```
!pip install sdv --quiet
!pip install ctgan --quiet
!pip install pyspark==3.3.2
!pip install boto3
!pip install scikit-learn
!pip install lightgbm
!pip install xgboost
!pip install matplotlib plotly seaborn pandas
```

```
import os
import pandas as pd
import numpy as np
from pyspark.sql import SparkSession
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import RandomForestClassifier
import lightgbm as lgb
import matplotlib.pyplot as plt
import plotly.express as px
from ctgan import CTGAN
from google.colab import drive
```

🚀 Fraud Detection Analysis

Author: Aryan Kaushik

- https://github.com/aryankaushik89

📚 Table of Contents

📊 1. Project Overview

This notebook demonstrates an end-to-end approach to financial fraud detection using a combination of advanced feature engineering, machine learning, and synthetic data generation. The workflow is built for transparency and reproducibility: every step, from data sourcing to model interpretation, is included as code and can be rerun from scratch. All analysis is designed to reflect both real-world fraud data challenges and the expectations of modern data science teams.

Data Sources

The foundational data for this project comes from the [PaySim Synthetic Mobile Money Transactions Dataset], which mimics one month of real-world mobile money transactions as found in emerging financial markets. PaySim's logs include over 6 million records, with a variety of transaction types, customer behaviors, and injected fraudulent activity. The use of such realistic, privacy-safe data makes it ideal for developing and benchmarking fraud analytics workflows.

Augmenting the Data with GANs

In addition to the original Kaggle data, this project demonstrates synthetic data generation using Generative Adversarial Networks (GANs). To further enrich the dataset, I generated additional synthetic data using a GANs, specifically CTGAN. GANs are powerful machine learning models that can learn the underlying distribution of a dataset and generate new, highly realistic records.

They are a family of deep learning models that can learn the statistical properties of a real dataset and create entirely new, artificial records that are nearly indistinguishable from real data. This technique is valuable for fraud detection projects as it:

- GANs allow us to create much larger and more varied datasets for robust ML experimentation, especially useful in rare-event scenarios like fraud.
- We can inject or simulate new types of risk patterns and attributes, such as device, geo, behavioral, and flag-based features, as is common in production fraud systems.
- Helps reduce class imbalance.

Final Dataset

In this notebook, GANs are used (via the CTGAN implementation) to create an augmented, feature-rich synthetic dataset, `synthetic_fraud_200k.csv`, containing 200,000 synthetic transaction records. The process adds additional attributes such as device type, location, VPN usage, and risk flags, which are common in real-world anti-fraud systems.

This demonstrates not only core modeling skills, but also the ability to generate realistic testbeds for algorithm development—a key asset in environments where data access is sensitive or limited.

Data Ingestion and Synthetic Data Generation

- Ingesting the original PaySim CSV (after download from Kaggle)
- Creating engineered features
- Training a GAN (CTGAN) on a stratified sample
- Generating and saving the synthetic dataset for modeling

```
from google.colab import drive

drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

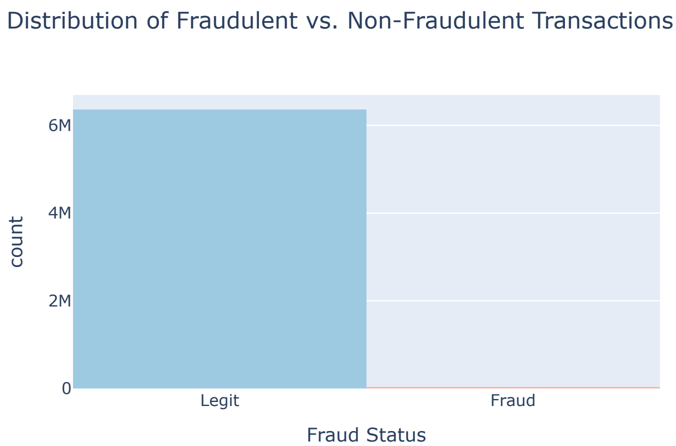
```
# --- Step 1: Ingest original PaySim data from Kaggle ---
csv_path = '/content/drive/MyDrive/Colab Notebooks/paysim_creditcard_data.csv'
df_paysim = pd.read_csv(csv_path)
print("Loaded creditcard.csv with shape:", df_paysim.shape)
df_paysim.head()
```

```
Loaded creditcard.csv with shape: (6362620, 11)
```

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

```
import plotly.express as px
```

```
PASTEL_BLUE = "#9ecae1"
PASTEL_PINK = "#fbb4ae"

#--- Plot the class balance: fraud vs. legitimate
fig = px.histogram(
    df_paysim, x="isFraud",
    color="isFraud",
    color_discrete_sequence=[PASTEL_BLUE, PASTEL_PINK],
    title="Distribution of Fraudulent vs. Non-Fraudulent Transactions",
    labels={"isFraud": "Fraud Status"},
    width=600, height=400
)
fig.update_xaxes(tickvals=[0,1], ticktext=["Legit", "Fraud"])
fig.update_layout(showlegend=False)
fig.show()
```



Insight:

- Fraudulent transactions are only a tiny fraction of the total. This reflects the real-world fraud data challenges where fraud rates are often under 1%. We will need to apply techniques like resampling and class weighting in later steps to balance out the data before modelling.

- Downsampling here to reduce the data size and stratified sampling make sure we keep the same fraud-to-non_fraud ratio as the original data, keeping class balance so that the GAN learns realistic patterns from both classes.

```
# --- Step 2: Stratified downsampling for GAN training ---
seed_sample = (
    df_paysim
    .groupby('isFraud', group_keys=False)
    .apply(lambda sub: sub.sample(min(len(sub), 5000), random_state=42))
    .reset_index(drop=True)
)
```

Show hidden output

- This code below is to augument the data further by adding engineered binary features to each record to simulate things like geography, device type, network risk (VPN, ISP), account behavior etc.
- This gives us a richer dataset to work with with added signals that arecommonly seen in real fraud data.

```
# --- Step 3: Augment with 15+ Engineered Binary Features ---
import numpy as np

def add_extra_features(df_in):
    np.random.seed(42)
    df = df_in.copy()
    n = len(df)
```

```
    # Geographic indicators
    df['loc_US']      = np.random.binomial(1, 0.7, n)
    df['loc_EU']      = np.random.binomial(1, 0.15, n)
    df['loc_APAC']    = np.random.binomial(1, 0.1, n)
    df['loc_OTHER']   = 1 - (df['loc_US'] + df['loc_EU'] + df['loc_APAC'])

    # Device type
    df['dev_mobile']  = np.random.binomial(1, 0.65, n)
    df['dev_desktop'] = 1 - df['dev_mobile']

    # Network flags
    df['isp_known']   = np.random.binomial(1, 0.85, n)
    df['vpn_used']    = np.random.binomial(1, 0.05 + 0.5 * df['isFraud'], n)

    # Account age / behavior
    df['new_account']   = np.random.binomial(1, 0.1 + 0.4 * df['isFraud'], n)
    df['multi_account'] = np.random.binomial(1, 0.05 + 0.3 * df['isFraud'], n)

    # Repeat transactions
    origin_counts = df['nameOrig'].map(df['nameOrig'].value_counts())
    df['repeat_txn']    = (origin_counts > 1).astype(int)

    # Time-based features
    df['hour']          = df['step'] % 24
    df['night_txn']     = df['hour'].between(0,6).astype(int)
    df['weekend_txn']   = ((df['step'] // 24) % 7).isin([5,6]).astype(int)

    # Fraud-flag patterns
    df['flag_manual_review'] = np.random.binomial(1, 0.02 + 0.6 * df['isFlaggedFraud'], n)
    df['email_link_clicked'] = np.random.binomial(1, 0.03 + 0.5 * df['isFlaggedFraud'], n)
    df['two_factor_used']    = np.random.binomial(1, 0.2 - 0.1 * df['isFraud'], n)
    df['suspicious_ip']      = np.random.binomial(1, 0.01 + 0.7 * df['isFraud'], n)
    df['browser_tor']        = np.random.binomial(1, 0.005 + 0.8 * df['isFraud'], n)

    return df

aug_seed = add_extra_features(seed_sample)
print("Augmented seed shape:", aug_seed.shape)
aug_seed.head()
```

Augmented seed shape: (10000, 30)

	step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig	nameDest	oldbalanceDest	newbalanceDest	isFraud
0	162	CASH_OUT	183806.32	C691771226	19391.00	0.00	C1416312719	382572.19	566378.51	0
1	137	PAYMENT	521.37	C203378011	0.00	0.00	M42773300	0.00	0.00	0
2	179	PAYMENT	3478.18	C1698571270	19853.00	16374.82	M643984524	0.00	0.00	0
3	355	PAYMENT	1716.05	C913764937	5769.17	4053.13	M1387429131	0.00	0.00	0
4	354	CASH_IN	253129.93	C2017736577	1328499.49	1581629.42	C407484102	2713220.48	2460090.55	0

5 rows × 30 columns

- This section below preps and trains a CTGAN model on our engineered fraud dataset.
- It first specifies which columns are continuous versus discrete (including all engineered risk and device flags), builds the training DataFrame, and converts all continuous columns to numeric just in case.
- The CTGAN is then trained to learn the structure of the data is used to generate 200K synthetic records.
- New records are saved as a CSV for use later in the EDA, feature engineering, and modeling.

```
# --- Step 4: Train a GAN (CTGAN) and generate 200K synthetic records ---

# 1) Define continuous & discrete features into lists
continuous_columns = [
    'step', 'amount',
    'oldbalanceOrg', 'newbalanceOrig',
    'oldbalanceDest', 'newbalanceDest'
]

discrete_columns = [
    'type','isFraud','isFlaggedFraud',
    'loc_US','loc_EU','loc_APAC','loc_OTHER',
    'dev_mobile','dev_desktop','isp_known','vpn_used',
```

```
    'new_account','multi_account','repeat_txn',
    'night_txn','weekend_txn',
    'flag_manual_review','email_link_clicked','two_factor_used',
    'suspicious_ip','browser_tor'
]

# 2) Build the training DataFrame
training_df = aug_seed[continuous_columns + discrete_columns].copy()

# 3) Ensure continuous columns are numeric
for col in continuous_columns:
    training_df[col] = pd.to_numeric(training_df[col], errors='coerce')

print("Training DF shape:", training_df.shape)
print("Continuous dtypes:", training_df[continuous_columns].dtypes)
print("Discrete dtypes:", training_df[discrete_columns].dtypes)

# 5) Train CTGAN
ctgan = CTGAN(
    embedding_dim=128,
    generator_lr=2e-4,
    discriminator_lr=2e-4,
    epochs=300,
    batch_size=500,
    verbose=True
)

ctgan.fit(training_df, discrete_columns=discrete_columns)


#-- Generate 200K Synthetic Records & Save
synthetic_df = ctgan.sample(200_000)
print("Synthetic dataset shape:", synthetic_df.shape)

out_path = '/content/drive/MyDrive/Colab Notebooks/synthetic_fraud_200k.csv'
synthetic_df.to_csv(out_path, index=False)
print("Saved synthetic data to:", out_path)


#--- Load data and show a preview of the data
import pandas as pd
df = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/synthetic_fraud_200k.csv')
display(df.head())

df = df.drop(columns=['nameDest'], errors='ignore')
```

	step	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	type	isFraud	isFlaggedFraud	nam
0	392	4.610590e+04	2.508372e+04	-2.800997e+04	75232.622289	7.516133e+04	TRANSFER	0	0	M15344
1	351	1.791248e+06	-6.482367e+04	-3.541537e+04	33623.418421	1.096449e+07	TRANSFER	1	1	C6095
2	270	1.418663e+06	7.618701e+06	1.037706e+07	-15636.788689	8.418022e+06	PAYMENT	1	0	C1781C
3	589	6.322663e+05	4.773069e+04	-2.589893e+04	29768.739771	1.121087e+05	CASH_OUT	1	0	M192C
4	359	2.410356e+04	5.051327e+05	-2.011339e+04	-45050.502124	3.754478e+04	PAYMENT	0	0	C12907

5 rows × 28 columns

🛠️ 2. Feature Engineering: Translating Domain Knowledge into Additional Features

- After reviewing the data and main risk indicators, I create new features that capture fraud-related behavior, such as account draining, risk flag counts, and device or location usage patterns.
- This step includes one-hot encoding categorical variables, applying log transforms to skewed numeric fields, and handling missing values.
- The goal is to give the models more and better information so the models can more effectively identify fraud patterns.

Engineering Fraud-Focused Features and Cleaning Data

- This function creates new variables that capture fraud risk factors, ratio-based features, operational flags, and time-based patterns.

- Log-scaling helps control for outliers; ratios highlight unusual behavior, and counting risk flags helps identify txns with multiple warnings sign.
- One-hot encoding and missing value filling ensure compatibility with modern ML libraries and is generally just easier to work with when modelling.

```
def feature_engineering(df):
    df = df.copy()

    #--- One-hot encode transaction type for model input
    if 'type' in df.columns:
        df = pd.get_dummies(df, columns=['type'], prefix='type')

    #--- Ensures all binary columns are integers (prevents issues with scikit-learn)
    for col in df.columns:
        if df[col].nunique() == 2 and df[col].dtype != int:
            df[col] = df[col].astype(int)

    #--- Log-scales monetary columns to handle heavy right skew and outliers
    for col in ['amount','oldbalanceOrg','newbalanceOrig','oldbalanceDest','newbalanceDest']:
        if col in df.columns:
            df[col+'_log'] = np.log1p(np.maximum(df[col], 0))

    #--- amount as proportion of balances: key for detecting unusual activity
    df['amt_over_oldbal'] = df['amount'] / (df['oldbalanceOrg'] + 1e-6)
    df['amt_over_oldbal'] = df['amt_over_oldbal'].replace([np.inf, -np.inf], 0).fillna(0)
    df['amt_over_destbal'] = df['amount'] / (df['oldbalanceDest'] + 1e-6)
    df['amt_over_destbal'] = df['amt_over_destbal'].replace([np.inf, -np.inf], 0).fillna(0)

    #--- Origin and destination account balance change
    df['org_balance_delta'] = (df['newbalanceOrig'] - df['oldbalanceOrg']) / (df['oldbalanceOrg'] + 1e-6)
    df['org_balance_delta'] = df['org_balance_delta'].replace([np.inf, -np.inf], 0).fillna(0)
    df['dest_balance_delta'] = (df['newbalanceDest'] - df['oldbalanceDest']) / (df['oldbalanceDest'] + 1e-6)
    df['dest_balance_delta'] = df['dest_balance_delta'].replace([np.inf, -np.inf], 0).fillna(0)

    df['drain_amount'] = df['oldbalanceOrg'] - df['newbalanceOrig']
    df['drained_to_zero'] = (df['newbalanceOrig'] == 0).astype(int)

    #--- High amount flag to flag outlier transactions
    if 'amount' in df.columns and any(c.startswith('type_') for c in df.columns):
        type_cols = [c for c in df.columns if c.startswith('type_')]
        df['high_amt_for_type'] = 0
        for col in type_cols:
            mask = df[col] == 1
            thresh = df.loc[mask, 'amount'].quantile(0.9) if mask.sum() > 0 else df['amount'].quantile(0.9)
            df.loc[mask, 'high_amt_for_type'] = (df.loc[mask, 'amount'] > thresh).astype(int)
    else:
        df['high_amt_for_type'] = 0

    #--- Time-based features: night-time and weekend indicator
    if 'hour' in df.columns:
        df['is_night'] = df['hour'].between(0, 6).astype(int)
    else:
        df['is_night'] = 0
    if 'step' in df.columns:
        df['day_of_week'] = ((df['step'] // 24) % 7)
        df['is_weekend'] = df['day_of_week'].isin([5,6]).astype(int)
    else:
        df['is_weekend'] = 0

    #--- Operational flags: zero destination balance after large transfer, transfer to new account
    df['suspicious_zero_dest_after'] = ((df['amount'] > df['amount'].quantile(0.75)) & (df['newbalanceDest'] == 0)).astype(int)
    df['dest_was_zero_before'] = (df['oldbalanceDest'] == 0).astype(int)

    #--- Stacking risk factors: count number of high-risk flags for each transaction
    high_risk_cols = [c for c in ['vpn_used', 'new_account', 'high_risk_country', 'suspicious_ip', 'browser_tor'] if c in df.col
    df['risk_factor_count'] = df[high_risk_cols].sum(axis=1) if high_risk_cols else 0

    #--- Remove name columns (not useful in modeling)
    for col in ['nameOrig','nameDest']:
        if col in df.columns:
            df.drop(col, axis=1, inplace=True)

    #--- Fills missing values
    df = df.fillna(0)
    return df
```

```
#--- Apply feature engineering fn
df_fe = feature_engineering(df)
print("Shape after feature engineering:", df_fe.shape)
df_fe.head()


df = df_fe.copy()
```

```
Shape after feature engineering: (200000, 49)
```

Insights:

- The resulting dataframe now contains the following new features:
- Amount ratios (amt_over_oldbal, amt_over_destbal):
  - If transactions are unusually high compared to the available balance, these features can help spot draining, laundering, or suspicious fund movement.
- Balance change features (org_balance_delta, dest_balance_delta):
  - This makes it possible to flag rapid or unexpected changes in account balances, which are often warning signs of fraud.
- Account draining indicators (drain_amount, drained_to_zero):
  - This is so that we can catch scenarios where an account is fully emptied in a single move, which could indicate ATO (account takeover).
- High-amount and type-specific outlier flag:
  - If a transaction stands out as much larger than others, this feature will highlight it as a potential anomaly.
- Time-based variables (hour, day_of_week, is_night, is_weekend):
  - These time-based features may provide insight into when fraud is more likely to occur, such as during nights or weekends.
- Risk stacking (risk_factor_count):
  - By counting the number of separate risk signals, this feature makes it easier to spot transactions that have multiple suspicious signals.

📊 3. Exploratory Data Analysis (EDA)

EDA Summary:

- Load the synthetic fraud dataset and check structure with summary statistics.
- Visualize the class balance to understand the ratio of fraud to non-fraud txns.
- Plot key numeric features, split by fraud status to see which variables separate the classes best.
- Highlight the top features that most distinguish fraud cases using bar charts of means and fraud rates.
- Examine operational, device, and location risk signals by plotting fraud rates for each category.
- Use correlation heatmap to find which features are closely related to the fraud flag.
- Analyze transaction timing with charts by hour of day and day of week to pick up on temporal fraud patterns.
- Segment results by recipient type, such as merchant versus customer, to find group specific trends.

```
#--- summary statistics for all numeric columns
display(df.describe())
```

	step	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
count	200000.000000	2.000000e+05	2.000000e+05	2.000000e+05	2.000000e+05	2.000000e+05	200000.000000	200000.000000
mean	304.738810	6.342728e+05	1.460000e+06	4.426793e+05	6.820825e+05	9.441224e+05	0.500945	0.011935
std	216.482925	1.569879e+06	3.540466e+06	2.111058e+06	1.989832e+06	2.270049e+06	0.500000	0.108594
min	-27.000000	-9.077059e+05	-4.890584e+05	-1.649973e+06	-1.821978e+05	-8.858426e+05	0.000000	0.000000
25%	150.000000	8.226629e+03	8.301572e+04	-3.826667e+04	3.653104e+04	3.532789e+04	0.000000	0.000000
50%	242.000000	8.223183e+04	3.421442e+05	-1.631537e+04	8.189809e+04	1.039465e+05	1.000000	0.000000
75%	432.000000	5.311762e+05	1.025053e+06	1.184355e+04	1.481787e+05	7.083043e+05	1.000000	0.000000
max	820.000000	1.176870e+07	3.735744e+07	3.287044e+07	3.628708e+07	3.091285e+07	1.000000	1.000000

8 rows × 49 columns

Insights:

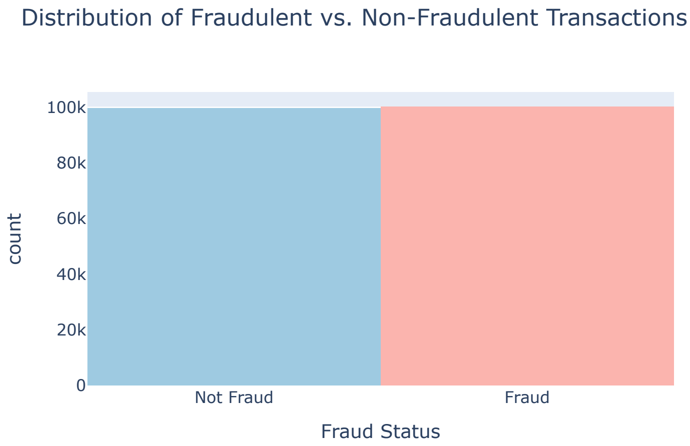
- The dataset is large enough (200,000 rows) and includes 50+ features, providing strong coverage for fraud analysis.
- Transaction amounts and balances show heavy skew and extreme outliers, so log-scaling is essential for effective modeling.
- Many accounts are fully drained by a transaction, which matches common fraud behaviors seen in practice.
- Most transactions have low risk factor counts, but a small subset display multiple high-risk flags. These are likely the main fraud signals in the data.

Visualizing Fraud Class

Distribution of Fraud vs. Non-Fraud Transactions

- This plot shows that the synthetic dataset was created with a perfectly balanced number of fraudulent and non-fraudulent transactions.
- A balanced class distribution like this is useful for model development and benchmarking, as it removes the extreme class imbalance that is common in real-world fraud data.

```
import plotly.express as px

PASTEL_BLUE = "#9ecae1"
PASTEL_PINK = "#fbb4ae"

#--- Plot the class balance
fig = px.histogram(
    df, x="isFraud",
    color="isFraud",
    color_discrete_sequence=[PASTEL_BLUE, PASTEL_PINK],
    title="Distribution of Fraudulent vs. Non-Fraudulent Transactions",
    labels={"isFraud": "Fraud Status"},
    width=600, height=400
)
fig.update_xaxes(tickvals=[0,1], ticktext=["Not Fraud", "Fraud"])
fig.update_layout(showlegend=False)
fig.show()
```



Visualizing Top 15 Features with Greatest Difference in Class (fraud vs non fraud)

```
from plotly.subplots import make_subplots
import plotly.graph_objects as go


# Gets all categorical/binary columns with not more than 10 categories
cat_cols = [
    c for c in df.columns
    if df[c].nunique() <= 10 and c not in ["isFraud", "isFlaggedFraud"]
]

def top_class_difference_features(df, target, n=10):
    feature_scores = []
    for col in cat_cols:
        counts = df.groupby(col)[target].mean()
        score = abs(counts.max() - counts.min())
        feature_scores.append((col, score))
    return [x[0] for x in sorted(feature_scores, key=lambda x: -x[1])[:n]]

top15_fraud = top_class_difference_features(df, "isFraud", n=15)
top15_flagged = top_class_difference_features(df, "isFlaggedFraud", n=10)


def plot_categorical_counts_side_by_side(df, features, target, palette, batch_size=3):
    fraud_map = {0: "Not Fraud", 1: "Fraud"}
    n = len(features)
    for i in range(0, n, batch_size):
        batch = features[i:i+batch_size]
        cols = len(batch)
        fig = make_subplots(rows=1, cols=cols, subplot_titles=[
            f"<b>{col.replace('_', ' ').title()}</b>" for col in batch])
        for j, col in enumerate(batch, 1):
            value_counts = df.groupby([col, target]).size().reset_index(name="count")
            cats = sorted(df[col].unique())
            for idx, t in enumerate(sorted(df[target].unique())):
                vals = value_counts[value_counts[target]==t]
                # Only show legend for the first subplot
                show_leg = (j == 1)
                name = fraud_map.get(t, str(t)) if show_leg else None
                fig.add_trace(go.Bar(
                    x=vals[col].astype(str),
                    y=vals["count"],
                    name=name,
                    marker_color=palette[idx],
                    width=0.55,
                    text=vals["count"],
                    textposition='auto',
                    opacity=0.95,
                    showlegend=show_leg
                ), row=1, col=j)
            # Axis labels
```

```
            # fig.update_xaxes(title_text=col, title_font=dict(size=8), row=1, col=j)
            fig.update_yaxes(title_text="Count", title_font=dict(size=8), row=1, col=j)
        fig.update_layout(
            barmode="group",
            width=1100, height=400,
            legend=dict(
                orientation="v",
                font=dict(size=9),
                x=1.02, xanchor="left", y=1, yanchor="top",
                bgcolor="rgba(255,255,255,0.7)",
                bordercolor="gray",
                borderwidth=1
            ),
            plot_bgcolor="white",
            paper_bgcolor="white"
        )
        fig.show()

# Example usage:
plot_categorical_counts_side_by_side(df, top15_fraud, "isFraud", [PASTEL_BLUE, PASTEL_PINK])
```


Risk Factor Count
73111
11863
19033
21001
5574
31109
1759
27242
334
8974
Browser Tor
89479
24509
10332
75680
Suspicious Ip
93044
41211
6767
58978
Type Cash In
75795
97527
24016
2662
Type Payment
81236
98052
18575
2137
Vpn Used
95141
70435
4670
29754
Night Txn
92897
64419
6914
35770
Multi Account
93688
71912
6123
28277
Type Transfer
87077
55907
12734
44282
New Account
84786
63760
36429
High Amt For Type
94399
85599
Loc Other
1856
21449
40914
43539
39045
32864
Count

Insights:

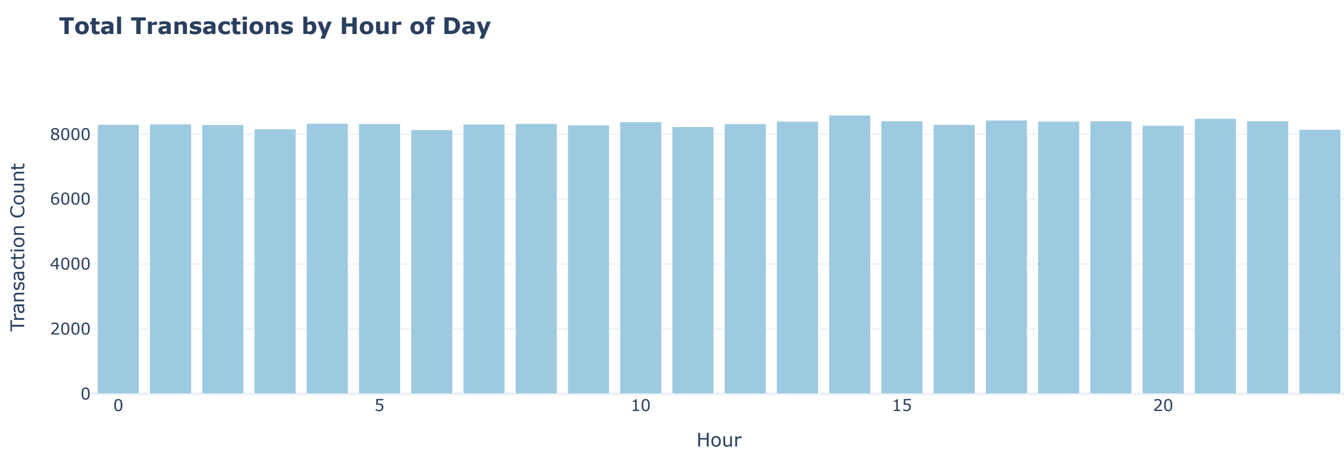
- Most non-fraud transactions have zero risk factors. Fraud cases are much more likely to have two or more stacked risk flags.
- Fraud is concentrated in CASH_OUT and TRANSFER transaction types. Legit transactions are mainly found in PAYMENT and CASH_IN.
- Tor browser and VPN use are rare for non fraud but common for fraud.Fraudsters rely heavily on these IP anonymizing tools.
- High-amount transactions are much more likely to be fraud. The high_amt_for_type feature helps surface these cases.
- Behavioral flags like multi_account, new_account, and night_txn are more frequent in fraud. These patterns help the model pick up attacker tactics.
- Fraud transactions more often involve clicking suspicious email links. This adds another layer of behavioral evidence.
- Device and location matter. Fraud is more likely on desktop and less likely in the loc_OTHER category.
- Manual review and risk stacking features both point to higher fraud likelihood, giving clear signals for risk teams.
- Fraud is also more common for accounts flagged as new. This suggests attackers may target recently opened accounts.

Time-Based Feature Distributions by Fraud Status

- This code defines and uses a helper function to plot the distributions time based features created from the 'step' column we were provided, by the fraud label.

```
from plotly.subplots import make_subplots
import plotly.graph_objects as go


# --- Deriving time-based features from 'step' coloumn
df['hour'] = df['step'] % 24
df['day_of_week'] = (df['step'] // 24) % 7
df['is_night'] = df['hour'].between(0, 6).astype(int)
df['is_weekend'] = df['day_of_week'].isin([5, 6]).astype(int)


# --- Prepare data for all four plots ---
fraud_by_hour = df.groupby('hour')['isFraud'].mean().reset_index()
fraud_by_day = df.groupby('day_of_week')['isFraud'].mean().reset_index()
fraud_by_night = df.groupby('is_night')['isFraud'].mean().reset_index()
fraud_by_weekend = df.groupby('is_weekend')['isFraud'].mean().reset_index()
```

```python
# --- Subplot: Four time-based fraud rate charts, 2x2 grid ---
fig = make_subplots(
    rows=2, cols=2,
    subplot_titles=[
        "<b>Fraud Rate by Hour of Day</b>",
        "<b>Fraud Rate by Day of Week</b>",
        "<b>Fraud Rate by Night vs. Day</b>",
        "<b>Fraud Rate by Weekend</b>"
    ]
)

fig.add_trace(go.Bar(
    x=fraud_by_hour['hour'], y=fraud_by_hour['isFraud'],
    marker_color=PASTEL_PINK,
    showlegend=False
), row=1, col=1)

fig.add_trace(go.Bar(
    x=fraud_by_day['day_of_week'], y=fraud_by_day['isFraud'],
    marker_color=PASTEL_PINK,
    showlegend=False
), row=1, col=2)

fig.add_trace(go.Bar(
    x=['Day', 'Night'],
    y=fraud_by_night.sort_values('is_night')['isFraud'],
    marker_color=PASTEL_PINK,
    showlegend=False
), row=2, col=1)

fig.add_trace(go.Bar(
    x=['Weekday', 'Weekend'],
    y=fraud_by_weekend.sort_values('is_weekend')['isFraud'],
    marker_color=PASTEL_PINK,
    showlegend=False
), row=2, col=2)

fig.update_layout(
    height=700, width=1100,
    plot_bgcolor="white", paper_bgcolor="white",
    title_text="<b>Time-based Fraud Risk Patterns</b>",
    font=dict(size=12, family="Arial"),
    margin=dict(l=40, r=30, t=80, b=40)
)

for i in range(1, 3):
    fig.update_xaxes(title_font=dict(size=10), row=1, col=i)
    fig.update_xaxes(title_font=dict(size=10), row=2, col=i)
    fig.update_yaxes(title_text="Fraud Rate", title_font=dict(size=10), row=1, col=i)
    fig.update_yaxes(title_text="Fraud Rate", title_font=dict(size=10), row=2, col=i)

fig.show()

# --- Transaction count over hour of day (as before) ---
import plotly.express as px
txns_per_hour = df['hour'].value_counts().sort_index()
px.bar(
    x=txns_per_hour.index, y=txns_per_hour.values,
    labels={'x': 'Hour', 'y': 'Transaction Count'},
    title="<b>Total Transactions by Hour of Day</b>",
    color_discrete_sequence=[PASTEL_BLUE], width=1100, height=400,
    template='plotly_white'
).show()
```

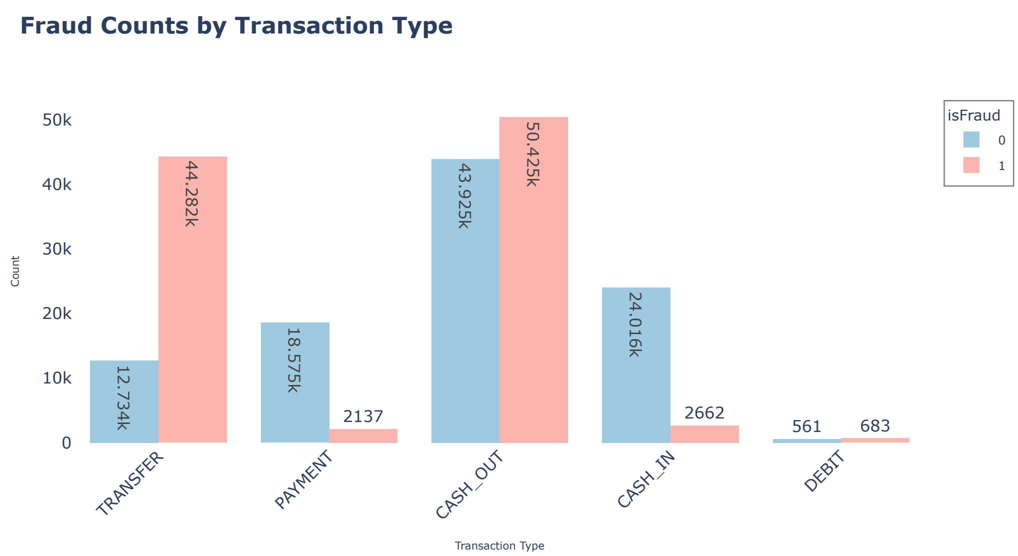


Insights:

- The overall transaction volume stays almost perfectly flat across all hours of the day. This suggests there's no business-driven transaction surge at any specific time, so time-of-day alone doesn't separate fraud from non-fraud.
- The fraud rate by hour is also extremely steady, with almost no difference between early morning, midday, or late night hours. Attackers may be distributing their activity to avoid time-based detection.
- There's only minimal variation in fraud rate across days of the week. No specific weekday or weekend effect is visible in this data; fraud risk remains consistent from Monday to Sunday.
- The fraud rate doesn't change meaningfully between night and day transactions. Nighttime does not pose extra risk in this synthetic data.
- There is also no significant difference in fraud risk between weekends and weekdays so these may not be the best predictors for our model.

Fraud Counts by Transaction Type

```
# Recreating the 'type' categorical coloumn since we one-hot encoded it earlier but dropped the original coloumn.
# We could just include that column but this is good to know how to do this.

type_cols = [c for c in df.columns if c.startswith('type_')]

def recover_type(row):
    for col in type_cols:
        if row[col] == 1:
            return col.replace('type_', '')
    return "OTHER"

if 'type' not in df.columns:
    df['type'] = df[type_cols].apply(recover_type, axis=1)

import plotly.express as px

fig = px.histogram(
    df, x='type', color='isFraud',
    barmode="group", text_auto=True,
    title="<b>Fraud Counts by Transaction Type</b>",
    color_discrete_sequence=[PASTEL_BLUE, PASTEL_PINK]
)
fig.update_layout(
    width=800, height=450,
    xaxis_tickangle=-45,
    xaxis_title=dict(text="Transaction Type", font=dict(size=8)),
    yaxis_title=dict(text="Count", font=dict(size=8)),
    legend=dict(
        orientation="v",
        font=dict(size=9),
        x=1.02, xanchor="left", y=1, yanchor="top",
        bgcolor="rgba(255,255,255,0.7)",
        bordercolor="gray",
        borderwidth=1
    ),
    plot_bgcolor="white",
    paper_bgcolor="white"
)
fig.show()
```



Insights:

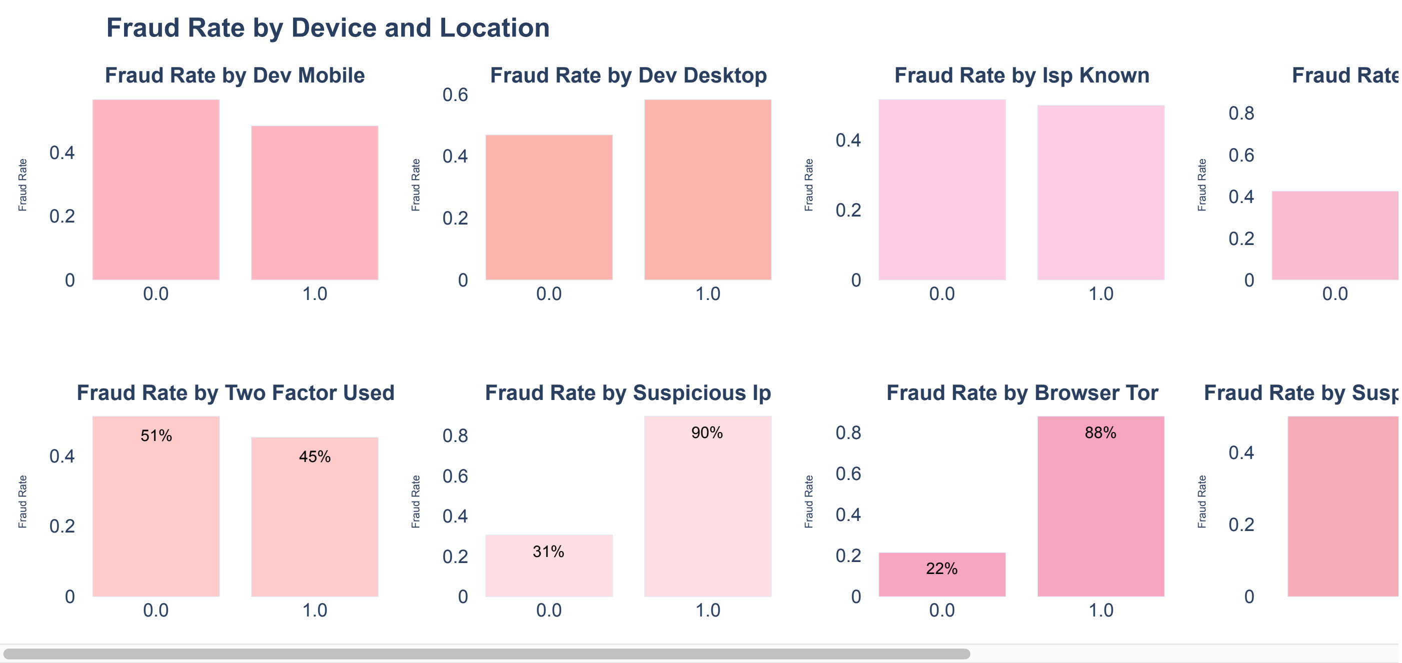
- Fraud is heavily concentrated in CASH_OUT and TRANSFER transactions. These types account for the vast majority of fraud cases in the data.
- Legitimate transactions make up the bulk of PAYMENT and CASH_IN activity, with very few fraud cases in these categories.
- Both fraud and non-fraud counts are low for DEBIT, indicating this type is rare overall.
- Transaction type clearly separates risky from safe activity, making it a key feature for any fraud model.

Exploring Risk Flags, Device, and Geo Patterns

- this section focuses on risk signals such as VPN use, Tor browser, suspicious IP, device type, and manual review triggers.
- Visualizes fraud rates for each flag, highlighting which features actually separate fraud from normal activity.

```
import plotly.express as px
from plotly.subplots import make_subplots
import plotly.graph_objects as go

risk_flag_cols = [
    'dev_mobile', 'dev_desktop', 'isp_known', 'vpn_used',
    'flag_manual_review', 'two_factor_used', 'suspicious_ip', 'browser_tor',
    'suspicious_zero_dest_after', 'risk_factor_count'
]

custom_pinks = [
    "#ffb6c1", "#fbb4ae", "#fdcce5", "#f8bbd0", "#e7b6c9",
    "#ffcccb", "#ffdde4", "#f8a5c2", "#f6abb6", "#ffafcc"
]

fraud_rates = []
for col in risk_flag_cols:
    temp = df.groupby(col, as_index=False)["isFraud"].mean()
    temp['feature'] = col
    fraud_rates.append(temp)
fraud_rates_df = pd.concat(fraud_rates, ignore_index=True)

fig = make_subplots(
    rows=2, cols=5,
    subplot_titles=[f"<b>Fraud Rate by {col.replace('_',' ').title()}</b>" for col in risk_flag_cols]
)

annotations = []

for i, col in enumerate(risk_flag_cols):
    r, c = divmod(i, 5)
    data = fraud_rates_df[fraud_rates_df['feature'] == col]
    x_labels = data[col].astype(str)
    y_vals = data["isFraud"]
    bar_color = custom_pinks[i % len(custom_pinks)]
    fig.add_trace(
        go.Bar(
            x=x_labels,
            y=y_vals,
            marker_color=bar_color,
            showlegend=False
        ),
        row=r+1, col=c+1
    )
    # Add annotation for each bar (inside top of bar)
    for xi, yi in zip(x_labels, y_vals):
        # Offset is a small fraction of the bar height so text is inside
        offset = 0.02 if yi > 0.1 else yi * 0.2
        annotations.append(dict(
            x=xi, y=yi - offset,
            text=f"{int(round(yi*100))}%",
            showarrow=False,
            font=dict(size=12, color="black"),
            xanchor="center",
            yanchor="top",
            xref=f"x{i+1}",
            yref=f"y{i+1}"
```

```
        ))

fig.update_layout(
    height=500, width=1500,
    plot_bgcolor="white", paper_bgcolor="white",
    title_text="<b>Fraud Rate by Device and Location</b>",
    font=dict(size=14, family="Arial"),
    margin=dict(l=40, r=30, t=80, b=40),
    annotations=annotations
)

for i in range(1, 6):
    fig.update_xaxes(title_font=dict(size=8), row=1, col=i)
    fig.update_xaxes(title_font=dict(size=8), row=2, col=i)
    fig.update_yaxes(title_text="Fraud Rate", title_font=dict(size=8), row=1, col=i)
    fig.update_yaxes(title_text="Fraud Rate", title_font=dict(size=8), row=2, col=i)

fig.show()
```



Insights:

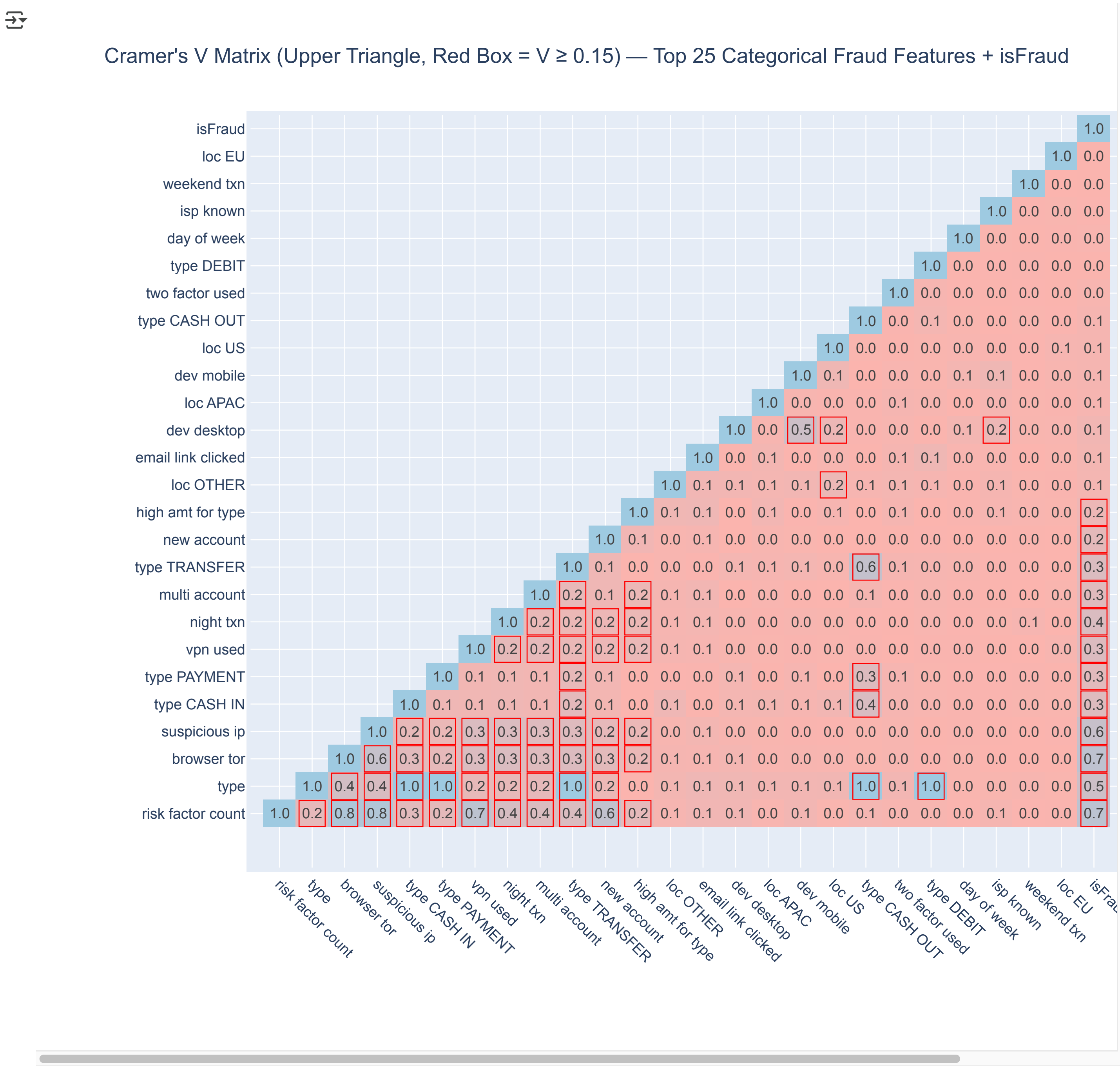
- Fraud jumps above 90% when multiple risk factors are stacked together, making risk_factor_count one of the strongest single indicators in the data.
- Tor browser and suspicious IP are very important; when either is present, the fraud rate reaches nearly 90%.
- Transactions with a suspicious zero destination after, or those flagged for manual review, show much higher fraud rates, typically around 50% or more.
- VPN use and multi-account activity are both strong risk signals, with fraud nearly doubling when these are detected.
- Two-factor authentication shows good indiciative value. Its associated with a lower fraud rate.

Investigating Correlations Among Features

- This correlation heatmap compares the relationships between the top 25 categorical and binary features and the fraud target using Cramér's V, a correlation measure which is suited for categorical vars.
- Cramér's V is used here because it handles binary and multi-class features, and finds non-linear associations that might be missed by linear methods.
- The upper triangle heatmap displays all pairwise correlations, with red boxes marking feature pairs that are significantly related.

```
import scipy.stats as ss

def cramers_v(x, y):
    confusion_matrix = pd.crosstab(x, y)
    chi2 = ss.chi2_contingency(confusion_matrix)[0]
    n = confusion_matrix.sum().sum()
    phi2 = chi2 / n
    r, k = confusion_matrix.shape
    phi2corr = max(0, phi2 - ((k-1)*(r-1))/(n-1))
    rcorr = r - ((r-1)**2)/(n-1)
    kcorr = k - ((k-1)**2)/(n-1)
    return np.sqrt(phi2corr / min((kcorr-1), (rcorr-1)))


# Get top 25 categorical/binary features by class separation
cat_cols = [c for c in df.columns if df[c].nunique() <= 10 and c not in ["isFraud", "isFlaggedFraud"]]
top25_cat = top_class_difference_features(df, "isFraud", n=25)

def cramers_v_matrix(df, cols):
    n = len(cols)
    mat = np.full((n, n), np.nan)  # Default all to blank/white
    for i, a in enumerate(cols):
        for j, b in enumerate(cols):
            if i <= j:   # Upper triangle (including diagonal)
                mat[i, j] = cramers_v(df[a], df[b])
    return mat

top25_cat_cols = top25_cat
mat = cramers_v_matrix(df, top25_cat_cols + ["isFraud"])

labels = [c.replace("_", "\n") for c in top25_cat_cols + ["isFraud"]]
thresh = 0.15

# Prepare annotation text matrix: blank for lower triangle, 1 decimal elsewhere
text_matrix = np.empty(mat.shape, dtype=object)
for i in range(mat.shape[0]):
    for j in range(mat.shape[1]):
        if i <= j and not np.isnan(mat[i, j]):
            text_matrix[i, j] = f"{mat[i, j]:.1f}"
        else:
            text_matrix[i, j] = ""

# Find coordinates for significant cells (not diagonal)
xs, ys = [], []
for i in range(mat.shape[0]):
    for j in range(mat.shape[1]):
        val = mat[i, j]
        if i < j and not np.isnan(val) and val >= thresh and val != 1:
            xs.append(labels[j])
            ys.append(labels[i])

import plotly.graph_objects as go

fig = go.Figure()

# Main heatmap
fig.add_trace(go.Heatmap(
    z=mat,
    x=labels,
    y=labels,
    colorscale=[[0, PASTEL_PINK], [1, PASTEL_BLUE]],
    colorbar=dict(title="Cramer's V"),
    zmin=0, zmax=1,
    hoverongaps=False,
    text=text_matrix,
    texttemplate="%{text}"
))

# Overlay scatter for red box markers (not on diagonal)
fig.add_trace(go.Scatter(
    x=xs,
    y=ys,
    mode="markers",
    marker=dict(
        symbol="square-open",
        color="red",
        size=25,
```

```
            line=dict(width=1)
        ),
        hoverinfo="skip",
        showlegend=False
    ))

    fig.update_layout(
        title=f"Cramer's V Matrix (Upper Triangle, Red Box = V ≥ {thresh}) — Top 25 Categorical Fraud Features + isFraud",
        width=1200, height=1000,
        font=dict(size=14, family="Arial"),
        margin=dict(l=200, r=60, b=180, t=100, pad=4)
    )
    fig.update_xaxes(tickangle=45, side="bottom")
    fig.show()
```



Insights:

- Risk factor count has strong overlap with key risk flags like browser_tor and suspicious_ip, which makes sense since it aggregates these indicators.

- Several fraud flags, including suspicious_ip, browser_tor, vpn_used, and type_TRANSFER, often appear together in the same transactions.
- Transaction type and device features show much weaker correlation, meaning they add fresh signals that aren't just repeats of other risk flags.
- No signs of direct leakage but the stacked risk feature, 'risk factor count' is highly related and could be a cuase of collinearity in the model.

🔧 4. Hyperparameter Tuning and Model Training

- With the new features engineered, the next step is to tune each model's key parameters to get the best possible fraud detection performance.
- Grid search and cross-validation are used to find the optimal hyperparameters.
- Hyperparameter tuning will be done before final training and testing, which keeps evaluation realistic and ensures all models are compared fairly.
- Three models are trained and evaluated: Logistic Regression for a baseline, plus Random Forest and LightGBM as our main prediction models. Performance is measured using AUC/ROC and precision-recall metrics.

Hyperparameter Tuning with Grid Search

- Splitting the dataset into training and testing partitions, using stratification to maintain the same fraud rate in both sets.

```
#--- Spliting data into train test sets.
from sklearn.model_selection import train_test_split

drop_cols = ['isFraud', 'isFlaggedFraud']
features = [c for c in df_fe.columns if c not in drop_cols]
X = df_fe[features]
y = df_fe['isFraud']

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

#--- Fills any remaining missing values
X_train = X_train.fillna(0)
X_test = X_test.fillna(0)
```

- This code below runs grid search over multiple combinations of tree depth, number of trees, and other model settings for both Random Forest and LightGBM. The best hyperparameters are then selected for final model evaluation.
- The code will also save the best hyperparameters into a local drive to avoid rerunning the code and maintaining consistent models.

```
#--- Hyperparameter Tuning for Random Forest and LightGBM with Save/Load Options

from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from lightgbm import LGBMClassifier
import json
import pickle
import os

# Define file paths in Google Drive
RF_PARAMS_PATH = '/content/drive/MyDrive/Colab Notebooks/best_rf_params.json'
LGBM_PARAMS_PATH = '/content/drive/MyDrive/Colab Notebooks/best_lgbm_params.json'
RF_MODEL_PATH = '/content/drive/MyDrive/Colab Notebooks/best_rf_model.pkl'
LGBM_MODEL_PATH = '/content/drive/MyDrive/Colab Notebooks/best_lgbm_model.pkl'

def tune_and_save(model, param_grid, X, y, params_path, model_path):
    gs = GridSearchCV(model, param_grid, scoring='roc_auc', cv=3, n_jobs=-1, verbose=1)
    gs.fit(X, y)
    print(f"Best params: {gs.best_params_} | Best ROC AUC: {gs.best_score_:.3f}")
    # Save best params
```

```
    with open(params_path, 'w') as f:
        json.dump(gs.best_params_, f)
    # Save model
    with open(model_path, 'wb') as f:
        pickle.dump(gs.best_estimator_, f)
    return gs.best_estimator_

def load_params_and_model(params_path, model_path, model_class):
    # Try loading model; if not found, load params
    if os.path.exists(model_path):
        print(f"Loading model from {model_path}")
        with open(model_path, 'rb') as f:
            model = pickle.load(f)
    elif os.path.exists(params_path):
        print(f"Loading params from {params_path}")
        with open(params_path, 'r') as f:
            params = json.load(f)
        model = model_class(**params, class_weight='balanced', random_state=42)
    else:
        raise FileNotFoundError(f"model or params were not found in {model_path}")
    return model

# Hyperparameter grids
rf_params = {
    'n_estimators': [100, 200],
    'max_depth': [10, None],
    'min_samples_split': [2, 5]
}
lgb_params = {
    'n_estimators': [100, 200],
    'max_depth': [10, -1],
    'learning_rate': [0.05, 0.1]
}

# --- Run tuning or load existing models ---
RUN_TUNING = False  # Set to True to re-run GridSearchCV (time-consuming)

if RUN_TUNING:
    best_rf = tune_and_save(
        RandomForestClassifier(class_weight='balanced', random_state=42), rf_params,
        X_train, y_train, RF_PARAMS_PATH, RF_MODEL_PATH
    )
    best_lgbm = tune_and_save(
        LGBMClassifier(class_weight='balanced', random_state=42), lgb_params,
        X_train, y_train, LGBM_PARAMS_PATH, LGBM_MODEL_PATH
    )
else:
    best_rf = load_params_and_model(RF_PARAMS_PATH, RF_MODEL_PATH, RandomForestClassifier)
    best_lgbm = load_params_and_model(LGBM_PARAMS_PATH, LGBM_MODEL_PATH, LGBMClassifier)
```

```
Loading model from /content/drive/MyDrive/Colab Notebooks/best_rf_model.pkl
Loading model from /content/drive/MyDrive/Colab Notebooks/best_lgbm_model.pkl
```

Insights:

- Increasing tree depth and the number of estimators improves performance up to a limit.
- LightGBM works best with a lower learning rate to balance accuracy and overfitting.
- Tuning these settings before training gives a better shot at catching rare fraud than just using the defaults.

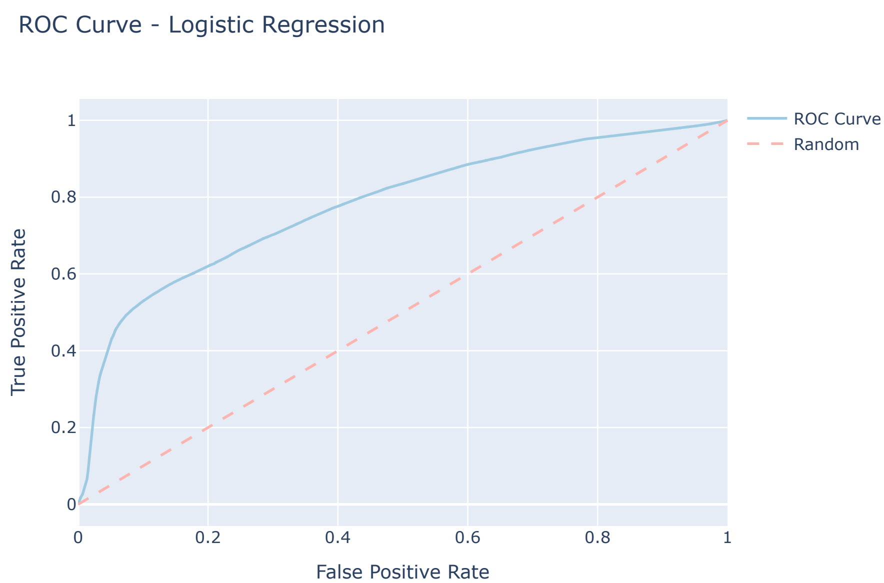
Model Training and Evaluation with Multiple Metrics

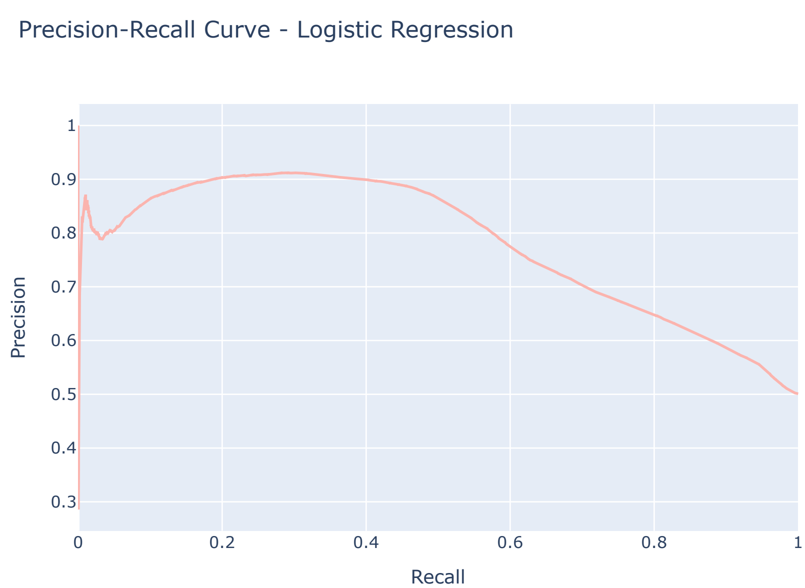
- Each model is trained and tested using a full set of evaluation metrics, including ROC AUC, precision-recall AUC, and the confusion matrix.
- ROC and precision-recall curves help show how well the model separates fraud from non-fraud, even when the data is imbalanced.
- The confusion matrix summarizes the true positives, false positives, showing exactly where the model succeeds and where it struggles.

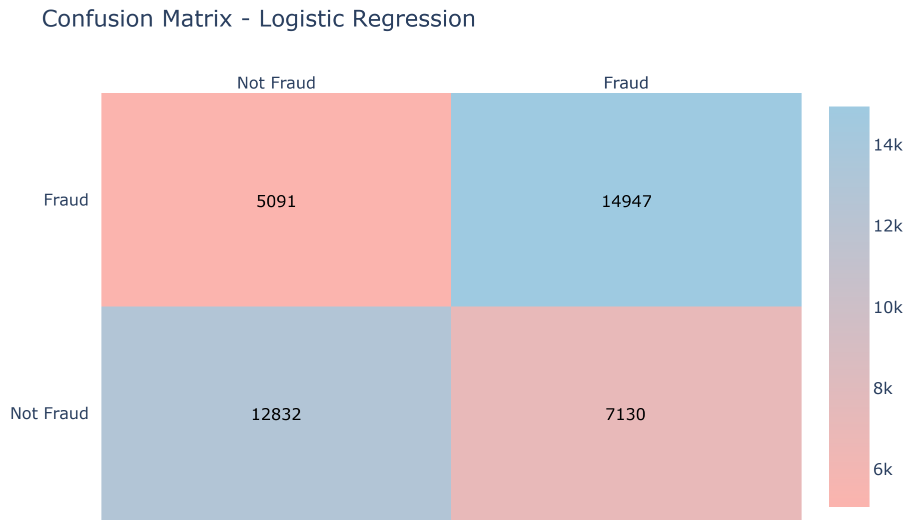
```
#--- Train and Evaluate Logistic Regression, Random Forest, LightGBM

from sklearn.linear_model import LogisticRegression
import plotly.graph_objs as go
```

```
import plotly.figure_factory as ff
from sklearn.metrics import (
    roc_auc_score, average_precision_score, f1_score,
    precision_recall_curve, roc_curve, confusion_matrix, classification_report
)

PASTEL_BLUE = "#9ecae1"
PASTEL_PINK = "#fbb4ae"

models = {
    'Logistic Regression': LogisticRegression(max_iter=500, class_weight='balanced', solver='lbfgs'),
    'Random Forest': best_rf,
    'LightGBM': best_lgbm
}
fitted_models = {}
results = []

for name, model in models.items():
    #--- Fit model (if not already fitted, e.g., if loading from params only)
    if not hasattr(model, "classes_"):
        model.fit(X_train, y_train)
    fitted_models[name] = model

    #--- Predict and score
    y_pred = model.predict(X_test)
    y_proba = model.predict_proba(X_test)[:,1]
    auc = roc_auc_score(y_test, y_proba)
    pr_auc = average_precision_score(y_test, y_proba)
    f1 = f1_score(y_test, y_pred)
    cm = confusion_matrix(y_test, y_pred)
    print(f"\n=== {name} ===")
    print("ROC AUC: {:.3f} | PR AUC: {:.3f} | F1: {:.3f}".format(auc, pr_auc, f1))
    print("Classification report:\n", classification_report(y_test, y_pred))
    print("Confusion matrix:\n", cm)
    results.append({'model':name, 'ROC_AUC':auc, 'PR_AUC':pr_auc, 'F1':f1})

    #--- ROC Curve
    fpr, tpr, _ = roc_curve(y_test, y_proba)
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=fpr, y=tpr, mode='lines', name='ROC Curve', line=dict(color=PASTEL_BLUE)))
    fig.add_trace(go.Scatter(x=[0,1], y=[0,1], mode='lines', name='Random', line=dict(dash='dash', color=PASTEL_PINK)))
    fig.update_layout(title=f"ROC Curve - {name}", xaxis_title="False Positive Rate", yaxis_title="True Positive Rate", width=70
    fig.show()

    #--- PR Curve
    precision, recall, _ = precision_recall_curve(y_test, y_proba)
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=recall, y=precision, mode='lines', name='PR Curve', line=dict(color=PASTEL_PINK)))
    fig.update_layout(title=f"Precision-Recall Curve - {name}", xaxis_title="Recall", yaxis_title="Precision", width=700, height
    fig.show()

    #--- Confusion Matrix visualization
    z = cm
    x = ['Not Fraud', 'Fraud']
    y_ = ['Not Fraud', 'Fraud']
    fig = ff.create_annotated_heatmap(
        z=z, x=x, y=y_, colorscale=[[0, PASTEL_PINK], [1, PASTEL_BLUE]], showscale=True,
        annotation_text=z, hoverinfo="z"
    )
    fig.update_layout(title=f"Confusion Matrix - {name}", width=700, height=500)
    fig.show()
```

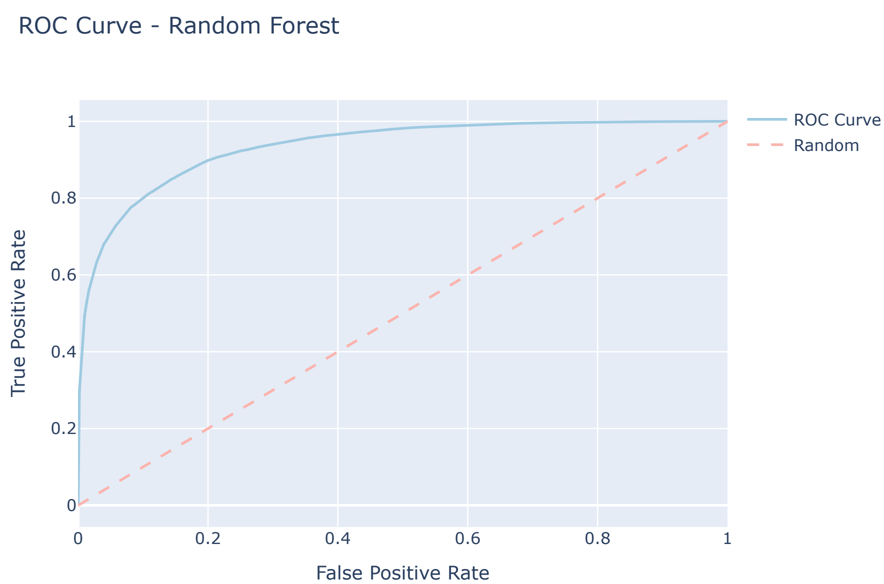
```
=== Logistic Regression ===
ROC AUC: 0.780 | PR AUC: 0.780 | F1: 0.710
Classification report:
              precision    recall  f1-score   support

           0       0.72      0.64      0.68     19962
           1       0.68      0.75      0.71     20038

    accuracy                           0.69     40000
   macro avg       0.70      0.69      0.69     40000
weighted avg       0.70      0.69      0.69     40000

Confusion matrix:
 [[12832  7130]
 [ 5091 14947]]
/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_logistic.py:465: ConvergenceWarning:

lbfgs failed to converge (status=1):
STOP: TOTAL NO. OF ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
```

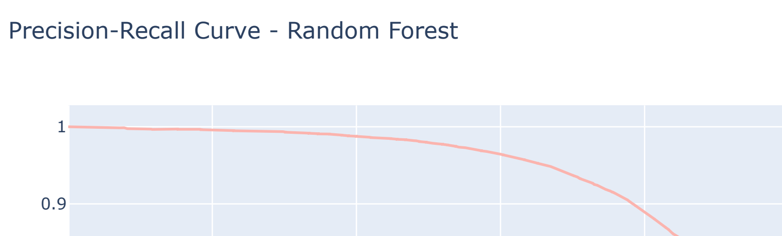


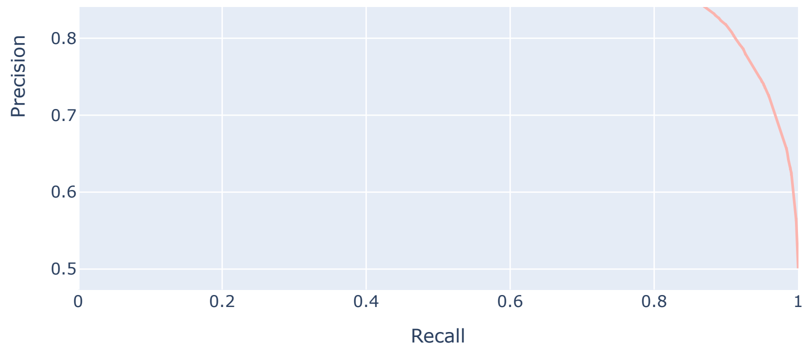


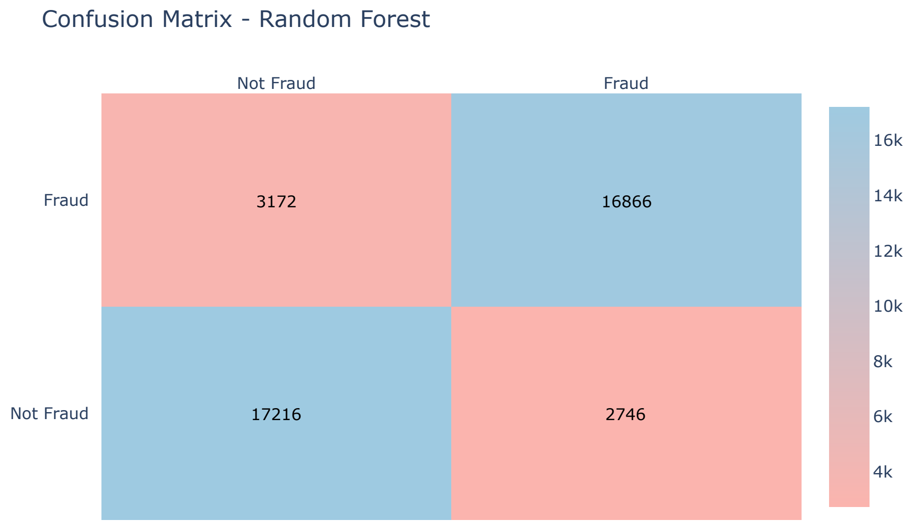



```
=== Random Forest ===
ROC AUC: 0.935 | PR AUC: 0.939 | F1: 0.851
Classification report:
               precision    recall  f1-score   support

           0       0.84      0.86      0.85     19962
           1       0.86      0.84      0.85     20038

    accuracy                           0.85     40000
   macro avg       0.85      0.85      0.85     40000
weighted avg       0.85      0.85      0.85     40000

Confusion matrix:
 [[17216  2746]
 [ 3172 16866]]
```

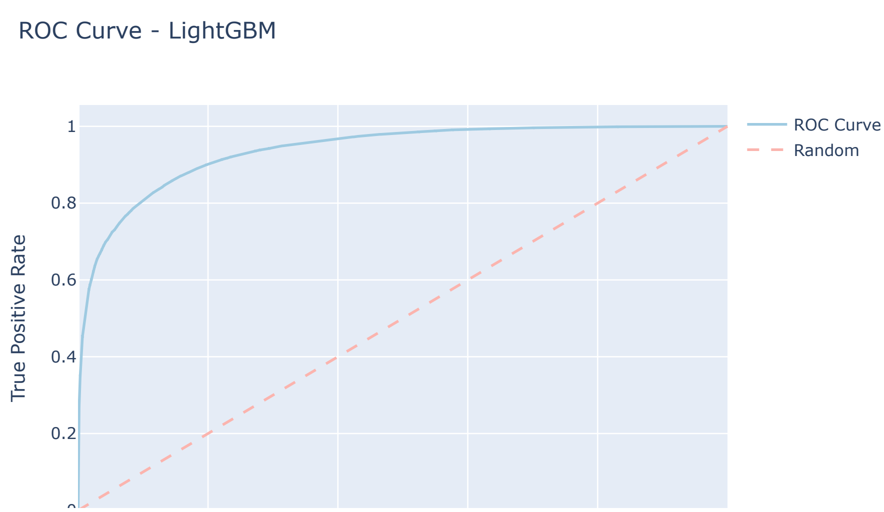


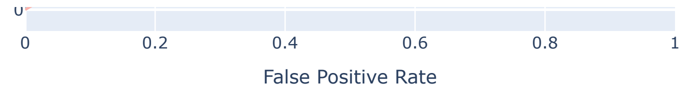




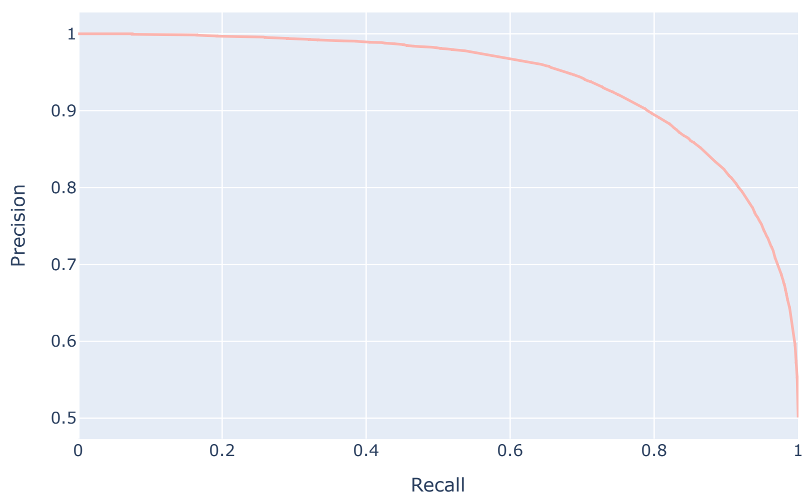



```
=== LightGBM ===
ROC AUC: 0.938 | PR AUC: 0.942 | F1: 0.855
Classification report:
               precision    recall  f1-score   support

           0       0.85      0.87      0.86     19962
           1       0.87      0.85      0.86     20038

    accuracy                           0.86     40000
   macro avg       0.86      0.86      0.86     40000
weighted avg       0.86      0.86      0.86     40000

Confusion matrix:
 [[17334  2628]
 [ 3104 16934]]
```

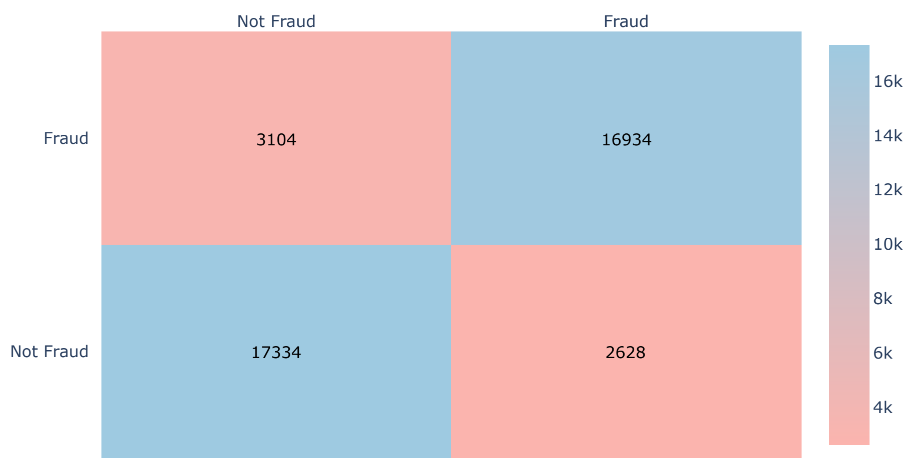




Precision-Recall Curve - LightGBM



Confusion Matrix - LightGBM



```
from sklearn.linear_model import LogisticRegression
import plotly.graph_objs as go
import plotly.figure_factory as ff
from sklearn.metrics import (
    roc_auc_score, average_precision_score, f1_score,
    precision_score, recall_score,
    precision_recall_curve, roc_curve, confusion_matrix, classification_report
)
import pandas as pd

PASTEL_BLUE = "#9ecae1"
PASTEL_PINK = "#fbb4ae"

models = {
    'Logistic Regression': LogisticRegression(max_iter=500, class_weight='balanced', solver='lbfgs'),
    'Random Forest': best_rf,
    'LightGBM': best_lgbm
}
fitted_models = {}
results = []

for name, model in models.items():
    #--- Fit model if not already fitted
    if not hasattr(model, "classes_"):
        model.fit(X_train, y_train)
    fitted_models[name] = model

    #--- Predict and score
    y_pred = model.predict(X_test)
    y_proba = model.predict_proba(X_test)[:,1]
    auc = roc_auc_score(y_test, y_proba)
    pr_auc = average_precision_score(y_test, y_proba)
    f1 = f1_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    cm = confusion_matrix(y_test, y_pred)
    tn, fp, fn, tp = cm.ravel()
    print(f"\n=== {name} ===")
    print("ROC AUC: {:.3f} | PR AUC: {:.3f} | F1: {:.3f}".format(auc, pr_auc, f1))
    print("Classification report:\n", classification_report(y_test, y_pred))
    print("Confusion matrix:\n", cm)
    results.append({
        "Model": name,
        "ROC AUC": auc,
        "PR AUC": pr_auc,
        "F1": f1,
        "Precision": precision,
        "Recall": recall,
        "False Positives": fp,
        "False Negatives": fn
    })

    #--- ROC Curve
    fpr, tpr, _ = roc_curve(y_test, y_proba)
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=fpr, y=tpr, mode='lines', name='ROC Curve', line=dict(color=PASTEL_BLUE)))
    fig.add_trace(go.Scatter(x=[0,1], y=[0,1], mode='lines', name='Random', line=dict(dash='dash', color=PASTEL_PINK)))
    fig.update_layout(title=f"ROC Curve - {name}", xaxis_title="False Positive Rate", yaxis_title="True Positive Rate", width=70
    fig.show()

    #--- PR Curve
    precision_arr, recall_arr, _ = precision_recall_curve(y_test, y_proba)
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=recall_arr, y=precision_arr, mode='lines', name='PR Curve', line=dict(color=PASTEL_PINK)))
    fig.update_layout(title=f"Precision-Recall Curve - {name}", xaxis_title="Recall", yaxis_title="Precision", width=700, height
    fig.show()

    #--- Confusion Matrix visualization
    z = cm
    x = ['Not Fraud', 'Fraud']
    y_ = ['Not Fraud', 'Fraud']
    fig = ff.create_annotated_heatmap(
        z=z, x=x, y=y_, colorscale=[[0, PASTEL_PINK], [1, PASTEL_BLUE]], showscale=True,
        annotation_text=z, hoverinfo="z"
    )
    fig.update_layout(title=f"Confusion Matrix - {name}", width=700, height=500)
```