Building Model For Fraud Card Detection:

In this project, we're on a mission to develop a state-of-the-art fraudulent card detection model. To achieve this goal, we've harnessed the power of diverse machine learning algorithms, including Logistic Regression, Support Vector Machines (SVM), and Random Forest. Our primary focus is to meticulously evaluate the performance of these algorithms using key metrics such as Mean Absolute Error (MAE) and F1 Score... By doing so, we aim to pinpoint the algorithm that best balances precision and recall, providing a robust defense against fraudulent transactions in today's digital financial landscape.

Import the necessary libraries:

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Upload dataset:

In [2]:
```
df = pd.read_csv(r"C:\Users\PC\Desktop\Project\Fraud Card Detection\creditcard.csv")
```

Explore data Analysis:

In [3]:
```
df.head()
```

Out[3]:

	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 × 31 columns

In [4]:
```
df.isnull().sum()
```

Out[4]:
```
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 [5]:
```
df.drop_duplicates()
```

Out[5]:

	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
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.305334	1.914428	...	0.213454	0.111864	1.014480	-0.509348	1.436807	0.250034	0.943651	0.823731	0.77	0
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.294869	0.584800	...	0.214205	0.924384	0.012463	-1.016226	-0.606624	-0.395255	0.068472	-0.053527	24.79	0
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.708417	0.432454	...	0.232045	0.578229	-0.037501	0.640134	0.265745	-0.087371	0.004455	-0.026561	67.88	0
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.679145	0.392087	...	0.265245	0.800049	-0.163298	0.123205	-0.569159	0.546668	0.108821	0.104533	10.00	0
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.414650	0.486180	...	0.261057	0.643078	0.376777	0.008797	-0.473649	-0.818267	-0.002415	0.013649	217.00	0

283726 rows × 31 columns

In [6]:
```
df.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
```

Checking for Distribution of legitimate transactions and fraudulent transactions: 0 for Normal transaction and 1 for fraud transaction:

In [7]:
```
class_counts = df['Class'].value_counts()
print(class_counts)
```

```
0    284315
1       492
Name: Class, dtype: int64
```

In [8]:
```
legit = df[df['Class'] == 0]
fraud = df[df['Class'] == 1]
```

In [9]:
```
print(legit.shape)
print(fraud.shape)
```

```
(284315, 31)
(492, 31)
```

Statistical measures of the model for legit and fraud transaction :

In [10]:
```
legit.Amount.describe()
```

Out[10]:
```
count    284315.000000
mean         88.291022
std         250.105092
min           0.000000
25%           5.650000
50%          22.000000
75%          77.050000
max       25691.160000
Name: Amount, dtype: float64
```

In [11]:
```
fraud.Amount.describe()
```

Out[11]:
```
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
```

Compare the value for both transaction :

In [12]:
```
df.groupby('Class').mean()
```

Out[12]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V20	V21	V22	V23	V24	V25	V26	V27	V28	Amou
Class																					
0	94838.202258	0.008258	-0.006271	0.012171	-0.007860	0.005453	0.002419	0.009637	-0.000987	0.004467	...	-0.000644	-0.001235	-0.000024	0.000070	0.000182	-0.000072	-0.000089	-0.000295	-0.000131	88.29102
1	80746.806911	-4.771948	3.623778	-7.033281	4.542029	-3.151225	-1.397737	-5.568731	0.570636	-2.581123	...	0.372319	0.713588	0.014049	-0.040308	-0.105130	0.041449	0.051648	0.170575	0.075667	122.21132

2 rows × 30 columns

Under sampling : Build new dataset containing similair distrubition of legit and fraud transaction:

In [13]:
```
legit_sample = legit.sample(n=492)
```

Concatenating of two dataframes:

In [14]:
```
new_dataset = pd.concat([legit_sample, fraud], axis=0)
```

Information about our new dataset:

In [15]:
```
new_dataset.describe()
```

Out[15]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	
count	984.000000	984.000000	984.000000	984.000000	984.000000	984.000000	984.000000	984.000000	984.000000	984.000000	...	984.000000	984.000000	984.000000	984.000000	984.000000	984.000000	984.000000	984.0
mean	88690.326220	-2.306440	1.804776	-3.467425	2.305764	-1.571569	-0.707137	-2.742143	0.296050	-1.296652	...	0.326836	-0.025625	-0.006222	-0.040320	0.024920	0.008302	0.079372	0.0
std	48152.139823	5.511233	3.645100	6.231239	3.168679	4.200583	1.719824	5.872149	4.842653	2.305448	...	2.780438	1.156389	1.143957	0.564019	0.658102	0.487175	1.021500	0.4
min	406.000000	-30.552380	-9.316517	-31.103685	-2.919435	-22.105532	-6.406267	-43.557242	-41.044261	-13.434066	...	-22.797604	-8.887017	-19.254328	-2.028024	-4.781606	-1.196621	-7.263482	-1.8
25%	46788.250000	-2.681068	-0.138871	-5.084967	-0.037838	-1.803200	-1.559879	-3.031843	-0.190608	-2.294075	...	-0.181608	-0.584562	-0.222721	-0.391971	-0.327491	-0.331073	-0.062199	-0.0
50%	82181.500000	-0.753514	0.964306	-1.197681	1.379664	-0.424836	-0.663836	-0.560046	0.149443	-0.715968	...	0.125216	-0.014205	-0.016786	0.011384	0.044969	-0.026702	0.050467	0.0
75%	135055.250000	1.042053	2.765175	0.346338	4.237855	0.478568	0.072293	0.274802	0.834851	0.198952	...	0.627548	0.542106	0.200264	0.386792	0.390240	0.298988	0.429253	0.2
max	172562.000000	2.358553	22.057729	3.428548	12.114672	11.095089	6.474115	8.300058	20.007208	5.593104	...	27.202839	8.361985	5.466230	1.146084	2.208209	2.745261	3.052358	2.6

8 rows × 31 columns

In [16]:
```
new_dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 984 entries, 284219 to 281674
Data columns (total 31 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Time    984 non-null    float64
 1   V1      984 non-null    float64
 2   V2      984 non-null    float64
 3   V3      984 non-null    float64
 4   V4      984 non-null    float64
 5   V5      984 non-null    float64
 6   V6      984 non-null    float64
 7   V7      984 non-null    float64
 8   V8      984 non-null    float64
 9   V9      984 non-null    float64
 10  V10     984 non-null    float64
 11  V11     984 non-null    float64
 12  V12     984 non-null    float64
 13  V13     984 non-null    float64
 14  V14     984 non-null    float64
 15  V15     984 non-null    float64
 16  V16     984 non-null    float64
 17  V17     984 non-null    float64
 18  V18     984 non-null    float64
 19  V19     984 non-null    float64
 20  V20     984 non-null    float64
 21  V21     984 non-null    float64
 22  V22     984 non-null    float64
 23  V23     984 non-null    float64
 24  V24     984 non-null    float64
 25  V25     984 non-null    float64
 26  V26     984 non-null    float64
 27  V27     984 non-null    float64
 28  V28     984 non-null    float64
 29  Amount  984 non-null    float64
 30  Class   984 non-null    int64  
dtypes: float64(30), int64(1)
memory usage: 246.0 KB
```

Spliting the new data into variables:

In [17]:
```
from sklearn.model_selection import train_test_split
```

In [18]:
```
x = new_dataset.drop(columns='Class', axis=1) # variable dependant
y = new_dataset['Class'] # Variable independant
```

In [19]:
```
print(x)
```

```
            Time        V1        V2        V3        V4        V5        V6  \
284219  172255.0  1.763899 -0.740882 -0.450826  1.036429 -0.306175  0.925229   
9553     14302.0  0.973180 -0.371077  1.089693  0.732777 -0.627498  0.667497   
83492    59882.0 -0.858196 -0.297441  0.624864 -2.585102 -0.523313 -0.563535   
234232  147886.0 -0.886470  0.435063 -0.482884 -1.155523  0.517037 -0.764747   
109269   71301.0  1.431104 -0.439030  0.296155 -0.780462 -0.831697 -0.806454   
...          ...       ...       ...       ...       ...       ...       ...   
279863  169142.0 -1.927883  1.125653 -4.518331  1.749293 -1.566487 -2.010494   
280143  169347.0  1.378559  1.289381 -5.004247  1.411850  0.442581 -1.326536   
280149  169351.0 -0.676143  1.126366 -2.213700  0.468308 -1.120541 -0.003346   
281144  169966.0 -3.113832  0.585864 -5.399730  1.817092 -0.840618 -2.943548   
281674  170348.0  1.991976  0.158476 -2.583441  0.408670  1.151147 -0.096695   

              V7        V8        V9  ...       V20       V21       V22  \
284219 -0.835354  0.343719  1.153559  ... -0.031515  0.102272  0.170025   
9553   -0.745972  0.199534  1.750708  ...  0.083785  0.060950  0.426771   
83492  -0.238861  0.397232 -3.000544  ... -0.292921 -0.400085 -1.261688   
234232  0.659790  0.492545  0.286702  ... -0.115077 -0.010341 -0.226868   
109269 -0.481373 -0.187189 -0.795558  ...  0.099078 -0.165700 -0.619657   
...          ...       ...       ...  ...       ...       ...       ...   
279863 -0.882850  0.697211 -2.064945  ...  1.252967  0.778584 -0.319189   
280143 -1.413170  0.248525 -1.127396  ...  0.226138  0.370612  0.028234   
280149 -2.234739  1.210158 -0.652250  ...  0.247968  0.751826  0.834108   
281144 -2.208002  1.058733 -1.632333  ...  0.306271  0.583276 -0.269209   
281674  0.223050 -0.068384  0.577829  ... -0.017652 -0.164350 -0.295135   

             V23       V24       V25       V26       V27       V28  Amount  
284219  0.006949 -1.502695 -0.262256 -0.669495  0.053287 -0.027273  112.59  
9553   -0.149403 -0.269263  0.240813  0.554656 -0.020598  0.017038   94.00  
83492   0.063138 -0.604560 -0.041993 -0.636147 -0.065565 -0.020187   53.44  
234232  0.109980 -0.639904 -0.535535 -0.910787  0.312123  0.147690   58.98  
109269  0.083704 -0.145826  0.287304 -0.486397  0.004415  0.018139   16.40  
...          ...       ...       ...       ...       ...       ...     ...  
279863  0.639419 -0.294885  0.537503  0.788395  0.292680  0.147968  390.00  
280143 -0.145640 -0.081049  0.521875  0.739467  0.389152  0.186637    0.76  
280149  0.190944  0.032070 -0.739695  0.471111  0.385107  0.194361   77.89  
281144 -0.456108 -0.183659 -0.328168  0.606116  0.884876 -0.253700  245.00  
281674 -0.072173 -0.450261  0.313267 -0.289617  0.002988 -0.015309   42.53  

[984 rows x 30 columns]
```

In [20]:
```
print(y)
```

```
284219    0
9553      0
83492     0
234232    0
109269    0
         ..
279863    1
280143    1
280149    1
281144    1
281674    1
Name: Class, Length: 984, dtype: int64
```

Split the data into training data and test data:

In [21]:
```
x_train, x_test, y_train, y_test = train_test_split(x,y,test_size=0.2, random_state=2)
```

In [22]:
```
print(x.shape, x_train.shape, x_test.shape)
```

```
(984, 30) (787, 30) (197, 30)
```

Train the model With logistic regression:

In [23]:
```
from sklearn.linear_model import LogisticRegression # import the necessary librarie

model = LogisticRegression()
model.fit(x_train, y_train)
```

Out[23]:
```
▾ LogisticRegression
LogisticRegression()
```

In [24]:
```
# Predict on the train Data:
x_train_prediction = model.predict(x_train)
```

In [25]:
```
# Predict on the test data:
x_test_prediction = model.predict(x_test)
```

Train the model With SVM (Support Vector Machine) Classifier :

In [26]:
```
from sklearn.svm import SVC # import the necessary librarie
```

In [27]:
```
# Train the model with SVM:
svm_model = SVC()
svm_model.fit(x_train, y_train)
```

Out[27]:
```
▾ SVC
SVC()
```

In [28]:
```
# Predict on the train data:
svm_x_train_prediction = svm_model.predict(x_train)
```

In [29]:
```
# Predict on the test data:
svm_x_test_prediction = svm_model.predict(x_test)
```

Train the model With Random Forest Classifier:

In [30]:
```
from sklearn.ensemble import RandomForestClassifier  # import the necessary librarie
```

In [31]:
```
# Train the model with Random Forest:
rf_model = RandomForestClassifier()
rf_model.fit(x_train, y_train)
```

Out[31]:
```
▾ RandomForestClassifier
RandomForestClassifier()
```

In [32]:
```
# Predict on the train data:
rf_x_train_prediction = rf_model.predict(x_train)
```

In [33]:
```
# Predict on the test data:
rf_x_test_prediction = rf_model.predict(x_test)
```

Evaluate the performance of the three algorithms with different metrics:

In [34]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix #import the necessary librarie
```

In [35]:
```
# Evaluate Logistic Regression
lr_accuracy = accuracy_score(y_test, x_test_prediction)
lr_precision = precision_score(y_test, x_test_prediction)
lr_recall = recall_score(y_test, x_test_prediction)
lr_f1 = f1_score(y_test, x_test_prediction)
lr_confusion = confusion_matrix(y_test, x_test_prediction)

# Evaluate SVM
svm_accuracy = accuracy_score(y_test, svm_x_test_prediction)
svm_precision = precision_score(y_test, svm_x_test_prediction)
svm_recall = recall_score(y_test, svm_x_test_prediction)
svm_f1 = f1_score(y_test, svm_x_test_prediction)
svm_confusion = confusion_matrix(y_test, svm_x_test_prediction)

# Evaluate Random Forest
rf_accuracy = accuracy_score(y_test, rf_x_test_prediction)
rf_precision = precision_score(y_test, rf_x_test_prediction)
rf_recall = recall_score(y_test, rf_x_test_prediction)
rf_f1 = f1_score(y_test, rf_x_test_prediction)
rf_confusion = confusion_matrix(y_test, rf_x_test_prediction)
```

In [36]:
```
# Print the evaluation metrics
print("Logistic Regression:")
print(f"Accuracy: {lr_accuracy}")
print(f"Precision: {lr_precision}")
print(f"Recall: {lr_recall}")
print(f"F1 Score: {lr_f1}")
print(f"Confusion Matrix:\n{lr_confusion}")

print("\nSVM:")
print(f"Accuracy: {svm_accuracy}")
print(f"Precision: {svm_precision}")
print(f"Recall: {svm_recall}")
print(f"F1 Score: {svm_f1}")
print(f"Confusion Matrix:\n{svm_confusion}")

print("\nRandom Forest:")
print(f"Accuracy: {rf_accuracy}")
print(f"Precision: {rf_precision}")
print(f"Recall: {rf_recall}")
print(f"F1 Score: {rf_f1}")
print(f"Confusion Matrix:\n{rf_confusion}")
```

```
Logistic Regression:
Accuracy: 0.9035532994923858
Precision: 0.9191919191919192
Recall: 0.8921568627450981
F1 Score: 0.9054726368159205
Confusion Matrix:
[[87  8]
 [11 91]]

SVM:
Accuracy: 0.4873096446700508
Precision: 0.5048543689320388
Recall: 0.5098039215686274
F1 Score: 0.5073170731707317
Confusion Matrix:
[[44 51]
 [50 52]]

Random Forest:
Accuracy: 0.9441624365482234
Precision: 0.9789473684210527
Recall: 0.9117647058823529
F1 Score: 0.9441624365482234
Confusion Matrix:
[[93  2]
 [ 9 93]]
```

Result analysis:

Based on the evaluation metrics, it appears that the Random Forest model is performing the best among the three models, with the highest accuracy, precision, recall, and F1 score. Here's a summary of the results:

Logistic Regression:

Accuracy: 0.8985 Precision: 0.91 Recall: 0.8922 F1 Score: 0.9010

SVM:

Accuracy: 0.5736 Precision: 0.6324 Recall: 0.4216 F1 Score: 0.5059

Random Forest:

Accuracy: 0.9239 Precision: 0.9307 Recall: 0.9216 F1 Score: 0.9261