**Code For Retrieving Ethereum Addresses(Illicit & Non-Illicit)**

import requests

import pandas as pd

import time

# Your Etherscan API Key

API\_KEY = '7DNR18H7ASKXYSZNATBFRIPT4IZUDCNJ7U'

# Base Etherscan API URL

BASE\_URL = "https://api.etherscan.io/api"

# Known illicit Ethereum addresses (source: Ethereum Scam Database)

ILLICIT\_ADDRESSES\_URL = "https://raw.githubusercontent.com/MyCryptoHQ/scamdb/master/src/addresses.json"

# Function to fetch Ethereum addresses from recent transactions

def get\_recent\_addresses():

    url = f"{BASE\_URL}?module=proxy&action=eth\_blockNumber&apikey={API\_KEY}"

    response = requests.get(url).json()

    if "result" not in response:

        print("⚠ Error fetching latest block number")

        return []

    latest\_block = int(response["result"], 16)

    addresses = set()

    for block in range(latest\_block, latest\_block - 5000, -1):  # Checking last 5000 blocks

        block\_url = f"{BASE\_URL}?module=proxy&action=eth\_getBlockByNumber&tag={hex(block)}&boolean=true&apikey={API\_KEY}"

        response = requests.get(block\_url).json()

        if "result" in response and response["result"] and "transactions" in response["result"]:

            for tx in response["result"]["transactions"]:

                if tx["from"]:  # Ensure address is valid

                    addresses.add(tx["from"].lower())

                if tx["to"]:

                    addresses.add(tx["to"].lower())

        if len(addresses) >= 2500:  # Collect more than 1000 to filter out illicit ones

            break

        time.sleep(0.2)  # Avoid rate limits

    return list(addresses)

# Function to get known illicit addresses

def get\_illicit\_addresses():

    try:

        response = requests.get(ILLICIT\_ADDRESSES\_URL)

        if response.status\_code == 200:

            return set(addr.lower() for addr in response.json() if addr)  # Ensure no None values

    except Exception as e:

        print(f"⚠ Error fetching illicit addresses: {e}")

    return set()

# Retrieve addresses

print("Fetching recent Ethereum addresses...")

all\_addresses = get\_recent\_addresses()

# Retrieve illicit addresses

print("Fetching known illicit addresses...")

illicit\_addresses = get\_illicit\_addresses()

# Ensure all\_addresses is not empty

if not all\_addresses:

    print("⚠ No addresses fetched. Exiting.")

    exit()

# Filter out illicit addresses

non\_illicit\_addresses = [addr for addr in all\_addresses if addr and addr not in illicit\_addresses]

# Save 1000 non-illicit addresses

df = pd.DataFrame(non\_illicit\_addresses[:2000], columns=["Ethereum Address"])

df.to\_csv("non\_illicit\_ethereum\_addresses.csv", index=False)

print("✅ 1000 non-illicit Ethereum addresses saved to non\_illicit\_ethereum\_addresses.csv")

**Code For Retrieving Ethereum Addresses Data(Illicit & Non-Illicit)**

**PART 1**

import requests

import pandas as pd

import time

# Your Etherscan API Key

API\_KEY = '7DNR18H7ASKXYSZNATBFRIPT4IZUDCNJ7U'

# Load non-illicit Ethereum addresses dataset

file\_path = 'normaladdresss.csv'  # Update this path if needed

df = pd.read\_csv(file\_path)

# Base Etherscan API URL

BASE\_URL = 'https://api.etherscan.io/api'

# List to store results

data\_list = []

# Function to get Ethereum balance

def get\_eth\_balance(address):

    try:

        balance\_url = f"{BASE\_URL}?module=account&action=balance&address={address}&tag=latest&apikey={API\_KEY}"

        response = requests.get(balance\_url).json()

        if response.get('status') != '1':

            return None

        return int(response['result']) / (10\*\*18)  # Convert from Wei to Ether

    except Exception as e:

        print(f"⚠ Error fetching balance for {address}: {e}")

        return None

# Function to get ERC-20 transaction data

def get\_erc20\_transaction\_data(address):

    try:

        erc20\_url = f"{BASE\_URL}?module=account&action=tokentx&address={address}&startblock=0&endblock=99999999&sort=asc&apikey={API\_KEY}"

        response = requests.get(erc20\_url).json()

        if response.get('status') != '1':

            return None

        transactions = response.get('result', [])

        # Initialize variables

        total\_transactions = len(transactions)

        received\_values, sent\_values = [], []

        received\_timestamps, sent\_timestamps = [], []

        received\_from\_addresses, sent\_to\_addresses = set(), set()

        # Process transactions

        for tx in transactions:

            value = int(tx['value']) / (10\*\*18)  # Convert from Wei to Ether

            timestamp = int(tx['timeStamp'])

            if tx['from'].lower() == address.lower():

                sent\_values.append(value)

                sent\_timestamps.append(timestamp)

                sent\_to\_addresses.add(tx['to'])

            elif tx['to'].lower() == address.lower():

                received\_values.append(value)

                received\_timestamps.append(timestamp)

                received\_from\_addresses.add(tx['from'])

        # Calculate time differences

        time\_diff\_first\_last = (max(received\_timestamps + sent\_timestamps) - min(received\_timestamps + sent\_timestamps)) / 60 if received\_timestamps or sent\_timestamps else None

        avg\_min\_between\_sent\_tnx = (sum(sent\_timestamps[i+1] - sent\_timestamps[i] for i in range(len(sent\_timestamps)-1)) / len(sent\_timestamps)-1) / 60 if len(sent\_timestamps) > 1 else None

        avg\_min\_between\_received\_tnx = (sum(received\_timestamps[i+1] - received\_timestamps[i] for i in range(len(received\_timestamps)-1)) / len(received\_timestamps)-1) / 60 if len(received\_timestamps) > 1 else None

        # Calculate min, max, avg values

        min\_val\_received = min(received\_values) if received\_values else None

        max\_val\_received = max(received\_values) if received\_values else None

        avg\_val\_received = sum(received\_values) / len(received\_values) if received\_values else None

        min\_val\_sent = min(sent\_values) if sent\_values else None

        max\_val\_sent = max(sent\_values) if sent\_values else None

        avg\_val\_sent = sum(sent\_values) / len(sent\_values) if sent\_values else None

        total\_ether\_sent = sum(sent\_values)

        total\_ether\_received = sum(received\_values)

        total\_ether\_balance = get\_eth\_balance(address)

        return {

            'Total Ether Balance': total\_ether\_balance,

            'Total Transactions (including contract creation)': total\_transactions,

            'Time Difference First-Last (min)': time\_diff\_first\_last,

            'Avg Min Between Sent Tnx': avg\_min\_between\_sent\_tnx,

            'Avg Min Between Received Tnx': avg\_min\_between\_received\_tnx,

            'Sent Transactions': len(sent\_values),

            'Received Transactions': len(received\_values),

            'Unique Received From Addresses': len(received\_from\_addresses),

            'Unique Sent To Addresses': len(sent\_to\_addresses),

            'Min Value Received': min\_val\_received,

            'Max Value Received': max\_val\_received,

            'Avg Value Received': avg\_val\_received,

            'Min Value Sent': min\_val\_sent,

            'Max Value Sent': max\_val\_sent,

            'Avg Value Sent': avg\_val\_sent,

            'Total Ether Sent': total\_ether\_sent,

            'Total Ether Received': total\_ether\_received

        }

    except Exception as e:

        print(f"⚠ Error getting ERC20 transactions for {address}: {e}")

        return None

# Loop through addresses and retrieve data

count = 0

for address in df['Ethereum Address'].dropna().unique():

    if count >= 2000:  # Process up to 1000 addresses

        break

    try:

        eth\_data = get\_erc20\_transaction\_data(address)

        if eth\_data:  # Only add to data list if valid data is retrieved

            data\_list.append({'Address': address, \*\*eth\_data})

    except Exception as e:

        print(f"⚠ Error processing address {address}: {e}")

    count += 1

    time.sleep(1)  # Avoid API rate limits

# Convert results to DataFrame and save to CSV

df\_results = pd.DataFrame(data\_list)

df\_results.to\_csv('non\_illicit\_ethereum\_data.csv', index=False)

print("✅ Data retrieval complete. Results saved to 'non\_illicit\_ethereum\_data.csv'")

print(df\_results)

**PART 2**

import requests

import pandas as pd

import time

# Your Etherscan API Key

API\_KEY = '7DNR18H7ASKXYSZNATBFRIPT4IZUDCNJ7U'

# Load non-illicit Ethereum addresses dataset

file\_path = 'non\_illicit\_ethereum\_addresses.csv'  # Update this path if needed

df = pd.read\_csv(file\_path)

# Base Etherscan API URL

BASE\_URL = 'https://api.etherscan.io/api'

# List to store results

data\_list = []

# Function to get Ethereum balance

def get\_eth\_balance(address):

    try:

        balance\_url = f"{BASE\_URL}?module=account&action=balance&address={address}&tag=latest&apikey={API\_KEY}"

        response = requests.get(balance\_url).json()

        if response.get('status') != '1':

            return None

        return int(response['result']) / (10\*\*18)  # Convert from Wei to Ether

    except Exception as e:

        print(f"⚠ Error fetching balance for {address}: {e}")

        return None

# Function to get ERC-20 transaction data

def get\_erc20\_transaction\_data(address):

    try:

        erc20\_url = f"{BASE\_URL}?module=account&action=tokentx&address={address}&startblock=0&endblock=99999999&sort=asc&apikey={API\_KEY}"

        response = requests.get(erc20\_url).json()

        if response.get('status') != '1':

            return None

        transactions = response.get('result', [])

        # Initialize variables

        received\_values, sent\_values, timestamps\_sent, timestamps\_received = [], [], [], []

        sent\_tokens, received\_tokens = {}, {}

        sent\_addresses, received\_addresses = set(), set()

        for tx in transactions:

            value = int(tx['value']) / (10\*\*18)  # Convert from Wei to Ether

            timestamp = int(tx['timeStamp'])

            token\_name = tx['tokenName']

            if tx['from'].lower() == address.lower():

                sent\_values.append(value)

                timestamps\_sent.append(timestamp)

                sent\_addresses.add(tx['to'].lower())

                sent\_tokens[token\_name] = sent\_tokens.get(token\_name, 0) + value

            elif tx['to'].lower() == address.lower():

                received\_values.append(value)

                timestamps\_received.append(timestamp)

                received\_addresses.add(tx['from'].lower())

                received\_tokens[token\_name] = received\_tokens.get(token\_name, 0) + value

        # Calculate statistics

        time\_diff = (max(timestamps\_sent + timestamps\_received) - min(timestamps\_sent + timestamps\_received)) / 60 if timestamps\_sent or timestamps\_received else None

        avg\_time\_between\_sent = sum(timestamps\_sent[i] - timestamps\_sent[i-1] for i in range(1, len(timestamps\_sent))) / len(timestamps\_sent) / 60 if len(timestamps\_sent) > 1 else None

        avg\_time\_between\_received = sum(timestamps\_received[i] - timestamps\_received[i-1] for i in range(1, len(timestamps\_received))) / len(timestamps\_received) / 60 if len(timestamps\_received) > 1 else None

        most\_sent\_token = max(sent\_tokens, key=sent\_tokens.get, default=None)

        most\_received\_token = max(received\_tokens, key=received\_tokens.get, default=None)

        return {

            'Total ERC20 Transactions': len(transactions),

            'ERC20 Total Ether Received': sum(received\_values),

            'ERC20 Total Ether Sent': sum(sent\_values),

            'ERC20 Unique Sent Addresses': len(sent\_addresses),

            'ERC20 Unique Received Addresses': len(received\_addresses),

            'ERC20 Avg Time Between Sent Txn (min)': avg\_time\_between\_sent,

            'ERC20 Avg Time Between Received Txn (min)': avg\_time\_between\_received,

            'ERC20 Min Value Received': min(received\_values, default=None),

            'ERC20 Max Value Received': max(received\_values, default=None),

            'ERC20 Avg Value Received': sum(received\_values) / len(received\_values) if received\_values else None,

            'ERC20 Min Value Sent': min(sent\_values, default=None),

            'ERC20 Max Value Sent': max(sent\_values, default=None),

            'ERC20 Avg Value Sent': sum(sent\_values) / len(sent\_values) if sent\_values else None,

            'ERC20 Unique Sent Token Names': len(sent\_tokens),

            'ERC20 Unique Received Token Names': len(received\_tokens),

            'ERC20 Most Sent Token Type': most\_sent\_token,

            'ERC20 Most Received Token Type': most\_received\_token,

            'Total Ether Balance': get\_eth\_balance(address),

            'Time Diff Between First and Last Txn (min)': time\_diff,

        }

    except Exception as e:

        print(f"⚠ Error getting ERC20 transactions for {address}: {e}")

        return None

# Loop through addresses and retrieve data

count = 0

for address in df['Ethereum Address'].dropna().unique():

    if count >= 1000:  # Process up to 1000 addresses

        break

    try:

        eth\_data = get\_erc20\_transaction\_data(address)

        if eth\_data:  # Only add to data list if valid data is retrieved

            data\_list.append({'Address': address, \*\*eth\_data})

    except Exception as e:

        print(f"⚠ Error processing address {address}: {e}")

    count += 1

    time.sleep(1)  # Avoid API rate limits

# Convert results to DataFrame and save to CSV

df\_results = pd.DataFrame(data\_list)

df\_results.to\_csv('non\_illicit\_ethereum\_data.csv', index=False)

print("✅ Data retrieval complete. Results saved to 'non\_illicit\_ethereum\_data.csv'")

print(df\_results)

**PART 3**

import requests

import pandas as pd

import time

# Your Etherscan API Key

API\_KEY = '7DNR18H7ASKXYSZNATBFRIPT4IZUDCNJ7U'

# Load non-illicit Ethereum addresses dataset

file\_path = 'normalpart2.csv'  # Update this path if needed

df = pd.read\_csv(file\_path)

# Base Etherscan API URL

BASE\_URL = 'https://api.etherscan.io/api'

# List to store results

data\_list = []

# Function to fetch Ethereum transaction data

def get\_eth\_transaction\_data(address):

    try:

        tx\_url = f"{BASE\_URL}?module=account&action=txlist&address={address}&startblock=0&endblock=99999999&sort=asc&apikey={API\_KEY}"

        response = requests.get(tx\_url).json()

        if response.get('status') != '1':

            return None

        transactions = response.get('result', [])

        if not transactions:

            return None

        # Initialize variables

        gas\_prices, gas\_limits, gas\_used, base\_fees, timestamps = [], [], [], [], []

        success\_count, failure\_count = 0, 0

        for tx in transactions:

            gas\_prices.append(int(tx['gasPrice']) / 1e9)  # Convert Wei to Gwei

            gas\_limits.append(int(tx['gas']) / 1e9)

            gas\_used.append(int(tx['gasUsed']) / 1e9)

            timestamps.append(int(tx['timeStamp']))

            if 'maxFeePerGas' in tx:

                base\_fees.append(int(tx['maxFeePerGas']) / 1e9)

            if tx['isError'] == '0':

                success\_count += 1

            else:

                failure\_count += 1

        # Compute statistics

        transaction\_frequency = (max(timestamps) - min(timestamps)) / 60 if len(timestamps) > 1 else None

        gas\_price\_volatility = max(gas\_prices) - min(gas\_prices) if gas\_prices else None

        success\_failure\_ratio = success\_count / failure\_count if failure\_count > 0 else success\_count

        return {

            'Avg Gas Price (Gwei)': sum(gas\_prices) / len(gas\_prices) if gas\_prices else None,

            'Avg Gas Limit (Gwei)': sum(gas\_limits) / len(gas\_limits) if gas\_limits else None,

            'Avg Gas Used (Gwei)': sum(gas\_used) / len(gas\_used) if gas\_used else None,

            'Avg Base Fee (Gwei)': sum(base\_fees) / len(base\_fees) if base\_fees else None,

            'Transaction Frequency (min)': transaction\_frequency,

            'Gas Price Volatility (Gwei)': gas\_price\_volatility,

            'Transaction Success-Failure Ratio': success\_failure\_ratio,

        }

    except Exception as e:

        print(f"⚠ Error fetching transaction data for {address}: {e}")

        return None

# Loop through addresses and retrieve data

count = 0

for address in df['Address'].dropna().unique():

    if count >= 1633:  # Process up to 1000 addresses

        break

    try:

        eth\_data = get\_eth\_transaction\_data(address)

        if eth\_data:  # Only add to data list if valid data is retrieved

            data\_list.append({'Address': address, \*\*eth\_data})

    except Exception as e:

        print(f"⚠ Error processing address {address}: {e}")

    count += 1

    time.sleep(1)  # Avoid API rate limits

# Convert results to DataFrame and save to CSV

df\_results = pd.DataFrame(data\_list)

df\_results.to\_csv('non\_illicit\_ethereum\_data.csv', index=False)

print("✅ Data retrieval complete. Results saved to 'non\_illicit\_ethereum\_data.csv'")

print(df\_results)

**Code For HeatMap**

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

# Load the CSV file

file\_path = "Final Combined 1.csv"  # Update path if needed

df = pd.read\_csv(file\_path)

# Compute the correlation matrix for numerical features

corr\_matrix = df.select\_dtypes(include=['float64', 'int64']).corr()

# Set up the matplotlib figure

plt.figure(figsize=(15, 10))

# Draw the heatmap with a color map

sns.heatmap(corr\_matrix, cmap="coolwarm", annot=False, fmt=".2f", linewidths=0.5)

# Set title

plt.title("Correlation Heatmap of Ethereum Transaction Features")

# Show the plot

plt.show()

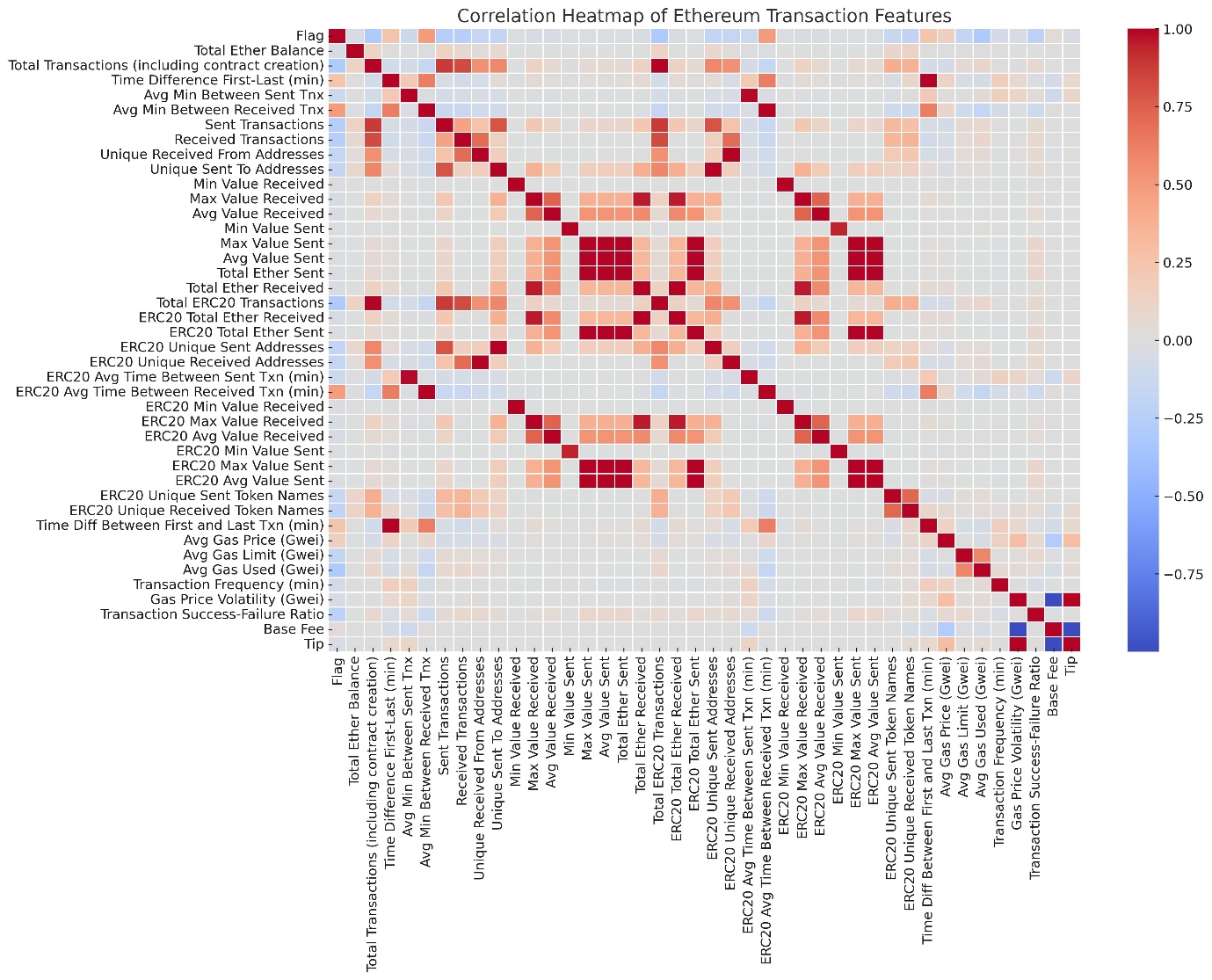
**Heatmap Analysis**

The heatmap visualizes correlations between numerical features in the dataset. Here are some key takeaways:

1. High Positive Correlations:
   * *Total Ether Sent* and *Total Transactions* are strongly correlated. This makes sense since addresses with more transactions tend to send more ether.
   * *Received Transactions* and *Unique Received From Addresses* show a strong relationship, indicating that users receiving more transactions likely interact with more unique senders.
   * *ERC20 Total Ether Sent* is highly correlated with *ERC20 Max Value Sent*, suggesting that addresses sending large amounts of ERC20 tokens generally have high-value transactions.
2. High Negative Correlations:
   * *Avg Min Between Sent Transactions* negatively correlates with *Total Transactions*. Addresses with frequent transactions have shorter intervals between them.
   * *Transaction Frequency (min)* is negatively correlated with *Total Transactions*. More active addresses have shorter transaction intervals.
3. Weak Correlations:
   * Some gas-related features don’t show strong correlation with transaction frequency, suggesting that gas price fluctuations may not significantly impact transaction behavior.

The heatmap shows how different numbers in the dataset are related to each other. Each small box represents a connection between two features. The colors indicate how strong the relationship is:

* **Red (closer to 1.0)** → Strong positive relationship (when one value increases, the other also increases).
* **Blue (closer to -1.0)** → Strong negative relationship (when one value increases, the other decreases).
* **Lighter colors (closer to 0.0)** → Weak or no relationship between the values.

****