Importing Libraries

```
!pip install -U kaleido
```

```
Collecting kaleido
  Downloading kaleido-0.2.1-py2.py3-none-manylinux1_x86_64.whl (79.9 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 79.9/79.9 MB 7.0 MB/s eta 0:00:00
Installing collected packages: kaleido
Successfully installed kaleido-0.2.1
```

```
# Libraries for exploring, handling and visualizing data
import pandas as pd, numpy as np, matplotlib.pyplot as plt, seaborn as sns, plotly.express as px
# Sklearn's preprocessing library
from sklearn.preprocessing import StandardScaler
# Importing train and test data split
from sklearn.model_selection import train_test_split
# Sklearn's metrics to evaluate our models
from sklearn.metrics import accuracy_score, precision_score, confusion_matrix, recall_score, f1_score
# Classifiers
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.tree import DecisionTreeClassifier


# Setting theme style and color palette to seaborn
sns.set_theme(context = 'notebook', style='darkgrid',palette='muted')
```

Obtaining Data

```
# Importing data
df = pd.read_csv("/content/creditcard.csv")
# Display dataframe
df.head()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V2
0	0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.11047
1	0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.10128
2	1	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.90941
3	1	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.19032
4	2	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.13745

5 rows × 31 columns

```
# Data type
df.dtypes
```

```
Time      int64
V1      float64
V2      float64
V3      float64
V4      float64
V5      float64
V6      float64
V7      float64
V8      float64
V9      float64
V10     float64
V11     float64
V12     float64
V13     float64
V14     float64
V15     float64
V16     float64
V17     float64
V18     float64
V19     float64
V20     float64
V21     float64
V22     float64
V23     float64
V24     float64
V25     float64
```

```
V26       float64
V27       float64
V28       float64
Amount    float64
Class     float64
dtype: object
```

All attributes are made of **numerical inputs**. Most of them displays floating point numbers (*float*) while **class** displays integer(*int*) and represents the categorical class of each transaction, whether they're a fraud or genuine.

```
# Verifying if there are any null values
df.isna().values.any()
```

```
True
```

```
# Statistics on the amounts
df.Amount.describe().round(2)
```

```
count    26231.00
mean        75.98
std        219.16
min          0.00
25%          6.14
50%         19.00
75%         68.00
max       7879.42
Name: Amount, dtype: float64
```

75% of transactions in the analyzed period were **up to €77.16**.

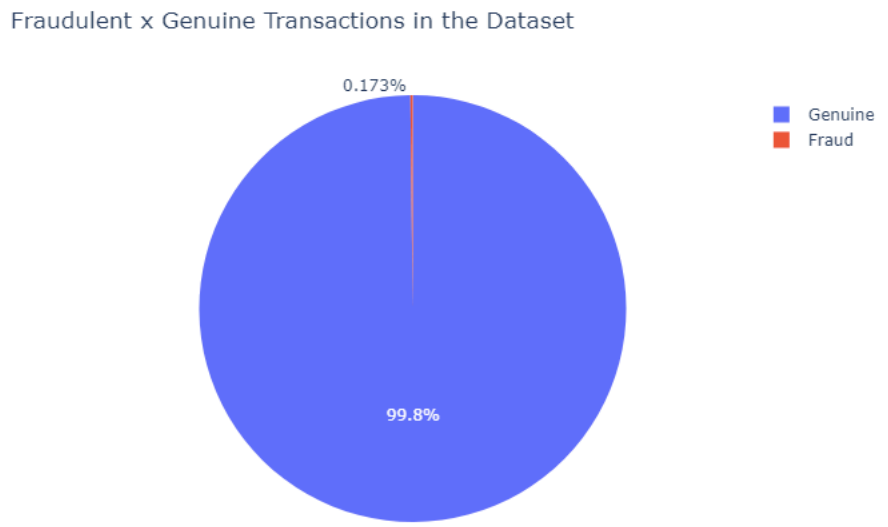
The **maximum amount** identified during this period was **€25,691.16**, way higher than the **average amount of €88.35**.

It looks like most transactions are genuine, represented by the **blue** dots on the chart above. We can also see that all high value transactions were genuine, with apparently **no fraudulent** transaction made being **above €5,000.00**

However, it seems hard to identify the fraudulent transactions, painted **yellow**, looking at the distribution of **amount values**. This leaves us with a question: **How many transactions were in fact fraud?**

Class Distribution

```
# Visualizing Class distribution
fig = px.pie(df.Class,values = df.Class.value_counts(),
             names=['Genuine', 'Fraud'], title='Fraudulent x Genuine Transactions in the Dataset')
fig.show('png')
```




```
df.Class.value_counts()
```

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

So it seems **only 492** transactions in the dataset were **fraudulent** which represents **only 0.173%** of data, there is a **huge class imbalance** that we have to work on here!

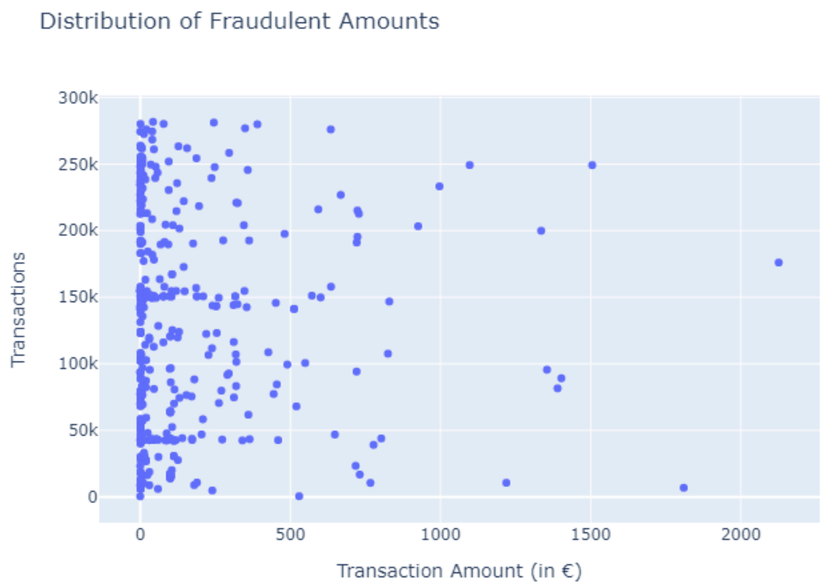
Let's see some statistics on the amounts of the frauds registered during the analyzed period.

```
df.query("Class ==1").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
```

75% percent of frauds were **below** the amount of **€105.89** and the largest fraud amount was €2,125.87. Let's see those values distributed in a chart.

```
# Distribution of fraudulent transactions amount
fig = px.scatter(df.query("Class==1"), x = 'Amount', y =df.query("Class==1").index,
                title = 'Distribution of Fraudulent Amounts')
fig.update_layout(xaxis_title='Transaction Amount (in €)',
                  yaxis_title='Transactions')
fig.show('png')
```



Preparing Data

For this project, we won't be using the **time** attribute, so we will remove it.

We will also use StandardScaler() to put all the data into the same scale, avoiding bias for a certain attribute when trying to predict our target variable, which is **Class**.

```
df = df.drop(columns = ['Time'], axis = 1)
df
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	...	V21	V2
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	...	-0.018307	0.2778
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	...	-0.225775	-0.6386
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	...	0.247998	0.7716
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	...	-0.108300	0.0052
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	...	-0.009431	0.7982
...	...	...	...	...	...	...	...	...	...	...	...	...	
284802	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.305334	1.914428	4.356170	...	0.213454	0.1118
284803	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.294869	0.584800	-0.975926	...	0.214205	0.9243
284804	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.708417	0.432454	-0.484782	...	0.232045	0.5782
284805	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.679145	0.392087	-0.399126	...	0.265245	0.8000
284806	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.414650	0.486180	-0.915427	...	0.261057	0.6430

284807 rows × 30 columns

Let's now divide our dataset into the **independent variables (X)** and the **target variable (y)**

```
X = df.drop(columns=['Class'], axis=1)
y = df.Class
```

```
y
```

```
0         0
1         0
2         0
3         0
4         0
         ..
284802    0
284803    0
284804    0
284805    0
284806    0
Name: Class, Length: 284807, dtype: int64
```

```
X
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	...	V20	V2
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	...	0.251412	-0.0183
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	...	-0.069083	-0.2257
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	...	0.524980	0.2479
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	...	-0.208038	-0.1083
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	...	0.408542	-0.0094
...	...	...	...	...	...	...	...	...	...	...	...	...	
284802	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.305334	1.914428	4.356170	...	1.475829	0.2134
284803	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.294869	0.584800	-0.975926	...	0.059616	0.2142
284804	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.708417	0.432454	-0.484782	...	0.001396	0.2320
284805	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.679145	0.392087	-0.399126	...	0.127434	0.2652
284806	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.414650	0.486180	-0.915427	...	0.382948	0.2610

284807 rows × 29 columns

Now, let's split our data into **training set** and **testing set**. I'll split them into a 70/30 proportion, where 70% of our data will be used for **training** while the 30% left will be used for **testing**.

```
train_x, test_x, train_y,test_y = train_test_split(X, y, test_size= .3, random_state = 123)
```

```
print('X Train size: ', train_x.shape)
print('X Test size: ', test_x.shape)
print('X Test proportion ', "%s%%"%round((len(test_x) / (len(train_x) + len(test_x))) * 100))
```

```
X Train size:  (199364, 29)
X Test size:  (85443, 29)
X Test proportion  30%
```

```
print('Y Train size: ', train_y.shape)
print('Y Test size: ', test_y.shape)
print('Y Test proportion ', "%s%%"%round((len(test_y) / (len(train_y) + len(test_y))) * 100))
```

```
Y Train size:  (199364,)
Y Test size:  (85443,)
Y Test proportion  30%
```

Normalizing 'Amount' feature with StandardScaler, separately on each set, in order to avoid **data leakage**.

```
# Scaling data on the training set
scaler = StandardScaler()
train_x['Amount'] = scaler.fit_transform(train_x.Amount.values.reshape(-1,1))
train_x
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	...	V20	V2
9057	1.223528	0.726064	-0.192303	1.315143	0.327134	-0.627426	0.103793	-0.166424	0.941614	-0.755328	...	-0.117242	-0.16810
197407	-1.531257	-0.845410	-0.661207	-0.010479	2.096034	-1.582374	0.644661	-0.146939	0.305072	-0.877905	...	-0.256196	-0.42538
257714	2.302551	-1.410263	-1.301974	-1.825564	-0.774062	0.000869	-1.163464	-0.018924	-1.428129	1.749254	...	-0.390696	-0.16447
201302	1.809691	0.232969	0.312680	3.745688	-0.357230	0.337521	-0.547228	0.286964	-0.538232	1.574565	...	-0.340672	-0.15248
167965	-2.449361	2.602426	-2.648017	0.169754	-0.043874	-1.789616	-0.259222	1.078845	-0.559213	-1.896160	...	-0.729064	0.47694
...	...	...	...	...	...	...	...	...	...	...	...	...	..
192476	2.085321	-1.119472	-0.260414	-0.829419	-1.373550	-0.504944	-1.224794	0.027016	0.011921	0.910736	...	-0.011516	0.52393
17730	-1.039001	0.950070	0.389899	-1.217401	1.855856	3.640886	-0.549604	1.505694	-0.559184	-0.619047	...	0.177503	-0.09052
28030	1.129333	0.471653	0.657500	2.454111	-0.091741	-0.089917	0.008581	0.060009	-0.623285	0.661338	...	-0.212507	-0.04824
277869	1.636784	-0.560857	-1.944589	0.405452	0.157569	-0.635650	0.315338	-0.200477	0.678971	-0.775364	...	0.276758	-0.10080
249342	1.906410	-0.531680	-1.175688	0.132728	-0.229802	-0.565090	-0.056376	-0.077301	0.978166	-0.041916	...	-0.156293	-0.10907

199364 rows × 29 columns

```
# Scaling data on the testing set
scaler = StandardScaler()
test_x['Amount'] = scaler.fit_transform(test_x.Amount.values.reshape(-1,1))
test_x
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	...	V20	V2
73129	-0.623235	1.097949	0.748810	0.763394	-0.179458	-0.258895	0.430106	0.466788	-0.935937	-0.283034	...	-0.066304	0.24313
229597	2.155748	-0.998223	-1.158978	-0.992298	-0.484600	-0.308857	-0.677077	-0.193517	-0.083026	0.705357	...	0.124636	0.17477
220218	1.614893	-0.194953	-2.050402	1.469645	0.540352	-0.665439	0.677713	-0.246032	-0.079937	-0.181429	...	0.173336	-0.02906
198374	1.908756	-2.517443	0.277391	-1.466555	-1.521858	3.005920	-2.800770	0.981435	0.349534	1.171678	...	-0.350968	0.05086
167980	2.120853	-1.048240	-1.895990	-1.236063	-0.038722	-0.274832	-0.388942	-0.196979	-0.649028	1.014140	...	0.176958	0.45595
...	...	...	...	...	...	...	...	...	...	...	...	...	..
64823	0.981634	-0.013797	0.702670	1.179459	0.141590	1.295216	-0.350124	0.487122	0.426048	-0.362567	...	-0.289286	-0.11179
144933	-0.188338	0.753188	0.535544	-0.242559	0.103888	-0.044430	0.423675	0.077219	0.369646	-0.980602	...	-0.078595	0.33994
31407	-0.959696	0.736918	1.722280	0.265029	0.769584	-0.443858	0.885135	-0.035855	-0.407058	-0.869518	...	0.003919	-0.07465
28343	1.293597	-0.527259	0.659631	-0.775476	-0.962009	-0.355633	-0.699761	0.061105	-1.084533	0.719309	...	0.071060	0.00271
173170	1.936092	-1.068767	-1.469130	-0.762940	-0.541493	-0.136569	-0.934574	0.083509	-0.144116	0.207809	...	0.269209	0.31798

85443 rows × 29 columns

Now, considering that we're dealing with **imbalanced** data, we must apply **SMOTE** in order to **oversample** our fraudulent data.

SMOTE will synthetically **generate more** samples of fraudulent data based on the frauds that we already have in the original dataset.

```
y.value_counts() # 0 = Genuine Transactions | 1 = Fraud
```

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

```
from imblearn.over_sampling import SMOTE
train_x, train_y = SMOTE().fit_resample(train_x,train_y) # Reshaping data
```

```
train_y.value_counts()
```

```
0    199032
1    199032
Name: Class, dtype: int64
```

Now we have a 50 | 50 data balance between genuine and fraudulent transactions.

Note: I've **only corrected the imbalance between transactions in the training set**, while maintaining the **test set** with its original proportions, because **the test set should be a representation of reality**.

Applying Classifiers

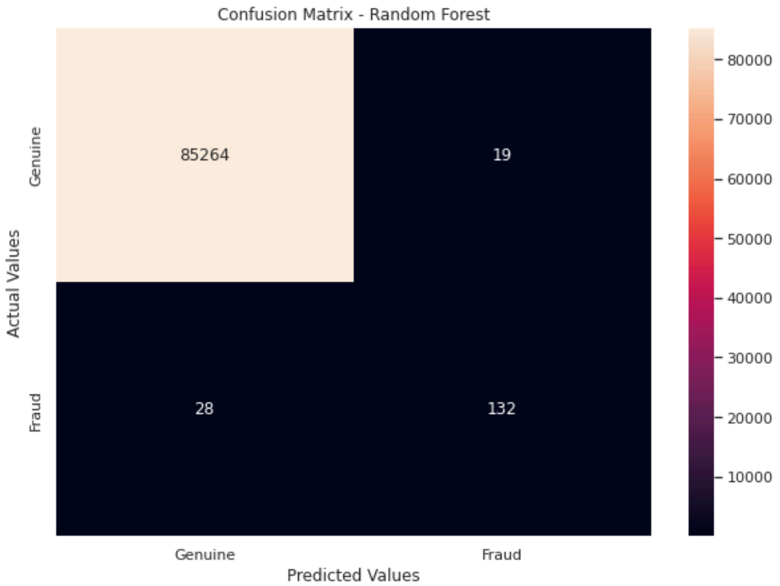
```
# Applying Random Forest Classifier
random_forest = RandomForestClassifier(n_estimators = 100, random_state = 123)
random_forest.fit(train_x,train_y)

y_predictions_rf = random_forest.predict(test_x)

# Applying Decision Tree Classifier
decision_tree = DecisionTreeClassifier(random_state = 123)
decision_tree.fit(train_x,train_y)

y_predictions_dt = decision_tree.predict(test_x)

# Applying Ada Boost Classifier
ada_boost = AdaBoostClassifier(n_estimators = 100, random_state = 123)
ada_boost.fit(train_x,train_y)

y_predictions_ab = ada_boost.predict(test_x)

# Applying Gradient Boosting Classifier
gradient_boosting = GradientBoostingClassifier(n_estimators = 100, random_state = 123)
gradient_boosting.fit(train_x,train_y)

y_prediction_gb = gradient_boosting.predict(test_x)
```

Random Forest Scores

```
# Printing Evaluation Metrics for Random Forest
metrics = [['Accuracy',(accuracy_score(test_y, y_predictions_rf))],
                         ['Precision',precision_score(test_y, y_predictions_rf)],
                         ['Recall', recall_score(test_y, y_predictions_rf)],
                         ['F1_score',f1_score(test_y, y_predictions_rf)]]
metrics_df = pd.DataFrame(metrics, columns = ['Metrics', 'Results'])
metrics_df
```

	Metrics	Results
0	Accuracy	0.999450
1	Precision	0.874172
2	Recall	0.825000
3	F1_score	0.848875

```
# Confusion Matrix for Random Forest
confusion_matrix_rf = confusion_matrix(test_y, y_predictions_rf)
# Visualization
plt.figure(figsize=(10,7))
ax = plt.subplot()
sns.heatmap(confusion_matrix_rf, annot=True, fmt='g', ax = ax)
ax.set_xlabel('Predicted Values')
ax.set_ylabel('Actual Values')
ax.set_title('Confusion Matrix - Random Forest')
ax.xaxis.set_ticklabels(['Genuine','Fraud'])
ax.yaxis.set_ticklabels(['Genuine','Fraud'])
plt.show()
```

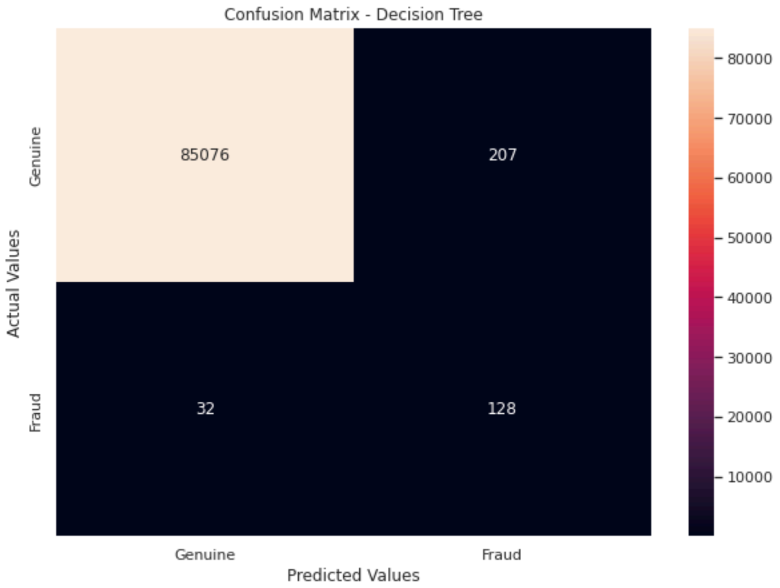


Decision Tree Scores

```
# Printing Evaluation Metrics for Decision Tree
metrics_df = [['Accuracy',(accuracy_score(test_y, y_predictions_dt))],
                          ['Precision',precision_score(test_y, y_predictions_dt)],
                          ['Recall', recall_score(test_y, y_predictions_dt)],
                          ['F1_score',f1_score(test_y, y_predictions_dt)]]
metrics_df_dt = pd.DataFrame(metrics_df, columns = ['Metrics', 'Results'])
metrics_df_dt
```

	Metrics	Results
0	Accuracy	0.997203
1	Precision	0.382090
2	Recall	0.800000
3	F1_score	0.517172

```
# Confusion Matrix for Decision Tree
confusion_matrix_dt = confusion_matrix(test_y, y_predictions_dt)
# Visualization
plt.figure(figsize=(10,7))
ax = plt.subplot()
sns.heatmap(confusion_matrix_dt, annot=True, fmt='g', ax = ax)
ax.set_xlabel('Predicted Values')
ax.set_ylabel('Actual Values')
ax.set_title('Confusion Matrix - Decision Tree')
ax.xaxis.set_ticklabels(['Genuine','Fraud'])
ax.yaxis.set_ticklabels(['Genuine','Fraud'])
plt.show()
```

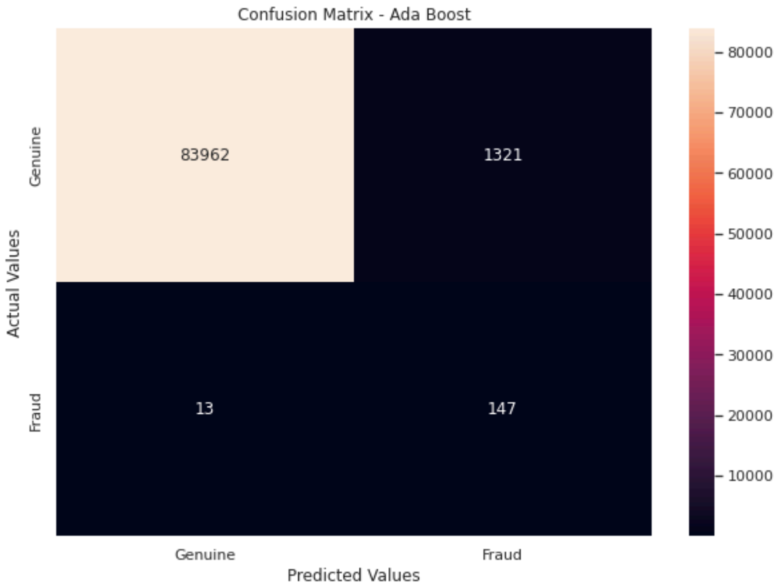


Ada Boost Scores

```
# Printing Evaluation Metrics for Ada Boost
metrics_ab = [['Accuracy',(accuracy_score(test_y, y_predictions_ab))],
                         ['Precision',precision_score(test_y, y_predictions_ab)],
                         ['Recall', recall_score(test_y, y_predictions_ab)],
                         ['F1_score',f1_score(test_y, y_predictions_ab)]]
metrics_df_ab = pd.DataFrame(metrics_ab, columns = ['Metrics', 'Results'])
metrics_df_ab
```

	Metrics	Results
0	Accuracy	0.984387
1	Precision	0.100136
2	Recall	0.918750
3	F1_score	0.180590

```
# Confusion Matrix for Ada Boost
confusion_matrix_ab = confusion_matrix(test_y, y_predictions_ab)
# Visualization
plt.figure(figsize=(10,7))
ax = plt.subplot()
sns.heatmap(confusion_matrix_ab, annot=True, fmt='g', ax = ax)
ax.set_xlabel('Predicted Values')
ax.set_ylabel('Actual Values')
ax.set_title('Confusion Matrix - Ada Boost')
ax.xaxis.set_ticklabels(['Genuine','Fraud'])
ax.yaxis.set_ticklabels(['Genuine','Fraud'])
plt.show()
```



Gradient Boosting Scores

```
# Printing Evaluation Metrics for Gradient Boosting
metrics_gb = [['Accuracy',(accuracy_score(test_y, y_prediction_gb))],
                         ['Precision',precision_score(test_y, y_prediction_gb)],
                         ['Recall', recall_score(test_y, y_prediction_gb)],
                         ['F1_score',f1_score(test_y, y_prediction_gb)]]
metrics_df_gb = pd.DataFrame(metrics_gb, columns = ['Metrics', 'Results'])
metrics_df_gb
```

	Metrics	Results
0	Accuracy	0.987547
1	Precision	0.120168
2	Recall	0.893750
3	F1_score	0.211852

```
# Confusion Matrix for Gradient Boosting
confusion_matrix_gb = confusion_matrix(test_y, y_prediction_gb)
# Visualization
plt.figure(figsize=(10,7))
ax = plt.subplot()
sns.heatmap(confusion_matrix_gb, annot=True, fmt='g', ax = ax)
ax.set_xlabel('Predicted Values')
ax.set_ylabel('Actual Values')
ax.set_title('Confusion Matrix - Gradient Boosting')
ax.xaxis.set_ticklabels(['Genuine','Fraud'])
ax.yaxis.set_ticklabels(['Genuine','Fraud'])
plt.show()
```