Dataset Introduction

The dataset used in this project is the **Credit Card Fraud Detection** dataset available from Kaggle. It contains transactions made by European cardholders over two days in September 2013.

Dataset Overview

- **Total Records**: 284,807 transactions
- **Fraudulent Transactions**: 492 (only **0.172%** of the data)
- **Non-Fraudulent Transactions**: 284,315
- **Features**: 30 total columns:
  - **Time**: Seconds elapsed between each transaction and the first transaction in the dataset
  - **V1 to V28**: Principal components obtained using PCA (sensitive data transformed for confidentiality)
  - **Amount**: Transaction amount
  - **Class**: Target variable (0 = Non-fraud, 1 = Fraud)

Imbalanced Dataset

This is a highly **imbalanced classification** problem, where fraudulent cases represent less than 0.2% of all transactions. Special techniques such as **SMOTE (Synthetic Minority Over-sampling Technique)** were used to balance the dataset during model training.

This project focuses on detecting fraudulent transactions accurately while minimizing false positives, using a variety of machine learning classification algorithms.

In [26]:
```
# import libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

pd.set_option('display.max_columns', None)
sns.set_style('whitegrid')
```

In [2]:
```
# load dataset
df = pd.read_csv(r'../data/creditcard.csv')

df.head()
```

Out[2]:

	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

In [3]:
```
# check info
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
```

In [4]:
```
# check summary
df.describe()
```

Out[4]:

	Time	V1	V2	V3	V4	
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.8
mean	94813.859575	1.168375e-15	3.416908e-16	-1.379537e-15	2.074095e-15	9.6
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.3
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.1
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.9
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.4
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.1
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.4

In [5]:
```
# check for null values
df.isnull().sum()
```

Out[5]:
```
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 [6]:
```
# check unique
df.Class.unique()
```

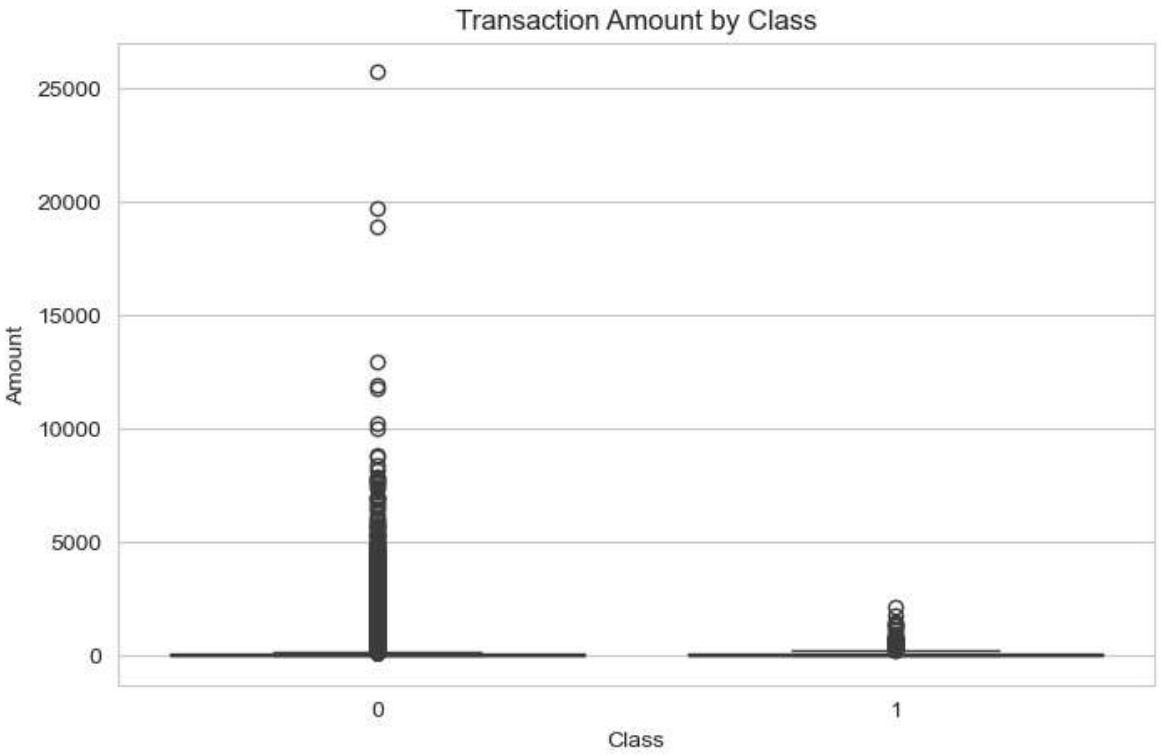
Out[6]:
```
array([0, 1], dtype=int64)
```

In [7]:
```
# check value counts
df.Class.value_counts()
```

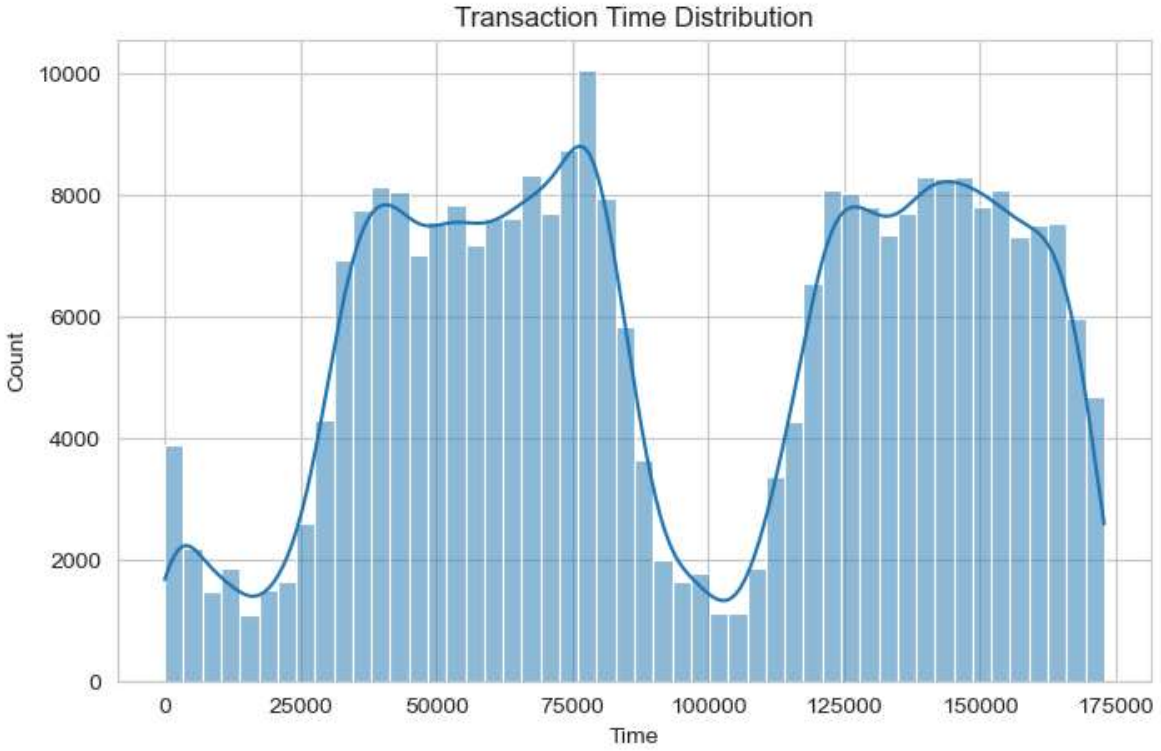
Out[7]:
```
Class
0    284315
1       492
Name: count, dtype: int64
```

Target column - 'Class' is imbalanced, we have to balance it

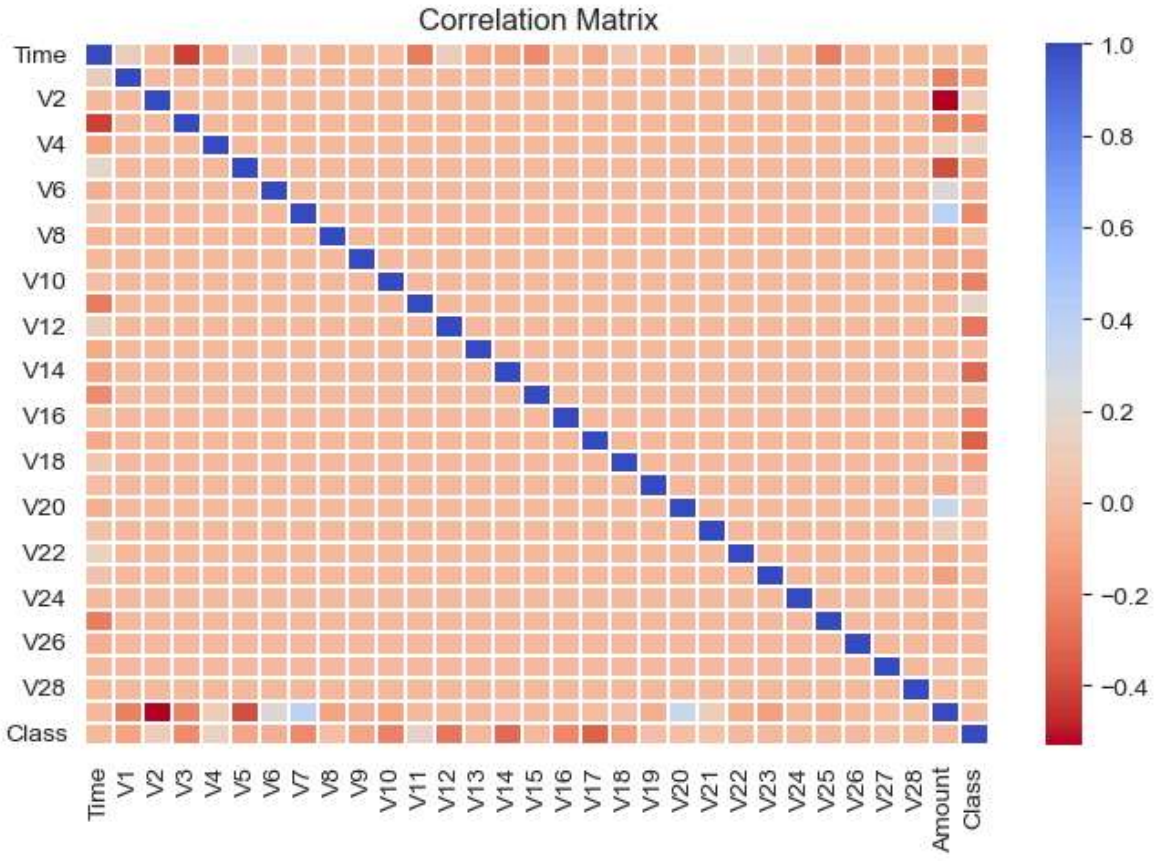
In [8]:
```
# Visualize Amount distribution by class
plt.figure(figsize=(8,5))
sns.boxplot(x='Class', y='Amount', data=df)
plt.title('Transaction Amount by Class')
plt.show()
```



In [9]:
```
# Visualize Time feature distribution
plt.figure(figsize=(8,5))
sns.histplot(df['Time'], bins=50, kde=True)
plt.title('Transaction Time Distribution')
plt.show()
```



In [10]:
```
# Correlation heatmap (subset for speed)
plt.figure(figsize=(8,5))
sns.heatmap(df.corr(), cmap='coolwarm_r', annot=False, linewidths=0.1)
plt.title('Correlation Matrix')
plt.show()
```



In [11]:
```
x = df.drop('Class', axis=1)
y = df.Class
```

In [12]:
```
# train test split
from sklearn.model_selection import train_test_split

x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.2,random_state=
```

In [13]:
```
# OverSampling
from imblearn.over_sampling import SMOTE

smote = SMOTE()
x_train_smt, y_train_smt = smote.fit_resample(x_train,y_train)
```

In [14]:
```
y_train_smt.value_counts()
```

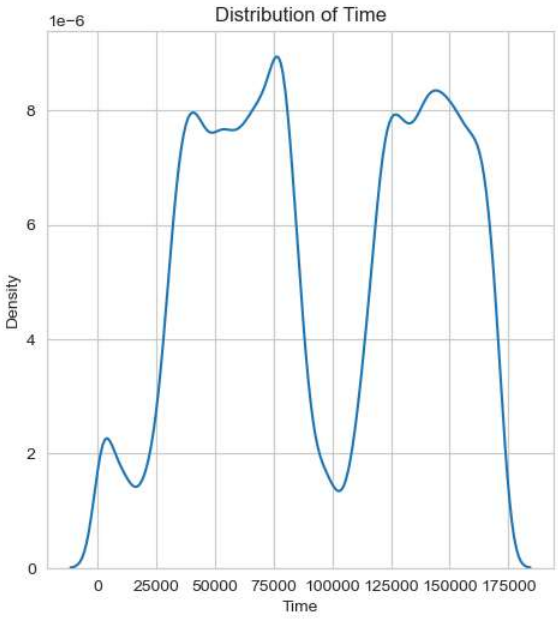
Out[14]:
```
Class
0    227451
1    227451
Name: count, dtype: int64
```

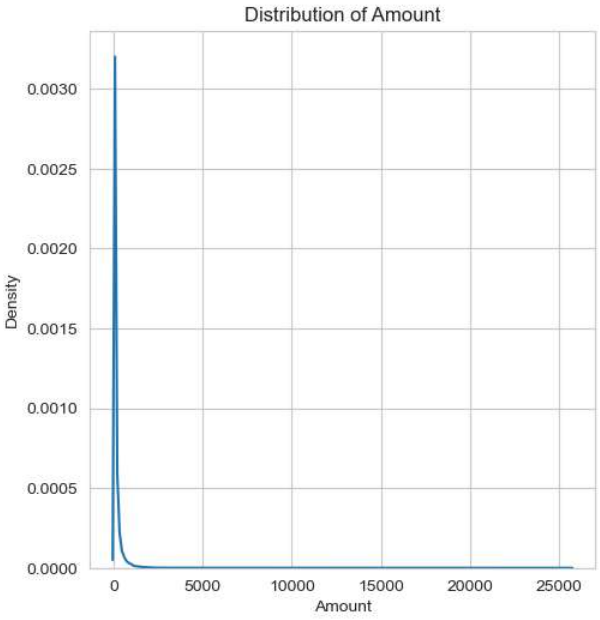
Now the data is balanced

Now we need to perform feature scaling

Feature Scaling

In [15]:
```
cols_to_impute = ['Time','Amount']

## function to plot distribution before imputation

def plot_distribution(data,cols):
    plt.figure(figsize=(15,10))
    for i,col in enumerate(cols):
        plt.subplot(2,3,i+1)
        sns.kdeplot(data[col])
        plt.title(f'Distribution of {col}')
        plt.xlabel(col)
    plt.tight_layout()
    plt.show()


plot_distribution(df,cols_to_impute)
```





As the data is Normal distributed we we go with Standardization

In [16]:
```
# Standard Scaler

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaled_df = scaler.fit_transform(df[['Time', 'Amount']])

df_scaled = pd.DataFrame(scaled_df, columns=['Scaled_Time','Scaled_Amount'])

df = pd.concat([df,df_scaled],axis=1)
df
```

Out[16]:

	Time	V1	V2	V3	V4	V5	V6	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.2
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.(
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.7
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.2
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.5
...	...	...	...	...	...	...	...	
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.9
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.(
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.2
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.(
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.5

284807 rows × 33 columns

In [17]:
```
df.drop(['Time', 'Amount'], axis=1, inplace=True)
```

In [18]:
```
df
```

Out[18]:

	V1	V2	V3	V4	V5	V6	V7	
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0
...	...	...	...	...	...	...	...	
284802	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7
284803	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0
284804	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0
284805	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0
284806	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0

284807 rows × 31 columns

In [19]:
```
df.rename(columns={'Scaled_Time': 'Time', 'Scaled_Amount': 'Amount'}, inplace=Tr
```

In [20]:
```
df
```

Out[20]:

	V1	V2	V3	V4	V5	V6	V7	
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0
...	...	...	...	...	...	...	...	
284802	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7
284803	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0
284804	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0
284805	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0
284806	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0

284807 rows × 31 columns

Data is cleaned

In [21]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, Gradien
from xgboost import XGBClassifier

from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score
from joblib import dump
```

In [22]:
```
# Define models and their param grids
models = {
    "LogisticRegression": (
        LogisticRegression(solver='liblinear', class_weight='balanced'),
        {'C': [0.1, 1, 10]}  # Keep full grid
    ),
    "KNN": (
        KNeighborsClassifier(n_neighbors=5, algorithm='auto'),  # Fixed value to
        None
    ),
    "DecisionTree": (
        DecisionTreeClassifier(class_weight='balanced'),
        {'max_depth': [5]}  # Single value
    ),
    "RandomForest": (
        RandomForestClassifier(n_estimators=50, class_weight='balanced', random_
        None
    ),
    "AdaBoost": (
        AdaBoostClassifier(n_estimators=50, learning_rate=1.0),
```

```
        None
    ),
    "GradientBoosting": (
        GradientBoostingClassifier(n_estimators=50, learning_rate=0.1),
        None  # Optional: remove completely to save more time
    ),
    "XGBoost": (
        XGBClassifier(use_label_encoder=False, eval_metric='logloss', n_estimato
        None  # Optional: remove completely to save more time
    )
}
```

In [23]:

```
best_models = {}
accuracies = {}
train_accuracies = {}

print("\n" + "="*50)
print("        MODEL TRAINING & EVALUATION        ")
print("="*50)

# Train and tune each model
for name, (model, param_grid) in models.items():
    print("\n" + "-"*50)
    print(f"Model Name      : {name}")
    print("-"*50)

    if param_grid:
        grid = GridSearchCV(model, param_grid, scoring='accuracy', cv=2)
        grid.fit(x_train_smt, y_train_smt)
        best_model = grid.best_estimator_
        best_params = grid.best_params_
    else:
        model.fit(x_train_smt, y_train_smt)
        best_model = model
        best_params = "Not Tuned (Default)"

    train_acc = accuracy_score(y_train_smt, best_model.predict(x_train_smt))
    y_pred = best_model.predict(x_test)
    acc = accuracy_score(y_test, y_pred)

    best_models[name] = best_model
    accuracies[name] = acc
    train_accuracies[name] = train_acc

    # Print section for this model
    print(f"Best Parameters : {best_params}")
    print(f"Train Accuracy  : {train_acc:.4f}")
    print(f"Test Accuracy   : {acc:.4f}")
    print("-"*50)

# Select the best performing model
best_model_name = max(accuracies, key=accuracies.get)
final_model = best_models[best_model_name]
final_accuracy = accuracies[best_model_name]

# Save the best model
model_filename = f"{best_model_name}_final_model.joblib"
dump(final_model, model_filename)

# Summary Footer
```

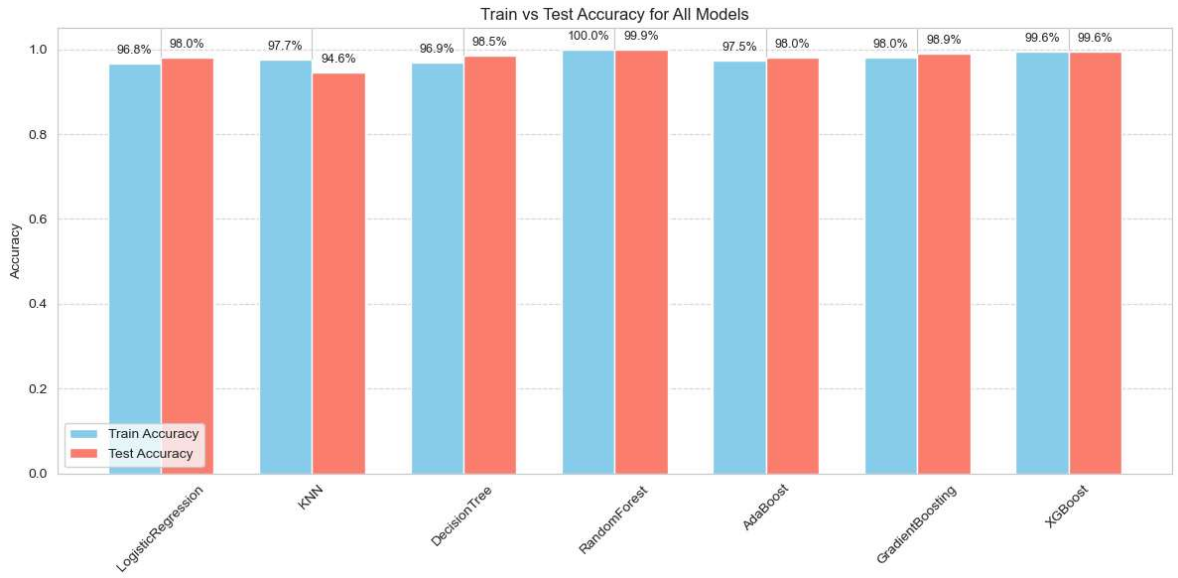
```
print("\n" + "="*50)
print("                 FINAL MODEL SUMMARY              ")
print("="*50)
print(f" Best Model    : {best_model_name}")
print(f" Accuracy      : {final_accuracy:.4f}")
print(f" Saved As      : {model_filename}")
print("="*50)
```

```
================================================
        MODEL TRAINING & EVALUATION
================================================

------------------------------------------------
Model Name      : LogisticRegression
------------------------------------------------
Best Parameters : {'C': 0.1}
Train Accuracy  : 0.9676
Test Accuracy   : 0.9801
------------------------------------------------

------------------------------------------------
Model Name      : KNN
------------------------------------------------
Best Parameters : Not Tuned (Default)
Train Accuracy  : 0.9765
Test Accuracy   : 0.9458
------------------------------------------------

------------------------------------------------
Model Name      : DecisionTree
------------------------------------------------
Best Parameters : {'max_depth': 5}
Train Accuracy  : 0.9694
Test Accuracy   : 0.9848
------------------------------------------------

------------------------------------------------
Model Name      : RandomForest
------------------------------------------------
Best Parameters : Not Tuned (Default)
Train Accuracy  : 1.0000
Test Accuracy   : 0.9994
------------------------------------------------

------------------------------------------------
Model Name      : AdaBoost
------------------------------------------------
Best Parameters : Not Tuned (Default)
Train Accuracy  : 0.9745
Test Accuracy   : 0.9802
------------------------------------------------

------------------------------------------------
Model Name      : GradientBoosting
------------------------------------------------
Best Parameters : Not Tuned (Default)
Train Accuracy  : 0.9801
Test Accuracy   : 0.9894
------------------------------------------------

------------------------------------------------
Model Name      : XGBoost
------------------------------------------------
Best Parameters : Not Tuned (Default)
Train Accuracy  : 0.9958
Test Accuracy   : 0.9955
------------------------------------------------
```

```
==================================================
                FINAL MODEL SUMMARY
==================================================
 Best Model    : RandomForest
 Accuracy      : 0.9994
 Saved As      : RandomForest_final_model.joblib
==================================================
```

In [24]:

```python
# Plotting train vs test accuracy
model_names = list(accuracies.keys())
train_vals = [train_accuracies[m] for m in model_names]
test_vals = [accuracies[m] for m in model_names]

x = np.arange(len(model_names))
width = 0.35

fig, ax = plt.subplots(figsize=(12, 6))
bars1 = ax.bar(x - width/2, train_vals, width, label='Train Accuracy', color='sk
bars2 = ax.bar(x + width/2, test_vals, width, label='Test Accuracy', color='salm

# Annotate accuracy values on top of bars
for bar in bars1 + bars2:
    height = bar.get_height()
    ax.annotate(f'{height*100:.1f}%',
                xy=(bar.get_x() + bar.get_width() / 2, height),
                xytext=(0, 5),  # Offset text above bar
                textcoords="offset points",
                ha='center', va='bottom', fontsize=9)

# Styling
ax.set_ylabel('Accuracy')
ax.set_title('Train vs Test Accuracy for All Models')
ax.set_xticks(x)
ax.set_xticklabels(model_names, rotation=45)
ax.legend()
ax.grid(axis='y', linestyle='--', alpha=0.7)

plt.tight_layout()
plt.show()
```



Conclusion

In this project, we built and evaluated multiple machine learning models to detect fraudulent credit card transactions. After applying data preprocessing and balancing techniques (SMOTE), we trained and compared various classification models using both default and tuned configurations.

Key Observations:

- **Logistic Regression** achieved a test accuracy of **98.01%**, showing strong generalization on balanced data.
- **K-Nearest Neighbors (KNN)** had a lower test accuracy (**94.58%**) and took longer to train, making it less suitable for this use case.
- **Decision Tree**, **Random Forest**, **AdaBoost**, **Gradient Boosting**, and **XGBoost** performed exceptionally well, with test accuracies all above **98%.**
- **Gradient Boosting** and **XGBoost** provided highly accurate results with minimal overfitting.
- The **Random Forest Classifier** stood out with:
  - **Training Accuracy**: 100%
  - **Test Accuracy**: **99.94%**
  - No tuning required and quick training.

Final Model Selection

We selected the **Random Forest Classifier** as the final model due to its:

- Superior accuracy
- Balanced performance across training and test sets
- Efficiency in training
- Robustness against overfitting

The final model has been saved as: RandomForest_final_model.joblib