

BDM 3014

AI Project Mid-point Report

Fraud Detection

To:

Mr. Bhavik Gandhi

Group No. 3:

Chibuike Okoroama

Eduardo Jr Morales

Eduardo Roberto Williams Cascante

Ezgi Tanyeli

Flora Mae Villarin

Haldo Jose Somoza Solis

Jumoke Yekeen


Marzieh Mohammadi Kokaneh

Modupeola Omodunni Oyatokun

Lambton College, Mississauga

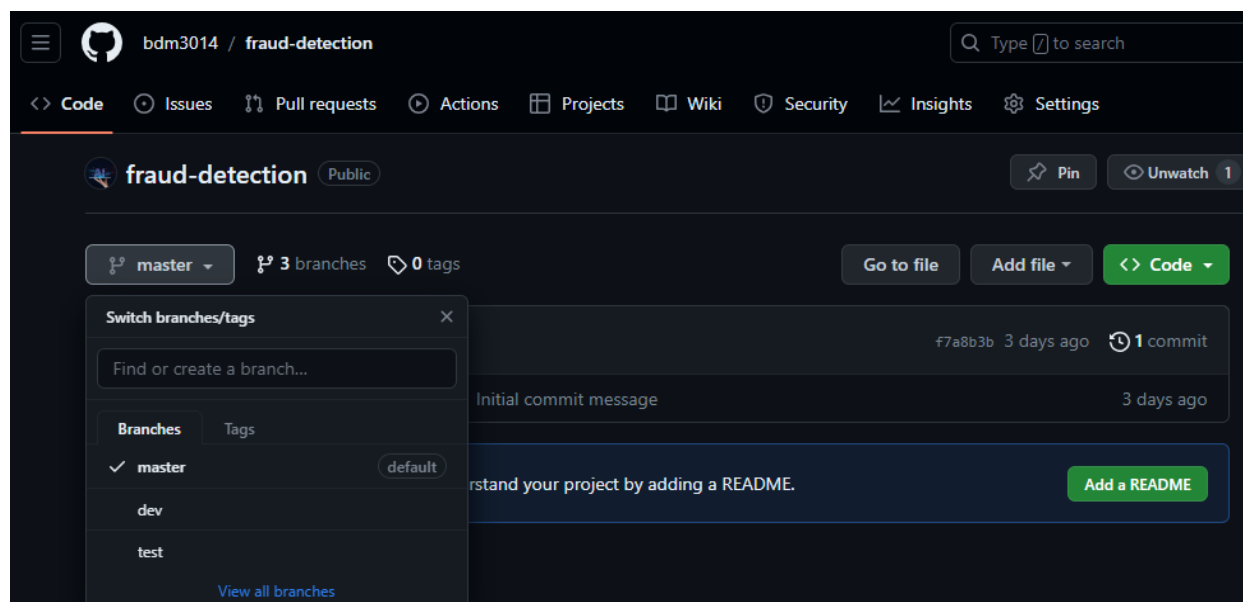
Nov 2023

Git Repository

 GitHub is used for tracking changes in our code, promoting collaboration among multiple developers with its version control capabilities. Our GitHub repository, located at <https://github.com/bdm3014/fraud-detection>, serves as the central hub for our project. It stores our code on the internet, guaranteeing accessibility from any location. The platform's functionalities, such as branching, pull requests, and continuous integration, enhance our well-organized development process.

Branches:

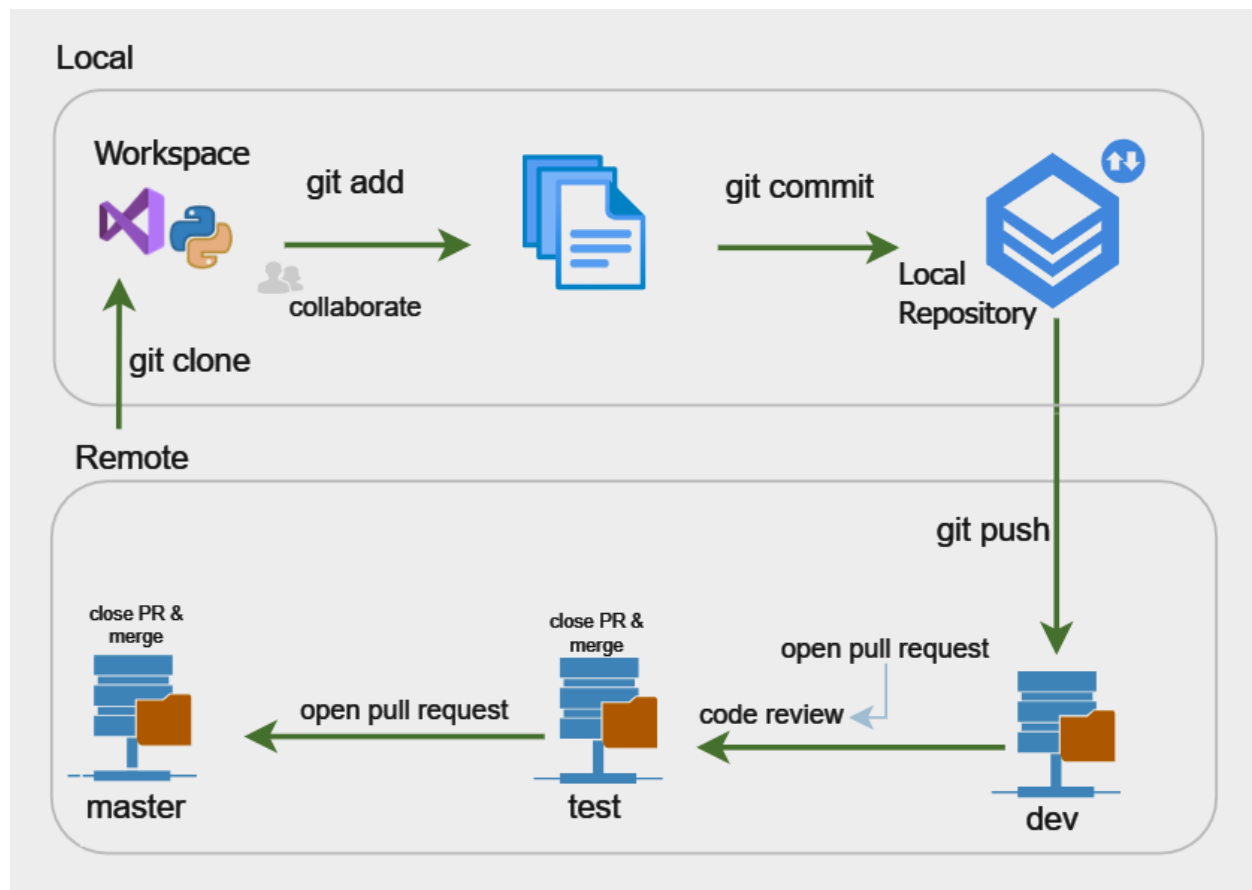
- master
 - production-ready code
 - protected with the requirement to open a pull request
- test
 - code that has been tested and reviewed
 - protected with the requirement to open a pull request and code review
- dev
 - development or working in progress code by developers



Prerequisites:

- Download and Install git – version control system: <https://git-scm.com/downloads>
- Visual Studio Code - <https://code.visualstudio.com/> or a code editor of your choice.
- GitHub Account - <https://github.com/>

Workflow:



Step-by-step GitHub workflow for effective code management and collaboration:

1. Clone Repository:
 - Clone the GitHub repository to a local machine or workspace using source control or the command: `git clone`
<https://github.com/bdm3014/fraud-detection.git>
2. Switch to Dev-Branch:
 - Change to the development branch to start working on new features or bug fixes using the command: `git checkout dev`
3. Start working or make necessary changes to the code as needed, ensure developers test the code on their individual local machines to identify issues early and collaborate with the team during the coding process.
4. Regular Stage and Commit Changes:

- Make small, frequent commits with clear messages to facilitate effective change tracking
 - Stage the changes by executing the command: `git add .` and then commit the modifications with a descriptive message using: `git commit -m "description of changes made."`
5. Push Changes to Dev-Branch:
- Push the committed changes to the dev branch: `git push origin dev`
6. Open a Pull Request:
- Visit the GitHub repository in your web browser.
 - Navigate to the "Pull Requests" tab.
 - Click on "New Pull Request".
 - Set the base branch to "test" and compare the branch to "dev".
 - Create the pull request.
7. Collaborate Code Review:
- Team members review the changes in the pull request.
 - Address and make any necessary adjustments based on feedback.
8. Approval Process:
- Obtain approval from at least one team member before merging.
9. Merge into Test Branch:
- Once the pull request is approved, close the PR and merge the changes into the test branch.
10. Master Branch Release:
- A pull request and approval are still required before push to the master branch.
 - This branch will be used for production release on another platform to bring our application live.

Ground Rules:

1. Merge into the master and test branches exclusively through pull requests, necessitating approval after a thorough code review. Direct code pushes to these branches are discouraged.
2. Prior to merging any pull request, a comprehensive code review must be conducted. A minimum of one approved review is essential before proceeding with the merge.

3. When committing changes, ensure your message clearly describes the alterations made in the commit.
4. Before committing changes, review your repository's status to assess the modifications thoroughly.

dev

Commits on Nov 12, 2023

Entry HTML interface ...
eduwil committed 2 days ago

Commits on Nov 9, 2023

Add files via upload
Jummylawal committed 5 days ago

Delete data-preprocessing/data-cleaning.py
Jummylawal committed 5 days ago

Commits on Nov 8, 2023

data-cleaning.py ...
Jummylawal committed last week

Delete data-preprocessing directory
Jummylawal committed last week

Create data-cleaning.py ...
Jummylawal committed last week

Early Stage of Web Application
haldosomoza committed last week

update file
mae committed last week

Data Preprocessing

The primary focus of this project is on fraud detection using gradient boosting algorithms (XGBoost). The specific functionality and details of the fraud detection process found in the sections of the code are provided here.

First we briefly explain the features and libraries that were used for this project. Below is the list of libraries and figure 1 is a screenshot of the code on VSCode.

1. FEATURES EXPLANATION FOR IEEE ONLINE FRAUD

Transaction Table

TransactionDT: timedelta from a given reference datetime (not an actual timestamp)

TransactionAMT: transaction payment amount in USD

ProductCD: product code, the product for each transaction

card1 - card6: payment card information, such as card type, card category, issue bank, country, etc.

addr: address

dist: distance

P_ and (R__) emaildomain: purchaser and recipient email domain

C1-C14: counting, such as how many addresses are found to be associated with the payment card, etc. The actual meaning is masked.

D1-D15: timedelta, such as days between previous transaction, etc.

M1-M9: match, such as names on card and address, etc.

Vxxx: Vesta engineered rich features, including ranking, counting, and other entity relations.

Categorical Features:

ProductCD

card1 - card6

addr1, addr2

P_emaildomain

R_emaildomain

M1 - M9

Identity Table

Variables in this table are identity information – network connection information (IP, ISP, Proxy, etc) and digital signature (UA/browser/os/version, etc) associated with transactions. They're collected by Vesta's fraud protection system and digital security partners. (The field names are masked and pairwise dictionary will not be provided for privacy protection and contract agreement)

Categorical Features:

DeviceType

DeviceInfo

id_12 - id_38

2. LIBRARIES

Data Manipulation Libraries:

Pandas: Used for data manipulation and analysis.

Numpy: Provides support for large, multi-dimensional arrays and matrices, along with mathematical functions to operate on these arrays.

Matplotlib.pyplot and seaborn: Used for data visualization.

Date and Time Libraries:

Datetime: Provides classes for working with dates and times.

USFederalHolidayCalendar from pandas.tseries.holiday: Utilized for handling US federal holidays.

Machine Learning Libraries:

Sklearn.preprocessing: Provides functions for preprocessing data before applying machine learning models.

xgboost: An optimized gradient boosting library.

lightgbm: A gradient boosting framework that uses tree-based learning algorithms.

catboost: An open-source gradient boosting on decision trees library.

Model Evaluation Libraries:

classification_report and confusion_matrix from sklearn.metrics: Used for evaluating classification models.

roc_auc_score from sklearn.metrics: Computes the area under the receiver operating characteristic curve (AUC-ROC).

Cross-Validation Libraries:

KFold and StratifiedKFold from sklearn.model_selection: Used for cross-validation.

Plotly and Cufflinks for Interactive Visualization:

chart_studio.plotly, plotly.graph_objs, plotly.tools, plotly.offline, cufflinks, and plotly.figure_factory: Utilized for creating interactive plots and visualizations.

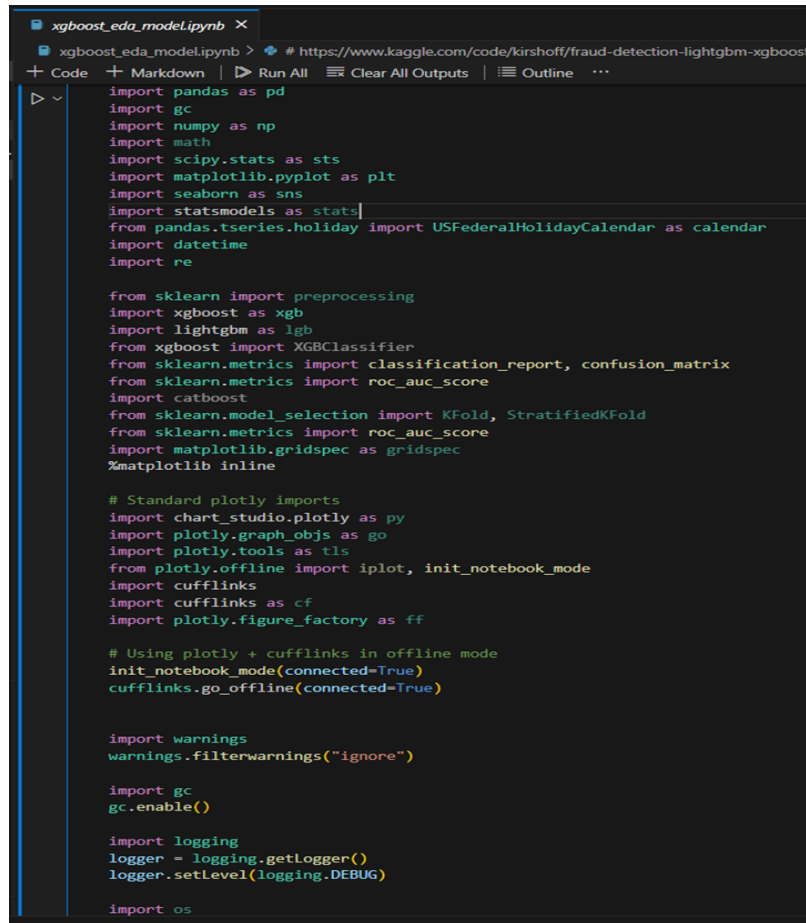
Other Utilities:

warnings: Used to control warning messages.

gc (garbage collector): Used to perform automatic memory management.

logging: Configure logging settings.

os: Provides a way to interact with the operating system.



```
import pandas as pd
import gc
import numpy as np
import math
import scipy.stats as sts
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels as stats
from pandas.tseries.holiday import USFederalHolidayCalendar as calendar
import datetime
import re

from sklearn import preprocessing
import xgboost as xgb
import lightgbm as lgb
from xgboost import XGBClassifier
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.metrics import roc_auc_score
import catboost
from sklearn.model_selection import KFold, StratifiedKFold
from sklearn.metrics import roc_auc_score
import matplotlib.gridspec as gridspec
%matplotlib inline

# Standard plotly imports
import chart_studio.plotly as py
import plotly.graph_objs as go
import plotly.tools as tls
from plotly.offline import iplot, init_notebook_mode
import cufflinks
import cufflinks as cf
import plotly.figure_factory as ff

# Using plotly + cufflinks in offline mode
init_notebook_mode(connected=True)
cufflinks.go_offline(connected=True)

import warnings
warnings.filterwarnings("ignore")

import gc
gc.enable()

import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)

import os
```

Fig 1

3. FEATURE CONVERSION

Furthermore, we aim to prepare and format both Identity and transaction data for further analysis. It defines relevant columns, reads the data into a DataFrame, and formats the date-related features. Sorting the DataFrame based on the transaction date facilitates time-based analysis. Fig 2 is the code snippet as shown below.

Here's a breakdown of the Transaction dataset data conversion code:

Column Definition:

Defines two lists (cols_t and cols_v) representing column names for the dataset. cols_t includes transaction-related features, and cols_v includes features labeled with 'V' followed by a number.

Creates a dictionary (types_v) specifying the data type for 'V' features as float32.

Lists columns (`columns_to_convert_to_object`) that need to be converted to the 'object' data type.

Data Reading:

Reads the data from a CSV file (`train_transaction.csv`) into a DataFrame (`transaction`).

Uses specific columns (`cols_t`, `'isFraud'`, `cols_v`) and custom data types.

Sets the 'TransactionID' column as the index.

Date Formatting:

Defines a start date (`START_DATE`) and generates a date range from October 1, 2017, to January 1, 2019.

Utilizes the calendar module to identify US holidays within the date range.

Creates a new 'DT' column by converting 'TransactionDT' values into datetime format.

Converts the 'DT' column to a DatetimeIndex object.

Sorts the DataFrame based on the 'DT' column.

Transaction Dataset

```

# Importing transaction data
# We are standardizing the column types in accordance with the data definition.

# Define column names for the dataset
cols_t = ['TransactionID', 'TransactionDT', 'TransactionAmt',
          'ProductCD', 'card1', 'card2', 'card3', 'card4', 'card5', 'card6',
          'addr1', 'addr2', 'dist1', 'dist2', 'P_emaildomain', 'R_emaildomain',
          'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9', 'C10', 'C11',
          'C12', 'C13', 'C14', 'D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7', 'D8',
          'D9', 'D10', 'D11', 'D12', 'D13', 'D14', 'D15', 'M1', 'M2', 'M3', 'M4',
          'M5', 'M6', 'M7', 'M8', 'M9']

# Generate column names for the 'V' features (V1 to V339)
cols_v = ['V'+str(x) for x in range(1, 340)]

# Define data types for the 'V' features as float32
types_v = {c: 'float32' for c in cols_v}

# Specify the columns that need to be converted to the 'object' data type
columns_to_convert_to_object = ['ProductCD', 'card1', 'card2', 'card3', 'card4', 'card5', 'card6',
                                'addr1', 'addr2', 'P_emaildomain', 'R_emaildomain', 'M1', 'M2', 'M3', 'M4',
                                'M5', 'M6', 'M7', 'M8', 'M9']

# Read the data from the CSV file into a DataFrame (train)
transaction = pd.read_csv(r'C:\Fraud_Data\data\train_transaction.csv',
                          usecols=cols_t+['isFraud']+cols_v,
                          dtype={**types_v, **{col: 'object' for col in columns_to_convert_to_object}}, index_col='TransactionID')

# The 'TransactionDT' feature represents a timedelta from a specific reference datetime, rather than an actual timestamp.
# It measures the time elapsed since the reference datetime in a timedelta format.

# Getting a real format of transaction date

# Predefined start date
START_DATE = datetime.datetime.strptime('2017-11-30', '%Y-%m-%d')

# Define the date range
dates_range = pd.date_range(start='2017-10-01', end='2019-01-01')
us_holidays = calendar().holidays(start=dates_range.min(), end=dates_range.max())

# Create 'DT' column using the 'TransactionDT' column
transaction['DT'] = transaction['TransactionDT'].apply(lambda x: (START_DATE + datetime.timedelta(seconds=x)))

# Convert 'DT' column to 'DatetimeIndex' object
transaction['DT'] = pd.to_datetime(transaction['DT'])

# Sorting the DataFrame based on the 'DT' column
transaction = transaction.sort_values(by='DT')

```

Fig 2

The selected columns of Identity dataset contain information about the device used for the transaction (DeviceInfo, DeviceType) and various identity-related features (id_38, id_37, ..., id_01). This identity data will be merged with the transaction data based on the 'TransactionID' index to enrich the overall dataset. See fig 3.

Column Definition:

Defines a list (cols_t) containing column names for the identity dataset. These columns seem to represent various identity-related features.

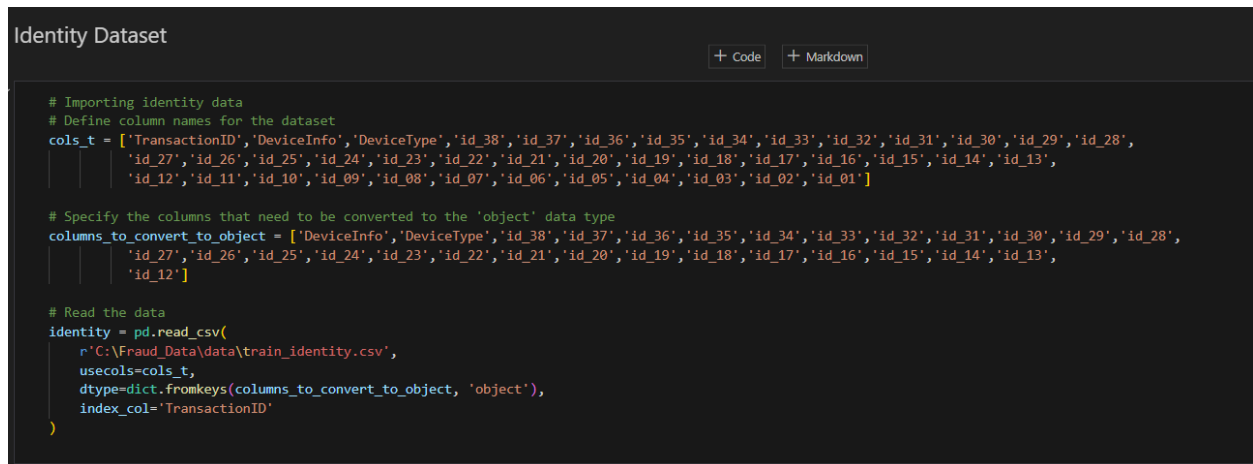
Defines another list (columns_to_convert_to_object) specifying columns that need to be converted to the 'object' data type.

Data Reading:

Reads the data from a CSV file (train_identity.csv) into the identity DataFrame.

Uses specific columns (cols_t) and converts selected columns to the 'object' data type as specified by columns_to_convert_to_object.

Sets the 'TransactionID' column as the index for the identity DataFrame.



```
Identity Dataset
+ Code + Markdown

# Importing identity data
# Define column names for the dataset
cols_t = ['TransactionID', 'DeviceInfo', 'DeviceType', 'id_38', 'id_37', 'id_36', 'id_35', 'id_34', 'id_33', 'id_32', 'id_31', 'id_30', 'id_29', 'id_28',
          'id_27', 'id_26', 'id_25', 'id_24', 'id_23', 'id_22', 'id_21', 'id_20', 'id_19', 'id_18', 'id_17', 'id_16', 'id_15', 'id_14', 'id_13',
          'id_12', 'id_11', 'id_10', 'id_09', 'id_08', 'id_07', 'id_06', 'id_05', 'id_04', 'id_03', 'id_02', 'id_01']

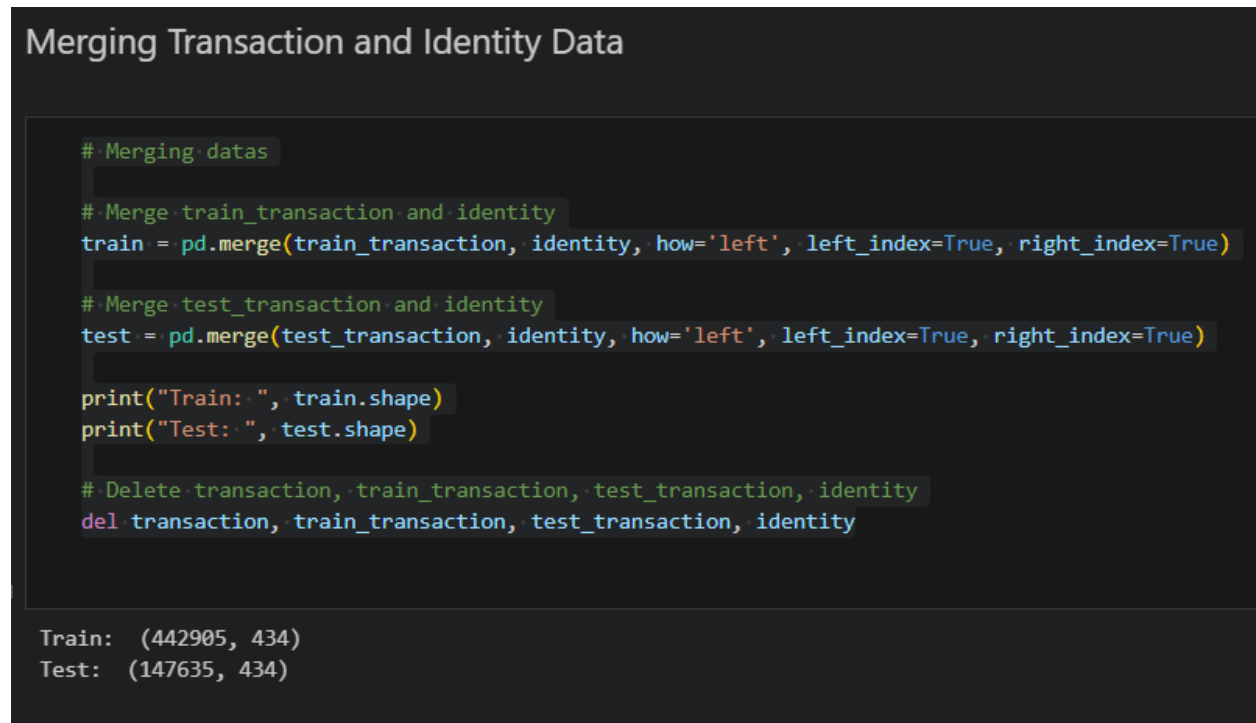
# Specify the columns that need to be converted to the 'object' data type
columns_to_convert_to_object = ['DeviceInfo', 'DeviceType', 'id_38', 'id_37', 'id_36', 'id_35', 'id_34', 'id_33', 'id_32', 'id_31', 'id_30', 'id_29', 'id_28',
                                'id_27', 'id_26', 'id_25', 'id_24', 'id_23', 'id_22', 'id_21', 'id_20', 'id_19', 'id_18', 'id_17', 'id_16', 'id_15', 'id_14', 'id_13',
                                'id_12']

# Read the data
identity = pd.read_csv(
    r'C:\Fraud_Data\data\train_identity.csv',
    usecols=cols_t,
    dtype=dict.fromkeys(columns_to_convert_to_object, 'object'),
    index_col='TransactionID'
)
```

Fig 3

4. MERGE DATASETS

Merging the transaction and identity data based on the 'TransactionID' allows for a more comprehensive dataset that includes both transaction details and identity-related information. Fig 4



```
Merging Transaction and Identity Data

# Merging datas

# Merge train_transaction and identity
train = pd.merge(train_transaction, identity, how='left', left_index=True, right_index=True)

# Merge test_transaction and identity
test = pd.merge(test_transaction, identity, how='left', left_index=True, right_index=True)

print("Train: ", train.shape)
print("Test: ", test.shape)

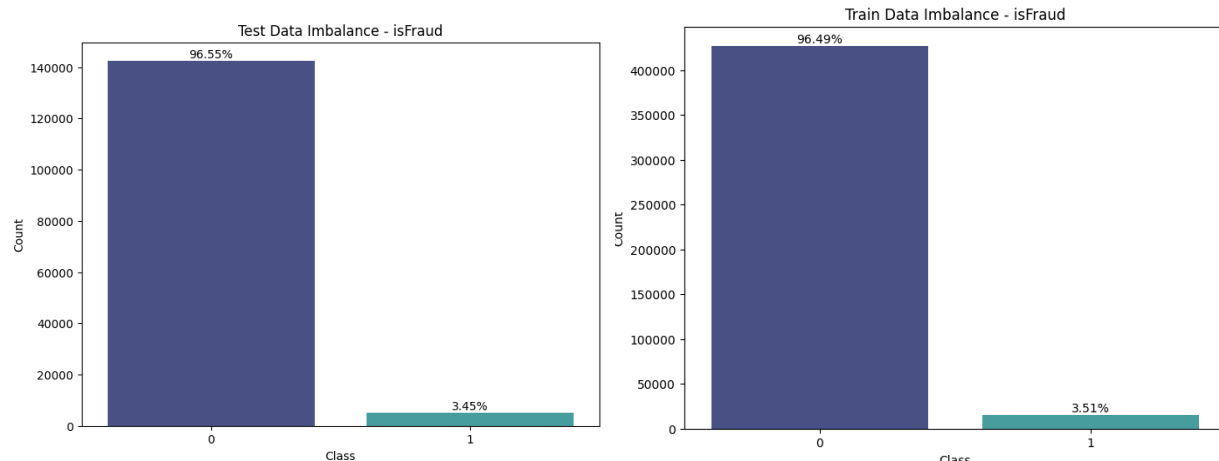
# Delete transaction, train_transaction, test_transaction, identity
del transaction, train_transaction, test_transaction, identity

Train: (442905, 434)
Test: (147635, 434)
```

Fig 4

6. TARGET VARIABLE DISTRIBUTION

The original dataset exhibits a substantial class imbalance, with the majority of transactions being non-fraudulent (96.49% in the training set and 96.55% in the test set). This imbalance, where only 3.51% of transactions are labeled as fraudulent in the training set and 3.45% in the test set, poses challenges for accurate predictive modeling and analysis. The risk of overfitting is highlighted, emphasizing the potential for models to inaccurately assume that the majority of transactions are not fraudulent. The primary objective is to develop models capable of accurately identifying patterns associated with fraudulent activities. Addressing this class imbalance is crucial to prevent models from being misled by the prevalence of non-fraudulent instances. The text underscores the importance of considering this imbalance when developing and evaluating predictive models for fraud detection. Fig 5



7. HANDLING MISSING VALUES

We aim to improve the efficiency and effectiveness of the predictive modeling process by removing columns that are likely to provide little or no information for the task at hand. This helps in reducing noise, improving computational efficiency, and potentially enhancing the performance of machine learning models.

Both Train and Test data sets have up to 45% missing values, as shown in fig 6 with the code.

```

Datasets have too much missing values.

# Total missing values of the train data
missing_count = train.isnull().sum()
cell_counts = np.product(train.shape)
missing_sum = missing_count.sum()
print ("%", (round(missing_sum/cell_counts,2)) * 100)

6]
% 45.0

# Total missing values of the test data
missing_count = test.isnull().sum()
cell_counts = np.product(test.shape)
missing_sum = missing_count.sum()
print ("%", (round(missing_sum/cell_counts,2)) * 100)

7]
% 45.0

```

Fig 6

We focused on identifying and dropping columns from the train and test DataFrames based on three criteria: having only one distinct value, having more than 90% null values, or being dominated by a single category that represents more than 90% of the column. The goal is to remove columns that are likely to be uninformative or redundant for predictive modeling, specifically in the context of fraud detection.

```
# Drop columns by using 3 criteria:
# 1. If a column only has only one distinct value
# 2. If a column has more than 90% null values
# 3. If one of the categories in a column dominates more than 90% of the column
# we are looking for these criterias in train, then dropping the columns from both train and test

# Initialize lists to store columns to be dropped based on different criteria
one_value_cols, many_null_cols, big_top_value_cols = [], [], []

# Iterate through only the train DataFrame
for df in [train]:
    # Identify columns with only one distinct value
    one_value_cols += [col for col in df.columns if df[col].nunique() == 1]

    # Identify columns with more than 90% null values
    many_null_cols += [col for col in df.columns if df[col].isnull().sum() / df.shape[0] > 0.9]

    # Identify columns where a single value dominates more than 90%
    big_top_value_cols += [col for col in df.columns if df[col].value_counts(dropna=False, normalize=True).values[0] > 0.9]

# Combine the lists of columns to be dropped, removing duplicates using set
cols_to_drop = list(set(one_value_cols + many_null_cols + big_top_value_cols))

# Check if 'isFraud' is in the list of columns to be dropped, and remove it if present
if 'isFraud' in cols_to_drop:
    cols_to_drop.remove('isFraud')

# Drop the identified columns from the train DataFrame
train = train.drop(cols_to_drop, axis=1)
test = test.drop(cols_to_drop, axis=1)

# Print the number of features that are going to be dropped for being considered useless
print(f'{len(cols_to_drop)} features are going to be dropped for being useless')
```

66 features are going to be dropped for being useless

Fig 7

HANDLING OUTLIERS

Our code addresses potential outliers by capping the transaction amount at 6450.97 for certain transactions in the training set, which may help mitigate the impact of extreme values on the model. The strip plots provide a visual representation of how the distribution of transaction amounts differs between fraudulent and non-fraudulent

transactions in both the training and testing datasets. The horizontal red line at 6500 serves as a visual aid.

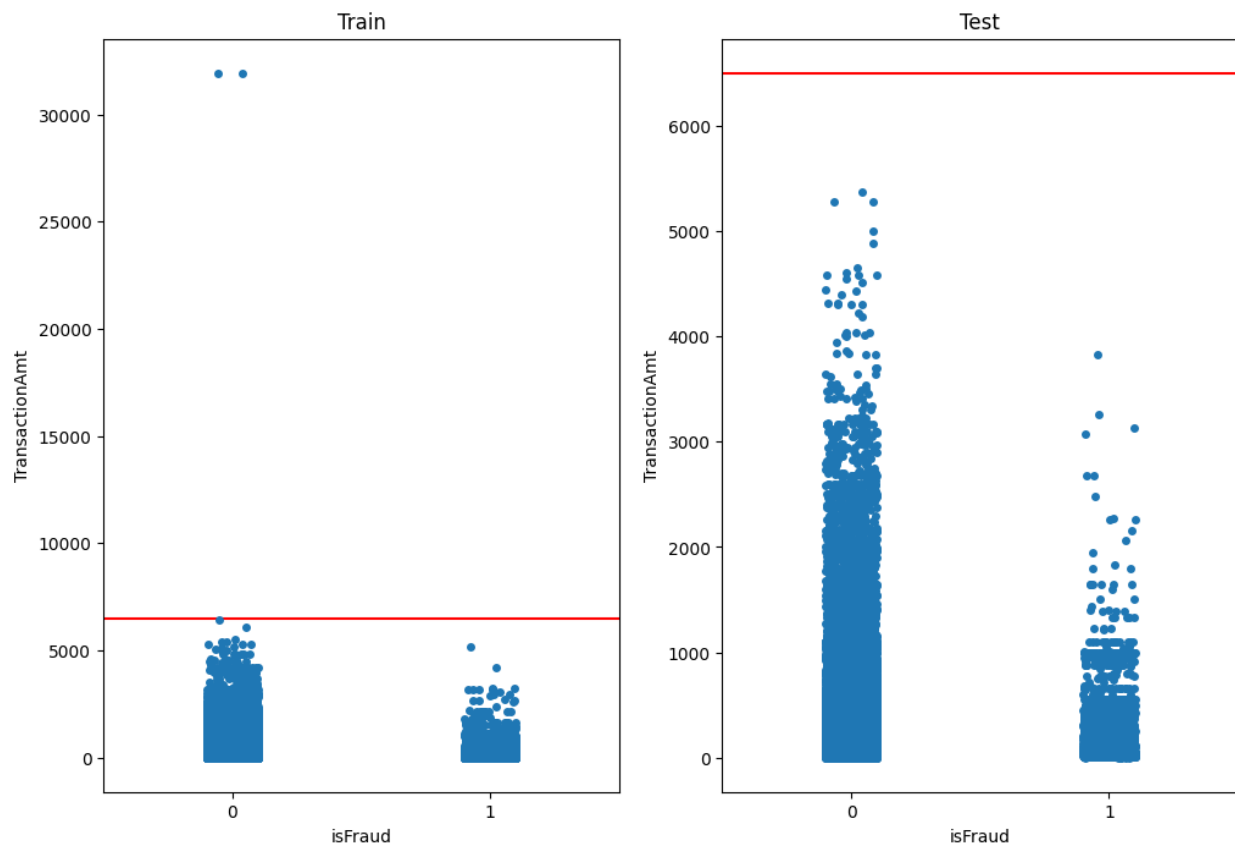


Fig 8

8. FEATURES EXPLORATION

We explored some of the features to understand their relationship with the isFraud column.

Firstly we start by looking into the distance column which is a numeric feature. The code below analyzes and visualizes the characteristics of the 'dist1' column in the dataset. The strip plots help understand the distribution of 'dist1' concerning the target variable 'isFraud'. Additionally, we identify and address potential outliers by capping the 'dist1' values for specific transactions in the training set. We aim to manage extreme values that might impact the performance of machine learning models.


```
dist1 : The distance (numeric)

column_details(regex='^dist', df=df)

Unique Values of the Features:
feature: DTYPE, NUNIQUE, NULL_RATE

dist1: float64, 1744, 154.87
[0.000e+00 1.000e+00 2.000e+00 ... 7.136e+03 8.081e+03      nan]

plt.figure(figsize=(12,8))
plt.subplot(121)
sns.stripplot(y='dist1', x='isFraud', data=train)
plt.axhline(5000, color='red')
plt.title('Train')

plt.subplot(122)
sns.stripplot(y='dist1', x='isFraud', data=test)
plt.axhline(5000, color='red')
plt.title('Test')

max_dist1 = train['dist1'][train['dist1'] < 5500].max()
print(f"The maximum dist1 below 5500 is: {max_dist1}")

The maximum dist1 below 5500 is: 5431.0

There are outliers in dist1 with isFraud == 0. These are above 5500. We find the maximum amount below 5500 ( 5431 ). We will capped the Train with 5432 below code.

# Identify transactions with isFraud == 0 and dist1 5431
condition = train['dist1'] > 5431

# Cap the dist1 at 5432 for the identified transactions
train.loc[condition, 'dist1'] = 5431

train['dist1'].max()

5431.0
```

Fig 9

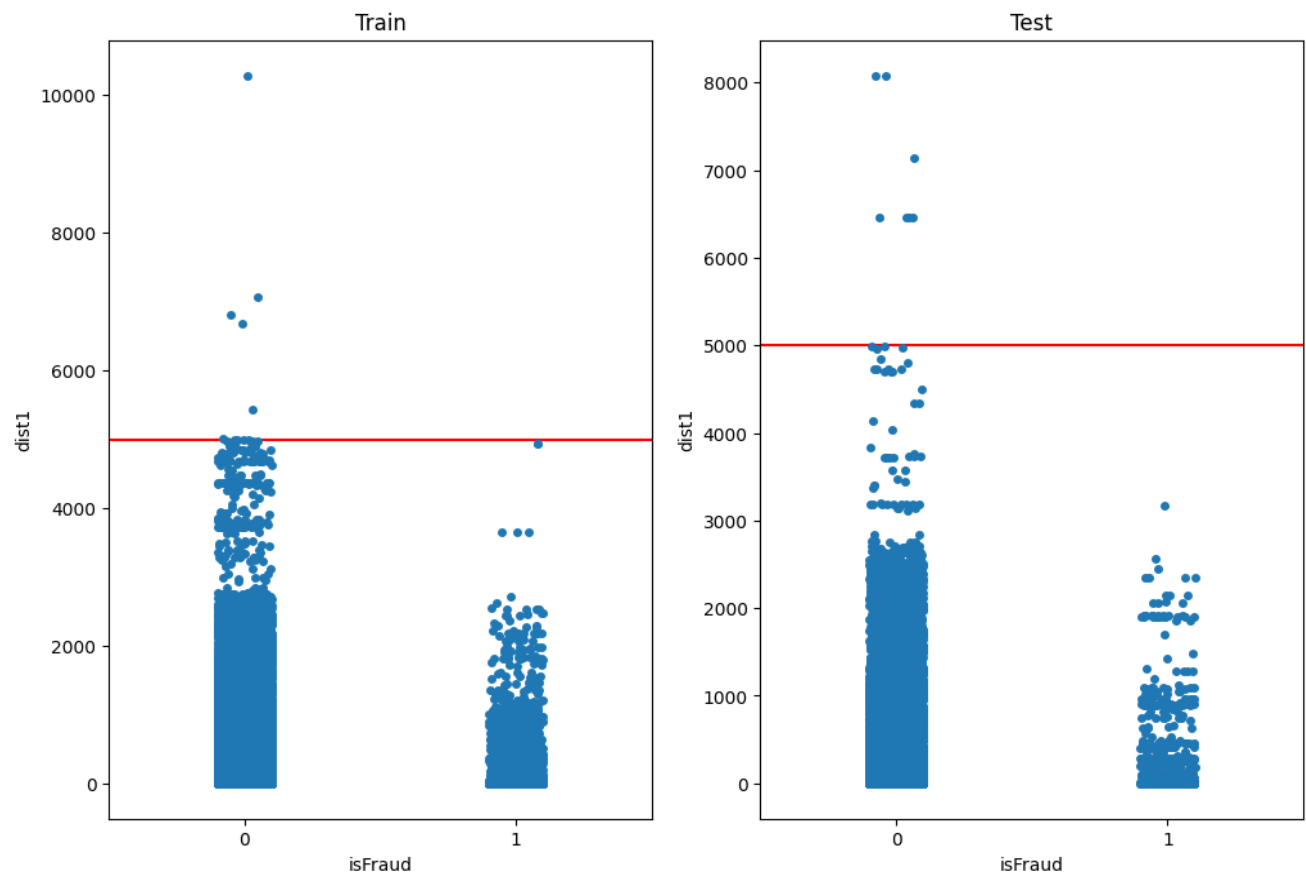


Fig 10

9. FEATURE ENGINEERING

One hot encoding is a common technique used to handle nominal categorical data by converting each category into a binary column. While label encoding can be suitable for ordinal categorical data, it's not the best choice for nominal categorical data due to its inherent ordering. Frequency encoding and target encoding provides an effective solution for nominal categorical data.

Due to having lots of distinct values in card1,2,3,5 columns(if we use target encoding most of the values will be too small near to 0 or will be 0) it is better to use frequency encoding by replacing each category with its observed frequency. We will use the target encoder for card4(4 distinct values) and card6 (2 distinct values).

Target encoding introduces information about the target variable, while frequency encoding captures the frequency of each value in a categorical column. These engineered features can potentially improve the performance of machine learning models by providing more meaningful representations of the categorical variables. The original columns are dropped after encoding to avoid redundancy in the dataset.

The code performs feature engineering, specifically target encoding and frequency encoding, on certain categorical columns in the train and test datasets as shown in the fig 9.

```
# List of columns to process (Number of unique values below 200) (taking into account the average of the target feature in train set)
columns_to_process = ['card3', 'card4', 'card5', 'card6']

# Loop through each column
for col in columns_to_process:
    # Calculate target encoding for the current column
    temp_dict = train.groupby([col])['isFraud'].agg(['mean']).to_dict()['mean']
    # Create a new column with the target encoding values
    train[f'{col}_target_encoded'] = train[col].replace(temp_dict)
    test[f'{col}_target_encoded'] = test[col].replace(temp_dict)

# Frequency encoding for card1 (Number of unique values above 200)
self_encode_False = ['card1', 'card2']
train, test = frequency_encoding(train, test, self_encode_False, self_encoding=False)

# List of original columns to drop
columns_to_drop = ['card1', 'card2', 'card3', 'card4', 'card5', 'card6']

# Drop the original columns
train.drop(columns_to_drop, axis=1, inplace=True)
test.drop(columns_to_drop, axis=1, inplace=True)
```

Fig 9

Secondly, we process, and engineer features related to address information. `addr1` and `addr2`: Address information related to the transaction (nominal categorical). Probably `addr1` - subzone / `addr2` - Country. Combines 'addr1' and 'addr2' columns into a new 'addr' column by concatenating them as strings, separated by an underscore. This new column represents a combination of subzones and countries.

Therefore, It includes combining columns to create a new feature, exploring the distribution of address-related columns, and applying frequency encoding to capture the frequency of each value in these columns. The original columns are dropped to streamline the dataset and potentially improve the performance of machine-learning models by reducing redundancy and noise.

The code in fig 10 is focused on processing and engineering features related to address information in the train and test datasets.

```
Probably addr1 - subzone / addr2 - Country

for df in [train, test]:
    column_details(regex='addr', df=df)

# First ten most frequent address1 values in train
print("Unique Subzones =", train['addr1'].nunique())

print("\nFirst Ten Address-2")
print('-----')
train.addr1.value_counts().head(9)

print("Unique Countries =", train['addr2'].nunique())
print("\nFirst Ten Address-2")
print('-----')
train.addr2.value_counts().head(9)

# By combining addr1 and addr2, we create a new addr column
for df in [train, test]:
    # Combine 'addr2' and 'addr1' columns as strings, separated by an underscore
    df['addr'] = (df['addr2'].astype(str) + '_' + df['addr1'].astype(str)).replace({'nan_nan': np.nan})

print("Unique Addresses =", train['addr'].nunique())
print("\nFirst Ten Addresses")
print('-----')
train.addr.value_counts().head(9)

# unique values for address columns in train
train['addr1'].nunique(), train['addr2'].nunique(), train['addr'].nunique()

(321, 69, 409)

# unique values for address columns in train
test['addr1'].nunique(), test['addr2'].nunique(), test['addr'].nunique()
```

```

# Frequency encoding for addr columns
self_encode_False=['addr']
train, test = frequency_encoding(train, test, self_encode_False, self_encoding=False)

self_encode_False=['addr1']
train, test = frequency_encoding(train, test, self_encode_False, self_encoding=False)

self_encode_False=['addr2']
train, test = frequency_encoding(train, test, self_encode_False, self_encoding=False)

# List of original columns to drop
columns_to_drop = ['addr', 'addr1', 'addr2']

# Drop the original columns
train.drop(columns_to_drop, axis=1, inplace=True)
test.drop(columns_to_drop, axis=1, inplace=True)

```

Fig 10

Next, we looked into the email domains and found that

P_emaildomain : categoric, 56 uniques. It's possible to make a subgroup feature from it or a general group.

R_emaildomain : categoric, 59 uniques. It's possible to make a subgroup feature from it or a general group.

Two highest fraud activity domains based on hosting:

es (Spain) Category: The category associated with Spain (es) has a higher fraud rate compared to other categories (%10). This suggests that transactions originating from Spain may require closer scrutiny in terms of fraud detection.

com (United States) Category: The category associated with the United States (com) has a significantly higher fraud rate compared to other categories (%8.32). This may imply the need for careful examination of transactions coming from this geographical region. Fig 11 shows the code.

```

# es has 10% fraud rate
fraud_rates = train.groupby('R_emaildomain_2')['isFraud'].mean().reset_index()
fraud_rates.rename(columns={'isFraud': 'FraudRate'}, inplace=True)
fraud_rates.sort_values(by='FraudRate', ascending=False).head()

```

	R_emaildomain_2	FraudRate
6	es	0.100000
2	com	0.083204
3	com.mx	0.017406
7	fr	0.015015
8	net	0.009868

Fig 11

However, when exploring further with the R_emaildomain, in fig 12 we found the below

Two highest fraud activity domains:

ProtonMail Category: The ProtonMail category has a significantly higher fraud rate compared to other categories (%95). This indicates that transactions originating from ProtonMail may pose a higher risk of fraud. Scrutiny of transactions associated with such accounts could be crucial.

Mail Category: This general 'mail' category exhibits a higher fraud rate compared to other categories (%36). This suggests that transactions from general email providers should be examined with caution.

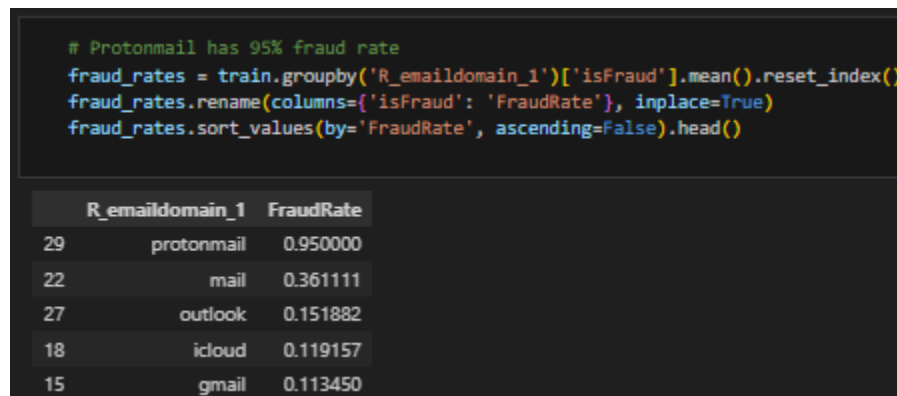


Fig 12

However we found that R_emaildomain_2 and P_emaildomain_2 are highly correlated, there we dropped P_emaildomain_2 and kept P_emaildomain_1, R_emaildomain_2 and R_emaildomain_1. In the fig 13 we can see the correlation of these features and which columns were dropped.

```

# correlation between R_emaildomain_2 and P_emaildomain_2 - train
cramers_v(train.R_emaildomain_2,train.P_emaildomain_2)

0.8926752413893435

# correlation between R_emaildomain_2 and P_emaildomain_2 - test
cramers_v(test.R_emaildomain_2,test.P_emaildomain_2)

0.8813311624433039

# correlation between R_emaildomain_1 and P_emaildomain_1 - train
cramers_v(train.R_emaildomain_1,train.P_emaildomain_1)

0.6060523585208291

# correlation between R_emaildomain_1 and P_emaildomain_1 - test
cramers_v(test.R_emaildomain_1,test.P_emaildomain_1)

0.5873334446544154

# R_emaildomain_2 and P_emaildomain_2 are categorically 90% correlated. So we dropped P_emaildomain_2.
# R_emaildomain_1, P_emaildomain_1, R_emaildomain_2 are staying
train = train.drop(['R_emaildomain', 'P_emaildomain', 'P_emaildomain_2'], axis=1)
test = test.drop(['R_emaildomain', 'P_emaildomain', 'P_emaildomain_2'], axis=1)

# Target Encoding For R_emaildomain_1, P_emaildomain_1, R_emaildomain_2 (taking into account the average of the target feature in train
temp_dict = train.groupby(['R_emaildomain_1'])['isFraud'].agg(['mean']).to_dict()['mean']
train['R_emaildomain_1_target_encoded'] = train['R_emaildomain_1'].replace(temp_dict)
test['R_emaildomain_1_target_encoded'] = test['R_emaildomain_1'].replace(temp_dict)

temp_dict = train.groupby(['P_emaildomain_1'])['isFraud'].agg(['mean']).to_dict()['mean']
train['P_emaildomain_1_target_encoded'] = train['P_emaildomain_1'].replace(temp_dict)
test['P_emaildomain_1_target_encoded'] = test['P_emaildomain_1'].replace(temp_dict)

temp_dict = train.groupby(['R_emaildomain_2'])['isFraud'].agg(['mean']).to_dict()['mean']
train['R_emaildomain_2_target_encoded'] = train['R_emaildomain_2'].replace(temp_dict)
test['R_emaildomain_2_target_encoded'] = test['R_emaildomain_2'].replace(temp_dict)

# List of original columns to drop
columns_to_drop = ['R_emaildomain_1', 'R_emaildomain_2', 'P_emaildomain_1']

# Drop the original columns
train.drop(columns_to_drop, axis=1, inplace=True)
test.drop(columns_to_drop, axis=1, inplace=True)

```

Fig 13

On the other hand, we used heatmap to explore columns that will not be used for further analysis. In this case, the C1 to C14 columns were explored for multicollinearity. The

threshold of 75% was set in the code below as shown in fig 14 that any column above 75% will be dropped. Result showed that only features C1 and C5 were left. The same technique was used on D1 to D15 features.

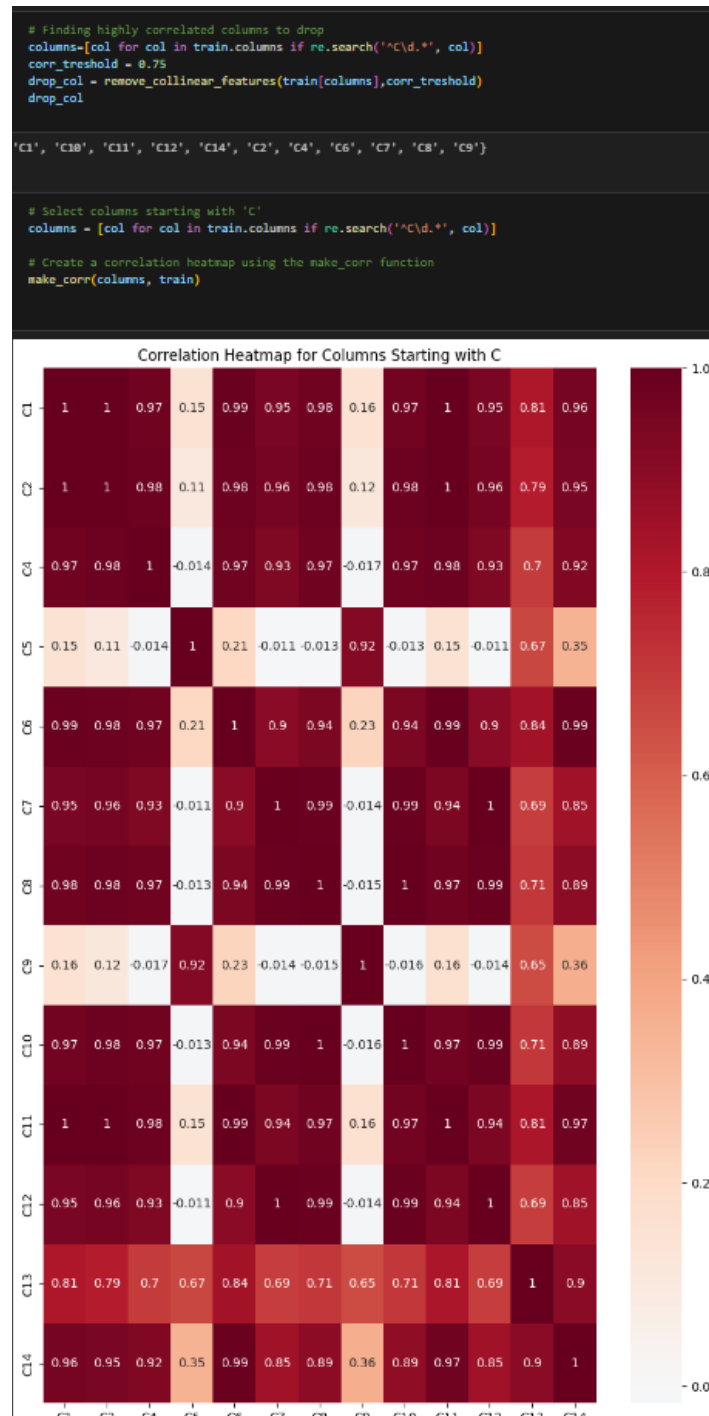


Fig 14

For the V1-V339 features, We identified redundancy and correlation among the 'V' columns, and dropped correlated columns with a correlation coefficient (r) greater than 0.75. This process resulted in retaining only 69 independent 'V' columns. Fig 15 shows the code for the above explanation.

```
# removing high correlated variables (222 eliminated)
corr_treshold = 0.75
drop_col = remove_collinear_features(train[colums],corr_treshold)
len(drop_col)

222

# dropping redundant Vs
train = train.drop(drop_col, axis=1)
test = test.drop(drop_col, axis=1)

# remaining Vs lenght (64 vars)
colums=[col for col in train.columns if re.search('^V\d+', col)]
len(colums)
```

Fig 15

As for the id features (1 to 11) numeric, The column_details function helps understand the characteristics of columns with names matching the specified pattern. The subsequent steps involve identifying and removing highly correlated variables to address multicollinearity, and the heatmap provides a visual representation of the correlation structure within the specified subset of columns. However, there are no correlation between id_10 and id_11.

```
id_1 ... id_11 (numeric)

column_details(regex='id_(1|2|3|4|5|6|7|8|9|10|11)$', df=train)

# removing high correlated variables
corr_treshold = 0.75
drop_col = remove_collinear_features(train[colums],corr_treshold)
len(drop_col)

0

plt.figure(figsize=(10,10))
sns.heatmap(train[colums].sample(frac=0.2).corr(),annot=False,cmap="RdBu_r")
```

Fig 16

As for the id_12 to id_38, categorical data type, we aim to preprocess related to the 'id_' columns in the dataset. It includes handling rare values, synchronizing datasets, calculating Cramers V values, removing collinear features, and performing target or frequency encoding based on distinct counts. Therefore, improve the quality of the data and prepare it for training machine learning models. Fig 17 and 18 shows the code written to achieve several results and fig 19 is the Cramer's V values.

```
id_12...id_38 (nominal categoric)

column_details(regex='id_(12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32|33|34|35|36|37|38)', df=train)

# Synchronize train and test
for col in ['id_12', 'id_13', 'id_14', 'id_15', 'id_17', 'id_19', 'id_30', 'id_31', 'id_32', 'id_34', 'id_36', 'id_37', 'id_38']:
    old_versions_col = set(train[col].unique()) - set(test[col].unique())
    new_versions_col = set(test[col].unique()) - set(train[col].unique())
    test[col] = test[col].apply(lambda x: np.nan if x in new_versions_col else x)
    train[col] = train[col].apply(lambda x: np.nan if x in old_versions_col else x)

rareIds = []
# Specify the range of columns you want to process
columns_to_process = ['id_12', 'id_13', 'id_14',
                      'id_15', 'id_17', 'id_19', 'id_30', 'id_31', 'id_32', 'id_34', 'id_36', 'id_37', 'id_38']

for col in columns_to_process:
    for k, df in enumerate([train, test]):
        rareIds = df[col].value_counts()
        rareIds = [value for value in rareIds.index if rareIds[value] < 3]
        rareIds += rareIds

    print(f"('TEST' if k else 'TRAIN')")
    print(f"Number of unique in {col}: {df[col].nunique()}")
    print(f"Number of unique values with frequency less than 3 in {col}: {len(rareIds)}\n")

rareIds = set(rareIds)

# Low frequency cats will change into nans
columns_to_process = [i for i in ['id_12', 'id_13', 'id_14',
                                'id_15', 'id_17', 'id_19', 'id_30', 'id_31', 'id_32', 'id_34', 'id_36', 'id_37', 'id_38']]

for col in columns_to_process:
    for df in [train, test]:
        df[col] = df[col].apply(lambda x: np.nan if x in rareIds else x)

id_columns = train.loc[:, 'id_12': 'id_38']

# Create an empty DataFrame to store the results
cramers_v_matrix = pd.DataFrame(index=id_columns.columns, columns=id_columns.columns, dtype=float)

# Fill in the Cramers V values for each pair of columns
for col1 in id_columns.columns:
    for col2 in id_columns.columns:
        crammers_v_matrix.loc[col1, col2] = crammers_v[id_columns[col1], id_columns[col2]]

# Heatmap
plt.figure(figsize=(12, 10))
sns.heatmap(cramers_v_matrix, cmap='RdBu_r', annot=True, fmt=".2f", linewidths=.5)
plt.title('Cramers V Matrix Heatmap')
plt.show()
```

Fig 17

```
# We want to remove collinear features in 'id_' columns with a threshold of 0.75
id_columns = [col for col in train.columns if re.search('^id_(12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32|33|34|35|36|37|38)', col)]
drop_columns = identify_collinear_categorical_features(train, id_columns, threshold=0.75)

# Display the columns to drop
print("Columns to drop:", drop_columns)

Columns to drop: ['id_29', 'id_20', 'id_28', 'id_16', 'id_37', 'id_33', 'id_35']

# Remove the collinear features
train = remove_collinear_categorical_features(train, drop_columns)
test = remove_collinear_categorical_features(test, drop_columns)

# Select only the columns that start with 'id_' and end with a number between 12 and 38
remaining_features = [col for col in train.columns if col.startswith('id_') and col[3:].isdigit() and 12 <= int(col[3:]) <= 38 and col not in drop_columns]

for col in remaining_features:
    distinct_count = train[col].nunique()

    if distinct_count < 200:
        # Target encode using target mean
        temp_dict = train.groupby([col])['isFraud'].agg(['mean']).to_dict()['mean']
        train[col + '_target_encoded'] = train[col].replace(temp_dict)
        test[col + '_target_encoded'] = test[col].replace(temp_dict)
    else:
        # Frequency encode
        train, test = frequency_encoding(train, test, columns=[col])

# Display the first few rows of the modified data
print(train.head())
print(test.head())

# Drop the original columns
train.drop(remaining_features, axis=1, inplace=True)
test.drop(remaining_features, axis=1, inplace=True)
```

Fig 18

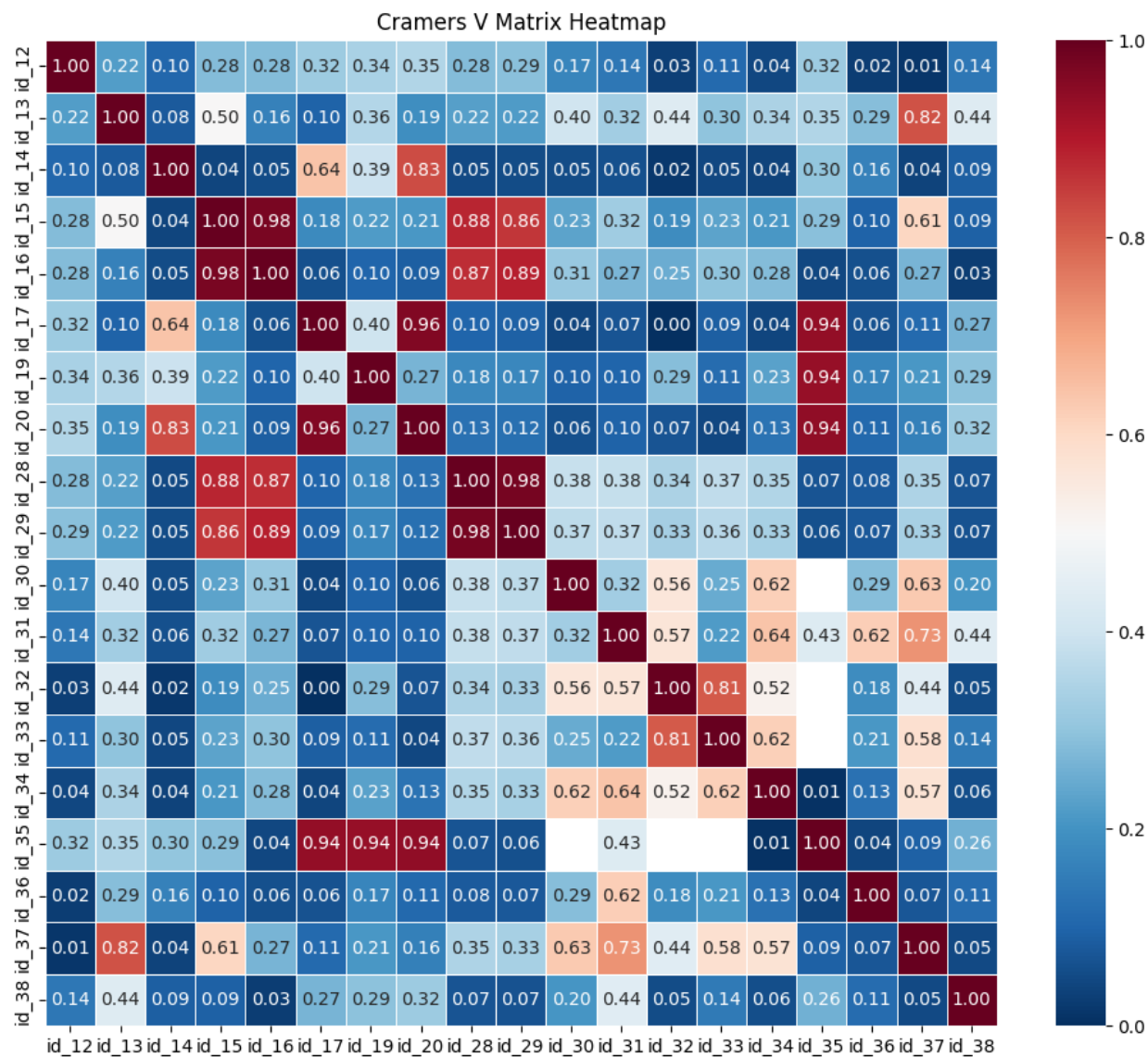


Fig 19

Several transactions were done with different types of device types, therefore it is imperative we understand if the use of some of these devices can lead to potential fraud.

We analyzed the 'DeviceType' column, by calculating fraud rates for mobile and desktop, performed target encoding based on these fraud rates, and then dropped the original 'DeviceType' column. So, we transformed categorical variables into a format that is more informative for the models. Target encoding helps to capture the relationship between the categorical feature and the target variable in order to improve model performance.

```
DeviceType (nominal categoric)

for df in [train, test]:
    column_details(regex='DeviceType', df=df)

Unique Values of the Features:
feature: DTYPE, NUNIQUE, NULL_RATE

DeviceType: object, 2, %74.62
['desktop' 'mobile' nan]

Unique Values of the Features:
feature: DTYPE, NUNIQUE, NULL_RATE

DeviceType: object, 2, %80.75
['desktop' 'mobile' nan]

plot_col('DeviceType', df=train)

fraud_rates_device_type = train.groupby('DeviceType')['isFraud'].mean().reset_index()
fraud_rates_device_type.rename(columns={'isFraud': 'FraudRate'}, inplace=True)
fraud_rates_device_type.sort_values(by='FraudRate', ascending=False, inplace=True)

print(fraud_rates_device_type)

DeviceType  FraudRate
1    mobile    0.098887
0    desktop    0.061458

#target encoding
col = 'DeviceType'
temp_dict = train.groupby([col])['isFraud'].agg(['mean']).to_dict()['mean']
train[col+'_target_encoded'] = train[col].replace(temp_dict)
test[col+'_target_encoded'] = test[col].replace(temp_dict)

# Drop the original column
train.drop('DeviceType', axis=1, inplace=True)
test.drop('DeviceType', axis=1, inplace=True)
```

Fig 19

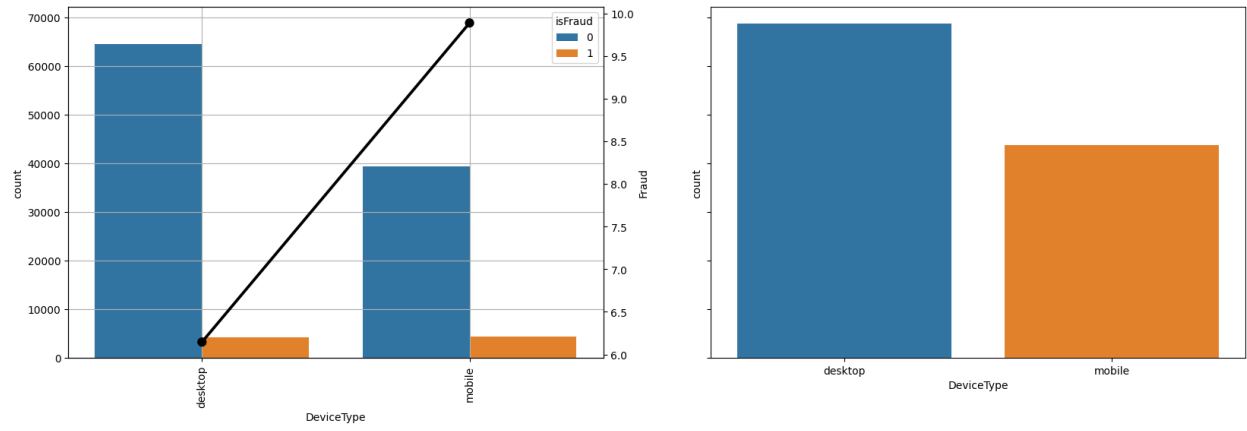


Fig 20

As for the DeviceInfo, we used frequency encoding to transform the categorical values into numerical representations based on their frequency of occurrence. We are dealing with categorical columns with a high cardinality (many unique values). The frequency encoding helps to understand the information about the distribution of each unique value in the 'DeviceInfo' column without introducing a large number of new columns (as one-hot encoding would). After encoding, the original 'DeviceInfo' column is dropped to keep the dataset concise.

```
DeviceInfo(nominal categoric)

for df in [train, test]:
    column_details(regex='DeviceInfo', df=df)

Unique Values of the Features:
feature: DTYPE, NUNIQUE, NULL_RATE

DeviceInfo: object, 1583, %78.49
['0PAJ5' '0PJ2' '0PM92' ... 'verykools5035' 'xs-Z47b7VqTMxs' nan]

Unique Values of the Features:
feature: DTYPE, NUNIQUE, NULL_RATE

DeviceInfo: object, 962, %84.16
['2PYB2' '4009F' '4013M Build/KOT49H' '4047A Build/NRD90M'
'4047G Build/NRD90M' '47418' '5010G Build/MRA58K' '5010S Build/MRA58K'
'5011A Build/NRD90M' '5012G Build/MRA58K' '5015A Build/LMY47I'
'5025G Build/LMY47I' '5044A' '5049W Build/NRD90M' '5051A Build/MMB29M'
'5054S Build/LMY47V' '5056A Build/MMB29M' '5056N' '5080A Build/MRA58K'
'5085B Build/MRA58K' '6037B' '6039A Build/LRX22G' '6045I Build/LRX22G'
'6055B' '7048A Build/LRX22G' '7_Plus' '8050G Build/LMY47I'
'8080 Build/LRX21M' '9008A Build/NRD90M' 'A0001' 'A3-A20' 'A463BG'
'A574BL Build/NMF26F' 'A577VL' 'AERIAL' 'AKUS' 'ALCATEL'
'ALCATEL ONE TOUCH 7047A Build/JDQ39' 'ALE-L21 Build/HuaweiALE-L21'
'ALE-L23 Build/HuaweiALE-L23' 'ALP-L09 Build/HUAWEIALP-L09'
'ALP-L09 Build/HUAWEIALP-L09S' 'AM508' 'ANE-LX3 Build/HUAWEIANE-LX3'
'ASUS_A001' 'ASUS_X008D Build/NRD90M' 'ASUS_X00HD Build/NMF26F'
'ASUS_X00ID' 'ASUS_X018D' 'ASUS_Z00AD Build/LRX21V' 'ASUS_Z017D'
'ASUS_Z018DC' 'AX1060' 'AX1070' 'AX820 Build/MRA58K' 'AX821 Build/MRA58K'
...
'rv:46.0' 'rv:47.0' 'rv:48.0' 'rv:50.0' 'rv:51.0' 'rv:52.0' 'rv:55.0'
'rv:56.0' 'rv:57.0' 'rv:58.0' 'rv:59.0' 'rv:60.0' 'rv:61.0'
'verykools5005' 'verykools5524' 'verykools5530 Build/LMY47I' 'vivo' nan]

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

```
# Frequency encoding for DeviceInfo
self_encode_False=['DeviceInfo']
train, test = frequency_encoding(train, test, self_encode_False, self_encoding=False)

# Drop the original column
train.drop('DeviceInfo', axis=1, inplace=True)
test.drop('DeviceInfo', axis=1, inplace=True)
```

Fig 21

Lastly, the Date columns have no usefulness in the model, therefore they are dropped from the dataframe. As we got the number of train columns left, we all used clipboards to copy out their names. Fig 22 shows the code written for this.

```
Removing Date columns

# Date columns are not indicators
train.drop('TransactionDT', axis=1, inplace=True)
train.drop('DT', axis=1, inplace=True)

test.drop('TransactionDT', axis=1, inplace=True)
test.drop('DT', axis=1, inplace=True)

pd.DataFrame(train.columns).to_clipboard()

# We have 128 independent variables 1 dependent variable last.
len(train.columns)

128

info = train.dtypes
info.to_clipboard()
```

Fig 22

10. Pickle

To save our work, we used pickle, to save the current state of the datasets to files so that we can easily load back into memory later. Due to the size of our dataset, it takes time to process most of our code, therefore reprocessing them will after memory usage. However, Pickling allows us to persist the data in a serialized format, preserving its structure and content. In this case, the pickled files are named 'train_3.pkl' and 'test_3.pkl', and they are stored in a specified directory.

```
Pickling Final Train and Test

#pickling datasets
#Save 'train' data to a pickle file named 'train_3.pkl'
train.to_pickle(r'C:\Fraud_Data\data\train_3.pkl')

#save 'test' data to a pickle file named 'test_3.pkl'
test.to_pickle(r'C:\Fraud_Data\data\test_3.pkl')

import pandas as pd
# Read the 'train_3.pkl' pickle file and load it into the 'train' DataFrame
train = pd.read_pickle('./train_3.pkl')

# Read the 'test_3.pkl' pickle file and load it into the 'test' DataFrame
test = pd.read_pickle('./test_3.pkl')
```

Fig 23

The code in fig 24 splits the data set into dependent (isFraud) and independent variables. This will be the criteria at which our model will be trained on to predict whether there is fraud or no fraud.

```
Defining X sets and Y

# Target variable for training set (y_train)
y_train = train['isFraud']

# Independent variables for training set (X_train)
X_train = train.drop(['isFraud'], axis=1)

# Target variable for test set (y_test)
y_test = test['isFraud']

# Independent variables for test set (X_test)
X_test = test.drop(['isFraud'], axis=1)
```

Fig 24

Building the Model

Model Structure

a. Model Algorithm

In the scope of this modeling work, the XGBoost algorithm has been employed. XGBoost (eXtreme Gradient Boosting) is a machine-learning algorithm based on decision trees. The algorithm aims to create a robust model by combining weak learners, typically decision trees. XGBoost focuses on correcting errors from previous trees, adding each tree sequentially. The model makes final predictions by taking the weighted average of the predictions from these trees.

This approach is commonly used to achieve high-performance and competitive results. The loss function of XGBoost is a measure that evaluates the model's errors, and trees are added to minimize this function. This allows the model to learn and generalize patterns in the dataset.

b. Parameters of XGBoost

Parameter	Description	Default Value
gamma	Minimum loss reduction required for further partitioning a leaf node. Larger gamma values lead to a more conservative algorithm.	0
min_child_weight	Minimum sum of instance weight (hessian) required in a child. If the sum of instance weight in a leaf node is less than min_child_weight, no further partitioning is done.	1
learning_rate	Step size shrinkage used in the update to prevent overfitting. A lower learning rate makes the optimization more robust by shrinking the feature weights.	0.3
max_depth	Maximum depth of a tree. Higher values make the model more complex and prone to overfitting. It's a crucial parameter to tune as it directly impacts model complexity.	6

base_score	Cutoff value for initial predictions. It sets the initial prediction to the logarithm of the ratio of the mean positive class frequency to the mean negative class frequency.	0.5
random_state	A number assigned to ensure the reproducibility of the model. If you want reproducible results, set a seed number (an integer) for random_state.	1003
subsample	Subsample ratio of the training instances. Setting it to 0.5 means that XGBoost randomly samples half of the training data before growing trees.	1
colsample_bytree	Subsample ratio of columns when constructing each tree. It's the fraction of features (columns) to be randomly sampled for each tree.	1
max_delta_step	Maximum delta step allowed for each leaf output. If 0, there is no constraint. It might help control the update step, especially in logistic regression when classes are extremely imbalanced.	0
missing	Imputation rule for missing data. It's the default value for imputation of missing values. If set, the tree algorithm will handle missing values during training.	nan
n_estimators	Number of trees used. It represents the number of boosting rounds (iterations) during the training process. A higher number of trees may lead to overfitting.	100
booster	Selected model algorithm. 'gbtree' is commonly used for tree-based models. Other options include 'gblinear' for linear models and 'dart' for dropout regularization.	'gbtree'
reg_alpha	L1 regularization term on weights. Larger values make the model more conservative by encouraging sparsity, meaning pulling weights towards zero.	0

reg_lambda	L2 regularization term on weights. Larger values make the model more conservative by penalizing large weights.	1
num_leaves	Number of leaf nodes to use. More leaves improve accuracy but may lead to overfitting. It's a critical parameter for controlling the complexity of the tree model.	31
min_child_samples	Minimum samples per leaf node. Larger values reduce overfitting by requiring a minimum number of samples in each leaf.	20
feature_fraction	Controls the subsampling of features used for training. It's the fraction of features to be randomly sampled for each boosting round.	1
bagging_fraction	Randomly bags or subsamples training data. It's the fraction of the training data that is randomly sampled for each boosting round.	1
scale_pos_weight	Controls the balance of positive and negative weights. It's useful for unbalanced classes. A typical value is the ratio of negative instances to positive instances.	1
eval_metric	Evaluation metric for validation data. The metric to be monitored during training and used for early stopping.	default according to objective rmse for regression, and logloss for classification, mean average precision for rank:map

c. Feature Importance

Fitted the model with default parameters, utilizing all 127 variables in `X_train`. The only modified parameter was `scale_pos_weight`, addressing the imbalanced dataset. Results were as follows:

First Model:

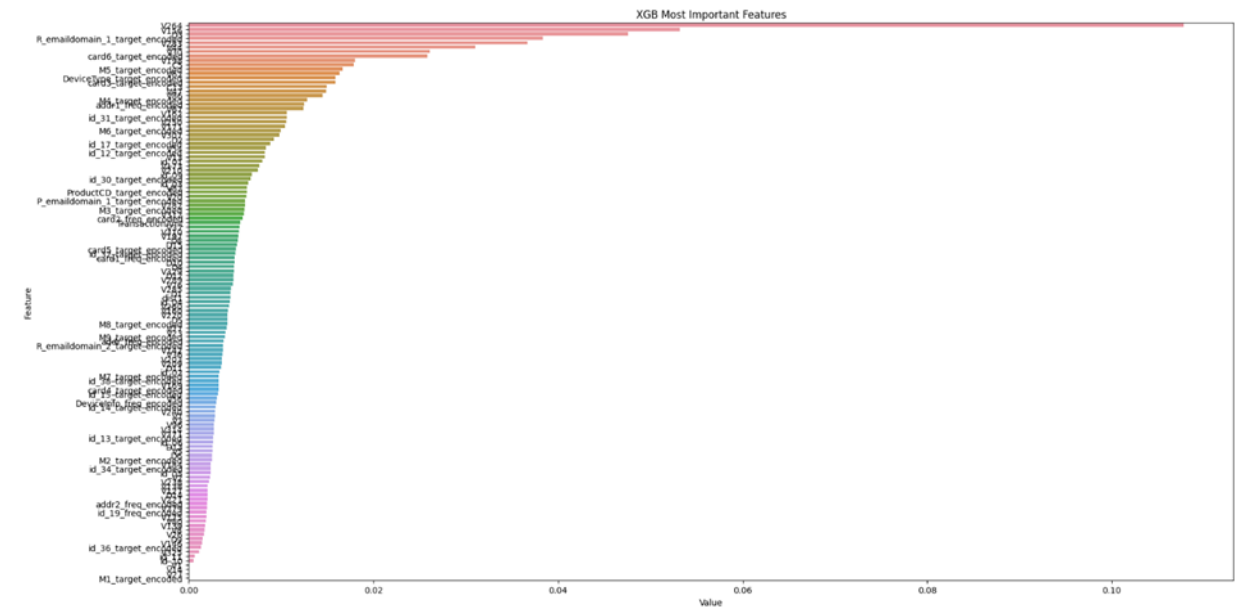
Train AUC: 0.9625262580415187

Test AUC: 0.8849532849516837

	precision	recall	f1-score	support
0	0.98	0.99	0.98	142535
1	0.50	0.42	0.46	5100
accuracy			0.97	147635
macro avg	0.74	0.70	0.72	147635
weighted avg	0.96	0.97	0.96	147635

[[0.98502824 0.01497176]
[0.57686275 0.42313725]]

Then, the feature importance was examined, and the top 50 variables were selected with the highest importance and used for the second model.



The results for the second model were:

Second Model:

Train AUC: 0.9514796162751223

Test AUC: 0.8728808582962423

	precision	recall	f1-score	support
0	0.98	0.98	0.98	142535
1	0.47	0.41	0.44	5100
accuracy			0.96	147635
macro avg	0.73	0.70	0.71	147635
weighted avg	0.96	0.96	0.96	147635

```
[[0.98364612 0.01635388]
 [0.58705882 0.41294118]]
```

Although 77 variables (127 - 50) have been removed, the AUC scores did not decrease significantly. The third model was tried with 30 variables initially, but the AUC scores dropped too much, so without including those results, another try with 40 variables, resulted as bellows

Third Model (Final):

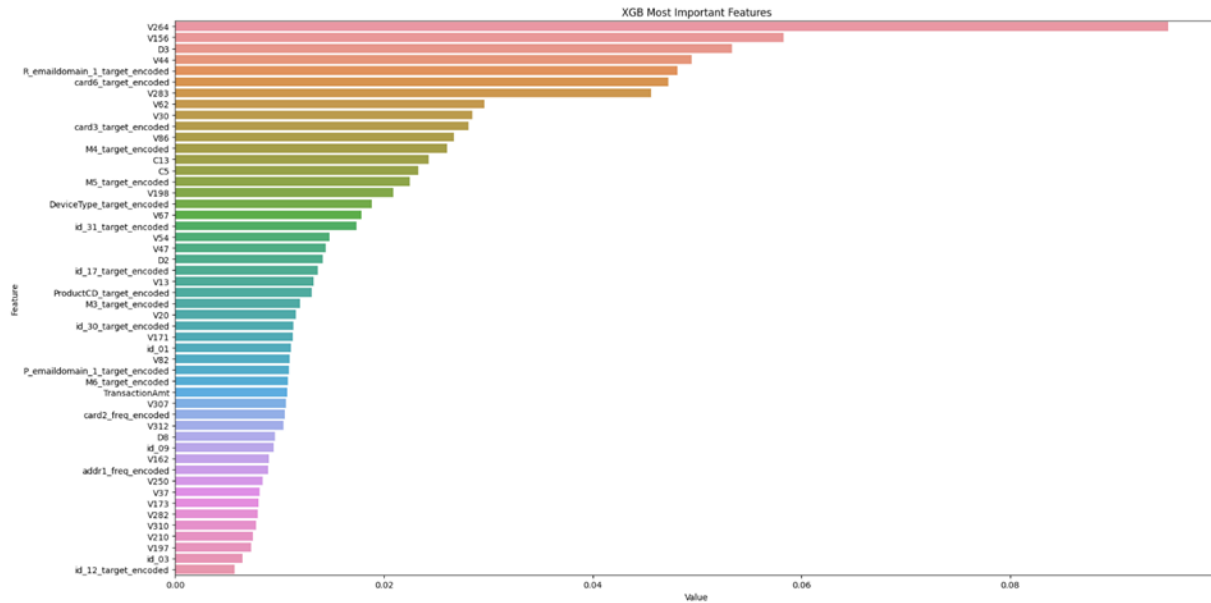
Train AUC: 0.9501872964334457

Test AUC: 0.8720227291955123

	precision	recall	f1-score	support
0	0.98	0.98	0.98	142535
1	0.45	0.42	0.44	5100
accuracy			0.96	147635
macro avg	0.72	0.70	0.71	147635
weighted avg	0.96	0.96	0.96	147635

```
[[0.98182201 0.01817799]
 [0.58019608 0.41980392]]
```

The AUC results for the model with 50 variables were almost the same as the model with 40 variables. Therefore, we decided to finalize the model with 40 variables.



Feature importance plot for first 40 variables

Actions To Take After Evaluation of The Model

1. Monitoring Model Performance:

Initially, our achieved AUC scores were quite high; however, considering the noticeable difference between training and test performance, there seems to be some overfitting. We are considering controlling overfitting for better generalization of the model.

Parameters to improve: colsample_bytree, max_depth, subsample, etc.

2. Variable Selection:

The performance of your final model improved with 40 variables. However, we need to closely examine the relationship between the selected variables' marked importance (feature importance) and model performance. For instance, we will evaluate the importance ranking of variables using methods like SHAP or Permutation Importance.

3. Model Tuning:

So far, we have worked with default parameters. We aim to further enhance performance by conducting model tuning (hyperparameter adjustment).

Parameters to improve: Specifically, adjusting parameters such as `learning_rate`, `max_depth`, `min_child_weight`, and `gamma` could be beneficial.

4. Data Balancing:

The `scale_pos_weight` parameter is a common practice for imbalanced datasets. However, we will also consider other balancing techniques (e.g., sampling methods or trying different algorithms).

Data Insights

Data insights are essential in the field of fraud detection, aiding organizations in recognizing and addressing fraudulent activities. Utilizing data visualization methods enables organizations to derive significant patterns, trends, and anomalies from their transactional data.

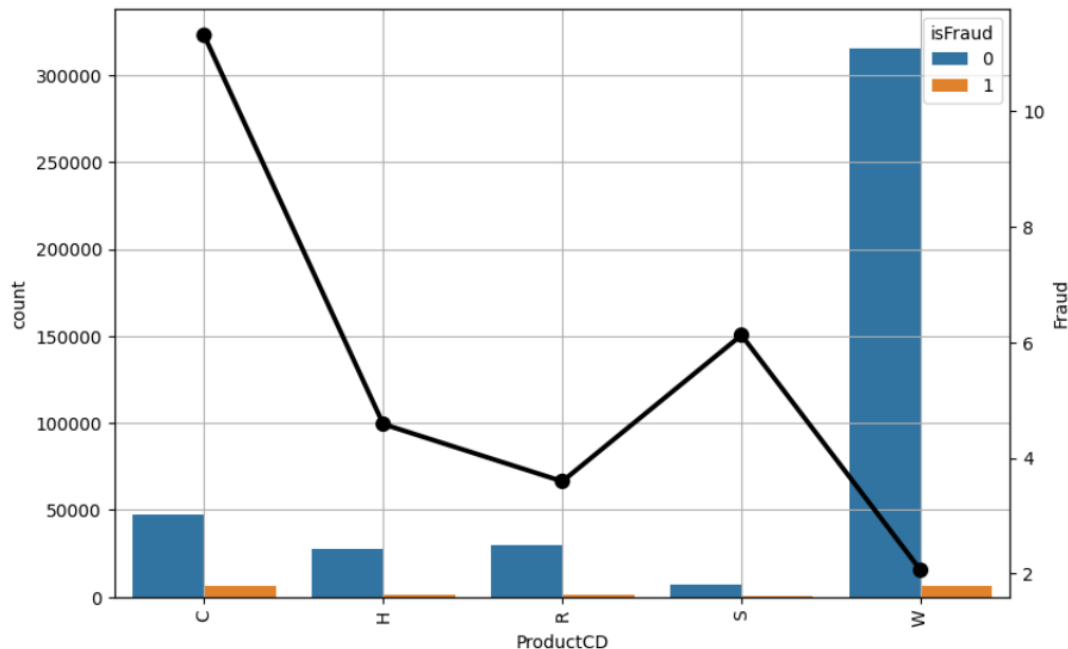
Product vs isFraud Analysis

This explores the correlation between Product Codes (C, H, R, S, W) and instances of fraud (isFraud) within a dataset.

The product codes correspond to different transaction types:

Product Code	Meaning
C	Credit card transactions
H	Debit card or ATM card transactions
R	Charge card transactions
S	Cash transactions
W	Digital wallets or payment applications

By evaluating both the count and percentage distribution of product codes alongside fraudulent and non-fraudulent, the analysis provides insights into transactional behavior. The focus is on highlighting products with the notable trends in fraud occurrences, aiming to inform decision-makers and investigators in strengthening security measures and mitigating risks. The dataset contains a feature named "ProductCD", which represents the product code associated with each transaction.

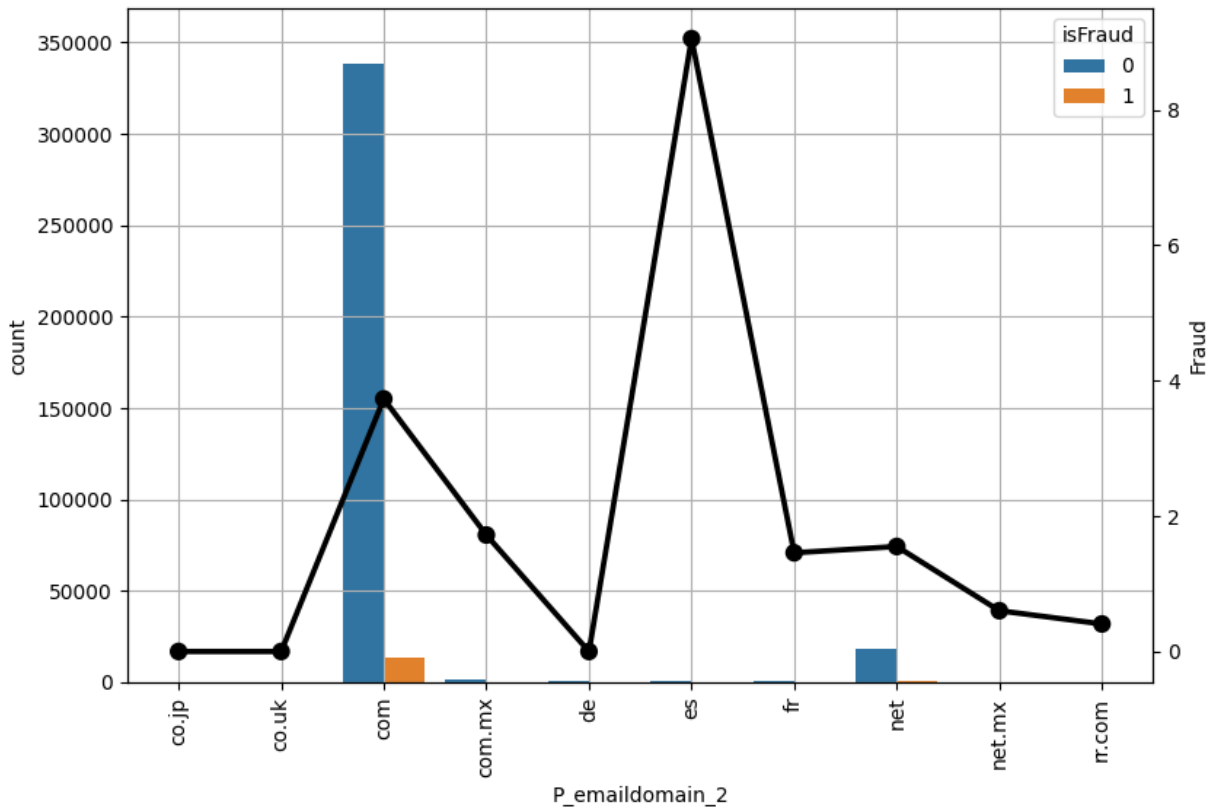


- Overall Distribution:
 - Product W is predominant in the dataset, constituting approximately 72.72% of the total records.
 - Products C, H, R, and S collectively account for the remaining 27.28%.
- isFraud Distribution:
 - For non-fraudulent transactions (isFraud=0), Product W has the highest representation, making up 73.82% of the total.
 - In contrast, for fraudulent transactions (isFraud=1), Product C and W contribute significantly, comprising 38.70% and 42.62%, respectively.
- Potential Impact:
 - Organizations can adjust their plans considering the prominence of Product W, potentially streamlining processes associated with its dominance.
 - The concentration of fraudulent activity in Products C and W highlights the need for targeted fraud prevention measures and resource allocation in these specific areas.
 - These insights assist decision-makers in tailoring strategies to address the unique challenges presented by different products, ultimately improving fraud detection and risk mitigation.

Email Domain vs isFraud Analysis

Discovered essential observations about both Purchaser and Recipient Email Domains; understanding the patterns associated with these domains is crucial for effective fraud detection. This investigates the link between email domains for both purchasers and recipients and instances of fraud (isFraud) within a provided dataset.

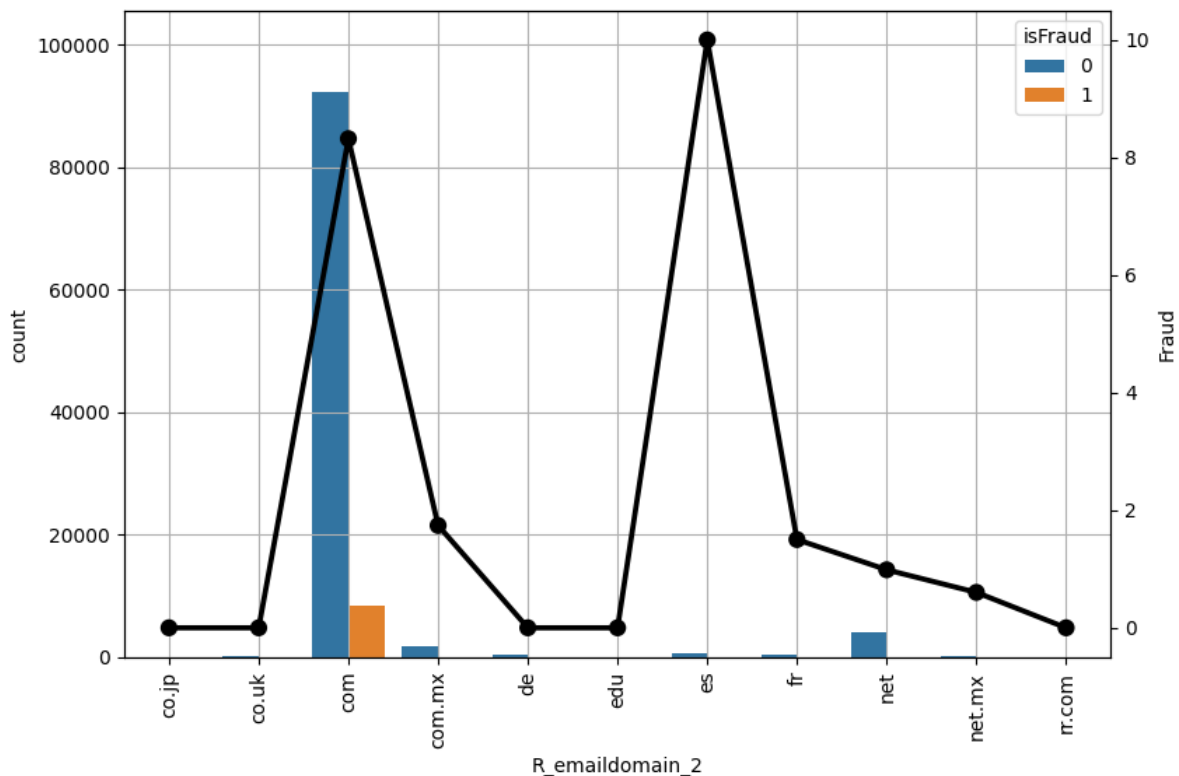
1. Purchase Email Domain:



- Overall Distribution:
 - The dataset's majority consists of transactions associated with the email "com" domain, constituting 93.88% of the total transactions.
 - Other domains, such as "net" and "es," contribute but represent a smaller fraction of the overall distribution.
- isFraud Distribution:
 - Fraudulent transactions are highly concentrated in the "com" domain, constituting a significant 97.11% of the total fraudulent cases.

- Other domains like "net" and "es" contribute to a smaller percentage of fraudulent transactions.

2. Recipient Email Domain:



- Overall Distribution:
 - The overall distribution of email domains in the dataset reveals a predominant presence of the "com" domain, constituting 92.96% of the total transactions.
- isFraud Distribution:
 - For non-fraudulent transactions (isFraud=0), the "com" domain remains prominent, making up 92.50% of the total. In fraudulent transactions (isFraud=1), the "com" domain shows an even higher representation at 98.35%.
- Potential Impact:

- Organizations are recommended to prioritize the "com" domain in their fraud detection efforts, given its significant presence in both non-fraudulent and fraudulent transactions.
- Focusing on prominent email domains, particularly "com," enhances the accuracy of fraud detection by minimizing false positives and negatives, ensuring a more reliable process.

Address vs isFraud Analysis

Understanding patterns and correlations in address values ('addr1' and 'addr2') is crucial for effective fraud detection in financial transactions. This report explores the top values of 'addr1' and 'addr2' associated with the highest fraud rates and examines the combined 'addr' values.

- Overall Distribution:
 - The top 'addr1' values associated with the highest fraud rates are 305.0, 466.0, 471.0, 483.0, and 501.0.
 - The top 'addr2' values contributing to elevated fraud rates are 10.0, 82.0, 46.0, 92.0, and 75.0.
- isFraud Distribution:
 - Transactions with 'addr1' equal to 305.0 have a fraud rate of 66.67%, indicating a notable association with fraudulent activities.
 - 'addr2' value 10.0 stands out with a 100% fraud rate, emphasizing a high likelihood of fraud for transactions with this 'addr2' value.
 - Combined 'addr' values like 10.0_296.0 show a consistent fraud rate of 100%, indicating a potential hotspot for fraudulent behavior.
- Potential Impact:
 - Organizations can focus their fraud detection efforts on transactions associated with top 'addr1' and 'addr2' values, optimizing resources for maximum impact.
 - Special attention should be given to combined 'addr' values, especially those consistently exhibiting high fraud rates like 10.0_296.0, guiding targeted interventions in those specific areas.

- Further investigation into the patterns of these 'addr' combinations can inform feature engineering and enhance the organization's ability to detect and prevent fraud effectively.