Project Title: Credit Card Fraud Detection

Objective: Develop a machine learning model to identify and prevent fraudulent credit card transactions.

Steps:

1. **Data Collection**:

   - Gather a comprehensive dataset of credit card transactions. This dataset should ideally contain both legitimate and fraudulent transactions.

   - Ensure data privacy and compliance with regulations like GDPR when collecting and handling sensitive financial data.

2. **Data Preprocessing**:

   - Clean the data by addressing missing values, duplicates, and outliers.

   - Explore the dataset to understand its structure and characteristics.

   - Encode categorical features and standardize numerical features for modeling.

3. **Data Splitting**:

   - Split the dataset into training, validation, and test sets. Typically, an 80-10-10 or 70-15-15 split is used.

4. **Feature Engineering**:

   - Create relevant features that capture transaction behavior, such as transaction frequency, transaction amounts, time of day, and more.

   - Consider feature scaling and transformation techniques to improve model performance.

5. **Model Selection**:

   - Choose appropriate machine learning algorithms for fraud detection. Common choices include logistic regression, decision trees, random forests, support vector machines, and neural networks.

   - Experiment with multiple models to find the one that performs best on the validation set.

6. **Model Training**:

   - Train the selected models on the training data.

- Tune hyperparameters to optimize model performance using techniques like grid search or random search.

7. **Model Evaluation**:

- Assess the model's performance using various metrics, including accuracy, precision, recall, F1-score, and ROC AUC.

- Evaluate the model's ability to detect fraud while minimizing false positives.

8. **Real-time Monitoring**:

- Implement real-time monitoring of credit card transactions using the trained model. This involves setting up a system that can process and analyze incoming transactions in real-time.

9. **Alerting System**:

- Develop an alerting system that triggers notifications (e.g., emails, SMS alerts) when potentially fraudulent transactions are detected.

10. **Continuous Learning**:

- Implement a feedback loop to continuously update and improve the model. New data and fraud patterns should be incorporated into the model to stay effective against evolving fraud tactics.

11. **Documentation**:

- Maintain thorough documentation of data sources, preprocessing steps, model architecture, and performance metrics.

12. **Privacy and Security**:

- Ensure robust data security measures to protect sensitive customer information throughout the project.

13. **Compliance**:

- Ensure that your project complies with legal and regulatory requirements related to data privacy, such as GDPR or local financial regulations.

14. **Scalability**:

   - Design the system to handle a growing volume of transactions as the credit card user base expands.

15. **Deployment**:

   - Deploy the model and monitoring system in a production environment, ensuring high availability and scalability.

16. **User Interface** :

   - Create a user interface for administrators to visualize and manage detected fraud cases.

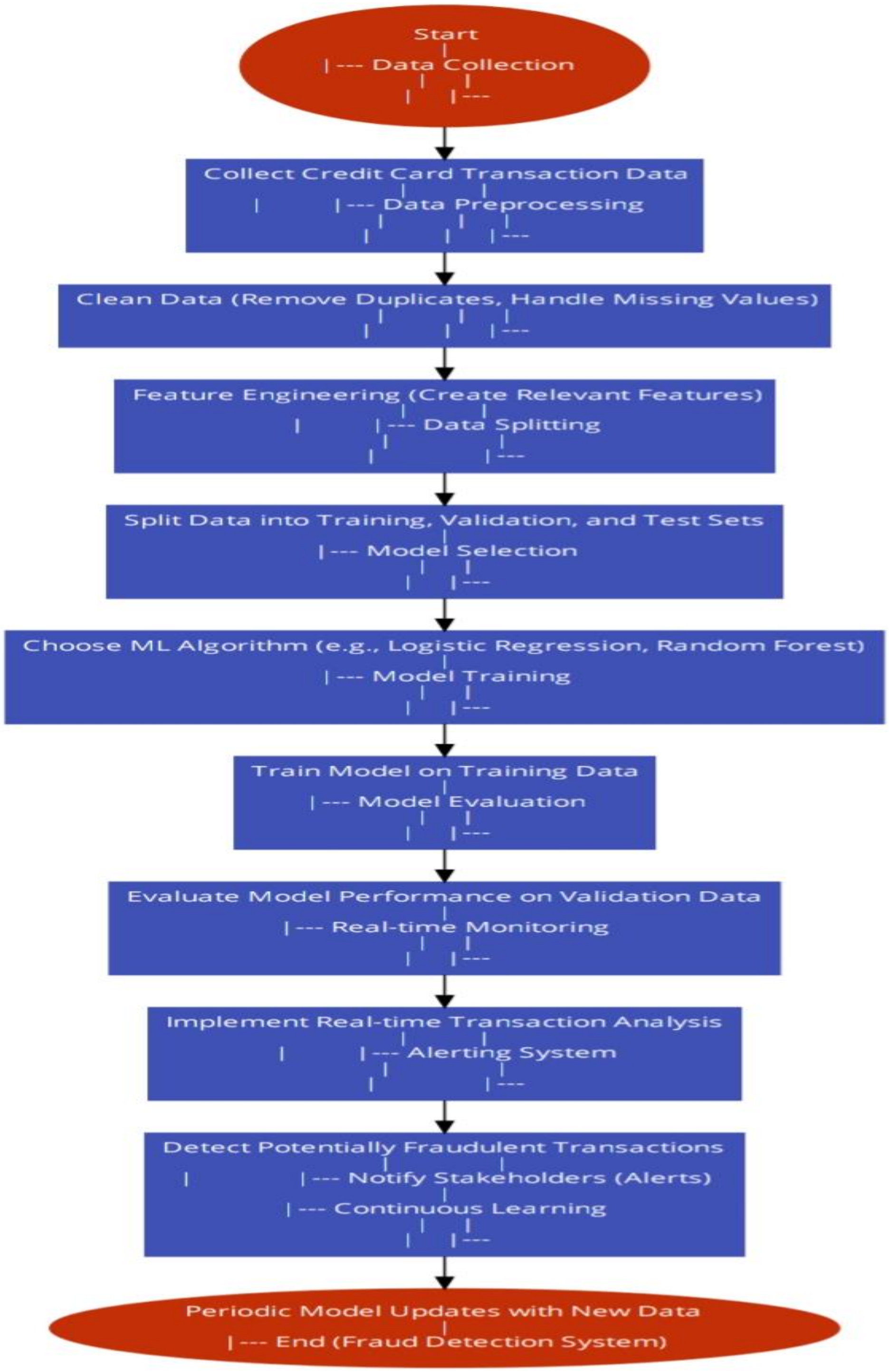
17. **Reporting**:

   - Generate regular reports summarizing the system's performance and highlighting any trends or patterns in detected fraud.

Keep the project concise, focusing on the core steps of data collection, preprocessing, modeling, and real-time monitoring to achieve effective credit card fraud detection.

Flow chart

Objective



The dataset contains a very minute percentage transacions, which are fraudulent. We need to find out those transactions which belong to the Fraud Class

Based on the data we have to generate a set of insights and recommendations that will help the credit card company from preventing the customers to be charged falsly!

In [2]:
```
import pandas as pd
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

import sys
import scipy
import sklearn

import warnings
warnings.filterwarnings('ignore') # To supress warnings
sns.set(style="whitegrid") # set the background for the graphs
```

In [3]:
```
#importing data from kaggle
data = pd.read_csv("/kaggle/input/creditcardfraud/creditcard.csv")
data.head(5)
```

Out[3]:

	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	0.0986
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.0851
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.2476
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.3774
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270

5 rows × 31 columns

So, we do not know what actually does V1,V2....V28 means due to data confidentiality, but what we know is they're going to help us draw insights from the data.

In [4]:

```python
print(data.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')
```

In [5]:

```python
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   Time    284807 non-null  float64
 1   V1      284807 non-null  float64
 2   V2      284807 non-null  float64
 3   V3      284807 non-null  float64
 4   V4      284807 non-null  float64
 5   V5      284807 non-null  float64
 6   V6      284807 non-null  float64
 7   V7      284807 non-null  float64
 8   V8      284807 non-null  float64
 9   V9      284807 non-null  float64
 10  V10     284807 non-null  float64
 11  V11     284807 non-null  float64
 12  V12     284807 non-null  float64
 13  V13     284807 non-null  float64
 14  V14     284807 non-null  float64
 15  V15     284807 non-null  float64
 16  V16     284807 non-null  float64
 17  V17     284807 non-null  float64
 18  V18     284807 non-null  float64
 19  V19     284807 non-null  float64
 20  V20     284807 non-null  float64
 21  V21     284807 non-null  float64
 22  V22     284807 non-null  float64
 23  V23     284807 non-null  float64
 24  V24     284807 non-null  float64
 25  V25     284807 non-null  float64
 26  V26     284807 non-null  float64
 27  V27     284807 non-null  float64
 28  V28     284807 non-null  float64
 29  Amount  284807 non-null  float64
 30  Class   284807 non-null  int64
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
```

The dataset does not contain any object data type, so we do not have to spend any time on conversion.
Lets see if our data contains any null values!

In [6]:

```
data.isnull().sum()
```

Out[6]:

```
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
```

In [7]:

```
data.describe()
```

Out[7]:

	Time	V1	V2	V3	V4	V5
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e
mean	94813.859575	3.918649e-15	5.682686e-16	-8.761736e-15	2.811118e-15	-1.552103
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e

8 rows × 31 columns

In [8]:

```
data.shape
```

Out[8]:

```
(284807, 31)
```

In [9]:

```
# random_state helps assure that you always get the same output when you split the data
# this helps create reproducible results and it does not actually matter what the number is
# frac is percentage of the data that will be returned
data = data.sample(frac = 0.2, random_state = 1)
print(data.shape)
```

```
(56961, 31)
```

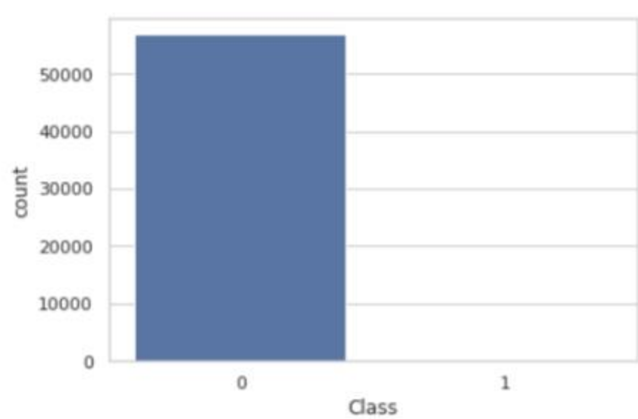
Now let us conduct some exploratory data analysis on our data!

In [10]:

```
# Visualize the count of survivors
sns.countplot('Class', data=data)
```

Out[10]:

```
<AxesSubplot:xlabel='Class', ylabel='count'>
```



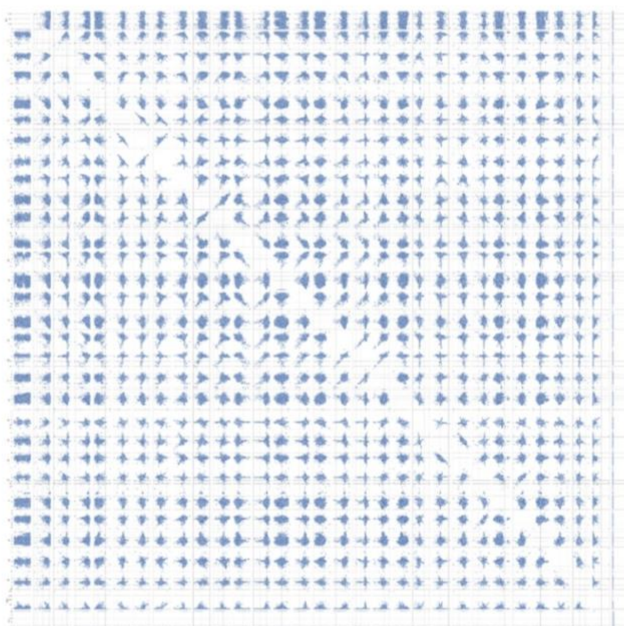
The count of fraudulent transactions as compared to the non fraudulent one’s is almost null. It makes it so difficult for us to classify the test data.

Remember, Rule 1 of the dataset is that the predicted value should be somewhat equally divided between the two classes!

Anyway, lets see how well we are able to perform!

In [11]:
```
sns.pairplot(data)
```

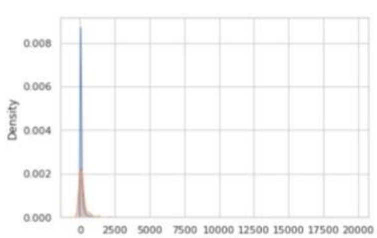
Out[11]:
```
<seaborn.axisgrid.PairGrid at 0x7f7434cd9710>
```



In [12]:
```
print("Fraud to NonFraud Ratio of {:.3f}%".format(492/284315*100))
```

```
Fraud to NonFraud Ratio of 0.173%
```

In [13]:
```
sns.kdeplot(data.Amount[data.Class == 0], label = 'Fraud', shade=True)
sns.kdeplot(data.Amount[data.Class == 1], label = 'NonFraud', shade=True)
plt.xlabel('Amount');
```



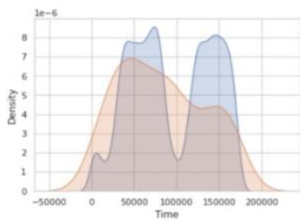
Looks like there a lot more instances of small fraud amounts than really large ones.

In [14]:

```
sns.kdeplot(data.Time[data.Class == 0], label = 'Fraud', shade=True)
sns.kdeplot(data.Time[data.Class == 1], label = 'NonFraud', shade=True)
plt.xlabel('Time')
```

Out[14]:

```
Text(0.5, 0, 'Time')
```

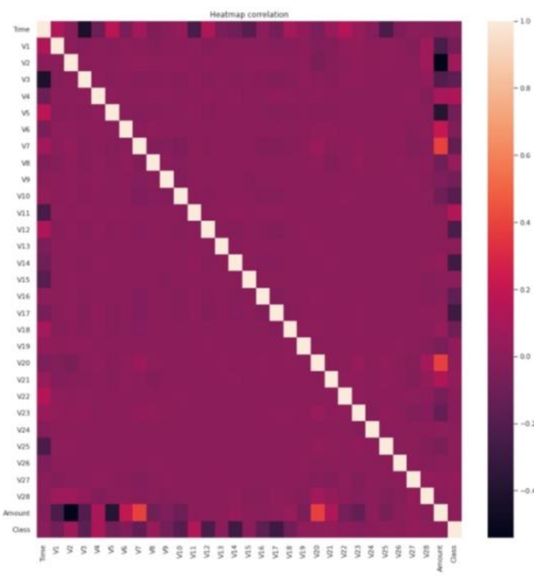


We notice that the feature time doesn't seem to have an impact in the frequency of frauds.

In [15]:

```
plt.figure(figsize=(15,15))
sns.heatmap(data.corr()) # Displaying the Heatmap

plt.title('Heatmap correlation')
plt.show()
```



As we can notice, most of the features are not correlated with each other.

What can generally be done on a massive dataset is a dimension reduction. By picking the most important dimensions, there is a possiblity of explaining most of the problem, thus gaining a considerable amount of time while preventing the accuracy to drop too much.

In [16]:

```
# get the columns from the dataframe
columns = data.columns.tolist()

# filter the columns to remove the data we do not want
columns = [c for c in columns if c not in ['Class']]

# store the variable we will be predicting on which is class
target = 'Class'

# X includes everything except our class column
X = data[columns]
# Y includes all the class labels for each sample
# this is also one-dimensional
Y = data[target]

# print the shapes of X and Y
print(X.shape)
print(Y.shape)
```

```
(56961, 30)
(56961,)
```

In [17]:

```
from sklearn.metrics import classification_report, accuracy_score
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor
```

In [18]:

```
# determine the number of fraud cases
fraud = data[data['Class'] == 1]
valid = data[data['Class'] == 0]

outlier_fraction = len(fraud) / float(len(valid))
print(outlier_fraction)

print('Fraud Cases: {}'.format(len(fraud)))
print('Valid Cases: {}'.format(len(valid)))
```

```
0.0015296972254457222
Fraud Cases: 87
Valid Cases: 56874
```

In [19]:

```
state = 1

# define the outlier detection methods
classifiers = {
    # contamination is the number of outliers we think there are
    'Isolation Forest': IsolationForest(max_samples = len(X),
                                        contamination = outlier_fraction,
                                        random_state = state),
    # number of neighbors to consider, the higher the percentage of outliers the h
igher you want to make this number
    'Local Outlier Factor': LocalOutlierFactor(
    n_neighbors = 20,
    contamination = outlier_fraction)
}
```

In [19]:

```
state = 1

# define the outlier detection methods
classifiers = {
    # contamination is the number of outliers we think there are
    'Isolation Forest': IsolationForest(max_samples = len(X),
                                        contamination = outlier_fraction,
                                        random_state = state),
    # number of neighbors to consider, the higher the percentage of outliers the h
igher you want to make this number
    'Local Outlier Factor': LocalOutlierFactor(
    n_neighbors = 20,
    contamination = outlier_fraction)
}
```

In [20]:

```
n_outliers = len(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_pred = clf.negative_outlier_factor_
    else:
        clf.fit(X)
        scores_pred = clf.decision_function(X)
        y_pred = clf.predict(X)



# reshape the prediction values to 0 for valid and 1 for fraud
    y_pred[y_pred == 1] = 0
    y_pred[y_pred == -1] = 1

    # calculate the number of errors
    n_errors = (y_pred != Y).sum()

    # classification matrix
    print('{}: {}'.format(clf_name, n_errors))
    print(accuracy_score(Y, y_pred))
    print(classification_report(Y, y_pred))
```

```
Isolation Forest: 127
0.997770404311722
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56874
           1       0.27      0.28      0.27        87

    accuracy                           1.00     56961
   macro avg       0.64      0.64      0.64     56961
weighted avg       1.00      1.00      1.00     56961

Local Outlier Factor: 173
0.9969628342199048
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56874
           1       0.01      0.01      0.01        87

    accuracy                           1.00     56961
   macro avg       0.50      0.50      0.50     56961
weighted avg       1.00      1.00      1.00     56961
```