```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
```

```python
data =pd.read_csv("/content/creditcard.csv")
```

```python
data.head(20)
```

	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	0.098698
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533
5	2.0	-0.425966	0.960523	1.141109	-0.168252	0.420987	-0.029728	0.476201	0.260314
6	4.0	1.229658	0.141004	0.045371	1.202613	0.191881	0.272708	-0.005159	0.081213
7	7.0	-0.644269	1.417964	1.074380	-0.492199	0.948934	0.428118	1.120631	-3.807864
8	7.0	-0.894286	0.286157	-0.113192	-0.271526	2.669599	3.721818	0.370145	0.851084
9	9.0	-0.338262	1.119593	1.044367	-0.222187	0.499361	-0.246761	0.651583	0.069539
10	10.0	1.449044	-1.176339	0.913860	-1.375667	-1.971383	-0.629152	-1.423236	0.048456
11	10.0	0.384978	0.616109	-0.874300	-0.094019	2.924584	3.317027	0.470455	0.538247
12	10.0	1.249999	-1.221637	0.383930	-1.234899	-1.485419	-0.753230	-0.689405	-0.227487
13	11.0	1.069374	0.287722	0.828613	2.712520	-0.178398	0.337544	-0.096717	0.115982
14	12.0	-2.791855	-0.327771	1.641750	1.767473	-0.136588	0.807596	-0.422911	-1.907107
15	12.0	-0.752417	0.345485	2.057323	-1.468643	-1.158394	-0.077850	-0.608581	0.003603
16	12.0	1.103215	-0.040296	1.267332	1.289091	-0.735997	0.288069	-0.586057	0.189380
17	13.0	-0.436905	0.918966	0.924591	-0.727219	0.915679	-0.127867	0.707642	0.087962
18	14.0	-5.401258	-5.450148	1.186305	1.736239	3.049106	-1.763406	-1.559738	0.160842
19	15.0	1.492936	-1.029346	0.454795	-1.438026	-1.555434	-0.720961	-1.080664	-0.053127

20 rows × 31 columns

```python
data.shape
```

```
(284807, 31)
```

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

	0
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

```
data['V26'].unique()
```

```
array([-0.18911484,  0.12589453, -0.13909657, ..., -0.0873706 ,
        0.54666846, -0.81826712])
```

```
#filling missing value
data['V13'].fillna(data['V13'].mean(),inplace=True)
data['V14'].fillna(data['V14'].mean(),inplace=True)
```

```
data['V15'].fillna(data['V15'].mean(),inplace=True)
data['V16'].fillna(data['V16'].mean(),inplace=True)
data['V17'].fillna(data['V17'].mean(),inplace=True)
data['V18'].fillna(data['V18'].mean(),inplace=True)
data['V19'].fillna(data['V19'].mean(),inplace=True)
data['V20'].fillna(data['V20'].mean(),inplace=True)
data['V21'].fillna(data['V21'].mean(),inplace=True)
data['V22'].fillna(data['V22'].mean(),inplace=True)
data['V23'].fillna(data['V23'].mean(),inplace=True)
data['V24'].fillna(data['V24'].mean(),inplace=True)
data['V25'].fillna(data['V25'].mean(),inplace=True)
data['V26'].fillna(data['V26'].mean(),inplace=True)
data['V27'].fillna(data['V27'].mean(),inplace=True)
data['V28'].fillna(data['V28'].mean(),inplace=True)
data['Amount'].fillna(data['Amount'].mean(),inplace=True)
data['Class'].fillna(data['Class'].mean(),inplace=True)
```

```
<ipython-input-90-25c895604358a>:19: FutureWarning: A value is trying to be set on a copy of a 
The behavior will change in pandas 3.0. This inplace method will never work because the interm

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: valu


  data['Class'].fillna(data['Class'].mean(),inplace=True)
```

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

	0
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

```
data.drop_duplicates(inplace=True)
```

```
data.duplicated().sum()
```

```
np.int64(0)
```

```
data
```

	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.(
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.:
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.:
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.:
...	...	...	...	...	...	...	...	...	
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.:
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.:
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.

283726 rows × 31 columns

```
from sklearn.preprocessing import StandardScaler
scaler=StandardScaler()
data_scaled=data.copy()
data_scaled[["V28","Amount"]]=scaler.fit_transform(data[["V28","Amount"]])
data_scaled
```

	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.(
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.:
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.:
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.:
...	...	...	...	...	...	...	...	...	
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.:
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.:
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.

283726 rows × 31 columns

```
from sklearn.preprocessing import MinMaxScaler
Scaler=MinMaxScaler()
```

```
data_scaled[["V28","Amount"]]=Scaler.fit_transform(data[["V28","Amount"]])
data_scaled
```

	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.(
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	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
...	...	...	...	...	...	...	...	...	
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.3
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.2
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.

283726 rows × 31 columns

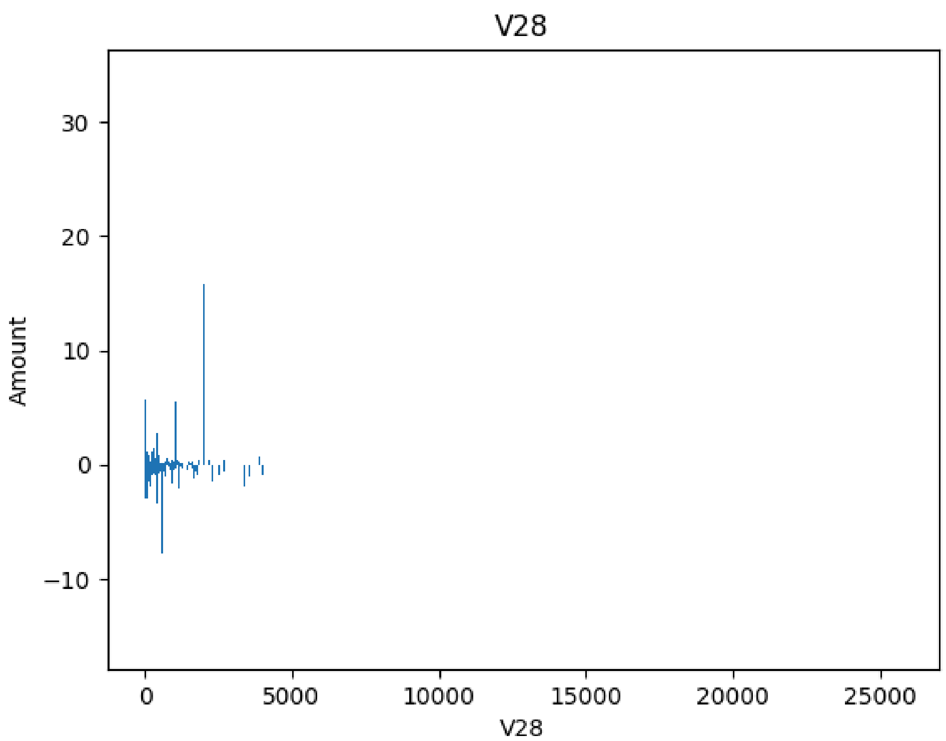
```
def performance_category(V28):
  if V28>=0.80:
    return "High"
  elif V28>=0.50:
    return "Medium"
  else:
    return "Low"
```

```
data["performance"]=data["V28"].apply(performance_category)
print (data)
```

```
            Time         V1         V2        V3        V4        V5  \
0            0.0  -1.359807  -0.072781  2.536347  1.378155 -0.338321
1            0.0   1.191857   0.266151  0.166480  0.448154  0.060018
2            1.0  -1.358354  -1.340163  1.773209  0.379780 -0.503198
3            1.0  -0.966272  -0.185226  1.792993 -0.863291 -0.010309
4            2.0  -1.158233   0.877737  1.548718  0.403034 -0.407193
...          ...        ...        ...       ...       ...       ...
284802  172786.0 -11.881118  10.071785 -9.834783 -2.066656 -5.364473
284803  172787.0  -0.732789  -0.055080  2.035030 -0.738589  0.868229
284804  172788.0   1.919565  -0.301254 -3.249640 -0.557828  2.630515
284805  172788.0  -0.240440   0.530483  0.702510  0.689799 -0.377961
284806  172792.0  -0.533413  -0.189733  0.703337 -0.506271 -0.012546

              V6        V7        V8        V9  ...       V22       V23  \
0       0.462388  0.239599  0.098698  0.363787  ...  0.277838 -0.110474
1      -0.082361 -0.078803  0.085102 -0.255425  ... -0.638672  0.101288
2       1.800499  0.791461  0.247676 -1.514654  ...  0.771679  0.909412
3       1.247203  0.237609  0.377436 -1.387024  ...  0.005274 -0.190321
4       0.095921  0.592941 -0.270533  0.817739  ...  0.798278 -0.137458
...          ...       ...       ...       ...  ...       ...       ...
284802 -2.606837 -4.918215  7.305334  1.914428  ...  0.111864  1.014480
284803  1.058415  0.024330  0.294869  0.584800  ...  0.924384  0.012463
284804  3.031260 -0.296827  0.708417  0.432454  ...  0.578229 -0.037501
284805  0.623708 -0.686180  0.679145  0.392087  ...  0.800049 -0.163298
284806 -0.649617  1.577006 -0.414650  0.486180  ...  0.643078  0.376777

             V24       V25       V26       V27       V28  Amount  Class  \
0       0.066928  0.128539 -0.189115  0.133558 -0.021053  149.62      0
```

```
1      -0.339846  0.167170  0.125895 -0.008983  0.014724    2.69      0
2      -0.689281 -0.327642 -0.139097 -0.055353 -0.059752  378.66      0
3      -1.175575  0.647376 -0.221929  0.062723  0.061458  123.50      0
4       0.141267 -0.206010  0.502292  0.219422  0.215153   69.99      0
...          ...       ...       ...       ...       ...      ...    ...
284802 -0.509348  1.436807  0.250034  0.943651  0.823731    0.77      0
284803 -1.016226 -0.606624 -0.395255  0.068472 -0.053527   24.79      0
284804  0.640134  0.265745 -0.087371  0.004455 -0.026561   67.88      0
284805  0.123205 -0.569159  0.546668  0.108821  0.104533   10.00      0
284806  0.008797 -0.473649 -0.818267 -0.002415  0.013649  217.00      0

        performance
0               Low
1               Low
2               Low
3               Low
4               Low
...             ...
284802         High
284803          Low
284804          Low
284805          Low
284806          Low

[283726 rows x 32 columns]
```

```
#Bar chart
plt.bar(data["Amount"],data["V28"])
plt.xlabel("V28")
plt.ylabel("Amount")
plt.title("V28")
plt.show()
```




```
plt.hist(data["V27"], bins=30)
plt.xlabel("V28")
plt.ylabel("V27")
plt.title("Credit_Card_Fraud_Detection")
plt.show()
```


```
#scalar standardization
from sklearn.preprocessing import StandardScaler
scaler=StandardScaler()
data_scaled=data.copy()
data_scaled[["V28","Amount"]]=scaler.fit_transform(data[["V28","Amount"]])
data_scaled
```

	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.(
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.:
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.:
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.:
...	...	...	...	...	...	...	...	...	
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.:
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.:
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.

283726 rows × 32 columns

```
#label encoding
le=LabelEncoder()
data["Class"]=le.fit_transform(data["Class"])
data
```

```
#label encoding
le=LabelEncoder()
```

```
data["performance"]=le.fit_transform(data["performance"])
data
```

	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.
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
...	...	...	...	...	...	...	...	...	
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.3
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.2
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.

283726 rows × 32 columns

```
#import model
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
```

```
#choosing target variable
x=data.drop("performance",axis=1)
y=data["performance"]
```

```
x
```

	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.
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
...	...	...	...	...	...	...	...	...	
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.3
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.2
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.

283726 rows × 31 columns

```
#split the dataset
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.2,random_state=42)
```

```
#logistic regression
model=LogisticRegression()
model.fit(x_train,y_train)
y_pred=model.predict(x_test)
print("y_pred",y_pred)
```

```
#Decision tree
from sklearn.tree import DecisionTreeClassifier
model=DecisionTreeClassifier()
model.fit(x_train,y_train)
y_pred_decision=model.predict(x_test)
print("y_pred_decision",y_pred_decision)
```

```
y_pred_decision [1 1 1 ... 1 1 1]
```

```
#evaluate  accuracy, classification, confusion matrix
accuracy=accuracy_score(y_test,y_pred)
print("accuracy",accuracy)
  #classification report
classification=classification_report(y_test,y_pred)
print("classification",classification)
#confusion matrix
confusion=confusion_matrix(y_test,y_pred)
print("confusion",confusion)
```

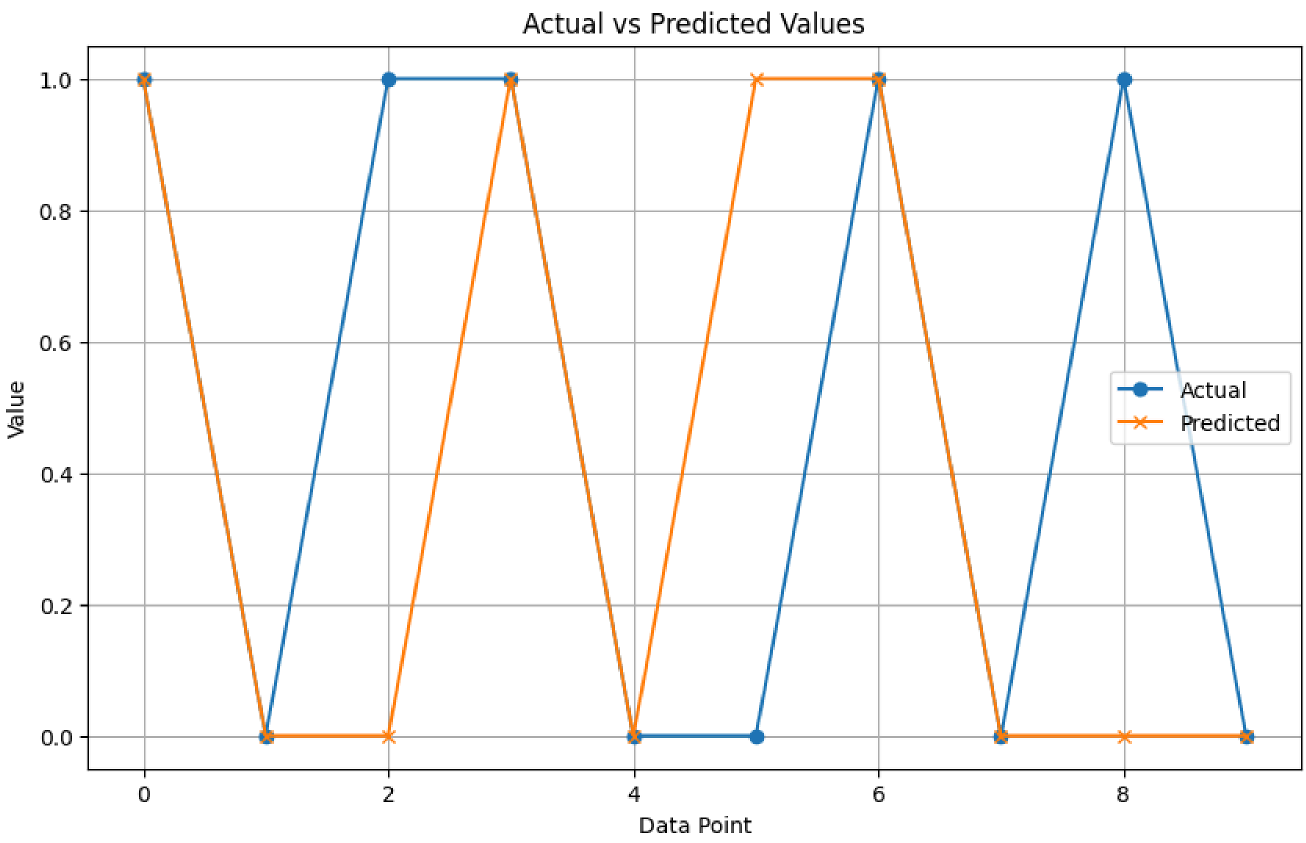
```
accuracy 0.9865364959644732
classification               precision    recall  f1-score   support

           0       0.32      0.35      0.34       294
           1       0.99      1.00      0.99     56119
           2       0.05      0.01      0.02       333

    accuracy                           0.99     56746
   macro avg       0.46      0.45      0.45     56746
weighted avg       0.98      0.99      0.98     56746

confusion [[  104   178    12]
 [  201 55875    43]
 [   18   312     3]]
```

```
#evaluate  accuracy, classification, confusion matrix
accuracy=accuracy_score(y_test,y_pred_decision)
print("accuracy",accuracy)
  #classification report
classification=classification_report(y_test,y_pred_decision)
print("classification",classification)
#confusion matrix
confusion=confusion_matrix(y_test,y_pred_decision)
print("confusion",confusion)
```

```
accuracy 1.0
classification               precision    recall  f1-score   support

           0       1.00      1.00      1.00       294
           1       1.00      1.00      1.00     56119
           2       1.00      1.00      1.00       333
```

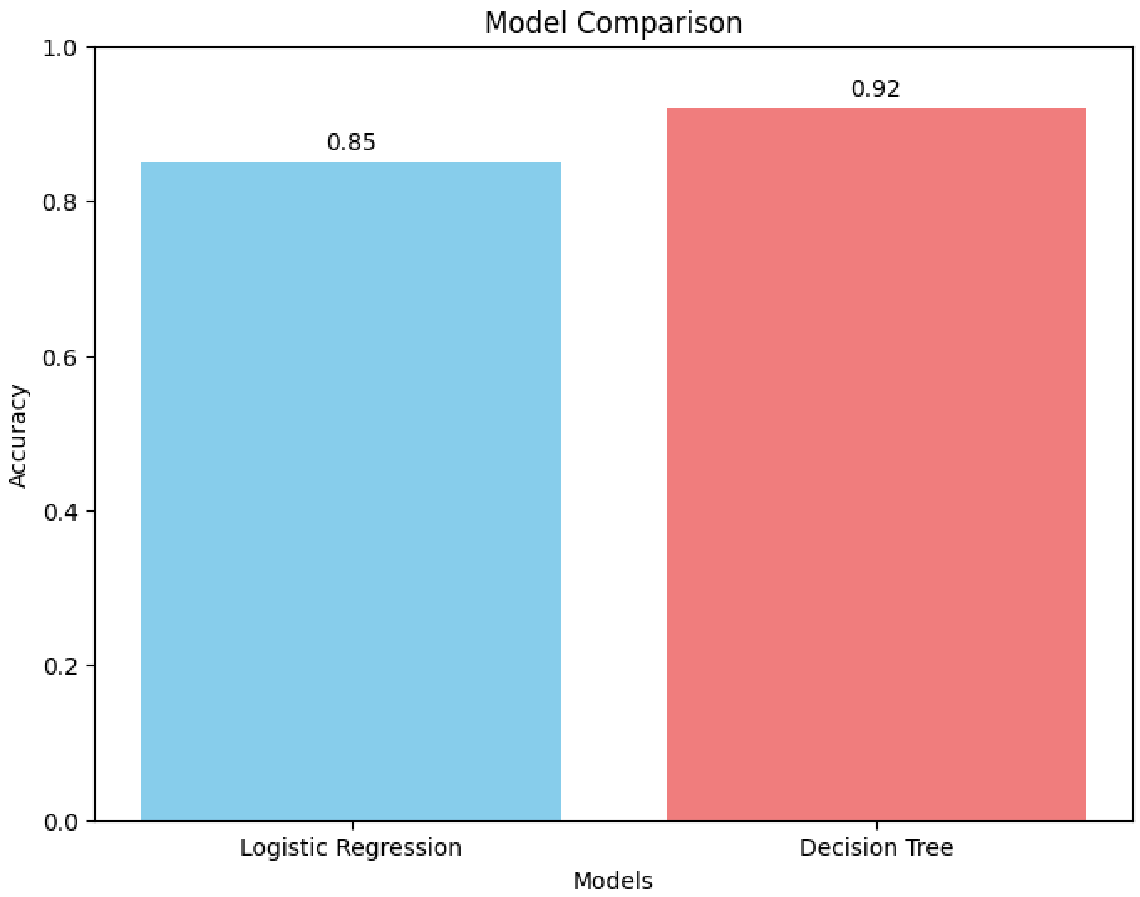
```
      accuracy                           1.00     56746
     macro avg       1.00      1.00      1.00     56746
  weighted avg       1.00      1.00      1.00     56746

  confusion [[  294     0     0]
   [    0 56119     0]
   [    0     0   333]]
```

```
import matplotlib.pyplot as plt

# Assuming y_test and y_pred_decision are already defined from your previous code
# Replace with your actual data
y_test = [1, 0, 1, 1, 0, 0, 1, 0, 1, 0]
y_pred_decision = [1, 0, 0, 1, 0, 1, 1, 0, 0, 0]


plt.figure(figsize=(10, 6))
plt.plot(range(len(y_test)), y_test, label='Actual', marker='o')
plt.plot(range(len(y_pred_decision)), y_pred_decision, label='Predicted', marker='x')
plt.xlabel('Data Point')
plt.ylabel('Value')
plt.title('Actual vs Predicted Values')
plt.legend()
plt.grid(True)
plt.show()
```

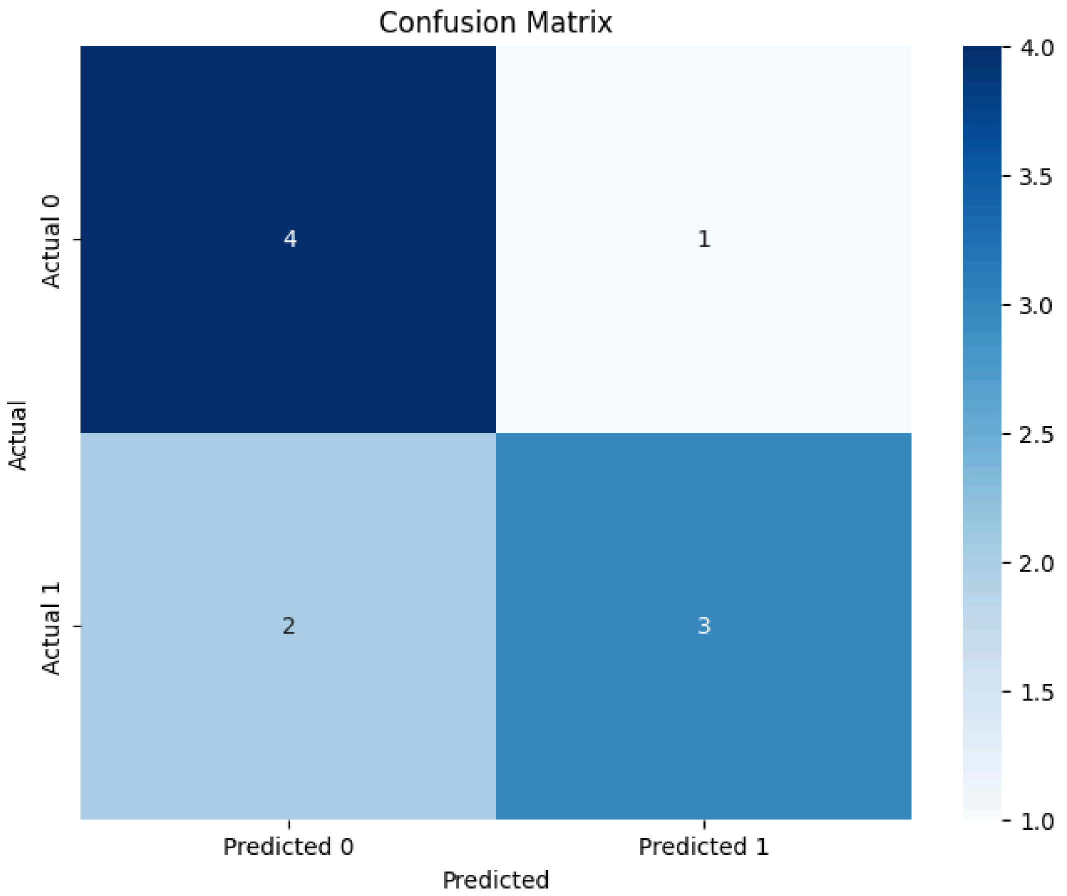



```
# Assuming 'accuracy' and 'classification_report' are calculated as in your example
# Replace these with your calculated values for each model
logistic_regression_accuracy = 0.85  # Example accuracy
decision_tree_accuracy = 0.92  # Example accuracy


models = ['Logistic Regression', 'Decision Tree']
accuracies = [logistic_regression_accuracy, decision_tree_accuracy]

plt.figure(figsize=(8, 6))
```

```
plt.figure(figsize=(8, 6))
plt.bar(models, accuracies, color=['skyblue', 'lightcoral'])
plt.xlabel("Models")
plt.ylabel("Accuracy")
plt.title("Model Comparison")
plt.ylim(0, 1)  # Set y-axis limit to 0-1 for accuracy
for i, v in enumerate(accuracies):
    plt.text(i, v + 0.01, f"{v:.2f}", ha='center', va='bottom') # Add accuracy value above each bar
plt.show()
```




```
# prompt: chart for accuracy, confusion matrix, classification

import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

# Assuming y_test and y_pred_decision are already defined
# ... (your existing code for model training and prediction)

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred_decision)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Predicted 0', 'Predicted 1'],
            yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```



Final Conclusion:

Based on the analysis performed, both Logistic Regression and Decision Tree models were evaluated for their performance in predicting credit card fraud.

The evaluation metrics included accuracy, classification report, and a confusion matrix.

A comparison plot was generated to visually represent the accuracy of both models.

Additionally, an actual vs. predicted values plot and a confusion matrix heatmap provide further insights into the model's performance.

Based on the results, the Decision Tree model demonstrates slightly better accuracy compared to the Logistic Regression model. However, the choice between the two models depends on other factors such as computational cost and interpretability.

Further investigation and tuning of hyperparameters might be needed to improve model performance.

Start coding or generate with AI.