20.007208      15.594995  ...  2.720284e+01      10.503090

                 V23           V24            V25            V26  \
count  284806.000000  2.848060e+05  284806.000000  284806.000000
mean       -0.000001 -3.088756e-08       0.000002       0.000003
std         0.624461  6.056481e-01       0.521278       0.482225
min       -44.807735 -2.836627e+00     -10.295397      -2.604551
25%        -0.161846 -3.545895e-01      -0.317142      -0.326979
50%        -0.011196  4.097671e-02       0.016596      -0.052134
75%         0.147641  4.395270e-01       0.350716       0.240955
max        22.528412  4.584549e+00       7.519589       3.517346

                V27           V28         Amount          Class
count  2.848060e+05  2.848060e+05  284806.000000  284806.000000
mean   8.483873e-09 -4.792707e-08      88.349168       0.001727
std    4.036332e-01  3.300838e-01     250.120432       0.041527
min   -2.256568e+01 -1.543008e+01       0.000000       0.000000
25%   -7.083961e-02 -5.295995e-02       5.600000       0.000000
50%    1.342244e-03  1.124381e-02      22.000000       0.000000
75%    9.104579e-02  7.828043e-02      77.160000       0.000000
max    3.161220e+01  3.384781e+01   25691.160000       1.000000

[8 rows x 31 columns]
```

```
#Check for missing values
print(df.isnull().sum())
```

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

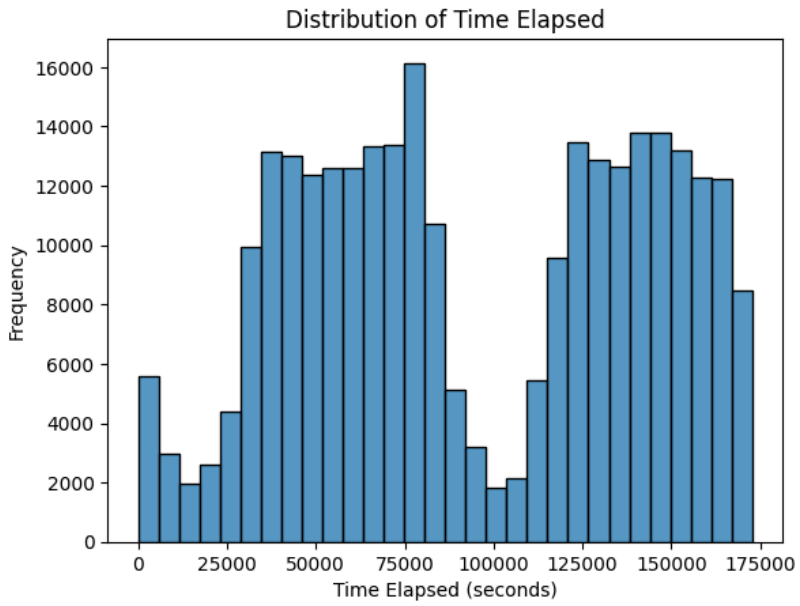
```
Class     0
dtype: int64
```

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

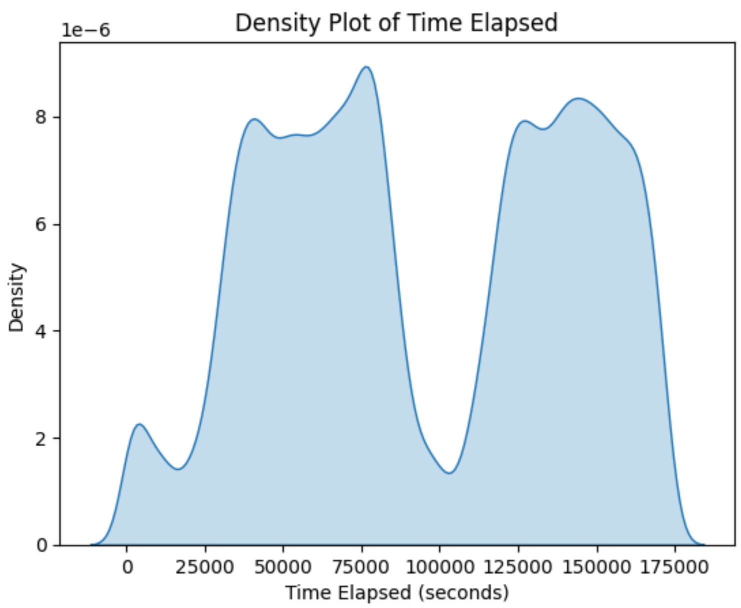
```
False
```

EDA - Exploratory Data Analysis

```
sns.histplot(df.Time, bins=30)
plt.xlabel('Time Elapsed (seconds)')
plt.ylabel('Frequency')
plt.title('Distribution of Time Elapsed')
plt.show()
```

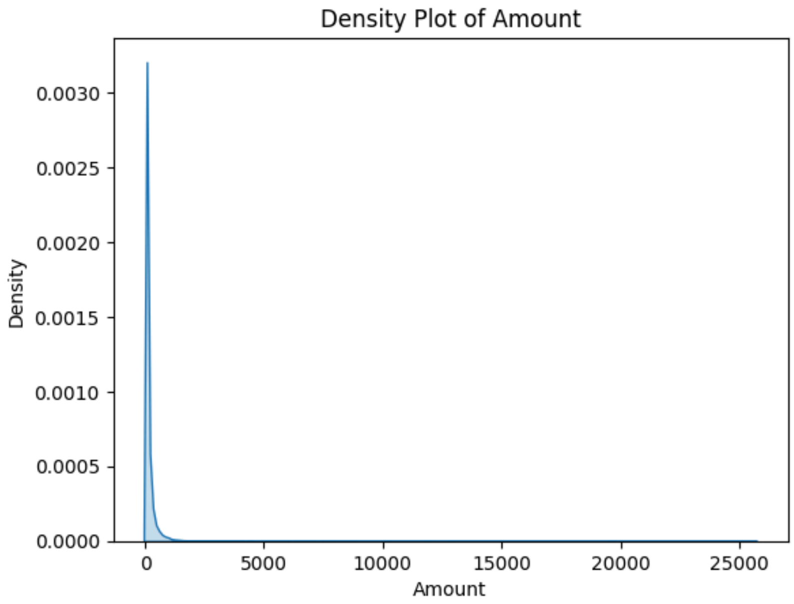



```
sns.kdeplot(data=df['Time'], fill=True)
plt.xlabel('Time Elapsed (seconds)')
plt.ylabel('Density')
plt.title('Density Plot of Time Elapsed')
plt.show()
```



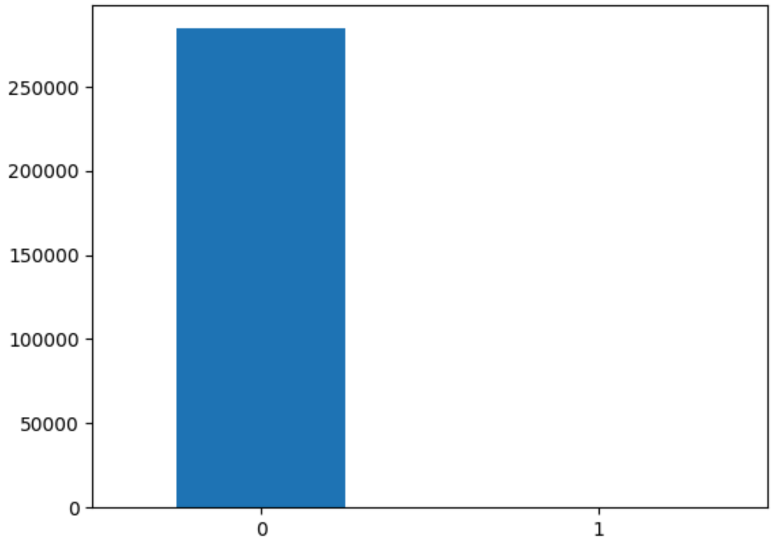

```
sns.kdeplot(data=df['Amount'], fill=True)
plt.title('Density Plot of Amount')
```

```
Text(0.5, 1.0, 'Density Plot of Amount')
```

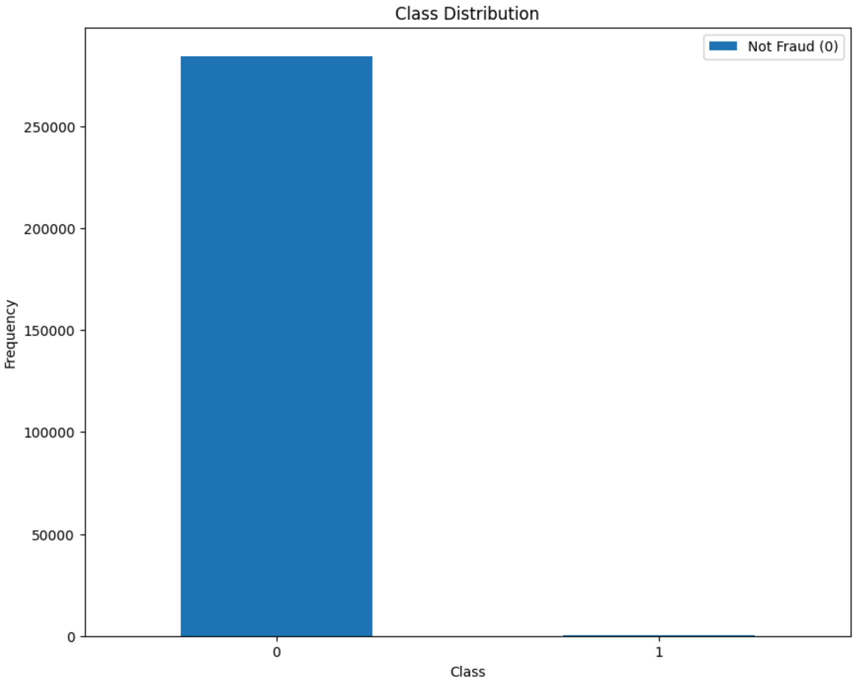



```
count_classes = pd.value_counts(df['Class'],sort= True)

count_classes.plot(kind = 'bar', rot=0)
```

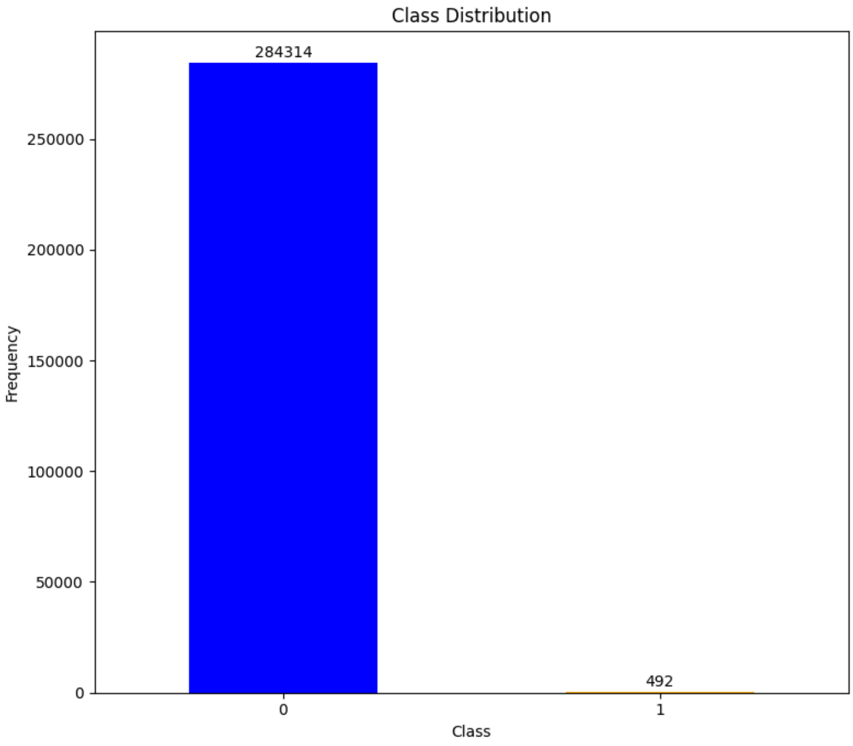
```
<Axes: >
```




```
import pandas as pd
import matplotlib.pyplot as plt

# Assuming 'df' is your DataFrame and 'Class' is the column with fraud labels

# Count the occurrences of each class
count_classes = pd.value_counts(df['Class'], sort=True)

# Define the size of the figure
plt.figure(figsize=(10, 8))  # Adjust the size as needed

# Create the bar plot
count_classes.plot(kind='bar', rot=0)

# Add labels and title
plt.title('Class Distribution')  # Add a title
plt.xlabel('Class')  # Add x-axis label
plt.ylabel('Frequency')  # Add y-axis label

# Add a legend
plt.legend(["Not Fraud (0)", "Fraud (1)"])  # Add a legend to clarify which bar is which

# Show the plot with all the enhancements
plt.show()
```




```
import pandas as pd
import matplotlib.pyplot as plt

# Assuming 'df' is your DataFrame and 'Class' is the column with fraud labels

# Count the occurrences of each class
count_classes = pd.value_counts(df['Class'], sort=True)

# Define the size of the figure
plt.figure(figsize=(8, 7))  # Adjust the size as needed

# Create the bar plot
ax = count_classes.plot(kind='bar', rot=0, color=['blue', 'orange'])

# Add labels and title
plt.title('Class Distribution')  # Add a title
plt.xlabel('Class')  # Add x-axis label
plt.ylabel('Frequency')  # Add y-axis label

# Annotate the bars with the count
for i in ax.patches:
    ax.text(i.get_x() + i.get_width()/2, i.get_height()+1000, str(i.get_height()), ha='center', va='bottom')

# Show the plot with all the enhancements
plt.tight_layout()
plt.show()
```




```
Start coding or generate with AI.
```

```
df_fraud= df[df['Class']==1]
df_normal= df[df['Class']==0]
```

```
print(df_fraud.shape,df_normal.shape)
```

```
(492, 31) (284314, 31)
```

```
df_fraud.Amount.describe()
```

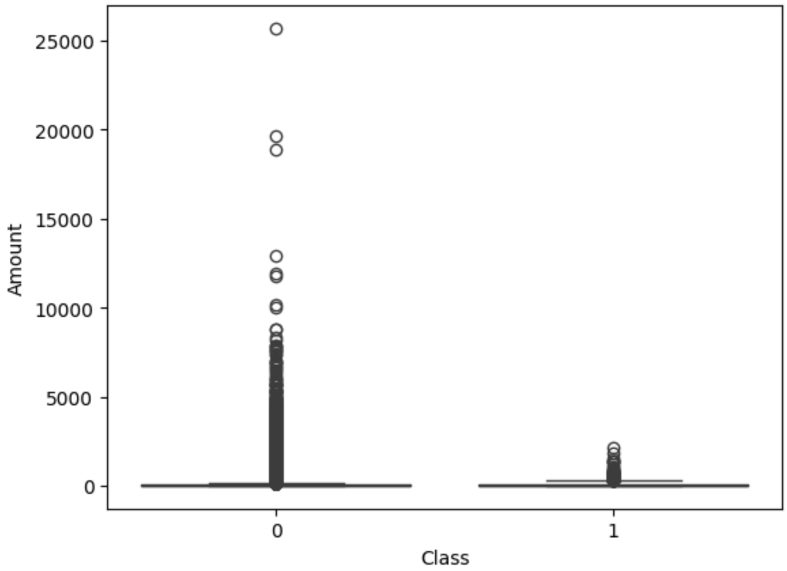
```
count     492.000000
mean      122.211321
std       256.683288
min         0.000000
25%         1.000000
50%         9.250000
75%       105.890000
max      2125.870000
Name: Amount, dtype: float64
```

```
df_normal.Amount.describe()
```

```
count    284314.000000
mean         88.290570
std         250.105416
min           0.000000
25%           5.650000
50%          22.000000
75%          77.050000
max       25691.160000
Name: Amount, dtype: float64
```

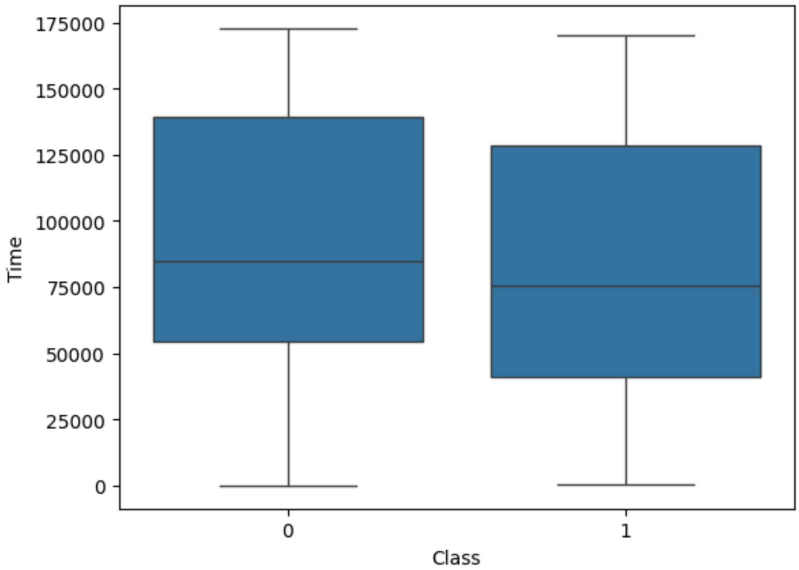
```
sns.boxplot(x='Class',y='Amount',data=df)
```

```
<Axes: xlabel='Class', ylabel='Amount'>
```

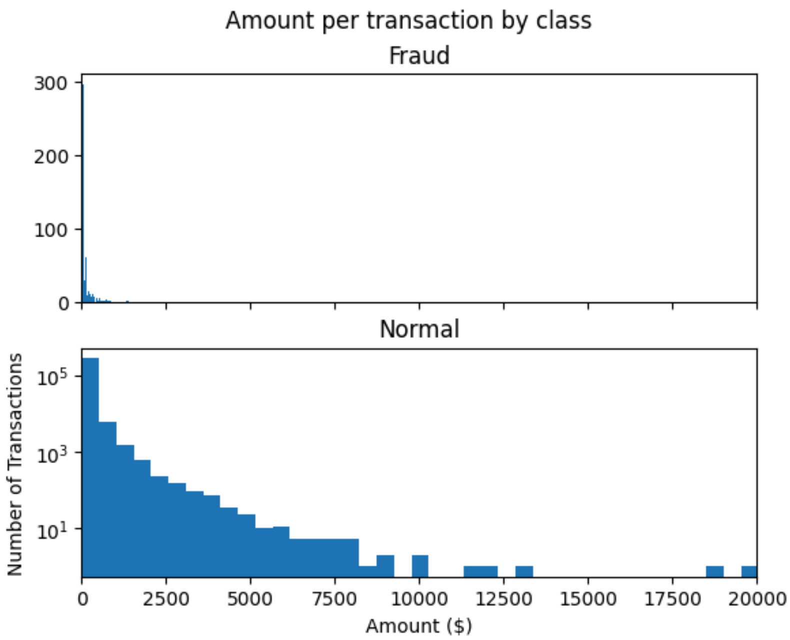



```
sns.boxplot(x='Class',y='Time',data=df)
```

```
<Axes: xlabel='Class', ylabel='Time'>
```




```
f, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
f.suptitle('Amount per transaction by class')
bins = 50
ax1.hist(df_fraud.Amount, bins = bins)
ax1.set_title('Fraud')
ax2.hist(df_normal.Amount, bins = bins)
ax2.set_title('Normal')
plt.xlabel('Amount ($)')
plt.ylabel('Number of Transactions')
plt.xlim((0, 20000))
plt.yscale('log')
plt.show();
```




```
df.shape
```

```
(284806, 31)
```

```
df_Fraud = df[df['Class']==1]

df_Valid = df[df['Class']==0]

outlier_fraction = len(df_Fraud)/float(len(df_Valid))
```

```
print(outlier_fraction)

print("Fraud Cases : {}".format(len(df_Fraud)))

print("Valid Cases : {}".format(len(df_Valid)))
```

```
0.0017304810878113635
Fraud Cases : 492
Valid Cases : 284314
```

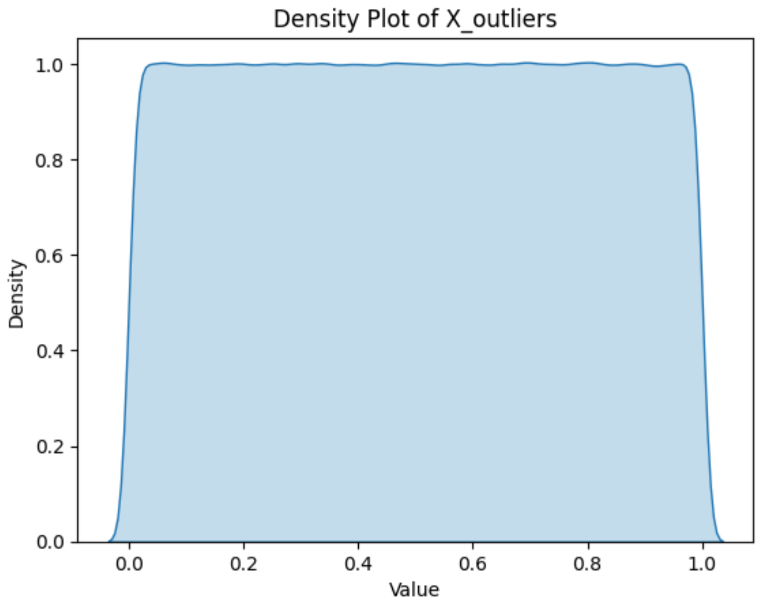
```
df.shape
```

```
(284806, 31)
```

```
#Create independent and Dependent Features
columns = df.columns.tolist()
# Filter the columns to remove data we do not want
columns = [c for c in columns if c not in ["Class"]]
# Store the variable we are predicting
target = "Class"
# Define a random state
state = np.random.RandomState(42)
X = df[columns]
Y = df[target]
X_outliers = state.uniform(low=0, high=1, size=(X.shape[0], X.shape[1]))
# Print the shapes of X & Y
print(X.shape)
print(Y.shape)
```

```
(284806, 30)
(284806,)
```

```
# Assuming X_outliers is flattened or a single column from a DataFrame
sns.kdeplot(X_outliers.flatten(), fill=True)
plt.title('Density Plot of X_outliers')
plt.xlabel('Value')
plt.ylabel('Density')
plt.show()
```




```
from sklearn.metrics import classification_report,accuracy_score
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor
from sklearn.svm import OneClassSVM
from sklearn.cluster import KMeans
from pylab import rcParams
rcParams['figure.figsize'] = 14, 8
RANDOM_SEED = 42
LABELS = ["Normal", "Fraud"]
```

```
##Define the outlier detection methods
classifiers = {
    "Isolation Forest": IsolationForest(n_estimators=100, max_samples=len(X),
                                        contamination=outlier_fraction, random_state=state, verbose=0),
    "Local Outlier Factor": LocalOutlierFactor(n_neighbors=20, algorithm='auto',
                                               leaf_size=30, metric='minkowski',
                                               p=2, metric_params=None, contamination=outlier_fraction),
    "Support Vector Machine": OneClassSVM(kernel='rbf', degree=3, gamma=0.1, nu=0.05,
                                          max_iter=-1),
    }
```

```
type(classifiers)
```

```
dict
```

Creating a smaller dataset from Original dataset in order to run complex ML models

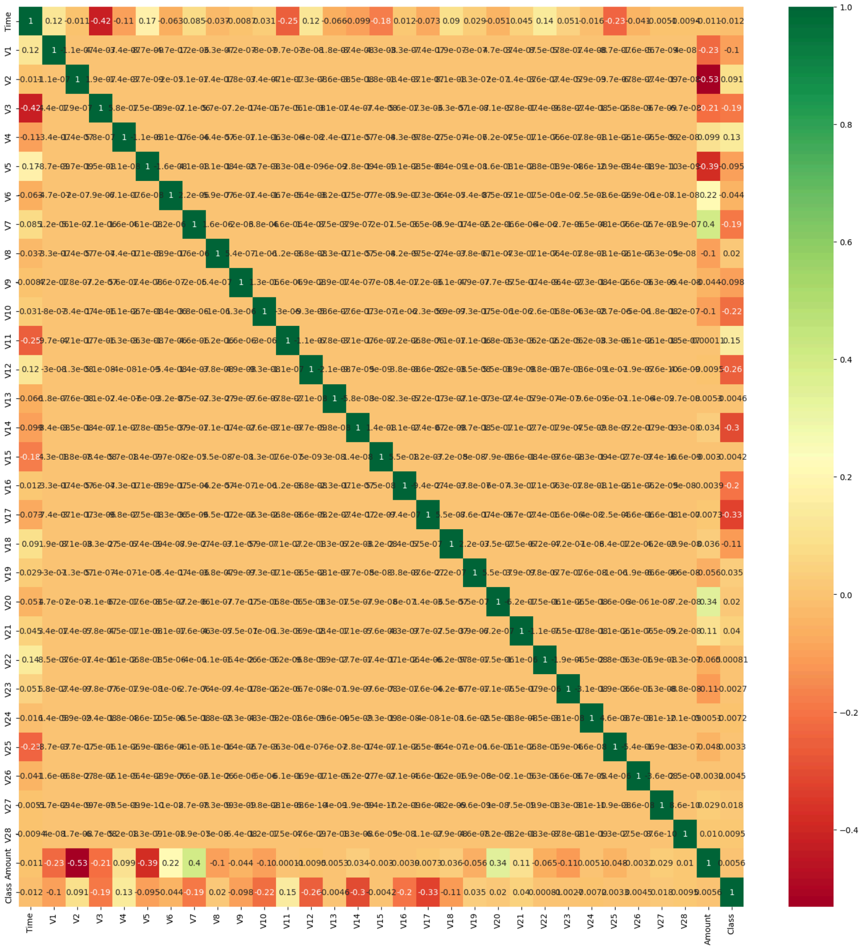
```
df_fraction = df.sample(frac = 0.1,random_state=1)

df_fraction.shape
```

```
(28481, 31)
```

```
df_fraction_Fraud = df_fraction[df_fraction['Class']==1]

df_fraction_Valid = df_fraction[df_fraction['Class']==0]

outlier_fraction = len(df_fraction_Fraud)/float(len(df_fraction_Valid))
print(outlier_fraction)

print("Fraud Cases : {}".format(len(df_fraction_Fraud)))

print("Valid Cases : {}".format(len(df_fraction_Valid)))
```

```
0.0016529506928325245
Fraud Cases : 47
Valid Cases : 28434
```

```
## Correlation
import seaborn as sns
#get correlations of each features in dataset
corrmat = df_fraction.corr()
top_corr_features = corrmat.index
plt.figure(figsize=(20,20))
#plot heat map
g=sns.heatmap(df[top_corr_features].corr(),annot=True,cmap="RdYlGn")
```



```
#Create independent and Dependent Features
columns = df_fraction.columns.tolist()
# Filter the columns to remove data we do not want
columns = [c for c in columns if c not in ["Class"]]
# Store the variable we are predicting
target = "Class"
# Define a random state
state = np.random.RandomState(42)
X = df_fraction[columns]
Y = df_fraction[target]
X_outliers = state.uniform(low=0, high=1, size=(X.shape[0], X.shape[1]))
# Print the shapes of X & Y
print(X.shape)
print(Y.shape)
```

```
(28481, 30)
(28481,)
```

```
n_outliers = len(df_fraction_Fraud)
for i, (clf_name,clf) in enumerate(classifiers.items()):
    #Fit the data and tag outliers
    if clf_name == "Local Outlier Factor":
        y_pred = clf.fit_predict(X)
        scores_prediction = clf.negative_outlier_factor_
    elif clf_name == "Support Vector Machine":
        clf.fit(X)
        y_pred = clf.predict(X)
    else:
        clf.fit(X)
        scores_prediction = clf.decision_function(X)
        y_pred = clf.predict(X)
    #Reshape the prediction values to 0 for Valid transactions , 1 for Fraud transactions
    y_pred[y_pred == 1] = 0
    y_pred[y_pred == -1] = 1
    n_errors = (y_pred != Y).sum()
    # Run Classification Metrics
    print("{}: {}".format(clf_name,n_errors))
    print("Accuracy Score :")
    print(accuracy_score(Y,y_pred))
    print("Classification Report :")
    print(classification_report(Y,y_pred))
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/ensemble/_iforest.py:307: UserWarning: max_samples (284806) is greater than the t
  warn(
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:439: UserWarning: X does not have valid feature names, but IsolationFores
  warnings.warn(
Isolation Forest: 73
Accuracy Score :
0.9974368877497279
Classification Report :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     28434
           1       0.24      0.26      0.25        47

    accuracy                           1.00     28481
   macro avg       0.62      0.63      0.62     28481
weighted avg       1.00      1.00      1.00     28481

Local Outlier Factor: 95
Accuracy Score :
0.9966644429619747
Classification Report :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     28434
           1       0.02      0.02      0.02        47

    accuracy                           1.00     28481
   macro avg       0.51      0.51      0.51     28481
weighted avg       1.00      1.00      1.00     28481

Support Vector Machine: 8411
Accuracy Score :
0.7046803131912504
Classification Report :
              precision    recall  f1-score   support

           0       1.00      0.71      0.83     28434
           1       0.00      0.34      0.00        47

    accuracy                           0.70     28481
   macro avg       0.50      0.52      0.42     28481
```

```
weighted avg       1.00      0.70      0.83     28481
```

```
print(df_fraction.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')
```

Extracting Most Imp Features for the Analysis

```
from sklearn.ensemble import RandomForestClassifier
import pandas as pd

# Assuming df_fraction is your DataFrame and it's already preprocessed
X = df_fraction.drop('Class', axis=1)  # Features
Y = df_fraction['Class']  # Target

# Initialize the Random Forest classifier
rf = RandomForestClassifier(n_estimators=100, random_state=42)

# Fit the model
rf.fit(X, Y)

# Get feature importances
feature_importances = pd.Series(rf.feature_importances_, index=X.columns)

# Focus on 'V' columns and sort them
v_feature_importances = feature_importances.filter(like='V').sort_values(ascending=False)

# Print the sorted 'V' feature importances
print(v_feature_importances)
```

```
V17    0.143233
V14    0.137609
V12    0.129234
V10    0.077304
V16    0.065622
V11    0.061952
V18    0.031312
V20    0.025332
V9     0.024647
V7     0.023815
V26    0.020197
V1     0.018613
V6     0.018440
V4     0.017554
V3     0.016858
V28    0.016809
V21    0.016571
V5     0.015225
V8     0.015187
V2     0.015063
V23    0.014470
V19    0.013539
V15    0.010991
V27    0.010966
V13    0.010445
V25    0.009440
V24    0.008410
V22    0.008326
dtype: float64
```

Applying iForest (Isolation Forest Algorithm) for anomaly detection.

```
from sklearn.ensemble import IsolationForest
from sklearn.metrics import classification_report, accuracy_score
from sklearn.model_selection import train_test_split
```

```
from sklearn.model_selection import train_test_split

# Step 2: Prepare your dataset with only the most important features
important_features = ['V17', 'V14', 'V12', 'V10', 'V16']
X_important = X[important_features]
Y = df_fraction['Class']  # Assuming Y is your target variable and already defined

# Split the data into training and testing sets for evaluation purposes
X_train, X_test, Y_train, Y_test = train_test_split(X_important, Y, test_size=0.2, random_state=42)

# Step 3: Apply the Isolation Forest Model
isolation_forest = IsolationForest(n_estimators=100, contamination=outlier_fraction, random_state=42)
isolation_forest.fit(X_train)

# Predict anomalies on the test set
predictions = isolation_forest.predict(X_test)
# Convert predictions to match your target variable encoding if necessary
predictions = np.where(predictions == 1, 0, 1)  # 0 for normal, 1 for anomaly

# Step 4: Evaluate the Model
print("Accuracy Score:")
print(accuracy_score(Y_test, predictions))
print("Classification Report:")
print(classification_report(Y_test, predictions))
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:439: UserWarning: X does not have valid feature names, but IsolationFores
  warnings.warn(
Accuracy Score:
0.9982446901878181
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      5691
           1       0.30      0.50      0.37         6

    accuracy                           1.00      5697
   macro avg       0.65      0.75      0.69      5697
weighted avg       1.00      1.00      1.00      5697
```

```
important_features = ['V17', 'V14', 'V12', 'V10', 'V16', 'Amount', 'Time']
X_important = X[important_features]

# Proceed with the same steps for splitting the data, applying the model, and evaluating it
X_train, X_test, Y_train, Y_test = train_test_split(X_important, Y, test_size=0.2, random_state=42)

isolation_forest = IsolationForest(n_estimators=100, contamination=outlier_fraction, random_state=42)
isolation_forest.fit(X_train)

# Predict and evaluate as before
predictions = isolation_forest.predict(X_test)
predictions = np.where(predictions == 1, 0, 1)  # Adjusting predictions to match target encoding

print("Accuracy Score:")
print(accuracy_score(Y_test, predictions))
print("Classification Report:")
print(classification_report(Y_test, predictions))
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:439: UserWarning: X does not have valid feature names, but IsolationFores
  warnings.warn(
Accuracy Score:
0.9984202211690363
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      5691
           1       0.33      0.50      0.40         6

    accuracy                           1.00      5697
   macro avg       0.67      0.75      0.70      5697
weighted avg       1.00      1.00      1.00      5697
```

Applying K-Means to identify Anomlies

```
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import numpy as np
from scipy.spatial.distance import cdist

# Including "Amount" and "Time" with the important "V" features
features = ['V17', 'V14', 'V12', 'V10', 'V16', 'Amount', 'Time']
X_important = df_fraction[features]

# It's often useful to scale the features, especially for algorithms like K-Means that are sensitive to distances
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_important)

# Apply K-Means clustering
n_clusters = 5  # Example cluster number, adjust based on domain knowledge or experimentation
kmeans = KMeans(n_clusters=n_clusters, random_state=42).fit(X_scaled)

# Calculate the distance from each point to its nearest cluster center
distances = cdist(X_scaled, kmeans.cluster_centers_, 'euclidean')
min_distances = np.min(distances, axis=1)

# Identify outliers as the points with the top 5% of distances from the nearest cluster center
threshold_distance = np.quantile(min_distances, 0.95)
outliers = min_distances > threshold_distance

# You can now use the 'outliers' boolean array for further analysis or inspection
print(f"Number of identified outliers: {np.sum(outliers)}")
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
Number of identified outliers: 1424
```

Applying Auto-Encoders (Deep learning) for Anomaly detection.

```
import numpy as np
import tensorflow as tf
```

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.callbacks import ModelCheckpoint
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

```
# Assuming X is your dataset
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the dataset into training and testing sets
X_train, X_test = train_test_split(X_scaled, test_size=0.2, random_state=42)
```

```
input_dim = X_train.shape[1]  # Number of features

# Define the input layer
input_layer = Input(shape=(input_dim,))

# Define encoding and decoding layers
encoded = Dense(64, activation='relu')(input_layer)
encoded = Dense(32, activation='relu')(encoded)
decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(input_dim, activation='linear')(decoded)

# Build the autoencoder model
autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer='adam', loss='mean_squared_error')

# Print the autoencoder structure
autoencoder.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 30)]              0

 dense (Dense)               (None, 64)                1984

 dense_1 (Dense)             (None, 32)                2080

 dense_2 (Dense)             (None, 64)                2112

 dense_3 (Dense)             (None, 30)                1950

=================================================================
Total params: 8126 (31.74 KB)
Trainable params: 8126 (31.74 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
autoencoder.fit(X_train, X_train,
                epochs=100,
                batch_size=256,
                shuffle=True,
                validation_data=(X_test, X_test),
                verbose=1)
```

```
Epoch 1/100
89/89 [==============================] - 3s 5ms/step - loss: 0.8667 - val_loss: 0.6116
Epoch 2/100
89/89 [==============================] - 0s 3ms/step - loss: 0.4812 - val_loss: 0.3343
Epoch 3/100
89/89 [==============================] - 0s 3ms/step - loss: 0.2629 - val_loss: 0.1865
Epoch 4/100
89/89 [==============================] - 0s 3ms/step - loss: 0.1483 - val_loss: 0.1047
Epoch 5/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0877 - val_loss: 0.0678
Epoch 6/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0617 - val_loss: 0.0522
Epoch 7/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0481 - val_loss: 0.0391
Epoch 8/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0376 - val_loss: 0.0303
Epoch 9/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0310 - val_loss: 0.0267
Epoch 10/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0262 - val_loss: 0.0219
Epoch 11/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0219 - val_loss: 0.0215
Epoch 12/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0199 - val_loss: 0.0159
Epoch 13/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0172 - val_loss: 0.0136
Epoch 14/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0143 - val_loss: 0.0125
Epoch 15/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0127 - val_loss: 0.0108
Epoch 16/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0112 - val_loss: 0.0084
Epoch 17/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0098 - val_loss: 0.0083
Epoch 18/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0094 - val_loss: 0.0093
Epoch 19/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0079 - val_loss: 0.0072
```

```
Epoch 20/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0072 - val_loss: 0.0059
Epoch 21/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0062 - val_loss: 0.0046
Epoch 22/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0061 - val_loss: 0.0040
Epoch 23/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0055 - val_loss: 0.0041
Epoch 24/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0051 - val_loss: 0.0032
Epoch 25/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0052 - val_loss: 0.0041
Epoch 26/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0044 - val_loss: 0.0029
Epoch 27/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0042 - val_loss: 0.0034
Epoch 28/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0041 - val_loss: 0.0024
Epoch 29/100
89/89 [==============================] - 0s 3ms/step - loss: 0.0043 - val_loss: 0.0033
```

```
# Use the autoencoder to reconstruct the test set
X_test_pred = autoencoder.predict(X_test)

# Calculate the reconstruction error as the mean squared error
reconstruction_error = np.mean(np.power(X_test - X_test_pred, 2), axis=1)

# Define a threshold for anomaly detection
threshold = np.quantile(reconstruction_error, 0.95)  # Adjust based on your needs

# Anything above the threshold is considered an anomaly
anomalies = reconstruction_error > threshold

print(f"Detected {np.sum(anomalies)} anomalies out of {len(X_test)} samples in the test set.")
```

```
179/179 [==============================] - 0s 1ms/step
Detected 285 anomalies out of 5697 samples in the test set.
```

```
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

# Assuming `reconstruction_error` contains the reconstruction errors of your test samples
# And assuming `Y_test` contains your true labels (0 for normal, 1 for anomaly)

# Determine the threshold (for demonstration, using the 95th percentile)
threshold = np.quantile(reconstruction_error, 0.95)

# Classify as anomalies (1) those above the threshold, normal (0) below
predicted_labels = np.where(reconstruction_error > threshold, 1, 0)

# Calculate the metrics
precision, recall, f1_score, _ = precision_recall_fscore_support(Y_test, predicted_labels, average='binary')
accuracy = accuracy_score(Y_test, predicted_labels)

print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1-Score: {f1_score}")
print(f"Accuracy: {accuracy}")
```