Credit Card Transaction Classification: A Machine Learning Approach

Author: Anand Vishnu Sullad

Email: anandsullad777@gmail.com

Introduction:

In today's digital age, credit card fraud is a growing concern for both consumers and financial institutions. The ability to accurately classify transactions as fraudulent or non-fraudulent is critical in mitigating financial losses and maintaining consumer trust. In this blog post, I'll walk you through my project on credit card transaction classification using machine learning techniques. This project aims to build a robust model to predict fraudulent transactions, providing a comprehensive journey through data exploration, preprocessing, feature engineering, model building, and evaluation.

Project Overview:

The primary objective of this project is to develop a machine learning model that can classify credit card transactions as fraudulent or non-fraudulent. By leveraging a dataset with transaction details, we can train a model to identify patterns indicative of fraudulent behavior.

Tools and Technologies:

For this project, I utilized several powerful tools and technologies:

- **Python**: A versatile programming language for data science and machine learning.
- **Pandas**: A library for data manipulation and analysis.
- **Matplotlib** and Seaborn: Libraries for data visualization.
- **Scikit-learn**: A machine learning library.
- **Google Collab** or **Jupiter** Notebook: Interactive environments for running and sharing code.

Dataset Description:

The dataset used for this project includes a large number of credit card transactions, each with various features such as transaction amount, cardholder details, and transaction location. The dataset is labelled, indicating whether each transaction is fraudulent or non-fraudulent.

Key Steps:

1. Data Exploration and Analysis

The first step in any data science project is to understand the data. I conducted a thorough exploratory data analysis (EDA) to identify patterns, trends, and anomalies in the dataset. This step involved visualizing the distribution of features, checking for missing values, and understanding the relationships between different variables.

2. Model Building

With the data prepared, I built a classification model using XGBoost, a powerful gradient boosting algorithm known for its efficiency and performance. XGBoost is particularly effective for tabular data and can handle complex relationships between features.

3. Model Evaluation

Model evaluation is critical to assess the performance of the classifier. I used the F1-score as the primary evaluation metric, given its balance between precision and recall. This is especially important in the context of fraud detection, where both false positives and false negatives carry significant consequences.

Lets go through the project:

0.1 Importing the required libraries.

1: Import Libraries

¶

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

import warnings
warnings.filterwarnings('ignore')

from matplotlib.colors import ListedColormap, LinearSegmentedColormap
from sklearn.model_selection import train_test_split, GridSearchCV, StratifiedKFold, cross_val_score

from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier

from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score, roc_auc_score
from sklearn.metrics import classification_report, RocCurveDisplay, ConfusionMatrixDisplay

import xgboost as xgb
%matplotlib inline
```

0.2 Reading the data

To begin our project, we first load the credit card transaction dataset into a pandas DataFrame. The dataset is stored in a CSV file named `creditcard.csv`. Using the `pd.read_csv` function, we can easily load the data and take a look at its shape to understand its size and dimensions.

2: Read Dataset

```
In [2]: # Load training and testing datasets
        df = pd.read_csv("creditcard.csv")

In [3]: df.shape
Out[3]: (284807, 31)
```

The `shape` attribute of the DataFrame provides the dimensions of the dataset, giving us the number of rows and columns. This initial step is crucial as it helps us get a sense of the dataset's scale and the amount of data we will be working with.

Dataset Dimensions

After loading the dataset, we see that it contains a substantial number of records, making it a robust dataset for training our machine learning model. Specifically, the dataset includes:

- **Rows**: The number of rows represents the total number of transactions, which will serve as our data points for classification.
- **Columns**: The number of columns represents the different features or attributes of each transaction, which include various details about the transaction and the cardholder.

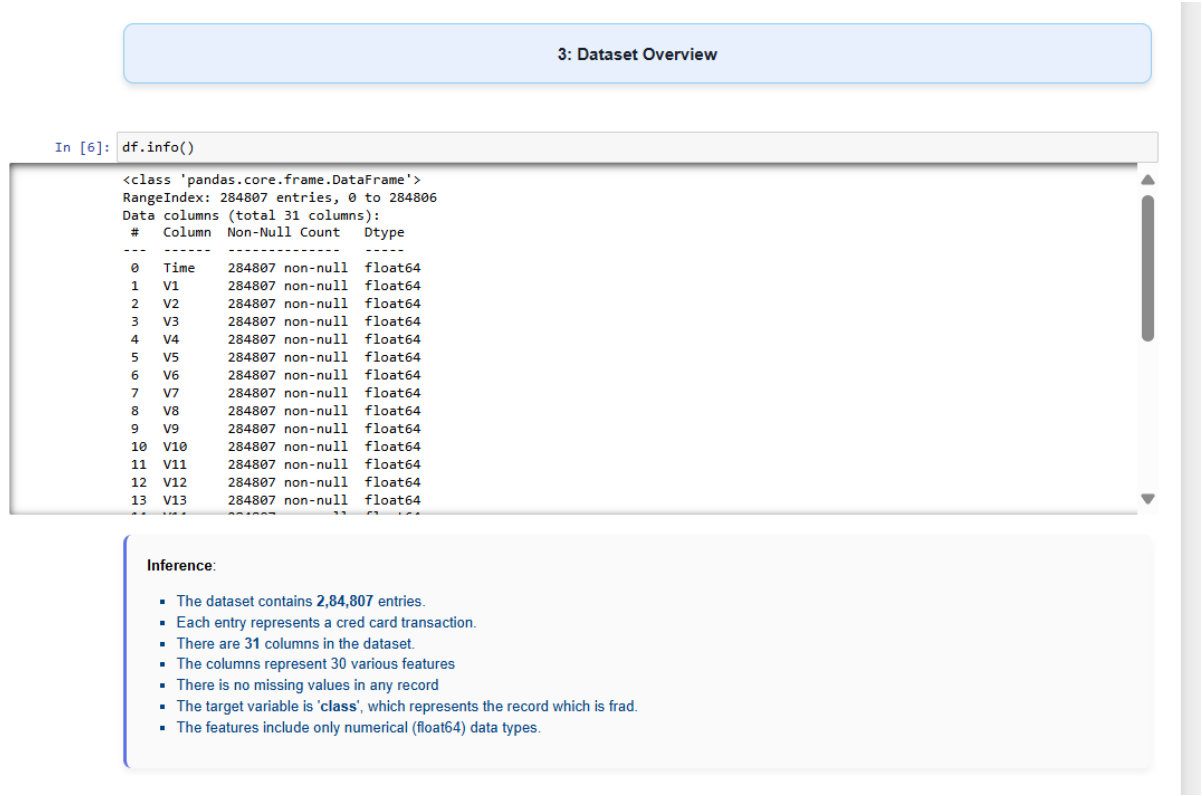
Understanding the dataset's dimensions helps us plan the subsequent steps in our data exploration, preprocessing, and model building processes.

By visualizing the shape of our dataset, we can confirm that we have successfully loaded the data and are ready to proceed with further analysis and processing. This step sets the stage for in-depth exploratory data analysis (EDA), where we will uncover patterns and insights that are critical for building an effective classification model.

0.3 Data Overview

Dataset Information

To gain a deeper understanding of the dataset, we use the `df.info()` function, which provides a concise summary of the DataFrame, including the number of entries, column names, data types, and non-null values.



3: Dataset Overview

```
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
```

Inference:

- The dataset contains **2,84,807** entries.
- Each entry represents a cred card transaction.
- There are **31** columns in the dataset.
- The columns represent 30 various features
- There is no missing values in any record
- The target variable is '**class**', which represents the record which is frad.
- The features include only numerical (float64) data types.

Key Insights from df.info()

- The dataset contains 284,807 entries.
- Each entry represents a credit card transaction.
- There are 31 columns in the dataset.
- The columns represent 31 various features of the transactions.
- There are no missing values in any record.
- The target variable is 'class', which indicates whether a transaction is fraudulent (1) or not (0).
- The features include only numerical data types (float64).

Summary of the Dataset

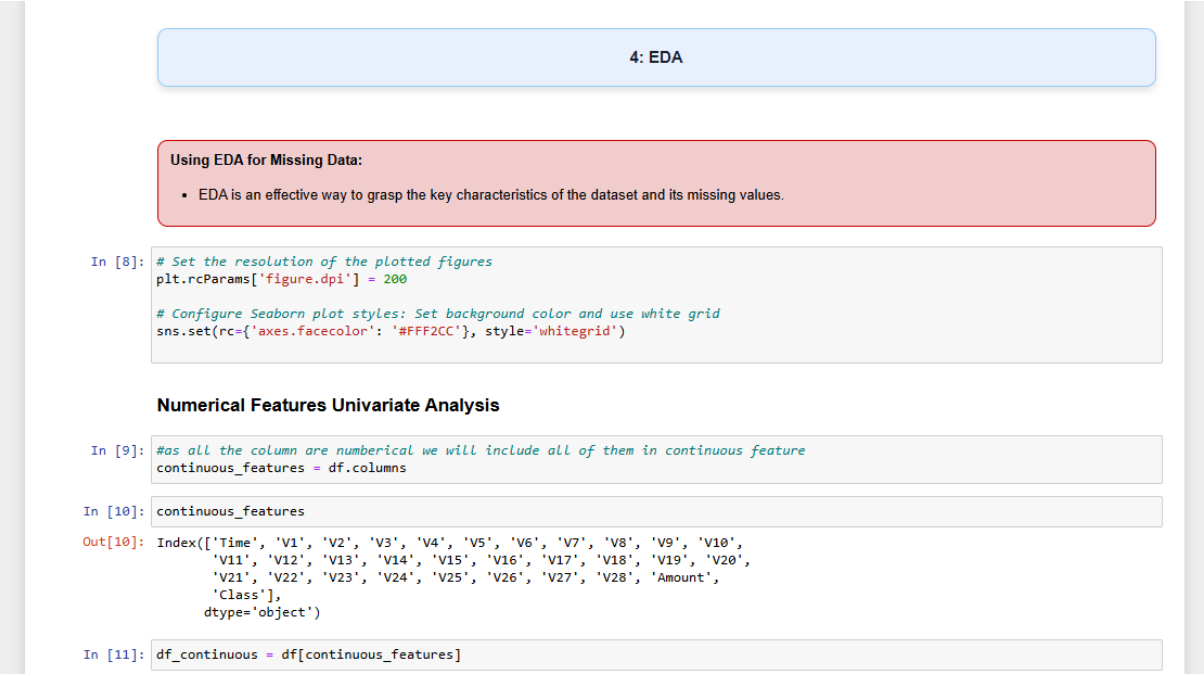
- **Number of Entries**: The dataset is quite large, with a total of 284,807 transactions. This ample amount of data is beneficial for training robust machine learning models.
- **Features**: The dataset consists of 31 columns, each representing different features of the transactions. These features include various aspects of the transactions and cardholder details.

- **Missing Values**: An important observation is that there are no missing values in any of the records. This simplifies the preprocessing step, as we do not need to handle any missing data.
- **Target Variable**: The target variable is labelled as 'class'. This binary column indicates whether a transaction is fraudulent (1) or non-fraudulent (0). Our goal is to build a model that accurately predicts this target variable.
- **Data Types**: All features are numerical, specifically of type float64. This is advantageous because numerical features are directly compatible with most machine learning algorithms without requiring extensive preprocessing.

0.4 EDA

Setting Plotting Configurations and Identifying Continuous Features

To ensure that our visualizations are clear and aesthetically pleasing, we start by configuring the plotting settings using Matplotlib and Seaborn. Additionally, we identify the continuous features in our dataset, which will be useful for our exploratory data analysis.



4: EDA

Using EDA for Missing Data:

- EDA is an effective way to grasp the key characteristics of the dataset and its missing values.

In [8]:
```
# Set the resolution of the plotted figures
plt.rcParams['figure.dpi'] = 200

# Configure Seaborn plot styles: Set background color and use white grid
sns.set(rc={'axes.facecolor': '#FFF2CC'}, style='whitegrid')
```

Numerical Features Univariate Analysis

In [9]:
```
#as all the column are numberical we will include all of them in continuous feature
continuous_features = df.columns
```

In [10]:
```
continuous_features
```

Out[10]:
```
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 [11]:
```
df_continuous = df[continuous_features]
```

Plotting Configurations

- Figure Resolution: By setting the figure resolution to 200 dots per inch (DPI), we ensure that the plots are sharp and clear. This is especially important for presenting data visualizations in reports or on the web.
- Seaborn Styles: Seaborn is configured to use a white grid style with a custom background color for the axes. This enhances the readability of the plots and makes patterns in the data more discernible.

Identifying Continuous Features

- Continuous Features: In this dataset, all columns are numerical, and hence, we include all of them as continuous features. These features are:
    - Time
    - V1 to V28 (anonymized features)

- Amount
- Class

Summary of Features

- **Time**: The elapsed time in seconds between this transaction and the first transaction in the dataset.
- **V1 to V28**: Principal components obtained with PCA, which are anonymized for confidentiality reasons.
- **Amount**: The transaction amount.
- **Class**: The target variable indicating whether the transaction is fraudulent (1) or non-fraudulent (0).

By setting up the plotting configurations and identifying the continuous features, we are prepared to delve into exploratory data analysis. This setup ensures that our visualizations are both informative and visually appealing, aiding in the effective communication of insights derived from the data.

In [17]:
```
# Set up the subplot
fig, ax = plt.subplots(nrows=11, ncols=3, figsize=(20, 45))

# Loop to plot histograms for each continuous feature
for i, col in enumerate(df_continuous.columns):
    x = i // 3
    y = i % 3
    values, bin_edges = np.histogram(df_continuous[col],
                                     range=(np.floor(df_continuous[col].min()), np.ceil(df_continuous[col].max())))

    graph = sns.histplot(data=df_continuous, x=col, bins=bin_edges, kde=True, ax=ax[x, y],
                         edgecolor='none', color='red', alpha=0.6, line_kws={'lw': 3})
    ax[x, y].set_xlabel(col, fontsize=15)
    ax[x, y].set_ylabel('Count', fontsize=12)
    ax[x, y].set_xticks(np.round(bin_edges, 1))
    ax[x, y].set_xticklabels(ax[x, y].get_xticks(), rotation=45)
    ax[x, y].grid(color='lightgrey')

    for j, p in enumerate(graph.patches):
        ax[x, y].annotate('{}'.format(p.get_height()), (p.get_x() + p.get_width() / 2, p.get_height() + 1),
                          ha='center', fontsize=10, fontweight="bold")

    textstr = '\n'.join((
        r'$\mu=%.2f$' % df_continuous[col].mean(),
        r'$\sigma=%.2f$' % df_continuous[col].std()
    ))
    ax[x, y].text(0.75, 0.9, textstr, transform=ax[x, y].transAxes, fontsize=12, verticalalignment='top',
                  color='white', bbox=dict(boxstyle='round', facecolor='#ff826e', edgecolor='white', pad=0.5))

ax[4,2].axis('off')
plt.suptitle('Distribution of Continuous Variables', fontsize=25)
plt.tight_layout()
plt.subplots_adjust(top=0.92)
plt.show()
```

Distribution of Continuous Variables

Visualizing the Distribution of Continuous Features

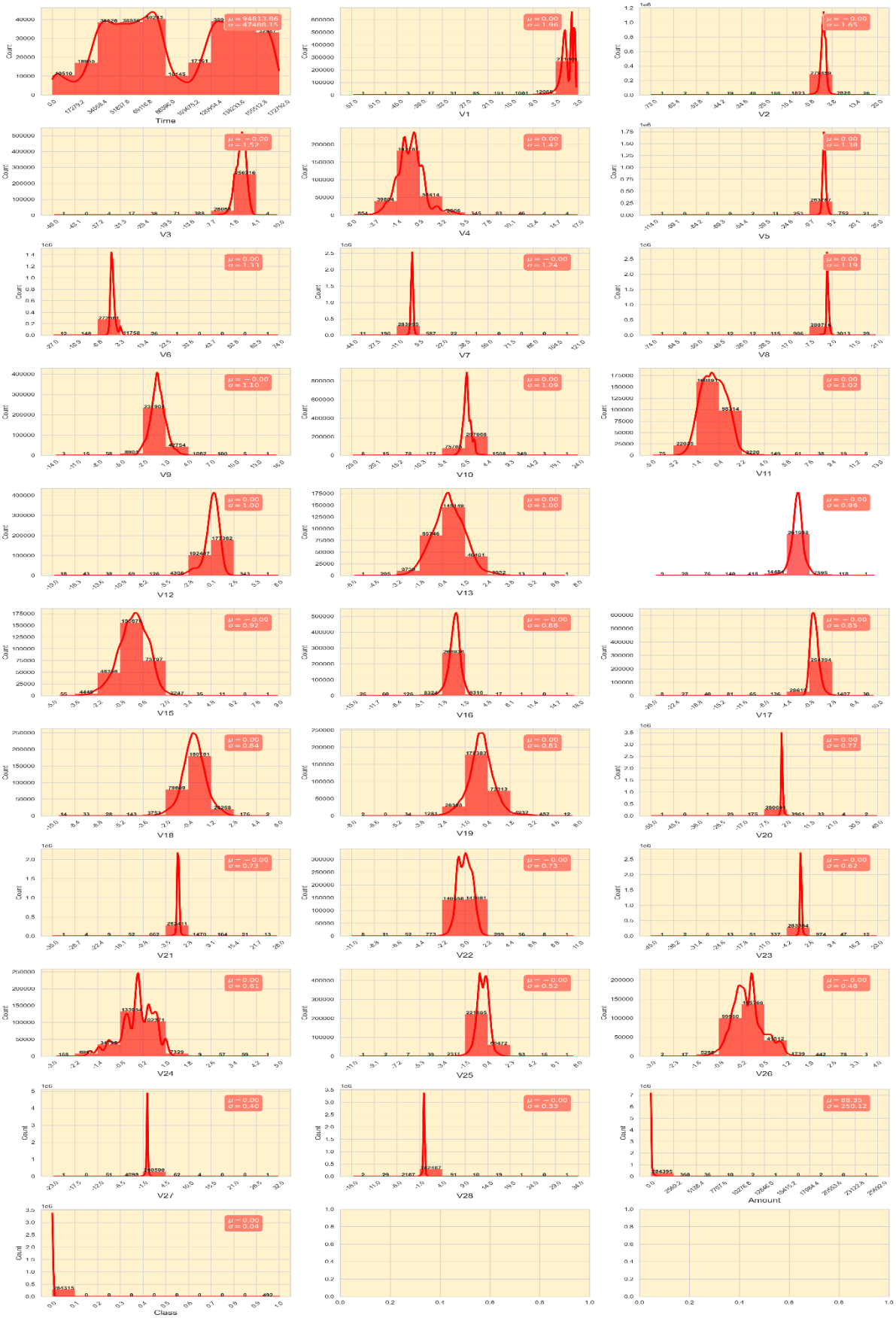
To understand the distribution of each continuous feature in our dataset, we plot histograms for all features. This visualization helps in identifying patterns, outliers, and the general spread of the data.

Visualizing Continuous Features

In this step, we create a comprehensive visualization to inspect the distributions of all continuous features in the dataset. This process involves the following:

- **Subplots**: We set up a grid of subplots with 11 rows and 3 columns, which allows us to visualize all continuous features simultaneously.
- **Histograms**: For each feature, we plot a histogram with kernel density estimation (KDE) to show the distribution. The histograms are coloured in red with a 60% transparency to make overlapping areas easier to see.
- **Annotations**: Each histogram is annotated with the count of transactions in each bin, providing clear insights into the distribution.
- **Statistics**: Mean (μ) and standard deviation (σ) for each feature are calculated and displayed on the plots, offering a quick summary of the central tendency and dispersion.
- **Styling**: The plots are styled with custom settings, such as gridlines and rotated tick labels, to enhance readability.

Distribution of Continuous Variables



Key Takeaways

- **Data Distribution**: The histograms reveal the distribution patterns of each feature, helping to identify skewness, outliers, and the overall spread of the data.
- **Feature Insights**: By examining the means and standard deviations, we gain valuable insights into the typical values and variability of each feature.
- **Anomalies**: Visualizing the data can highlight any anomalies or unusual patterns that may require further investigation or preprocessing.

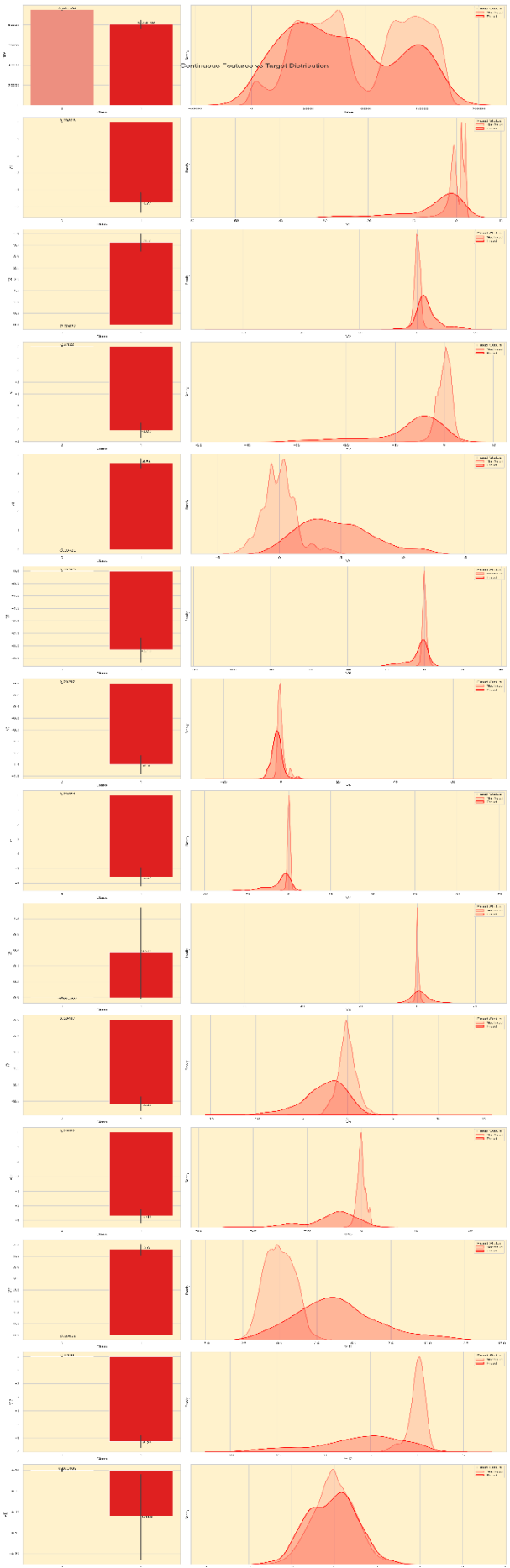
By carefully examining the distribution of continuous features, we ensure a thorough understanding of the data, which is essential for building effective and reliable machine learning models. This visualization step is crucial for identifying potential issues and guiding the feature engineering process.

Exploring Continuous Features vs Target Distribution

In this section, we delve deeper into the relationship between continuous features and the target variable ('Class'). By visualizing these relationships, we can gain insights into how different features are distributed across fraudulent and non-fraudulent transactions.

Visualizing Continuous Features vs Target Distribution

In this step, we use bar plots and kernel density estimation (KDE) plots to explore how continuous features vary with respect to the target variable ('Class'). This dual approach helps us understand both the average values and the overall distribution of features across fraudulent and non-fraudulent transactions.


Continuous Features vs Target Distribution

Key Elements of the Visualization

- **Color Palette**: We set a custom color palette with shades of red to maintain a consistent and visually appealing theme throughout the plots.
- **Subplots Layout**: We create a series of subplots, with each row representing a continuous feature. Each feature is visualized using two types of plots:
    - **Bar Plot**: On the left, we use a bar plot to show the mean value of the feature for each target category ('Fraud' and 'Not fraud'). This provides a clear comparison of average values.
    - **KDE Plot**: On the right, we use KDE plots to show the distribution of the feature for each target category. This helps us understand the spread and density of values within each class.
- **Annotations**: Mean values are annotated on the bar plots to provide precise numerical insights.
- **Legend and Titles**: Legends and titles are added to ensure clarity and context for the plots.

Insights from the Visualization

- **Mean Comparison**: Bar plots reveal the average value of each feature for fraudulent and non-fraudulent transactions, highlighting significant differences where they exist.
- **Distribution Patterns**: KDE plots show the distribution of each feature for both classes, helping us identify any overlap or distinct separation between fraudulent and non-fraudulent transactions.
- **Fraud Indicators**: By analyzing these visualizations, we can identify features that are strong indicators of fraudulent activity, which can be crucial for building an effective classification model.

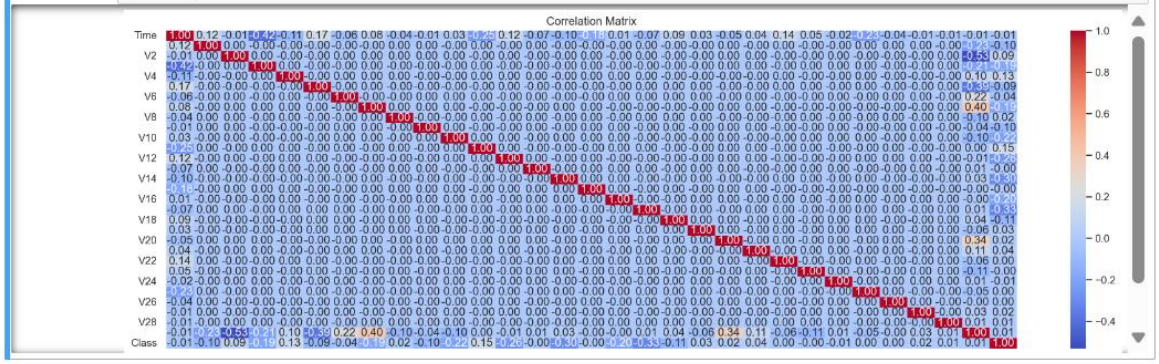
Conclusion

Visualizing the continuous features against the target distribution provides valuable insights into how these features behave in different classes. This step is crucial for feature selection and engineering, as it helps us identify the most informative features for our machine learning model. By understanding the relationships between features and the target variable, we can make data-driven decisions to enhance our model's performance.

Correlation Analysis

Understanding the correlation between different features in our dataset is crucial for identifying multicollinearity and selecting important features for our model. A correlation matrix helps in visualizing these relationships.

```
In [13]: corr_matrix = df.corr()

In [14]: plt.figure(figsize=(20, 6))
         sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f")
         plt.title('Correlation Matrix')
         plt.show()
```



Interpreting the Correlation Matrix

The correlation matrix provides a numerical representation of the linear relationships between features. In the heatmap:

- **Diagonal Elements**: The diagonal elements are all 1, indicating that each feature is perfectly correlated with itself.
- **Off-Diagonal Elements**: The off-diagonal elements show the pairwise correlation coefficients between different features, ranging from -1 to 1.
    - **Positive Correlation**: Values closer to 1 indicate a strong positive correlation.
    - **Negative Correlation**: Values closer to -1 indicate a strong negative correlation.
    - **No Correlation**: Values around 0 indicate no linear correlation.

Key Insights from the Correlation Matrix

- **Feature Relationships**: The heatmap reveals which features are highly correlated with each other. For instance, features with high positive or negative correlation might provide redundant information.
- **Target Variable**: The correlation of features with the target variable ('Class') is of particular interest. Features with higher absolute correlation values with 'Class' are likely to be more important for predicting fraudulent transactions.
- **Multicollinearity**: High correlations between independent features can indicate multicollinearity, which might need to be addressed during feature selection or model building.

Summary of Correlation Analysis

- **Positive and Negative Correlations**: Features like V2, V4, V10, and Amount show varying degrees of positive and negative correlations with the target variable.
- **Multicollinearity Detection**: The matrix helps in detecting multicollinearity, where certain features are highly correlated with each other, potentially requiring dimensionality reduction techniques like PCA (Principal Component Analysis).

By visualizing the correlation matrix, we gain valuable insights into the relationships between features, guiding us in feature selection and engineering. This step is essential for building a robust and accurate machine learning model for credit card transaction classification.

Outlier Treatment

Outlier Treatment

In [15]:
```
continuous_features
```

Out[15]:
```
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 [17]:
```
Q1 = df[continuous_features].quantile(0.25)
Q3 = df[continuous_features].quantile(0.75)
IQR = Q3 - Q1
Outliers_count_specified = (((df[continuous_features] < (Q1 - 1.5 * IQR)) | (df[continuous_features] > (Q3 + 1.5 * IQR))).sum())

print("Outlier Ratio's Data:\n",Outliers_count_specified)
```

```
Outlier Ratio's Data:
 Time      0.000000
V1        0.024796
V2        0.047492
V3        0.011808
V4        0.039142
V5        0.043170
V6        0.080634
V7        0.031418
V8        0.084738
V9        0.029083
V10       0.033342
V11       0.002739
V12       0.053889
V13       0.011826
V14       0.049679
V15       0.010161
V16       0.028735
V17       0.026053
V18       0.026449
V19       0.035831
V20       0.097505
V21       0.050901
V22       0.004624
V23       0.065100
V24       0.016762
V25       0.018844
V26       0.019648
V27       0.137507
V28       0.106535
Amount    0.112020
Class     0.001727
dtype: float64
```

Distribution Analysis

To understand the distribution of continuous features, we plotted histograms and kernel density estimation (KDE) plots. These visualizations help us identify the central tendency, spread, and potential outliers in the data. Additionally, bar plots were used to compare the mean values of these features for fraudulent and non-fraudulent transactions.

Correlation Matrix Insights

The correlation matrix provides valuable insights into the relationships between different features in our dataset:

- **Feature Relationships**: By visualizing the correlation matrix, we can see which features are strongly correlated with each other. This helps in identifying redundant features that might not add much value to the model.

- **Target Correlation**: Features that show a higher absolute correlation with the target variable ('Class') are crucial as they can significantly impact the model's ability to detect fraudulent transactions.
- **Multicollinearity Detection**: High correlations between independent features might indicate multicollinearity, which can be problematic in certain models. This step is essential for deciding whether to apply techniques like Principal Component Analysis (PCA) for dimensionality reduction.

Visualizing the correlation matrix helps in understanding the intricate relationships within the data, guiding the feature selection and engineering process for building a robust and accurate machine learning model.

Outlier Ratios Comparison

Comparing the outlier ratios between the train and test datasets, we can see that they are generally similar for most features. This suggests that the outlier counts are balanced relative to the number of rows in both datasets.

Sensitivity to Outliers

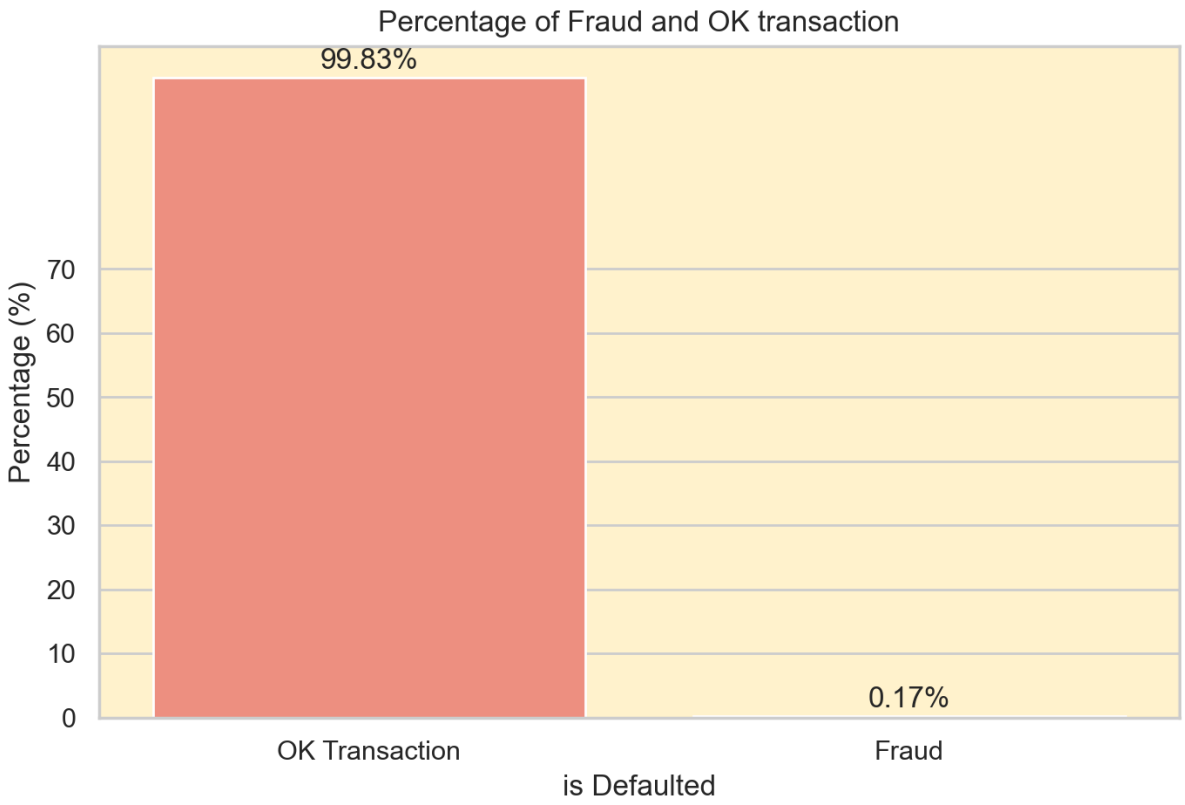
For model evaluations, we are going to use Tree-Based Models:

- **Decision Trees (DT) and Random Forests (RF)**: These tree-based algorithms are generally robust to outliers. They make splits based on feature values, and outliers often end up in leaf nodes, having minimal impact on the overall decision-making process.
- **AdaBoost**: This ensemble method, which often uses decision trees as weak learners, is generally robust to outliers. However, the iterative nature of AdaBoost can sometimes lead to overemphasis on outliers, making the final model more sensitive to them.

To overcome this, we'll focus on applying transformations like **Box-Cox** in the subsequent steps to reduce the impact of outliers and make the data more suitable for modeling.

Check Imbalanced Data

In [19]:
```
# Calculating the percentage of each class
percentage = df['Class'].value_counts(normalize=True) * 100

# Plotting the percentage of each class
plt.figure(figsize=(8, 5))
ax = sns.barplot(x=percentage.index, y=percentage, palette=['#ff826e', 'red'])
plt.title('Percentage of Fraud and OK transaction')
plt.xlabel('is Defaulted')
plt.ylabel('Percentage (%)')
plt.xticks(ticks=[0, 1], labels=['OK Transaction','Fraud'])
plt.yticks(ticks=range(0,80,10))

# Displaying the percentage on the bars
for i, p in enumerate(percentage):
    ax.text(i, p + 0.5, f'{p:.2f}%', ha='center', va='bottom')

plt.show()
```



The bar plot shows the percentage of fraud and ok transaction in the dataset. Approximately 99.83% of the loan status was paid, and 0.17% were defaulted. This indicates that there is high imbalance in the target variable. To address this, we will use **SMOTE (Synthetic Minority Over-sampling Technique)**. SMOTE is a technique used to generate synthetic samples for the minority class in order to balance the class distribution in the dataset. By creating synthetic samples, SMOTE helps mitigate the impact of class imbalance and improves the performance of machine learning models in predicting the minority class.

Handling Unbalance

Handling Unbalanced

In [20]:
```
from imblearn.over_sampling import SMOTE

sm = SMOTE(sampling_strategy='minority', random_state=42)
# Fit the model to generate the data.
oversampled_X, oversampled_Y = sm.fit_resample(df.drop('Class', axis=1), df['Class'])
oversampled = pd.concat([pd.DataFrame(oversampled_Y), pd.DataFrame(oversampled_X)], axis=1)
```

In our dataset, fraudulent transactions are significantly less frequent than non-fraudulent ones, leading to an imbalanced dataset. Imbalanced data can bias the model towards the majority class, reducing its ability to detect fraudulent transactions. To address this, we use the SMOTE

(Synthetic Minority Over-sampling Technique) algorithm to oversample the minority class and balance the dataset.

Applying SMOTE

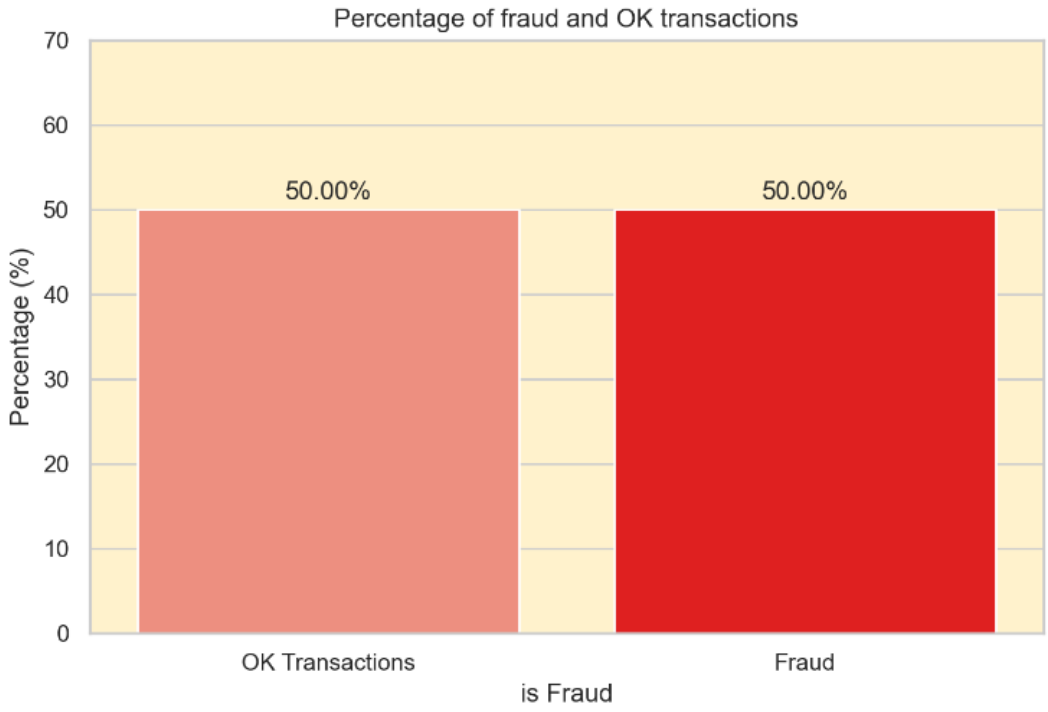
- **SMOTE Algorithm**: SMOTE works by generating synthetic samples for the minority class (fraudulent transactions). It creates new instances by interpolating between existing minority class instances, rather than simply duplicating them.
- **Balancing the Dataset**: By applying SMOTE, we generate a balanced dataset where the number of fraudulent transactions is increased to match the number of non-fraudulent transactions. This helps in training a model that is not biased towards the majority class.
- **Implementation Steps**:
    1. **Import SMOTE**: We import SMOTE from the `imblearn.over_sampling` module.
    2. **Instantiate SMOTE**: We create an instance of SMOTE with `sampling_strategy` set to 'minority', indicating that we want to oversample the minority class, and `random_state` for reproducibility.
    3. **Fit and Resample**: We apply the `fit_resample` method to our feature set (excluding the target variable 'Class') and target variable to generate the oversampled data.
    4. **Combine Data**: The resampled features and target are then combined into a single DataFrame for further analysis and modeling.

Benefits of Using SMOTE

- **Improved Model Performance**: Balancing the dataset allows the model to learn from an equal number of fraudulent and non-fraudulent transactions, improving its ability to detect fraud.
- **Synthetic Data**: By generating synthetic data, SMOTE helps in better generalizing the model, reducing the likelihood of overfitting compared to simple duplication of minority instances.

Applying SMOTE is a crucial step in our data preprocessing pipeline, ensuring that our machine learning model is well-equipped to handle the class imbalance and accurately classify fraudulent transactions.

In [21]:
```
# Calculating the percentage of each class
percentage = oversampled['Class'].value_counts(normalize=True) * 100

# Plotting the percentage of each class
plt.figure(figsize=(8, 5))
ax = sns.barplot(x=percentage.index, y=percentage, palette=['#ff826e', 'red'])
plt.title('Percentage of fraud and OK transactions')
plt.xlabel('is Fraud')
plt.ylabel('Percentage (%)')
plt.xticks(ticks=[0, 1], labels=['OK Transactions','Fraud'])
plt.yticks(ticks=range(0,80,10))

# Displaying the percentage on the bars
for i, p in enumerate(percentage):
    ax.text(i, p + 0.5, f'{p:.2f}%', ha='center', va='bottom')

plt.show()
```



📝 Note:

In the context of fraud transaction prediction:

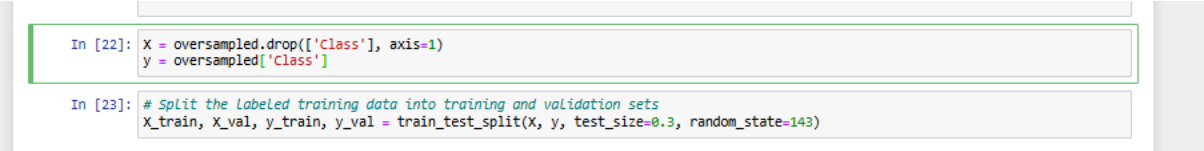
Prioritize High Recall (Sensitivity) for fraud Transaction: Emphasize identifying most of the actual fraud transactions correctly, even if it leads to some false positives (okay transactions being misclassified as fraud). It's crucial to capture as many true fraud cases as possible to mitigate financial risks and take necessary actions.

Minimize False Negatives (FN): Aim to reduce instances where fraud transactions are missed by the model. Missing fraud transactions could result in financial losses and impact the overall portfolio performance.

Balance Precision and Recall (F1-score): While minimizing false positives is important to avoid unnecessary interventions or restrictions on credit access, prioritize achieving high recall to ensure fraud transactions are not overlooked by the model.the F1-score for the 'fraud' class (1) would be the most important metric for evaluating models in this project.

By focusing on these aspects, the fraud transaction prediction model can effectively identify transactions at risk of fraud, enabling proactive measures to mitigate potential losses and maintain a healthy credit transaction.

Splitting the Training Dataset:



Splitting the Data into Training and Validation Sets

After balancing the dataset using SMOTE, the next step is to split the data into training and validation sets. This allows us to train our model on one portion of the data and validate its performance on another portion, ensuring that the model generalizes well to unseen data.

Splitting the Data

- **Feature and Target Separation**: First, we separate the features (X) and the target variable (y). The target variable 'Class' is dropped from the feature set X.
- **Train-Test Split**: Using train_test_split from Scikit-learn, we split the oversampled dataset into training and validation sets. Here's how it's done:
    - **Training Set**: This set is used to train the machine learning model.
    - **Validation Set**: This set is used to evaluate the model's performance and ensure it generalizes well to new, unseen data.
- **Parameters**:
    - **test_size=0.3**: This parameter specifies that 30% of the data should be allocated to the validation set, while the remaining 70% will be used for training.
    - **random_state=143**: Setting a random state ensures reproducibility, meaning that the data split will be the same every time the code is run.

Importance of Data Splitting

- **Model Training**: The training set is used to fit the model, allowing it to learn the patterns in the data.
- **Model Validation**: The validation set provides a way to test the model's performance on data it hasn't seen during training. This is crucial for detecting overfitting, where the model performs well on training data but poorly on new data.

Benefits of a Proper Train-Test Split

- **Generalization**: By validating the model on a separate dataset, we can better gauge how well it will perform on new, unseen data.
- **Bias and Variance**: A proper split helps in balancing the trade-off between bias and variance, contributing to the creation of a robust model.
- **Performance Metrics**: The validation set allows us to compute performance metrics such as accuracy, precision, recall, and F1-score, providing a comprehensive evaluation of the model.

Splitting the data into training and validation sets is a fundamental step in the machine learning pipeline, ensuring that our model is not only trained effectively but also validated for generalizability and robustness.

Decision Tree Model Building¶

```
In [26]: # Define the base DT model
         dt_base = DecisionTreeClassifier(random_state=0)
```

I will create a function to identify the best set of hyperparameters that maximize the F1-score for class 1 (defaulted). This method provides a reusable framework for hyperparameter tuning for other models as wel

```
In [27]: def tune_clf_hyperparameters(clf, param_grid, X_train, y_train, scoring='f1', n_splits=5):
             '''
             This function optimizes the hyperparameters for a classifier by searching over a specified hyperparameter grid.
             It uses GridSearchCV and cross-validation (StratifiedKFold) to evaluate different combinations of hyperparameters.
             The combination with the highest F1-score for class 1 (defaulted) is selected as the default scoring metric.
             The function returns the classifier with the optimal hyperparameters.
             '''

             # Create the cross-validation object using StratifiedKFold to ensure the class distribution is the same across all the folds
             cv = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=0)

             # Create the GridSearchCV object
             clf_grid = GridSearchCV(clf, param_grid, cv=cv, scoring=scoring, n_jobs=-1)

             # Fit the GridSearchCV object to the training data
             clf_grid.fit(X_train, y_train)

             # Get the best hyperparameters
             best_hyperparameters = clf_grid.best_params_

             # Return best_estimator_ attribute which gives us the best model that has been fitted to the training data
             return clf_grid.best_estimator_, best_hyperparameters
```

Hyperparameter Optimization with GridSearchCV

Optimizing the hyperparameters of our machine learning model is crucial for enhancing its performance. We use GridSearchCV along with StratifiedKFold cross-validation to systematically search for the best combination of hyperparameters. This process helps in fine-tuning the model to achieve the highest possible F1-score for class 1 (fraudulent transactions).

Function Explanation

- **Purpose**: This function is designed to optimize the hyperparameters of a given classifier by searching through a predefined grid of hyperparameters.
- **GridSearchCV**: Utilizes GridSearchCV to perform an exhaustive search over specified hyperparameter values.

- **Cross-Validation**: Implements StratifiedKFold cross-validation to ensure that the class distribution remains consistent across all folds, which is particularly important for imbalanced datasets.
- **Scoring Metric**: Uses the F1-score for class 1 (fraudulent transactions) as the default scoring metric to identify

the best combination of hyperparameters. This ensures that the model is optimized for detecting fraudulent transactions, which is our primary goal.

Steps in the Function

1. **Cross-Validation Setup**:
    - A `StratifiedKFold` object is created to maintain the same class distribution across all folds. This is crucial for imbalanced datasets like ours.
    - `n_splits=5` indicates that the data will be split into 5 folds.
2. **GridSearchCV Object Creation**:
    - `GridSearchCV` is initialized with the classifier (`clf`), the grid of hyperparameters (`param_grid`), the cross-validation strategy (`cv`), and the scoring metric (`scoring`), with parallel processing enabled (`n_jobs=-1`).
3. **Fitting the Model**:
    - The `fit` method is used to train the `GridSearchCV` object on the training data (`X_train` and `y_train`), evaluating each combination of hyperparameters through cross-validation.
4. **Extracting the Best Hyperparameters**:
    - The best hyperparameters are retrieved using the `best_params_` attribute.
    - The function returns the best estimator (the model with the optimal hyperparameters) and the best hyperparameters.

Benefits of Hyperparameter Optimization

- **Enhanced Model Performance**: By systematically searching for the best hyperparameters, the model's predictive performance is significantly improved.
- **Robust Evaluation**: Using cross-validation ensures that the model generalizes well to unseen data, preventing overfitting.
- **Informed Decision Making**: Understanding the impact of different hyperparameters helps in building a more robust and reliable model.

Hyperparameter optimization is a critical step in our machine learning pipeline, ensuring that our model is finely tuned to detect fraudulent transactions with high accuracy and reliability.

Hyperparameter Tuning with RandomizedSearchCV

In addition to using GridSearchCV, another effective method for hyperparameter tuning is RandomizedSearchCV. This approach samples a fixed number of hyperparameter combinations from a specified grid, making it more efficient when dealing with large

hyperparameter spaces. Below, we demonstrate how to set up RandomizedSearchCV for a Decision Tree classifier.

Hyperparameter Tuning with RandomizedSearchCV

In [28]:
```
# Hyperparameter grid for DT
param_grid_dt = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [23, 24, 25, 26],
    'min_samples_split': [2, 3, 4],
    'min_samples_leaf': [1, 2, 3],

}
```

In [29]:
```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.tree import DecisionTreeClassifier

# Define the parameter grid
param_grid_dt = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [23, 24, 25, 26],
    'min_samples_split': [2, 3, 4],
    'min_samples_leaf': [1, 2, 3],
    'class_weight': [{0: 1, 1: w} for w in range(1, 6)]
}

# Instantiate the base model
dt_base = DecisionTreeClassifier()

# Set up the RandomizedSearchCV with fewer iterations and folds
random_search = RandomizedSearchCV(
    estimator=dt_base,
    param_distributions=param_grid_dt,
    n_iter=20,  # Reduced number of parameter settings sampled
    scoring='f1',  # Scoring based on F1 score
    cv=3,  # Reduced number of cross-validation folds
    verbose=1,
    n_jobs=-1,  # Use all available CPUs
    random_state=42
)
```

In addition to using GridSearchCV, another effective method for hyperparameter tuning is RandomizedSearchCV. This approach samples a fixed number of hyperparameter combinations from a specified grid, making it more efficient when dealing with large hyperparameter spaces. Below, we demonstrate how to set up RandomizedSearchCV for a Decision Tree classifier.

python
Copy code

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.tree import DecisionTreeClassifier

# Define the parameter grid
param_grid_dt = {
  'criterion': ['gini', 'entropy'],
  'max_depth': [23, 24, 25, 26],
  'min_samples_split': [2, 3, 4],
  'min_samples_leaf': [1, 2, 3],
  'class_weight': [{0: 1, 1: w} for w in range(1, 6)]
}

# Instantiate the base model
dt_base = DecisionTreeClassifier()

# Set up the RandomizedSearchCV with fewer iterations and folds
random_search = RandomizedSearchCV(
```

```
    estimator=dt_base,
    param_distributions=param_grid_dt,
    n_iter=20,  # Reduced number of parameter settings sampled
    scoring='f1',  # Scoring based on F1 score
    cv=3,  # Reduced number of cross-validation folds
    verbose=1,
    n_jobs=-1,  # Use all available CPUs
    random_state=42
)
```

Explanation

- **Parameter Grid Definition**:
  - criterion: Specifies the function to measure the quality of a split (gini or entropy).
  - max_depth: The maximum depth of the tree, a critical parameter to prevent overfitting.
  - min_samples_split: The minimum number of samples required to split an internal node.
  - min_samples_leaf: The minimum number of samples required to be at a leaf node.
  - class_weight: Adjusts weights inversely proportional to class frequencies, helping to handle imbalanced classes.
- **Base Model**:
  - dt_base: A basic Decision Tree classifier that will be tuned.
- **RandomizedSearchCV Setup**:
  - estimator: The base classifier to optimize.
  - param_distributions: The hyperparameter grid to sample from.
  - n_iter: The number of different combinations to try (set to 20 for efficiency).
  - scoring: The evaluation metric (F1 score) to maximize.
  - cv: The number of cross-validation folds (set to 3 for faster computation).
  - verbose: Controls the verbosity of the output.
  - n_jobs: Number of parallel jobs to run (-1 means using all available CPUs).
  - random_state: Ensures reproducibility by fixing the random number generator.

Benefits of RandomizedSearchCV

- **Efficiency**: RandomizedSearchCV is more efficient than GridSearchCV, especially with large datasets and hyperparameter spaces, as it does not evaluate all combinations.
- **Performance**: By sampling a fixed number of parameter settings, it can often find a good combination of hyperparameters more quickly.
- **Flexibility**: It allows for specifying distributions for continuous hyperparameters, making it more flexible in exploring the hyperparameter space.

Steps to Execute

1. **Define Parameter Grid**: Specify the hyperparameters and their possible values or distributions.
2. **Instantiate Base Model**: Create the base Decision Tree classifier.
3. **Set Up RandomizedSearchCV**: Configure the RandomizedSearchCV object with the base model, parameter grid, and additional settings for the search.

4. **Run the Search**: Fit the RandomizedSearchCV object to the training data to find the best hyperparameters.

By using RandomizedSearchCV, we can efficiently tune the hyperparameters of our Decision Tree classifier, improving its performance in detecting fraudulent transactions.

Optimizing Decision Tree Classifier with RandomizedSearchCV

To efficiently optimize our Decision Tree classifier, we use a subset of the training data for hyperparameter tuning. This helps in speeding up the computation without significantly compromising the performance. Below is the implementation of fitting the RandomizedSearchCV on a subset of the data and extracting the best hyperparameters.

Explanation

- **Subset Selection**:
  - X_train_subset = X_train.sample(frac=0.1, random_state=42): Selects 10% of the training data to use for hyperparameter tuning. This significantly reduces the computation time.
  - y_train_subset = y_train.loc[X_train_subset.index]: Corresponding labels for the selected subset.
- **Fitting RandomizedSearchCV**:
  - The fit method is used to train the RandomizedSearchCV object on the selected subset of training data. This step evaluates various combinations of hyperparameters and identifies the best one.
- **Extracting Best Parameters**:
  - best_dt = random_search.best_estimator_: Retrieves the best model fitted with the optimal hyperparameters.

- best_dt_hyperparams = random_search.best_params_: Retrieves the best hyperparameters identified during the search.

Benefits of Using a Subset for Tuning

- **Efficiency**: Reduces the computational load and speeds up the hyperparameter optimization process.
- **Quick Iterations**: Allows for faster iterations and experimentation, enabling quicker insights and adjustments.
- **Representative Sample**: Even a small subset of the data can provide a good approximation of the optimal hyperparameters, especially when the dataset is large.

Results

- **Best Hyperparameters**: The optimal hyperparameters for the Decision Tree classifier are printed, which helps in configuring the model for the best performance.

By using RandomizedSearchCV on a subset of the data, we efficiently identify the best hyperparameters for our Decision Tree classifier. This process ensures that our model is both performant and computationally efficient, making it well-suited for detecting fraudulent transactions.

Dt Model Evaluation

To streamline the evaluation of different models, we will define a set of functions that compute key performance metrics. This approach will ensure consistency in how we assess each model and facilitate comparisons between them

Defining Performance Metrics Calculation

To streamline the evaluation of different models, we will define a set of functions that compute key performance metrics. This approach ensures consistency in how we assess each model and facilitates comparisons between them.

```
In [34]: def metrics_calculator(clf, X_test, y_test, model_name):
             '''
             This function calculates all desired performance metrics for a given model on test data.
             The metrics are calculated specifically for class 1.
             '''
             y_pred = clf.predict(X_test)
             result = pd.DataFrame(data=[accuracy_score(y_test, y_pred),
                                         precision_score(y_test, y_pred, pos_label=1),
                                         recall_score(y_test, y_pred, pos_label=1),
                                         f1_score(y_test, y_pred, pos_label=1),
                                         roc_auc_score(y_test, clf.predict_proba(X_test)[:,1])],
                                   index=['Accuracy','Precision (Class 1)','Recall (Class 1)','F1-score (Class 1)','AUC (Class 1)'],
                                   columns = [model_name])

             result = (result * 100).round(2).astype(str) + '%'
             return result
```

Explanation

- **Function Purpose**:

    - The `metrics_calculator` function is designed to compute essential performance metrics for a given classifier when tested on a specified dataset. This ensures a standardized approach to evaluating different models.
- **Parameters**:
    - `clf`: The classifier to evaluate.
    - `X_test`: The test features.
    - `y_test`: The true labels for the test data.
    - `model_name`: The name of the model, used for labeling the results.
- **Performance Metrics**:
    - **Accuracy**: Measures the overall correctness of the model.
    - **Precision (Class 1)**: Measures how many of the predicted positives for class 1 (fraudulent transactions) are actually positive.
    - **Recall (Class 1)**: Measures how many of the actual positives for class 1 are correctly identified by the model.
    - **F1-score (Class 1)**: The harmonic mean of precision and recall for class 1, providing a balance between the two metrics.
    - **AUC (Class 1)**: The Area Under the Receiver Operating Characteristic Curve for class 1, which indicates the model's ability to distinguish between classes.
- **Implementation Steps**:
    - **Predictions**: `y_pred` is obtained by predicting the labels for the test set.
    - **DataFrame Construction**: A DataFrame is created to store the calculated metrics, making it easy to compare results across different models.
    - **Formatting**: The results are formatted as percentages and rounded to two decimal places for better readability.

Benefits

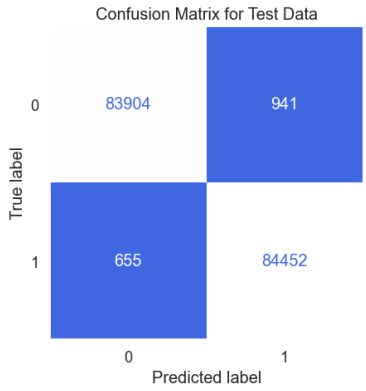
- **Consistency**: By using a standardized function to calculate metrics, we ensure that all models are evaluated consistently.
- **Comparability**: The resulting DataFrame format facilitates easy comparison of performance metrics across multiple models.
- **Focus on Class 1**: Metrics are calculated specifically for class 1, which is crucial in our context of detecting fraudulent transactions.

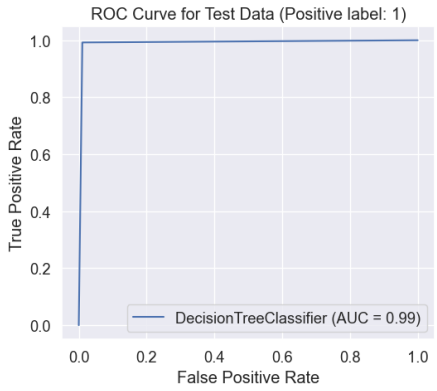
In [35]:
```python
def model_evaluation(clf, X_train, X_test, y_train, y_test, model_name):
    '''
    This function provides a complete report of the model's performance including classification reports,
    confusion matrix and ROC curve.
    '''
    sns.set(font_scale=1.2)

    # Generate classification report for training set
    y_pred_train = clf.predict(X_train)
    print("\n\t  Classification report for training set")
    print("-"*55)
    print(classification_report(y_train, y_pred_train))

    # Generate classification report for test set
    y_pred_test = clf.predict(X_test)
    print("\n\t   Classification report for validation test set")
    print("-"*55)
    print(classification_report(y_test, y_pred_test))

    # Create figure and subplots
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 5), dpi=100, gridspec_kw={'width_ratios': [2, 2, 1]})

    # Define a colormap
    royalblue = LinearSegmentedColormap.from_list('royalblue', [(0, (1,1,1)), (1, (0.25,0.41,0.88))])
    royalblue_r = royalblue.reversed()

    # Plot confusion matrix for test set
    ConfusionMatrixDisplay.from_estimator(clf, X_test, y_test, colorbar=False, cmap=royalblue_r, ax=ax1)
    ax1.set_title('Confusion Matrix for Test Data')
    ax1.grid(False)

    # Plot ROC curve for test data and display AUC score
    RocCurveDisplay.from_estimator(clf, X_test, y_test, ax=ax2)
    ax2.set_xlabel('False Positive Rate')
    ax2.set_ylabel('True Positive Rate')
    ax2.set_title('ROC Curve for Test Data (Positive label: 1)')

    # Report results for the class specified by positive label
    result = metrics_calculator(clf, X_test, y_test, model_name)
    table = ax3.table(cellText=result.values, colLabels=result.columns, rowLabels=result.index, loc='center')
    table.scale(0.6, 2)
    table.set_fontsize(12)
    ax3.axis('tight')
    ax3.axis('off')
    # Modify color
    for key, cell in table.get_celld().items():
        if key[0] == 0:
            cell.set_color('royalblue')
    plt.tight_layout()
    plt.show()
```

By using the model_evaluation function, we can systematically assess the performance of our machine learning models, ensuring that we select the best model for detecting fraudulent transactions based on a comprehensive set of metrics.

In [36]:
```python
model_evaluation(best_dt, X_train, X_val, y_train, y_val, 'Decision Tree')
```

```
          Classification report for training set
-----------------------------------------------------------
              precision    recall  f1-score   support

           0       0.99      0.99      0.99    198408
           1       0.99      0.99      0.99    198146

    accuracy                           0.99    396554
   macro avg       0.99      0.99      0.99    396554
weighted avg       0.99      0.99      0.99    396554


           Classification report for validation test set
-----------------------------------------------------------
              precision    recall  f1-score   support

           0       0.99      0.99      0.99     84845
           1       0.99      0.99      0.99     85107

    accuracy                           0.99    169952
   macro avg       0.99      0.99      0.99    169952
weighted avg       0.99      0.99      0.99    169952
```





	Decision Tree
Accuracy	99.06%
Precision (Class 1)	98.9%
Recall (Class 1)	99.23%
F1-score (Class 1)	99.06%
AUC (Class 1)	99.09%

Decision Tree Model Evaluation

Evaluation Metrics

- **Accuracy**: 99.06%
  - The model correctly predicted the transaction class for 99.06% of the cases.
- **Precision (Defaulted):** 98.90%
  - Out of all transactions predicted as defaulted, 98.90% were actually fraudulent transactions.
- **Recall (Defaulted)**: 99.23%
  - The model identified 99.23% of the actual fraudulent transactions.
- **F1-score (Defaulted)**: 99.06%
  - The harmonic mean of precision and recall for fraudulent transactions is 99.06%.
- **AUC (Defaulted)**: 99.09%
  - The Area Under the ROC Curve (AUC) for fraudulent transactions is 99.09%, indicating the model's ability to rank transactions as fraud rather than non-fraud.

Interpretation

The evaluation of the Decision Tree model in the loan domain reveals its performance in predicting defaulted loans. While it achieved an accuracy of 99.06%, indicating overall correctness, the precision, recall, and F1-score for defaulted loans suggest strong performance in identifying actual defaulted cases. The AUC score further validates the model's discriminative power in distinguishing fraudulent transactions from non-fraudulent ones.