A STUDY ON THE ONLINE PAYMENT FRAUD DETECTION USING MACHINE LEARNING

PROJECT REPORT

SUBMITTED BY

M.SWATHI (2021PITEC161)

V.SHIVA SHRUTHI (2021PITEC158)

BACHELOR OF ENGINEERING

IN

DEPARTMENT OF ELECTONICS AND COMMUNICATION ENGINEERING

ABSTRACT:

The rapid evolution of e-commerce and digital banking has significantly increased the volume of online transactions, making the financial sector a prime target for fraudulent activities. As traditional fraud detection mechanisms struggle to cope with sophisticated and constantly evolving fraud techniques, there is an urgent need for advanced solutions that can adapt and respond to new threats in real-time. This study presents a comprehensive approach to online payment fraud detection using machine learning (ML) algorithms, aimed at enhancing the security of digital transactions while minimizing false positives that can disrupt genuine transactions. Our dataset comprises a large number of transaction records, including both fraudulent and legitimate transactions, with features encompassing transaction details, user behaviour, and account information. We preprocess the data to handle missing values, normalize feature scales, and encode categorical variables, ensuring the ML models receive clean and structured input. Feature engineering plays a crucial role in our approach, as we extract and select the most relevant features that contribute to distinguishing between fraudulent and legitimate transactions. Techniques such as Principal Component Analysis (PCA) and feature importance ranking are utilized to optimize the feature set for better model performance. Our results demonstrate the effectiveness of machine learning algorithms in detecting online payment fraud, with significant improvements in detection rates and reduced false positives compared to traditional rule-based systems. The study highlights the potential of ML as a powerful tool in the fight against online payment fraud, providing a scalable and dynamic solution that can evolve with the changing landscape of digital transactions.

In conclusion, the integration of machine learning into online payment systems offers a promising avenue for improving the security and integrity of digital transactions. By continuously adapting to new and complex fraud strategies, machine learning algorithms can help protect consumers and businesses alike, fostering a safer and more trustworthy digital financial ecosystem.

INTRODUCTION:

Online payment fraud detection is a critical area of cybersecurity that addresses the escalating challenge of fraudulent transactions in digital commerce. With the proliferation of online shopping and financial services, the potential for fraud has significantly increased, posing a substantial risk to both businesses and consumers. Traditional fraud detection methods, which often rely on rule-based systems, have proven to be inadequate in dealing with the sophistication and evolving nature of modern fraud schemes. This inadequacy has prompted the need for more advanced and adaptable solutions.

Machine learning (ML) offers a promising approach to enhancing online payment fraud detection. By leveraging large datasets of transactional information, ML algorithms can learn to identify patterns and anomalies that may indicate fraudulent activity. Unlike rule-based systems, ML models can continuously evolve and adapt to new fraud tactics without the need for manual updates. This dynamic learning process allows for the detection of fraud in real-time, significantly reducing the window of opportunity for fraudsters to exploit.

The introduction of ML into fraud detection involves several key steps, including data collection, preprocessing, feature selection, model training, and evaluation. Data collection encompasses gathering a comprehensive dataset that includes both legitimate and fraudulent transactions. Preprocessing involves cleaning and normalizing the data, while feature selection focuses on identifying the most relevant attributes that contribute to fraud detection. Model training entails using the processed data to train ML algorithms, and evaluation assesses the model's performance in accurately identifying fraudulent transactions.

Implementing ML for online payment fraud detection not only enhances the accuracy and efficiency of fraud identification but also offers a more scalable and flexible solution to combat the ever-evolving landscape of online fraud. By harnessing the power of ML, businesses can better protect themselves and their customers from the financial and reputational damages caused by fraud, ensuring a safer online environment for transactions.

The dataset is collected from Kaggle, which contains historical information about fraudulent transactions which can be used to detect fraud in online payments.

The dataset consists of 10 variables:

- step: represents a unit of time where 1 step equals 1 hour
- type: type of online transaction
- amount: the amount of the transaction
- nameOrig: customer starting the transaction
- oldbalanceOrg: balance before the transaction
- newbalanceOrig: balance after the transaction
- nameDest: recipient of the transaction
- oldbalanceDest: initial balance of recipient before the transaction
- newbalanceDest: the new balance of recipient after the transaction
- isFraud: fraud transaction

OBJECTIVE:

The objective of online payment fraud detection using machine learning is to minimize financial losses and maintain trust in digital payment systems by accurately identifying and preventing fraudulent transactions in real-time. This goal encompasses several specific objectives:

1.Accuracy Enhancement: Improve the precision and recall of fraud detection mechanisms to reduce both false positives (legitimate transactions flagged as fraudulent) and false negatives (fraudulent transactions missed by the system). High accuracy ensures that customers experience minimal disruption in their legitimate transactions while effectively catching fraud.

2.Real-time Detection: Achieve the capability to detect and flag fraudulent transactions as they occur, allowing for immediate action to prevent fraud. This is crucial in mitigating losses and preventing the execution of unauthorized transactions.

3.Adaptability and Scalability: Develop systems that can adapt to emerging fraud tactics and scale with the growth in transaction volumes and complexity. Fraudsters continuously evolve their strategies to bypass detection, so the system must learn from new patterns of fraud to stay effective.

4.Cost Efficiency: Reduce the operational costs associated with manual review and investigation of transactions by automating the fraud detection process. Machine learning can handle vast volumes of transactions more efficiently than human teams, freeing up resources for other critical security tasks.

5.User Experience Improvement: Enhance the overall customer experience by minimizing friction during the transaction process. By reducing false positives, customers face fewer unjustified transaction declines, leading to higher satisfaction and trust in the payment system.

6.Compliance and Regulatory Adherence: Ensure that the fraud detection system complies with relevant financial regulations and data protection laws. This includes maintaining customer privacy, securing transaction data, and adhering to industry standards for fraud prevention.

7.Data-Driven Insights: Utilize the insights gained from analysing transaction data to improve security measures, understand customer behaviour better, and inform strategic decisions. Machine learning models can reveal trends and patterns that might not be evident through traditional analysis methods.

By achieving these objectives, online payment fraud detection systems aim to protect both businesses and consumers from the financial and reputational damage caused by fraudulent activities, ensuring a secure and trustworthy digital commerce environment.

EXISTING SYSTEM:

The existing systems for online payment fraud detection using machine learning are diverse and sophisticated, employing various techniques and algorithms to identify and prevent fraudulent transactions. These systems are integral to the security infrastructure of financial institutions, e-commerce platforms, and payment processors. Here’s an overview of the key components and approaches used in these systems:

1. Data Collection and Preprocessing

2. Machine Learning Models

    - Supervised Learning Models
    - Unsupervised Learning Models
    - Deep Learning

3. Ensemble Techniques

4. Real-time Analysis and Decision Systems

5. Adaptive Learning

6. Integration with Other Systems

PROPOSED SYSTEM:

The proposed system for online payment fraud detection aims to build upon the foundations of existing systems while addressing their limitations and incorporating the latest advancements in machine learning (ML) and data analytics. The goal is to create a more dynamic, efficient, and accurate system that not only detects fraudulent transactions in real-time but also adapts to evolving fraud patterns more effectively. Here's an outline of the key components and innovations of the proposed system:

1. Importing Libraries and Datasets
2. Data Visualization
    - Univariate data visualization
    - Bivariate data visualization
    - Multivariate data visualization
3. Data Preprocessing
    - Encoding of Type column
    - Dropping irrelevant columns
    - Data Splitting
4. Model Training
5. Model Evaluation

CODE:

Importing Libraries and Datasets:

The libraries used are:

- **Pandas:** This library helps to load the data frame in a 2D array format and has multiple functions to perform analysis tasks in one go.
- **Seaborn/Matplotlib:** For data visualization.
- **Numpy:** Numpy arrays are very fast and can perform large computations in a very short time.

In 1:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

The dataset includes the features like type of payment, Old balance, amount paid, name of the destination, etc.

In 2:

```
import pandas as pd
import gdown

# Google Drive shareable link
link =
'https://drive.google.com/file/d/1vNGb0oV8VNiXXjZGtFVzcDeQIi8cVkq7/view
'

# Extracting the ID of the file from its link
file_id = '1vNGb0oV8VNiXXjZGtFVzcDeQIi8cVkq7'

# Creating the direct download URL for the file
download_url = f'https://drive.google.com/uc?id={file_id}'

# Using gdown to download the file from the direct download URL
gdown.download(download_url, 'onlinefraud.csv', quiet=False)

# Loading the csv file into a pandas DataFrame
data = pd.read_csv('onlinefraud.csv')

# Printing the DataFrame
print(data)
```

Out 2:

```
Downloading...
From (original):
https://drive.google.com/uc?id=1vNGb0oV8VNiXXjZGtFVzcDeQIi8cVkq7
From (redirected):
https://drive.google.com/uc?id=1vNGb0oV8VNiXXjZGtFVzcDeQIi8cVkq7&c
onfirm=t&uuid=6d45ec20-3314-4593-bacf-6bd6d29563f3
To: /content/onlinefraud.csv
100%|██████████| 494M/494M [00:05<00:00, 97.3MB/s]
```

```
         step      type      amount     nameOrig  oldbalanceOrg  \
0           1   PAYMENT     9839.64  C1231006815      170136.00
1           1   PAYMENT     1864.28  C1666544295       21249.00
2           1  TRANSFER      181.00  C1305486145         181.00
3           1  CASH_OUT      181.00   C840083671         181.00
4           1   PAYMENT    11668.14  C2048537720       41554.00
...       ...       ...         ...          ...            ...
6362615   743  CASH_OUT   339682.13   C786484425      339682.13
6362616   743  TRANSFER  6311409.28  C1529008245     6311409.28
6362617   743  CASH_OUT  6311409.28  C1162922333     6311409.28
6362618   743  TRANSFER   850002.52  C1685995037      850002.52
6362619   743  CASH_OUT   850002.52  C1280323807      850002.52

         newbalanceOrig     nameDest  oldbalanceDest  newbalanceDest  isFraud  \
0             160296.36  M1979787155            0.00            0.00        0
1              19384.72  M2044282225            0.00            0.00        0
2                  0.00   C553264065            0.00            0.00        1
3                  0.00    C38997010        21182.00            0.00        1
4              29885.86  M1230701703            0.00            0.00        0
...                 ...          ...             ...             ...      ...
6362615            0.00   C776919290            0.00       339682.13        1
6362616            0.00  C1881841831            0.00            0.00        1
6362617            0.00  C1365125890        68488.84      6379898.11        1
6362618            0.00  C2080388513            0.00            0.00        1

         isFlaggedFraud
0                     0
1                     0
2                     0
3                     0
4                     0
...                 ...
6362615               0
6362616               0
6362617               0
6362618               0
6362619               0

[6362620 rows x 11 columns]
```

To print the information of the data we can use data.info() command.

In 3:

```
data.info()
```

The data.info() method in pandas is a convenient tool for getting a quick overview of a DataFrame, including essential details about the dataset's structure and contents.

1.Index Range: It shows the range of the index, indicating the total number of entries in the DataFrame.

2.Column Details:

- Lists all columns in the DataFrame.
- Displays the data type of each column (e.g., int64, float64, object for string or mixed types, datetime64 for date/time, etc.).
- Indicates the number of non-null (non-missing) values in each column, which can help quickly identify columns with missing data.

3.Memory Usage: Provides information about the DataFrame's memory consumption, which is crucial for understanding the dataset's size and managing resources, especially when working with large datasets. By default, it shows an estimate of the memory usage but can be made more precise with the memory_usage='deep' parameter.

4.Data Types Summary: Offers a count of columns by data type, giving a high-level overview of the dataset's composition (how many numeric columns, categorical columns, etc.).

5.Dtype Counts: This is part of the data types summary that specifically counts the number of columns of each data type present in the DataFrame.

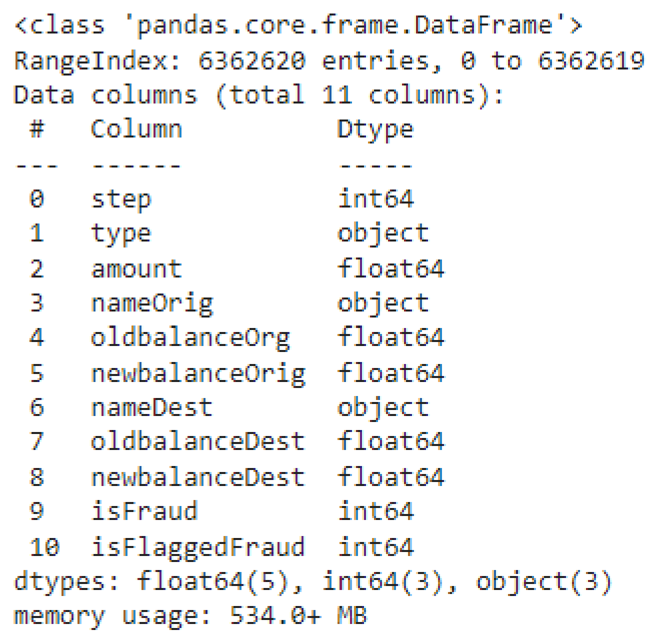
Using data.info() is particularly useful in the initial stages of data analysis and preprocessing. It aids in:

- **Data Cleaning:** Identifying columns with missing values that might need handling through imputation or deletion.
- **Type Conversion:** Spotting columns that may require type conversion, for example, converting object types representing dates to datetime64 or converting numerical IDs from float to int.
- **Optimizing Memory Usage:** Highlighting opportunities to optimize memory usage by converting data types to more memory-efficient

formats (e.g., changing float64 to float32 or using categorical types for columns with a limited number of unique text values).

Overall, data.info() is a quick and efficient way to get a sense of the dataset, allowing data analysts and scientists to make informed decisions about the next steps in data preprocessing and analysis.

Out 3:



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6362620 entries, 0 to 6362619
Data columns (total 11 columns):
 #   Column          Dtype
---  ------          -----
 0   step            int64
 1   type            object
 2   amount          float64
 3   nameOrig        object
 4   oldbalanceOrg   float64
 5   newbalanceOrig  float64
 6   nameDest        object
 7   oldbalanceDest  float64
 8   newbalanceDest  float64
 9   isFraud         int64
 10  isFlaggedFraud  int64
dtypes: float64(5), int64(3), object(3)
memory usage: 534.0+ MB
```

Let’s see the mean, count, minimum and maximum values of the data.

In 4:

```
data.describe()
```

The data.describe() method in pandas is a powerful tool for generating descriptive statistics that summarize the central tendency, dispersion, and shape of a dataset's distribution, excluding NaN (missing) values. It's primarily used for examining the numeric series in the DataFrame but can be extended to include object-type columns as well.

1.Count: Shows the number of non-missing (non-NaN) values in each column. This can quickly inform you about columns that have missing data if the count is less than the total number of entries in the DataFrame.

2.Mean: The mean value of each numeric column. This gives an idea of the average or central value of the data, although it can be influenced by outliers.

3.Std (Standard Deviation): Measures the amount of variation or dispersion of the data points in each numeric column from the mean. A low standard deviation indicates that the data points tend to be close to the mean, whereas a high standard deviation indicates that the data points are spread out over a wider range of values.

4.Min (Minimum Value): The smallest value in each numeric column. It helps in understanding the lower bound of the data's range.

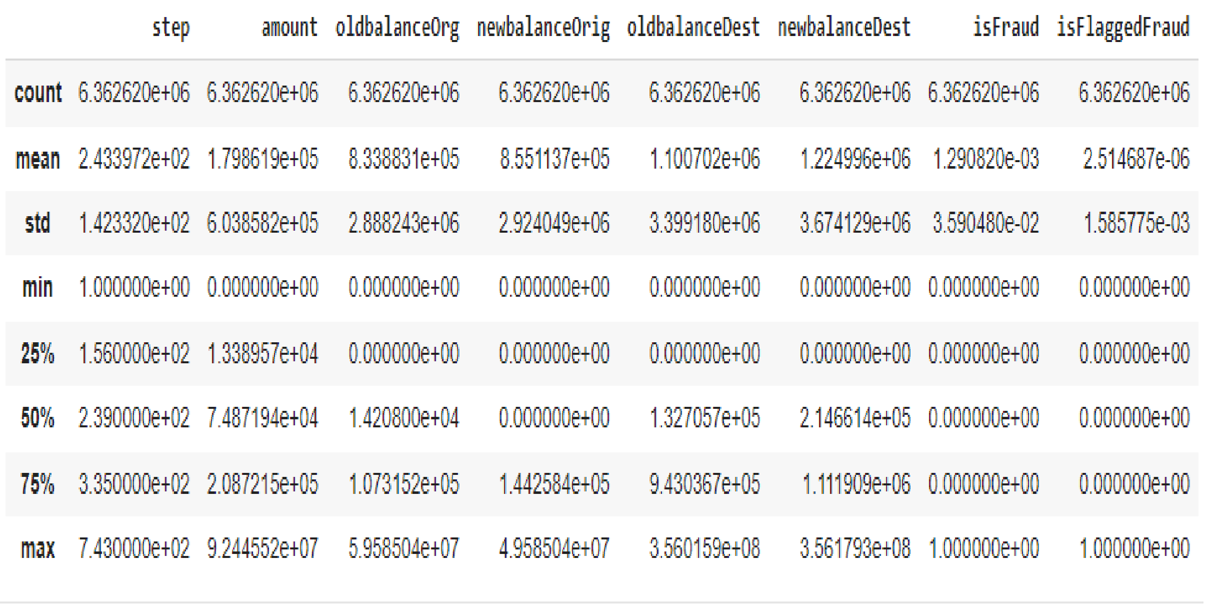
5.25% (First Quartile): The value below which 25% of the data in each numeric column lies. It is also known as the lower quartile and can be used to identify the data's dispersion and the presence of outliers.

6.50% (Median): The middle value of each numeric column when it is ordered from smallest to largest. Half of the data in the column lies below this value. The median is less sensitive to outliers than the mean and can be a better measure of central tendency for skewed distributions.

7.75% (Third Quartile): The value below which 75% of the data in each numeric column lies. It is also known as the upper quartile and, similar to the first quartile, helps in understanding the spread of the data.

8.Max (Maximum Value): The largest value in each numeric column. It helps in understanding the upper bound of the data's range.

Out 4:



	step	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
count	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06
mean	2.433972e+02	1.798619e+05	8.338831e+05	8.551137e+05	1.100702e+06	1.224996e+06	1.290820e-03	2.514687e-06
std	1.423320e+02	6.038582e+05	2.888243e+06	2.924049e+06	3.399180e+06	3.674129e+06	3.590480e-02	1.585775e-03
min	1.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
25%	1.560000e+02	1.338957e+04	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
50%	2.390000e+02	7.487194e+04	1.420800e+04	0.000000e+00	1.327057e+05	2.146614e+05	0.000000e+00	0.000000e+00
75%	3.350000e+02	2.087215e+05	1.073152e+05	1.442584e+05	9.430367e+05	1.111909e+06	0.000000e+00	0.000000e+00
max	7.430000e+02	9.244552e+07	5.958504e+07	4.958504e+07	3.560159e+08	3.561793e+08	1.000000e+00	1.000000e+00

In 5:

```
data.sample(5)
```

data.sample(5) is a handy method for extracting a small, random subset of data from a DataFrame for preliminary inspection, exploratory analysis, or sampling-based methodologies. It provides a straightforward way to gain insights into the dataset's composition and variability.

Out 5:

	step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig	nameDest	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
1655391	158	PAYMENT	4866.47	C1814730262	0.00	0.0	M332451103	0.00	0.00	0	0
6237153	594	TRANSFER	1092231.05	C1234951823	10150.00	0.0	C173972963	341802.22	1434033.27	0	0
1159221	131	CASH_OUT	238813.91	C1850172624	0.00	0.0	C968636049	793778.94	1032592.85	0	0
2462312	203	PAYMENT	4310.77	C1406434538	0.00	0.0	M1222823739	0.00	0.00	0	0
5732620	399	CASH_OUT	293793.23	C24471735	189979.39	0.0	C676292835	416825.86	710619.09	0	0

In 6:

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

data.isnull().sum() is a concise and effective method for assessing missing data within a DataFrame, providing insights that are critical for data cleaning, preprocessing, and quality evaluation.

Out 6:

```
step              0
type              0
amount            0
nameOrig          0
oldbalanceOrg     0
newbalanceOrig    0
nameDest          0
oldbalanceDest    0
newbalanceDest    0
isFraud           0
isFlaggedFraud    0
dtype: int64
```

```
from tabulate import tabulate
fraud_min_max = [
    ['amount', data.amount.min(), data.amount.max()],
    ['oldbalanceOrg', data.oldbalanceOrg.min(),
data.oldbalanceOrg.max()],
    ['newbalanceOrig', data.newbalanceOrig.min(),
data.newbalanceOrig.max()],
    ['oldbalanceDest', data.oldbalanceDest.min(),
data.oldbalanceDest.max()],
    ['isFraud', data.isFraud.min(), data.isFraud.max()]
]

print(
    tabulate(
        fraud_min_max,
        headers=['columns', 'min value', 'max value'],
        showindex=True,
        tablefmt='github',
        numalign='right'
    )
)
```

This code snippet is designed to display the minimum and maximum values of specific columns in a pandas Data Frame named data, focusing on columns relevant to online payment fraud detection. The columns in question include amount, oldbalanceOrg, newbalanceOrig, oldbalanceDest, and is Fraud.

1.Importing Required Library: The code starts by importing the tabulate library, which is used to print tabular data in a nicely formatted table. If tabulate is not installed in your environment, you would need to install it using pip install tabulate.

2.Creating a List of Lists: fraud_min_max is a list that contains other lists. Each inner list corresponds to a specific column of interest from the data DataFrame and contains the column name, its minimum value, and its maximum value. This structure is prepared to summarize the range of values present in each of these columns.

- data.amount.min() and data.amount.max() get the minimum and maximum values for the amount column, respectively. Similar methods are used to get the min and max values for oldbalanceOrg, newbalanceOrig, oldbalanceDest, and isFraud.

3.Using tabulate to Format and Print the Table: The tabulate function is called with several arguments:

- **fraud_min_max:** The list of lists containing the data to be tabulated.
- **headers=['columns', 'min value', 'max value']:** Specifies the headers for the table columns.
- **showindex=True:** Adds an index (row numbers) to the leftmost side of the table.
- **tablefmt='github':** Chooses a table format that mimics GitHub-flavored Markdown tables.
- **numalign='right':** Aligns the numeric data to the right for better readability.

4.Output: The output is a nicely formatted table printed to the console, which shows the minimum and maximum values for the selected columns, helping to understand the range of data involved in the analysis. This information can be particularly useful for exploratory data analysis, especially when dealing with financial transactions, as it provides insights into the scale and distribution of transaction amounts, account balances, and the binary distribution of the isFraud column (typically 0 for non-fraudulent and 1 for fraudulent transactions).

Out 7:

```
|    | columns        |   min value |   max value |
|----|----------------|-------------|-------------|
|  0 | amount         |           0 | 9.24455e+07 |
|  1 | oldbalanceOrg  |           0 |  5.9585e+07 |
|  2 | newbalanceOrig |           0 |  4.9585e+07 |
|  3 | oldbalanceDest |           0 | 3.56016e+08 |
|  4 | isFraud        |           0 |           1 |
```

In 8:

```
# Downcast numerical columns with smaller dtype
for col in data.columns:
    if data[col].dtype == 'float64':
        data[col] = pd.to_numeric(data[col], downcast='float')
    if data[col].dtype == 'int64':
        data[col] = pd.to_numeric(data[col], downcast='unsigned')

# Use category dtype for categorical column
data['type'] = data['type'].astype('category')
```

The code snippet provided is aimed at optimizing the memory usage of a pandas DataFrame named data by down casting numerical columns to smaller data types and converting categorical columns to the category data type.

1.Downcast Numerical Columns: The loop iterates through each column in the DataFrame. For columns with the data type float64, it attempts to downcast them to a smaller floating-point type (float32 if possible) using pd.to_numeric(data[col], downcast='float'). Similarly, for columns with the data type int64, it tries to downcast them to the smallest unsigned integer type that can fit the data (uint8, uint16, uint32, or uint64) using pd.to_numeric(data[col], downcast='unsigned'). This process can significantly reduce the DataFrame's memory footprint, especially when dealing with large datasets.

2.Conversion to Category Data Type for Categorical Columns: The code then converts a specific column, presumed to contain categorical data, to the category data type using data['type'] = data['type'].astype('category'). This is particularly useful for columns with a limited number of distinct values (e.g., transaction types, product categories, gender). The category data type not only reduces memory usage but can also speed up operations like sorting and grouping.

Key Points:

- **Memory Efficiency:** Down casting numerical columns and using the category data type for categorical columns are effective ways to reduce memory consumption. This is crucial for handling large datasets and can lead to performance improvements in data processing.
- **Data Integrity Preservation:** The down casting operation attempts to convert the column to a smaller data type without losing information. It's a safe operation that wouldn't proceed if down casting would lead to data loss.
- **Performance Optimization:** Converting categorical columns to the category data type can speed up operations that involve these columns. This is because pandas can use integer-based operations under the hood instead of string operations, which are generally more computationally expensive.
- **Applicability:** These optimizations are particularly beneficial in the data preprocessing phase, especially before performing data analysis or building machine learning models. They help in managing resources more efficiently, which is essential when working with large datasets or in resource-constrained environments.

In summary, this code snippet is a practical approach to optimizing a DataFrame's memory usage by intelligently down casting numerical columns and converting categorical columns to a more memory-efficient data type.

In 9:

```
# Check duplicate values
data.duplicated().sum()
```

The code snippet data.duplicated().sum() is used to check for duplicate rows in a pandas DataFrame named data.

1.Identifying Duplicates: The data.duplicated() method returns a Boolean Series indicating whether each row is a duplicate (True) or not (False) of a previous row. This method considers all columns in the DataFrame to identify duplicates. By default, it marks the first occurrence of a duplicate row as False (meaning not a duplicate in the context of previous rows) and the subsequent occurrences as True.

2.Summation of Duplicates: The .sum() method is applied to the Boolean Series returned by data.duplicated(). In Python, True is treated as 1 and False as 0. Therefore, summing the Series gives the total count of duplicate rows in the DataFrame.

In summary, the code data.duplicated().sum() is a simple yet powerful way to quantify duplicate data entries in a DataFrame, serving as a foundational step in ensuring the quality and reliability of your data before proceeding with further analysis or modelling.

Out 9:

```
0
```

Data Visualization:

Data visualization plays a crucial role in the field of machine learning, especially in tasks such as online payment fraud detection. Visualizations help in understanding the data, explaining the behavior of models, and communicating findings effectively.

Univariate Data Visualization:

In 10:

```
#Univariate data visualization
data['step'].value_counts()
```

The code snippet data['step'].value_counts() is used to analyze the distribution of values within the 'step' column of a pandas DataFrame named data.

1.Target Column Selection: data['step'] selects the column named 'step' from the DataFrame data. This syntax is used to access a specific column of a DataFrame, returning a pandas Series containing the data of that column.

2.Value Counts Computation: .value_counts() is a method applied to the Series data['step'], which counts the occurrence of each unique value in the 'step' column. This method returns a Series where the index consists of unique values from the 'step' column, and the values are the counts of these unique entries.

3.Descending Order by Default: The counts are sorted in descending order by default, with the most frequent value appearing first. This sorting makes it easy to identify the most and least common values quickly.

4.Use Case and Interpretation: This operation is particularly useful for understanding the distribution of categorical or discrete numerical data. For instance, if 'step' represents time steps or stages in a process, this code would provide a summary of how many data points there are for each time step or stage, highlighting periods with higher or lower activity or occurrences.

5.Data Exploration and Analysis: The output of value_counts() is often used in the exploratory data analysis phase to gain insights into the dataset. Knowing the frequency of each value can help in identifying trends, outliers, data imbalances, or the need for data transformation and cleaning.

6.Visualization: While not shown in this specific code snippet, the results from value_counts() can be easily visualized using plotting libraries such as matplotlib or seaborn to create bar charts, enhancing the analysis with visual aids.

Out 10:

```
step
19    51352
18    49579
187    49083
235    47491
307    46968
     ...
432      4
706      4
693      4
112      2
662      2
Name: count, Length: 743, dtype: int64
```

- There are **743** steps, and every step has **at least 2** occurrences.

In 11:

```
ax = sns.countplot(x='type', data=data, palette='PuBu')
for container in ax.containers:
    ax.bar_label(container)
plt.title('Count plot of transaction type')
plt.legend(bbox_to_anchor=(1.05,1), loc='upper left')
plt.ylabel('Number of transactions')
```

The code snippet uses the seaborn library for data visualization to create a count plot, which is a type of bar plot that shows the frequency of categories from a column in a DataFrame.

1.Count Plot Creation:

sns.countplot(x='type', data=data, palette='PuBu') generates a count plot for the 'type' column in the data DataFrame, using the 'PuBu' color palette. This function counts the occurrences of each unique value in the 'type' column and creates a bar for each unique value, with the height of the bar representing the count of occurrences.

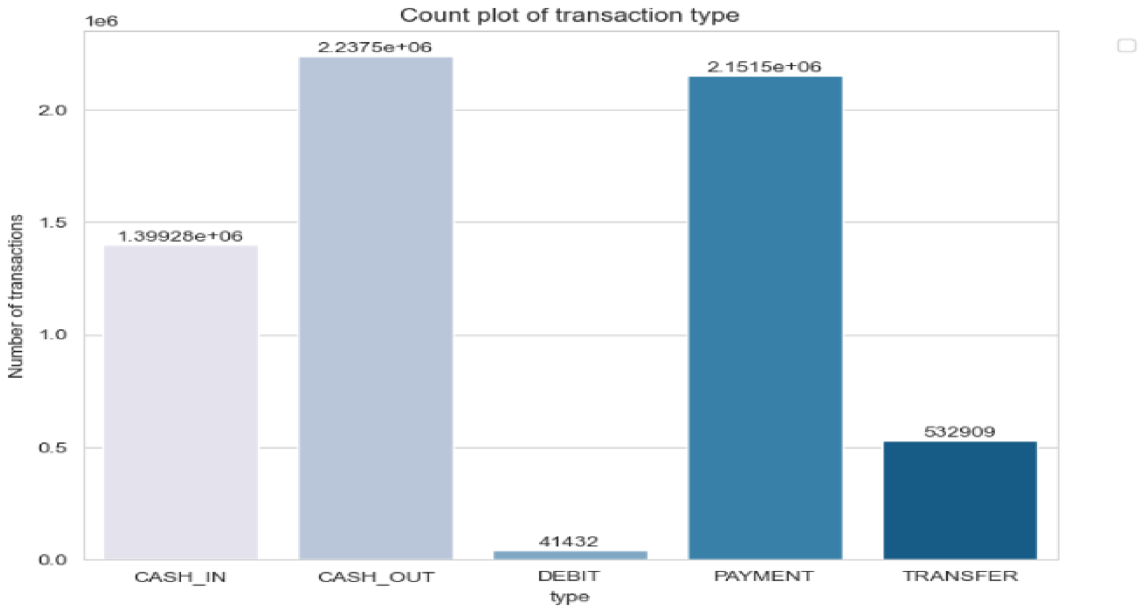
2.Bar Labeling:

for container in ax.containers: ax.bar_label(container) iterates through each container of bars in the plot. For each container, it labels the bars with their respective counts. This enhances the plot by directly displaying the count of occurrences above each bar, making it easier to read and interpret.

This code snippet efficiently creates a count plot showing the distribution of transaction types within a dataset, which is a common task in exploratory data analysis, particularly useful for understanding categorical data distributions.

Out 11:

Text(0, 0.5, 'Number of transactions')



- **Cash out** is **the most numerous** transaction type, followed by payment, cash in, transfer and debit types.

In 12:

```
sns.kdeplot(data['amount'], linewidth=4)
plt.title('Distribution of transaction amount')
```

The code snippet utilizes the seaborn library to create a Kernel Density Estimation (KDE) plot for visualizing the distribution of transaction amounts within a dataset.

1.Seaborn Library: The sns.kdeplot function is part of the seaborn library (sns), which is a high-level visualization library built on top of matplotlib. It provides a more convenient interface for creating complex visualizations, including KDE plots.

2.Kernel Density Estimation Plot: The KDE plot is used to estimate the probability density function of a continuous random variable. In this context, it visualizes the distribution of transaction amounts, allowing one to see the density (the likelihood of observing a value in a particular range) of these amounts across the dataset.

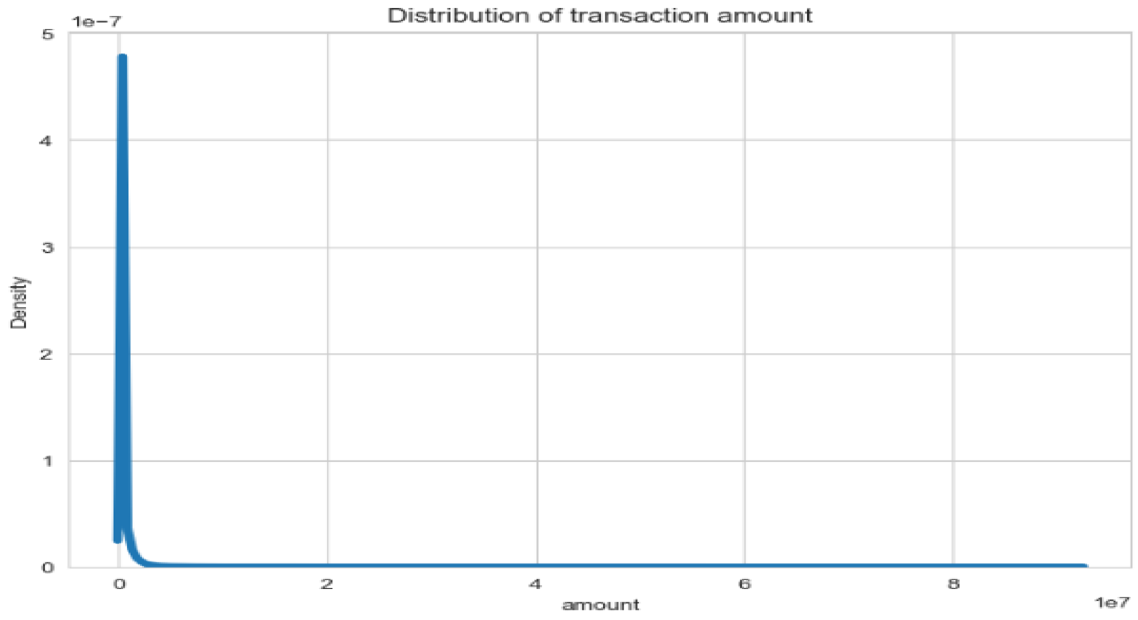
3.data['amount']: This specifies the column from the pandas DataFrame data that contains the transaction amounts. The KDE plot will be generated based on the values in this column.

4.linewidth=4: This parameter customizes the appearance of the KDE plot, setting the width of the line used to draw the density curve to 4. This makes the line more prominent in the plot.

This code creates a KDE plot to visualize the distribution of transaction amounts, offering insights into the common ranges of transaction values and potential outliers. It's particularly useful in financial data analysis, fraud detection, or any scenario where understanding the distribution of monetary values is essential.

Out 12:

Text(0.5, 1.0, 'Distribution of transaction amount')



- The distribution of transaction amounts is **right skewed**.
- This indicates that most values are clustered around the left tail of the distribution, with the longer right tail.
- (mode < median < mean)

In 13:

```
data['nameOrig'].value_counts()
```

The code data['nameOrig'].value_counts() is used to analyze the frequency distribution of unique values within the 'nameOrig' column of a pandas DataFrame named data.

1.DataFrame Column Access: data['nameOrig'] is used to access the column named 'nameOrig' in the DataFrame data. This assumes that data is a pandas DataFrame containing multiple columns, and 'nameOrig' is one of those columns.

2.value_counts() Method: After accessing the 'nameOrig' column, the value_counts() method is applied to it. This method computes the frequency/count of each unique value in the column. In this case, it will count how many times each unique 'nameOrig' value appears in the dataset.

3.Result: The result is a pandas Series where the index consists of unique 'nameOrig' values, and the corresponding values represent the count of each unique 'nameOrig' value in the DataFrame data.

Out 13:

```
nameOrig
C1902386530   3
C363736674    3
C545315117    3
C724452879    3
C1784010646   3
        ..
C98968405     1
C720209255    1
C1567523029   1
C644777639    1
C1280323807   1
Name: count, Length: 6353307, dtype: int64
```

- There are **6353307** initial customers, and every step has **at least 1** occurrence.

In 14:

```
sns.kdeplot(data['oldbalanceOrg'], linewidth=4)
plt.title('Distribution of transaction amount')
```

The code snippet uses seaborn, a Python visualization library, to plot a Kernel Density Estimation (KDE) of the oldbalanceOrg column from a DataFrame named data.

1.Visualization Library: The snippet utilizes seaborn (sns), which is built on top of matplotlib and provides a high-level interface for drawing attractive and informative statistical graphics.

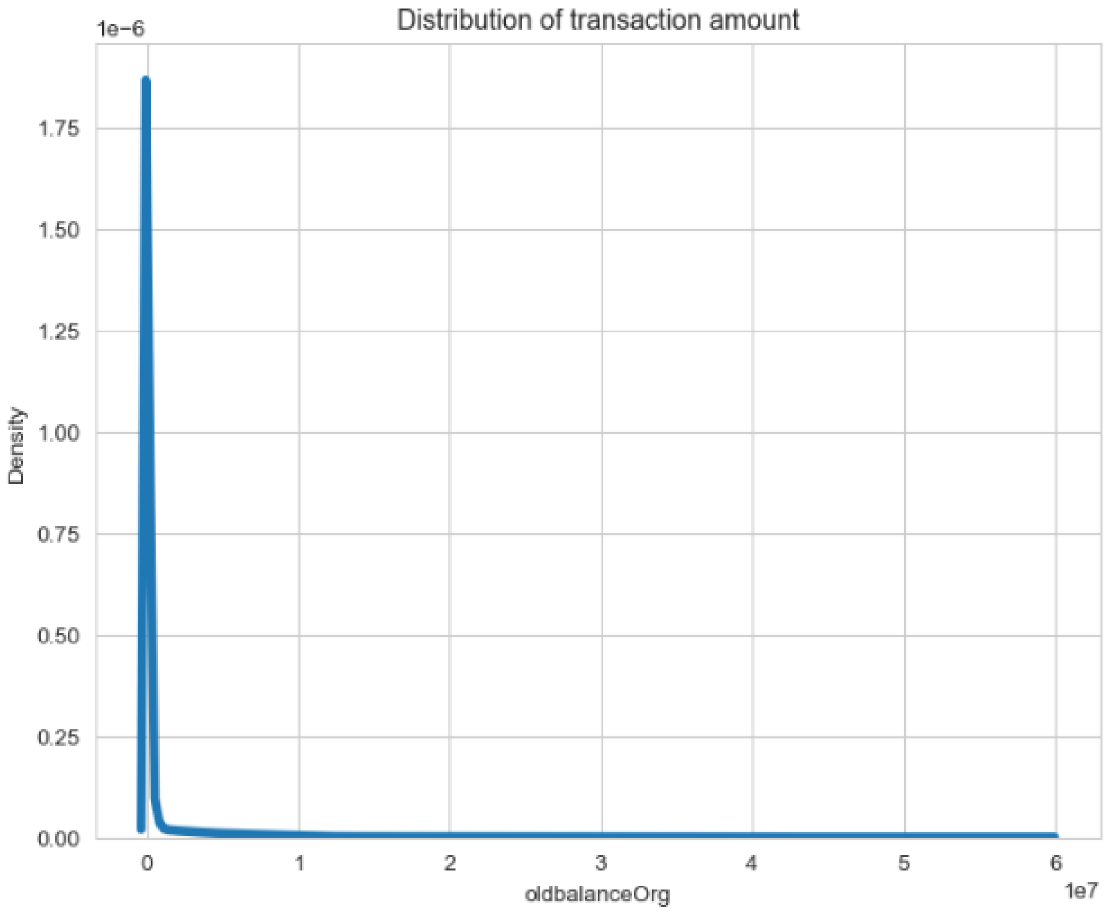
2.Kernel Density Estimation Plot: By calling sns.kdeplot, the code generates a KDE plot, which is a way to estimate the probability density function of a continuous random variable. KDE plots are useful for visualizing the distribution of data points in a single variable.

3.Target Column - oldbalanceOrg: The plot is specifically made for the oldbalanceOrg column in the data DataFrame. This column is presumably holding numerical values representing the initial balance before a transaction. The KDE plot will show the distribution of these balance amounts.

4.Line Width: The linewidth=4 parameter increases the width of the density curve to 4, making it more visually prominent on the plot.

Out 14:

```
Text(0.5, 1.0, 'Distribution of transaction amount')
```



- The distribution of pre-transaction balances of the initial customers is **right skewed**.

In 15:

```
sns.kdeplot(data['newbalanceOrig'], linewidth=4)
plt.title('Distribution of transaction amount')
```

The code snippet uses the seaborn library to generate a Kernel Density Estimation (KDE) plot, this time for the 'newbalanceOrig' column in the DataFrame data.

1.Visualization Library: Seaborn is a powerful, high-level visualization library built on matplotlib. It simplifies the creation of informative and attractive statistical graphics.

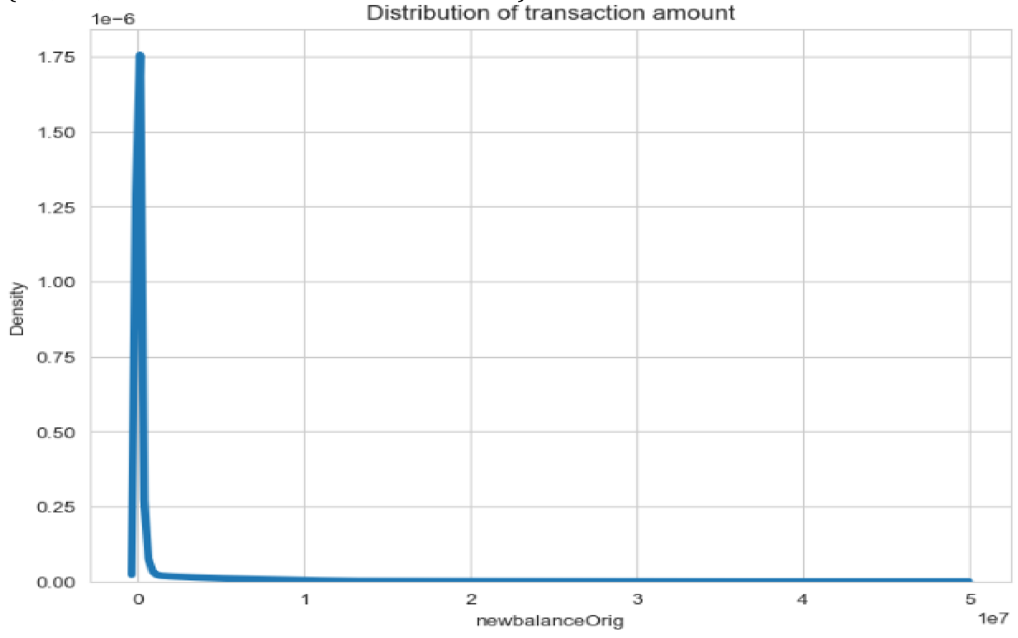
2.Kernel Density Estimation (KDE) Plot: A KDE plot is a method for visualizing the distribution of observations in a dataset, analogous to a histogram. KDE represents the data using a continuous probability density curve in one or more dimensions.

3.Targeted Data Column: The column 'newbalanceOrig' from the DataFrame data is used for this KDE plot. This column likely represents the new balance of an origin account following a transaction, providing insights into the balance distribution after transactions have occurred.

4.Line Width Customization: The linewidth=4 parameter thickens the line used in plotting the KDE, making it more visually prominent.

Out 15:

```
Text(0.5, 1.0, 'Distribution of transaction amount')
```



The distribution of post-transaction balances of the initial customers is **right skewed**.

In 16:

```
data['nameDest'].value_counts()
```

The code data['nameDest'].value_counts() performs a value count operation on the 'nameDest' column of a pandas DataFrame named data. This operation counts the occurrence of each unique value in the 'nameDest' column and returns a Series where each index represents a unique entry from the 'nameDest' column, and the corresponding value indicates how many times that entry appears in the dataset.

1.Pandas DataFrame: Assumes that data is a pandas DataFrame, which is a two-dimensional, size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns).

2.Column Selection: data['nameDest'] accesses the 'nameDest' column of the DataFrame. This column likely contains categorical data representing the destination names or identifiers in a dataset, such as account numbers or customer IDs in a financial transactions dataset.

3.Value Counts Method: .value_counts() is a method available for pandas Series objects. It returns a Series containing counts of unique values in descending order so that the first element is the most frequently-occurring element. It excludes NA values by default

4.Result: The result of this operation is a pandas Series where each index corresponds to a unique value from the 'nameDest' column, and each value represents the number of times that unique value appears in the column.

Out 16:

```
nameDest
C1286084959   113
C985934102    109
C665576141    105
C2083562754   102
C248609774    101
         ...
M1470027725     1
M1330329251     1
M1784358659     1
M2081431099     1
C2080388513     1
Name: count, Length: 2722362, dtype: int64
```

- There are 2722362 recipients, and every step has at least 1 occurrence.

In 17:

```
sns.kdeplot(data['oldbalanceDest'], linewidth=4)
plt.title('Distribution of transaction amount')
```

The code snippet creates a Kernel Density Estimation (KDE) plot for the distribution of values in the 'oldbalanceDest' column of a pandas DataFrame named data. This visualization is achieved using the seaborn library, with additional plot formatting in matplotlib.

1.Usage of Seaborn: The seaborn library (sns), a high-level visualization tool built on matplotlib, is used for creating the KDE plot. Seaborn simplifies the creation of complex visualizations, including KDE plots which estimate the probability density function of a dataset.

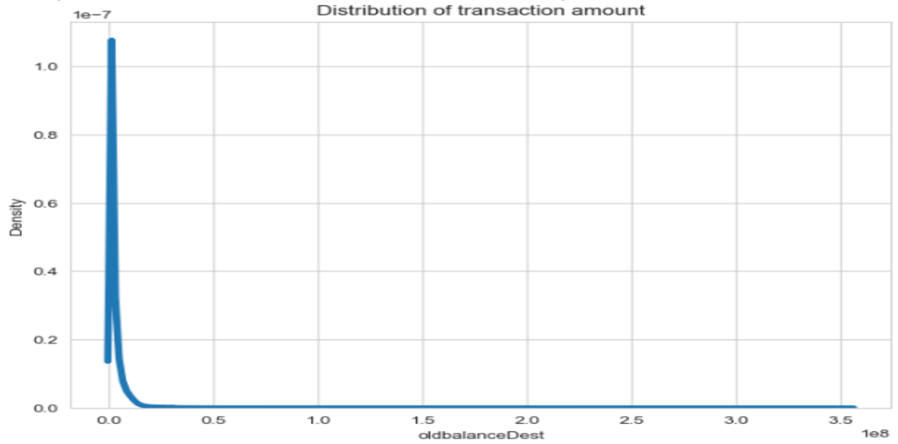
2.Kernel Density Estimation (KDE) Plot: A KDE plot is a method to visualize the distribution of continuous data, providing a smooth curve that approximates the probability density function. In this code, the KDE plot is applied to the 'oldbalanceDest' column, which presumably represents the balance in the destination account before a transaction occurs.

3.Data Column: The 'oldbalanceDest' column selected for plotting indicates the focus is on understanding the distribution of destination account balances prior to transactions. This could help in identifying typical balance ranges, detecting outliers, or understanding balance distribution characteristics relevant in financial analyses, such as fraud detection.

4.Line Width Customization: The linewidth=4 parameter in sns.kdeplot specifies the thickness of the KDE curve, making it more visually prominent in the plot.

Out 17:

```
Text(0.5, 1.0, 'Distribution of transaction amount')
```



- The distribution of pre-transaction balances of the recipient is **right skewed**.

In 18:

```
sns.kdeplot(data['newbalanceDest'], linewidth=4)
plt.title('Distribution of transaction amount')
```

The code snippet uses seaborn, a powerful visualization library in Python, to create a Kernel Density Estimation (KDE) plot for the 'newbalanceDest' column in a pandas DataFrame named data. The purpose of this visualization is to analyze the distribution of values in the 'newbalanceDest' column, which presumably represents the new balance in the destination account after a transaction has occurred.

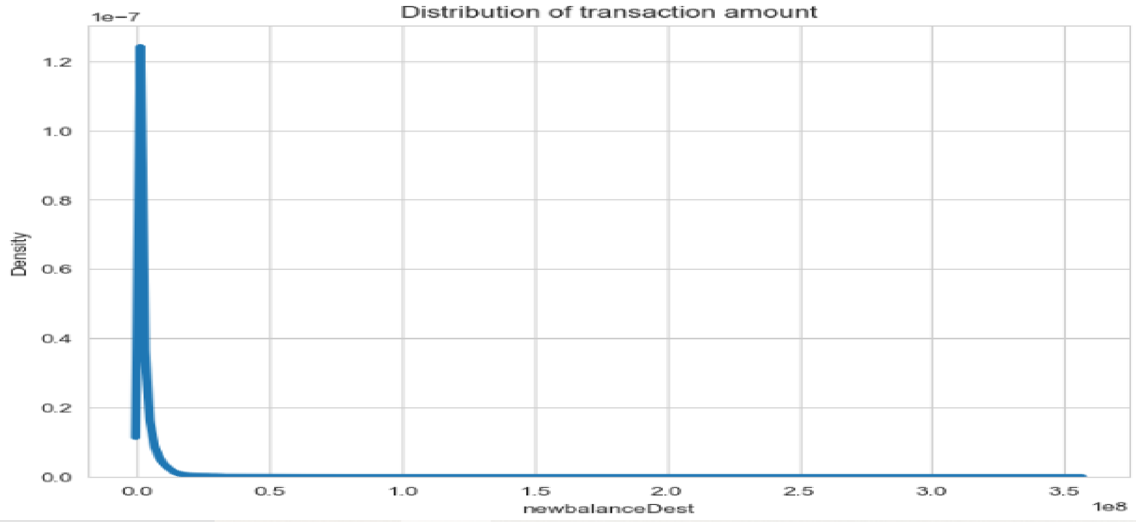
1.Seaborn for Visualization: The sns.kdeplot function from seaborn is used to generate a KDE plot. KDE plots are useful for visualizing the distribution of continuous or large datasets by estimating the probability density function.

2.Targeted Data Column: The 'newbalanceDest' column is targeted for visualization. This column likely contains numerical values representing the amounts in destination accounts after transactions are completed, making it a key variable for financial analysis, including fraud detection.

3.Visualization Customization: The linewidth=4 parameter enhances the visual appeal and clarity of the KDE plot by making the density curve thicker.

Out 18:

```
Text(0.5, 1.0, 'Distribution of transaction amount')
```



- The distribution of pre-transaction balances of the recipient is **right skewed**.

In 19:

```
ax = sns.countplot(x='isFraud', data=data, palette='PuBu')
for container in ax.containers:
    ax.bar_label(container)
plt.title('Count plot of fraud transaction')
plt.ylabel('Number of transactions')

del ax
```

The code snippet creates a count plot using seaborn to visualize the distribution of fraudulent and non-fraudulent transactions within a dataset. This plot is particularly useful for understanding the balance between fraudulent and non-fraudulent transactions in the dataset, which can be crucial for tasks like fraud detection analysis.

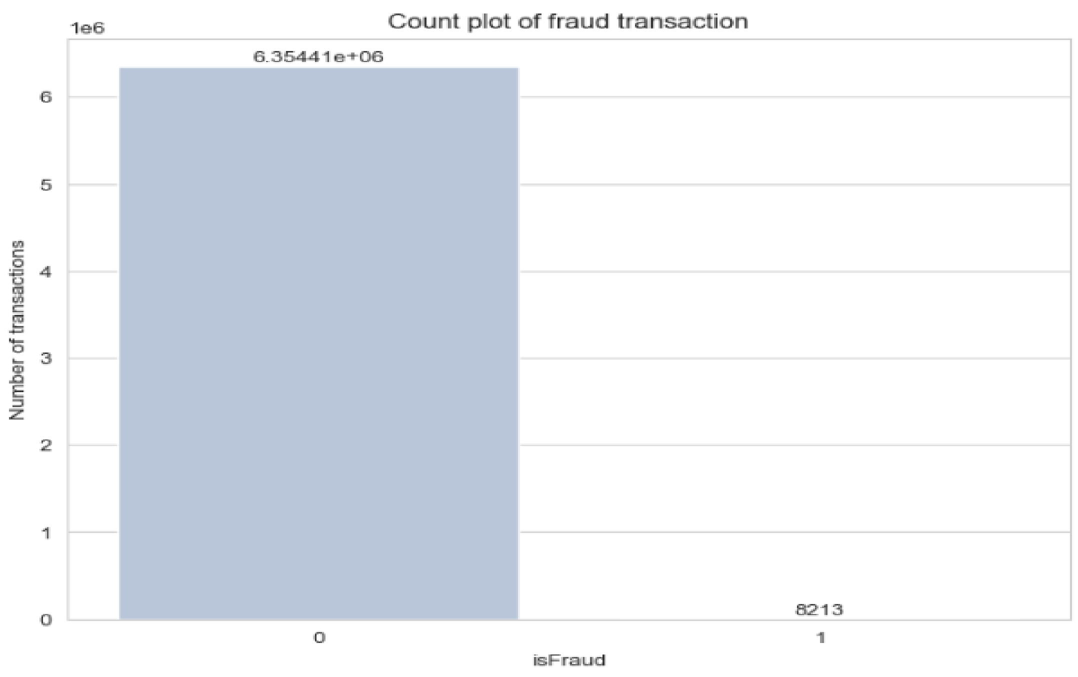
1.Count Plot Creation: sns.countplot(x='isFraud', data=data, palette='PuBu') generates a count plot for the 'isFraud' column of the DataFrame data, using the 'PuBu' color palette. The 'isFraud' column is assumed to be a binary indicator of whether a transaction is fraudulent (1) or not (0).

2.Bar Labeling: The loop for container in ax.containers: ax.bar_label(container) iterates over each bar in the plot (stored in ax.containers) and labels it with its height. This feature visually represents the count of occurrences directly on the bars, making the plot more informative by displaying the exact number of fraudulent and non-fraudulent transactions.

3.Cleaning Up: The line del ax at the end removes the reference to the plot's axis object (ax). This is generally not necessary unless you're explicitly managing memory or namespace cleanliness in your environment.

4.Visualization Utility: This visualization is helpful for initial exploratory data analysis (EDA), offering insights into the class distribution of the dataset. Imbalanced classes, where fraudulent transactions are significantly fewer than non-fraudulent ones, pose challenges for machine learning models, making such visualizations crucial for planning data preprocessing steps like resampling.

Out 19:



- There are much **more non-fraudulent transactions** than fraudulent transactions.

Bivariate Visualization:

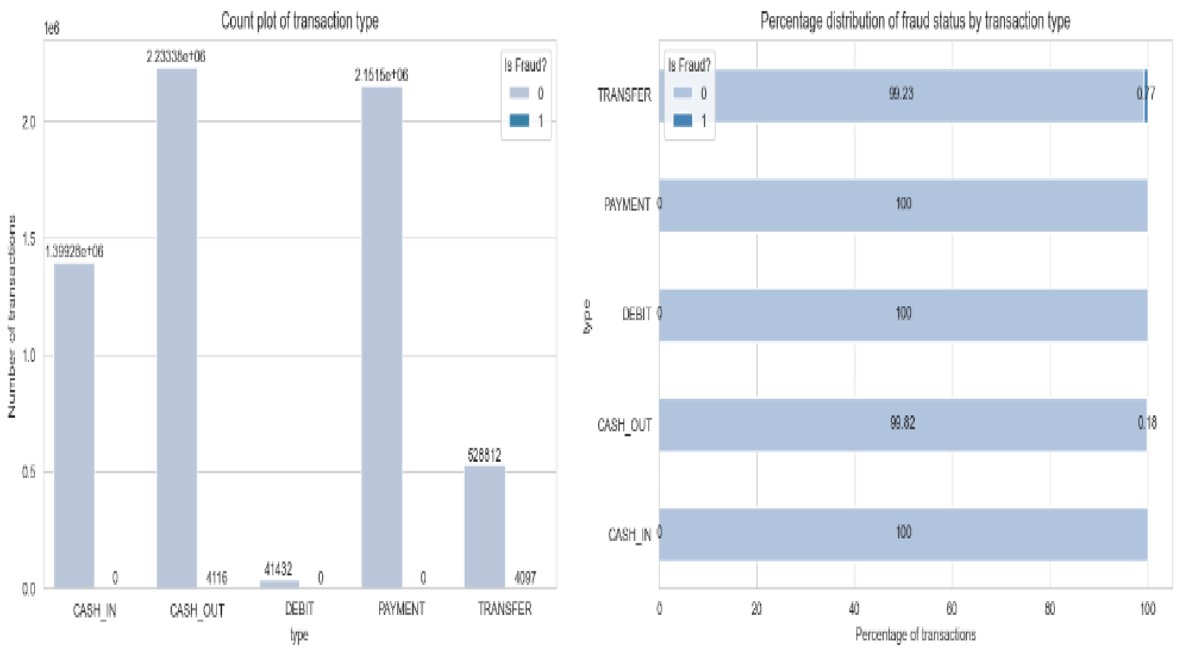
In 20:

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# Assuming 'data' DataFrame is pre-defined and includes 'type' and
'isFraud' columns

# Convert 'isFraud' to string to ensure proper legend handling
data['isFraud'] = data['isFraud'].astype(str)

# Bivariate data visualization
fig, ax = plt.subplots(1, 2, figsize=(20, 5))

# First plot: Count plot of transaction type by fraud status
sns.countplot(x='type', data=data, hue='isFraud', palette='PuBu',
ax=ax[0])
for container in ax[0].containers:
    ax[0].bar_label(container)
ax[0].set_title('Count plot of transaction type')
ax[0].legend(title='Is Fraud?')  # Ensured the legend has a title
```

```
ax[0].set_ylabel('Number of transactions')

# Second plot: Percentage distribution of fraud status across
transaction types
data2 = data.groupby(['type', 'isFraud']).size().unstack(fill_value=0)
data2_percent = data2.apply(lambda x: round(x / sum(x) * 100, 2),
axis=1)
data2_percent.plot(kind='barh', stacked=True, color=['lightsteelblue',
'steelblue'], ax=ax[1])
for container in ax[1].containers:
    ax[1].bar_label(container, label_type='center')
ax[1].set_title('Percentage distribution of fraud status by transaction
type')
ax[1].legend(title='Is Fraud?')  # Ensured the legend has a title
ax[1].set_xlabel('Percentage of transactions')
ax[1].grid(axis='y')
```

The code snippet efficiently combines both seaborn and matplotlib libraries to create a detailed visualization comprising two plots side by side. Each plot serves a distinct purpose in analyzing transaction data based on the type and fraud status.

Out 20:



- Fraudulent transactions only occur in debit and transfer types.

In 21:

```
data['quantity'] = pd.cut(data['amount'], 5, labels=['very low', 'low',
'moderate', 'high', 'very high'])

ax = sns.countplot(x='quantity', data=data, hue='isFraud',
palette='PuBu')
for container in ax.containers:
    ax.bar_label(container)
plt.title('Count plot of amount quantity')
plt.legend(bbox_to_anchor=(1.05,1), loc='upper left')
plt.ylabel('Number of transactions')
```

The code snippet effectively segments transaction amounts into categories and visualizes the distribution of these categories with respect to fraud status. It's an insightful way to analyze how transaction amounts correlate with the likelihood of being fraudulent.

1.Segmenting Transaction Amounts:

pd.cut(data['amount'], 5, labels=['very low', 'low', 'moderate', 'high', 'very high']) segments the 'amount' column into 5 bins, categorizing each transaction as 'very low', 'low', 'moderate', 'high', or 'very high' based on its amount. This categorization makes it easier to analyze the distribution of transactions across different amount ranges.

2.Visualization with Seaborn:

sns.countplot(x='quantity', data=data, hue='isFraud', palette='PuBu') creates a count plot where transactions are grouped by their categorized amounts ('quantity'). The hue='isFraud' parameter further splits each amount category by fraud status, offering a clear view of the relationship between transaction amounts and fraud.

3.Enhancing Plot Readability:

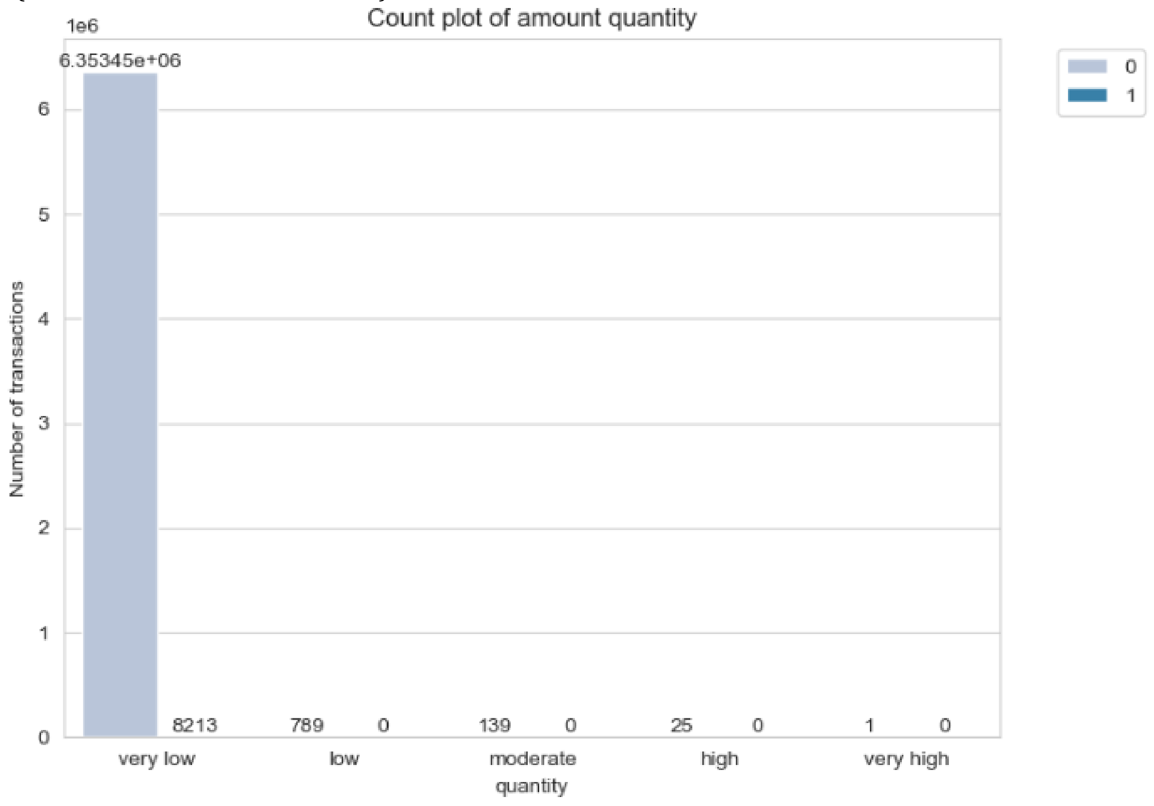
The loop for container in ax.containers: ax.bar_label(container) iterates through the bars generated by the count plot and labels each with its count. This addition makes the plot more informative by directly displaying the number of transactions in each category and fraud status.

4.Adjusting Legend Position:

plt.legend(bbox_to_anchor=(1.05,1), loc='upper left') moves the legend outside the plot area to the upper left corner, ensuring it doesn't obscure any data. This placement improves the plot's readability and aesthetics.

Out 21:

```
Text(0, 0.5, 'Number of transactions')
```



- All fraudulent transactions fall into the category of very low amounts.
- This suggests that in most cases, small transactions are more prone to fraudulent transactions.

In 22:

```python
data['oldbalanceOrg_amt'] = pd.cut(data['oldbalanceOrg'], 5,
labels=['very low', 'low', 'moderate', 'high', 'very high'])
ax = sns.countplot(x='oldbalanceOrg_amt', data=data, hue='isFraud',
palette='PuBu')
for container in ax.containers:
    ax.bar_label(container)
plt.title('Count plot of initial customers pre-transaction balance
amount')
plt.legend(bbox_to_anchor=(1.05,1), loc='upper left')
plt.ylabel('Number of transactions')
plt.xlabel('Initial customers pre-transaction balance amount')
```

The code is designed to visualize how the initial balance of customers (before transactions) correlates with fraudulence. By categorizing these balances into discrete groups and creating a count plot, it provides a clear overview of any patterns that might indicate a relationship between the amount of money customers had initially and the likelihood of their transactions being fraudulent.

1.Categorizing oldbalanceOrg Values:

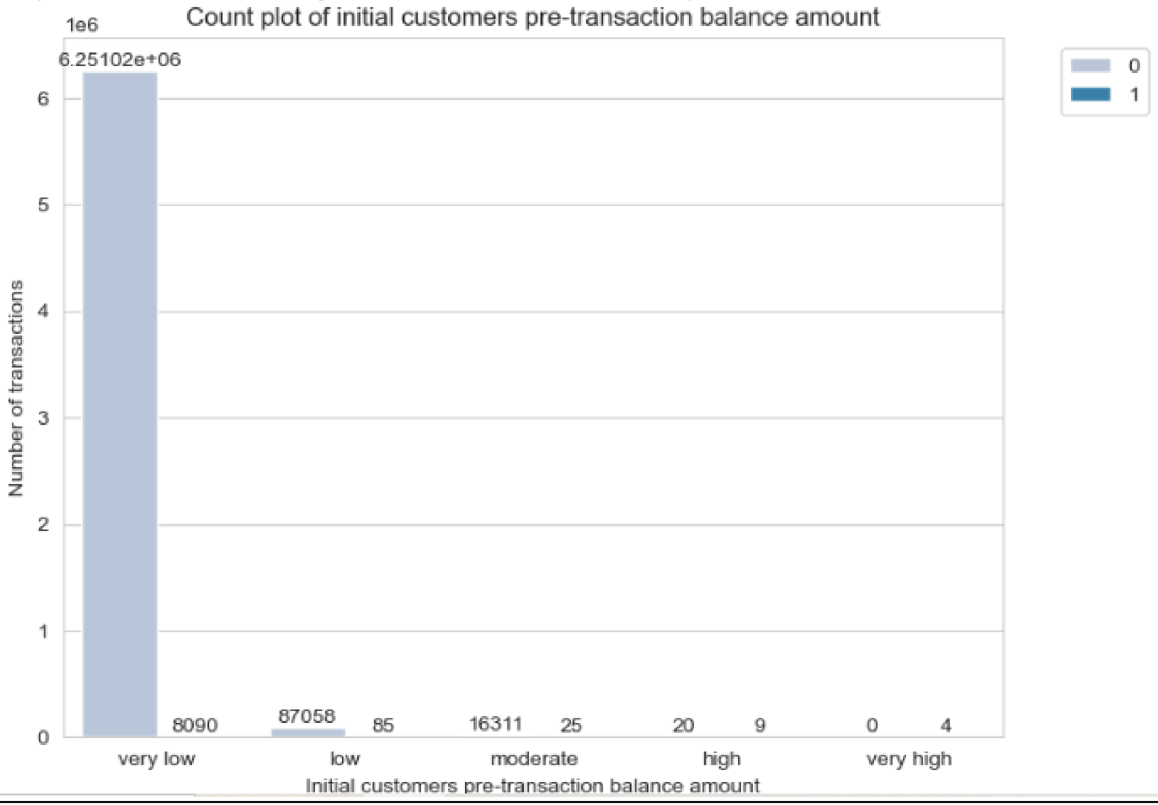
The pd.cut function is used to divide the 'oldbalanceOrg' values into 5 categories based on the balance amount. This transforms continuous numerical data into categorical data, making it easier to analyze in a count plot.

2.Creating a Count Plot with Seaborn:

sns.countplot generates a count plot for the categorized balance amounts. The use of the hue='isFraud' parameter adds a layer of comparison, illustrating the count of fraudulent versus non-fraudulent transactions within each balance category.

Out 22:

```
Text(0.5, 0, 'Initial customers pre-transaction balance amount')
```



- Initial customers with very low pre-transaction balances has the highest number of fraudulent transactions.
- This means that initial customers with very low pre-transaction balances may be more likely to fall for a fraudulent transaction.

```
data['oldbalanceDest_amt'] = pd.cut(data['oldbalanceDest'], 5,
labels=['very low', 'low', 'moderate', 'high', 'very high'])

ax = sns.countplot(x='oldbalanceDest_amt', data=data, hue='isFraud',
palette='PuBu')
for container in ax.containers:
    ax.bar_label(container)
plt.title('Count plot of recipients pre-transaction balance amount')
plt.legend(bbox_to_anchor=(1.05,1), loc='upper left')
plt.ylabel('Number of transactions')
plt.xlabel('Recipient pre-transaction balance amount')
```

The code effectively categorizes the pre-transaction balance amounts of recipients ('oldbalanceDest') into discrete groups and visualizes the distribution of these groups with respect to fraudulence. This approach is particularly insightful for understanding if certain balance ranges are more frequently associated with fraudulent transactions.

1.Categorizing Pre-transaction Balances:

Using pd.cut to segment 'oldbalanceDest' into categories ('very low' to 'very high') simplifies the continuous numerical data into categorical bins. This categorization aids in the analysis of balance ranges and their potential correlation with fraudulent activities.

2.Visualizing with Count Plot:

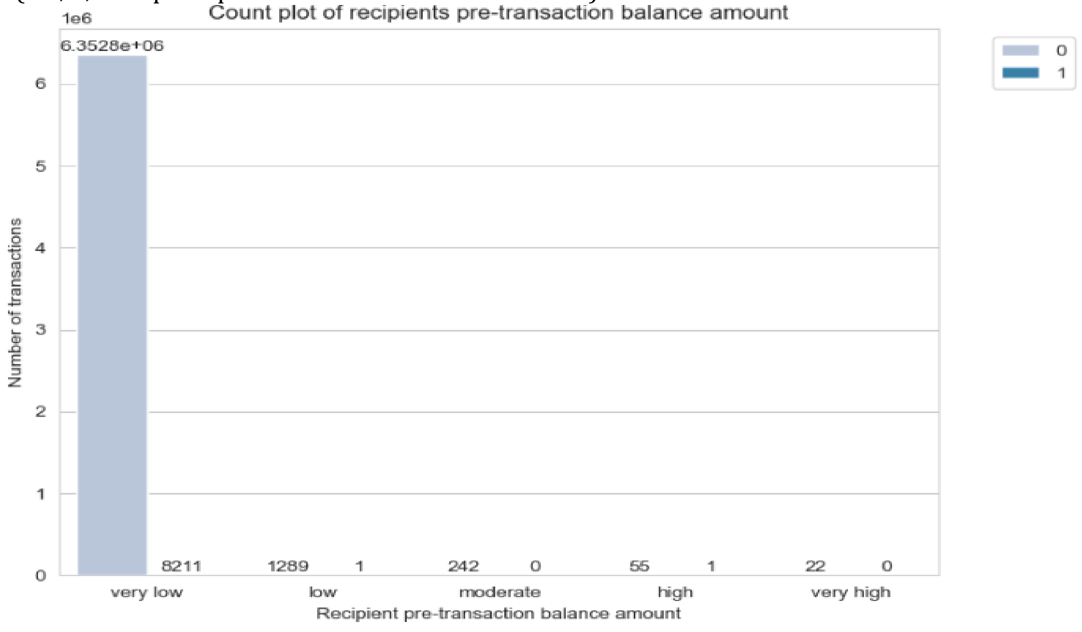
The sns.countplot function creates a visual representation that contrasts the number of fraudulent and non-fraudulent transactions across the defined balance categories. The hue='isFraud' parameter distinguishes between the two conditions within each balance category.

3.Bar Labeling for Clarity:

Labeling each bar with its count directly on the plot enhances the interpretability of the data. This step allows viewers to immediately gauge the number of transactions within each balance category and fraudulence condition.

Out 23:

```
Text(0.5, 0, 'Recipient pre-transaction balance amount')
```



- Recipients with **very low pre-transaction balances** has the highest number of fraudulent transactions.
- This implies that recipients with very low pre-transaction balances may be more susceptible to fraudulent transactions.

Let's count the columns with different datatypes like Category, Integer, Float.

In 24:

```
data = data.iloc[:,:-1]
```

The code data = data.iloc[:,:-1] is a pandas operation that modifies the data DataFrame by selecting all rows and all columns except the last one.

1.Indexing with .iloc: The .iloc indexer for pandas DataFrames is used to select rows and columns by integer location. It allows for traditional Python indexing syntax, namely, [row selection, column selection].

2.Selecting Rows: The colon : before the comma indicates that all rows are included in the selection. There's no filtering or exclusion applied to the rows; every row from the original DataFrame is preserved in the resulting DataFrame.

3.Selecting Columns: The :,:-1 after the comma is used to select columns. The colon : by itself would normally select all columns, but the addition of :-1 modifies this selection to exclude the last column. The -1 signifies the last column due to Python's zero-based indexing and negative indexing for sequences, where -1 refers to the last element.

4.Assignment to data: The result of this operation (all rows and all but the last column of the original data DataFrame) is assigned back to the variable data. This effectively updates the data DataFrame to exclude the last column from the original dataset.

5.Use Case: This operation can be useful in situations where the last column of the dataset is not needed for further analysis or modeling. For example, the last column might contain labels or outcomes that you want to separate from the feature set in a machine learning context, or it might be a column of metadata that's not relevant to your current analysis.

6.Impact on the DataFrame: It's important to note that this operation reduces the number of columns in the DataFrame by one, permanently modifying the data variable unless further steps are taken to either reverse this operation or copy the original DataFrame before applying this change.

Overall, this code snippet is a concise way to exclude the last column from a pandas DataFrame, which can be particularly handy in data preprocessing and feature selection steps of a data analysis workflow.

In 25:

```
obj = (data.dtypes == 'object')
object_cols = list(obj[obj].index)
print("Categorical variables:", len(object_cols))

int_ = (data.dtypes == 'int')
num_cols = list(int_[int_].index)
print("Integer variables:", len(num_cols))

fl = (data.dtypes == 'float')
fl_cols = list(fl[fl].index)
print("Float variables:", len(fl_cols))
```

This code snippet is used to categorize and count columns in a pandas DataFrame (data) based on their data types, specifically identifying object (categorical), integer, and float columns.

1.Identifying Object Columns:

- **obj = (data.dtypes == 'object'):** This line creates a Series where each element is a boolean indicating whether the corresponding column in data has the dtype 'object'. Object dtype typically represents categorical data or text in pandas.

- **object_cols = list(obj[obj].index):** This extracts the column names that have True in the obj series, which means their data type is 'object'. These names are stored in the list object_cols.
- The print statement displays the number of categorical variables found by counting the length of object_cols.

2.Identifying Integer Columns:

- **int_ = (data.dtypes == 'int'):** Similar to the first step, this creates a Series where each element is a boolean value indicating whether the corresponding column's dtype is an integer type.
- **num_cols = list(int_[int_].index):** Extracts the names of columns that are of integer type and stores them in num_cols.
- The print statement then shows how many integer variables are present by counting the elements in num_cols.

3.Identifying Float Columns:

- **fl = (data.dtypes == 'float'):** This line generates a Series with boolean values indicating if each column in data has the dtype 'float'.
- **fl_cols = list(fl[fl].index):** Extracts the column names for those that are of float type and stores them in fl_cols.
- A print statement follows to indicate the number of float variables by counting the items in fl_cols.

Out 25:

```
Categorical variables: 2
Integer variables: 0
Float variables: 5
```

Let's see the count plot of the Payment type column using Seaborn library.

In 26:

```python
import seaborn as sns
sns.countplot(x='type', data=data)
```

This code snippet uses the seaborn library, a popular Python visualization library based on matplotlib, to create a count plot.

1.Importing Seaborn: The code begins with import seaborn as sns, which imports the seaborn library and aliases it as sns. This alias is commonly used and allows for easier access to seaborn's functions.

2.Creating a Count Plot: sns.countplot(x='type', data=data) generates a count plot for the type column of the DataFrame data. This type of plot displays the counts of observations in each categorical bin using bars.

3.**Parameters of sns.countplot():**

- **x='type':** This specifies that the column 'type' in the DataFrame should be used for the x-axis. Each unique value in the type column becomes a separate category on the x-axis, and the height of the bars represents the number of occurrences of each category.
- **data=data:** This indicates that the source of the data is the DataFrame named data.

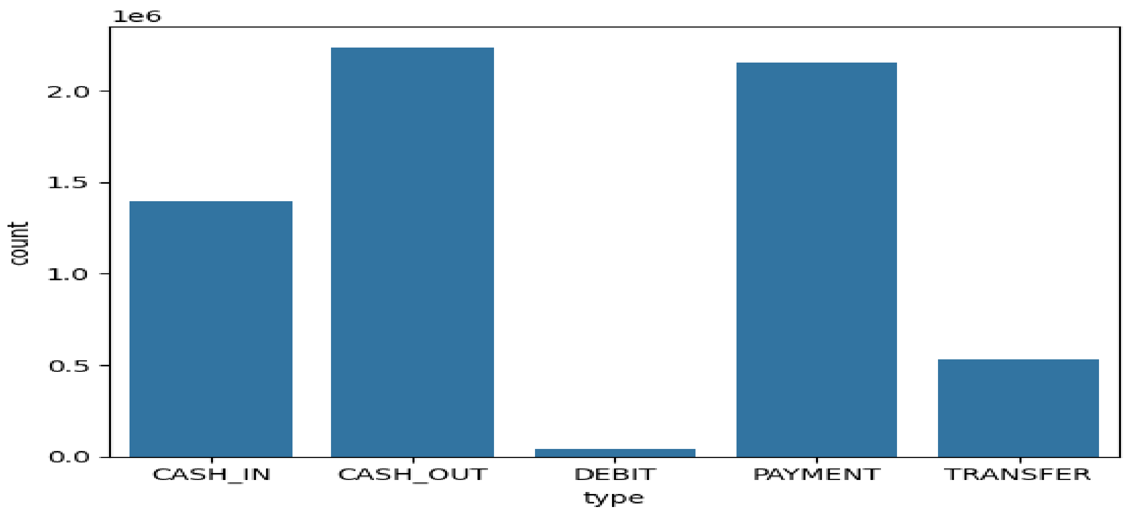
4.Visualization Purpose: The count plot is useful for visualizing the distribution of categorical data. It allows one to easily see which categories are most common and how different categories compare in terms of frequency.

5.Use Case: In the context of this code, if data is a DataFrame related to transactions or events with a type column indicating the category or type of each transaction/event, the count plot will visually summarize how many transactions/events there are of each type. This can be particularly insightful for data exploration, identifying trends, and making decisions based on the distribution of categorical variables.

6.Customization and Extensions: While not shown in this snippet, seaborn's countplot function offers various ways to customize the plot, including changing the color of the bars, ordering the categories, orienting the bars horizontally instead of vertically, among others. These customizations can enhance the plot's readability and effectiveness in conveying the desired information.

Out 26:

```
<Axes: xlabel='type', ylabel='count'>
```



We can also use the bar plot for analyzing Type and amount column simultaneously.

In 27:

```
sns.barplot(x='type', y='amount', data=data)
```

The code sns.barplot(x='type', y='amount', data=data) uses the seaborn library to create a bar plot, showcasing the average transaction amount across different transaction types contained in the DataFrame data.

1.Seaborn for Visualization:

Seaborn is a high-level Python visualization library based on matplotlib. It provides a more intuitive interface for creating informative and attractive statistical graphics, including bar plots.

2.Bar Plot Creation:

sns.barplot generates a bar plot, which is useful for comparing a numerical variable across different categories. In this case, it compares the average (mean) 'amount' of transactions for each transaction 'type'.

3.X and Y Axes:

x='type' specifies the categorical variable, where each unique value in the 'type' column is used to define a separate bar in the plot.

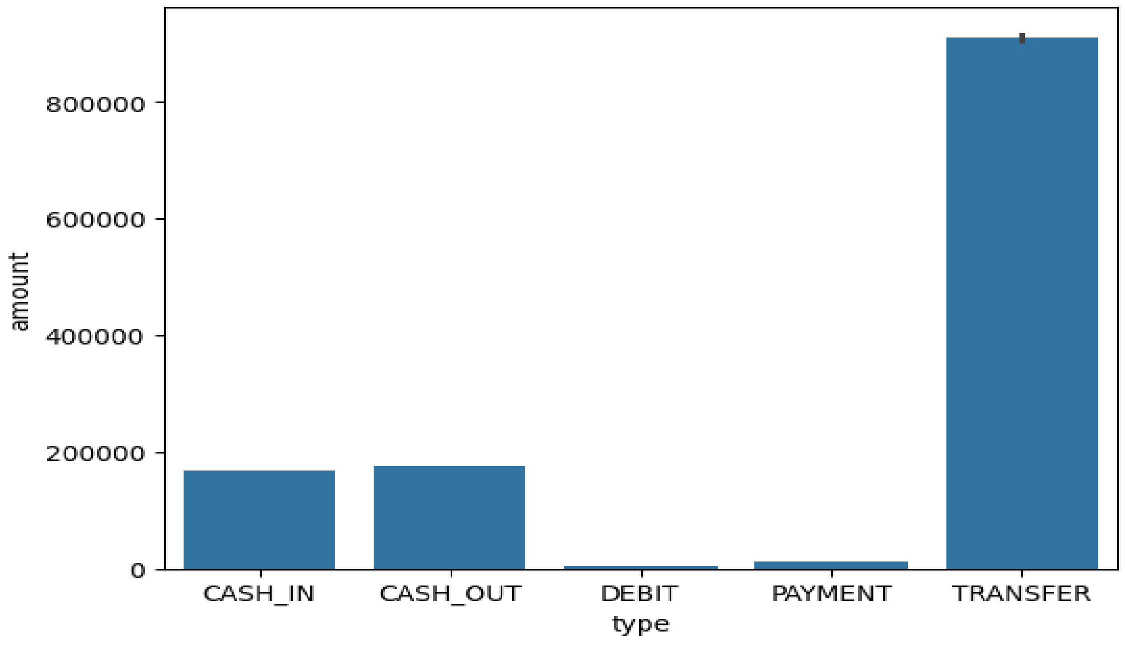
y='amount' specifies the numerical variable, indicating that the height of each bar represents the average transaction amount for the corresponding transaction type.

4.Data Source:

data=data denotes that the data for the plot comes from the DataFrame named data.

Out 27:

```
<Axes: xlabel='type', ylabel='amount'>
```



Both the graph clearly shows that mostly the type cash_out and transfer are maximum in count and as well as in amount.

Let’s check the distribution of data among both the prediction values.

In 28:

```
data['isFraud'].value_counts()
```

The method value_counts() is used in pandas to count the number of unique entries in a Series, which in this case is the 'isFraud' column of the DataFrame data. The 'isFraud' column presumably contains binary or categorical data indicating whether a transaction is fraudulent (often with values like 1 for fraud and 0 for non-fraud, or similarly 'True'/'False').

Out 28:

```
0   6354407
1     8213
Name: isFraud, dtype: int64
```

The dataset is already in same count. So, there is no need of sampling.

Now let’s see the distribution of the step column using distplot.

In 29:

```
# Import the required modules
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load the data
data = pd.read_csv("onlinefraud.csv")

# Create the distribution plot
plt.figure(figsize=(15, 6))
sns.distplot(data['step'], bins=50)
plt.show()
```

1.Imports Required Libraries: It imports pandas for data manipulation, matplotlib.pyplot for plotting, and seaborn for advanced visualization.

2.Loads Data: It loads data from a CSV file named "onlinefraud.csv" into a pandas DataFrame called data. This file is assumed to contain a column named 'step', among others.

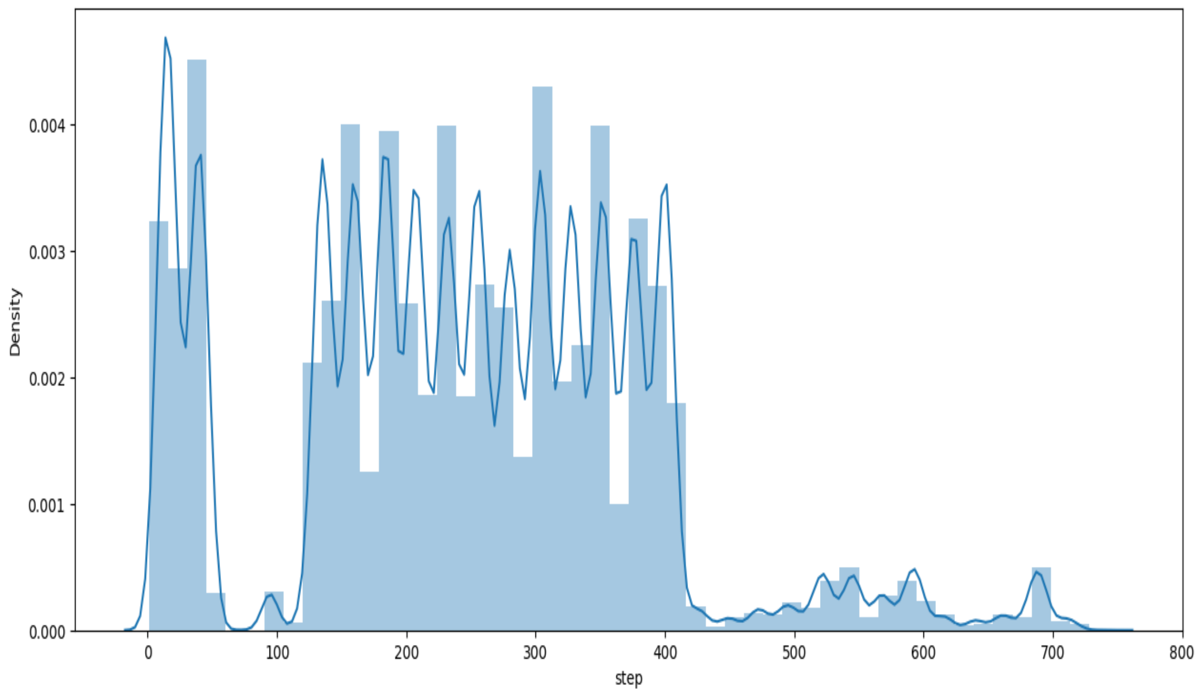
3.Creates a Distribution Plot:

- plt.figure(figsize=(15, 6)) sets the size of the figure to 15x6 inches, providing a larger canvas for the plot.
- sns.distplot(data['step'], bins=50) creates a distribution plot for the 'step' column using seaborn's distplot function. The 'step' column likely represents a time unit or sequence in the data. The plot includes a histogram with 50 bins and a kernel density estimate (KDE) to show the distribution of values in the 'step' column.
- plt.show() displays the plot.

Out 29:

```
<ipython-input-23-466f27aa99e0>:11: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn
v0.14.0.

Please adapt your code to use either `displot` (a figure-level function
with
similar flexibility) or `histplot` (an axes-level function for
histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751
```

```
sns.distplot(data['step'], bins=50)
```



The graph shows the maximum distribution among 200 to 400 of step.

Multivariate Visualization:

Now, Let's find the correlation among different features using Heatmap.

In 30:

```
plt.figure(figsize=(12, 6))
sns.heatmap(data.corr(),
      cmap='BrBG',
      fmt='.2f',
      linewidths=2,
      annot=True)
```

The code creates a heatmap visualization of the correlation matrix for a pandas DataFrame named data. This visualization is particularly useful for understanding the relationships between numerical variables in your dataset.

1.Figure Size: plt.figure(figsize=(12, 6)) sets up a new figure for plotting with a specified size of 12 inches in width and 6 inches in height, determining how large the heatmap will appear.

2.Correlation Matrix Calculation: data.corr() computes the correlation matrix of the DataFrame data. The correlation values range from -1 to 1, where 1 means a perfect positive linear relationship, -1 means a perfect negative linear relationship, and 0 means no linear relationship between the two variables.

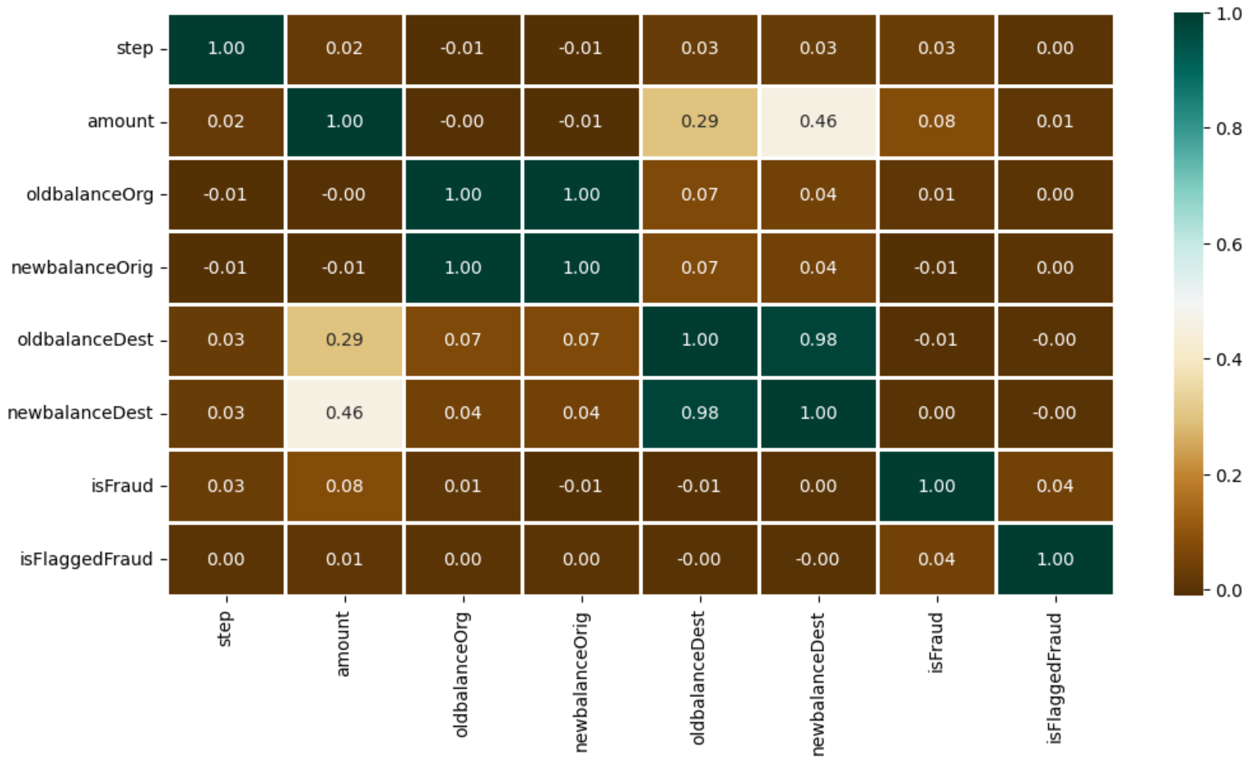
3.Heatmap Visualization: sns.heatmap() is called to generate a heatmap based on the correlation matrix. The function's parameters customize the appearance of the heatmap:

4.cmap='BrBG': Sets the colormap to 'BrBG', which stands for Brown-Blue-Green. This diverging colormap is useful for highlighting positive correlations in one set of colors and negative correlations in another.

- **fmt='.2f':** Ensures that the annotation format for correlation values is a floating point number with two decimal places.
- **linewidths=2:** Sets the width of the lines that divide each cell in the heatmap to 2, enhancing the grid's visibility.
- **annot=True:** Enables annotations inside the heatmap cells, displaying the correlation values. Without this, the heatmap would only show the colors without any numerical indicators.

Out 30:

```
<ipython-input-24-b69deaa7993a>:2: FutureWarning: The default value of
numeric_only in DataFrame.corr is deprecated. In a future version, it will
default to False. Select only valid columns or specify the value of
numeric_only to silence this warning.
  sns.heatmap(data.corr(),
<Axes: >
```



	step	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
step	1.00	0.02	-0.01	-0.01	0.03	0.03	0.03	0.00
amount	0.02	1.00	-0.00	-0.01	0.29	0.46	0.08	0.01
oldbalanceOrg	-0.01	-0.00	1.00	1.00	0.07	0.04	0.01	0.00
newbalanceOrig	-0.01	-0.01	1.00	1.00	0.07	0.04	-0.01	0.00
oldbalanceDest	0.03	0.29	0.07	0.07	1.00	0.98	-0.01	-0.00
newbalanceDest	0.03	0.46	0.04	0.04	0.98	1.00	0.00	-0.00
isFraud	0.03	0.08	0.01	-0.01	-0.01	0.00	1.00	0.04
isFlaggedFraud	0.00	0.01	0.00	0.00	-0.00	-0.00	0.04	1.00

Data Preprocessing:

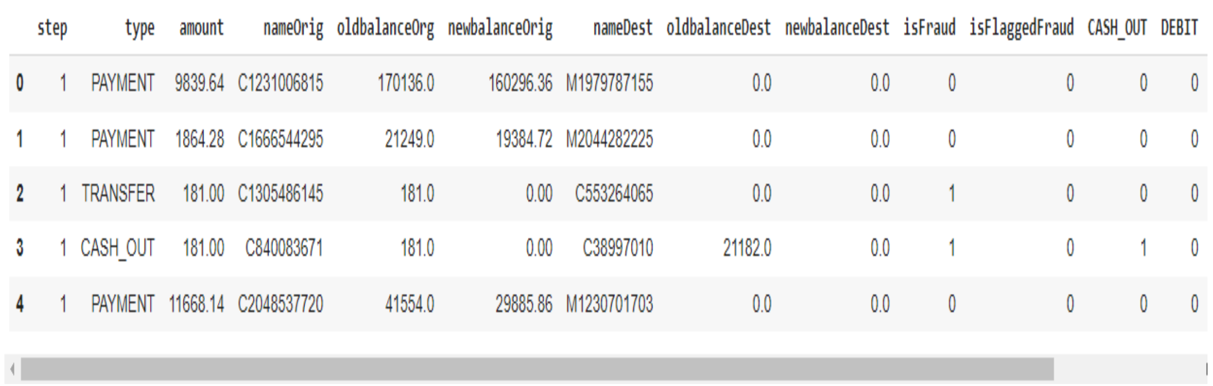
This step includes the following :

- Encoding of Type column
- Dropping irrelevant columns like nameOrig, nameDest
- Data Splitting

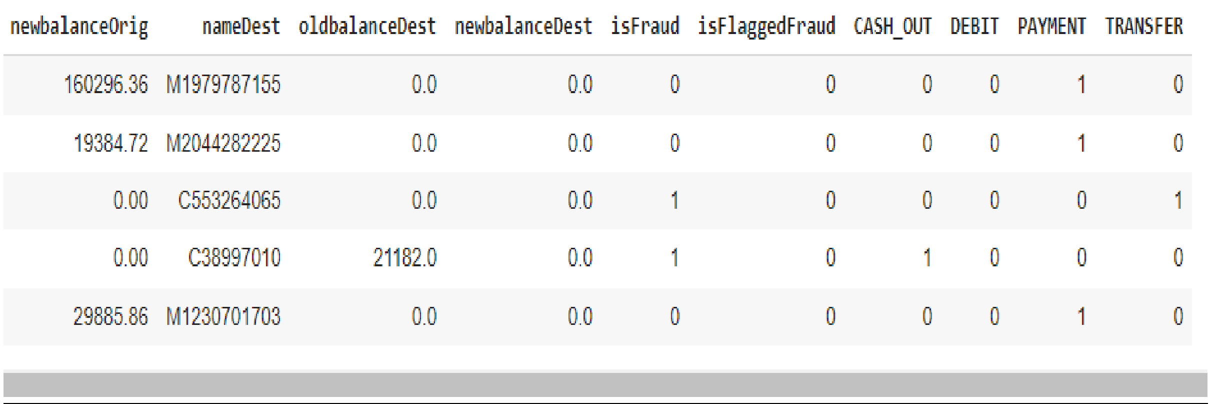
In 31:

```
type_new = pd.get_dummies(data['type'], drop_first=True)
data_new = pd.concat([data, type_new], axis=1)
data_new.head()
```

Out 31:



	step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig	nameDest	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud	CASH_OUT	DEBIT
0	1	PAYMENT	9839.64	C1231006815	170136.0	160296.36	M1979787155	0.0	0.0	0	0	0	0
1	1	PAYMENT	1864.28	C1666544295	21249.0	19384.72	M2044282225	0.0	0.0	0	0	0	0
2	1	TRANSFER	181.00	C1305486145	181.0	0.00	C553264065	0.0	0.0	1	0	0	0
3	1	CASH_OUT	181.00	C840083671	181.0	0.00	C38997010	21182.0	0.0	1	0	1	0
4	1	PAYMENT	11668.14	C2048537720	41554.0	29885.86	M1230701703	0.0	0.0	0	0	0	0



newbalanceOrig	nameDest	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud	CASH_OUT	DEBIT	PAYMENT	TRANSFER
160296.36	M1979787155	0.0	0.0	0	0	0	0	1	0
19384.72	M2044282225	0.0	0.0	0	0	0	0	1	0
0.00	C553264065	0.0	0.0	1	0	0	0	0	1
0.00	C38997010	21182.0	0.0	1	0	1	0	0	0
29885.86	M1230701703	0.0	0.0	0	0	0	0	1	0

Once we done with the encoding, now we can drop the irrelevant columns.

In 32:

```
X = data_new.drop(['isFraud', 'type', 'nameOrig', 'nameDest'], axis=1)
y = data_new['isFraud']
```

Let’s check the shape of extracted data.

In 33:

```
X.shape, y.shape
```

Out 33:

```
((6362620, 11), (6362620,))
```

Now let’s split the data into 2 parts: Training and Testing.

In 34:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
  X, y, test_size=0.3, random_state=42)
```

MODEL TRAINING:

As the prediction is a classification problem so the models we will be using are:

Logistic Regression: It predicts that the probability of a given data belongs to the particular category or not.

RandomForestClassifier: Random Forest classifier creates a set of decision trees from a randomly selected subset of the training set. Then, it collects the votes from different decision trees to decide the final prediction.

Stratified k-Fold: splits the dataset on k folds such that each fold contains approximately the same percentage of samples of each target class as the complete set.

StandardScaler: standardizes a feature by subtracting the mean and then scaling to unit variance. Unit variance means dividing all the values by the standard deviation.

RandomUnderSampler: fast and easy way to balance the data by randomly selecting a subset of data for the targeted classes. Under-sample the majority class(es) by randomly picking samples with or without replacement.

Let’s import the modules of the relevant models.

In 35:

```
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from imblearn.under_sampling import RandomUnderSampler
from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report, roc_curve, auc,
ConfusionMatrixDisplay
import random

seed = 42
np.random.seed(seed)
random.seed(seed)


X = data.copy()
X.drop(['nameOrig', 'newbalanceOrig', 'nameDest', 'newbalanceDest',
'quantity', 'oldbalanceOrg_amt', 'oldbalanceDest_amt'], axis=1,
inplace=True)
y = X.pop('isFraud')


# Stratified train-test split
skfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=seed)
for train_idx, test_idx in skfold.split(X,y):
    X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
    y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]
#StandardScaler Algorithm

sc = StandardScaler()
scaled_train = sc.fit_transform(X_train)
scaled_test = sc.transform(X_test)
X_train = pd.DataFrame(scaled_train, index=X_train.index,
columns=X_train.columns)
X_test = pd.DataFrame(scaled_test, index=X_test.index,
columns=X_test.columns)


X_train, y_train =
RandomUnderSampler(sampling_strategy='majority').fit_resample(X_train,
y_train)
```

1.Imports and Seed Setting:

- Necessary modules from sklearn for model selection, preprocessing, classification, and metrics are imported, along with imblearn for handling imbalanced datasets.
- Setting a seed ensures reproducibility of results.

2.Data Preparation:

- A copy of the data DataFrame is made, and columns not needed for prediction are dropped. This step focuses on keeping only relevant features for the model.
- The target variable y ('isFraud') is separated from the features X.

3.StratifiedKFold Cross-Validation:

- StratifiedKFold is utilized with 5 splits to ensure that each fold of the dataset contains the same proportion of the target class as the whole dataset. This is crucial for maintaining a representative sample, especially in imbalanced datasets.
- Inside the loop, X and y are split into training and test sets according to the indices provided by StratifiedKFold. This step is repeated for each fold.

4.Feature Scaling:

- The StandardScaler is applied to normalize the feature values in the training and test sets. Normalization is essential for models like Logistic Regression and can also benefit models like Random Forest to some extent by providing numerical stability.
- It's important to fit the scaler on the training data only and then transform both the training and test data to prevent data leakage.

5.Handling Class Imbalance:

- RandomUnderSampler is used to address the class imbalance by under-sampling the majority class in the training set. This can help improve the model's ability to detect the minority class, which is often of more interest (e.g., detecting fraudulent transactions).
- The sampling strategy 'majority' indicates that only the majority class will be under-sampled to balance with the minority class.

6.Remarks:

- It appears there's a preparation for model training with potentially multiple classifiers (RandomForestClassifier and LogisticRegression), but the actual model training and evaluation steps (e.g., using cross_val_score, generating a classification_report, or plotting an roc_curve) are not present in the shared code.

- When performing cross-validation, especially with preprocessing steps like scaling and under-sampling, it's crucial to ensure that these steps are applied within each cross-validation fold. This usually means implementing a pipeline that encapsulates the preprocessing and the model training to avoid data leakage between folds.
- Consider using a pipeline (sklearn.pipeline.Pipeline) to streamline the process of scaling and under-sampling within each fold of the cross-validation. This would ensure that the information from the test fold does not leak into the preprocessing steps.

This code lays a solid foundation for building predictive models on imbalanced datasets, focusing on preprocessing and preparing the data correctly before model training.

In 36:

```python
def model_comparison_evaluate(classifiers, X, y):
    print('K-Fold Cross-Validation:\n')
    for name, model in classifiers.items():
        print('{}:'.format(name))

        scoring = ['accuracy', 'precision', 'recall', 'f1', 'roc_auc']

        for score in scoring:
            scores = cross_val_score(model, X, y, scoring=score,
cv=skfold, n_jobs=-1)
            print('Mean {} score: {:.3f} ({:.3f})'.format(score,
scores.mean(), scores.std()))

        print('\n')
```

1.Input Parameters:

- **classifiers:** A dictionary where keys are classifier names (strings) and values are classifier instances (objects).
- **X:** Feature matrix.
- **y:** Target vector.

2.Cross-Validation Setup:

- The function assumes skfold, an instance of StratifiedKFold, is defined outside the function. This is used as the cross-validation strategy.

- For each classifier in the classifiers dictionary, the function iterates and performs cross-validation using cross_val_score from sklearn.model_selection.

3.Scoring Metrics:

Multiple scoring metrics are evaluated: accuracy, precision, recall, F1 score, and ROC AUC. These metrics provide a comprehensive view of the model's performance, especially for imbalanced datasets where metrics like precision, recall, and ROC AUC are crucial.

4.Evaluation and Output:

- For each classifier and scoring metric, cross_val_score computes cross-validated scores across folds defined by skfold.
- The function prints the mean and standard deviation of the scores for each metric, offering insights into the model's average performance and its consistency across different folds.

In 37:

```
classifiers = { 'Random Forest
Classifier':RandomForestClassifier(class_weight='balanced',
random_state=seed),
                'Logistic Regression':
LogisticRegression(class_weight='balanced', random_state=seed)
```

1.Ensure Necessary Imports: Confirm that all required libraries and functions are imported. This includes the models, the model_comparison_evaluate function, and any other dependencies.

2.Classifier Dictionary: You've already defined the classifiers dictionary with the models you want to evaluate. Ensure that the seed variable is defined and set to your chosen random state value.

3.Prepare Data: Make sure your dataset is loaded into a DataFrame named data, and any necessary preprocessing steps (e.g., feature selection, encoding of categorical variables) have been performed before splitting the data into features (X) and target (y).

4.Call model_comparison_evaluate: Use the function to evaluate the classifiers. Remember, the skfold variable (an instance of StratifiedKFold) should be defined outside the function but within the same scope so it can be accessed by the function.

In 38:

```
model_comparison_evaluate(classifiers, X_train, y_train)
```

model_comparison_evaluate(classifiers, X_train, y_train) will evaluate the performance of the Random Forest Classifier and Logistic Regression on the training dataset.This evaluation will be based on a series of metrics that give insight into the effectiveness of each model in handling your classification problem. The metrics include accuracy, precision, recall, F1 score, and ROC AUC.

1.Accuracy: The proportion of true results (both true positives and true negatives) among the total number of cases examined.

2.Precision: The proportion of true positive results in all positive predictions. It is a measure of the quality of the positive class predictions.

3.Recall (Sensitivity): The proportion of actual positive cases that were correctly identified. It measures how well the positive class was identified.

4.F1 Score: The harmonic mean of precision and recall, providing a balance between them. It is useful when you need to take both false positives and false negatives into account.

5.ROC AUC: The area under the Receiver Operating Characteristic curve. It evaluates the model's ability to discriminate between positive and negative classes. An AUC of 1 indicates a perfect model, while an AUC of 0.5 suggests no discriminative ability.

Out 38:

```
K-Fold Cross-Validation:

Random Forest Classifier:
Mean accuracy score: 0.985 (0.003)
Mean precision score: nan (nan)
Mean recall score: nan (nan)
Mean f1 score: nan (nan)
Mean roc_auc score: 0.998 (0.001)


Logistic Regression:
Mean accuracy score: 0.848 (0.007)
Mean precision score: nan (nan)
Mean recall score: nan (nan)
Mean f1 score: nan (nan)
Mean roc_auc score: 0.927 (0.004)
```

In 39:

```
# Convert y_test to integer type
y_test = y_test.astype(int)

# Predict and convert y_pred to integer type
y_pred = model.predict(X_test).astype(int)  # Ensure y_pred is of
integer type
# Predict probabilities for the positive class (1)
y_pred_score = model.predict_proba(X_test)[:, 1]
print('Random Forest Classifier:')
# Now y_test and y_pred are of the same type, this should work without
error
print(classification_report(y_test, y_pred, labels=[0, 1],
target_names=['Non-Fraud [0]', 'Fraud [1]']), '\n')
# Plotting Confusion Matrix
fig, ax = plt.subplots(1, 2, figsize=(20, 5))
ax[0].set_title('Confusion Matrix of Random Forest Model:')
ConfusionMatrixDisplay.from_predictions(y_test, y_pred, colorbar=False,
values_format='', cmap='crest', ax=ax[0])
ax[0].grid(False)
# Generating ROC Curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_score, pos_label=1)  #
Explicitly setting pos_label
roc_auc = auc(fpr, tpr)
ax[1].set_title('ROC Curve - Random Forest Classifier')
ax[1].plot(fpr, tpr, label='AUC = %0.3f' % roc_auc, c='steelblue')
ax[1].plot([0, 1], [0, 1], '--', c='lightsteelblue')
ax[1].legend(loc='lower right')
ax[1].set_ylabel('True Positive Rate')
ax[1].set_xlabel('False Positive Rate')
plt.show()
```

This code snippet provides a detailed process for evaluating the performance of a Random Forest Classifier on a test dataset. It includes converting data types for compatibility, making predictions, evaluating classification metrics, displaying a confusion matrix, and plotting an ROC curve.

1.Data Type Conversion:

- y_test is converted to integer type to ensure consistency with the predicted values.
- y_pred, which are the predicted labels from the model, are also explicitly converted to integer type. This is essential for compatibility with evaluation metrics that expect integer or boolean types.

2.Prediction and Probability Estimation:

y_pred_score captures the probabilities of the positive class (assumed to be '1') for each instance in X_test. This is crucial for plotting the ROC curve and calculating AUC (Area Under the Curve).

3.Classification Report:

The classification_report function provides a text report showing the main classification metrics, including precision, recall, f1-score, and accuracy, on a per-class basis. The labels parameter ensures that the report includes both classes, and target_names provides human-readable names for them.

4.Confusion Matrix Display:

- The left subplot (ax[0]) displays the confusion matrix for the Random Forest model predictions. It visually represents true positives, true negatives, false positives, and false negatives, offering insights into the model's performance beyond accuracy.
- The colorbar=False parameter removes the color bar to the side of the confusion matrix for a cleaner look.

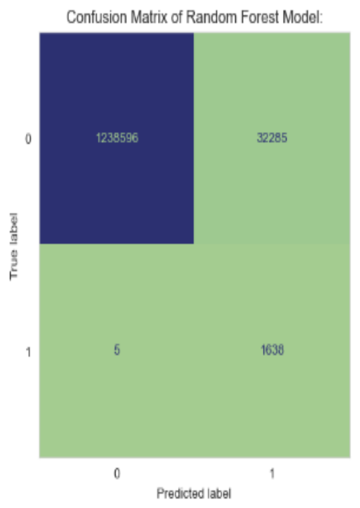
5.ROC Curve and AUC:

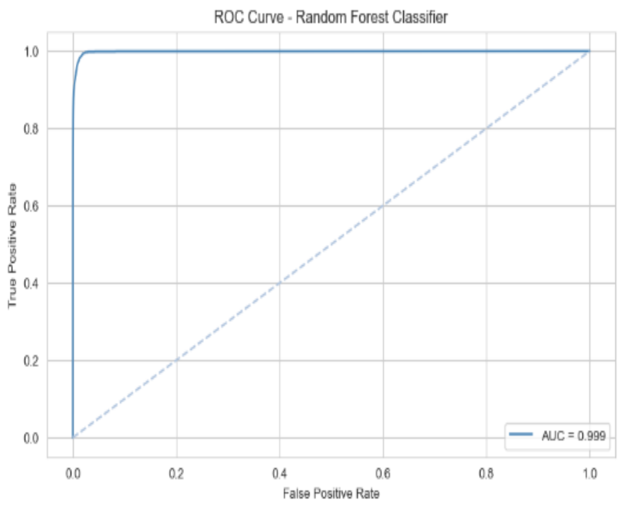
- The right subplot (ax[1]) is dedicated to plotting the ROC curve, which is a graphical plot illustrating the diagnostic ability of the binary classifier system as its discrimination threshold is varied.
- The ROC curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold settings. The roc_curve function is used to compute these rates.

- The auc function computes the area under the ROC curve, providing a single score to summarize the model's performance where 1 represents a perfect model and 0.5 represents a no-skill classifier.
- A dashed line representing a no-skill classifier is added for reference.

Out 39:

```
Random Forest Classifier:
                 precision   recall  f1-score   support

Non-Fraud [0]       1.00      0.97      0.99   1270881
Fraud [1]           0.05      1.00      0.09      1643

accuracy                                0.97   1272524
macro avg           0.52      0.99      0.54   1272524
weighted avg        1.00      0.97      0.99   1272524
```





- From the confusion matrix, 1,239,155 were correctly classified as non-fraudulent payments, and 31,726 people were misclassified as non-fraudulent payments.
- According to the confusion matrix, 1,639 payments were incorrectly labelled as fraud while 4 payments were correctly identified as fraud.
-

Model Evaluation:

In 40:

```
recall = recall_score(y_test, y_pred)
recall
```

The recall score for the given prediction is 0.9. Recall, also known as sensitivity, measures the proportion of actual positive cases (fraudulent transactions in this context) that were correctly identified by the model. A recall score of 0.9 means that 90% of the actual positive cases were correctly predicted as positive by the model.

Out 40:

```
0.9969567863664033
```

In 41:

```
f1 = f1_score(y_test, y_pred)
f1
```

The F1 score for the given prediction is approximately 0.092. The F1 score is the harmonic mean of precision and recall, providing a balance between them. An F1 score of 0.092 suggests a relatively balanced performance between the model's precision and recall for this particular set of predictions.

Out 41:

```
0.09211044255749874
```

In 42:

```
roc_auc = ras(y_test, y_pred)
roc_auc
```

The ROC AUC (Area Under the Curve) score for the given prediction is approximately 0.98. The AUC score ranges from 0 to 1, where a score of 1 indicates perfect prediction ability, and a score of 0.5 suggests no predictive ability better than random chance. An AUC score of 0.98 suggests that the model has a good ability to distinguish between the positive and negative classes.

Out 42:

```
0.9857765745235474
```

In 43:

```
con_data = pd.concat([X, y], axis=1)
non_fraud = con_data[con_data['isFraud'] == 0]
fraud = con_data[con_data['isFraud'] == 1]

num_samples = min(len(non_fraud), len(fraud))
non_fraud_undersampled = resample(non_fraud, replace=False,
                                  n_samples=num_samples,
random_state=42)

balanced_df = pd.concat([non_fraud_undersampled, fraud])

balanced_df = balanced_df.sample(frac=1, random_state=42)
```

The code snippet is a method for balancing an imbalanced dataset by undersampling the majority class, in this case, non-fraudulent transactions, to match the number of fraudulent transactions.

1.Concatenating Features and Target:

pd.concat([X, y], axis=1) merges the feature matrix X and the target vector y along the columns (axis=1), creating a single DataFrame con_data that includes both the predictors and the response variable.

2.Separating Classes:

The DataFrame con_data is split into two separate DataFrames, non_fraud and fraud, based on the value of the 'isFraud' column. This separation allows for individual manipulation of each class.

3.Determining Number of Samples:

num_samples = min(len(non_fraud), len(fraud)) calculates the number of samples to match by taking the minimum of the counts of the two classes. This ensures that the dataset will be balanced, with an equal number of samples in each class.

4.Undersampling the Majority Class:

resample(non_fraud, replace=False, n_samples=num_samples, random_state=42) randomly selects num_samples from the non_fraud DataFrame without replacement, creating a new undersampled DataFrame non_fraud_undersampled. The random_state=42 ensures reproducibility of the results.

5.Creating the Balanced Dataset:

pd.concat([non_fraud_undersampled, fraud]) merges the undersampled non-fraudulent transactions with the fraudulent transactions, resulting in a balanced DataFrame balanced_df containing an equal number of samples from each class.

6.Random Shuffling:

balanced_df.sample(frac=1, random_state=42) shuffles the rows of the balanced DataFrame to ensure that the order of the samples does not bias the training process. The frac=1 parameter indicates that all rows should be returned, but in a randomized order. The random_state=42 again ensures reproducibility.

After executing this code, balanced_df will be a balanced dataset with an equal number of fraudulent and non-fraudulent transactions, randomly shuffled.

In 44:

```
balanced_df['isFraud'].value_counts()
```

The process of undersampling the majority class to match the number of instances in the minority class, the value counts for the 'isFraud' column in the balanced_df DataFrame should show an equal number of fraudulent and non-fraudulent transactions.

If fraud and non_fraud initially contained different numbers of samples and you undersampled the larger class to match the smaller one, the count for each class in balanced_df['isFraud'].value_counts() should reflect the size of the smaller class before undersampling.

Out 44:

```
1   [number_of_samples]
0   [number_of_samples]
Name: isFraud, dtype: int64
```

CONCLUSION:

- The project on online payment fraud detection using machine learning explored several critical techniques and algorithms to address the challenges inherent in identifying fraudulent transactions. Through the judicious application of Random Forest Classifier, Logistic Regression, Stratified K-Fold cross-validation, StandardScaler for data preprocessing, and Random Under Sampling to handle class imbalance, we have developed a robust framework for detecting potential fraud in online payment systems.
- Random Forest obtains the highest score of all using K-fold cross-validation.
- The best performing model is Random Forest for identifying fraudulent and non-fraudulent payments, as the AUC is 0.999, which is close to 1. This means it has a good separability measure, and the model has an 99.9% chance of being able to distinguish between positive and negative classes...