Credit Card Fraud Detection

Generating Transaction Data

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random
import seaborn as sns
import os
import datetime
import math
```

```
# Step 1 - Generation of customer profiles

def generate_customer_profiles_table(n_customers, random_state=0):

    np.random.seed(random_state)

    customer_id_properties=[]

    # Generate customer properties from random distributions
    for customer_id in range(n_customers):

        latitude = np.random.uniform(-90, 90)  # Latitude ranges from -90 to 90 degrees
        longitude = np.random.uniform(-180, 180)  # Longitude ranges from -180 to 180 degrees

        mean_amount = np.random.uniform(5,10000) # Arbitrary (but sensible) value
        std_amount = mean_amount/2 # Arbitrary (but sensible) value

        mean_nb_tx_per_day = np.random.uniform(0,8) # Arbitrary (but sensible) value

        # add a gender column
        gender = np.random.choice(['F', 'M'])  # Randomly select 'F' or 'M' for gender

        customer_id_properties.append([customer_id,
                                      latitude, longitude,
                                      mean_amount, std_amount,
                                      mean_nb_tx_per_day, gender])

    customer_profiles_table = pd.DataFrame(customer_id_properties, columns=['CUSTOMER_ID',
                                                                      'customer_latitude', 'customer_longitude',
                                                                      'mean_amount', 'std_amount',
                                                                      'mean_nb_tx_per_day', 'GENDER'])

    return customer_profiles_table


n_customers = 15
customer_profiles_table = generate_customer_profiles_table(n_customers, random_state = 0)
customer_profiles_table
```

	CUSTOMER_ID	customer_latitude	customer_longitude	mean_amount	std_amount	mean_nb_tx_per_day	G
0	0	8.786431	77.468172	6029.619944	3014.809972	4.359065	
1	1	22.241465	-41.622585	2978.858392	1489.429196	0.453704	
2	2	-20.980527	105.021014	5291.304723	2645.652361	4.544356	
3	3	60.494177	-58.537382	6483.477861	3241.738931	2.945932	

```
# Step 2 - Generation of terminal profiles

def generate_terminal_profiles_table(n_terminals, random_state=0):

    np.random.seed(random_state)

    terminal_id_properties=[]

    # Generate terminal properties from random distributions
    for terminal_id in range(n_terminals):

        latitude = np.random.uniform(-90, 90)  # Latitude ranges from -90 to 90 degrees
        longitude = np.random.uniform(-180, 180)  # Longitude ranges from -180 to 180 degrees

        terminal_id_properties.append([terminal_id,
                                      latitude, longitude])

    terminal_profiles_table = pd.DataFrame(terminal_id_properties, columns=['TERMINAL_ID',
                                                                           'terminal_latitude', 'terminal_longitude'])

    return terminal_profiles_table


n_terminals = 15
terminal_profiles_table = generate_terminal_profiles_table(n_terminals, random_state = 0)
terminal_profiles_table
```

	TERMINAL_ID	terminal_latitude	terminal_longitude
0	0	8.786431	77.468172
1	1	18.497408	16.157946
2	2	-13.742136	52.521881
3	3	-11.234302	141.038280
4	4	83.459297	-41.961053
5	5	52.510507	10.402171
6	6	12.248021	153.214790
7	7	-77.213510	-148.633452
8	8	-86.360688	119.743144
9	9	50.068215	133.204373
10	10	86.151302	107.697083
11	11	-6.933715	100.990503
12	12	-68.710603	50.371568
13	13	-64.196408	160.080810
14	14	3.932698	-30.721702

```
# Step 3 - Association of customer profiles to terminals

def get_list_terminals_within_radius(customer_profile, x_y_terminals, r):

    # Use numpy arrays in the following to speed up computations

    # Location (x,y) of customer as numpy array
    x_y_customer = customer_profile[['customer_latitude','customer_longitude']].values.astype(float)

    # Squared difference in coordinates between customer and terminal locations
    squared_diff_x_y = np.square(x_y_customer - x_y_terminals)

    # Sum along rows and compute suared root to get distance
```

```
    dist_x_y = np.sqrt(np.sum(squared_diff_x_y, axis=1))

    # Get the indices of terminals which are at a distance less than r
    available_terminals = list(np.where(dist_x_y<r)[0])

    # Return the list of terminal IDs
    return available_terminals


# We first get the geographical locations of all terminals as a numpy array
x_y_terminals = terminal_profiles_table[['terminal_latitude','terminal_longitude']].values.astype(float)
# And get the list of terminals within radius of $50$ for the last customer
get_list_terminals_within_radius(customer_profiles_table.iloc[14], x_y_terminals=x_y_terminals, r=50)
```

```
[7]
```

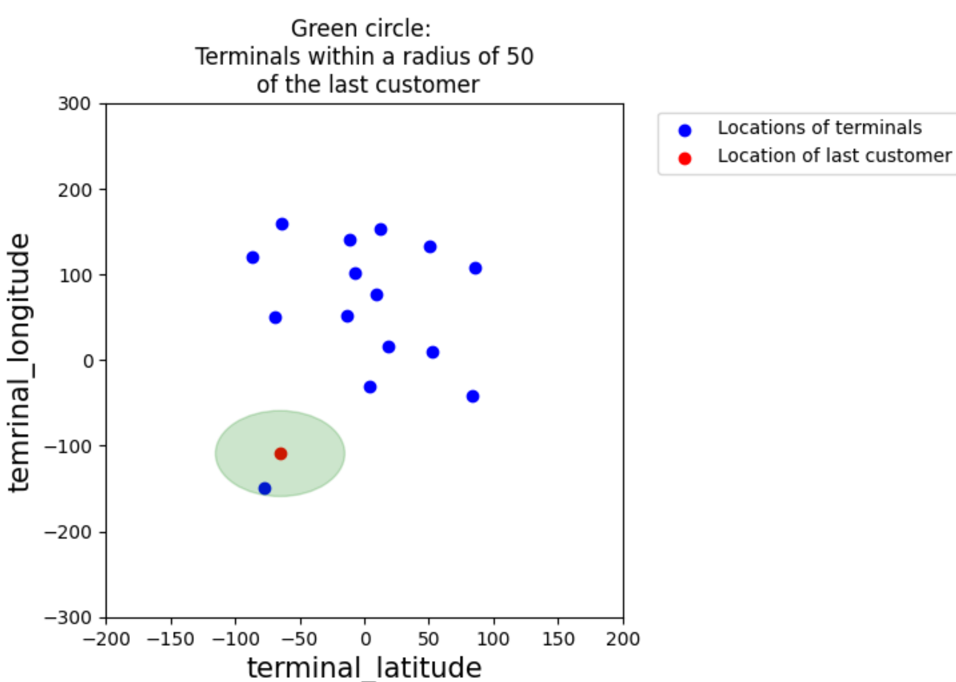
```
# Haversine formula implementation
def haversine(lat1, lon1, lat2, lon2):
    # distance between latitudes
    # and longitudes
    dLat = (lat2 - lat1) * math.pi / 180.0
    dLon = (lon2 - lon1) * math.pi / 180.0

    # convert to radians
    lat1 = (lat1) * math.pi / 180.0
    lat2 = (lat2) * math.pi / 180.0

    # apply formula
    a = (pow(math.sin(dLat / 2), 2) +
         pow(math.sin(dLon / 2), 2) *
             math.cos(lat1) * math.cos(lat2));
    rad = 6371
    c = 2 * math.asin(math.sqrt(a))
    return rad * c


# Alternate Step 3 - use the haversine formula to compute terminals
def get_list_terminals_within_distance(customer_profile, terminals, radius):
  customer_lat = customer_profile['customer_latitude']
  customer_lon = customer_profile['customer_longitude']
  available_terminals = []

  for index in terminals.index:
    terminal_lat = terminals['terminal_latitude'][index]
    terminal_long = terminals['terminal_longitude'][index]
    distance = haversine(customer_lat, customer_lon, terminal_lat, terminal_long)

    if distance <= radius:
      available_terminals.append(terminals['TERMINAL_ID'][index])

  return available_terminals




# Get the new list of terminals within 4500 km for the last customer
get_list_terminals_within_distance(customer_profiles_table.iloc[14],terminal_profiles_table, 4500)
```

```
[7, 8, 13]
```

```
terminal_profiles_table
```

	TERMINAL_ID	terminal_latitude	terminal_longitude
0	0	8.786431	77.468172
1	1	18.497408	16.157946
2	2	-13.742136	52.521881
3	3	-11.234302	141.038280
4	4	83.459297	-41.961053
5	5	52.510507	10.402171
6	6	12.248021	153.214790

```
%%capture

terminals_available_to_customer_fig, ax = plt.subplots(figsize=(5,5))

# Plot locations of terminals
ax.scatter(terminal_profiles_table.terminal_latitude.values,
           terminal_profiles_table.terminal_longitude.values,
           color='blue', label = 'Locations of terminals')

# Plot location of the last customer
customer_id=14
ax.scatter(customer_profiles_table.iloc[customer_id].customer_latitude,
           customer_profiles_table.iloc[customer_id].customer_longitude,
           color='red',label="Location of last customer")

ax.legend(loc = 'upper left', bbox_to_anchor=(1.05, 1))

# Plot the region within a radius of 50 of the last customer
circ = plt.Circle((customer_profiles_table.iloc[customer_id].customer_latitude,
                   customer_profiles_table.iloc[customer_id].customer_longitude), radius=50, color='g', alpha=0.2)
ax.add_patch(circ)

fontsize=15

ax.set_title("Green circle: \n Terminals within a radius of 50 \n of the last customer")
ax.set_xlim([-200, 200])
ax.set_ylim([-300, 300])

ax.set_xlabel('terminal_latitude', fontsize=fontsize)
ax.set_ylabel('temrinal_longitude', fontsize=fontsize)


terminals_available_to_customer_fig
```



```
customer_profiles_table['available_terminals']=customer_profiles_table.apply(lambda x : get_list_terminals_within_distance(x, terminal_profil
customer_profiles_table
```

	CUSTOMER_ID	customer_latitude	customer_longitude	mean_amount	std_amount	mean_nb_tx_per_day	G
0	0	8.786431	77.468172	6029.619944	3014.809972	4.359065	
1	1	22.241465	-41.622585	2978.858392	1489.429196	0.453704	
2	2	-20.980527	105.021014	5291.304723	2645.652361	4.544356	
3	3	60.494177	-58.537382	6483.477861	3241.738931	2.945932	
4	4	50.068215	133.204373	9786.290331	4893.145165	6.393269	
5	5	3.685946	64.396631	7207.723384	3603.861692	4.656158	
6	6	80.040405	7.865396	4149.546090	2074.773045	2.116445	
7	7	42.645272	-102.041872	1356.505643	678.252822	2.593128	
8	8	20.177230	42.096239	9437.762045	4718.881022	5.454562	
9	9	20.351422	144.845490	997.307102	498.653551	7.758473	
10	10	30.714817	-104.262278	1293.618345	646.809173	2.523427	
11	11	-31.491499	-166.166846	6344.569209	3172.284605	7.671594	
12	12	-52.402184	-121.928574	6532.817713	3266.408857	2.026333	
13	13	-4.554450	44.463636	3383.386110	1691.693055	5.398019	
14	14	-65.127069	-109.230350	3690.408081	1845.204040	6.567946	

```
# Step 4 - Generation of transactions

def generate_transactions_table(customer_profile, start_date = "2022-04-01", nb_days = 30):

    customer_transactions = []

    random.seed(int(customer_profile.CUSTOMER_ID))
    np.random.seed(int(customer_profile.CUSTOMER_ID))

    # For all days
    for day in range(nb_days):

        # Random number of transactions for that day
        nb_tx = np.random.poisson(customer_profile.mean_nb_tx_per_day)

        # If nb_tx positive, let us generate transactions
        if nb_tx>0:

            for tx in range(nb_tx):

                # Time of transaction: Around noon, std 20000 seconds. This choice aims at simulating the fact that
                # most transactions occur during the day.
                time_tx = int(np.random.normal(86400/2, 20000))

                # If transaction time between 0 and 86400, let us keep it, otherwise, let us discard it
                if (time_tx>0) and (time_tx<86400):

                    # Amount is drawn from a normal distribution
                    amount = np.random.normal(customer_profile.mean_amount, customer_profile.std_amount)

                    # If amount negative, draw from a uniform distribution
                    if amount<0:
                        amount = np.random.uniform(0,customer_profile.mean_amount*2)

                    amount=np.round(amount,decimals=2)

                    if len(customer_profile.available_terminals)>0:

                        terminal_id = random.choice(customer_profile.available_terminals)

                        customer_transactions.append([time_tx+day*86400, day,
                                                      customer_profile.CUSTOMER_ID,
                                                      terminal_id, amount])

    customer_transactions = pd.DataFrame(customer_transactions, columns=['TX_TIME_SECONDS', 'TX_TIME_DAYS', 'CUSTOMER_ID', 'TERMINAL_ID', 'TX
```

```
    if len(customer_transactions)>0:
        customer_transactions['TX_DATETIME'] = pd.to_datetime(customer_transactions["TX_TIME_SECONDS"], unit='s', origin=start_date)
        customer_transactions=customer_transactions[['TX_DATETIME','CUSTOMER_ID', 'TERMINAL_ID', 'TX_AMOUNT','TX_TIME_SECONDS', 'TX_TIME_DAYS

    return customer_transactions
```

```
transaction_table_customer_0=generate_transactions_table(customer_profiles_table.iloc[3],
                                                         start_date = "2022-04-01",
                                                         nb_days = 100)
transaction_table_customer_0
```

	TX_DATETIME	CUSTOMER_ID	TERMINAL_ID	TX_AMOUNT	TX_TIME_SECONDS	TX_TIME_DAYS
0	2022-04-01 09:26:39	3	4	6295.71	33999	0
1	2022-04-01 23:32:31	3	10	4534.18	84751	0
2	2022-04-01 17:13:04	3	10	2656.90	61984	0
3	2022-04-01 10:04:13	3	4	6698.40	36253	0
4	2022-04-02 08:53:03	3	5	12380.14	118383	1
...	...	...	...	...	...	...
309	2022-07-08 05:05:18	3	5	8490.84	8485518	98
310	2022-07-08 11:11:52	3	10	5781.52	8507512	98
311	2022-07-09 15:56:46	3	4	1873.20	8611006	99
312	2022-07-09 11:53:25	3	4	8522.21	8596405	99
313	2022-07-09 11:52:02	3	10	5988.80	8596322	99

314 rows × 6 columns

```
transactions_df=customer_profiles_table.groupby('CUSTOMER_ID').apply(lambda x : generate_transactions_table(x.iloc[0], nb_days=30)).reset_ind
transactions_df
```

	TX_DATETIME	CUSTOMER_ID	TERMINAL_ID	TX_AMOUNT	TX_TIME_SECONDS	TX_TIME_DAYS
0	2022-04-01 12:48:00	0	2	10413.98	46080	0
1	2022-04-01 16:13:40	0	2	6396.45	58420	0
2	2022-04-01 14:27:57	0	0	7035.58	52077	0
3	2022-04-01 20:18:01	0	2	5411.11	73081	0
4	2022-04-01 13:44:21	0	11	3454.68	49461	0
...	...	...	...	...	...	...
1726	2022-04-30 16:37:07	14	8	5272.01	2565427	29
1727	2022-04-30 15:20:51	14	8	6278.88	2560851	29
1728	2022-04-30 12:19:11	14	8	3562.26	2549951	29
1729	2022-04-30 07:25:23	14	7	4560.90	2532323	29
1730	2022-04-30 15:05:39	14	8	4786.12	2559939	29

1731 rows × 6 columns

```
import time
```

```
def generate_dataset(n_customers = 5000, n_terminals = 1000000, nb_days=100, start_date="2022-04-01", r=100):

    start_time=time.time()
    customer_profiles_table = generate_customer_profiles_table(n_customers, random_state = 0)
    print("Time to generate customer profiles table: {0:.2}s".format(time.time()-start_time))

    start_time=time.time()
    terminal_profiles_table = generate_terminal_profiles_table(n_terminals, random_state = 1)
    print("Time to generate terminal profiles table: {0:.2}s".format(time.time()-start_time))

    start_time=time.time()
    #x_y_terminals = terminal_profiles_table[['terminal_latitude','terminal_longitude']].values.astype(float)
    customer_profiles_table['available_terminals'] = customer_profiles_table.apply(lambda x : get_list_terminals_within_distance(x, terminal_
    customer_profiles_table['nb_terminals']=customer_profiles_table.available_terminals.apply(len)
```

```
    print("Time to associate terminals to customers: {0:.2}s".format(time.time()-start_time))

    start_time=time.time()
    transactions_df=customer_profiles_table.groupby('CUSTOMER_ID').apply(lambda x : generate_transactions_table(x.iloc[0], nb_days=nb_days)).
    print("Time to generate transactions: {0:.2}s".format(time.time()-start_time))

    # Sort transactions chronologically
    transactions_df=transactions_df.sort_values('TX_DATETIME')
    # Reset indices, starting from 0
    transactions_df.reset_index(inplace=True,drop=True)
    transactions_df.reset_index(inplace=True)
    # TRANSACTION_ID are the dataframe indices, starting from 0
    transactions_df.rename(columns = {'index':'TRANSACTION_ID'}, inplace = True)

    return (customer_profiles_table, terminal_profiles_table, transactions_df)
```

```
(customer_profiles_table, terminal_profiles_table, transactions_df)=\
    generate_dataset(n_customers = 5000,
                     n_terminals = 10000,
                     nb_days=100,
                     start_date="2022-04-01",
                     r=4500)
```

```
transactions_df.shape
```

```
transactions_df
```

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

fig, ax = plt.subplots(1, 2, figsize=(18, 4))

amount_val = transactions_df[transactions_df.TX_TIME_DAYS < 10]['TX_AMOUNT'].sample(n=10000).values
time_val = transactions_df[transactions_df.TX_TIME_DAYS < 10]['TX_TIME_SECONDS'].sample(n=10000).values

sns.histplot(amount_val, ax=ax[0], color='r', kde=False)
ax[0].set_title('Distribution of transaction amounts', fontsize=14)
ax[0].set_xlim([min(amount_val), max(amount_val)])
ax[0].set_xlabel("Amount")
ax[0].set_ylabel("Number of transactions")

# We divide the time variables by 86400 to transform seconds to days in the plot
sns.histplot(time_val / 86400, ax=ax[1], color='b', bins=100, kde=False)
ax[1].set_title('Distribution of transaction times', fontsize=14)
ax[1].set_xlim([min(time_val / 86400), max(time_val / 86400)])
ax[1].set_xticks(range(10))
ax[1].set_xlabel("Time (days)")
ax[1].set_ylabel("Number of transactions")

plt.show()
```

```
# Step 5 - Generation of fraud scenarios

def add_frauds(customer_profiles_table, terminal_profiles_table, transactions_df):

    # By default, all transactions are genuine
    transactions_df['TX_FRAUD']=0
    transactions_df['TX_FRAUD_SCENARIO']=0

    # Scenario 1
    #transactions_df.loc[transactions_df.TX_AMOUNT>220, 'TX_FRAUD']=1
    #transactions_df.loc[transactions_df.TX_AMOUNT>220, 'TX_FRAUD_SCENARIO']=1
    #nb_frauds_scenario_1=transactions_df.TX_FRAUD.sum()
    #print("Number of frauds from scenario 1: "+str(nb_frauds_scenario_1))

    # Scenario 2 - two terminals drawn at random and marked as fraud for 28 days (phishing)
    for day in range(transactions_df.TX_TIME_DAYS.max()):

        compromised_terminals = terminal_profiles_table.TERMINAL_ID.sample(n=2, random_state=day)

        compromised_transactions=transactions_df[(transactions_df.TX_TIME_DAYS>=day) &
                                                 (transactions_df.TX_TIME_DAYS<day+28) &
```

```
                                                    (transactions_df.TERMINAL_ID.isin(compromised_terminals))]

        transactions_df.loc[compromised_transactions.index,'TX_FRAUD']=1
        transactions_df.loc[compromised_transactions.index,'TX_FRAUD_SCENARIO']=2

    nb_frauds_scenario_2=transactions_df.TX_FRAUD.sum()
    print("Number of frauds from scenario 2: "+str(nb_frauds_scenario_2))

    # Scenario 3 - 3 customers drawn at random and 1/3 of thier transactions are high amounts (stolen card numbers)
    for day in range(transactions_df.TX_TIME_DAYS.max()):

        compromised_customers = customer_profiles_table.CUSTOMER_ID.sample(n=3, random_state=day).values

        compromised_transactions=transactions_df[(transactions_df.TX_TIME_DAYS>=day) &
                                                    (transactions_df.TX_TIME_DAYS<day+14) &
                                                    (transactions_df.CUSTOMER_ID.isin(compromised_customers))]

        nb_compromised_transactions=len(compromised_transactions)


        random.seed(day)
        index_fauds = random.sample(list(compromised_transactions.index.values),k=int(nb_compromised_transactions/3))

        transactions_df.loc[index_fauds,'TX_AMOUNT']=transactions_df.loc[index_fauds,'TX_AMOUNT']*5
        transactions_df.loc[index_fauds,'TX_FRAUD']=1
        transactions_df.loc[index_fauds,'TX_FRAUD_SCENARIO']=3


    nb_frauds_scenario_3=transactions_df.TX_FRAUD.sum()-nb_frauds_scenario_2
    print("Number of frauds from scenario 3: "+str(nb_frauds_scenario_3))

    return transactions_df


%time transactions_df = add_frauds(customer_profiles_table, terminal_profiles_table, transactions_df)


# Percentage of fraudulent transactions:
transactions_df.TX_FRAUD.mean()


transactions_df.TX_FRAUD.sum()


transactions_df[transactions_df.TX_FRAUD_SCENARIO==2].shape


transactions_df[transactions_df.TX_FRAUD_SCENARIO==3].shape


def get_stats(transactions_df):
    #Number of transactions per day
    nb_tx_per_day=transactions_df.groupby(['TX_TIME_DAYS'])['CUSTOMER_ID'].count()
    #Number of fraudulent transactions per day
    nb_fraud_per_day=transactions_df.groupby(['TX_TIME_DAYS'])['TX_FRAUD'].sum()
    #Number of fraudulent cards per day
    nb_fraudcard_per_day=transactions_df[transactions_df['TX_FRAUD']>0].groupby(['TX_TIME_DAYS']).CUSTOMER_ID.nunique()

    return (nb_tx_per_day,nb_fraud_per_day,nb_fraudcard_per_day)

(nb_tx_per_day,nb_fraud_per_day,nb_fraudcard_per_day)=get_stats(transactions_df)

n_days=len(nb_tx_per_day)
tx_stats=pd.DataFrame({"value":pd.concat([nb_tx_per_day/50,nb_fraud_per_day,nb_fraudcard_per_day])})
tx_stats['stat_type']=["nb_tx_per_day"]*n_days+["nb_fraud_per_day"]*n_days+["nb_fraudcard_per_day"]*n_days
tx_stats=tx_stats.reset_index()


%%capture

sns.set(style='darkgrid')
sns.set(font_scale=1.4)

fraud_and_transactions_stats_fig = plt.gcf()

fraud_and_transactions_stats_fig.set_size_inches(15, 8)

sns_plot = sns.lineplot(x="TX_TIME_DAYS", y="value", data=tx_stats, hue="stat_type", hue_order=["nb_tx_per_day","nb_fraud_per_day","nb_fraudc
```

```
sns_plot.set_title('Total transactions, and number of fraudulent transactions \n and number of compromised cards per day', fontsize=20)
sns_plot.set(xlabel = "Number of days since beginning of data generation", ylabel="Number")

sns_plot.set_ylim([0,700])

labels_legend = ["# transactions per day (/50)", "# fraudulent txs per day", "# fraudulent cards per day"]

sns_plot.legend(loc='upper left', labels=labels_legend,bbox_to_anchor=(1.05, 1), fontsize=15)


fraud_and_transactions_stats_fig


# Step 6 - Saving the dataset

DIR_OUTPUT = "./simulated-data-raw/"

if not os.path.exists(DIR_OUTPUT):
    os.makedirs(DIR_OUTPUT)

#start_date = datetime.datetime.strptime("2018-04-01", "%Y-%m-%d")

# for day in range(transactions_df.TX_TIME_DAYS.max() + 1):

#     transactions_day = transactions_df[transactions_df.TX_TIME_DAYS == day].sort_values('TX_TIME_SECONDS')

#date = start_date + datetime.timedelta(days=day)
#filename_output = date.strftime("%Y-%m-%d") + '.csv'

# Convert GENDER column to non-categorical, Female is 0 and Male is 1
gender_mapping = {'F': 0, 'M': 1}
customer_profiles_table['GENDER'].replace(gender_mapping, inplace=True)

# Merge tranaction_df and other dataset
merged_df = pd.merge(transactions_df, customer_profiles_table, how='left', left_on='CUSTOMER_ID', right_on='CUSTOMER_ID')
final_merged_df = pd.merge(merged_df, terminal_profiles_table, how='left', left_on='TERMINAL_ID', right_on='TERMINAL_ID')

# Save the transactions_day DataFrame as a CSV file
final_merged_df.to_csv(os.path.join(DIR_OUTPUT, "transaction_test.csv"), index=False)
```

Dealing with class imbalance using SMOTE

```
data = pd.read_csv("./simulated-data-raw/transaction_test.csv")


# only use certain columns in the dataset
data.drop(columns = ['available_terminals'], inplace = True)


from imblearn.over_sampling import BorderlineSMOTE
from datetime import datetime, timedelta


#SMOTE cannot handle datetime type. Hence, Dropping the column before sampling and adding it back after sampling
original_datetime = data['TX_DATETIME']

# Extract features and target variable
X = data.drop(['TX_DATETIME', 'TX_FRAUD'], axis=1)
y = data['TX_FRAUD']

# Initialize SMOTE
smote = BorderlineSMOTE(random_state=42)

# Apply SMOTE to create synthetic samples
X_resampled, y_resampled = smote.fit_resample(X, y)

# Create a new balanced DataFrame
transactions_balanced = pd.DataFrame(X_resampled, columns=X.columns)
transactions_balanced.insert(6, 'TX_FRAUD', y_resampled)
transactions_balanced.insert(1, 'TX_DATETIME', original_datetime)

#Fill the NA values in TX_DATETIME column with random values within the below mentioned range
start_date = datetime(2022, 4, 1)
end_date = start_date + timedelta(days=100)
```

```
def generate_random_datetime():
    random_date = start_date + timedelta(days=random.randint(0, 100))
    random_time = timedelta(seconds=random.randint(0, 86400))  # 86400 seconds in a day
    return random_date + random_time

transactions_balanced['TX_DATETIME'] = transactions_balanced['TX_DATETIME'].fillna(transactions_balanced['TX_DATETIME'].apply(lambda x: gener

# Check the class distribution
print("Class Distribution in Original Dataset:")
print(y.value_counts(normalize=True))

print("\nClass Distribution in Balanced Dataset (after SMOTE):")
print(transactions_balanced['TX_FRAUD'].value_counts(normalize=True))
```

```
DIR_OUTPUT = "./simulated-data-raw/"

if not os.path.exists(DIR_OUTPUT):
    os.makedirs(DIR_OUTPUT)
```

```
transactions_balanced.to_csv(os.path.join(DIR_OUTPUT, "transactions_balanced.csv"), index=False)
```

Splitting & Normalizing Data

```
# upload the csv
from google.colab import files
uploaded = files.upload()
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
import pandas as pd
from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold, GridSearchCV
from sklearn.metrics import confusion_matrix,accuracy_score, classification_report
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
```

```
data = pd.read_csv('transactions_balanced.csv')
```

```
# Drop columns that won't train the model well and may result in overfitting
drop_columns = ["TRANSACTION_ID", "TX_FRAUD_SCENARIO", "GENDER", "nb_terminals"]
data = data.drop(drop_columns, axis=1)
```

```
# Convert TX_DATETIME to datetime type
data["TX_DATETIME"] = pd.to_datetime(data["TX_DATETIME"])
data.sort_values(by=["TERMINAL_ID", "TX_DATETIME"], inplace=True)
```

```
# Feature engineering for terminals to get info without overfitting
data["TERMINAL_TOTAL_TRANSACTIONS_30D"] = (
    data.groupby("TERMINAL_ID")["TX_DATETIME"]
    .rolling(window=30, min_periods=1)
    .count()
    .reset_index(level=0, drop=True)
)

data["TERMINAL_FRAUD_COUNT_30D"] = (
    data.groupby("TERMINAL_ID")["TX_FRAUD"]
    .rolling(window=30, min_periods=1)
    .sum()
    .reset_index(level=0, drop=True)
)

data["TERMINAL_FRAUD_RATE_30D"] = (
    data["TERMINAL_FRAUD_COUNT_30D"] / data["TERMINAL_TOTAL_TRANSACTIONS_30D"]
).fillna(0)
```

```
# Feature engineering for customers
data["CUSTOMER_TOTAL_TRANSACTIONS_30D"] = (
    data.groupby("CUSTOMER_ID")["TX_DATETIME"]
    .rolling(window=30, min_periods=1)
    .count()
    .reset_index(level=0, drop=True)
)

data["CUSTOMER_FRAUD_COUNT_30D"] = (
    data.groupby("CUSTOMER_ID")["TX_FRAUD"]
    .rolling(window=30, min_periods=1)
    .sum()
    .reset_index(level=0, drop=True)
)

data["CUSTOMER_FRAUD_RATE_30D"] = (
    data["CUSTOMER_FRAUD_COUNT_30D"] / data["CUSTOMER_TOTAL_TRANSACTIONS_30D"]
).fillna(0)

# The number of terminals the customer spent (different from all available terminals to the customer)
data["Num Terminals"] = data.groupby("CUSTOMER_ID")["TERMINAL_ID"].transform("nunique")

# Drop original categorical columns
data = data.drop(["TERMINAL_ID", "CUSTOMER_ID", "TX_DATETIME" ], axis=1)


# Normalize the data
scaler = MinMaxScaler()
columns_to_normalize = data.columns.difference(["TX_FRAUD"])

# Apply MinMaxScaler
data[columns_to_normalize] = scaler.fit_transform(data[columns_to_normalize])


correlation_matrix = data.corr()

plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center=0)
plt.title('Correlation Matrix')
plt.show()


data.head()


# Split the data into train and test sets
train_data, test_data = train_test_split(data, test_size=0.2, random_state=42)


# Split into feature and label values
X_test = test_data[test_data.columns.difference(["TX_FRAUD"])]
y_test= test_data["TX_FRAUD"]
X_train = train_data[train_data.columns.difference(["TX_FRAUD"])]
y_train= train_data["TX_FRAUD"]
```

SVM

Use bayesian optimization to pick good hyperparameters to run SVM with

```
!pip install scikit-optimize
```

```
Requirement already satisfied: scikit-optimize in /opt/conda/lib/python3.10/site-packages (0.9.0)
Requirement already satisfied: joblib>=0.11 in /opt/conda/lib/python3.10/site-packages (from scikit-optimize) (1.2.0)
Requirement already satisfied: pyaml>=16.9 in /opt/conda/lib/python3.10/site-packages (from scikit-optimize) (23.7.0)
Requirement already satisfied: numpy>=1.13.3 in /opt/conda/lib/python3.10/site-packages (from scikit-optimize) (1.23.5)
Requirement already satisfied: scipy>=0.19.1 in /opt/conda/lib/python3.10/site-packages (from scikit-optimize) (1.11.1)
Requirement already satisfied: scikit-learn>=0.20.0 in /opt/conda/lib/python3.10/site-packages (from scikit-optimize) (1.2.2)
Requirement already satisfied: PyYAML in /opt/conda/lib/python3.10/site-packages (from pyaml>=16.9->scikit-optimize) (6.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /opt/conda/lib/python3.10/site-packages (from scikit-learn>=0.20.0->scikit-optimi
```

```
from sklearn.svm import LinearSVC
```

```
param_space = {'C': (0.01, 1000.0, 'log-uniform')}

svm_model = LinearSVC()

# Initialize BayesSearchCV
bayes_search = BayesSearchCV(svm_model, param_space, cv=5, n_iter=10, n_jobs=-1, verbose=2, scoring='accuracy')
bayes_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 1 candidates, totalling 5 fits
/opt/conda/lib/python3.10/site-packages/scipy/__init__.py:146: UserWarning: A NumPy vers
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
/opt/conda/lib/python3.10/site-packages/scipy/__init__.py:146: UserWarning: A NumPy vers
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
[CV] END ................................C=2.001669345611118; total time= 8.5min
[CV] END ................................C=2.001669345611118; total time= 8.6min
[CV] END ................................C=2.001669345611118; total time= 8.6min
[CV] END ................................C=2.001669345611118; total time= 8.7min
[CV] END ................................C=2.001669345611118; total time= 6.8min
Fitting 5 folds for each of 1 candidates, totalling 5 fits
/opt/conda/lib/python3.10/site-packages/scipy/__init__.py:146: UserWarning: A NumPy vers
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
[CV] END ..............................C=0.20376397486142794; total time= 1.1min
[CV] END ..............................C=0.20376397486142794; total time= 1.2min
[CV] END ..............................C=0.20376397486142794; total time= 1.1min
[CV] END ..............................C=0.20376397486142794; total time= 1.1min
[CV] END ..............................C=0.20376397486142794; total time=  53.8s
Fitting 5 folds for each of 1 candidates, totalling 5 fits
[CV] END ...............................C=0.7257501589124487; total time= 3.4min
[CV] END ...............................C=0.7257501589124487; total time= 3.5min
[CV] END ...............................C=0.7257501589124487; total time= 3.4min
[CV] END ...............................C=0.7257501589124487; total time= 3.4min
[CV] END ...............................C=0.7257501589124487; total time= 2.7min
Fitting 5 folds for each of 1 candidates, totalling 5 fits
[CV] END ..............................C=0.07536539639538105; total time=  34.9s
[CV] END ..............................C=0.07536539639538105; total time=  35.0s
[CV] END ..............................C=0.07536539639538105; total time=  35.1s
[CV] END ..............................C=0.07536539639538105; total time=  35.1s
[CV] END ..............................C=0.07536539639538105; total time=  25.4s
Fitting 5 folds for each of 1 candidates, totalling 5 fits
/opt/conda/lib/python3.10/site-packages/sklearn/svm/_base.py:1244: ConvergenceWarning: L
  warnings.warn(
/opt/conda/lib/python3.10/site-packages/sklearn/svm/_base.py:1244: ConvergenceWarning: L
  warnings.warn(
[CV] END ................................C=5.263840765894765; total time=17.5min
[CV] END ................................C=5.263840765894765; total time=17.5min
/opt/conda/lib/python3.10/site-packages/sklearn/svm/_base.py:1244: ConvergenceWarning: L
  warnings.warn(
[CV] END ................................C=5.263840765894765; total time=18.5min
/opt/conda/lib/python3.10/site-packages/sklearn/svm/_base.py:1244: ConvergenceWarning: L
  warnings.warn(
[CV] END ................................C=5.263840765894765; total time=18.6min
/opt/conda/lib/python3.10/site-packages/sklearn/svm/_base.py:1244: ConvergenceWarning: L
  warnings.warn(
[CV] END ................................C=5.263840765894765; total time=14.5min
Fitting 5 folds for each of 1 candidates, totalling 5 fits
/opt/conda/lib/python3.10/site-packages/scipy/__init__.py:146: UserWarning: A NumPy vers
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
[CV] END ..............................C=0.15225114371971116; total time=  53.6s
[CV] END ..............................C=0.15225114371971116; total time=  53.4s
[CV] END ..............................C=0.15225114371971116; total time=  53.5s
[CV] END ..............................C=0.15225114371971116; total time=  54.4s
[CV] END ..............................C=0.15225114371971116; total time=  42.3s
Fitting 5 folds for each of 1 candidates, totalling 5 fits
[CV] END ................................C=2.363436172820095; total time=10.9min
[CV] END ................................C=2.363436172820095; total time=11.0min
[CV] END ................................C=2.363436172820095; total time=10.4min
[CV] END ................................C=2.363436172820095; total time=12.8min
[CV] END ................................C=2.363436172820095; total time= 8.9min
Fitting 5 folds for each of 1 candidates, totalling 5 fits
/opt/conda/lib/python3.10/site-packages/scipy/__init__.py:146: UserWarning: A NumPy vers
```

```python
print("Best: %f using %s" % (bayes_search.best_score_, bayes_search.best_params_))
means = bayes_search.cv_results_['mean_test_score']
stds = bayes_search.cv_results_['std_test_score']
params = bayes_search.cv_results_['params']
df_svc = pd.DataFrame(columns=['C', 'accuracy'])
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
    df_svc.loc[len(df_svc)]= ["{:.3f}".format(param.get('C')),mean]
```

```
Best: 0.862106 using OrderedDict([('C', 43.666896678358455)])
0.861972 (0.000376) with: OrderedDict([('C', 2.001669345611118)])
0.862025 (0.000410) with: OrderedDict([('C', 0.20376397486142794)])
0.861985 (0.000376) with: OrderedDict([('C', 0.7257501589124487)])
0.862022 (0.000373) with: OrderedDict([('C', 0.07536539639538105)])
0.861968 (0.000386) with: OrderedDict([('C', 5.263840765894765)])
0.862024 (0.000396) with: OrderedDict([('C', 0.15225114371971116)])
0.861966 (0.000382) with: OrderedDict([('C', 2.363436172820095)])
0.862006 (0.000365) with: OrderedDict([('C', 0.053495401191491085)])
0.792668 (0.064570) with: OrderedDict([('C', 65.5122479999367)])
0.862106 (0.001665) with: OrderedDict([('C', 43.666896678358455)])
```

Running SVM on entire dataset

```
/opt/conda/lib/python3.10/site-packages/sklearn/svm/_base.py:1244: ConvergenceWarning: L
```

```python
svm_final_model = LinearSVC(C=0.15)
```

```
  warnings.warn(
```

```python
svm_final_model.fit(X_train, y_train)
```

```
▾ LinearSVC
LinearSVC(C=0.15)
```

```
/opt/conda/lib/python3.10/site-packages/sklearn/svm/_base.py:1244: ConvergenceWarning: L
```

```python
# Make predicition of training and testing data
train_pred = svm_final_model.predict(X_train)
test_pred = svm_final_model.predict(X_test)

# Calculate accuracy for training and testing
train_acc = accuracy_score(y_train, train_pred)
test_acc = accuracy_score(y_test, test_pred)

print("Train Accuracy: ",train_acc)
print("Test Accuracy: ",test_acc)
```

```
Train Accuracy:  0.8620302542612106
Test Accuracy:  0.8612463745843877
```

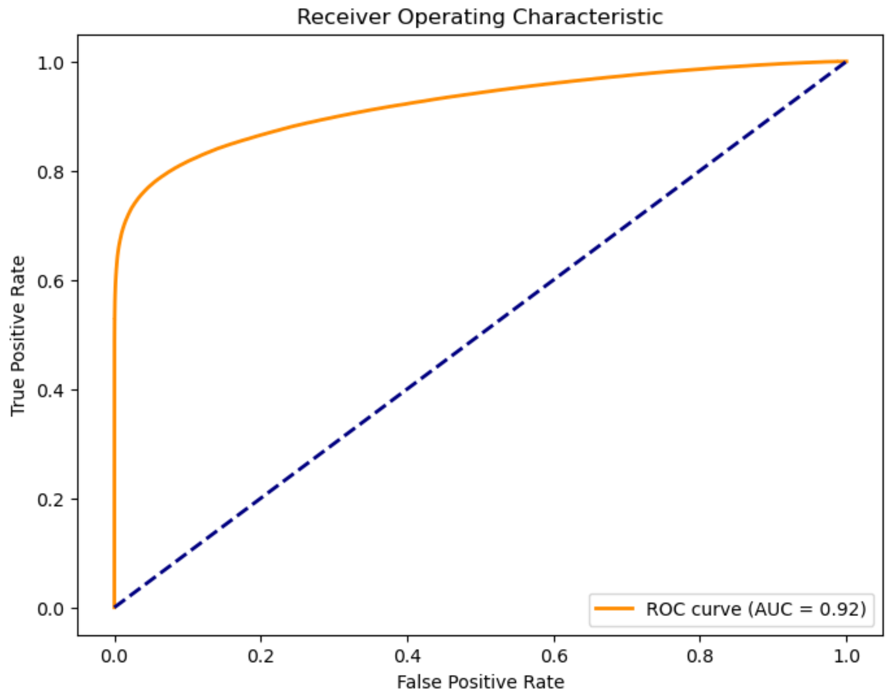
```python
print("\nClassification Report:")
print(classification_report(y_test, test_pred))
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.82      0.92      0.87    383551
           1       0.91      0.80      0.85    384292

    accuracy                           0.86    767843
   macro avg       0.87      0.86      0.86    767843
weighted avg       0.87      0.86      0.86    767843
```

Plotting ROC-AUC curve

```python
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
```

```python
# Plotting roc-auc curve
y_prob = svm_final_model.decision_function(X_test)

fpr, tpr, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```


Receiver Operating Characteristic
True Positive Rate
False Positive Rate
ROC curve (AUC = 0.92)
0.0
0.2
0.4
0.6
0.8
1.0

In [32]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_rep
from sklearn.model_selection import GridSearchCV
import matplotlib.pyplot as plt
```

In [34]:
```
# Using scikit-learn's DecisionTreeClassifier, train a supervised learning model

# Create a DecisionTreeClassifier instance
dtc = DecisionTreeClassifier(random_state=2)

# Define the hyperparameters and their possible values
param_grid = {
    'max_depth': [None, 50],
    'min_samples_split': [2, 3],
    'max_features': [None, 'sqrt', 'log2']
}

# Perform Grid Search Cross Validation
grid_search = GridSearchCV(dtc, param_grid, cv=7, scoring='accuracy')
grid_search.fit(X_train, y_train)
```

Out[34]:
```
GridSearchCV(cv=7, estimator=DecisionTreeClassifier(random_state=2),
             param_grid={'max_depth': [None, 50],
                         'max_features': [None, 'sqrt', 'log2'],
                         'min_samples_split': [2, 3]},
             scoring='accuracy')
```

In [9]:
```
# Get the best parameters and best model
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_

print("Best Parameters:", best_params)
print("Best Model:", best_model)

# Fit the best model on the training data
best_model.fit(X_train, y_train)

# Make predictions
y_pred = best_model.predict(X_test)
```

```
Best Parameters: {'max_depth': 50, 'max_features': None, 'min_samples_split': 3}
Best Model: DecisionTreeClassifier(max_depth=50, min_samples_split=3, random_sta
te=2)
```

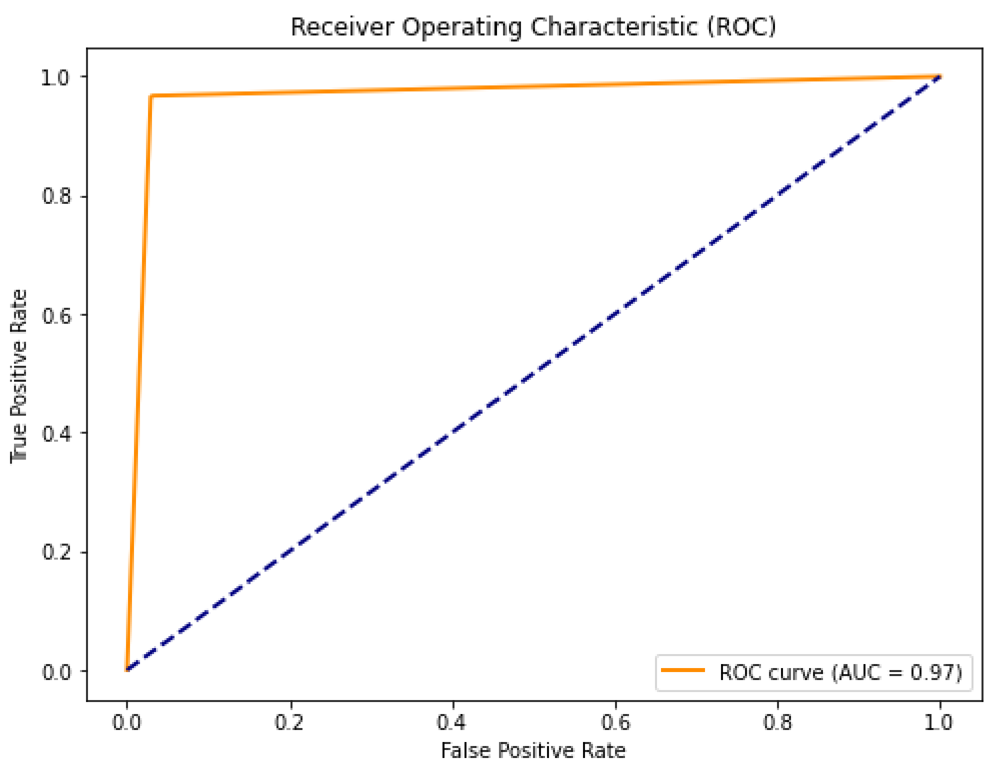
In [10]:
```
# Evaluate the model's performance using accuracy, confusion matrix, and classif
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
report = classification_report(y_test, y_pred)

print("Accuracy:", accuracy)
print("Confusion Matrix:\n", confusion)
print("Classification Report:\n", report)
```

```
Accuracy: 0.968212251723334
Confusion Matrix:
```

```
[[372283  11268]
 [ 13140 371152]]
Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.97      0.97    383551
           1       0.97      0.97      0.97    384292

    accuracy                           0.97    767843
   macro avg       0.97      0.97      0.97    767843
weighted avg       0.97      0.97      0.97    767843
```

In [12]:

```
# Predict probabilities for the positive class
y_prob = best_model.predict_proba(X_test)[:, 1]

# Calculate ROC-AUC score
roc_auc = roc_auc_score(y_test, y_prob)

# Calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_prob)

# Plot ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC = {:.2f})'.fo
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC)')
plt.legend(loc="lower right")
plt.show()
```



In [20]:

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

```
f1 = f1_score(y_test, y_pred)
roc_auc = roc_auc_score(y_test, y_prob)

print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)
print("ROC-AUC:", roc_auc)
```

```
Precision: 0.9705350138591078
Recall: 0.9658072507364192
F1-score: 0.9681653606569349
ROC-AUC: 0.96896281828251
```

Logistic Regression

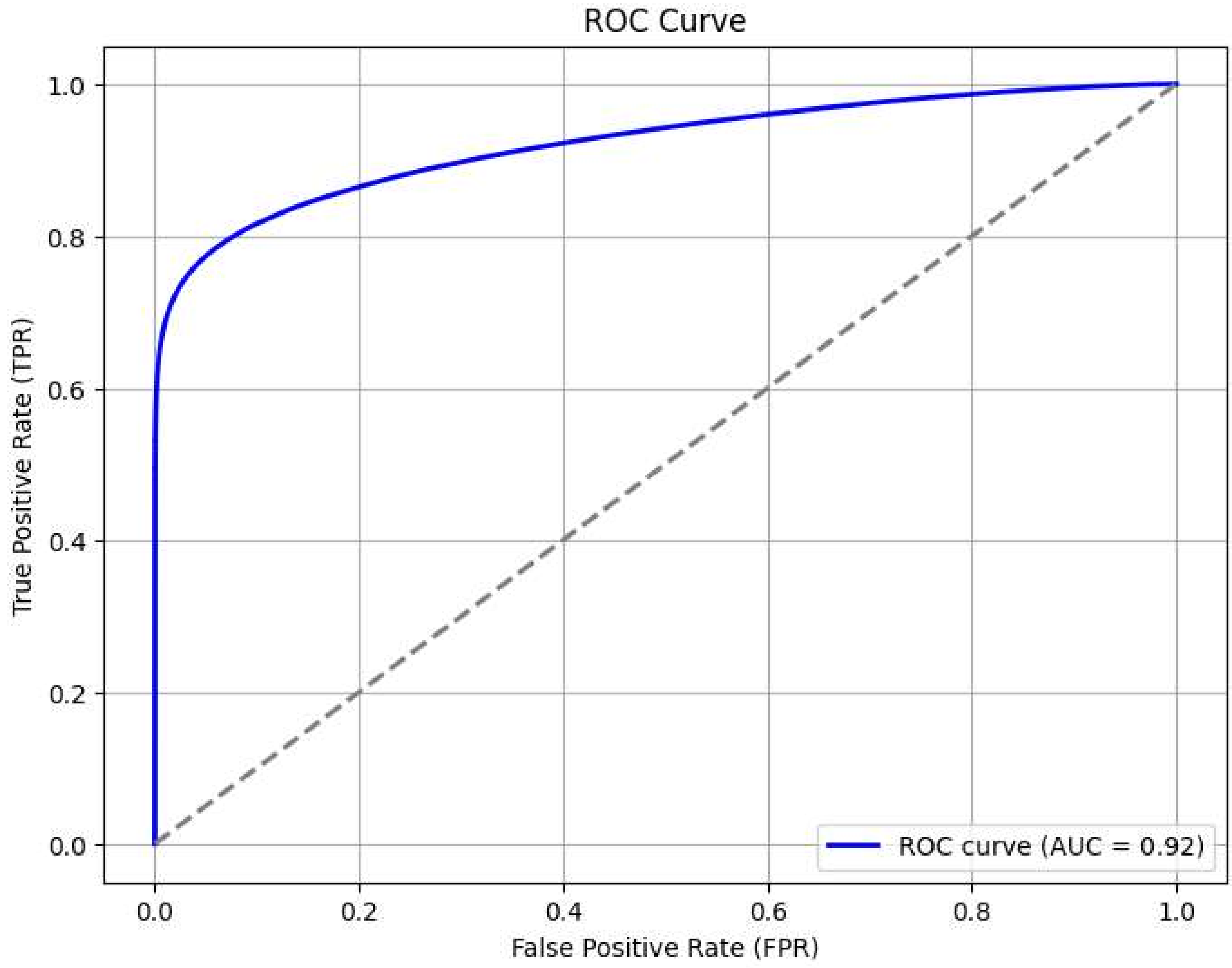
In [20]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score,confusion_matrix, classification_report, roc_curve, roc_auc_score
```

In [24]:
```
# Create an instance of the Logistic Regression model
logistic_regression = LogisticRegression(max_iter=1000)

#Fit the model on the training data
logistic_regression.fit(X_train, y_train)

#Predict the test data using the trained model
y_pred = logistic_regression.predict(X_test)
```

In [25]:
```
#Evaluate the model's performance
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:\n", accuracy)
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)
report = classification_report(y_test, y_pred)
print("Classification Report:\n", report)
```

```
Accuracy:
 0.8598958380814828
Confusion Matrix:
 [[349520  34031]
 [ 73547 310745]]
Classification Report:
               precision    recall  f1-score   support

           0       0.83      0.91      0.87    383551
           1       0.90      0.81      0.85    384292

    accuracy                           0.86    767843
   macro avg       0.86      0.86      0.86    767843
weighted avg       0.86      0.86      0.86    767843
```

In [26]:
```
# Probability predictions for class 1 (fraud)
y_pred_prob = logistic_regression.predict_proba(X_test)[:, 1]

# Calculate the ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

# Calculate the ROC-AUC score
roc_auc = roc_auc_score(y_test, y_pred_prob)

# Plot the ROC-AUC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--', lw=2)
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('ROC Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```


ROC Curve
True Positive Rate (TPR)
False Positive Rate (FPR)
ROC curve (AUC = 0.92)
0.0
0.2
0.4
0.6
0.8
1.0

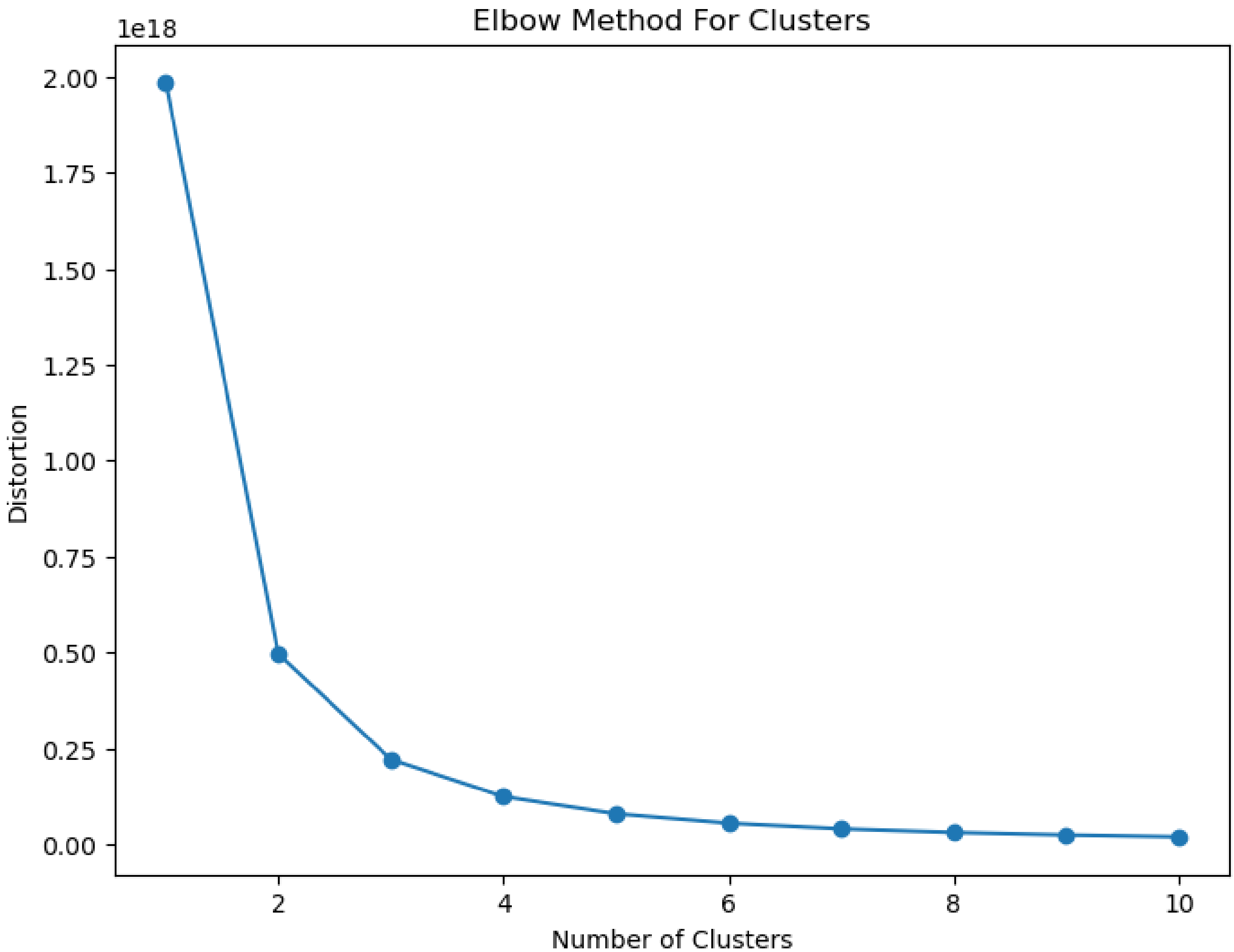
K-Means

In [34]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import recall_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score
from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
from sklearn.model_selection import train_test_split
from sklearn.metrics import davies_bouldin_score
import warnings
warnings.filterwarnings("ignore")
```

In [35]:
```
data = pd.read_csv("transactions_balanced.csv")
```

In [40]:
```
# Separate features and target
X = data[data.columns.difference(["TX_FRAUD"])]
y = data["TX_FRAUD"]

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42

df_final = pd.DataFrame(columns=['model', 'data_set', 'accuracy'])
```

In [41]:
```
distortions = []
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k, random_state=0,algorithm="elkan")
    kmeans.fit(X_train)
    distortions.append(kmeans.inertia_)

plt.figure(figsize=(8, 6))
plt.plot(range(1, 11), distortions, marker='o')
plt.title('Elbow Method For Clusters')
plt.xlabel('Number of Clusters')
plt.ylabel('Distortion')
plt.show()
```



In [29]:
```
kmeans = KMeans(n_clusters=2, init='k-means++', random_state=0,algorithm="elkan").fit(X_
kmeans_labels = kmeans.labels_

y_pred = kmeans.predict(X_test)
```

In [30]:
```
for i in range(len(y_pred)):
    if y_pred[i] == 0:
        y_pred[i] = 1
    else:
        y_pred[i] = 0
```

In [31]:
```
# Summary of the predictions made by the classifier
print('Classification Report : \n',classification_report(y_test, y_pred))
print('Confusion Matrix : \n',confusion_matrix(y_test, y_pred))

# Accuracy score
print('Accuracy : ',accuracy_score(y_pred,y_test))

# F-1 score
print('F-1 score : ',f1_score(y_pred,y_test))

# Recall score
print('Recall score : ',recall_score(y_test, y_pred))

# Davies Bouldin score
print('Davies Bouldin Score : ',davies_bouldin_score(X_train, kmeans_labels))
```

```
Classification Report :
               precision    recall  f1-score   support
```

```
           0       0.99      0.50      0.67    223430
           1       0.01      0.63      0.02      1520

    accuracy                           0.50    224950
   macro avg       0.50      0.56      0.34    224950
weighted avg       0.99      0.50      0.66    224950

Confusion Matrix :
 [[111579 111851]
 [   564    956]]
Accuracy :  0.5002667259390976
F-1 score :  0.01672395847000271
Recall score :  0.6289473684210526
Davies Bouldin Score :  0.5000290444611917
```

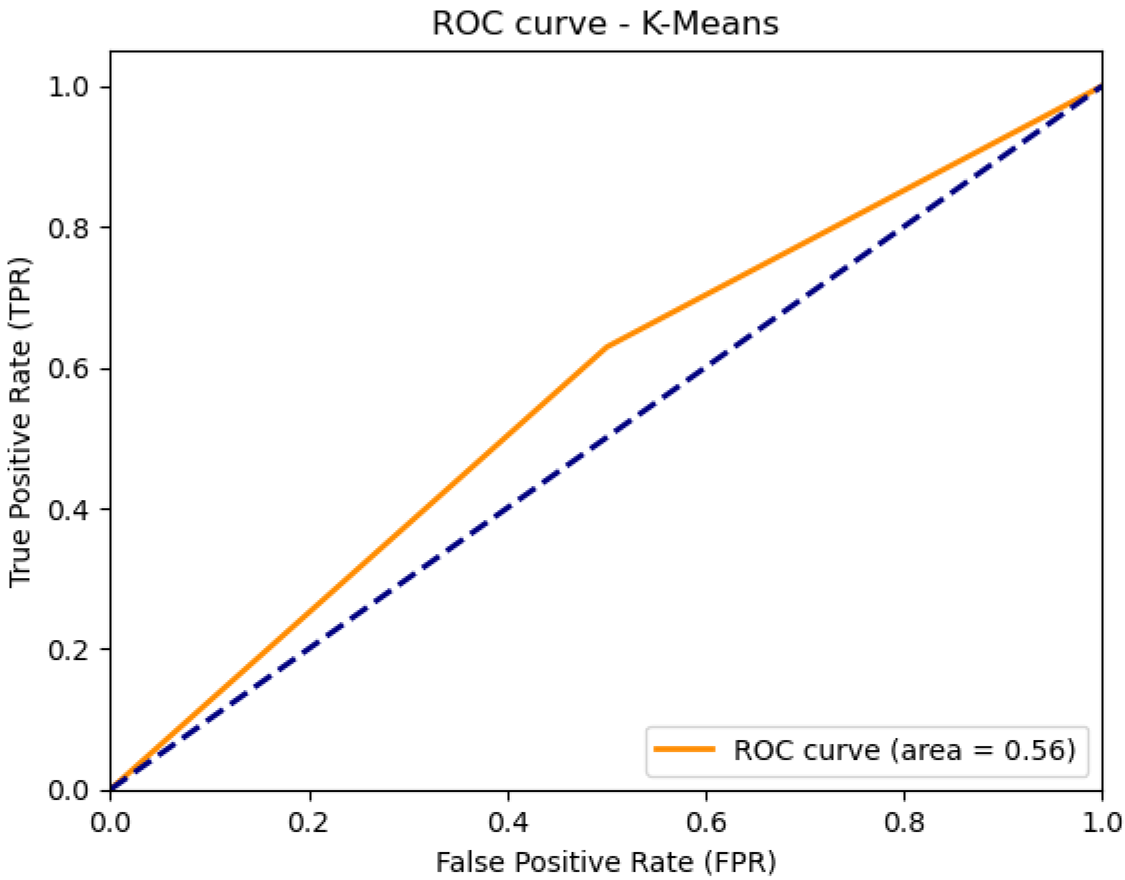
In [32]:

```
fpr, tpr, thresholds = roc_curve(y_test, y_pred)
roc_auc = auc(fpr, tpr)

print(fpr)
print(tpr)
print(thresholds)
print(roc_auc)
```

```
[0.         0.50060869 1.        ]
[0.         0.62894737 1.        ]
[2 1 0]
0.5641693383303849
```

In [33]:

```
plt.figure()
lw = 2
plt.plot(fpr, tpr, color='darkorange',
         lw=lw, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('ROC curve - K-Means')
plt.legend(loc="lower right")
plt.show()
```



ROC curve - K-Means
True Positive Rate (TPR)
False Positive Rate (FPR)
ROC curve (area = 0.56)
0.0
0.2
0.4
0.6
0.8
1.0

Split the dataset into training sets and test sets

In [116]:

```
# Separate features and target
X = data[data.columns.difference(["TX_FRAUD"])]
y = data["TX_FRAUD"]

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
```

Train a Random Forest model with certain parameters

In [118]:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import accuracy_score

# Create and train the Random Forest classifier
rf_model = RandomForestClassifier(n_estimators = 20, max_depth = 10, n_jobs = -1, min_sar
 min_samples_leaf = 1)
rf_model.fit(X_train, y_train)
```

Out[118]:

```
RandomForestClassifier(max_depth=10, n_estimators=20, n_jobs=-1)
```

Report Feature Importance

In [119]:

```
# Get the feature importances
feature_importances = rf_model.feature_importances_

print("Feature Importances:", feature_importances)
```

```
Feature Importances: [0.07181535 0.09467252 0.00101415 0.02385727 0.02943329 0.
03453648
 0.02276158 0.5386301  0.01605005 0.02058315 0.00401333 0.01142164
 0.01969532 0.06800228 0.02476648 0.00743814 0.01130887]
```

Evaluation of performance

In [120]:

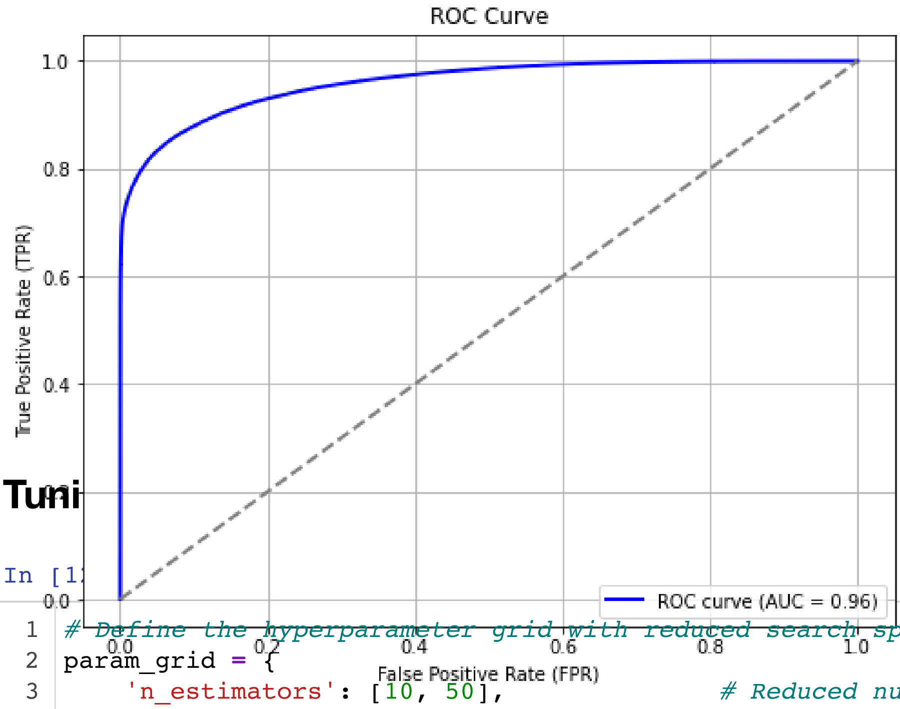
```
# Make predictions on the test set
y_pred = rf_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")

# Get the train and test scores
train_score = rf_model.score(X_train, y_train)
test_score = rf_model.score(X_test, y_test)

print("Train Score:", train_score)
print("Test Score:", test_score)
```

```
Accuracy: 0.8862814403465292
Train Score: 0.8869595284708546
Test Score: 0.8862814403465292
```

In [123]:

```
metrics import precision_score, recall_score, f1_score, confusion_matrix, classification_repo
tlib.pyplot as plt

recision, recall, and F1 score
recision_score(y_test, y_pred)
ll_score(y_test, y_pred)
(y_test, y_pred)

OC-AUC score
odel.predict_proba(X_test)[:, 1]

atrix
n_matrix(y_test, y_pred)
ion Matrix:\n", cm)

ion Report
sification_report(y_test, y_pred)
fication Report:\n", report)

_auc_score(y_test, y_prob)
ion:", precision)
:", recall)
re:", f1)
C Score:", roc_auc)

he ROC curve
esholds = roc_curve(y_test, y_prob)

C-AUC curve
gsize=(8, 6))
 tpr, color='blue', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
1], [0, 1], color='gray', linestyle='--', lw=2)
alse Positive Rate (FPR)')
rue Positive Rate (TPR)')
C Curve')
c='lower right')
)
```

```
Confusion Matrix:
 [[371202  12349]
 [ 74969 309323]]
Classification Report:
               precision    recall  f1-score   support

           0       0.83      0.97      0.89    383551
           1       0.96      0.80      0.88    384292

    accuracy                           0.89    767843
   macro avg       0.90      0.89      0.89    767843
weighted avg       0.90      0.89      0.89    767843

Precision: 0.9616099629436196
Recall: 0.8049165738552976
F1 Score: 0.8763138063697299
ROC-AUC Score: 0.9603445260527668
```



Tuni

In [1

```
# Define the hyperparameter grid with reduced search space
param_grid = {
    'n_estimators': [10, 50],              # Reduced number of estimators
    'max_depth': [None, 10],                      # Limited range for max_depth
    'min_samples_split': [2, 5],                   # Smaller range for min_samples_sp
    'min_samples_leaf': [1, 2]                     # Smaller range for min_samples_le
}

# Create the Random Forest classifier
rf_classifier = RandomForestClassifier()

# Use RandomizedSearchCV for more efficient search
random_search = RandomizedSearchCV(estimator=rf_classifier, param_distributions=param_gr
random_search.fit(X_train, y_train)

# Get the best parameters and best model
best_params = random_search.best_params_
best_model = random_search.best_estimator_
```

In [129]:

```
best_params
```

Out[129]:

```
{'n_estimators': 10,
 'min_samples_split': 2,
 'min_samples_leaf': 1,
 'max_depth': None}
```

Report Feature Importance

In [130]:

```
# Get the feature importances
feature_importances = best_model.feature_importances_

print("Feature Importances:", feature_importances)
```

```
Feature Importances: [0.05324838 0.0957005  0.0024873  0.07138631 0.01908595 0.
02149713
 0.01540818 0.33409654 0.02626065 0.04390266 0.03329492 0.04670616
 0.04214659 0.09374525 0.04510693 0.02766542 0.02826114]
```

Evaluation of Performance

In [131]:

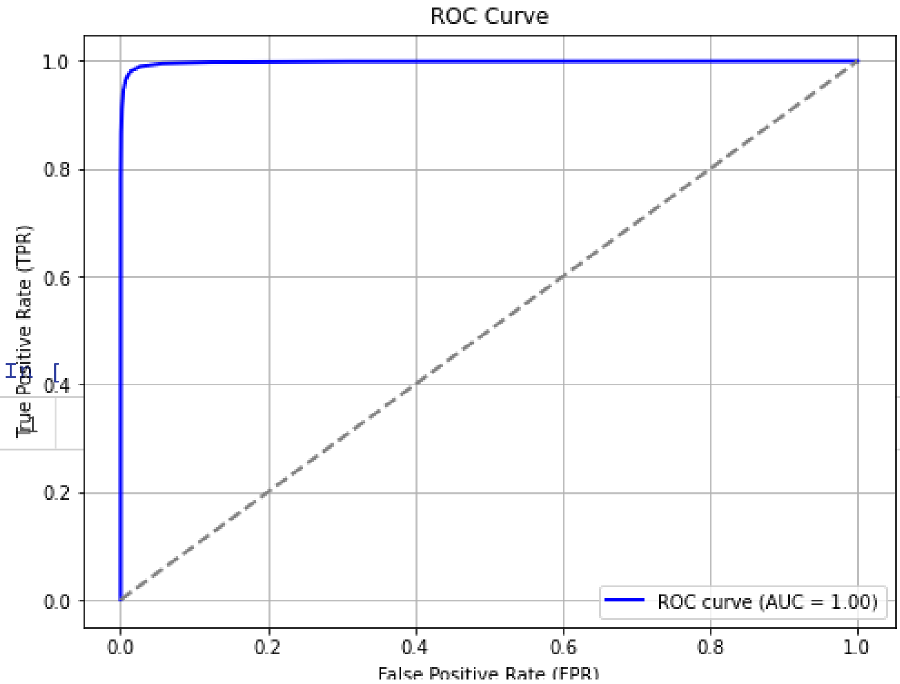
```
# Make predictions on the test set
y_pred = best_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")

# Get the train and test scores
train_score = best_model.score(X_train, y_train)
test_score = best_model.score(X_test, y_test)

print("Train Score:", train_score)
print("Test Score:", test_score)
```

```
Accuracy: 0.9798917226568452
Train Score: 0.9991808213210461
Test Score: 0.9798917226568452
```

In [132]:

```python
# Calculate precision, recall, and F1 score
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

# Calculate ROC-AUC score
y_prob = best_model.predict_proba(X_test)[:, 1]

# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)

# Classification Report
report = classification_report(y_test, y_pred)
print("Classification Report:\n", report)

roc_auc = roc_auc_score(y_test, y_prob)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
print("ROC-AUC Score:", roc_auc)

# Calculate the ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_prob)

# Plot the ROC-AUC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--', lw=2)
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('ROC Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```

```
Confusion Matrix:
 [[380844   2707]
 [ 12733 371559]]
Classification Report:
               precision    recall  f1-score   support

           0       0.97      0.99      0.98    383551
           1       0.99      0.97      0.98    384292

    accuracy                           0.98    767843
   macro avg       0.98      0.98      0.98    767843
weighted avg       0.98      0.98      0.98    767843

Precision: 0.9927671762863846
Recall: 0.9668663412196975
F1 Score: 0.9796455907129052
ROC-AUC Score: 0.9980771196410041
```


ROC Curve
True Positive Rate (TPR)
False Positive Rate (FPR)
ROC curve (AUC = 1.00)
0.0
0.2
0.4
0.6
0.8
1.0