Fraud Detection Model

Dicember 2024

Dataset Description

A simulated credit card dataset containing legitimate and fraudulent transactions from 01/01/2019 to 21/06/2020 has been used to train the model. The dataset includes approximately 1.3 million transactions from 983 customers, providing for each of them the following information:

1. Name (first, last)
2. Gender (gender)
3. Job (job)
4. Date of Birth (dob)
5. Address (Street, City, State, Zip, Lat, Long)
6. City Population (city_pop)
7. Credit Card Number (cc_num)

Additionally, the dataset includes details about the transactions made by each customer, such as:

1. Date and Time (trans_date_trans_time)
2. Item Purchased (merchant, category)
3. Transaction Number (trans_num)
4. Transaction Timestamp Encoding (unix_time)
5. Merchant's Location (merch_lat, merch_long)
6. Fraud Indicator (is_fraud)

Objectives

The aim of this report is to develop and identify a reliable model capable of detecting illegitimate transactions made by individuals using point-of-sale (P.O.S) systems. These fraudulent activities represent a small fraction of all transactions, making them particularly challenging to identify. The focus will be on creating a robust and accurate framework that can effectively handle the rarity of such events.

Plan

The initial step involves correctly reading the dataset into Python, ensuring that all data is loaded without errors. Once the data is successfully imported, a thorough inspection of the variables is conducted to identify potential issues, such as errors, inconsistencies, or missing values.

Following this, a detailed exploratory data analysis (EDA) is performed. This phase includes visualizing the data, understanding the relationships between variables, and summarizing key patterns or anomalies that might influence the modeling process.

After preparing the data, different predictive or analytical models are implemented and evaluated.

Finally, the best-performing model is selected based on its results from the performance evaluation, ensuring it aligns with the project objectives. The chosen model is then interpreted and validated to confirm its robustness.

Chapter 1

Exploratory Data Analysis (EDA)

1.1 Data Inspection

After loading the data on Colab, the first thing to do is to check if Python correctly reads the variables.

trans_date_trans_time	object
cc_num	int64
merchant	object
category	object
amt	float64
first	object
last	object
gender	object
street	object
city	object
state	object
zip	int64
lat	float64
long	float64
city_pop	int64
job	object
dob	object
trans_num	object
unix_time	int64
merch_lat	float64
merch_long	float64
is_fraud	int64

Figure 1.1: Python Data Types

Most variables aren't correctly recognized by the system. To properly analyze the data, it's necessary to adjust them.

```
from datetime import datetime
for i in (1,2,3, 7,8,9,10,15,21):
  df[df.columns[i]] = df[df.columns[i]].astype('category')

for i in (5,6,17):
  df[df.columns[i]] = df[df.columns[i]].astype('string')



df['trans_date_trans_time'] = pd.to_datetime(df['trans_date_trans_time'],errors="coerce")
df['dob'] = pd.to_datetime(df['dob'])
```

This code selects the positions of the columns that need their data types to be changed and transforms them into categorical, strings or other appropriate data types.

trans_date_trans_time	datetime64[ns]
cc_num	category
merchant	category
category	category
amt	float64
first	string[python]
last	string[python]
gender	category
street	category
city	category
state	category
zip	int64
lat	float64
long	float64
city_pop	int64
job	category
dob	datetime64[ns]
trans_num	string[python]
unix_time	int64
merch_lat	float64
merch_long	float64
is_fraud	category

Figure 1.2: Corrected Data Types

The variables are correctly represented.

1.1.1 Missing Values Check

It's important to check if there are any missing values in the data.

```
np.sum(df.isna())
```

trans_date_trans_time	0
cc_num	0
merchant	0
category	0
amt	0
first	0
last	0
gender	0
street	0
city	0
state	0
zip	0
lat	0
long	0
city_pop	0
job	0
dob	0
trans_num	0
unix_time	0
merch_lat	0
merch_long	0
is_fraud	0

Figure 1.3: Missing Values x Variable

At first glance, there are no missing values in the data. However, we have a variable <trans_date_trans_time>, which is the combination of both the transaction date and time. It is possible that one transaction has the date but not the time, or vice versa. We should split the variable into <trans_date and trans_time> and check for any missing values.

```
df[['trans_date', 'trans_time']] = df['trans_date_trans_time'].str.split(' ', expand=True)
np.sum(df["trans_date"].isna())
```

```
0
```

```
np.sum(df["trans_time"].isna())
```

```
0
```

There are no missing values in the dataset

1.2 Data Anonymization and Feature Transformation

A valid fraud detection model shouldn't be able to recognize specific individuals or specific points in space too closely, otherwise we could risk going in overfitting.

Before doing any other transformation on the dataset, a copy of the original one should be made in order to preserve the original data and easily recover it if needed.

```
df1=df.copy()
```

1.2.1 Feature Removal

The variables that make the transaction unique are dropped.

```
df1 = df1.drop(["cc_num","first","last","trans_num"], axis = 1)
df1.head()
```

	trans_date_trans_time	merchant	category	amt	gender	street	city	state	zip	lat	long	city_pop	job	dob	unix_time	merch_lat	merch_long	is_fraud
0	2019-01-01 00:00:18	fraud_Rippin, Kub and Mann	misc_net	4.97	F	561 Perry Cove	Moravian Falls	NC	28654	36.0788	-81.1781	3495	Psychologist, counselling	1988-03-09	1325376018	36.011293	-82.048315	0
1	2019-01-01 00:00:44	fraud_Heller, Gutmann and Zieme	grocery_pos	107.23	F	43039 Riley Greens Suite 393	Orient	WA	99160	48.8878	-118.2105	149	Special educational needs teacher	1978-06-21	1325376044	49.159047	-118.186462	0
2	2019-01-01 00:00:51	fraud_Lind-Buckridge	entertainment	220.11	M	594 White Dale Suite 530	Malad City	ID	83252	42.1808	-112.2620	4154	Nature conservation officer	1962-01-19	1325376051	43.150704	-112.154481	0
3	2019-01-01 00:01:16	fraud_Kutch, Hermiston and Farrell	gas_transport	45.00	M	9443 Cynthia Court Apt. 038	Boulder	MT	59632	46.2306	-112.1138	1939	Patent attorney	1967-01-12	1325376076	47.034331	-112.561071	0
4	2019-01-01 00:03:06	fraud_Keeling-Crist	misc_pos	41.96	M	408 Bradley Rest	Doe Hill	VA	24433	38.4207	-79.4629	99	Dance movement psychotherapist	1986-03-28	1325376186	38.674999	-78.632459	0

1.2.2 Extraction of the Haversine Distance

The idea is to transform <lat, long, merch_lat and merch_long> into one variable that measures the distance between the customer and the merchant. By computing only the difference between the two latitudes and longitudes we obtain the angular difference between the two points. Haversine formula instead gives us the orthodromic distance, which is preferable.

```
from haversine import haversine_vector, Unit
df1["distance"]=0
for i in range (len(df1)):
  df1["distance"][i]=haversine_vector((df1["lat"][i], df1["long"][i]), (df1["merch_lat"][i],df1["
      merch_long"][i]), unit=Unit.KILOMETERS)
df1["distance"]=np.round(df1["distance"],3)
df1.head()
```

	trans_date_trans_time	merchant	category	amt	gender	street	city	state	zip	lat	long	city_pop	job	dob	unix_time	merch_lat	merch_long	is_fraud	distance
0	2019-01-01 00:00:18	fraud_Rippin, Kub and Mann	misc_net	4.97	F	561 Perry Cove	Moravian Falls	NC	28654	36.0788	-81.1781	3495	Psychologist, counselling	1988-03-09	1325376018	36.011293	-82.048315	0	78.598
1	2019-01-01 00:00:44	fraud_Heller, Gutmann and Zieme	grocery_pos	107.23	F	43039 Riley Greens Suite 393	Orient	WA	99160	48.8878	-118.2105	149	Special educational needs teacher	1978-06-21	1325376044	49.159047	-118.186462	0	30.212
2	2019-01-01 00:00:51	fraud_Lind-Buckridge	entertainment	220.11	M	594 White Dale Suite 530	Malad City	ID	83252	42.1808	-112.2620	4154	Nature conservation officer	1962-01-19	1325376051	43.150704	-112.154481	0	108.206
3	2019-01-01 00:01:16	fraud_Kutch, Hermiston and Farrell	gas_transport	45.00	M	9443 Cynthia Court Apt. 038	Boulder	MT	59632	46.2306	-112.1138	1939	Patent attorney	1967-01-12	1325376076	47.034331	-112.561071	0	95.673
4	2019-01-01 00:03:06	fraud_Keeling-Crist	misc_pos	41.96	M	408 Bradley Rest	Doe Hill	VA	24433	38.4207	-79.4629	99	Dance movement psychotherapist	1986-03-28	1325376186	38.674999	-78.632459	0	77.557

Contextually we drop the latitude and longitude of both customer and merchant:

```
df1=df1.drop(["lat","long","merch_lat","merch_long"],axis=1)
```

1.2.3 Transaction Date and Time vs Unix Time

In the following it's proved that <unix_time> is just an encoding of <trans_date_trans_time>:

```
print("trans_date_trans_time:",df1["trans_date_trans_time"][0], "unix_time:",df1["unix_time"][0])
print("trans_date_trans_time:",df1["trans_date_trans_time"][81], "unix_time:",df1["unix_time"][81])
print("Difference between 2 units in trans_date_trans_time:", df1["trans_date_trans_time"][81] - df1
    ["trans_date_trans_time"][0])
print("Difference between 2 units unix time:", df1["unix_time"][81] - df1["unix_time"][0])
```

```
trans_date_trans_time: 2019-01-01 00:00:18 unix_time: 1325376018
trans_date_trans_time: 2019-01-01 01:00:19 unix_time: 1325379619
Difference between 2 units in trans_date_trans_time: 0 days 01:00:01
Difference between 2 units unix time: 3601
```

The difference between the two units in <trans_date_trans_time> corresponds to 1h and 1sec which equals to 3601sec that corresponds to the difference between the two units in <unix_time>. Having two redundant variables can be misleading; therefore, one of them should be dropped to avoid redundancy and improve model interpretability.
Fraud detection might not depend on the exact timestamp but could exhibit patterns based on specific time periods or categories. For these reason we drop <unix_time> and keep <trans_date_trans_time>.

```
df1=df1.drop(["unix_time"],axis=1)
```

1.2.4 Extraction of Age Variable

From the date of birth (<dob>) we can extract the age of the costumer at the time of the transaction.

```
df1['age'] = (df1['trans_date_trans_time'] - df1['dob']).dt.days // 365
df1['age'] = df1['age'].astype('int')
```

To avoid redundancy in our variables, we drop <dob>.

```
df1=df1.drop(["dob"],axis=1)
```

1.2.5 Address Analysis

The address of the customer is composed by <Street, City, State and Zip>. <Zip> can be eliminated due to it's redundancy. The <Street> is generally too specific and often includes house numbers, which are not necessary for the analysis in most cases. Therefore, it can be eliminated too.

```
df1=df1.drop(["zip","street"],axis=1)
```

1.3 Categorical Variables

In this section we'll study the relation between the categorical variables and our target variable. Before doing this it's convenient to create a copy of <df1>, which should have all the features correctly addressed.

```
df2=df1.copy()
```

1.3.1 Fraud

Our target variable is <is_fraud>. It's a boolean variable with:

- is_fraud = 1 ⇒The transaction is a fraud
- is_fraud = 0 ⇒The transaction is legitimate

Let's see how it is distributed in the dataset.

```
ax = (df2.groupby(["is_fraud"]).size() / len(df2) * 100).plot(kind='bar', figsize=(8, 5))
for bar in ax.patches:
    ax.text(
        bar.get_x() + bar.get_width() / 2,
        bar.get_height(),
        f'{bar.get_height():.2f}%',
        ha='center',
        va='bottom',
        fontsize=10
    )
plt.title('Fraud Percentage Distribution')
plt.xlabel('Is Fraud (0 = No, 1 = Yes)')
plt.ylabel('Percentage (%)')
plt.tight_layout()
plt.show()
```

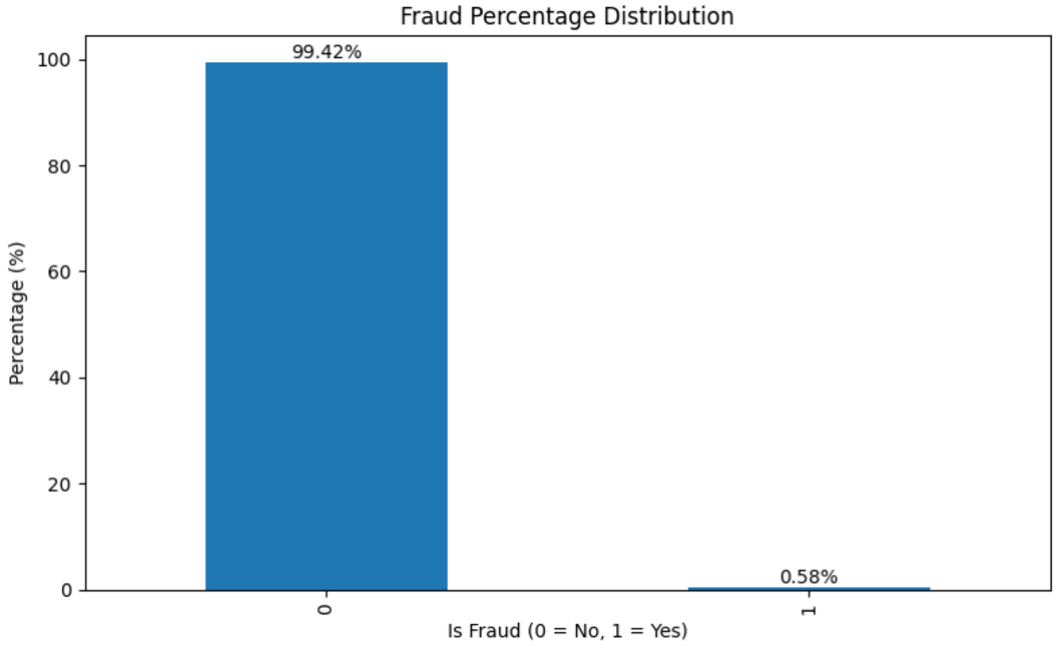


Figure 1.4: Legitimate and Illegitimate Transactions

There is a clear problem of imbalanced data.

1.3.2 Transaction Day

```
df2['d'] = df2['trans_date_trans_time'].dt.day_name()
df2["d"] = df2["d"].astype("category")
(df2.groupby('d').size()).reindex(
    ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
).plot(kind='bar')

plt.title('Number of Transactions by Day of the Week')
plt.xlabel('Day of the Week')
plt.ylabel('Frequency')
plt.show()
```

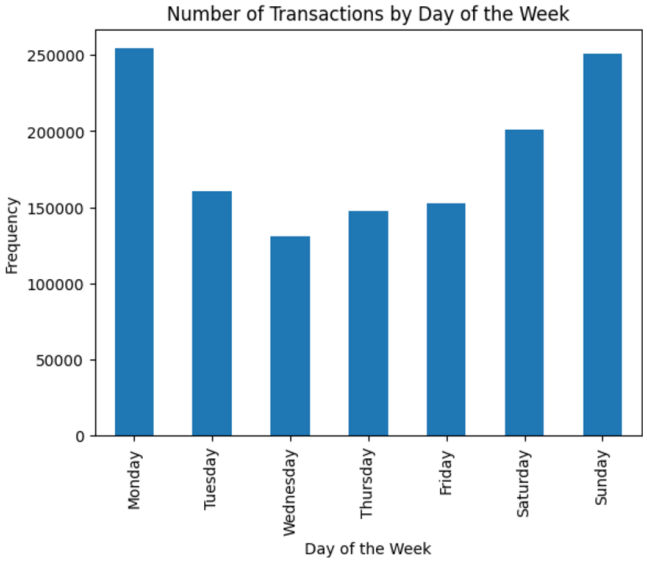


Figure 1.5: Transaction Days x Week

Monday and Sunday stand out as the most active days, with transaction volumes exceeding 250,000, suggesting a peak in activity at the start and end of the week. In contrast, Wednesday sees the lowest number of transactions, indicating a mid-week dip. This pattern could reflect behavioral trends in transaction habits, such as weekend shopping or work-week financial activity.

```
((df2[df2["is_fraud"] == 1].groupby(df2['d']).size() * 100) /(df2.groupby(df2['d']).size())).reindex
    (['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']).plot(kind="bar"
    , color="r")
plt.axhline(y=0.58, color="b", linestyle="--", linewidth=2, label="Threshold 0.58")
plt.title('% of Fraudulent Transactions by Day of the Week')
plt.xlabel('Day of the Week')
plt.ylabel('% of Fraudulent Transactions')
plt.legend()
plt.show()
```

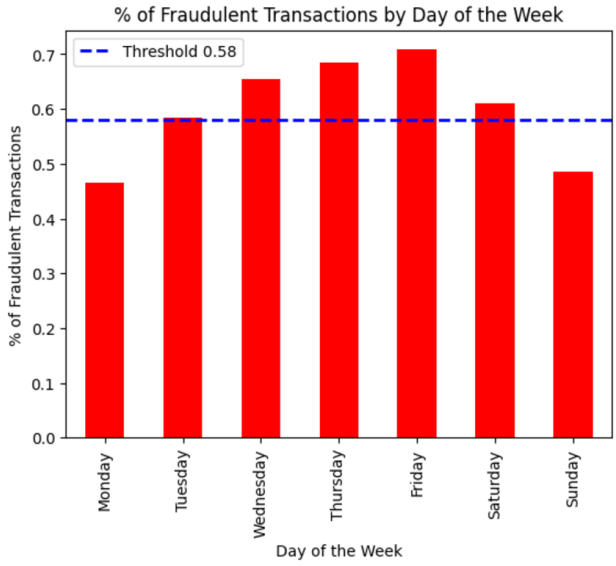


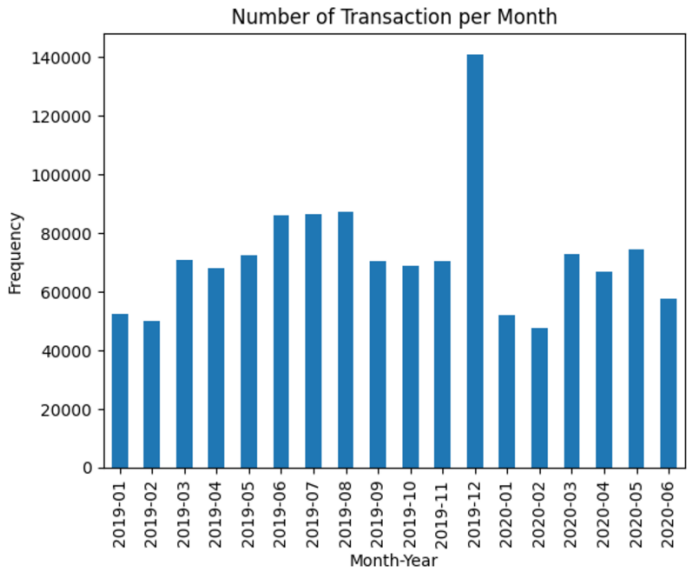
Figure 1.6: % Frauds x Day of the Week

Tuesday and Thursday show the highest rates of fraud, exceeding the threshold of 0.58, indicating these days might be more prone to fraudulent activity. Other days, like Monday, Friday, and Saturday, display moderate levels of fraud, while Sunday stands out with the lowest percentage. This pattern suggests that fraudulent activity is not evenly distributed throughout the week and could inform targeted fraud detection efforts.

1.3.3 Transaction Year-Month

We can plot the frequencies of legitimate and fraudulent transactions for each month.

```
df2['ym'] = df2['trans_date_trans_time'].dt.strftime('%Y-%m')
df2["ym"] = df2["ym"].astype("category")
(df2.groupby('ym').size()).plot(kind ='bar')
plt.title('Number of Transaction per Month')
plt.xlabel('Month-Year')
plt.ylabel('Frequency')
plt.show()
```

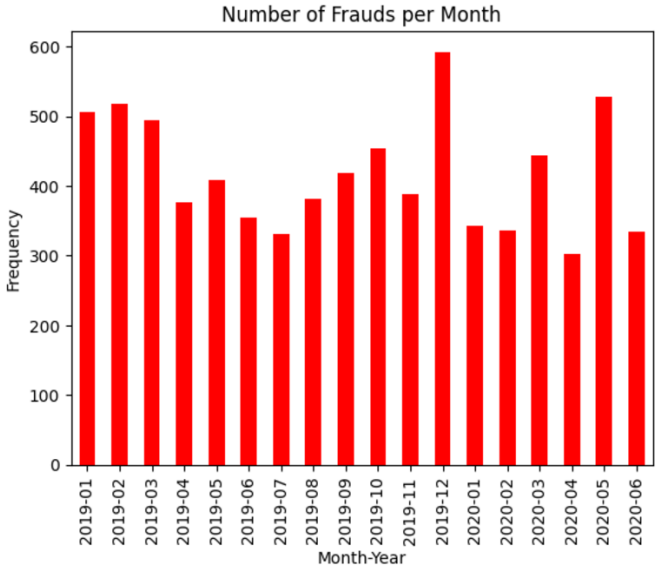


December is by far the month with the most transactions, likely due to Christmas. January and February are probably affected by the high volume of transactions in December, as the frequencies in both 2019 and 2020 are almost the same. During the rest of the year, the number of transactions remains constant, with a slight shift during the summer months.

Figure 1.7: Number of Transactions per Month

We can see the same plot, but only with the fraudulent transactions.

```
df2[df2["is_fraud"]==1].groupby("ym").size().plot(kind='bar',color="r")
plt.title('Number of Fraudulent Transactions per Month')
plt.xlabel('Month-Year')
plt.ylabel('Frequency')
plt.show()
```

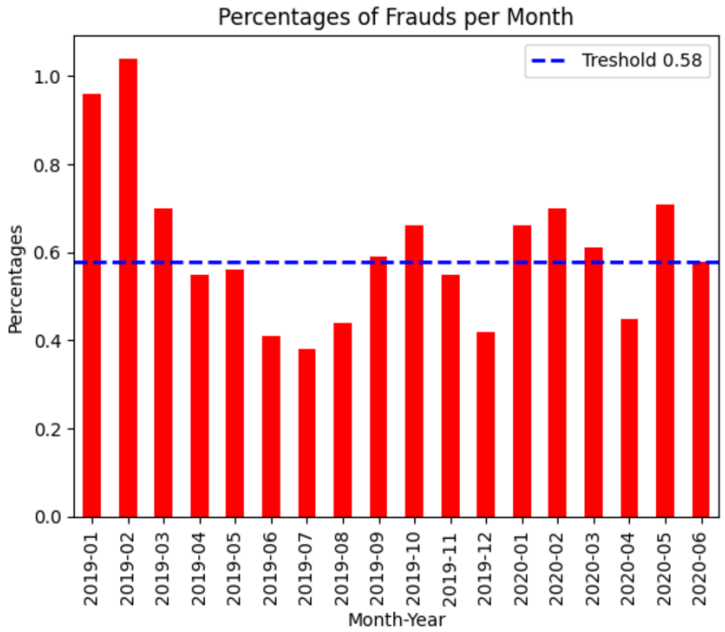


December is the month with the highest number of frauds, reflecting the fact that it also has the most transactions overall. However, some months have a disproportionately high number of frauds compared to the total transactions, such as January 2019, February 2019, March 2020, and May 2020

Figure 1.8: Number of Frauds per Month

A way to analyze the number of frauds per month without being influenced by the total number of transactions in the corresponding month is to calculate the percentages.

```
round((df2[df2['is_fraud'] == 1].groupby('ym').size()*100)/df2['is_fraud'].groupby(df2["ym"]).size()
    ,2).plot(kind="bar",color="r")
plt.axhline(y=0.58,color="b",linestyle="--",linewidth=2, label="Threshold 0.58")
plt.title('Percentages of Frauds per Month')
plt.xlabel('Month-Year')
plt.ylabel('Percentages')
plt.legend()
plt.show()
```



We can observe that the highest percentage of frauds was committed in January and February of 2019. Although December had the highest number of frauds in absolute terms, it has one of the lowest percentages.

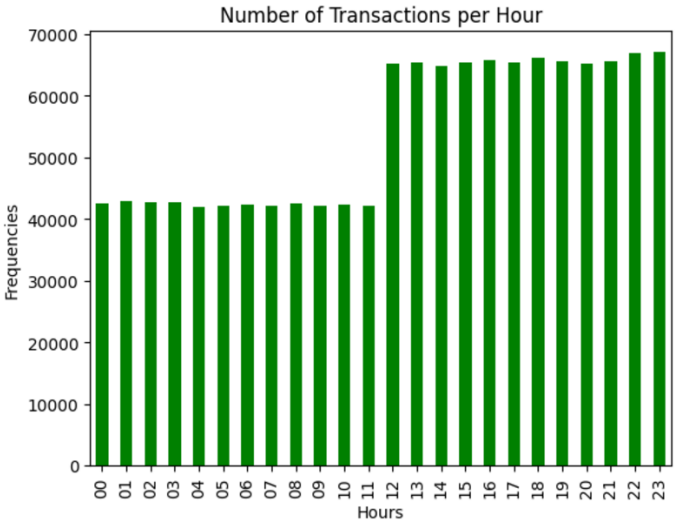
Figure 1.9: Percentage of Frauds per Month

The percentages of frauds for each month do not align with the overall fraud percentages in the dataset, suggesting that the transaction month might be an explanatory variable.

1.3.4 Transaction Time

The same analysis that has been made on the transaction day and month can be made on the transaction time.

```
df2['hour'] = df2['trans_date_trans_time'].dt.strftime('%H')
df2["hour"] = df2["hour"].astype("category")
(df2.groupby("hour").size()).plot(kind="bar",color="g")
plt.title('Number of Transactions per Hour')
plt.xlabel('Hours')
plt.ylabel('Frequencies')
plt.show()
```

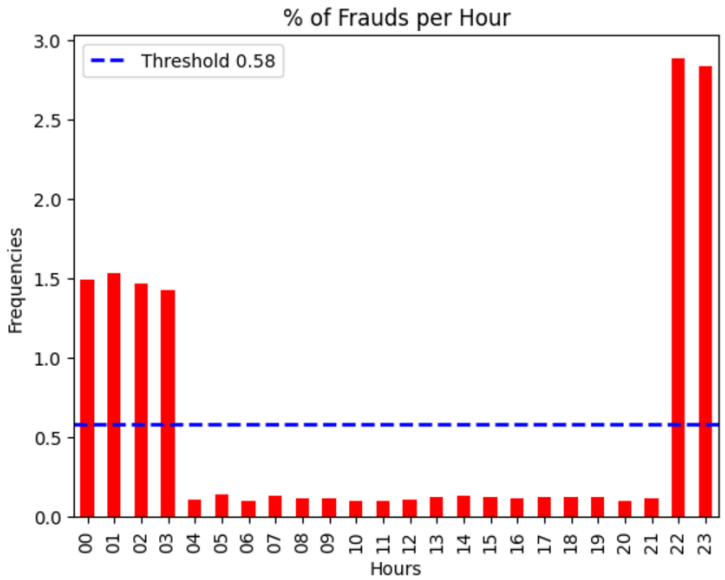


There is an initial period (from midnight to 11 a.m.) during which consumers make approximately the same number of transactions per hour (roughly 43 thousand). This is followed by a second period (from noon to 11 p.m.) where the number of transactions increases, remaining consistent at around 75 thousand per hour.

Figure 1.10: Number of Transactions per Hour

We can check if illegitimate transactions, reflect the proportions of the Figure 1.7

```
((df2[df2['is_fraud'] == 1].groupby('hour').size()*100)/df2.groupby("hour").size()).plot(kind = 'bar
    ', color = 'r')
plt.axhline(y=0.58,color="b",linestyle="--",linewidth=2, label="Threshold 0.58")
plt.title('% of Frauds per Hour')
plt.xlabel('Hours')
plt.ylabel('Frequencies')
plt.legend()
plt.show()
```



Nighttime transactions are more likely to be fraudulent.

Figure 1.11: Number of Frauds per Hour

1.3.5 Transaction Day, Year-Month and Time

The variable <trans_date_trans_time> is encoded as a datetime64, which could pose a problem since most models cannot directly process this type of data. To address this issue, we use the newly created variables <d> (day of the week), <ym> (year and month), and <hour>. These variables will replace <trans_date_trans_time>in the dataset <df1>.

```
df1=df1.drop("trans_date_trans_time",axis=1)
df1=pd.merge(df1,df2[["d","ym","hour"]],left_index=True, right_index=True)
```

1.3.6 Merchant Category

Overall merchant jobs are divided into 14 categories.

```
round((df2["category"].value_counts() * 100) / len(df2),2)
```

category	count
gas_transport	10.15
grocery_pos	9.54
home	9.49
shopping_pos	9.00
kids_pets	8.72
shopping_net	7.52
entertainment	7.25
food_dining	7.05
personal_care	7.00
health_fitness	6.62
misc_pos	6.14
misc_net	4.88
grocery_net	3.51
travel	3.12

To assess whether <category> is explanatory for our target variable, the percentages of frauds within each category should be calculated. If these percentages align with those presented in Figure 1.4, then <category> can be considered explanatory for the target variable. Instead to find out which category is more prone to have a transaction classified as fraud the percentages of frauds per category in respect to the tot amount of frauds should be computed.

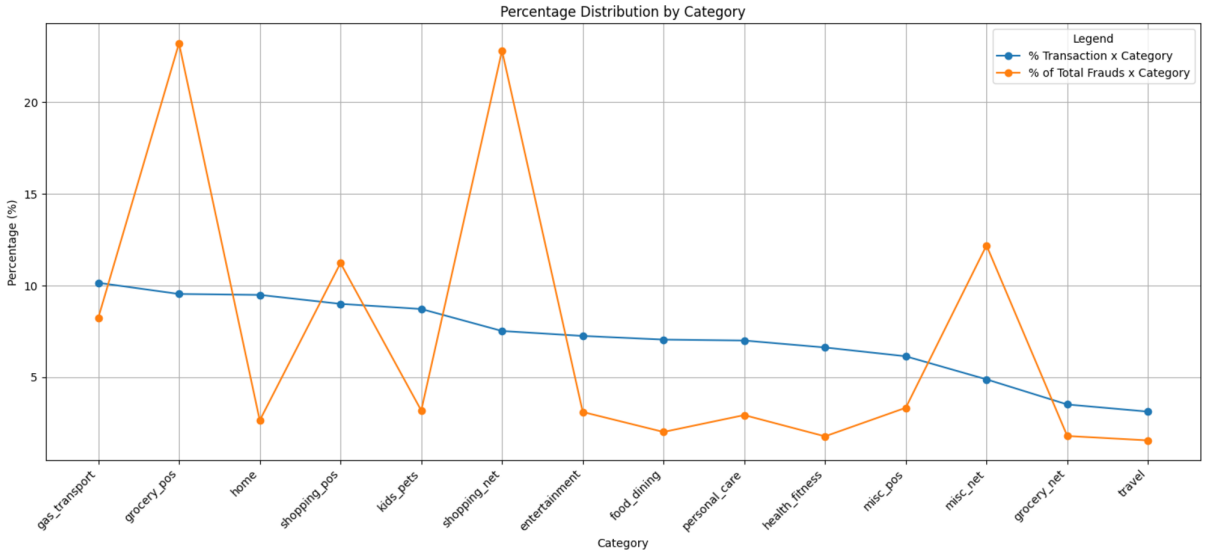
Figure 1.12: Percentage of Transactions per Category

```
df_percentages = pd.concat([
    (df2["category"].value_counts() * 100) / len(df2),
    (df2[df2['is_fraud'] == 1].groupby('category').size() * 100) / df2.groupby("category").size()]
, axis=1, keys=['% Transaction x Category', '% of Total Frauds x Category']).round(2)
print(df_percentages)
```

category	% Transaction x Category	% of Total Frauds x Category
gas_transport	10.15	0.47
grocery_pos	9.54	1.41
home	9.49	0.16
shopping_pos	9.00	0.72
kids_pets	8.72	0.21
shopping_net	7.52	1.76
entertainment	7.25	0.25
food_dining	7.05	0.17
personal_care	7.00	0.24
health_fitness	6.62	0.15
misc_pos	6.14	0.31
misc_net	4.88	1.45
grocery_net	3.51	0.29
travel	3.12	0.29

To better understand the difference between how the total transactions and the total frauds are distributed into categories we can plot the data.

```
df2_percentages = pd.concat([
    (df2["category"].value_counts() * 100) / len(df),
    (df2[df2['is_fraud'] == 1].groupby('category').size() * 100) / len(df2[df2['is_fraud'] == 1])
], axis=1, keys=['% Transaction x Category', '% of Total Frauds x Category']).round(2)

ax = df2_percentages.plot(kind='line', marker='o', figsize=(15, 7))
plt.title('Percentage Distribution by Category')
plt.xlabel('Category')
plt.ylabel('Percentage (%)')
ax.set_xticks(range(len(df2_percentages)))
ax.set_xticklabels(df2_percentages.index, rotation=45, ha="right")
plt.grid(True)
plt.legend(title='Legend')
plt.tight_layout()
plt.show()
```



If the percentage of frauds within a category is higher than its percentage of total transactions, transactions in that category, on average, have a higher likelihood of being fraudulent. Conversely, if the percentage of frauds in a category is lower than its percentage of total transactions, that category can be considered relatively safer.

As stated before, to see if <category> is explanatory of the variable we should calculate it's percentage of frauds and compare it to % of is_fraud = 1.

```
tab=round((df1[df1['is_fraud'] == 1].groupby('category').size() * 100) / df1.groupby('category').
    size(),2).sort_values(ascending=False)
ax=(round((df2[df2['is_fraud'] == 1].groupby('category').size() * 100) / df2.groupby('category').
    size(),2).sort_values(ascending=False)).plot(kind='bar',color="r")
plt.title('Percentages of Frauds per Category')
plt.xlabel('Category')
plt.ylabel('Percentages')
plt.axhline(y=0.58,linestyle="--",color="b",linewidth=2,label="Treshold 0.58")
plt.legend()
plt.show()
plt.show()
print(tab)
```

category	
shopping_net	1.76
misc_net	1.45
grocery_pos	1.41
shopping_pos	0.72
gas_transport	0.47
misc_pos	0.31
grocery_net	0.29
travel	0.29
entertainment	0.25
personal_care	0.24
kids_pets	0.21
food_dining	0.17
home	0.16
health_fitness	0.15

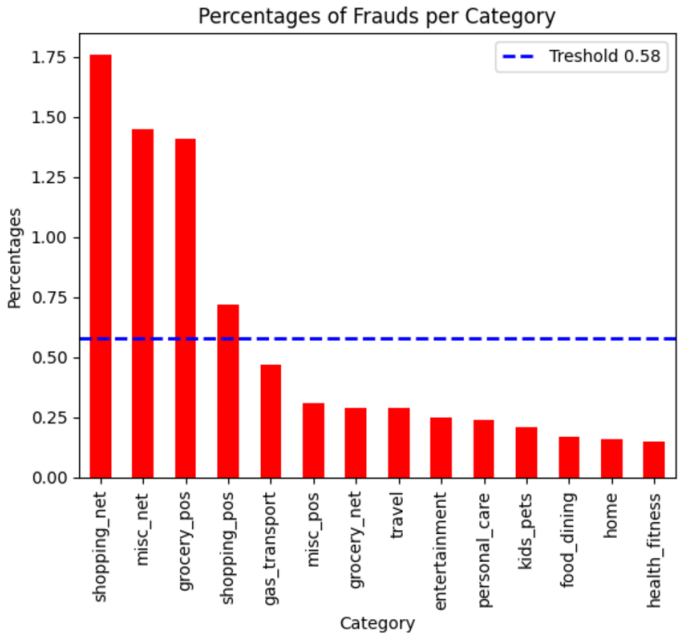


Figure 1.13: % of Frauds x Category

```
import seaborn as sns
c = 0.58
a=round((df2[df2['is_fraud'] == 1].groupby('category').size() * 100) / df2.groupby('category').size
    () ,2).sort_values(ascending=False)
b=a-c
b_df = b.reset_index()
b_df.columns = ['category', 'difference']
ax=sns.barplot(y='category',x='difference',data=b_df.sort_values('difference',ascending=False))
ax.set_xlabel('Percentage Difference')
ax.set_ylabel('Transaction Category')
plt.title('\% Fraudelent Transactions per Category - % Fraudulent Transactions in the Whole Dataset'
    )
```

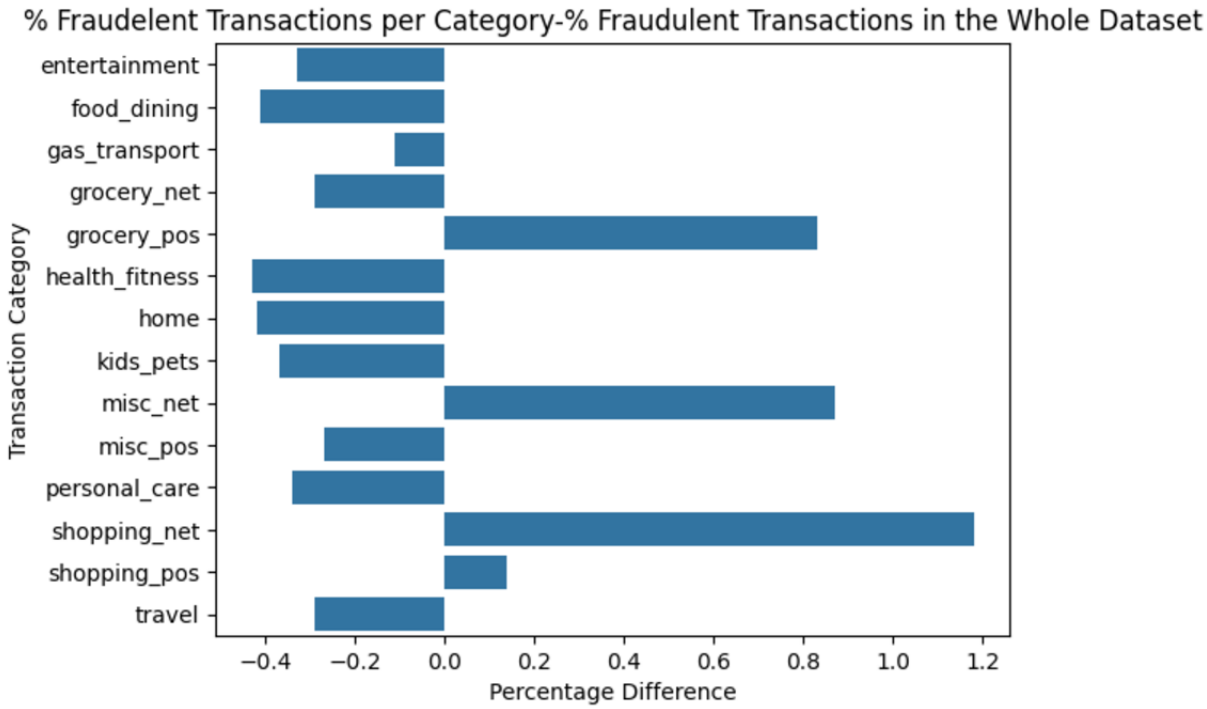


Figure 1.14: Difference between % of Frauds x Category and % of Total Frauds

1.3.7 Gender

As stated earlier, we need to analyze how the target variable is distributed across the categories of a given variable.

```
df2_percentages = pd.concat([
    (df2['gender'].value_counts()*100)/len(df2),
    (df2[df2['is_fraud'] == 1].groupby('gender').size() * 100) / len(df2[df2['is_fraud'] == 1])
], axis=1, keys=['% Gender Distribution', '% of Frauds for each Gender']).round(2)

print(df2_percentages)
```

gender	% Gender Distribution	% of Frauds for each Gender
F	54.74	49.76
M	45.26	50.24

Figure 1.15: % of transaction and frauds for each gender

Compared to the probability of <is_fraud = 1> the gender variable demonstrates explanatory power for the target.

1.3.8 State

The state categorical variable, groups the transactions occurred in each U.S.A. state, including the District of Columbia.

```
ax=((df2[df2['is_fraud'] == 1].groupby('state').size() * 100) / df2.groupby('state').size()).plot(
    kind="bar",figsize=(15,7))
plt.title("Percentage of Fraudulent Transactions per State")
plt.xlabel("State")
plt.ylabel("Percentage")
plt.xticks(rotation=45, fontsize=7)
```

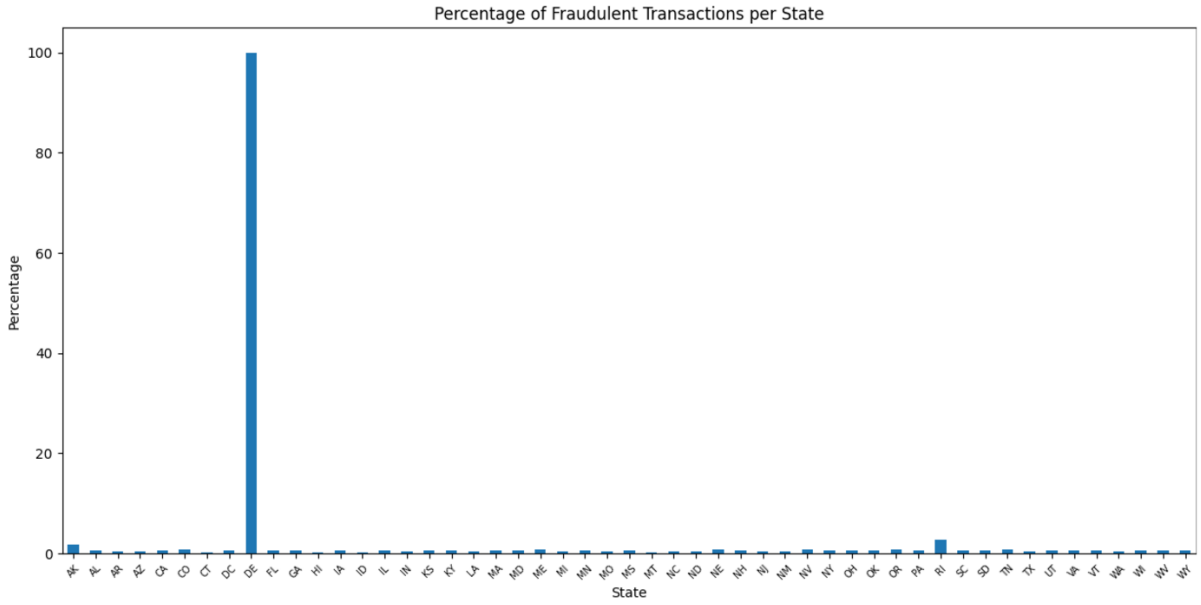


Figure 1.16: % of Frauds x State

DE (Delaware) distorts the plot. It has few transactions (9), made by the same person, that are all fraudulent.

```
df[df["state"]=="DE"].head(100)
```

	trans_date_trans_time	cc_num	merchant	category	amt	first	last	gender	street	city	...	lat	long	city_pop	job	dob	trans_num	unix_time	merch_lat	merch_long	is_fraud
233631	2019-04-28 00:41:37	601182631703477	fraud_Schamberger-O'Keefe	grocery_pos	294.91	Christine	Johnson	F	9612 Robert Light Apt. 340	Georgetown	...	38.679	-75.3932	18799	Designer, multimedia	2000-03-16	8f85e701f609911b99bea627eb71aa49	1335573697	38.166715	-75.023367	1
233744	2019-04-28 01:49:28	601182631703477	fraud_Kuphal-Predovic	misc_net	868.29	Christine	Johnson	F	9612 Robert Light Apt. 340	Georgetown	...	38.679	-75.3932	18799	Designer, multimedia	2000-03-16	8c941f2849ccac44d32ef08ea784fc67	1335577768	38.910900	-74.856625	1
233826	2019-04-28 02:33:09	601182631703477	fraud_Goodwin-Nitzsche	grocery_pos	280.26	Christine	Johnson	F	9612 Robert Light Apt. 340	Georgetown	...	38.679	-75.3932	18799	Designer, multimedia	2000-03-16	0fc1109cb9a10961595e79736ccc1293	1335580389	38.523208	-74.460889	1
234625	2019-04-28 10:24:34	601182631703477	fraud_Huel, Hammes and Witting	grocery_pos	296.25	Christine	Johnson	F	9612 Robert Light Apt. 340	Georgetown	...	38.679	-75.3932	18799	Designer, multimedia	2000-03-16	b4248cfc8b44fec61937f67918d49b334	1335608674	38.593896	-76.031493	1
236522	2019-04-28 23:03:24	601182631703477	fraud_Waters-Cruickshank	health_fitness	20.58	Christine	Johnson	F	9612 Robert Light Apt. 340	Georgetown	...	38.679	-75.3932	18799	Designer, multimedia	2000-03-16	567ed9b2d7d10ad519cdd697cd338f70	1335654204	39.003922	-75.852152	1
236643	2019-04-28 23:41:43	601182631703477	fraud_Cormier LLC	shopping_net	1012.77	Christine	Johnson	F	9612 Robert Light Apt. 340	Georgetown	...	38.679	-75.3932	18799	Designer, multimedia	2000-03-16	00d158609d7ac753da792e9d0eb88a0b	1335656503	39.596186	-75.559281	1
237095	2019-04-29 03:29:41	601182631703477	fraud_Ruecker Group	misc_net	885.96	Christine	Johnson	F	9612 Robert Light Apt. 340	Georgetown	...	38.679	-75.3932	18799	Designer, multimedia	2000-03-16	e0613e4142b2d05867299832ac453e01	1335670181	39.666374	-76.358642	1
237740	2019-04-29 10:13:47	601182631703477	fraud_Smitham-Schiller	grocery_net	10.93	Christine	Johnson	F	9612 Robert Light Apt. 340	Georgetown	...	38.679	-75.3932	18799	Designer, multimedia	2000-03-16	788a8fea8199a4e8a6a47ffab824e571	1335694427	37.691598	-76.058416	1
239854	2019-04-29 23:51:22	601182631703477	fraud_Baumbach, Feeney and Morar	shopping_net	960.49	Christine	Johnson	F	9612 Robert Light Apt. 340	Georgetown	...	38.679	-75.3932	18799	Designer, multimedia	2000-03-16	4193b853223a1c9432f9d3c380e67be9	1335743482	39.073319	-75.545153	1

To be able to correctly analyze the percentages of the other state variable, we drop "DE".

```
df2bis=df2[df2["state"]!="DE"]
ax=((df2bis[df2bis['is_fraud'] == 1].groupby('state').size() * 100) / df2bis.groupby('state').size()
    ).plot(kind="bar",figsize=(15,7))
plt.axhline(y=0.58,color="b",linestyle="--",linewidth=2,label="Treshold 0.58")
plt.title("Percentage of Fraudulent Transactions per State (Without DE)")
plt.xlabel("State")
plt.ylabel("Percentage")
plt.xticks(rotation=45, fontsize=7)
```

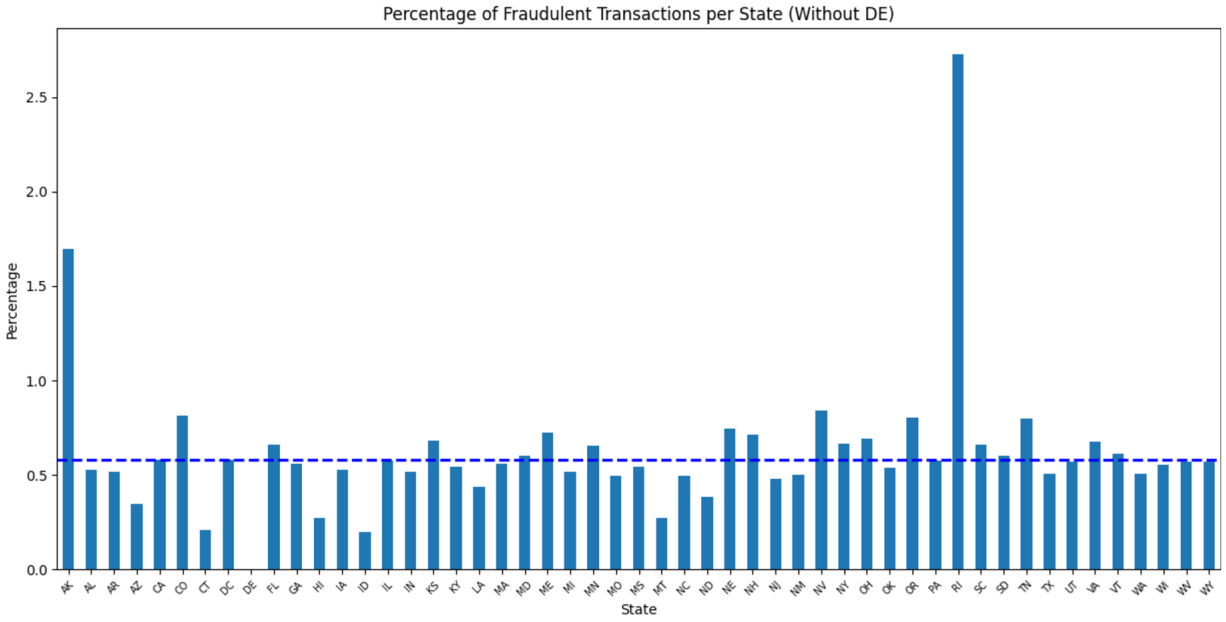


Figure 1.17: % of Frauds x State without DE

Very few states have an internal probability of fraud that is close to the threshold.

1.3.9 Merchant, City, Job

These three categorical variables have a large number of unique values, making it impractical to study them as we did with the previous categorical variables.

```
df1[["job","merchant","city"]].describe()
```

	job	merchant	city
count	1296675	1296675	1296675
unique	494	693	894
top	Film/video editor	fraud_Kilback LLC	Birmingham
freq	9779	4403	5617

Figure 1.18: Characteristics of job, merchant, city

Theoretically, <merchant> and <city> are subcategories of <category> and <state>, respectively. Therefore, we could consider dropping these variables since their characteristics are already generally represented by the other two variables.

Regarding <job> variable, there are no other variables in the dataset that could be considered its subcategories. To study the relationship between <job> and the target variable, Multiple Correspondence Analysis (MCA) could be applied. MCA is an exploratory statistical technique used to analyze and visualize relationships between categorical variables.

However, when working with unbalanced data, there are important considerations to ensure that the analysis is meaningful and that the results are not overly influenced by dominant categories. Techniques such as weighting observations or subsampling can help address the imbalance.

Given our limited knowledge of banking fraud and the potential relationships between a person's job and fraudulent transactions, there is a risk of incorrectly applying weights or subsampling techniques. To avoid this, it may be more prudent to initially evaluate the model's performance without the <job> variable. If performance is suboptimal, we could later experiment with including the variable to assess its impact.

1.4 Numerical Variables

We use the correlation matrix to study the relation between numerical variables and target.

1.4.1 Correlation Matrix

A data-frame containing the target (coded as an integer) and all the numerical variables is computed.

```
dfnum=df1[['amt',"city_pop","age","distance","is_fraud"]]
```

The correlation matrix is determined:

```
p.random.RandomState(0)
df = pd.DataFrame(rs.rand(10, 10))
corr = dfnum.corr()
corr.style.background_gradient(cmap='coolwarm')
```

	amt	city_pop	age	distance	is_fraud
amt	1.000000	0.005818	-0.009753	-0.001085	0.219404
city_pop	0.005818	1.000000	-0.092451	0.010901	0.002136
age	-0.009753	-0.092451	1.000000	-0.004592	0.012244
distance	-0.001085	0.010901	-0.004592	1.000000	0.000403
is_fraud	0.219404	0.002136	0.012244	0.000403	1.000000

The amount of the transaction <amt> is the numerical variable more correlated with the target.

Figure 1.19: Correlation Matrix

In addition to creating a correlation matrix for numerical variables, we can divide them into bins and analyze them as if they were categorical variables.

1.4.2 Age

```
df2["age"].describe()
```

	age
count	1.296675e+06
mean	4.552822e+01
std	1.740895e+01
min	1.300000e+01
25%	3.200000e+01
50%	4.400000e+01
75%	5.700000e+01
max	9.500000e+01

The <age> variable ranges from 13 to 95 years old, which is acceptable and indicates that there are no apparent errors.

Figure 1.20: Description of the variable "Age"

The <age> variable is divided into 21 bins, each with a range of 4 years. The goal is to create bins that reflect similar groups, such as adolescents, young adults, and so on.

```
bins=[i for i in range (10,96,4)]
df2["age2"]=pd.cut(df2["age"],bins=bins)
ax=((df2[df2["is_fraud"]==1].groupby("age2").size()*100)/df2.groupby("age2").size()).plot(kind="bar"
    )
plt.axhline(y=0.58, color="blue", linestyle="--", linewidth=2, label="Threshold 0.58")
plt.title("% Fraudulent Transactions per Age Group")
plt.xlabel("Age Group")
plt.ylabel("Percentage")
```

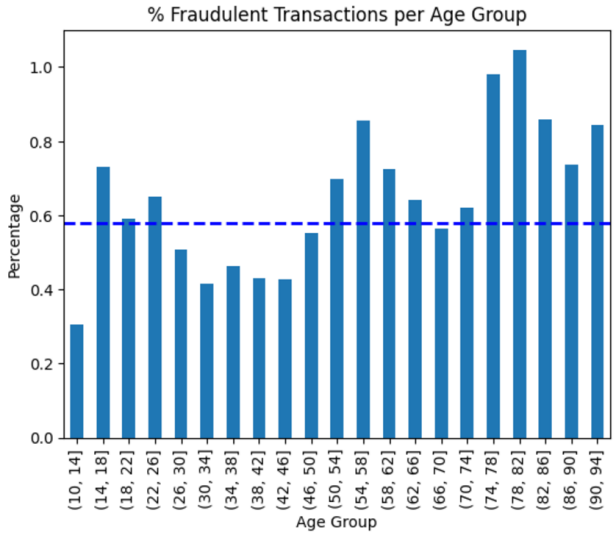


Figure 1.21: % Frauds x Age Groups

Teens, middle-aged individuals, and the elderly are more likely to fall victim to fraud, perhaps due to their inexperience, particularly among teens and older adults.

1.4.3 Customer-Merchant Distance

The <distance> between the customer's residence and the merchant's location is a numerical variable, but it can be divided into bins (transforming it into a categorical variable) to analyze the fraud percentages within each bin.

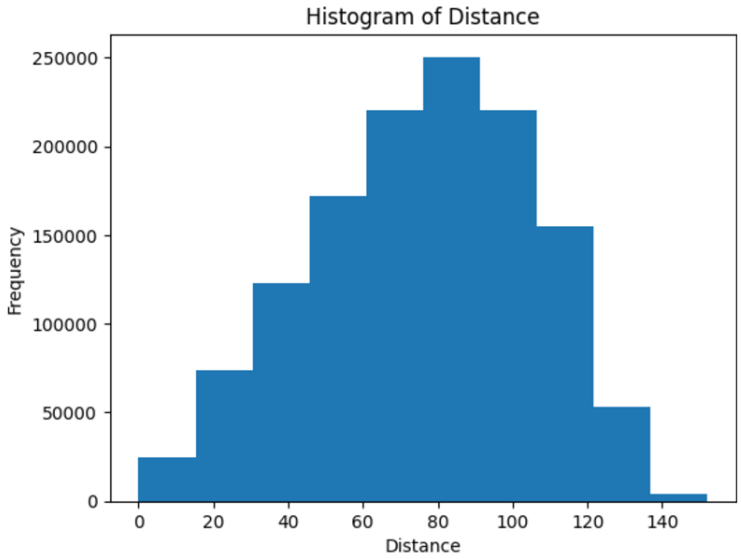
```
df2["distance"].describe()
```

	distance
count	1.296675e+06
mean	7.611476e+01
std	2.911698e+01
min	2.200000e-02
25%	5.533500e+01
50%	7.823200e+01
75%	9.850300e+01
max	1.521170e+02

Figure 1.22: Description of the varible "Distance"

The <distance> space from a minimum of 22m to a maximum of 152.117 km.

```
ax=df2["distance"].plot(kind="hist")
plt.title("Histogram of Distance")
plt.xlabel("Distance")
plt.ylabel("Frequency")
```



<Distance> follows an approximately normal distribution with a slight right skew.

Figure 1.23: Distribution of Distances

Next, we divide the variable into 16 equal bins and analyze how the fraud probability is distributed across them.

```
bins=[i for i in range (0,161,10)]
df2["distance"]=pd.cut(df2["distance"],bins=bins)
df2["distance"] = df2["distance"].astype('category')
ax=((df2[df2["is_fraud"]==1].groupby("distance").size()*100)/df2.groupby("distance").size()).plot(
    kind="bar",color="r")
plt.axhline(y=0.58, color="blue", linestyle="--", linewidth=2, label="Threshold 0.58")
plt.title("Percentage of Fraudulent Transactions per Distance")
plt.xlabel("Distance Bins")
plt.ylabel("Percentage")
print(round((df2[df2["is_fraud"]==1].groupby("distance").size()*100)/df2.groupby("distance").size()
    ,2))
plt.show()
```

distance	
(0, 10]	0.50
(10, 20]	0.53
(20, 30]	0.55
(30, 40]	0.57
(40, 50]	0.57
(50, 60]	0.60
(60, 70]	0.61
(70, 80]	0.59
(80, 90]	0.57
(90, 100]	0.58
(100, 110]	0.58
(110, 120]	0.58
(120, 130]	0.57
(130, 140]	0.53
(140, 150]	0.62
(150, 160]	0.00

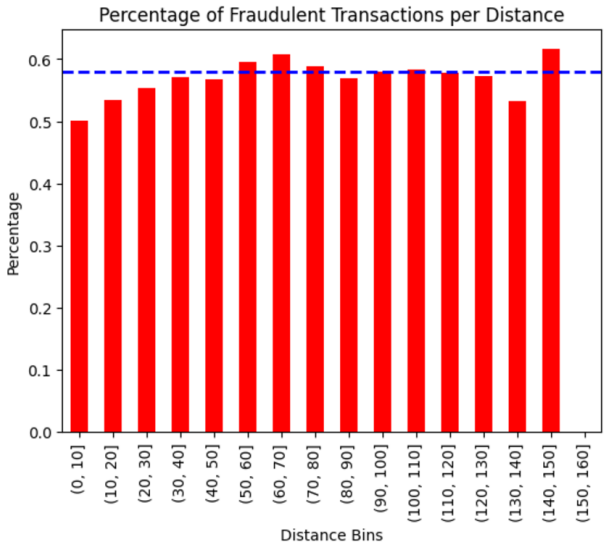


Figure 1.24: Fraud Percentage in each bin

1.4.4 Amount of the Transaction

```
df2["amt"].describe()
```

	amt
count	1.296675e+06
mean	7.035104e+01
std	1.603160e+02
min	1.000000e+00
25%	9.650000e+00
50%	4.752000e+01
75%	8.314000e+01
max	2.894890e+04

Figure 1.25: Description of the variable "amt"

The dataset range over transaction with a minimum of 1$ to a max of 29,000$.

The aim is to create bins that have approximately the same amount of transactions inside.

```
df2["amt_bins"] = pd.qcut(df2["amt"], q=20, duplicates='drop')
df_percentages = pd.concat([
    df2["amt_bins"].value_counts(sort=False),
    ((df2["amt_bins"].value_counts(sort=False) * 100) / len(df2)),
    (round(df2[df2["is_fraud"] == 1].groupby("amt_bins").size() * 100 /
           df2.groupby("amt_bins").size(), 2))
], axis=1, keys=['Transactions x Bins', "% Transactions x Bins", '% Frauds x Bin']).round(2)
print(df_percentages)
```

amt_bins	Transactions x Bins	% Transactions x Bins	% Frauds x Bin
(0.999, 2.44]	64868	5.00	0.01
(2.44, 4.11]	64927	5.01	0.02
(4.11, 5.9]	64983	5.01	0.02
(5.9, 7.75]	64901	5.01	0.22
(7.75, 9.65]	64646	4.99	0.44
(9.65, 15.74]	64751	4.99	0.66
(15.74, 23.74]	64816	5.00	0.98
(23.74, 32.13]	64814	5.00	0.05
(32.13, 40.14]	64819	5.00	0.00
(40.14, 47.52]	64887	5.00	0.01
(47.52, 54.15]	64802	5.00	0.12
(54.15, 60.94]	64803	5.00	0.00
(60.94, 68.11]	64847	5.00	0.00
(68.11, 75.03]	64822	5.00	0.00
(75.03, 83.14]	64877	5.00	0.00
(83.14, 94.68]	64788	5.00	0.00
(94.68, 110.84]	64849	5.00	0.05
(110.84, 136.67]	64817	5.00	0.16
(136.67, 196.31]	64836	5.00	0.03
(196.31, 28948.9]	64822	5.00	8.80

Figure 1.26: Properties of amount bins

If we compute more general bins it becomes more interesting.

```
bins=[i for i in range(0, 1000,100)]+ [float('inf')]
df2["amt_bins"]=pd.cut(df2["amt"],bins=bins,right=False)
df_percentages = pd.concat([
    df2["amt_bins"].value_counts(sort=False),
    ((df2["amt_bins"].value_counts(sort=False)*100)/len(df2)),
    (round(df2[df2["is_fraud"]==1].groupby("amt_bins").size()*100/df2.groupby("amt_bins").size(),2))
], axis=1, keys=['Transactions x Bins',"% Transactions x Bins" ,'% of Frauds x Bin']).round(2)
print(df_percentages)
```

	Transactions x Bins	% Transactions x Bins	% of Frauds x Bin
amt_bins			
[0.0, 100.0)	1061728	81.88	0.16
[100.0, 200.0)	173017	13.34	0.09
[200.0, 300.0)	31631	2.44	2.51
[300.0, 400.0)	8600	0.66	13.49
[400.0, 500.0)	6068	0.47	1.66
[500.0, 600.0)	4558	0.35	2.06
[600.0, 700.0)	1945	0.15	8.59
[700.0, 800.0)	1910	0.15	34.71
[800.0, 900.0)	1679	0.13	49.49
[900.0, inf)	5539	0.43	34.18

Figure 1.27: Proprieties of amount bins

A total of 81.88% of transactions fall within the $1–$100 range, with an internal fraud rate of just 0.16%. The data clearly shows that transactions involving higher amounts are substantially more likely to be fraudulent. Dividing the <amt> variable into two broad bins can provide a clearer view of this disparity.

```
bins=[i for i in range(0, 201,200)]+ [float('inf')]
df2["amt_bins"]=pd.cut(df2["amt"],bins=bins,right=False)
df_percentages = pd.concat([
    ((df2["amt_bins"].value_counts(sort=False)*100)/len(df2)),
    (round(df2[df2["is_fraud"]==1].groupby("amt_bins").size()*100/df2.groupby("amt_bins").size(),2))
    ,
    (round(df2[df2["is_fraud"]==1].groupby("amt_bins").size()*100/len(df2[df2['is_fraud'] == 1]),2))
], axis=1, keys=['% Transaction of Dataset', '% of Frauds x Bin',"% of Total Frauds"]).round(2)
print(df_percentages)
```

	% Transaction of Dataset	% of Frauds x Bin	% of Total Frauds
amt_bins			
[0.0, 200.0)	95.22	0.15	24.01
[200.0, inf)	4.78	9.21	75.99

Figure 1.28: Amount varible divided in 2 bins

1.4.5 City Population

```
df2["city_pop"].describe()
```

	city_pop
count	1.296675e+06
mean	8.882444e+04
std	3.019564e+05
min	2.300000e+01
25%	7.430000e+02
50%	2.456000e+03
75%	2.032800e+04
max	2.906700e+06

Figure 1.29: Description of the variable "city population"

The transactions in the dataset come from customers residing in both small towns (minimum of 23 people) and large cities (up to nearly 3 million people).

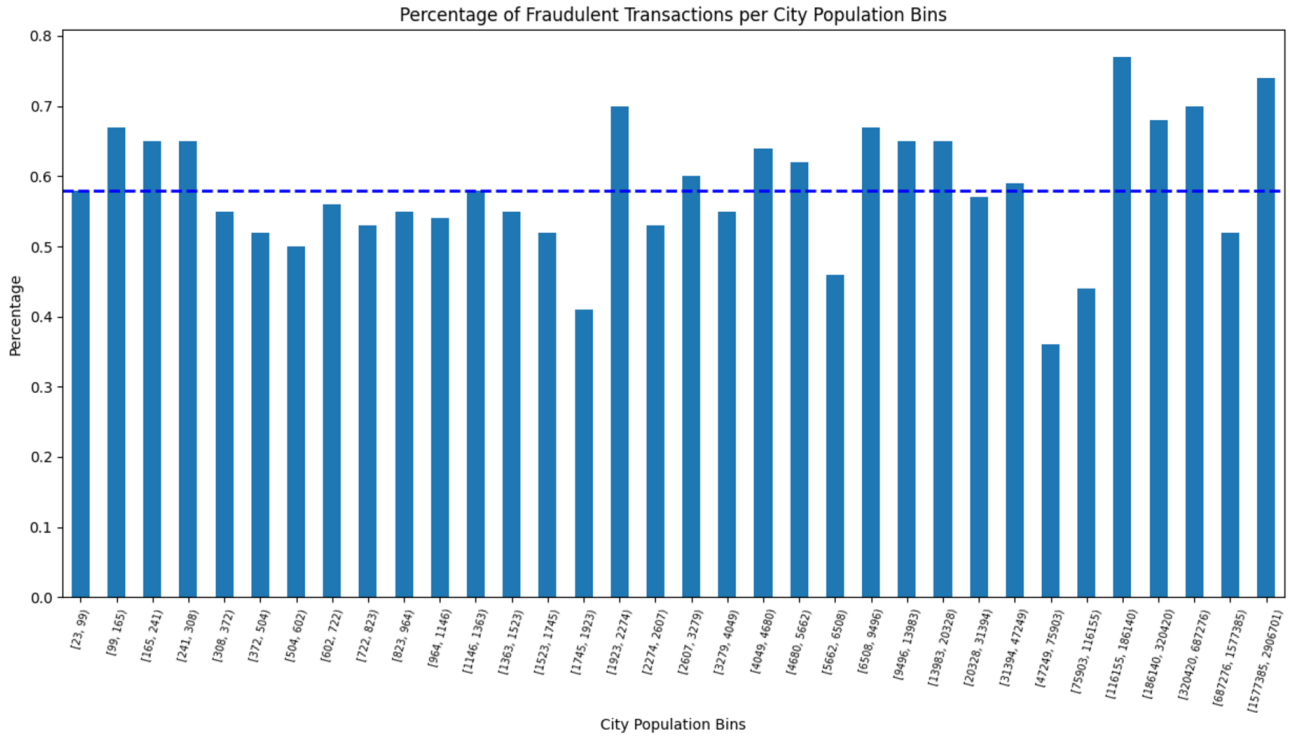
The city population can be divided into bins to analyze whether the fraud percentage is evenly distributed across them. To compare bins with similar frequencies, we divide the data into 34 bins, aiming to ensure each bin contains roughly the same number of observations.

```
percent_per_bin = 3 / 100
sorted_data = np.sort(df2['city_pop'])
cumulative_counts = np.cumsum(np.ones_like(sorted_data)) / len(sorted_data)

bin_edges = [sorted_data[0]]
current_threshold = percent_per_bin
while current_threshold < 1:
    edge = sorted_data[np.searchsorted(cumulative_counts, current_threshold)]
    bin_edges.append(edge)
    current_threshold += percent_per_bin
bin_edges.append(sorted_data[-1] + 1)
bin_edges = np.unique(bin_edges)

df2["city_bins"] = pd.cut(df2["city_pop"], bins=bin_edges, right=False)
transactions_per_bin = df2["city_bins"].value_counts(sort=False)
percent_transactions_per_bin = (transactions_per_bin * 100 / len(df2)).round(2)
fraud_percent_per_bin = (
    (df2[df2["is_fraud"] == 1].groupby("city_bins").size() * 100 / df2.groupby("city_bins").size())
    .fillna(0)
    .round(2)
)

df_percentages = pd.concat([
    transactions_per_bin,
    percent_transactions_per_bin,
    fraud_percent_per_bin
], axis=1, keys=['Transactions x Bin', '%Transactions x Bin', '% Frauds x Bin']).round(2)

print(df_percentages)
```

city_bins	Transactions x Bin	%Transactions x Bin	% Frauds x Bin
[23, 99)	36973	2.85	0.58
[99, 165)	39201	3.02	0.67
[165, 241)	37390	2.88	0.65
[241, 308)	41931	3.23	0.65
[308, 372)	37522	2.89	0.55
[372, 504)	38923	3.00	0.52
[504, 602)	39797	3.07	0.50
[602, 722)	38955	3.00	0.56
[722, 823)	37429	2.89	0.53
[823, 964)	40749	3.14	0.55
[964, 1146)	38823	2.99	0.54
[1146, 1363)	38854	3.00	0.58
[1363, 1523)	36226	2.79	0.55
[1523, 1745)	41295	3.18	0.52
[1745, 1923)	38339	2.96	0.41
[1923, 2274)	39924	3.08	0.70
[2274, 2607)	38767	2.99	0.53
[2607, 3279)	38087	2.94	0.60
[3279, 4049)	39046	3.01	0.55
[4049, 4680)	39519	3.05	0.64
[4680, 5662)	38671	2.98	0.62
[5662, 6508)	39336	3.03	0.46
[6508, 9496)	38938	3.00	0.67
[9496, 13983)	38515	2.97	0.65
[13983, 20328)	37441	2.89	0.65
[20328, 31394)	39221	3.02	0.57
[31394, 47249)	38965	3.00	0.59
[47249, 75903)	40347	3.11	0.36
[75903, 116155)	38880	3.00	0.44
[116155, 186140)	37186	2.87	0.77
[186140, 320420)	39952	3.08	0.68
[320420, 687276)	39126	3.02	0.70
[687276, 1577385)	37932	2.93	0.52
[1577385, 2906701)	14415	1.11	0.74

To facilitate a better comparison with the fraud probability, the plot is generated.

```
df_percentages['% Frauds x Bin'].plot(kind="bar",figsize=(15,7))
plt.axhline(y=0.58, color="blue", linestyle="--", linewidth=2, label="Threshold 0.58")
plt.title("Percentage of Fraudulent Transactions per City Population Bins")
plt.xlabel("City Population Bins")
plt.ylabel("Percentage")
plt.xticks(rotation=75, fontsize=7)
```



Many bins are pretty close to the treshold

Chapter 2

Training and Test

Since we have two different datasets, one for training and one for testing, we need to apply the same feature anonymization and feature transformation to the test data as we did to the training data to ensure the model processes it correctly.

2.0.1 Test Data Upload

```
dftest = pd.read_csv(path+'/fraudTest.csv')
dftest.dtypes
```

trans_date_trans_time	object
cc_num	int64
merchant	object
category	object
amt	float64
first	object
last	object
gender	object
street	object
city	object
state	object
zip	int64
lat	float64
long	float64
city_pop	int64
job	object
dob	object
trans_num	object
unix_time	int64
merch_lat	float64
merch_long	float64
is_fraud	int64

Figure 2.1: Test Data Types

Python reads both the test and train data equally when they are first loaded into the code.

Hence the same transformation made on <df> (Training Data) are made on <dftest> (Test Data).

```
for i in (1,2,3, 7,8,9,10,15,21):
  dftest[dftest.columns[i]] = dftest[dftest.columns[i]].astype('category')

for i in (5,6,17):
  dftest[dftest.columns[i]] = dftest[dftest.columns[i]].astype('string')

dftest['trans_date_trans_time'] = pd.to_datetime(dftest['trans_date_trans_time'],errors="coerce")
dftest['dob'] = pd.to_datetime(dftest['dob'])
```

Now that <dftest> is correctly read by Colab, all the feature transformations and eliminations can be performed

```
dftest = dftest.drop(["cc_num","first","last","trans_num"], axis = 1)

from haversine import haversine_vector, Unit
dftest["distance"]=0
for i in range (len(dftest)):
  dftest["distance"][i]=haversine_vector((dftest["lat"][i], dftest["long"][i]), (dftest["merch_lat"
      ][i],dftest["merch_long"][i]), unit=Unit.KILOMETERS)
dftest["distance"]=np.round(dftest["distance"],3)

dftest=dftest.drop(["lat","long","merch_lat","merch_long"],axis=1)

dftest=dftest.drop(["unix_time"],axis=1)

dftest['age'] = (dftest['trans_date_trans_time'] - dftest['dob']).dt.days // 365
dftest['age'] = dftest['age'].astype('int')

dftest=dftest.drop(["dob"],axis=1)

dftest=dftest.drop(["zip","street"],axis=1)

dftest['d'] = dftest['trans_date_trans_time'].dt.day_name()
dftest["d"] = dftest["d"].astype("category")

dftest['ym'] = dftest['trans_date_trans_time'].dt.strftime('%Y-%m')
dftest["ym"] = dftest["ym"].astype("category")

dftest['hour'] = dftest['trans_date_trans_time'].dt.strftime('%H')
dftest["hour"] = dftest["hour"].astype("category")

dftest=dftest.drop(["trans_date_ttrans_time"],axis=1)
```

The final step in preparing both train and test data is to separate the target variable from the feature variables.

```
X_test = dftest.drop('is_fraud', axis=1)
y_test = dftest['is_fraud']
X_train = df1.drop('is_fraud', axis=1)
y_train = df1['is_fraud']
```

2.0.2 Training and Test Differences

After a brief analysis of both <Training and Test> data, it has emerged that the transactions have happened in completely different periods.

```
print(X_train["ym"].value_counts())
print(X_test["ym"].value_counts())
```

ym	
2019-12	141060
2019-08	87359
2019-07	86596
2019-06	86064
2020-05	74343
2020-03	72850
2019-05	72532
2019-03	70939
2019-09	70652
2019-11	70421
2019-10	68758
2019-04	68078
2020-04	66892
2020-06	57747
2019-01	52525
2020-01	52202
2019-02	49866
2020-02	47791

Figure 2.2: Freq Count Train

ym	
2020-12	139538
2020-08	88759
2020-07	85848
2020-11	72635
2020-09	69533
2020-10	69348
2020-06	30058

Figure 2.3: Freq Count Test

Additionally, the <state> variable in the test dataset is missing a category, specifically Delaware.

```
set(X_train["state"].unique())-set(X_test["state"].unique())
```

```
{'DE'}
```

2.0.3 Imbalanced Data Handling

When the majority class has significantly more samples than the minority class, <under-sampling> is a reasonable strategy to balance the dataset and improve model performance. All the illegitimate transactions must be used to train the model; we only need to extract the same number of legitimate transactions from the <df>.

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from imblearn.under_sampling import RandomUnderSampler

rus = RandomUnderSampler(random_state=42)
X_train_resampled, y_train_resampled = rus.fit_resample(X_train, y_train)
print(df1["is_fraud"].value_counts())
print(y_train_resampled.value_counts())
```

```
is_fraud
0    1289169
1       7506
Name: count, dtype: int64
is_fraud
0    7506
1    7506
Name: count, dtype: int64
```

Figure 2.4: Verify under-sampling correct application.

Under-sampling is applied only to the training data. The success of the process can be verified by checking if the values of <y_train_resampled> are balanced and correspond to the total frequency of the minority class in the original imbalanced target variable.

Chapter 3

Model Evaluation

The general idea is to create both a classification tree (CatBoost) and a Logistic Regression model, and compare their performance in fraud detection. In the EDA chapter, all numerical variables were divided into bins and analyzed with respect to the target. This binning was done for exploration purposes only. However, when training the model, we believe it is better to use the original, untransformed variables as encoded in <df1>, which used to create the training and test datasets.

```
df1.dtypes
```

trans_date_trans_time	datetime64[ns]
merchant	category
category	category
amt	float64
gender	category
city	category
state	category
city_pop	int64
job	category
is_fraud	category
distance	float64
age	int64

Figure 3.1: df1 Data Types

More specifically:

- Classification trees automatically divide numerical variables into optimal bins based on the data. Manually creating bins can be redundant or even worsen performance. Additionally, with large datasets, binning is unnecessary because the tree will have enough data to find the optimal splits on its own.
- Logistic Regression does not require discretizing numerical variables if there is a linear (or nearly linear) relationship with the target. However, if many bins are used or there is no clear rule for binning, the model may become overly complex and prone to overfitting.

The idea is to compute both a classification tree and logistic regression models twice. The first time, we will perform the analysis using a selection of variables based on the exploratory data analysis (EDA). The second time, we will apply Lasso regularization on the logistic regression model to select variables, and then use only the variables with non-zero coefficients for the classification tree. Afterward, we can compare the models' performance by evaluating ROC curves and cumulative gain charts.

3.0.1 Models with EDA and LASSO Selected Variables

The first thing to do is select the variables based on the Eda Analysis that are going to be used for both Logistic Regressione and CatBoost Classifier.

Variable Selection

Based on the results of the Exploratory Data Analysis (Chapter 1), the following variables are selected from <X_train> and stored in a new variable called EDAX_train:

```
EDAX_train_res=X_train_resampled[["d","ym","hour","category","state","gender","age","amt"]]
EDAX_train_res.dtypes
```

d	category
ym	category
hour	category
category	category
state	category
gender	category
age	int64
amt	float64

Figure 3.2: EDAX_train Variables

We have to choose the same variables in the test data <X_test>

```
EDAX_test=X_test[["d","ym","hour","category","state","gender","age","amt"]]
```

Logistic Regression

Logistic regression is a supervised machine learning algorithm widely used for binary classification tasks. In it's basic form this model doesn't natively handle categorical variable. It requires numerical input, so categorical variables must be converted into a numerical format before they can be used in the model.

One-Hot Encoding

One-Hot Encoding transforms each category of a categorical variable into a dummy variable. The problem with this encoding technique is that if we don't have the same categories in both the test and training sets, we end up with a different number of variables, which causes issues when running models like logistic regression.

In Fig 2.2 and Fig 2.3, it can be observed that the test data only contains a few months of 2020, compared to the training set, which includes transactions from 01/2019 to 05/2020. Additionally, the test data does not contain any transactions from the state of Delaware.

To solve these issues after performing the encoding, we can:

1. Remove the dummy variable corresponding to the Delaware state <(state_DE)>from the test data, if it is not present.

2. Remove the year and add the missing months as new dummy variables in the test data, setting their values to zero. This ensures that the test set has the same number of features as the training set.

```
EDAX_test1=EDAX_test.copy()
EDAX_train_res1=EDAX_train_res.copy()
EDAX_test1["ym"]= [s[5:] for s in EDAX_test1["ym"]]
EDAX_train_res1["ym"]= [s[5:] for s in EDAX_train_res1["ym"]]
EDAX_train_res1= pd.get_dummies(EDAX_train_res1, columns=['d","ym","hour","category","state","gender
    "], drop_first=False)
EDAX_test1= pd.get_dummies(EDAX_test1, columns=['d","ym","hour","category","state","gender"],
    drop_first=False)
```

After encoding, the necessary corrections are applied, ensuring that <EDAX_train_res1 and EDAX_test1> have the same variables.

```
EDAX_train_res1=EDAX_train_res1.drop(["state_DE"],axis=1)
EDAX_test1["ym_01"]=0
EDAX_test1["ym_02"]=0
EDAX_test1["ym_03"]=0
EDAX_test1["ym_04"]=0
EDAX_test1["ym_05"]=0
```

Standardizing

Another issue with Logistic Regression is that it cannot standardize the data on its own; therefore, it is our responsibility to perform this step.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
EDAX_train_res1scaled= scaler.fit_transform(EDAX_train_res1)
EDAX_test1scaled = scaler.transform(EDAX_test1)
```

Model Training

Before training the model, it is important not only to ensure that the training and test datasets have the same variables but also that the variables are in the same order.

```
EDAX_test1scaled=EDAX_test1[EDAX_train_res1scaled.columns]
```

Now we can correctly train the model.

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression(random_state=1, max_iter=1000, solver='lbfgs')
logreg.fit(EDAX_train_res1scaled, y_train_resampled)
y_pred=logreg.predict(EDAX_test1scaled)
```

To evaluate its performance, we examine the confusion matrix.

```
from sklearn.metrics import confusion_matrix, classification_report, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=logreg.classes_)
disp.plot(cmap='Blues')

plt.title('Confusion Matrix')
plt.show()
```

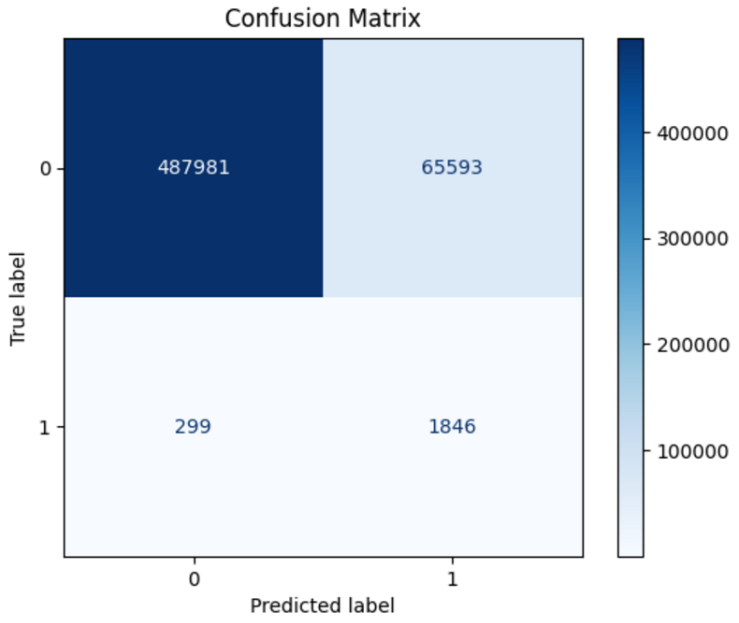


Figure 3.3: EDAX Variable Logistic Regression

Ridge and LASSO Logistic Regression

Untill now we have used our Exploratory Data Analysis to understand which variables are more correlated with our target variable, and used them to define our model. We can automatize this process by a data driven procedure, introducing a penalty in the loss function. If we introduce a penalty which is the sum of squares of the coefficients, we talk about Ridge regression, if the penalty is the sum of the magnitude of coefficients it's LASSO regression. This procedure should select the most important variables and in the case of the LASSO, some coefficients should go to 0.

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
LASSOX_train = X_train_resampled.copy()
LASSOX_test = X_test.copy()
categorical_columns = ["merchant","category","gender","city","state","job","d","ym","hour"]


encoders = {}
for col in categorical_columns:
    encoders[col] = LabelEncoder()

    all_values = pd.concat([LASSOX_train[col], LASSOX_test[col]], ignore_index=True).unique()
    encoders[col].fit(all_values)

    LASSOX_train[col] = encoders[col].transform(LASSOX_train[col])
    LASSOX_test[col] = encoders[col].transform(LASSOX_test[col])

scaler = StandardScaler()
LASSOX_train_scaled= scaler.fit_transform(LASSOX_train)
LASSOX_test_scaled = scaler.transform(LASSOX_test)

ridge_log_reg = LogisticRegression(penalty='l2', C=1.0, solver='lbfgs', random_state=42)
ridge_log_reg.fit(LASSOX_train_scaled, y_train_resampled)
y_pred_ridge = ridge_log_reg.predict(LASSOX_test_scaled)

lasso_log_reg = LogisticRegression(penalty='l1', C=1.0, solver='liblinear', random_state=42)
lasso_log_reg.fit(LASSOX_train_scaled, y_train_resampled)
y_pred_lasso = lasso_log_reg.predict(LASSOX_test_scaled)

cm_ridge = confusion_matrix(y_test, y_pred_ridge)
cm_lasso = confusion_matrix(y_test, y_pred_lasso)

fig, axes = plt.subplots(1, 2, figsize=(12, 6))

sns.heatmap(cm_ridge, annot=True, fmt="d", cmap="Blues", xticklabels=[0, 1], yticklabels=[0, 1], ax=
    axes[0])
axes[0].set_title('Ridge␣Logistic␣Regression')
axes[0].set_xlabel('Predicted')
axes[0].set_ylabel('Actual')

sns.heatmap(cm_lasso, annot=True, fmt="d", cmap="Blues", xticklabels=[0, 1], yticklabels=[0, 1], ax=
    axes[1])
axes[1].set_title('Lasso␣Logistic␣Regression')
axes[1].set_xlabel('Predicted')
axes[1].set_ylabel('Actual')

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

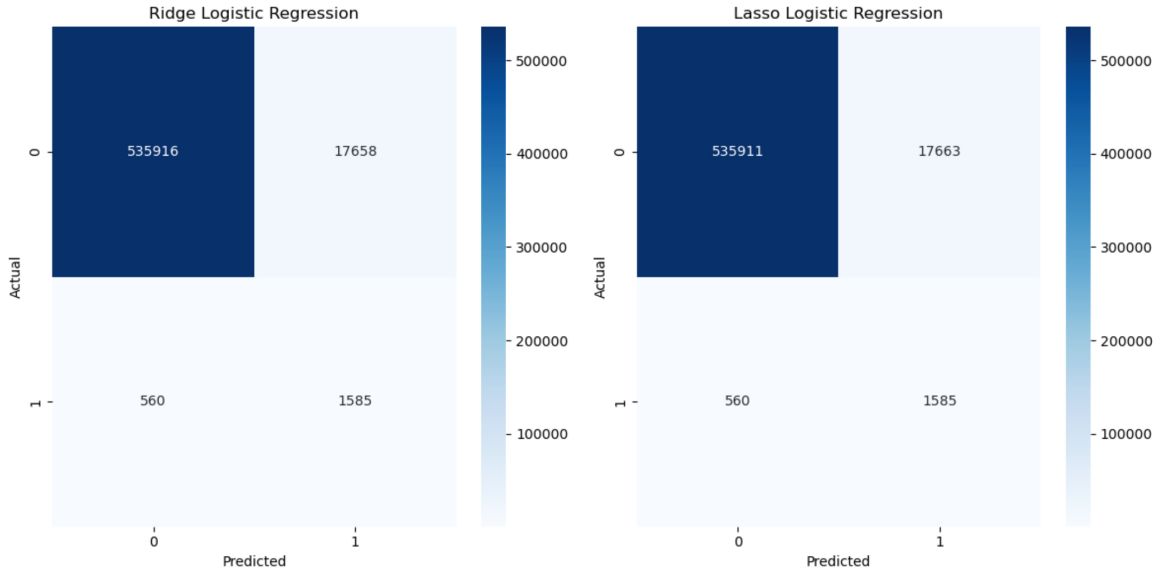


Figure 3.4: Ridge and Lasso Confusion Matrix

we can plot also the Precision-Recall curve

```
    y_scores_ridge = ridge_log_reg.decision_function(LASSOX_test_scaled)
y_scores_lasso = lasso_log_reg.decision_function(LASSOX_test_scaled)

precision_ridge, recall_ridge, _ = precision_recall_curve(y_test, y_scores_ridge)
average_precision_ridge = average_precision_score(y_test, y_scores_ridge)

precision_lasso, recall_lasso, _ = precision_recall_curve(y_test, y_scores_lasso)
average_precision_lasso = average_precision_score(y_test, y_scores_lasso)

plt.figure(figsize=(10, 6))
plt.plot(recall_ridge, precision_ridge, label=f"Ridge (AP = {average_precision_ridge:.2f})",
    linewidth=2)
plt.plot(recall_lasso, precision_lasso, label=f"Lasso (AP = {average_precision_lasso:.2f})",
    linewidth=2)

plt.title("Curva Precision-Recall: Ridge vs Lasso", fontsize=14)
plt.xlabel("Recall", fontsize=12)
plt.ylabel("Precision", fontsize=12)
plt.legend(loc="lower left", fontsize=12)
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```

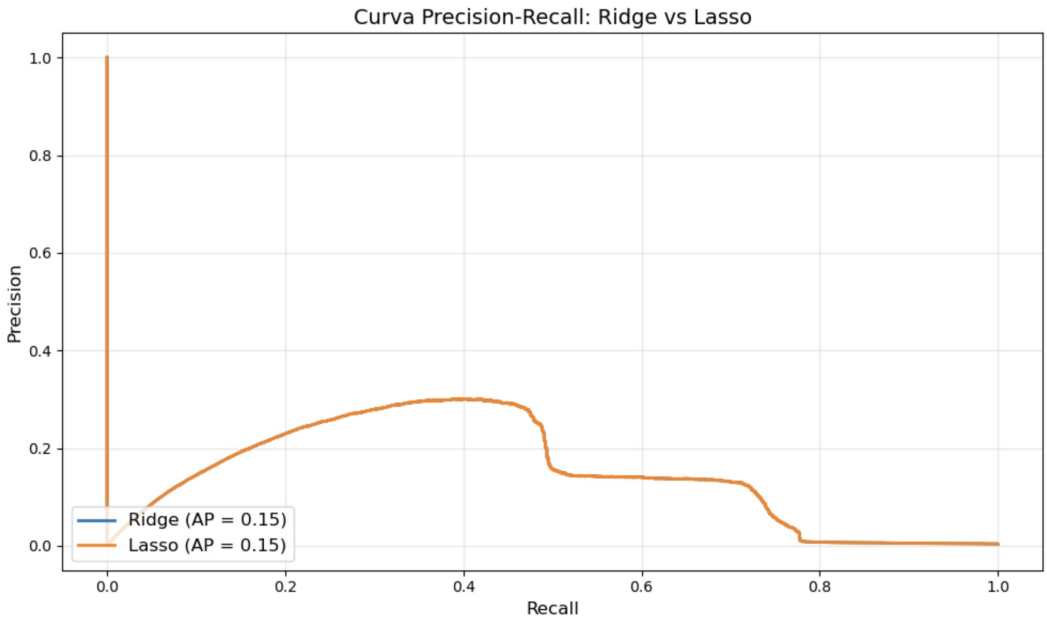


Figure 3.5: Ridge and Lasso PR Curve

As we can see, the two curves are overlapping. In fact, if we print the coefficients of the two regression:

```
    ridge_coefficients = pd.DataFrame({
    'Feature': LASSOX_train.columns,
    'Ridge Coefficient': ridge_log_reg.coef_[0]
})

lasso_coefficients = pd.DataFrame({
    'Feature': LASSOX_train.columns,
    'Lasso Coefficient': lasso_log_reg.coef_[0]
})

coefficients_df = pd.merge(ridge_coefficients, lasso_coefficients, on="Feature")

coefficients_df = coefficients_df.sort_values(by='Ridge Coefficient', key=abs, ascending=False)

print(coefficients_df)

fig, ax = plt.subplots(figsize=(12, 8))
coefficients_df.set_index('Feature').plot(kind='barh', ax=ax, width=0.8)
plt.title('Coefficienti delle Variabili: Ridge vs Lasso')
plt.xlabel('Peso')
plt.ylabel('Variabile')
plt.tight_layout()
plt.show()
```

Feature	Ridge Coefficient	Lasso Coefficient
amt	2.884386	2.887445
category	−0.375675	−0.375013
ym	−0.113118	−0.112725
age	0.071834	0.071580
hour	−0.039771	−0.039183
state	0.038116	0.037833
d	0.035820	0.035363
gender	−0.035261	−0.034732
merchant	0.018638	0.018123
distance	−0.018221	−0.017762
city	−0.017012	−0.016565
city_pop	0.011878	0.011611
job	0.000933	0.000647

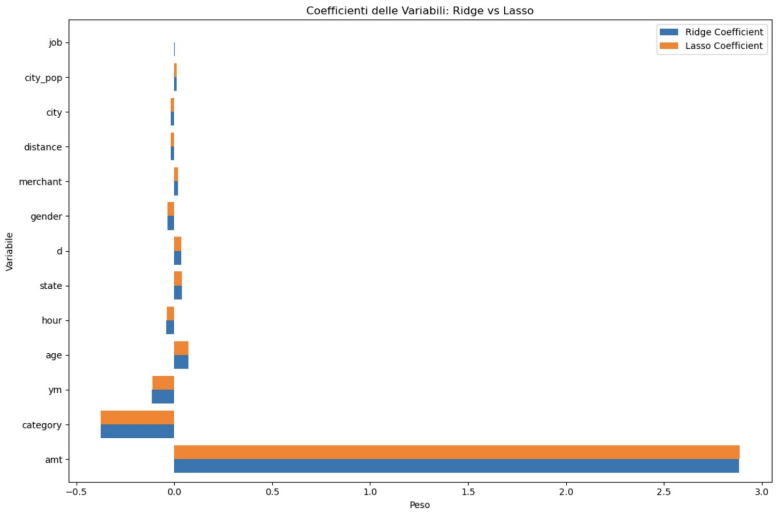


Figure 3.6: Ridge vs LASSO coefficients

The coefficients for the two regression seems to be almost equal, and LASSO regression seems to send only "job" to 0, which we have not included in our model from the EDA. For these two logistic regression we have used label encoding even though it is not the right way to encode our features because it implied an order in our categories that does not exist. The right way to encode our variables should be the one-hot encoding but the great amount of different distinct modalities makes this process computationally expencive, so we have used a more powerful machine to implement this.

```
    LASSOX_train = X_train_resampled.copy()
LASSOX_test = X_test.copy()
categorical_columns = ["merchant","category","gender","city","state","job","d","ym","hour"]

LASSOX_test["ym"]= [s[5:] for s in LASSOX_test["ym"]]
LASSOX_train["ym"]= [s[5:] for s in LASSOX_train["ym"]]
LASSOX_train= pd.get_dummies(LASSOX_train, columns=categorical_columns, drop_first=False)
LASSOX_test= pd.get_dummies(LASSOX_test, columns=categorical_columns, drop_first=False)

missing_in_test = set(LASSOX_train.columns) - set(LASSOX_test.columns)
missing_in_train = set(LASSOX_test.columns) - set(LASSOX_train.columns)

for col in missing_in_test:
    LASSOX_test[col] = 0
for col in missing_in_train:
    LASSOX_train[col] = 0
LASSOX_test = LASSOX_test[LASSOX_train.columns]

scaler = StandardScaler()
LASSOX_train_scaled= scaler.fit_transform(LASSOX_train)
LASSOX_test_scaled = scaler.transform(LASSOX_test)

ridge_log_reg = LogisticRegression(penalty='l2', C=1.0, solver='lbfgs', random_state=42)
ridge_log_reg.fit(LASSOX_train_scaled, y_train_resampled)
y_pred_ridge = ridge_log_reg.predict(LASSOX_test_scaled)

lasso_log_reg = LogisticRegression(penalty='l1', C=1.0, solver='liblinear', random_state=42)
lasso_log_reg.fit(LASSOX_train_scaled, y_train_resampled)
y_pred_lasso = lasso_log_reg.predict(LASSOX_test_scaled)

cm_ridge = confusion_matrix(y_test, y_pred_ridge)
cm_lasso = confusion_matrix(y_test, y_pred_lasso)

fig, axes = plt.subplots(1, 2, figsize=(12, 6))

sns.heatmap(cm_ridge, annot=True, fmt="d", cmap="Blues", xticklabels=[0, 1], yticklabels=[0, 1], ax=
    axes[0])
axes[0].set_title('Ridge Logistic Regression')
axes[0].set_xlabel('Predicted')
axes[0].set_ylabel('Actual')
```

```
sns.heatmap(cm_lasso, annot=True, fmt="d", cmap="Blues", xticklabels=[0, 1], yticklabels=[0, 1], ax=
    axes[1])
axes[1].set_title('Lasso Logistic Regression')
axes[1].set_xlabel('Predicted')
axes[1].set_ylabel('Actual')

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

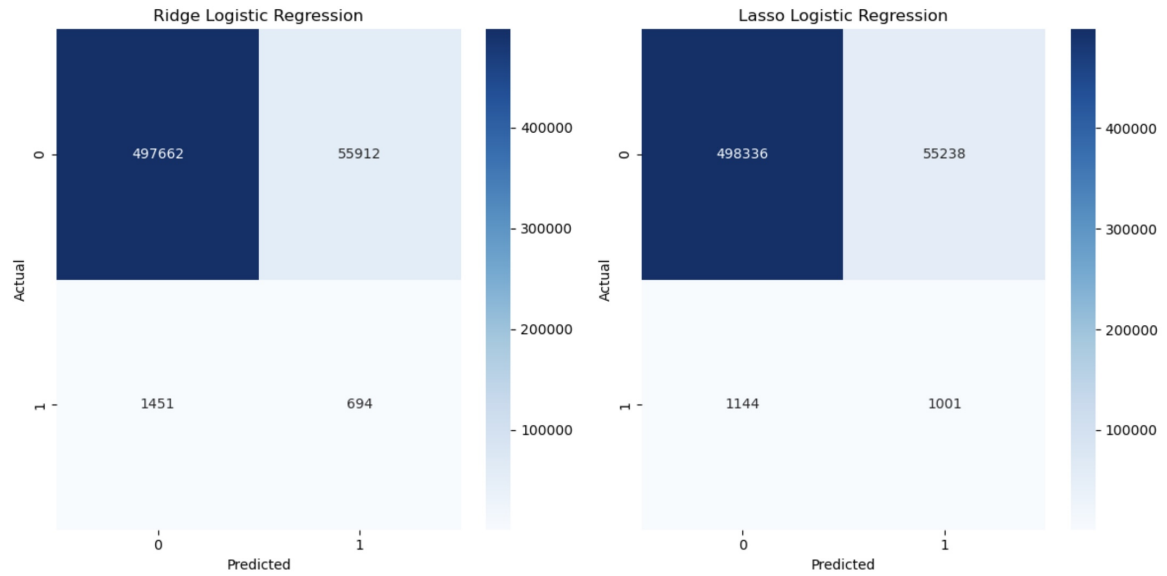


Figure 3.7: Ridge and Lasso Confusion Matrix with One-Hot Encoding

```
    import matplotlib.pyplot as plt
from sklearn.metrics import precision_recall_curve, average_precision_score

y_scores_ridge = ridge_log_reg.decision_function(LASSOX_test_scaled)
y_scores_lasso = lasso_log_reg.decision_function(LASSOX_test_scaled)

precision_ridge, recall_ridge, _ = precision_recall_curve(y_test, y_scores_ridge)
average_precision_ridge = average_precision_score(y_test, y_scores_ridge)

precision_lasso, recall_lasso, _ = precision_recall_curve(y_test, y_scores_lasso)
average_precision_lasso = average_precision_score(y_test, y_scores_lasso)

plt.figure(figsize=(10, 6))
plt.plot(recall_ridge, precision_ridge, label=f"Ridge (AP = {average_precision_ridge:.2f})",
    linewidth=2)
plt.plot(recall_lasso, precision_lasso, label=f"Lasso (AP = {average_precision_lasso:.2f})",
    linewidth=2)

plt.title("Curva Precision-Recall: Ridge vs Lasso", fontsize=14)
plt.xlabel("Recall", fontsize=12)
plt.ylabel("Precision", fontsize=12)
plt.legend(loc="upper right", fontsize=12)
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```

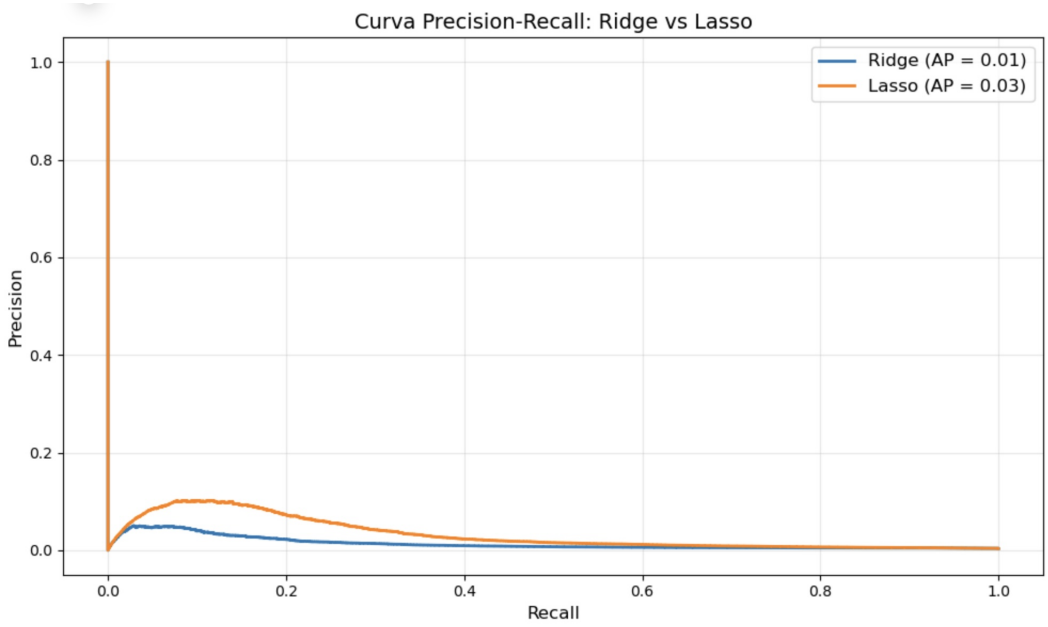


Figure 3.8: Ridge and Lasso PR Curve with One-Hot Encoding

We can see that even though label encoding is not the right choice theoretically, it brings better results, maybe because we have a lot of category in each variable.

CatBoost Classification

CatBoost is a supervised machine learning method that uses decision trees for both regression and classification tasks. It has two main features: it works efficiently with categorical data (the "Cat") and leverages gradient boosting (the "Boost").
Gradient boosting is a machine learning technique that builds a strong predictive model by combining the outputs of several weaker models, typically decision trees. It works iteratively by training each new model to minimize the residual errors of the previous models, optimizing the predictions using gradient descent on a specified loss function.
The model will be run using the same variables as the previous logistic regression. The key difference is that it is not necessary to encode the categorical variables or standardize the data, as CatBoost can handle these tasks automatically.

```
from catboost import CatBoostClassifier
from sklearn.metrics import confusion_matrix, classification_report, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
categorical_features = ['d', 'ym', 'hour',"category","state","gender"]

catboost_model = CatBoostClassifier(
    iterations=500,
    learning_rate=0.1,
    depth=6,
    random_seed=1,
    eval_metric='F1',
)

catboost_model.fit(
    EDAX_train_res,
    y_train_resampled,
    cat_features=categorical_features
)

y_pred = catboost_model.predict(EDAX_test)

cm = confusion_matrix(y_test, y_pred)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=catboost_model.classes_)
disp.plot(cmap='Blues')
plt.title('Confusion Matrix - CatBoost')
plt.show()
```

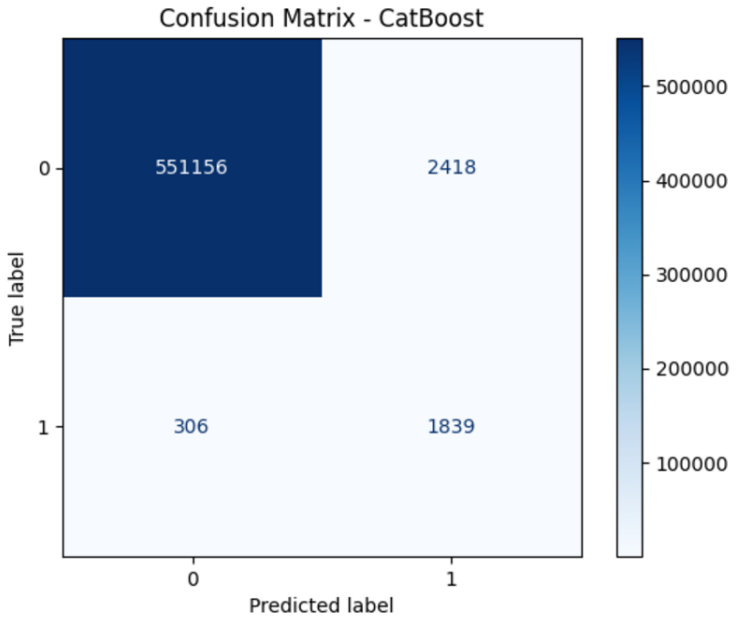


Figure 3.9: EDAX Variable CatBoost

In a CatBoost model, the parameters can be tuned to identify the optimal values and create the best model for our needs. To achieve this, we apply a 10-fold cross-validation, a technique used to evaluate a model's performance and ensure its ability to generalize.

In 10-fold cross-validation, the dataset is divided into 10 parts (folds). In each iteration, 9 folds are used for training and 1 for testing. This process is repeated for all combinations, ensuring every fold is used as a test set exactly once.

The primary goal of this technique is to optimize the model's hyperparameters, such as the maximum tree depth or the learning rate. During hyperparameter tuning (e.g., using grid search or random search), cross-validation helps identify the combination that delivers the best average performance across all folds. This approach minimizes the risk of overfitting while maximizing the effective use of the available data.

```
from sklearn.model_selection import StratifiedKFold, RandomizedSearchCV
from sklearn.utils import parallel_backend
from scipy.stats import randint

param_dist = {
    "learning_rate": [0.01, 0.03, 0.05, 0.1, 0.15],
    "max_depth": randint(4, 6),
    "n_estimators": [200, 300, 400],
    "l2_leaf_reg": [1, 2, 3, 4],
    "rsm": [0.5, 0.7, 0.9, 1, 1.1],
}

model = CatBoostClassifier(
    random_seed=1,
    cat_features=['d', 'ym', 'hour', "category", "state", "gender"],
    verbose=0
)

cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)

with parallel_backend('threading', n_jobs=-1):
    rscv = RandomizedSearchCV(
        estimator=model,
        param_distributions=param_dist,
        scoring='f1',
        cv=cv,
        n_iter=20,
        random_state=1
    )
    rscv.fit(EDAX_train_res, y_train_resampled)
```

```
print("Best Parameters Found:", rscv.best_params_)

best_model = rscv.best_estimator_
best_model.fit(EDAX_train_res, y_train_resampled)

y_pred = best_model.predict(EDAX_test)

cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=best_model.classes_)
disp.plot(cmap='Blues')
plt.title('Confusion Matrix - CatBoost (Tuned)')
plt.show()
```

```
Best Parameters Found:  {'l2_leaf_reg': 4, 'learning_rate': 0.15, 'max_depth': 5, 'n_estimators': 300, 'rsm': 0.7}
```

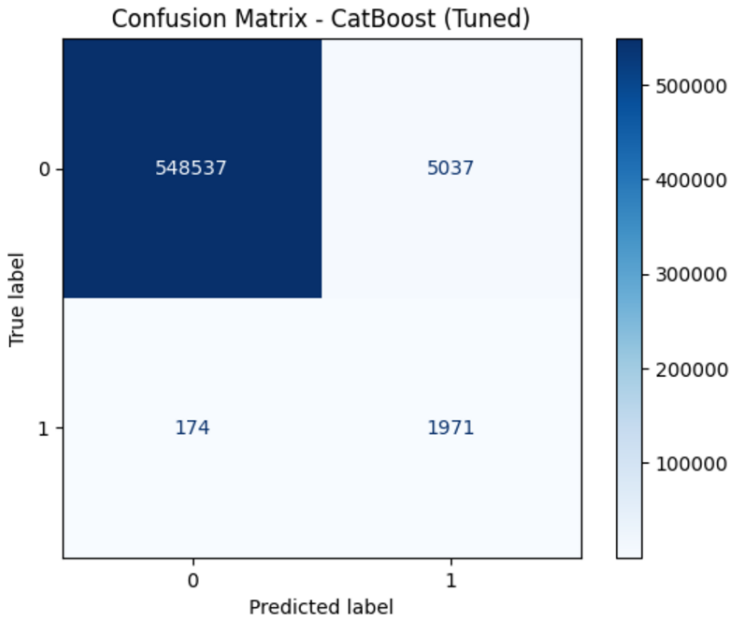


Figure 3.10: EDAX Variable Tuned CatBoost

After identifying the best parameters used to compute the model, we can refine the <param_dist> by focusing it around these values to further tune and potentially improve the model.

```
param_dist = {
    "learning_rate": [0.1, 0.12, 0.15, 0.18],
    "max_depth": randint(4, 7),
    "n_estimators": [250, 300, 350],
    "l2_leaf_reg": [3, 4, 5, 6],
    "rsm": [0.6, 0.7, 0.8],
}

model = CatBoostClassifier(
    random_seed=1,
    cat_features=['d', 'ym', 'hour', "category", "state", "gender"],
    verbose=0
)

cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)

with parallel_backend('threading', n_jobs=-1):
    rscv = RandomizedSearchCV(
        estimator=model,
        param_distributions=param_dist,
        scoring='f1',
        cv=cv,
        n_iter=20,
        random_state=1
    )
```

```
    rscv.fit(EDAX_train_res, y_train_resampled)

print("Best Parameters Found: ", rscv.best_params_)

best_model = rscv.best_estimator_
best_model.fit(EDAX_train_res, y_train_resampled)

y_pred = best_model.predict(EDAX_test)

cm = confusion_matrix(y_test, y_pred)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=best_model.classes_)
disp.plot(cmap='Blues')
plt.title('Confusion Matrix - CatBoost (Tuned)')
plt.show()
```

```
Best Parameters Found:  {'l2_leaf_reg': 3, 'learning_rate': 0.15, 'max_depth': 5, 'n_estimators': 350, 'rsm': 0.7}
```

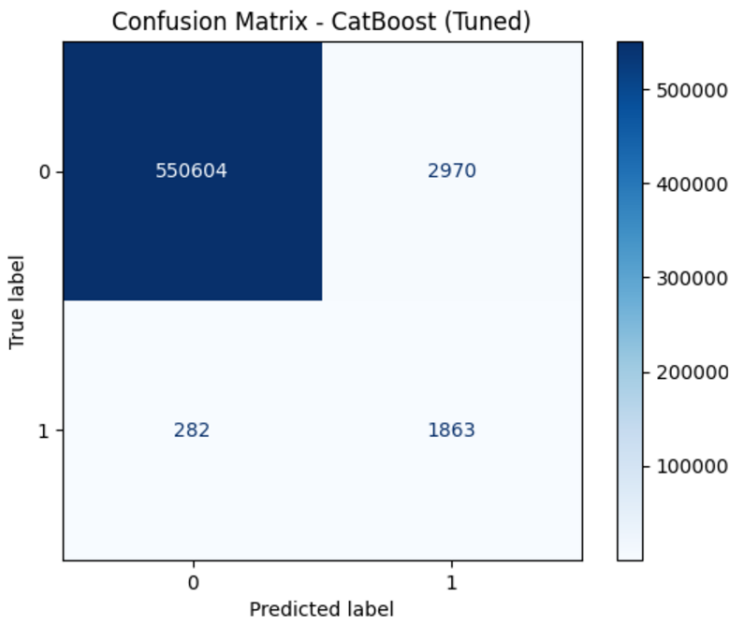


Figure 3.11: EDAX Re-Tuned CatBoost

Scoring

The <scoring parameter> defines the metric used to evaluate the model's performance during hyperparameter tuning. In Classsification we can choose between:

- Accurancy
- F1
- Precison
- Recall
- Roc_auc

The problem with imbalanced data is that accuracy becomes unreliable, as the larger class dominates the predictions, overshadowing the smaller class. However, our goal is to build a model that effectively recognizes the patterns of the smaller class, which is more challenging.

Another issue in such cases is the uncertainty regarding whether a hypothetical client would prefer fewer false positives or fewer false negatives. In a real-world scenario, the client might incur a cost for allowing a fraudulent transaction to go unnoticed, a cost for trying to block a possible fraud and a benefit when successfully blocking a fraud. Depending on this trade-off, we could adjust the model to prioritize Precision (fewer false positives) or

Recall (fewer false negatives).

When this information is unavailable, we use the F1-score as a scoring metric. The F1-score provides a balanced measure that combines Precision and Recall, considering both equally important.

An alternative metric could have been the ROC-AUC score, which evaluates the model's performance across all possible decision boundaries. While ROC-AUC offers a global view of model performance, we initially chose F1-score to prioritize fraud recognition. After selecting the best model based on F1-score, we could further validated its performance by measuring the area under the ROC curve (AUC-ROC).

Performance Evaluation

The ROC curve and the Cumulative Gain Curve are widely used to evaluate the performance of classification models, each highlighting different aspects of model behavior.

However, in this scenario, neither the ROC Curve nor the Cumulative Gain Curve proves to be effective as evaluation metrics. This is due to a significant class imbalance in the dataset. The model excels at predicting the majority class (zeros), leading to an overly optimistic and seemingly perfect ROC Curve. Similarly, the Cumulative Gain Curve, expressed in percentages, is heavily skewed by the dominance of the majority class. As a result, with the majority class (zeros) accounting for almost the entire dataset, the Cumulative Gain Curve also appears artificially ideal.

The Precision-Recall (PR) curve is a crucial tool for assessing model performance, particularly in scenarios characterized by significant class imbalance. Unlike traditional evaluation metrics that may not fully capture the model's ability to identify the minority class, the PR curve explicitly focuses on the performance of the positive class, illustrating the trade-off between precision and recall across different thresholds.

A PR curve that is closer to the top-right corner signifies a model with both high precision and recall, indicating superior performance. By analyzing the PR curve, one can determine the optimal classification threshold tailored to specific application requirements. For instance, if minimizing false negatives is of utmost importance, the model may prioritize recall, even at the cost of slightly reduced precision.

Threshold tuning involves selecting the probability threshold at which your model classifies an instance as belonging to the positive class (1). By default, most classification models use a threshold of 0.5, meaning predictions with probabilities 0.5 are classified as positive. However, this default threshold might not be ideal, especially when precision or recall needs to be prioritized.

```
import matplotlib.pyplot as plt
from sklearn.metrics import precision_recall_curve, average_precision_score

y_prob1 = logreg1.predict_proba(EDAX_test1scaled)[:, 1]
precision1, recall1, _ = precision_recall_curve(y_test, y_prob1)
average_precision1 = average_precision_score(y_test, y_prob1)

y_prob2 = catboost_model1.predict_proba(EDAX_test)[:, 1]
precision2, recall2, _ = precision_recall_curve(y_test, y_prob2)
average_precision2 = average_precision_score(y_test, y_prob2)

y_prob4 = best_model.predict_proba(EDAX_test)[:, 1]
precision4, recall4, _ = precision_recall_curve(y_test, y_prob4)
average_precision4 = average_precision_score(y_test, y_prob4)

y_prob5 = best_model2.predict_proba(EDAX_test)[:, 1]
precision5, recall5, _ = precision_recall_curve(y_test, y_prob5)
average_precision5 = average_precision_score(y_test, y_prob5)

plt.figure(figsize=(10, 8))
plt.plot(recall1, precision1, label=f'Logistic Regression (AP = {average_precision1:.2f})', color='
    blue', linewidth=2)
plt.plot(recall2, precision2, label=f'CatBoost (AP = {average_precision2:.2f})', color='orange',
    linewidth=2)
plt.plot(recall4, precision4, label=f'CatBoost Tuned (AP = {average_precision4:.2f})', color='green'
    , linewidth=2)
plt.plot(recall5, precision5, label=f'CatBoost Tuned (Best Prev Params) (AP = {average_precision5:.2
    f})', color='red', linewidth=2)
```

```
plt.xlabel('Recall', fontsize=12)
plt.ylabel('Precision', fontsize=12)
plt.title('Precision-Recall Curves for Multiple Models', fontsize=14)
plt.legend(loc='lower left', fontsize=10)
plt.grid(True)
plt.show()
```

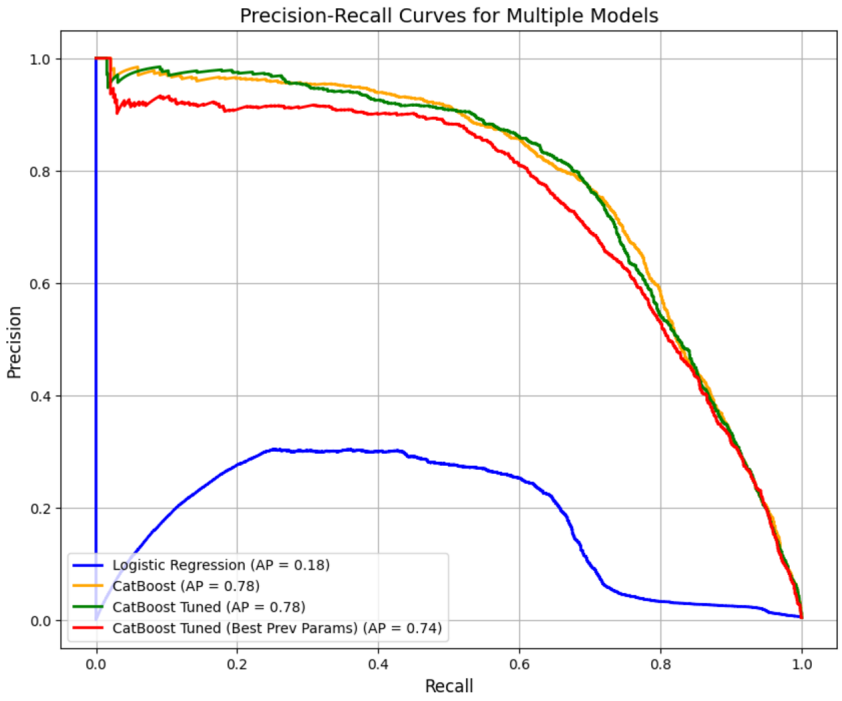


Figure 3.12: Precision-Recall Curves

When deciding which model performs better, it is crucial to evaluate their trade-offs in terms of precision and recall. Among the four evaluated models, two stand out with the same Average Precision (AP) score of 0.78: the baseline CatBoost model and the tuned CatBoost variant. This identical AP score, the highest among all models, might initially appear surprising, but it arises because AP summarizes the total area under the Precision-Recall (PR) curve. While AP offers a concise measure of overall performance, it can obscure significant differences in how each model balances precision and recall.
A closer examination of the PR curves reveals the following:

- Baseline CatBoost demonstrates higher precision at lower recall values, indicating a more conservative approach. This model focuses on reducing false positives, making fewer mistakes when predicting positive outcomes.
- Conversely, the Tuned CatBoost sacrifices some precision to achieve higher recall, making it better at identifying positives even if it results in more false positives.

These differences illustrate how the two models adopt distinct strategies, even with the same AP score. In essence, the models complement each other: the Baseline CatBoost minimizes false positives, while the Tuned CatBoost emphasizes capturing more positives, even at the expense of precision.
The choice of the better model ultimately depends on the task's objective:

- If minimizing false positives is critical—such as in scenarios where mistakes are costly (e.g., fraud detection, where false accusations can have severe repercussions)—then the **Baseline CatBoost** is the preferred model.
- If minimizing false negatives is the priority—such as in healthcare applications where missing a positive case could have critical consequences—then the **Tuned CatBoost** is the better choice.

Thus, while AP provides a valuable summary, a deeper analysis of the PR curves is essential to understand the strengths and weaknesses of each model and to align the choice with the specific objectives of the application.

3.0.2 Best Model

In the idea of minimizing false negatives, the Tuned CatBoost model appears to be the best choice. However, the model has been trained on a single sample of the training data and with a fixed random seed. To demonstrate that the selected model is robust, it needs to be evaluated using different training data samples and without a fixed random seed. If the model maintains consistent characteristics across these conditions, then it can be considered robust.

```
final_params = {
    "learning_rate": 0.15,
    "max_depth": 5,
    "n_estimators": 300,
    "l2_leaf_reg": 4,
    "rsm": 0.7,
    "cat_features": ['d', 'ym', 'hour', "category", "state", "gender"],
    "verbose": 0
}

results = []

for i in range(10):
    rus = RandomUnderSampler(random_state=None)
    X_train_resampled, y_train_resampled = rus.fit_resample(X_train, y_train)
    EDAX_train_res = X_train_resampled[["d", "ym", "hour", "category", "state", "gender", "age", "
        amt"]]
    EDAX_test = X_test[["d", "ym", "hour", "category", "state", "gender", "age", "amt"]]

    model = CatBoostClassifier(**final_params)
    model.fit(EDAX_train_res, y_train_resampled)
    y_pred = model.predict(EDAX_test)

    report = classification_report(y_test, y_pred, output_dict=True)
    if '1' in report:
        results.append({
            "Iteration": i + 1,
            "Class": 1,
            "Recall": report['1']['recall'],
            "Precision": report['1']['precision']
        })

results_df = pd.DataFrame(results)

mean_recall = results_df['Recall'].mean()
mean_precision = results_df['Precision'].mean()

summary = pd.DataFrame([{
    "Iteration": "Overall",
    "Class": 1,
    "Recall": mean_recall,
    "Precision": mean_precision
}])
results_df = pd.concat([results_df, summary], ignore_index=True)
print(results_df)
```

	Iteration	Class	Recall	Precision
0	1	1	0.784615	0.512797
1	2	1	0.765967	0.548764
2	3	1	0.879720	0.383303
3	4	1	0.847086	0.446218
4	5	1	0.846620	0.440670
5	6	1	0.878322	0.386938
6	7	1	0.882517	0.367716
7	8	1	0.855012	0.455200
8	9	1	0.846620	0.467921
9	10	1	0.892774	0.347046
10	Overall	1	0.847925	0.435657

Figure 3.13: Performance Table

Performing cross-validation in this case is unnecessary, as we are already testing the model's robustness by varying the training set. Based on the results shown in the table, the model demonstrates stability across different training sets.

3.0.3 Feature Importance

```
feature_importances = rscv2.best_estimator_.feature_importances_
importance_df = pd.DataFrame({
    'Feature': EDAX_test.columns,
    'Importance': feature_importances
}).sort_values(by='Importance', ascending=False)

print(importance_df)

plt.figure(figsize=(10, 6))
plt.barh(importance_df['Feature'], importance_df['Importance'], color='blue')
plt.xlabel('Importance')
plt.ylabel('Features')
plt.title('Feature Importance')
plt.gca().invert_yaxis()
plt.show()
```

	Feature	Importance
7	amt	51.149080
3	category	20.310361
2	hour	8.925779
4	state	5.449268
1	ym	5.102157
0	d	3.866076
6	age	3.191486
5	gender	2.005794

Figure 3.14: Performance Table

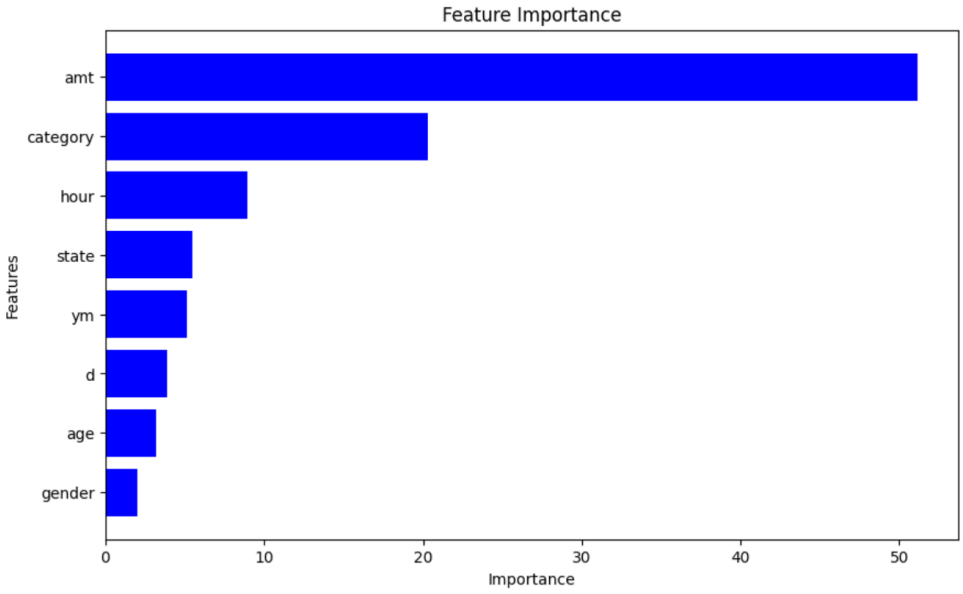


Figure 3.15: Performance Table

The analysis shows that the top three variables (<amt>, <category>, and <hour>) contribute approximately 90% to the model's predictive power. This suggests that these features capture the majority of the information needed for accurate predictions.
In practical applications, it may be possible to create a simplified model using only these three variables. While this would likely lead to a slight decrease in overall model performance, it could offer significant advantages for the business, such as cost reduction and operational simplicity. For example:

- Cost Efficiency: Collecting fewer variables reduces the expenses associated with data acquisition, storage, and processing. If certain features, like <state> or <age>, are expensive or resource-intensive to gather, their exclusion can save resources.
- Privacy Compliance: Variables like <gender> or <state> may raise privacy concerns or be subject to strict data protection regulations. A simplified model reduces reliance on sensitive data, minimizing legal risks and enhancing user trust.
- Operational Simplicity: A model with fewer variables is easier to implement and maintain. It can also process predictions faster, which is particularly beneficial in time-sensitive applications.

Ultimately, the acceptability of this trade-off depends on the specific business context. To assess its feasibility, it is recommended to test the simplified model and compare its performance with the full version to quantify any potential degradation in predictive accuracy.

It has been considered plotting a tree during cross-validation in CatBoost is generally not a good idea because of how the cross-validation process works. In cross-validation, the model is trained on different subsets of the data (the folds), and each fold produces its own unique set of decision trees. These trees may differ in structure, depending on the data seen by each fold, and there isn't a single, consolidated tree that represents the overall model. Since cross-validation's purpose is to assess model generalization and performance across different data splits, trying to visualize a tree from this intermediate process can be misleading.

CatBoost's use of oblivious trees (symmetrical decision trees where the same feature is split on at each level) adds another layer of complexity. Since each fold uses a different training subset, the tree structures will vary, making it difficult to capture the full picture with a single tree plot.

Chapter 4

Conclusion

In conclusion, this project successfully developed and evaluated models to identify fraudulent transactions within a simulated credit card dataset. The dataset, containing over 1.3 million transactions, presented a significant challenge due to the inherent class imbalance between legitimate and fraudulent transactions. Through detailed exploratory data analysis (EDA), key patterns and relationships were identified, guiding the selection of impactful features. These features were carefully engineered, with redundant and less informative variables removed to streamline the modeling process.

The project employed multiple machine learning approaches, including Logistic Regression with LASSO regularization and the CatBoost classifier, to tackle this classification problem. Logistic Regression demonstrated the importance of feature scaling and careful encoding, while the CatBoost model excelled in leveraging categorical variables without requiring additional preprocessing. To address class imbalance, an undersampling strategy was applied during training, ensuring both classes were adequately represented without overfitting the model.

Performance metrics such as confusion matrices, ROC curves, and precision-recall analyses provided insights into each model's ability to distinguish fraudulent transactions. The CatBoost classifier emerged as the most effective approach, showcasing its robustness and adaptability in handling the complexities of fraud detection. Hyperparameter tuning and cross-validation further enhanced model performance, leading to improved precision and recall rates, critical for minimizing false positives and false negatives in fraud detection.

This work underscores the significance of integrating domain knowledge with advanced machine learning techniques to address real-world challenges. The findings provide a strong foundation for deploying fraud detection systems in real-world scenarios, offering scalability and adaptability. Future directions include incorporating real-time detection mechanisms, exploring deep learning methods for enhanced pattern recognition, and expanding the dataset with more recent and diverse transaction records to improve generalizability and robustness.