report

November 27, 2024

1 Credit Card Fraud Detection

1.1 Required libraries

[171]:
```
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
```

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

[172]:
```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	Numerical (Ratio)	Index of the transaction record.
trans_date_trans_time	Temporal	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.
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	Continuous Numerical	Latitude of the merchant's location.
merch_long	Continuous Numerical	Longitude of the merchant's location.
merchant_id	Categorical	Unique identifier for the merchant.

Attribute	Data Type	Description
first	Categorical	Customer's first name.
last	Categorical	Customer's last name.
gender	Categorical	Customer's gender.
street	Categorical	Customer's street address.
city	Categorical	City where the customer resides.
zip	Numerical	Zip code of the customer's address.
job	Categorical	Customer's job/profession.
dob	Temporal	Customer's date of birth.
name	Categorical	Name of the city.
lat	Continuous Numerical	Latitude of the city.
long	Continuous Numerical	Longitude of the city.
city_pop	Numerical	Population of the city.
state	Categorical	State where the city is located.

2.2.3 Data Summarization

[173]:
```
# 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
```

[174]:
```python
# Display summary statistics
print("\nSummary Statistics:")
print(merged_data.describe()) ## NOTEEE:: Por so as variaveis que importam (E.g␣
↪tirar o index....)
merged_data.describe(include=["O"]) # variaveis categoricas
```

```
Summary Statistics:
             index        cc_num           amt     unix_time      is_fraud  \
count  30000.00000  3.000000e+04  29900.000000  3.000000e+04  30000.000000
mean   14994.93820  5.638691e+15    250.063287  1.705650e+09      0.019033
std     8664.71394  2.743709e+15    144.106058  1.530499e+07      0.136644
min        0.00000  1.001432e+15      1.010000  1.672534e+09      0.000000
25%     7478.75000  3.256119e+15    125.235000  1.696269e+09      0.000000
50%    14999.50000  5.491563e+15    249.625000  1.706376e+09      0.000000
75%    22499.25000  8.149117e+15    375.242500  1.718328e+09      0.000000
max    29999.00000  1.000000e+16    499.970000  1.730124e+09      1.000000

                zip     merch_lat    merch_long   merchant_id           lat  \
count  29784.000000  29401.000000  29990.000000  29990.000000  10020.000000
mean   58070.908944      2.990787     -7.727705     50.446215     35.726876
std    24749.348964     55.651821    103.254575     28.939210      4.531306
min    10008.000000    -88.616543   -178.256215      1.000000     29.760400
25%    39192.000000    -46.105529   -101.993026     25.000000     33.448400
50%    58583.000000      0.067189    -16.648430     50.000000     34.052200
75%    78251.000000     49.823343     90.051574     76.000000     40.712800
max    99994.000000     89.069132    178.663853    100.000000     41.878100

               long      city_pop
count  10020.000000  1.002000e+04
mean     -98.630250  3.704410e+06
std       15.963517  2.323382e+06
min     -118.243700  1.680992e+06
25%     -112.074000  2.328000e+06
50%      -95.369800  2.716000e+06
```

```
75%      -87.629800  3.979576e+06
max      -74.006000  8.419600e+06
```

[174]:

```
       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
```

Correlation Matrix 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.

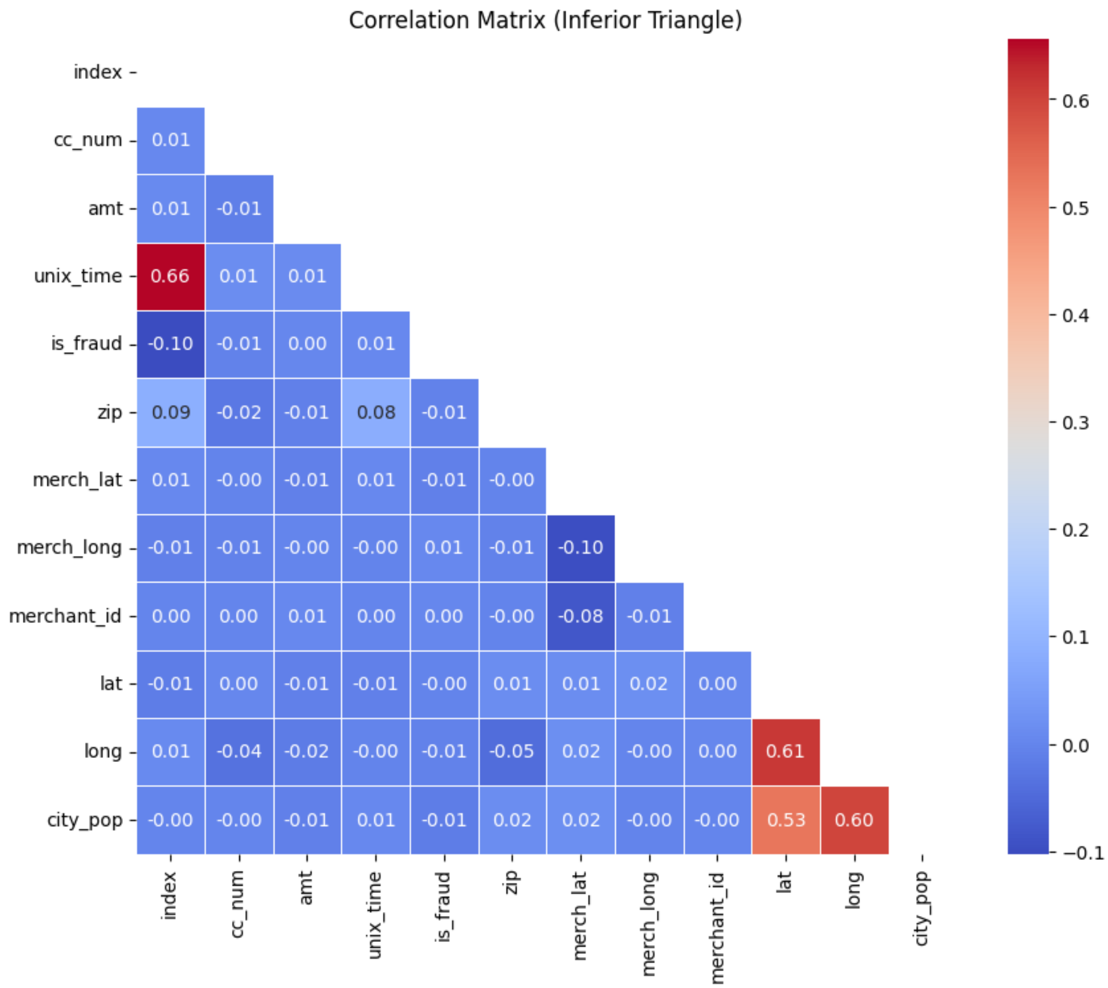
Variables related to city-level information, such as `city_pop`, `lat`, and `long`, exhibit moderate correlations. Specifically, `city_pop` has a positive correlation with both `lat` (0.53) and `long` (0.60), suggesting that high-population cities tend to cluster in specific geographic regions. This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

[175]:

```
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)
```

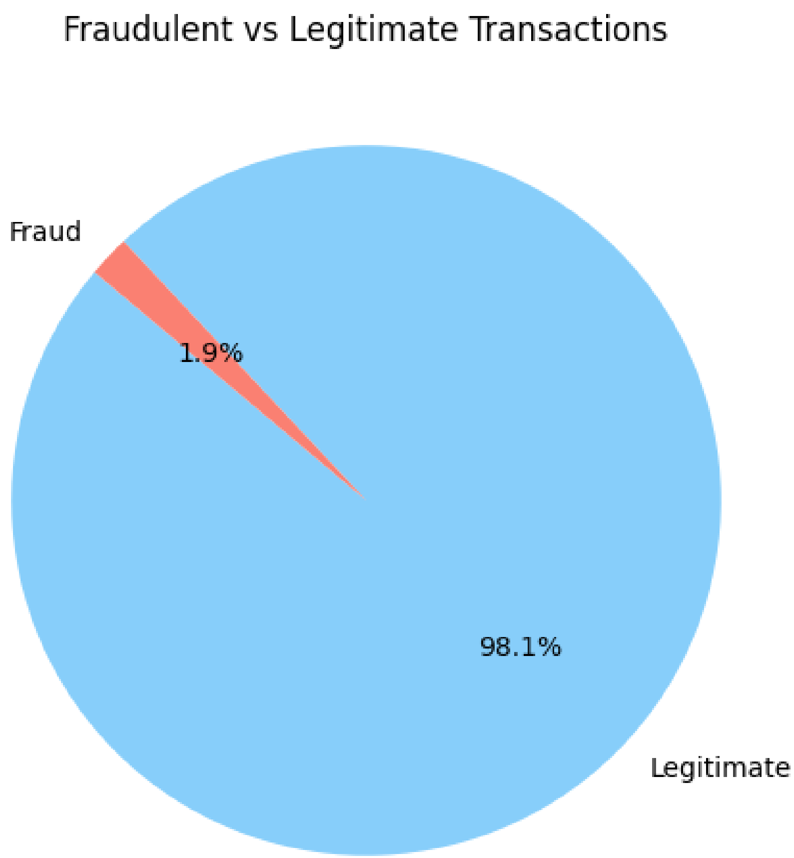
```
plt.title('Correlation Matrix (Inferior Triangle)')
plt.show()
```



2.2.4 Data Visualization

Fraud distribution

[176]:
```
# 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.

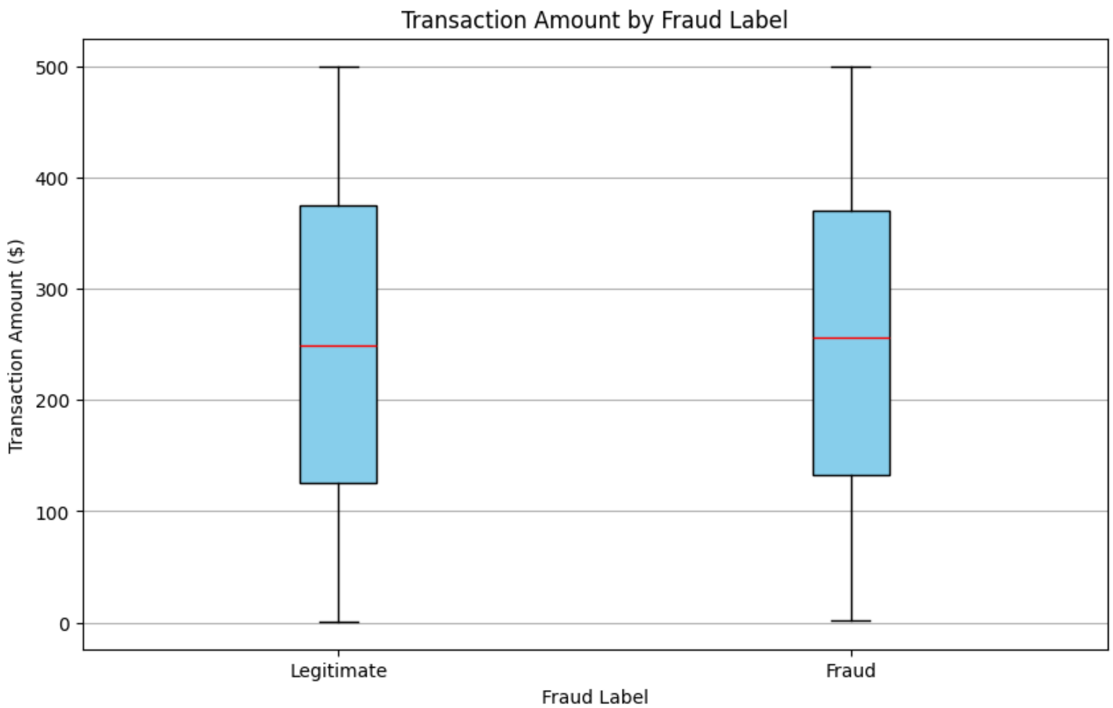
Analysis:

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

[177]:
```
# 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
```

```python
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:

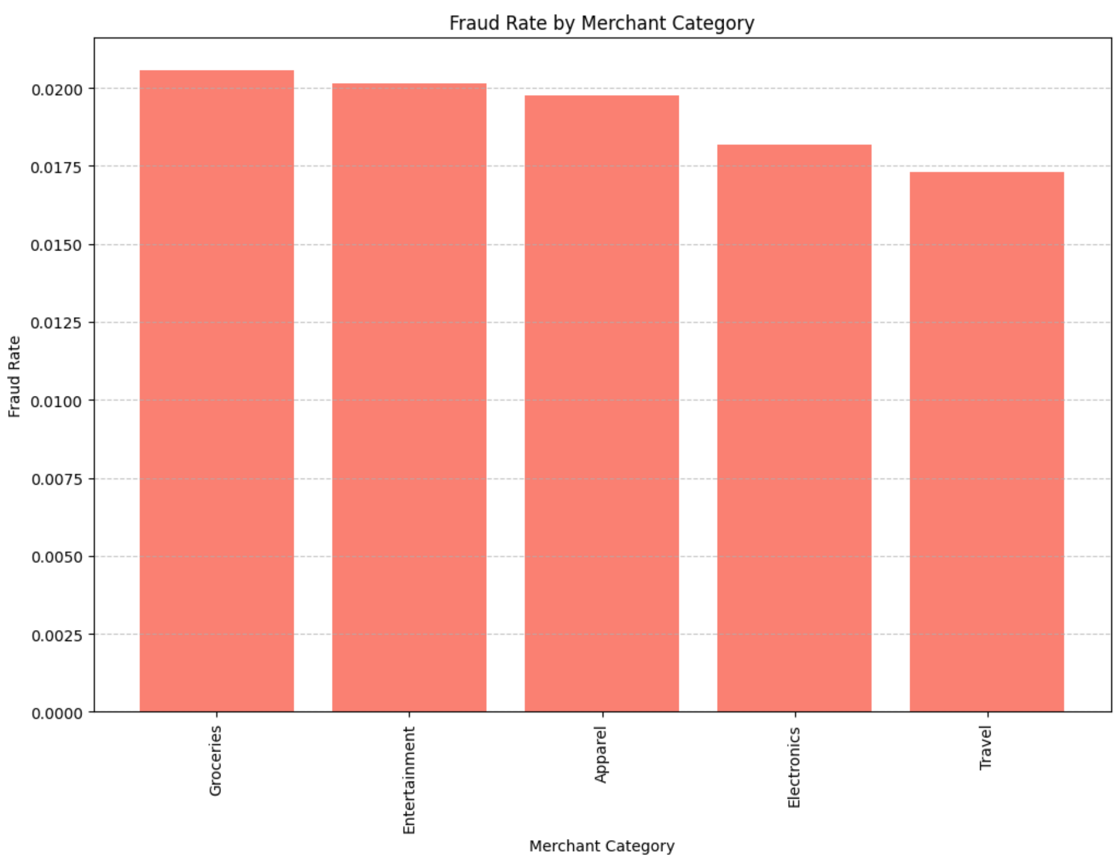
A box plot was created to compare the transaction amounts for fraudulent and legitimate transactions. To improve visibility, the range was limited to transactions under $500, because there are no transactions above that value.

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

[178]:
```
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:

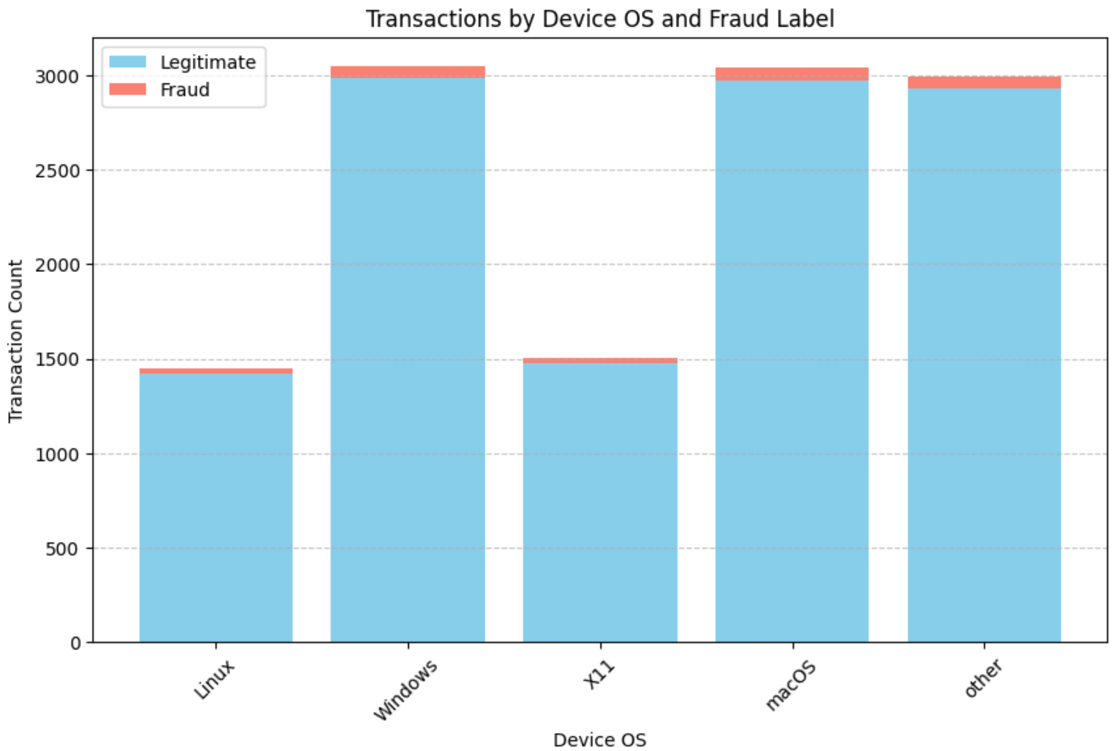
A bar plot was used to display the average fraud rate for each merchant category, calculated by grouping the data by `category` and taking the mean of `is_fraud`.

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

[179]:
```
# 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).

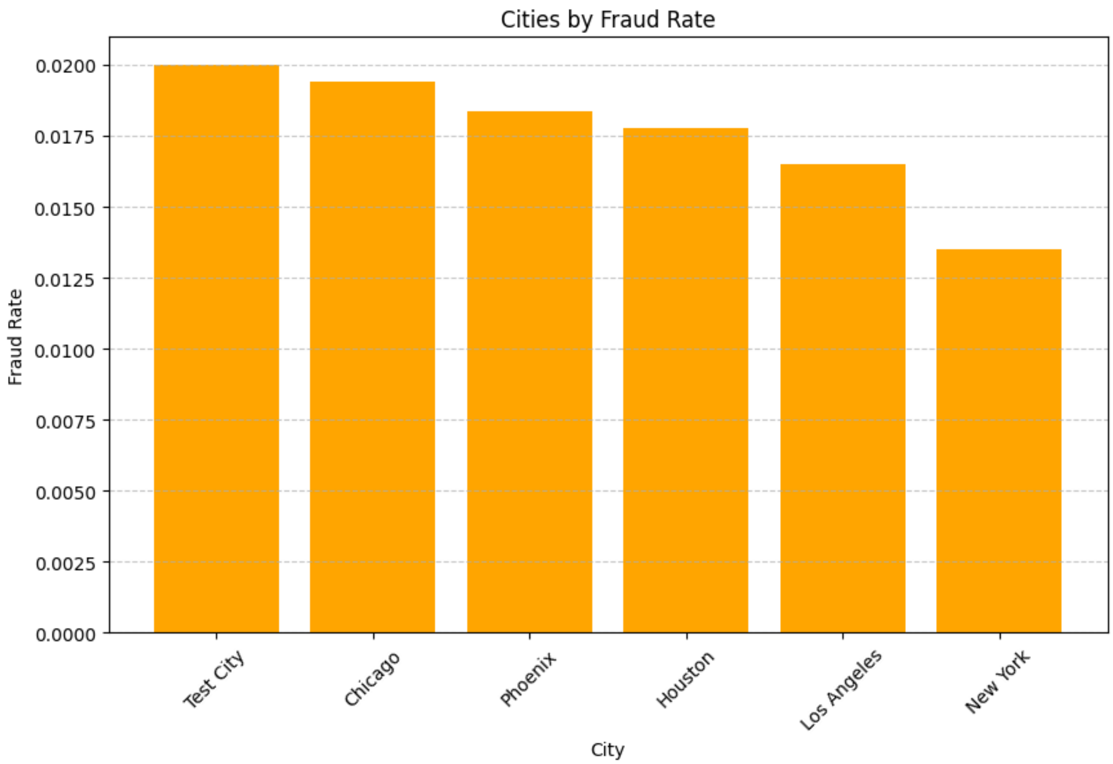
Analysis:

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

[180]:
```
# 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:

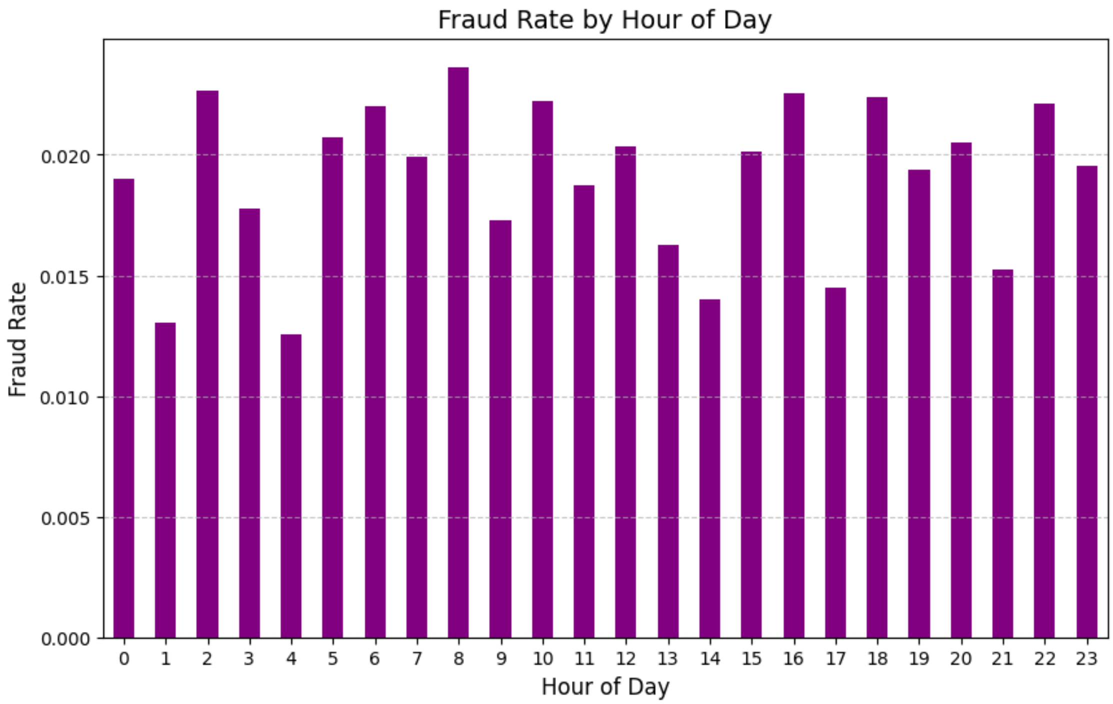
The chart indicates that major metropolitan areas such as Chicago, Phoenix and Houston have higher fraud rates. Upon further inspection, the inclusion of "Test City" appears to be a synthetic or placeholder entry in the dataset rather than a real location. This type of entry is likely used for testing 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

[181]:
```
# Convert Unix time to datetime and extract the hour
merged_data['hour'] = pd.to_datetime(merged_data['unix_time'], unit='s').dt.hour
```

```python
# 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:

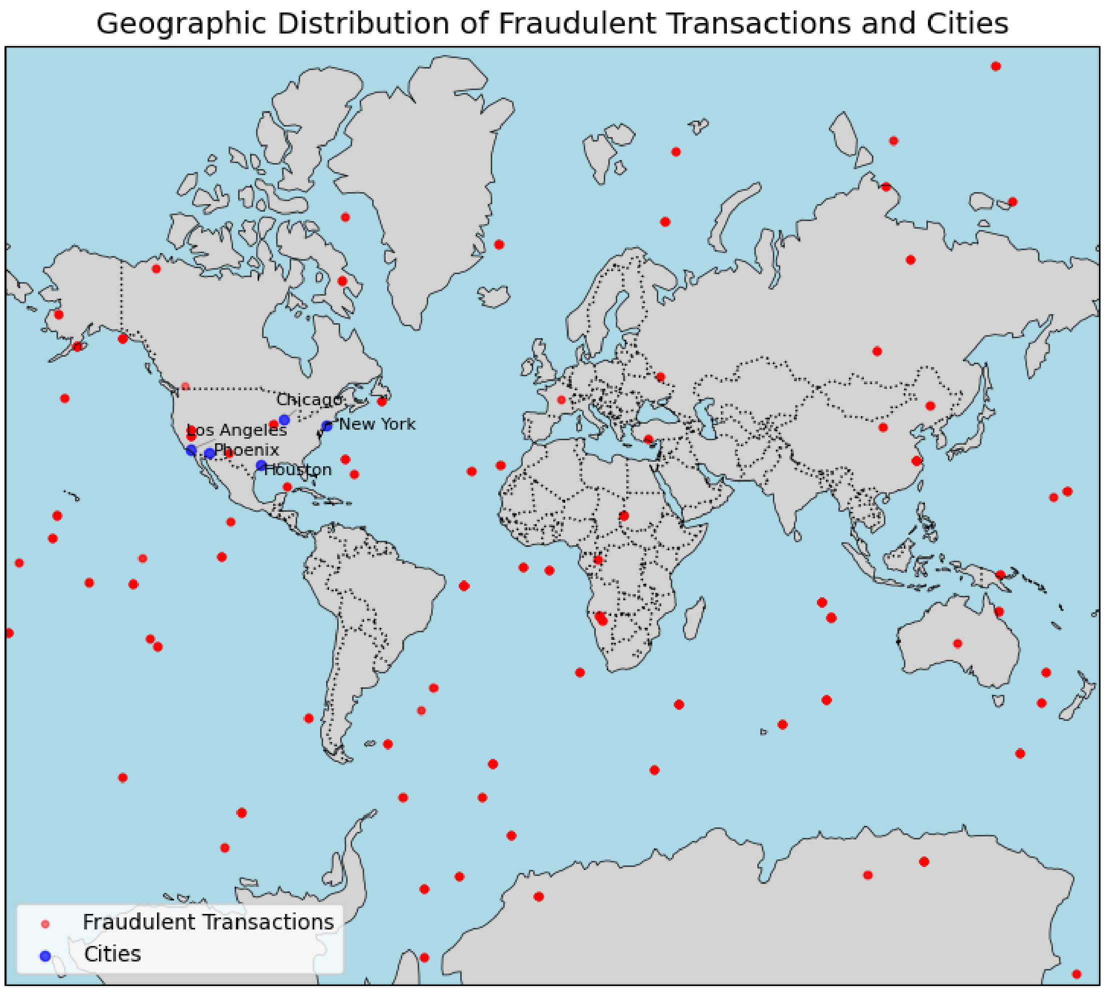
While there are peaks and dips, the fraud rate does not vary drastically across hours, suggesting that fraud occurs throughout the day with certain periods being slightly riskier.

Geographic Distribution of Fraudulent Transactions and Cities

[182]:

```python
# 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.

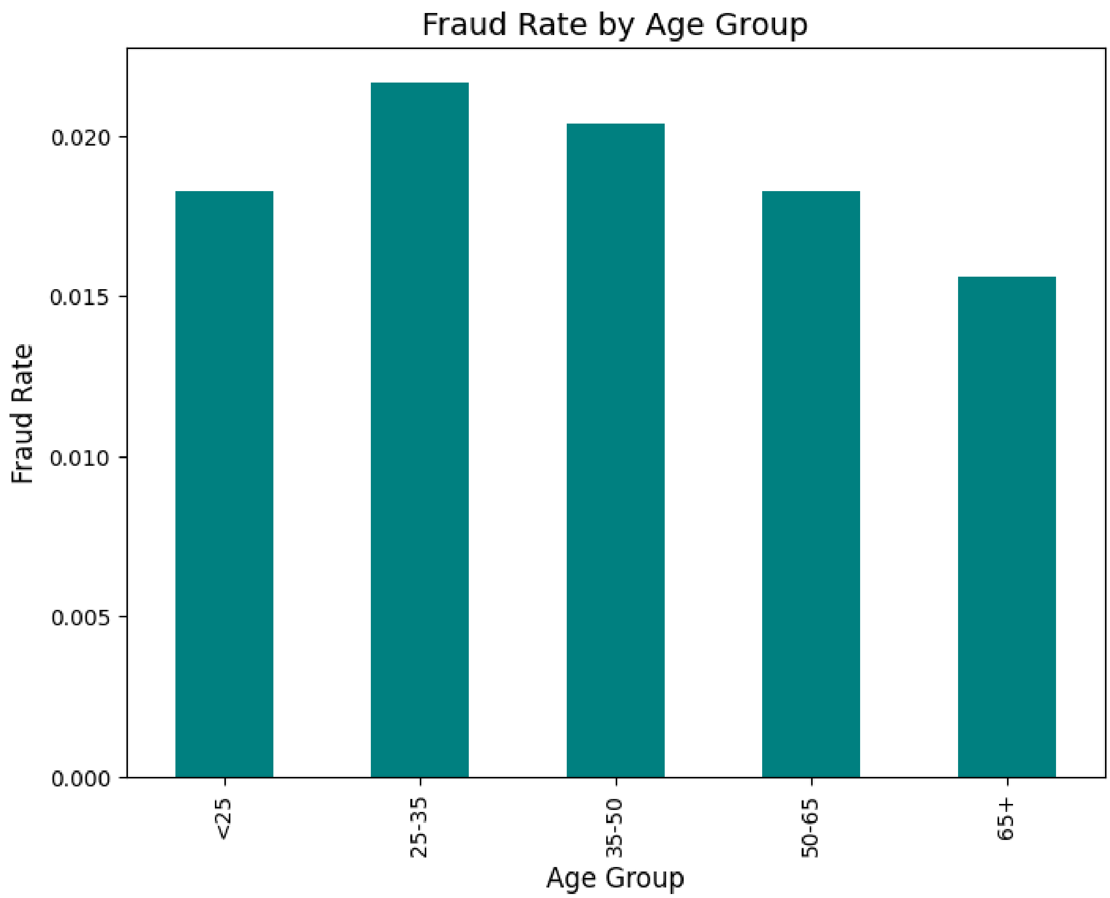
Analysis:

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

[183]:
```
# 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()

# 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_28913/3298143012.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.

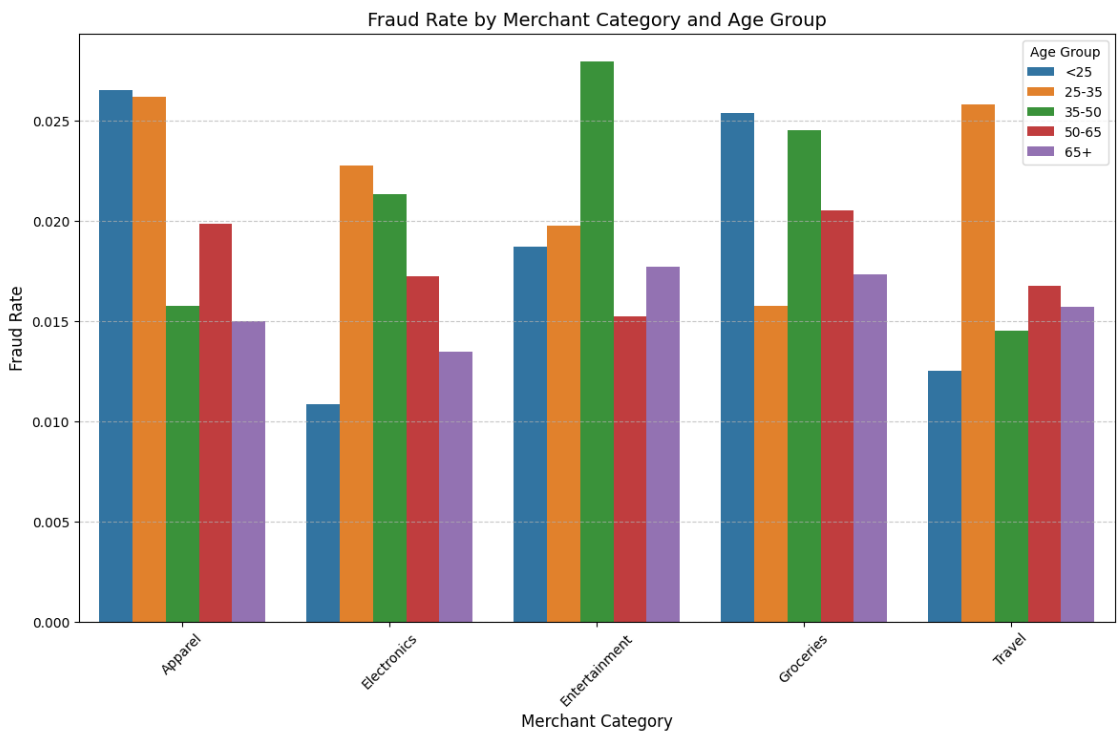
Analysis:

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

[184]:
```
# 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()

# 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.

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 Handle Missing Values

[185]:
```python
# Check for missing values
print("\nMissing Values:")
print(merged_data.isnull().sum())
```

```
Missing Values:
index                        0
trans_date_trans_time      100
cc_num                       0
device_os                17964
merchant                     0
amt                        100
trans_num                    0
unix_time                    0
is_fraud                     0
first                       10
last                        10
gender                      10
street                      10
city                        10
zip                        216
job                        216
dob                         10
category                   599
merch_lat                  599
merch_long                  10
```

```
merchant_id              10
lat                   19980
long                  19980
city_pop              19980
state                 19980
hour                      0
age                      10
age_group                10
dtype: int64
```

[186]:
```python
merged_data['amt'] = merged_data['amt'].fillna(merged_data['amt'].mean())
```

[187]:
```python
merged_data['category'] = merged_data['category'].fillna('Unknown')
```

[188]:
```python
merged_data.dropna(subset=['lat', 'long'], inplace=True)
```

[189]:
```python
# Check for missing values
print("\nMissing Values:")
print(merged_data.isnull().sum())
```

```
Missing Values:
index                       0
trans_date_trans_time     100
cc_num                      0
device_os                5993
merchant                    0
amt                         0
trans_num                   0
unix_time                   0
is_fraud                    0
first                       0
last                        0
gender                      0
street                      0
city                        0
zip                       206
job                       206
dob                         0
category                    0
merch_lat                 205
merch_long                  0
merchant_id                 0
lat                         0
long                        0
city_pop                    0
state                       0
hour                        0
```

```
age                      0
age_group                0
dtype: int64
```

2.3.2 Handle Duplicate Values

[190]:
```python
# Check for duplicate transactions
print("\nDuplicate Transactions:")
print(merged_data.duplicated(subset='trans_num').sum())
```

```
Duplicate Transactions:
78
```

[191]:
```python
# Delete duplicate transactions
merged_data = merged_data.drop_duplicates(subset='trans_num', keep='first')
```

2.3.3 Encode Categorical Variables

[192]:
```python
# One-hot encoding example
merged_data = pd.get_dummies(merged_data, columns=['category', 'device_os'],␣
 ↪drop_first=True)
```

2.3.4 Feature Engineering

[193]:
```python
merged_data['hour'] = pd.to_datetime(merged_data['trans_date_trans_time']).dt.
 ↪hour
merged_data['day_of_week'] = pd.
 ↪to_datetime(merged_data['trans_date_trans_time']).dt.dayofweek
merged_data['month'] = pd.to_datetime(merged_data['trans_date_trans_time']).dt.
 ↪month
```

[194]:
```python
merged_data['age'] = 2023 - pd.to_datetime(merged_data['dob']).dt.year
```

[195]:
```python
merged_data['distance'] = merged_data.apply(lambda row: geodesic((row['lat'],␣
 ↪row['long']), (row['merch_lat'], row['merch_long'])).km, axis=1)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[195], line 1
----> 1 merged_data['distance'] =␣
 ↪merged_data.apply(lambda row: geodesic((row['lat'], row['long']), (row['merch_lat'], row['r

File ~/Desktop/uni/mestrado/fraude/project/venv/lib/python3.12/site-packages/
 ↪pandas/core/frame.py:10374, in DataFrame.apply(self, func, axis, raw,␣
 ↪result_type, args, by_row, engine, engine_kwargs, **kwargs)
  10360 from pandas.core.apply import frame_apply
  10362 op = frame_apply(
```

```
  10363     self,
  10364     func=func,
   (…)
  10372     kwargs=kwargs,
  10373 )
> 10374 return op.apply().__finalize__(self, method="apply")

File ~/Desktop/uni/mestrado/fraude/project/venv/lib/python3.12/site-packages/
↪pandas/core/apply.py:916, in FrameApply.apply(self)
    913 elif self.raw:
    914     return self.apply_raw(engine=self.engine, engine_kwargs=self.
↪engine_kwargs)
--> 916 return self.apply_standard()

File ~/Desktop/uni/mestrado/fraude/project/venv/lib/python3.12/site-packages/
↪pandas/core/apply.py:1063, in FrameApply.apply_standard(self)
   1061 def apply_standard(self):
   1062     if self.engine == "python":
-> 1063         results, res_index = self.apply_series_generator()
   1064     else:
   1065         results, res_index = self.apply_series_numba()

File ~/Desktop/uni/mestrado/fraude/project/venv/lib/python3.12/site-packages/
↪pandas/core/apply.py:1081, in FrameApply.apply_series_generator(self)
   1078 with option_context("mode.chained_assignment", None):
   1079     for i, v in enumerate(series_gen):
   1080         # ignore SettingWithCopy here in case the user mutates
-> 1081         results[i] = self.func(v, *self.args, **self.kwargs)
   1082         if isinstance(results[i], ABCSeries):
   1083             # If we have a view on v, we need to make a copy because
   1084             #  series_generator will swap out the underlying data
   1085             results[i] = results[i].copy(deep=False)

Cell In[195], line 1, in <lambda>(row)
----> 1 merged_data['distance'] = merged_data.apply(lambda row:␣
↪geodesic((row['lat'], row['long']), (row['merch_lat'], row['merch_long'])).km␣
↪axis=1)

File ~/Desktop/uni/mestrado/fraude/project/venv/lib/python3.12/site-packages/
↪geopy/distance.py:540, in geodesic.__init__(self, *args, **kwargs)
    538 self.set_ellipsoid(kwargs.pop('ellipsoid', 'WGS-84'))
    539 major, minor, f = self.ELLIPSOID
--> 540 super().__init__(*args, **kwargs)

File ~/Desktop/uni/mestrado/fraude/project/venv/lib/python3.12/site-packages/
↪geopy/distance.py:276, in Distance.__init__(self, *args, **kwargs)
    274 elif len(args) > 1:
    275     for a, b in util.pairwise(args):
```

```
--> 276         kilometers += self.measure(a, b)
    278 kilometers += units.kilometers(**kwargs)
    279 self.__kilometers = kilometers

File ~/Desktop/uni/mestrado/fraude/project/venv/lib/python3.12/site-packages/
↪geopy/distance.py:556, in geodesic.measure(self, a, b)
    555 def measure(self, a, b):
--> 556     a, b = Point(a), Point(b)
    557     _ensure_same_altitude(a, b)
    558     lat1, lon1 = a.latitude, a.longitude

File ~/Desktop/uni/mestrado/fraude/project/venv/lib/python3.12/site-packages/
↪geopy/point.py:175, in Point.__new__(cls, latitude, longitude, altitude)
    171             raise TypeError(
    172                 "Failed to create Point instance from %r." % (arg,)
    173             )
    174         else:
--> 175             return cls.from_sequence(seq)
    177 if single_arg:
    178     raise ValueError(
    179         'A single number has been passed to the Point '
    180         'constructor. This is probably a mistake, because '
   (…)
    184         'to get rid of this error.'
    185     )

File ~/Desktop/uni/mestrado/fraude/project/venv/lib/python3.12/site-packages/
↪geopy/point.py:472, in Point.from_sequence(cls, seq)
    469 if len(args) > 3:
    470     raise ValueError('When creating a Point from sequence, it '
    471                      'must not have more than 3 items.')
--> 472 return cls(*args)

File ~/Desktop/uni/mestrado/fraude/project/venv/lib/python3.12/site-packages/
↪geopy/point.py:188, in Point.__new__(cls, latitude, longitude, altitude)
    177 if single_arg:
    178     raise ValueError(
    179         'A single number has been passed to the Point '
    180         'constructor. This is probably a mistake, because '
   (…)
    184         'to get rid of this error.'
    185     )
    187 latitude, longitude, altitude = \
--> 188     _normalize_coordinates(latitude, longitude, altitude)
    190 self = super().__new__(cls)
    191 self.latitude = latitude
```

```
File ~/Desktop/uni/mestrado/fraude/project/venv/lib/python3.12/site-packages/
↪geopy/point.py:63, in _normalize_coordinates(latitude, longitude, altitude)
     61 is_all_finite = all(isfinite(x) for x in (latitude, longitude, altitude))
     62 if not is_all_finite:
---> 63     raise ValueError('Point coordinates must be finite. %r has been␣
↪passed '
     64                      'as coordinates.' % ((latitude, longitude,␣
↪altitude),))
     66 if abs(latitude) > 90:
     67     warnings.warn('Latitude normalization has been prohibited in the␣
↪newer '
     68                   'versions of geopy, because the normalized value␣
↪happened '
     69                   'to be on a different pole, which is probably not what␣
↪was '
   (…)
     72                   '(latitude, longitude) or (y, x) in Cartesian terms.'
     73                   UserWarning, stacklevel=3)

ValueError: Point coordinates must be finite. (nan, 76.43321219151005, 0.0) has
↪been passed as coordinates.
```

2.3.5 Normalize/Scale Numerical Features

[ ]:
```
numerical_columns = ['amt', 'age', 'city_pop', 'distance'] # Maybe add more␣
↪numeric collums
scaler = StandardScaler()
merged_data[numerical_columns] = scaler.
↪fit_transform(merged_data[numerical_columns])
```

2.3.6 Class Imbalance

[ ]:
```
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, y)
```

2.3.7 Drop Redundant or Unnecessary Columns

[ ]:
```
merged_data.drop(['index', 'trans_num', 'unix_time'], axis=1, inplace=True)
```

2.3.8 Split Data into Train and Test Sets

[ ]:
```
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)
```

2.4 1.3- Clustering

2.4.1 DBSCAN

[ ]:
```
# 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()
```

2.4.2 K-Means

[ ]:
```
# 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]
```

```python
# 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()
```

3 Task 2: Predictive Modelling