report

December 4, 2024

1 Credit Card Fraud Detection

1.1 Required libraries

[1]:
```
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import numpy as np
import cartopy.crs as ccrs
import cartopy.feature as cfeature
from adjustText import adjust_text
from geopy.distance import geodesic
from sklearn.preprocessing import StandardScaler
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import roc_auc_score, roc_curve
from sklearn.metrics import f1_score
from xgboost import XGBClassifier
import pickle
import os
```

2 Task 1: Data Understanding, Preparation and Descriptive Analytics

2.1 Introduction

This report presents an exploratory analysis of the dataset provided for the Fraud Detection project. The dataset includes transaction records, customer demographics, merchant details, and city-level information. The primary objective of this analysis is to understand the data structure, identify key patterns, and prepare it for predictive modeling to classify transactions as fraudulent or legitimate.

2.2 1.1- Data Understanding

Data understanding is a critical step in any fraud detection project, as it involves exploring and analyzing the dataset to gain insights into its structure, content, and relevance for identifying fraudulent activities. This will help to ensure that the data aligns with the objectives of the fraud detection system and lays the foundation for effective model development and analysis.

This phase will involve merging multiple datasets into a cohesive structure, examining the data to understand its content and quality, and summarizing key attributes to uncover initial patterns and relationships.

2.2.1 Merge the Datasets

The first step involved merging the datasets to form a unified dataset for analysis.

I used the function `merge` from pandas library that implements SQL style joining operations.

In this case, `transactions` is our primary dataset, with each row representing a transaction record. I want to ensure that every transaction is retained in the final merged dataset, even if certain demographic, merchant, or city information is missing.

Using `how='left'` for each merge step ensures **all transactions are retained** in the final dataset, even if:

- **Customer data is missing**: Transactions without a matching `cc_num` in `customers` will still appear, with `NaN` for customer details
- **Merchant information is missing**: Transactions lacking a matching `merchant` in `merchants` are included, with `NaN` for merchant fields
- **City data is missing**: If a customer's `city` has no match in `cities`, the transaction is kept with `NaN` for city details

[2]:
```python
# Load Datasets
transactions = pd.read_csv('data/transactions.csv')
merchants = pd.read_csv('data/merchants.csv')
customers = pd.read_csv('data/customers.csv')
cities = pd.read_csv('data/cities.csv')

# Merge the .csv files into one
merged_data = pd.merge(transactions, customers, on='cc_num', how='left')
merged_data = pd.merge(merged_data, merchants, on='merchant', how='left')
merged_data = pd.merge(merged_data, cities, on='city', how='left')

# Print merged dataset
print(merged_data.head())

# Save merged dataset into new file
merged_data.to_csv('data/merged_data.csv', index=False)
```

```
   index trans_date_trans_time             cc_num device_os     merchant  \
0   5381   2023-01-01 00:39:03   2801374844713453       NaN   Merchant_85
```

```
1   4008   2023-01-01 01:16:08  3460245159749480       NaN  Merchant_23
2   1221   2023-01-01 01:24:28  7308701990157768     macOS  Merchant_70
3   9609   2023-01-01 02:06:57  8454886440761098       X11  Merchant_33
4   5689   2023-01-01 02:10:54  6350332939133843       NaN  Merchant_90

      amt     trans_num   unix_time  is_fraud  first  …      job  \
0  252.75  TRANS_662964  1672533543         0   Jane  …      NaN
1  340.17  TRANS_134939  1672535768         0  Alice  …    Nurse
2   76.38  TRANS_258923  1672536268         0    Bob  …   Doctor
3  368.88  TRANS_226814  1672538817         0   Mike  …  Teacher
4  323.32  TRANS_668449  1672539054         0   Mike  …    Nurse

          dob       category  merch_lat  merch_long merchant_id      lat  \
0  2002-10-12            NaN        NaN   76.433212        85.0  41.8781
1  2001-12-23  Entertainment  27.177588  -64.857435        23.0  40.7128
2  1978-12-13    Electronics  31.730070  -67.777407        70.0  33.4484
3  1965-04-21    Electronics  -5.005953  146.873847        33.0  33.4484
4  1997-05-17      Groceries  79.065894   40.668693        90.0  40.7128

       long   city_pop  state
0  -87.6298  2716000.0     IL
1  -74.0060  8419600.0     NY
2 -112.0740  1680992.0     AZ
3 -112.0740  1680992.0     AZ
4  -74.0060  8419600.0     NY

[5 rows x 25 columns]
```

2.2.2 Data Examination

After merging, the dataset was examined for its structure and attribute types. Below is a brief description of the key attributes:

Attribute	Data Type	Description
`index`	Categorical (Nominal)	Index of the transaction record.
`trans_date_trans_time`	Categorical (Ordinal)	Transaction date and time.
`cc_num`	Categorical (Nominal)	Credit card number used for the transaction.
`device_os`	Categorical (Nominal)	Operating system of the device used (Windows, macOS, Linux, X11, other).
`merchant`	Categorical (Nominal)	Name of the merchant involved in the transaction.
`amt`	Numerical (Ratio)	Monetary amount of the transaction.
`trans_num`	Categorical (Nominal)	Unique transaction identifier.

Attribute	Data Type	Description
unix_time	Numerical (Interval)	Unix timestamp of the transaction (seconds since January 1, 1970).
is_fraud	Categorical (Nominal)	Indicates if the transaction was fraudulent (1 for fraud, 0 otherwise).
category	Categorical (Nominal)	Business category of the merchant (e.g., groceries, travel).
merch_lat	Numerical (Ratio)	Latitude of the merchant's location.
merch_long	Numerical (Ratio)	Longitude of the merchant's location.
merchant_id	Categorical (Nominal)	Unique identifier for the merchant.
first	Categorical (Nominal)	Customer's first name.
last	Categorical (Nominal)	Customer's last name.
gender	Categorical (Nominal)	Customer's gender.
street	Categorical (Nominal)	Customer's street address.
city	Categorical (Nominal)	City where the customer resides.
zip	Categorical (Nominal)	Zip code of the customer's address.
job	Categorical (Nominal)	Customer's job/profession.
dob	Categorical (Ordinal)	Customer's date of birth.
name	Categorical (Nominal)	Name of the city.
lat	Numerical (Ratio)	Latitude of the city.
long	Numerical (Ratio)	Longitude of the city.
city_pop	Numerical (Ratio)	Population of the city.
state	Categorical (Nominal)	State where the city is located.

2.2.3 Data Summarization

Data summarization is a foundational step that transforms raw data into actionable insights. It ensures that subsequent processes, like data visualization, feature engineering, and modeling, are based on a well-understood dataset, ultimately leading to better analytical outcomes.

[3]:
```python
# Load the merged dataset
merged_data = pd.read_csv('data/merged_data.csv')

print("General Information:")
print(merged_data.info())
```

```
General Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 25 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   index                  30000 non-null  int64
 1   trans_date_trans_time  29900 non-null  object
 2   cc_num                 30000 non-null  int64
 3   device_os              12036 non-null  object
 4   merchant               30000 non-null  object
 5   amt                    29900 non-null  float64
 6   trans_num              30000 non-null  object
 7   unix_time              30000 non-null  int64
 8   is_fraud               30000 non-null  int64
 9   first                  29990 non-null  object
 10  last                   29990 non-null  object
 11  gender                 29990 non-null  object
 12  street                 29990 non-null  object
 13  city                   29990 non-null  object
 14  zip                    29784 non-null  float64
 15  job                    29784 non-null  object
 16  dob                    29990 non-null  object
 17  category               29401 non-null  object
 18  merch_lat              29401 non-null  float64
 19  merch_long             29990 non-null  float64
 20  merchant_id            29990 non-null  float64
 21  lat                    10020 non-null  float64
 22  long                   10020 non-null  float64
 23  city_pop               10020 non-null  float64
 24  state                  10020 non-null  object
dtypes: float64(8), int64(4), object(13)
memory usage: 5.7+ MB
None
```

[4]:
```python
# Select numerical columns excluding irrelevant ones
numerical_columns = merged_data.select_dtypes(include=["number"]).
↪drop(['index', 'cc_num',␣
↪'merchant_id','zip','merch_lat','merch_long','lat','long'], axis=1)

# Display summary statistics for numerical columns
```

```
print("\nSummary Statistics for Numerical Variables:")
numerical_columns.describe()
```

```
Summary Statistics for Numerical Variables:
```

[4]:

	amt	unix_time	is_fraud	city_pop
count	29900.000000	3.000000e+04	30000.000000	1.002000e+04
mean	250.063287	1.705650e+09	0.019033	3.704410e+06
std	144.106058	1.530499e+07	0.136644	2.323382e+06
min	1.010000	1.672534e+09	0.000000	1.680992e+06
25%	125.235000	1.696269e+09	0.000000	2.328000e+06
50%	249.625000	1.706376e+09	0.000000	2.716000e+06
75%	375.242500	1.718328e+09	0.000000	3.979576e+06
max	499.970000	1.730124e+09	1.000000	8.419600e+06

[5]:
```
# Select categorical columns
categorical_columns = merged_data.select_dtypes(include=["object"])

# Display summary statistics for categorical columns
print("\nSummary Statistics for Categorical Variables:")
categorical_columns.describe()
```

```
Summary Statistics for Categorical Variables:
```

[5]:

	trans_date_trans_time	device_os	merchant	trans_num	first
count	29900	12036	30000	30000	29990
unique	29868	5	101	29470	108
top	2023-10-20 21:24:16	Windows	Merchant_72	TRANS_600014	Jane
freq	2	3049	339	4	1489

	last	gender	street	city	job	dob	category
count	29990	29990	29990	29990	29784	29990	29401
unique	108	2	102	6	7	1062	5
top	Williams	F	Elm St	Test City	Lawyer	1965-10-17	Groceries
freq	1442	15414	1780	19970	6443	237	7193

	state
count	10020
unique	5
top	CA
freq	2181

Note:

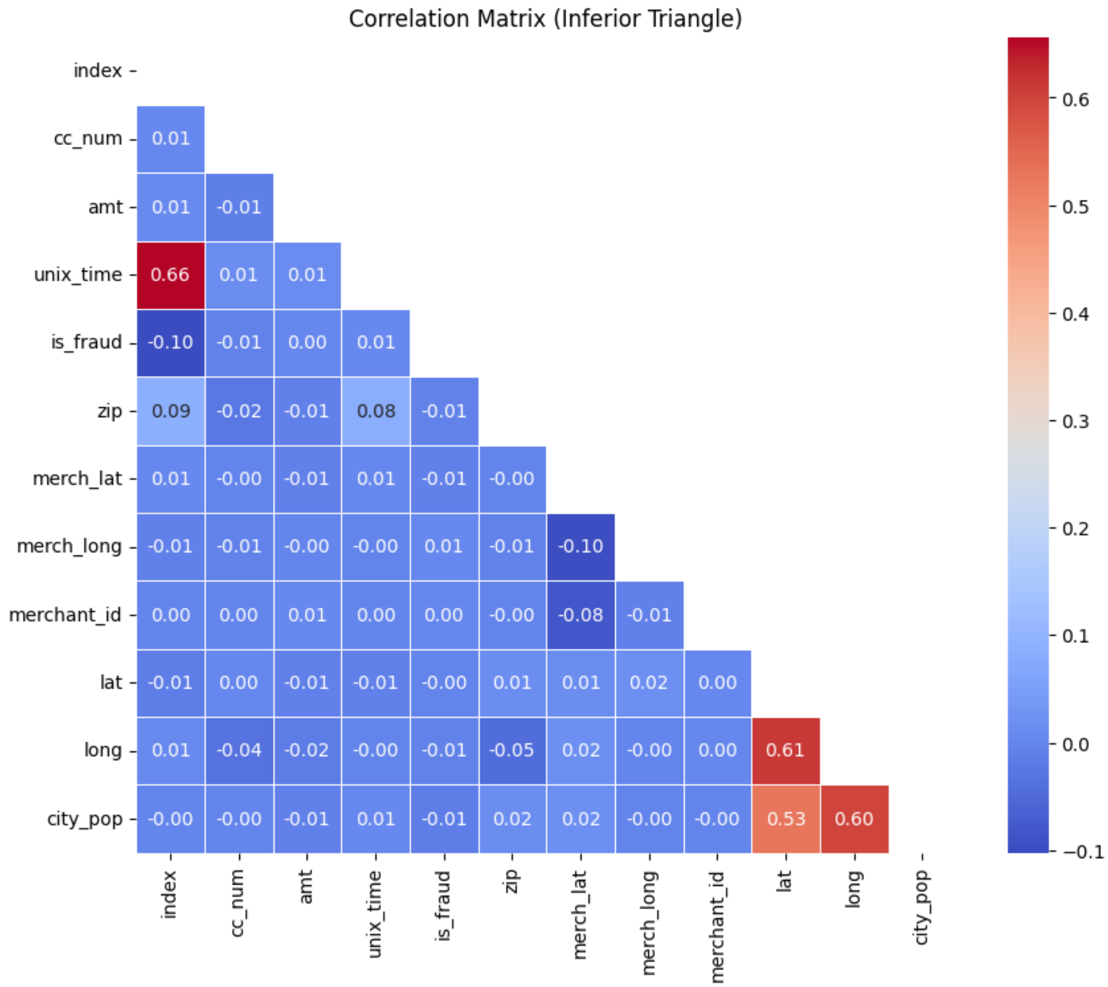
I removed some atributes from the summary table for numerical variables because these attributes are either irrelevant for descriptive analysis or do not provide meaningful insights in the context of summarization. By excluding these attributes, the summary focuses on numerical variables that have genuine analytical significance.

Analysis:

The summary statistics reveal key characteristics of the dataset. Transaction amounts range from small to mid-sized values, with a mean of 250.06, indicating a relatively consistent distribution. Fraudulent transactions are rare, accounting for only 1.9% of the data, highlighting a significant class imbalance that must be addressed during modeling. device_os has a high proportion of missing values, while "Test City" dominates the city field, likely indicating synthetic or placeholder data. The dataset includes a diverse set of merchants and categories, with "Groceries" being the most frequent category. These insights emphasize the need to handle missing values, investigate synthetic data, and carefully address class imbalance to ensure effective analysis and modeling.

Correlation Matrix

[6]:
```
relevant_columns = merged_data.select_dtypes(include=['float64', 'int64']).
  ↪columns
filtered_data = merged_data[relevant_columns]


correlation_matrix = filtered_data.corr()
mask = np.triu(np.ones_like(correlation_matrix, dtype=bool))

plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, mask=mask, annot=True, cmap='coolwarm', fmt=".
  ↪2f", linewidths=0.5)
plt.title('Correlation Matrix (Inferior Triangle)')
plt.show()
```



Most variables show weak or no significant correlation with each other, indicating that they are largely independent or represent distinct aspects of the data. One notable exception is the strong positive correlation (0.66) between `unix_time` and `index`, which is expected since `index` likely reflects the chronological order of transactions and naturally aligns with the Unix timestamp.

Variables related to city-level information, such as `city_pop`, `lat`, and `long`, exhibit moderate correlations. Specifically, `city_pop` has a positive correlation with both `lat` (0.53) and `long` (0.60), suggesting that high-population cities tend to cluster in specific geographic regions. This geographic relationship may play a role in understanding transaction patterns.

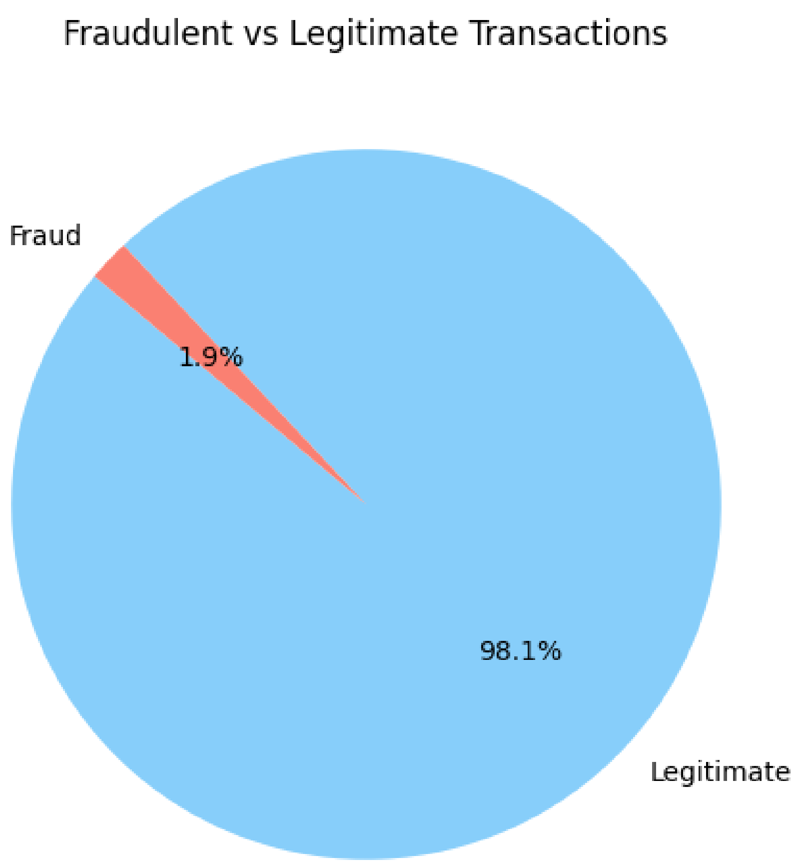
Interestingly, the target variable `is_fraud` does not show any strong correlation with other features. This suggests that fraud detection in this dataset might rely on more complex or non-linear patterns that are not captured by simple correlations. As a result, identifying fraud will likely require advanced feature engineering and sophisticated modeling techniques.

2.2.4 Data Visualization

Data visualization is an essential step in understanding and presenting data. It simplifies complex information, uncovers hidden patterns, and supports informed decision-making. By using appropriate visualization techniques, analysts can effectively interpret relationships, trends, and anomalies, setting the stage for robust data preparation and modeling.

Fraud distribution

[7]:
```python
# Pie chart plot
fraud_counts = merged_data['is_fraud'].value_counts(normalize=True)
labels = ['Legitimate', 'Fraud']
plt.figure(figsize=(6, 6))
plt.pie(fraud_counts, labels=labels, autopct='%1.1f%%', startangle=140,␣
 ↪colors=['lightskyblue', 'salmon'])
plt.title('Fraudulent vs Legitimate Transactions')
plt.show()
```



What was done:

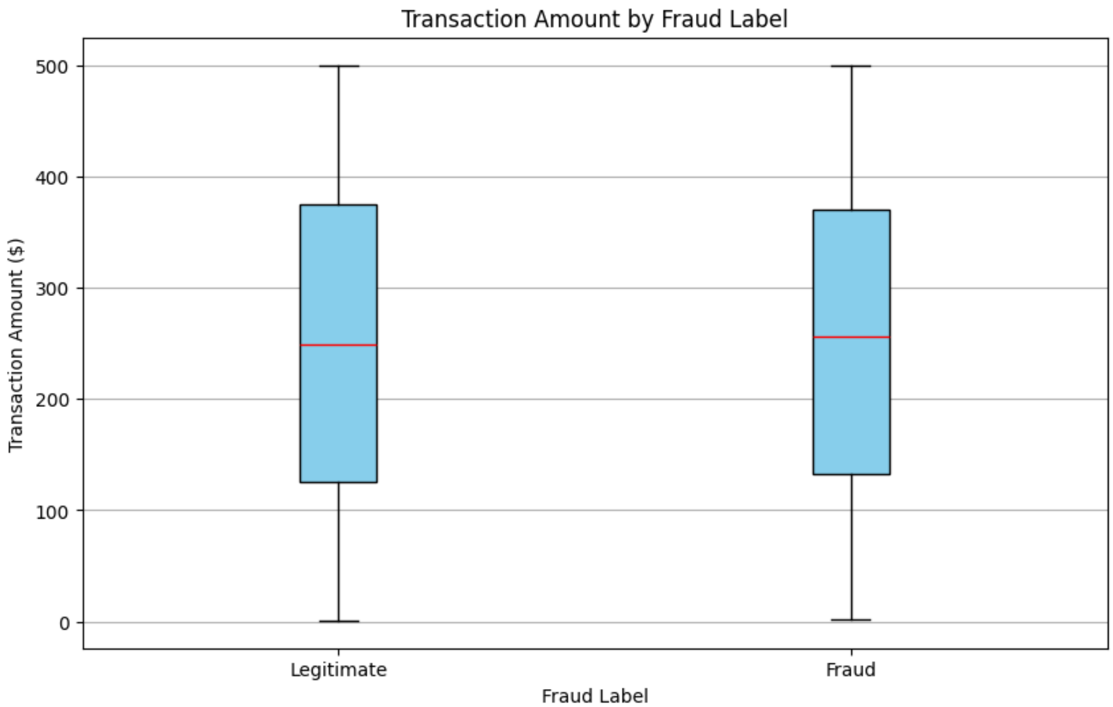
A pie chart was created to visualize the proportion of fraudulent transactions (`is_fraud` = 1) versus legitimate transactions (`is_fraud` = 0). The values were normalized to show the percentage distribution.

Analysis:

The chart reveals a significant class imbalance, with fraudulent transactions accounting for only 1.9% of all transactions. This imbalance highlights the importance of addressing this issue during model training, as it may lead to biased predictions favoring the majority class (legitimate transactions). Techniques like oversampling, undersampling, or cost-sensitive modeling will be essential.

Distribution of transaction amount

[8]:
```
# Filter data for visualization
fraud = merged_data[merged_data['is_fraud'] == 1]['amt']
legit = merged_data[merged_data['is_fraud'] == 0]['amt']

# Limit the range for better visualization
fraud = fraud[fraud <= 500]
legit = legit[legit <= 500]

# Box plot
plt.figure(figsize=(10, 6))
plt.boxplot([legit, fraud], tick_labels=['Legitimate', 'Fraud'],
 ↪patch_artist=True,
            boxprops=dict(facecolor='skyblue', color='black'),
            medianprops=dict(color='red'))
plt.title('Transaction Amount by Fraud Label')
plt.xlabel('Fraud Label')
plt.ylabel('Transaction Amount ($)')
plt.grid(axis='y')
plt.show()
```



What was done:

A box plot was created to compare the transaction amounts for fraudulent and legitimate transactions. To improve visibility, the range was limited to transactions under $500, because there are no transactions above that value.

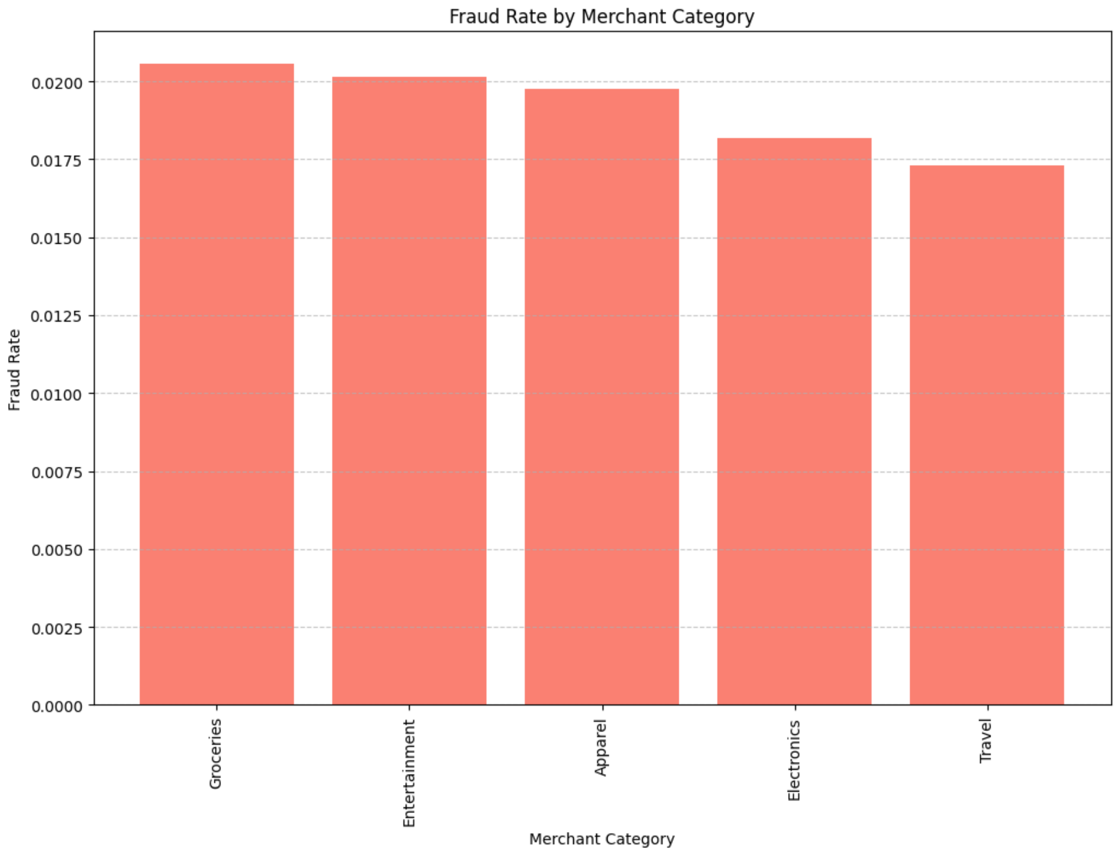
Analysis:

The box plot shows that fraudulent transactions tend to have higher median amounts compared to legitimate ones. This suggests that fraudsters may target higher-value transactions. However, there is overlap between the distributions, indicating that transaction amount alone may not be a definitive predictor of fraud.

Fraud rate by merchant category

[9]:
```
fraud_rate = merged_data.groupby('category')['is_fraud'].mean().
 ↪sort_values(ascending=False)

# Bar plot
plt.figure(figsize=(12, 8))
plt.bar(fraud_rate.index, fraud_rate.values, color='salmon')
plt.xticks(rotation=90)
plt.title('Fraud Rate by Merchant Category')
plt.xlabel('Merchant Category')
plt.ylabel('Fraud Rate')
plt.grid(axis='y', linestyle='--', alpha=0.7)
```

```
plt.show()
```



What was done:

A bar plot was used to display the average fraud rate for each merchant category, calculated by grouping the data by `category` and taking the mean of `is_fraud`.

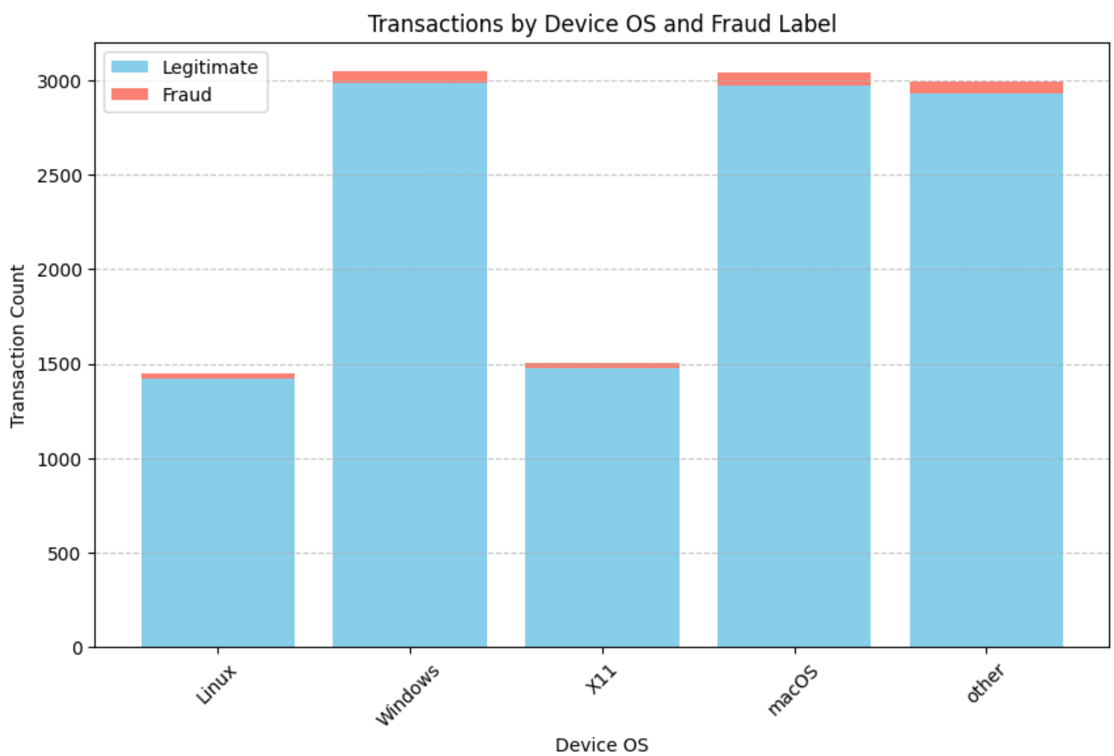
Analysis:

The chart reveals that certain merchant categories, such as "Groceries" and "Entertainment," have slightly higher fraud rates. This insight could be useful for identifying high-risk merchant categories. However, the differences between categories are not dramatic, suggesting that other factors may play a more significant role in fraud.

OS Used in Transactions

[10]:
```
# Count of transactions by device_os and fraud label
device_os_counts = merged_data.groupby(['device_os', 'is_fraud']).size().
    ↪unstack(fill_value=0)

# Stacked bar plot
device_os = device_os_counts.index
legit_counts = device_os_counts[0]
```

```
fraud_counts = device_os_counts[1]

x = range(len(device_os))
plt.figure(figsize=(10, 6))
plt.bar(x, legit_counts, label='Legitimate', color='skyblue')
plt.bar(x, fraud_counts, label='Fraud', bottom=legit_counts, color='salmon')

plt.xticks(x, device_os, rotation=45)
plt.title('Transactions by Device OS and Fraud Label')
plt.xlabel('Device OS')
plt.ylabel('Transaction Count')
plt.legend()
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```



What was done:

A stacked bar plot was created to compare the number of transactions for each device_os, split by fraud label (is_fraud).

Analysis:

The chart shows that Windows and macOS have the highest number of both legitimate and fraudulent transactions, likely reflecting their popularity among users. However, the proportion of fraudulent transactions does not vary significantly across operating systems. This indicates that device

OS may not be a strong standalone feature for fraud detection.

Fraud Rate by City

[11]:
```python
# Calculate fraud rate per city
city_fraud_rate = merged_data.groupby('city')['is_fraud'].mean().
 ↪sort_values(ascending=False)

# Bar plot for fraud rate by city
plt.figure(figsize=(10, 6))
plt.bar(city_fraud_rate.index, city_fraud_rate.values, color='orange')
plt.title('Cities by Fraud Rate')
plt.xlabel('City')
plt.ylabel('Fraud Rate')
plt.xticks(rotation=45)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

What was done:

A bar plot was created to show the cities with the highest fraud rates. The fraud rate for each city was calculated as the mean of `is_fraud`.

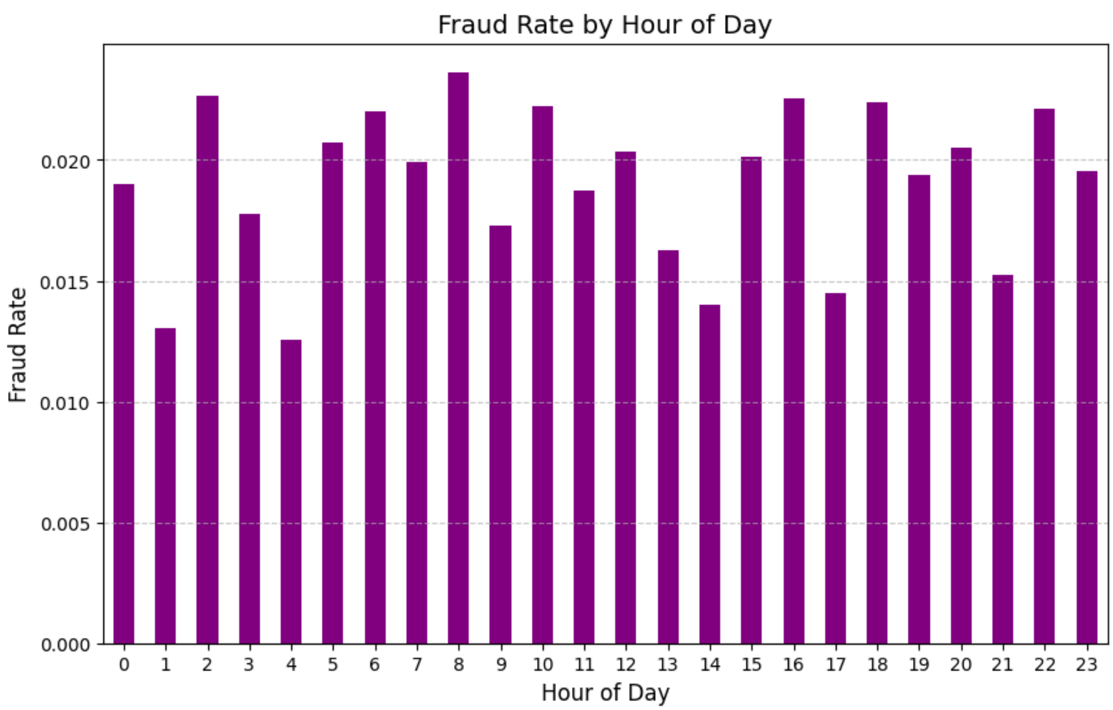
Analysis:

The chart indicates that major metropolitan areas such as Chicago, Phoenix and Houston have higher fraud rates. Upon further inspection, the inclusion of “Test City” appears to be a synthetic or placeholder entry in the dataset rather than a real location. This type of entry is likely used for testing purposes or as a default value and does not represent actual transactional data.

Its presence can distort the analysis by introducing artificial patterns or biasing the interpretation of fraud rates. For this reason, “Test City” should be excluded from the analysis to ensure that insights are based solely on genuine and reliable data. Further exploration of the relationship between fraud rates and factors such as city population, transaction volume, or merchant density can help uncover the underlying dynamics contributing to higher fraud rates in specific cities.

Fraud Rate by Hour of Day

[12]:
```python
# Convert Unix time to datetime and extract the hour
merged_data['hour'] = pd.to_datetime(merged_data['unix_time'], unit='s').dt.hour

# Group by hour and calculate fraud rate
fraud_rate_by_hour = merged_data.groupby('hour')['is_fraud'].mean()

# Bar plot
plt.figure(figsize=(10, 6))
fraud_rate_by_hour.plot(kind='bar', color='purple')
plt.title('Fraud Rate by Hour of Day', fontsize=14)
plt.xlabel('Hour of Day', fontsize=12)
plt.ylabel('Fraud Rate', fontsize=12)
plt.xticks(rotation=0)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```



What was done:

This bar plot visualizes the fraud rate across different hours of the day. The hour was extracted from the unix_time column, and the fraud rate (is_fraud) was calculated as the mean of fraud labels for each hour. This provides insight into the temporal patterns of fraudulent activity.

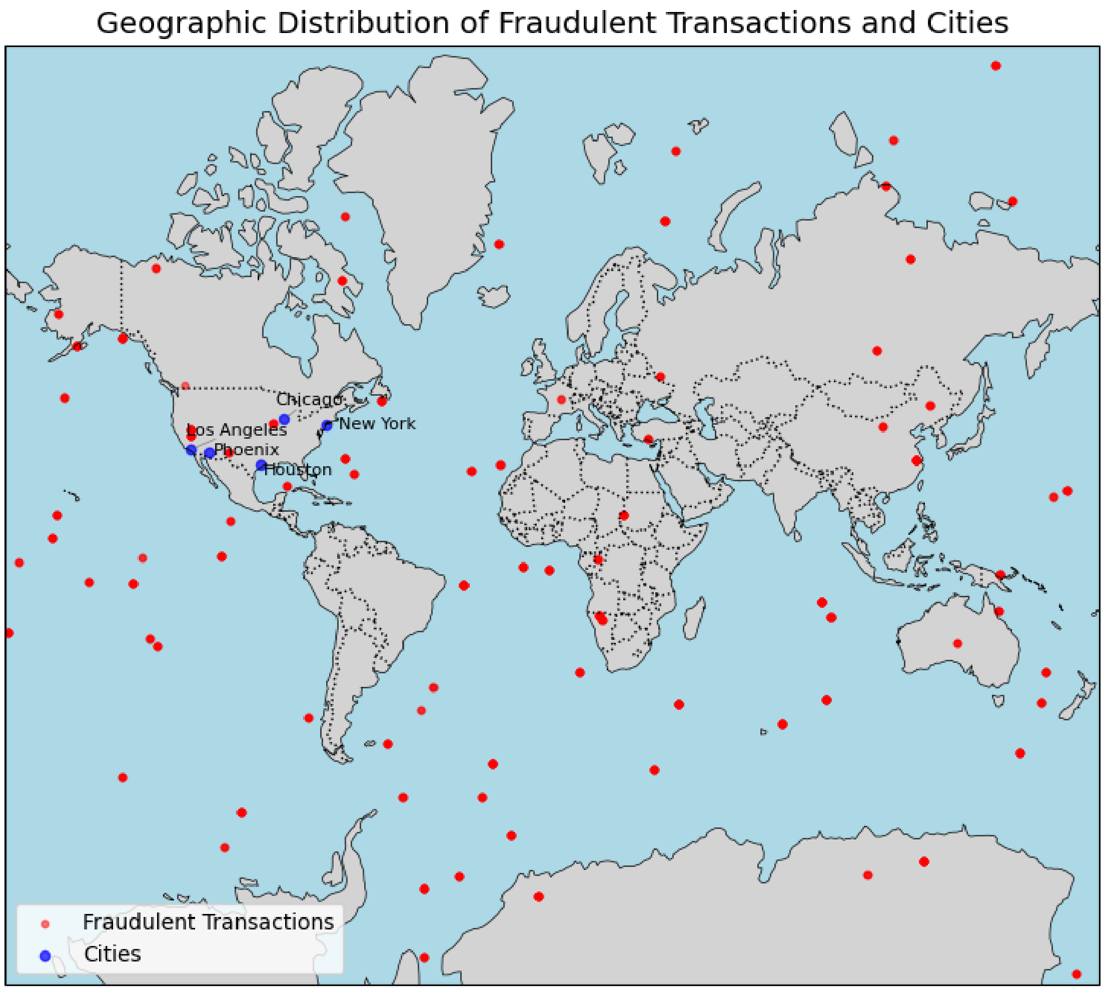
Analysis:

While there are peaks and dips, the fraud rate does not vary drastically across hours, suggesting that fraud occurs throughout the day with certain periods being slightly riskier.

Geographic Distribution of Fraudulent Transactions and Cities

[13]:
```
# Filter fraudulent transactions
fraud_data = merged_data[merged_data['is_fraud'] == 1]

# Filter unique cities with valid coordinates
city_data_clean = merged_data[['city', 'lat', 'long']].drop_duplicates().
 ↪dropna(subset=['lat', 'long'])

# Create a figure and set up a map projection (Mercator)
fig = plt.figure(figsize=(12, 8))
ax = plt.axes(projection=ccrs.Mercator())

# Add map features
ax.add_feature(cfeature.COASTLINE, linewidth=0.5)
ax.add_feature(cfeature.BORDERS, linestyle=':')
ax.add_feature(cfeature.LAND, facecolor='lightgray')
ax.add_feature(cfeature.OCEAN, facecolor='lightblue')

# Plot fraudulent transactions as scatter points
plt.scatter(
    fraud_data['merch_long'], fraud_data['merch_lat'],
    color='red', alpha=0.5, s=10, transform=ccrs.PlateCarree(),
    label='Fraudulent Transactions'
)

# Plot city locations as blue scatter points
plt.scatter(
    city_data_clean['long'], city_data_clean['lat'],
    color='blue', alpha=0.7, s=20, transform=ccrs.PlateCarree(),
    label='Cities'
)

# Add city labels with adjustText
texts = []
for _, row in city_data_clean.iterrows():
```

```
    texts.append(plt.text(
        row['long'], row['lat'], row['city'],
        fontsize=8, transform=ccrs.PlateCarree(), color='black'
    ))
# Adjust text to avoid overlaps
adjust_text(texts, arrowprops=dict(arrowstyle="->", color='gray', lw=0.5))

# Add title and legend
plt.title('Geographic Distribution of Fraudulent Transactions and Cities',␣
 ↪fontsize=14)
plt.legend(loc='lower left', fontsize=10)

# Set extent (map boundaries) based on the data
plt.gca().set_extent([
    min(city_data_clean['long'].min(), fraud_data['merch_long'].min()) - 1,
    max(city_data_clean['long'].max(), fraud_data['merch_long'].max()) + 1,
    min(city_data_clean['lat'].min(), fraud_data['merch_lat'].min()) - 1,
    max(city_data_clean['lat'].max(), fraud_data['merch_lat'].max()) + 1
], crs=ccrs.PlateCarree())

# Show the plot
plt.show()
```



What was done:

The map shows the geographic distribution of fraudulent transactions (red points) overlaid with city locations (blue points). To improve readability, city labels were dynamically adjusted using the adjustText library to avoid overlapping, with arrows indicating their original positions.

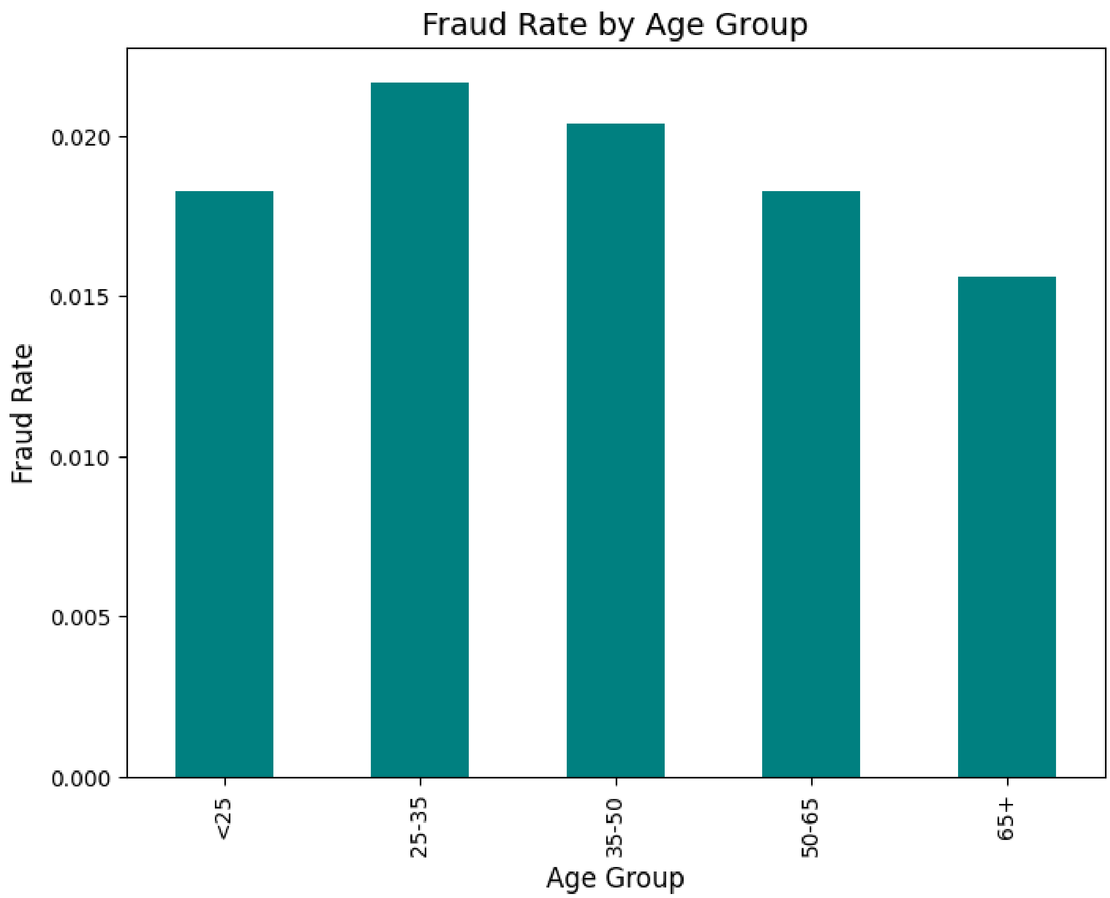
Analysis:

While some fraudulent transactions are near major urban centers like New York, Chicago, and Los Angeles, many points appear as outliers, such as those in the ocean or sparsely populated regions. This suggests that the geographic coordinates may lack strong correlation with city locations, likely due to errors or placeholders in the dataset. However, there are clusters of fraudulent transactions near certain cities that could warrant further analysis to identify potential patterns or high-risk areas.

Fraud Rate by Customer Age Group

[14]:
```python
# Calculate age
merged_data['age'] = pd.to_datetime('2023-01-01') - pd.
    ↪to_datetime(merged_data['dob'])
merged_data['age'] = merged_data['age'].dt.days // 365

# Bin age into groups
bins = [0, 25, 35, 50, 65, 100]
labels = ['<25', '25-35', '35-50', '50-65', '65+']
merged_data['age_group'] = pd.cut(merged_data['age'], bins=bins, labels=labels,␣
    ↪right=False)

# Calculate fraud rate by age group
fraud_rate_by_age = merged_data.groupby('age_group')['is_fraud'].mean()

merged_data.drop('age_group', axis=1, inplace=True)

# Bar plot
plt.figure(figsize=(8, 6))
fraud_rate_by_age.plot(kind='bar', color='teal')
plt.title('Fraud Rate by Age Group', fontsize=14)
plt.xlabel('Age Group', fontsize=12)
plt.ylabel('Fraud Rate', fontsize=12)
plt.show()
```

```
/tmp/ipykernel_41543/2815110579.py:11: FutureWarning: The default of
observed=False is deprecated and will be changed to True in a future version of
pandas. Pass observed=False to retain current behavior or observed=True to adopt
the future default and silence this warning.
  fraud_rate_by_age = merged_data.groupby('age_group')['is_fraud'].mean()
```



What was done:

The bar chart displays the fraud rate across different age groups.

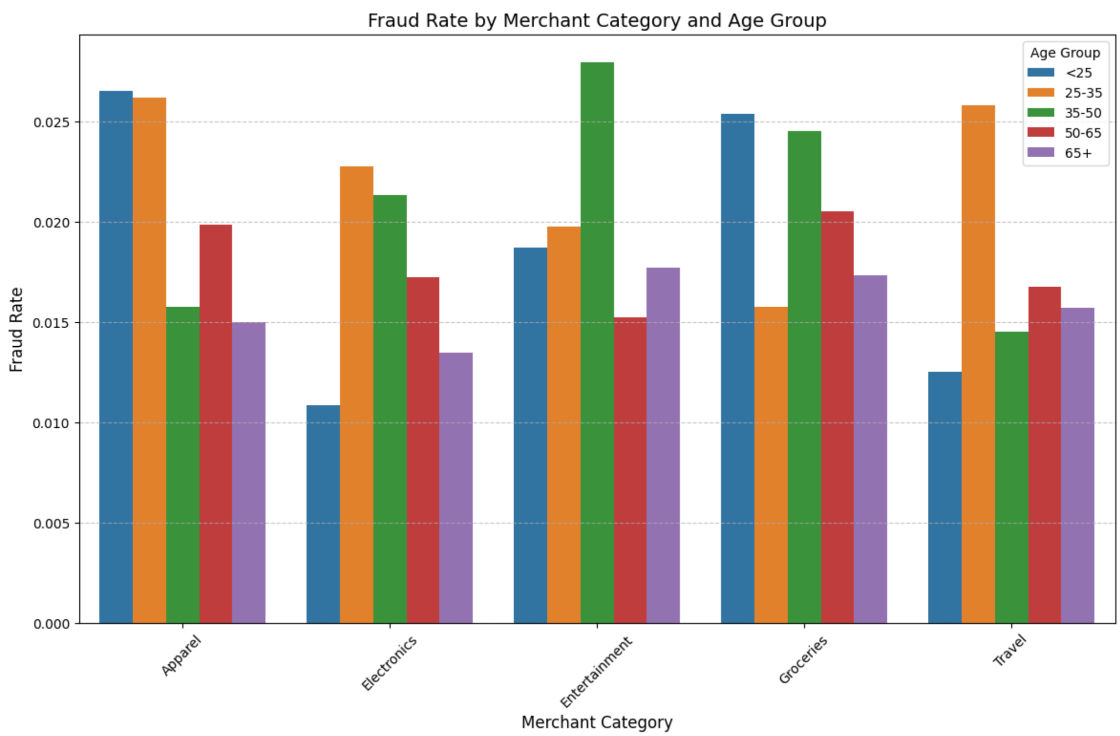
Analysis:

Fraud rates are higher among individuals aged 25-35 and 35-50, suggesting that these groups may be more frequently targeted by fraudsters or engage more in high-risk transaction behaviors. Conversely, the fraud rate is lower for individuals aged 65+, which could be due to lower transaction volumes or more cautious spending habits in this demographic. While the differences between age groups are not drastic, these insights could inform targeted fraud prevention strategies for higher-risk groups.

Fraud Rate by Merchant Category and Age Group

[15]:
```
# Calculate age
merged_data['age'] = pd.to_datetime('2023-01-01') - pd.
  ↪to_datetime(merged_data['dob'])
merged_data['age'] = merged_data['age'].dt.days // 365

# Bin age into groups
```

```python
bins = [0, 25, 35, 50, 65, 100]
labels = ['<25', '25-35', '35-50', '50-65', '65+']
merged_data['age_group'] = pd.cut(merged_data['age'], bins=bins, labels=labels,␣
↪right=False)

# Group data by age group and merchant category, then calculate fraud rate
fraud_rate_by_category_age = merged_data.groupby(['age_group', 'category'],␣
↪observed=True)['is_fraud'].mean().reset_index()

merged_data.drop('age_group', axis=1, inplace=True)

# Plot a grouped bar plot
plt.figure(figsize=(14, 8))
sns.barplot(data=fraud_rate_by_category_age, x='category', y='is_fraud',␣
↪hue='age_group', errorbar=None)
plt.title('Fraud Rate by Merchant Category and Age Group', fontsize=14)
plt.xlabel('Merchant Category', fontsize=12)
plt.ylabel('Fraud Rate', fontsize=12)
plt.xticks(rotation=45)
plt.legend(title='Age Group')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```



What was done:

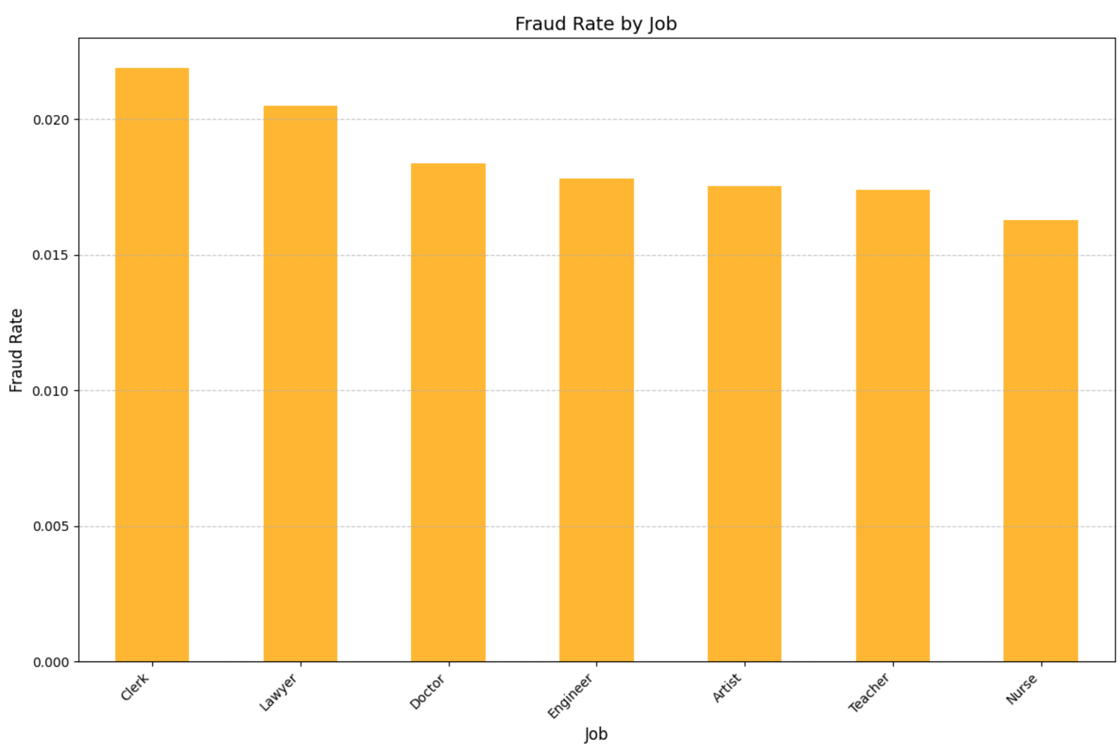
The bar chart shows the fraud rate across merchant categories for different age groups.

Analysis:

Younger groups (<25 and 25-35) have higher fraud rates in categories like Apparel and Travel, while middle-aged groups (35-50) show peaks in Entertainment. Older groups (65+) generally experience lower fraud rates across categories. These patterns suggest that fraudsters may target specific demographics based on category-related behaviors, such as younger individuals in Apparel and Travel or middle-aged individuals in Entertainment.

2.2.5 Fraud Rate by Job

[16]:
```
# Calculate fraud rate for each job
fraud_rate_by_job = merged_data.groupby('job')['is_fraud'].mean().
 ↪sort_values(ascending=False)

# Plot the fraud rate by job
plt.figure(figsize=(12, 8))
fraud_rate_by_job.plot(kind='bar', color='orange', alpha=0.8)
plt.title('Fraud Rate by Job', fontsize=14)
plt.xlabel('Job', fontsize=12)
plt.ylabel('Fraud Rate', fontsize=12)
plt.xticks(rotation=45, ha='right')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```



What was done:

A bar chart was created to analyze the relationship between customers' jobs and the fraud rate.

Analysis:

The chart shows that certain professions, such as Clerks and Lawyers, have slightly higher fraud rates compared to other professions like Teachers and Nurses. These differences might reflect behavioral patterns, spending habits, or exposure to fraud based on the nature of the profession. The fraud rates across jobs are relatively close, indicating that job type alone may not be a strong predictor of fraud but could be considered alongside other features

Conclusion on Data Visualization The analysis revealed several key insights about fraudulent transactions. Fraud is relatively rare in the dataset, accounting for only 1.9% of all transactions, highlighting the challenge of identifying such rare events. Geographic patterns showed clusters of fraudulent transactions near major urban centers, though significant outliers and inconsistent coordinates suggest that location data may not be highly reliable. Fraud rates varied by age group, with younger and middle-aged individuals (25-35 and 35-50) being more frequently targeted, particularly in categories like Apparel, Travel, and Entertainment. Older age groups (65+) generally experienced lower fraud rates. Certain categories, such as Travel and Entertainment, showed higher fraud activity, suggesting specific areas where fraudsters exploit vulnerabilities.

2.3 1.2- Data Preparation

Data preparation is a critical step in the machine learning pipeline, ensuring that the dataset is clean, consistent, and structured for effective modeling. This process involves handling missing values, encoding categorical variables, creating meaningful features, and addressing potential issues like class imbalance. Proper data preparation enhances the quality of the input data, reduces noise, and helps models better capture underlying patterns. In this project, the data preparation phase focuses on transforming the provided transaction data into a format suitable for building a predictive model to detect fraudulent transactions. This includes cleaning the dataset, engineering new features, scaling numerical variables, and addressing the imbalanced nature of the target variable. These steps aim to improve the accuracy and reliability of the predictive models in identifying fraud.

2.3.1 Split Data into Train and Test Sets

[17]:
```
X = merged_data.drop('is_fraud', axis=1)
y = merged_data['is_fraud']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42, stratify=y)
```

2.3.2 Handle Missing Values

[18]:
```
# Check for missing values
print("\nMissing Values:")
print(X_train.isnull().sum())
```

```
Missing Values:
index                       0
trans_date_trans_time      79
cc_num                      0
device_os               14349
merchant                    0
amt                        79
trans_num                   0
unix_time                   0
first                       9
last                        9
gender                      9
street                      9
city                        9
zip                       171
job                       171
dob                         9
category                  464
merch_lat                 464
merch_long                  9
merchant_id                 9
lat                     15920
long                    15920
city_pop                15920
state                   15920
hour                        0
age                         9
dtype: int64
```

[19]:
```
X_train['amt'] = X_train['amt'].fillna(X_train['amt'].mean())
X_test['amt'] = X_train['amt'].fillna(X_test['amt'].mean())
```

[20]:
```
#X_train['category'] = X_train['category'].fillna('Unknown')
```

[21]:
```
#X_train['job'] = X_train['job'].fillna('Unknown')
```

[22]:
```
X_train['device_os'] = X_train['device_os'].fillna('Unknown')
X_test['device_os'] = X_test['device_os'].fillna('Unknown')
```

[23]:
```
#merged_data.dropna(subset=['lat', 'long', 'merch_lat'], inplace=True)
```

[24]:
```
# Check for missing values
print("\nMissing Values:")
print(X_train.isnull().sum())
```

```
Missing Values:
index                        0
trans_date_trans_time       79
cc_num                       0
device_os                    0
merchant                     0
amt                          0
trans_num                    0
unix_time                    0
first                        9
last                         9
gender                       9
street                       9
city                         9
zip                        171
job                        171
dob                          9
category                   464
merch_lat                  464
merch_long                   9
merchant_id                  9
lat                      15920
long                     15920
city_pop                 15920
state                    15920
hour                         0
age                          9
dtype: int64
```

2.3.3 Handle Duplicate Values

[25]:
```
# Check for duplicate transactions
print("\nDuplicate Transactions:")
print(X_train.duplicated(subset='trans_num').sum())
```

```
Duplicate Transactions:
337
```

[26]:
```
# Delete duplicate transactions
# Mudar isto para procurar as linahs duplicadas com menos nas e remover as que
 ↪tem mais nas (fazer funçao para isso)
#
```

```
#kaggle_data = kaggle_data.drop_duplicates(subset='trans_num', keep='first')
```

2.3.4 Feature Engineering

[27]:
```
# Convert unix_time to datetime
X_train['datetime'] = pd.to_datetime(X_train['unix_time'], unit='s')

# Extract hour, day of the week, and month
X_train['hour'] = X_train['datetime'].dt.hour
X_train['day_of_week'] = X_train['datetime'].dt.dayofweek  # 0=Monday, 6=Sunday
X_train['month'] = X_train['datetime'].dt.month

# Drop the intermediate 'datetime' column if not needed
X_train.drop('datetime', axis=1, inplace=True)



# Convert unix_time to datetime
X_test['datetime'] = pd.to_datetime(X_test['unix_time'], unit='s')

# Extract hour, day of the week, and month
X_test['hour'] = X_test['datetime'].dt.hour
X_test['day_of_week'] = X_test['datetime'].dt.dayofweek  # 0=Monday, 6=Sunday
X_test['month'] = X_test['datetime'].dt.month

# Drop the intermediate 'datetime' column if not needed
X_test.drop('datetime', axis=1, inplace=True)
```

[28]:
```
#merged_data['age'] = 2023 - pd.to_datetime(merged_data['dob']).dt.year
```

[29]:
```
#merged_data['distance'] = merged_data.apply(lambda row: geodesic((row['lat'],␣
↪row['long']), (row['merch_lat'], row['merch_long'])).km, axis=1)
```

[30]:
```
# Calculate age
#merged_data['age'] = pd.to_datetime('2023-01-01') - pd.
↪to_datetime(merged_data['dob'])
#merged_data['age'] = merged_data['age'].dt.days // 365

# Bin age into groups
#bins = [0, 25, 35, 50, 65, 100]
#labels = ['<25', '25-35', '35-50', '50-65', '65+']
#merged_data['age_group'] = pd.cut(merged_data['age'], bins=bins,␣
↪labels=labels, right=False)
```

2.3.5 Encode Categorical Variables

[31]:
```
# One-hot encoding example
X_train = pd.get_dummies(X_train, columns=['device_os','merchant'],␣
 ↪drop_first=False)
X_test = pd.get_dummies(X_test, columns=['device_os','merchant'],␣
 ↪drop_first=False)
```

2.3.6 Drop Redundant or Unnecessary Columns

[32]:
```
X_train.drop(['trans_num', 'unix_time',␣
 ↪'trans_date_trans_time','zip','first','last','gender','street','dob','state'
                ␣
 ↪,'lat','long','merch_lat','merch_long','job','city','age','category','merchant_id'
                 ,'city_pop','merchant_Unknown_Merchant'], axis=1,␣
 ↪inplace=True)

X_test.drop(['trans_num', 'unix_time',␣
 ↪'trans_date_trans_time','zip','first','last','gender','street','dob','state'
                ␣
 ↪,'lat','long','merch_lat','merch_long','job','city','age','category','merchant_id'
                 ,'city_pop','merchant_Unknown_Merchant'], axis=1,␣
 ↪inplace=True)
```

[33]:
```
X_train.head()
```

[33]:
```
       index            cc_num     amt  hour  day_of_week  month  \
13457  13457  1752467965316559  328.06     8            5     12
25315  25315  2762537615033297  313.53     1            3     10
29572  29572  7841160067409545  255.81     1            5      1
9273    3160  7466488688331597  222.52    15            3     12
25806  25806  9812171407923350  117.32     1            3      5

       device_os_Linux  device_os_Unknown  device_os_Windows  device_os_X11  \
13457            False               True              False          False
25315            False              False              False          False
29572            False               True              False          False
9273              True              False              False          False
25806            False               True              False          False

       …  merchant_Merchant_90  merchant_Merchant_91  merchant_Merchant_92  \
13457  …                 False                 False                 False
25315  …                 False                 False                 False
29572  …                 False                 False                 False
9273   …                 False                 False                 False
25806  …                 False                 False                 False
```

```
       merchant_Merchant_93  merchant_Merchant_94  merchant_Merchant_95  \
13457                 False                 False                 False
25315                 False                 False                 False
29572                 False                 False                 False
9273                  False                 False                 False
25806                 False                 False                 False

       merchant_Merchant_96  merchant_Merchant_97  merchant_Merchant_98  \
13457                 False                 False                 False
25315                 False                 False                 False
29572                 False                 False                 False
9273                  False                 False                 False
25806                 False                 False                 False

       merchant_Merchant_99
13457                 False
25315                 False
29572                 False
9273                  False
25806                 False

[5 rows x 112 columns]
```

2.3.7 Normalize/Scale Numerical Features

[34]:
```
numerical_columns = ['amt'] # Maybe add more numeric collums
scaler = StandardScaler()
X_train[numerical_columns] = scaler.fit_transform(X_train[numerical_columns])
```

2.3.8 Class Imbalance

[35]:
```
#X = merged_data.drop('is_fraud', axis=1)
#y = merged_data['is_fraud']


smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
```

2.4 1.3- Clustering

2.4.1 DBSCAN

[36]:
```
"""
# Select numerical features for clustering
features = ['amt', 'hour']  # Replace with features relevant to your data
data_subset = merged_data[features].dropna()

# Standardize the features
```

```
scaler = StandardScaler()
scaled_features = scaler.fit_transform(data_subset)

# Apply DBSCAN
dbscan = DBSCAN(eps=1.5, min_samples=10)  # Adjust `eps` and `min_samples` as␣
↪needed
clusters = dbscan.fit_predict(scaled_features)

# Add cluster labels to the dataset
merged_data['cluster'] = clusters

# Visualize the clusters
sns.scatterplot(data=merged_data, x='amt', y='hour', hue='cluster',␣
↪palette='tab10')
plt.title('DBSCAN Clustering of Transactions')
plt.show()
"""
```

[36]:

```
"\n# Select numerical features for clustering\nfeatures = ['amt', 'hour']  #
Replace with features relevant to your data\ndata_subset =
merged_data[features].dropna()\n\n# Standardize the features\nscaler =
StandardScaler()\nscaled_features = scaler.fit_transform(data_subset)\n\n# Apply
DBSCAN\ndbscan = DBSCAN(eps=1.5, min_samples=10)  # Adjust `eps` and
`min_samples` as needed\nclusters = dbscan.fit_predict(scaled_features)\n\n# Add
cluster labels to the dataset\nmerged_data['cluster'] = clusters\n\n# Visualize
the clusters \nsns.scatterplot(data=merged_data, x='amt', y='hour',
hue='cluster', palette='tab10')\nplt.title('DBSCAN Clustering of
Transactions')\nplt.show()\n"
```

2.4.2 K-Means

[37]:

```
"""


# Aggregate data by customer
customer_data = merged_data.groupby('cc_num').agg({
    'amt': 'mean',  # Average transaction amount
    'distance': 'mean',  # Average distance
    'is_fraud': 'mean',  # Fraud rate per customer
    'trans_num': 'count'  # Number of transactions
}).reset_index()

# Select features for clustering
features = ['amt', 'distance', 'is_fraud', 'trans_num']
customer_features = customer_data[features]

# Standardize the data
scaler = StandardScaler()
```

```
scaled_features = scaler.fit_transform(customer_features)

# Apply K-Means
kmeans = KMeans(n_clusters=4, random_state=42)  # Adjust `n_clusters` as needed
customer_data['cluster'] = kmeans.fit_predict(scaled_features)

# Visualize clusters (e.g., fraud rate vs transaction amount)
sns.scatterplot(data=customer_data, x='amt', y='is_fraud', hue='cluster',␣
↪palette='tab10')
plt.title('Customer Clustering Based on Behavior')
plt.show()

"""
```

[37]:
```
"\n\n# Aggregate data by customer\ncustomer_data =
merged_data.groupby('cc_num').agg({\n    'amt': 'mean',  # Average transaction
amount\n    'distance': 'mean',  # Average distance\n    'is_fraud': 'mean',  #
Fraud rate per customer\n    'trans_num': 'count'  # Number of
transactions\n}).reset_index()\n\n# Select features for clustering\nfeatures =
['amt', 'distance', 'is_fraud', 'trans_num']\ncustomer_features =
customer_data[features]\n\n# Standardize the data\nscaler =
StandardScaler()\nscaled_features = scaler.fit_transform(customer_features)\n\n#
Apply K-Means\nkmeans = KMeans(n_clusters=4, random_state=42)  # Adjust
`n_clusters` as needed\ncustomer_data['cluster'] =
kmeans.fit_predict(scaled_features)\n\n# Visualize clusters (e.g., fraud rate vs
transaction amount)\nsns.scatterplot(data=customer_data, x='amt', y='is_fraud',
hue='cluster', palette='tab10')\nplt.title('Customer Clustering Based on
Behavior')\nplt.show()\n\n"
```

3 Task 2: Predictive Modelling

[38]:
```
kaggle_data = pd.read_csv('kaggle-data/test_transactions.csv')

kaggle_data['amt'] = kaggle_data['amt'].fillna(kaggle_data['amt'].mean())
kaggle_data['device_os'] = kaggle_data['device_os'].fillna('Unknown')

# Convert unix_time to datetime
kaggle_data['datetime'] = pd.to_datetime(kaggle_data['unix_time'], unit='s')

# Extract hour, day of the week, and month
kaggle_data['hour'] = kaggle_data['datetime'].dt.hour
kaggle_data['day_of_week'] = kaggle_data['datetime'].dt.dayofweek  # 0=Monday,␣
↪6=Sunday
kaggle_data['month'] = kaggle_data['datetime'].dt.month

# Drop the intermediate 'datetime' column if not needed
```

```
kaggle_data.drop('datetime', axis=1, inplace=True)


#kaggle_data['age'] = 2023 - pd.to_datetime(kaggle_data['dob']).dt.year

# Calculate age
#kaggle_data['age'] = pd.to_datetime('2023-01-01') - pd.
 ↪to_datetime(kaggle_data['dob'])
#kaggle_data['age'] = kaggle_data['age'].dt.days // 365

# Bin age into groups
#bins = [0, 25, 35, 50, 65, 100]
#labels = ['<25', '25-35', '35-50', '50-65', '65+']
#kaggle_data['age_group'] = pd.cut(kaggle_data['age'], bins=bins,␣
 ↪labels=labels, right=False)

# One-hot encoding example
kaggle_data = pd.get_dummies(kaggle_data, columns=['device_os','merchant'],␣
 ↪drop_first=False)

numerical_columns = ['amt'] # Maybe add more numeric collums
scaler = StandardScaler()
kaggle_data[numerical_columns] = scaler.
 ↪fit_transform(kaggle_data[numerical_columns])

kaggle_data.drop(['trans_num', 'unix_time', 'trans_date_trans_time'], axis=1,␣
 ↪inplace=True)

kaggle_data.insert(6,'device_os_Linux',False)
```

[39]:
```
kaggle_data.head()
```

[39]:
```
   index            cc_num       amt  hour  day_of_week  month  \
0  30000  7554841364236395 -0.115599     0            4     10
1  30001  8299329211991767 -1.005189     0            0      7
2  30002  1231978459576854  0.069148     0            2     12
3  30003  2342135124331538  0.723117     0            6      8
4  30004  3265698529432098 -0.007218     0            1      5

   device_os_Linux  device_os_Unknown  device_os_Windows  device_os_X11  …  \
0            False              False               True          False  …
1            False               True              False          False  …
2            False              False              False           True  …
3            False              False              False          False  …
4            False              False              False          False  …

   merchant_Merchant_90  merchant_Merchant_91  merchant_Merchant_92  \
```

```
0                 False                 False                 False
1                 False                 False                 False
2                 False                 False                 False
3                 False                 False                 False
4                 False                 False                 False

   merchant_Merchant_93  merchant_Merchant_94  merchant_Merchant_95  \
0                 False                 False                 False
1                 False                 False                 False
2                 False                 False                 False
3                 False                 False                 False
4                 False                 False                 False

   merchant_Merchant_96  merchant_Merchant_97  merchant_Merchant_98  \
0                 False                 False                 False
1                 False                 False                 False
2                 False                 False                 False
3                 False                 False                 False
4                 False                 False                 False

   merchant_Merchant_99
0                 False
1                 False
2                 False
3                 False
4                  True

[5 rows x 112 columns]
```

3.1 Random Forest Classifier

[40]:
```python
# Train a Random Forest Classifier
clf = RandomForestClassifier(random_state=42)
clf.fit(X_train, y_train)
```

[40]:
```
RandomForestClassifier(random_state=42)
```

[41]:
```python
# Predict fraud (binary labels)
y_pred = clf.predict(X_test)

f1 = f1_score(y_test, y_pred)
print(f"F1-Score: {f1}")
```

```
F1-Score: 0.7472527472527473
```

[42]:
```python
# Predict probabilities for the positive class
y_probs = clf.predict_proba(X_test)[:, 1]  # Get probabilities for class 1
```

```python
# Calculate the AUC-ROC score
auc_score = roc_auc_score(y_test, y_probs)

print(f"AUC-ROC Score: {auc_score:.2f}")
```

```
AUC-ROC Score: 0.88
```

[43]:
```python
# Predict probabilities for the positive class (fraud)
test_probs = clf.predict_proba(kaggle_data)[:, 1]  # Probabilities for class 1␣
↪(fraud)

submission = pd.DataFrame({
    'index': kaggle_data['index'],
    'is_fraud': test_probs          # Predicted probabilities
})

# Save to CSV
submission.to_csv('submission/submission_random_forest.csv', index=False)

print("Submission file created: 'submission_random_forest.csv'")
```

```
Submission file created: 'submission_random_forest.csv'
```

Score on Kaggle: 0.45396

3.2 Grid Search - Random Forest Classifier

[44]:
```python
best_model_path = "models/grid_search_random_forest.pkl"

if os.path.exists(best_model_path):
    # Carregar o modelo salvo
    with open(best_model_path, 'rb') as file:
        best_rf = pickle.load(file)
    print("Output Best params: {'max_depth': 10, 'min_samples_leaf': 1,␣
↪'min_samples_split': 2, 'n_estimators': 100}")
else:
    param_grid = {
        'n_estimators': [100, 200, 500],
        'max_depth': [None, 10, 20],
        'min_samples_split': [2, 5],
        'min_samples_leaf': [1, 2, 4]
    }

    grid_search = GridSearchCV(estimator=clf, param_grid=param_grid, cv=5,␣
↪scoring='roc_auc', n_jobs=-1,verbose=2)
    grid_search.fit(X_train, y_train)

    best_rf = grid_search.best_estimator_
```

```
    print("Best params:", grid_search.best_params_)

    with open(best_model_path, 'wb') as file:
        pickle.dump(best_rf, file)
    print("Best model saved in:", best_model_path)
```

Output Best params: {'max_depth': 10, 'min_samples_leaf': 1,
'min_samples_split': 2, 'n_estimators': 100}

[45]:
```
# Predict fraud (binary labels)
y_pred = best_rf.predict(X_test)

f1 = f1_score(y_test, y_pred)
print(f"F1-Score: {f1}")
```

F1-Score: 0.0

[46]:
```
y_probs = best_rf.predict_proba(X_test)[:, 1]

auc_score = roc_auc_score(y_test, y_probs)

print(f"AUC-ROC Score: {auc_score:.2f}")
```

AUC-ROC Score: 0.94

[47]:
```
# Predict probabilities for the positive class (fraud)
test_probs = best_rf.predict_proba(kaggle_data)[:, 1]  # Probabilities for␣
 ↪class 1 (fraud)

submission = pd.DataFrame({
    'index': kaggle_data['index'],
    'is_fraud': test_probs          # Predicted probabilities
})

# Save to CSV
submission.to_csv('submission/submission_grid_search_random_forest.csv',␣
 ↪index=False)

print("Submission file created: 'submission_grid_search_random_forest.csv'")
```

Submission file created: 'submission_grid_search_random_forest.csv'

Score on Kaggle: 0.39635

3.3 Random Search - Random Forest Classifier

[48]:
```
best_model_path = "models/random_search_random_forest.pkl"

if os.path.exists(best_model_path):
    # Carregar o modelo salvo
    with open(best_model_path, 'rb') as file:
        best_rf = pickle.load(file)
    print("Output Best params: {'n_estimators': 100, 'min_samples_split': 10,␣
 ↪'min_samples_leaf': 2, 'max_depth': 10}")
else:
    param_distributions = {
        'n_estimators': [100, 200, 500,1000],
        'max_depth': [None, 10, 20, 30],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4, 5],
    }

    random_search = RandomizedSearchCV(estimator=clf,␣
 ↪param_distributions=param_distributions, n_iter=20, cv=5, scoring='roc_auc',␣
 ↪random_state=42, n_jobs=-1,verbose=2)
    random_search.fit(X_train, y_train)

    best_rf = random_search.best_estimator_

    print("Best params:", random_search.best_params_)


    with open(best_model_path, 'wb') as file:
        pickle.dump(best_rf, file)
    print("Best model saved in:", best_model_path)
```

```
Output Best params: {'n_estimators': 100, 'min_samples_split': 10,
'min_samples_leaf': 2, 'max_depth': 10}
```

[49]:
```
# Predict fraud (binary labels)
y_pred = best_rf.predict(X_test)

f1 = f1_score(y_test, y_pred)
print(f"F1-Score: {f1}")
```

```
F1-Score: 0.0
```

[50]:
```
y_probs = best_rf.predict_proba(X_test)[:, 1]

auc_score = roc_auc_score(y_test, y_probs)

print(f"AUC-ROC Score: {auc_score:.2f}")
```

```
AUC-ROC Score: 0.94
```

[51]:
```python
# Predict probabilities for the positive class (fraud)
test_probs = best_rf.predict_proba(kaggle_data)[:, 1]  # Probabilities for␣
↪class 1 (fraud)

submission = pd.DataFrame({
    'index': kaggle_data['index'],
    'is_fraud': test_probs          # Predicted probabilities
})

# Save to CSV
submission.to_csv('submission/submission_random_search_random_forest.csv',␣
↪index=False)

print("Submission file created: 'submission_random_search_random_forest.csv'")
```

```
Submission file created: 'submission_random_search_random_forest.csv'
```

Score on Kaggle: 0.45330

3.4 XGBOOST

[52]:
```python
xgb = XGBClassifier(n_estimators=500, max_depth=5, learning_rate=0.1,␣
↪random_state=42)
xgb.fit(X_train, y_train)
```

[52]:
```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.1, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=5, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=500, n_jobs=None,
              num_parallel_tree=None, random_state=42, …)
```

[53]:
```python
# Predict fraud (binary labels)
y_pred = xgb.predict(X_test)

f1 = f1_score(y_test, y_pred)
print(f"F1-Score: {f1}")
```

```
F1-Score: 0.676923076923077
```

[54]:
```python
# Predict probabilities for the positive class
y_probs = xgb.predict_proba(X_test)[:, 1]  # Get probabilities for class 1
```

```
# Calculate the AUC-ROC score
auc_score = roc_auc_score(y_test, y_probs)

print(f"AUC-ROC Score: {auc_score:.2f}")
```

```
AUC-ROC Score: 0.93
```

[55]:
```
# Predict probabilities for the positive class (fraud)
test_probs = xgb.predict_proba(kaggle_data)[:, 1]  # Probabilities for class 1␣
↪(fraud)

submission = pd.DataFrame({
    'index': kaggle_data['index'],
    'is_fraud': test_probs          # Predicted probabilities
})

# Save to CSV
submission.to_csv('submission/submission_xgboost.csv', index=False)

print("Submission file created: 'submission_xgboost.csv'")
```

```
Submission file created: 'submission_xgboost.csv'
```

Score on Kaggle: 0.55979 (Before changing the data preparation)

Score on Kaggle: 0.56503 (After changing the data preparation)

3.5 Random Search - XGBOOST

[56]:
```
best_model_path = "models/random_search_xgboost.pkl"

if os.path.exists(best_model_path):
    # Carregar o modelo salvo
    with open(best_model_path, 'rb') as file:
        best_rf = pickle.load(file)
    print("Output Best params: {'subsample': 1.0, 'reg_lambda': 50, 'reg_alpha':
↪ 1, 'n_estimators': 100, 'min_child_weight': 7, 'max_depth': 5,␣
↪'learning_rate': 0.3, 'colsample_bytree': 1.0}")
else:
    param_distributions = {
        'n_estimators': [100, 200, 300, 500, 1000],
        'learning_rate': [0.01, 0.05, 0.1, 0.2, 0.3],
        'max_depth': [3, 5, 7, 10],
        'subsample': [0.6, 0.8, 1.0],
        'colsample_bytree': [0.6, 0.8, 1.0],
        'reg_alpha': [0, 0.1, 1, 10],
        'reg_lambda': [1, 10, 50],
        'min_child_weight': [1, 3, 5, 7]
```

```
    }

    random_search = RandomizedSearchCV(estimator=xgb,␣
↪param_distributions=param_distributions, n_iter=50, cv=5, scoring='roc_auc',␣
↪random_state=42, n_jobs=-1,verbose=2)
    random_search.fit(X_train, y_train)

    best_rf = random_search.best_estimator_

    print("Best params:", random_search.best_params_)


    with open(best_model_path, 'wb') as file:
        pickle.dump(best_rf, file)
    print("Best model saved in:", best_model_path)
```

Output Best params: {'subsample': 1.0, 'reg_lambda': 50, 'reg_alpha': 1, 'n_estimators': 100, 'min_child_weight': 7, 'max_depth': 5, 'learning_rate': 0.3, 'colsample_bytree': 1.0}

[57]:
```
# Predict fraud (binary labels)
y_pred = best_rf.predict(X_test)

f1 = f1_score(y_test, y_pred)
print(f"F1-Score: {f1}")
```

```
F1-Score: 0.696969696969697
```

[58]:
```
# Predict probabilities for the positive class
y_probs = best_rf.predict_proba(X_test)[:, 1]  # Get probabilities for class 1

# Calculate the AUC-ROC score
auc_score = roc_auc_score(y_test, y_probs)

print(f"AUC-ROC Score: {auc_score:.2f}")
```

```
AUC-ROC Score: 0.94
```

[59]:
```
# Predict probabilities for the positive class (fraud)
test_probs = best_rf.predict_proba(kaggle_data)[:, 1]  # Probabilities for␣
↪class 1 (fraud)

submission = pd.DataFrame({
    'index': kaggle_data['index'],
    'is_fraud': test_probs          # Predicted probabilities
})

# Save to CSV
```

```
submission.to_csv('submission/submission_random_search_xgboost.csv',␣
↪index=False)

print("Submission file created: 'submission_random_search_xgboost.csv'")
```

```
Submission file created: 'submission_random_search_xgboost.csv'
```

Score on Kaggle: 0.42539