README

1. **How to run**-

Step 1: Install and Open PyCharm

1. **Install PyCharm**: If you haven't already, download and install PyCharm Community Edition from the official website: PyCharm Download.
2. **Open PyCharm**: Launch PyCharm by clicking on its icon.

Step 2: Locate Your .py File

1. In the **Project** tool window on the left, navigate to the folder containing your .py file.
2. Double-click on the .py file to open it in the editor.

Step 3: Set Up the Python Interpreter

1. Go to **File** > **Settings** (or **PyCharm** > **Preferences** on macOS).
2. In the **Settings/Preferences** window, navigate to **Project: [Your Project Name]** > **Python Interpreter**.
3. Ensure that the correct Python interpreter (version) is selected. If it's not, you can add or configure one here.

Step 4: Run the necessary library to run .py File

Step 5: Run the .py File

1. With your .py file open, right-click anywhere inside the editor.
2. Select **Run '[Your Filename]'** from the context menu, or simply press **Shift + F10** to run the file.

Step 6: View Output

- The **Run** tool window at the bottom of PyCharm will display the output or any errors from your script.

Step 7: Viewing Images

- **Images**: You can view generated images directly within PyCharm if they're saved in supported formats like PNG or JPEG.

Data Preprocessing for Credit Card Transactions

This section processes a credit card transaction dataset to prepare it for analysis by performing various data cleaning and transformation tasks.

Prerequisites

Ensure the pandas library is installed. You can install it using:
pip install pandas

The script uses the pandas library for data manipulation and analysis. The dataset is read from the specified file path using pd.read_csv, identifies duplicate rows in the dataset and optionally removes them and also extracts the first 924,849 rows of the dataset.Key steps include splitting the trans_date_trans_time column into separate transaction_date and transaction_time columns, formatting the date to a readable dd-mm-yyyy format, and converting the time to a 24-hour format. It extracts details like transaction hours and months, rounds transaction amounts to the nearest integer, and categorizes age into generational groups. Redundant columns are dropped, and rows can be filtered based on specific transaction hours for targeted analysis. The script also includes error handling for invalid time formats and ensures seamless data manipulation, making the dataset ready for insights and visualization. Install pandas using pip install pandas before running the script.

Stepwise Description:

1. **Split Date and Time**

df[['transaction_date', 'transaction_time']] = df['trans_date_trans_time'].str.split(' ', expand=True)

Splits the trans_date_trans_time column into transaction_date and transaction_time based on a space separator.

2. **Format Transaction Date**

df['transaction_date'] = pd.to_datetime(df['transaction_date']).dt.strftime('%d%m%Y')

df['transaction_date'] = pd.to_datetime(df['transaction_date'], format='%d%m%Y').dt.strftime('%d-%m-%Y')

Formats the transaction_date column to a readable format (dd-mm-yyyy).

3. **Extract Transaction Hour**

df['transaction_time'] = pd.to_datetime(df['transaction_time'], format='%H:%M:%S')

df['transaction_hour'] = df['transaction_time'].dt.hour

Converts transaction_time to a datetime format and extracts the hour (transaction_hour).

4. **Extract Transaction Month**

df['transaction_month'] = df['transaction_date'].dt.month

Extracts the month from the transaction_date column.

5. **Round Off Transaction Amount**

df['rounded_amt'] = df['amt'].round()

Rounds the amt column to the nearest integer and stores it in a new column, rounded_amt.

6. **Categorize Generations**

df['generation'] = df['age'].apply(categorize_generation)

Categorizes age into generational groups using a custom function and creates a new column, generation.

7. **Convert transaction_time to 24-hour format**

df['transaction_time'] = pd.to_datetime(df['transaction_time'], errors='coerce')

df['transaction_time_24hr'] = df['transaction_time'].dt.strftime('%H:%M:%S')

Converts the transaction_time column to pandas datetime format and reformats it to a 24-hour time string (HH:mm:ss).

- **Error Handling**: errors='coerce' ensures invalid time values are converted to NaT (Not a Time) instead of causing errors.

8. **View and Drop Columns**

print(df.columns)

df = df.drop(columns=['transaction_hour'])

Drops the transaction_hour column, which might be unnecessary due to redundancy or recalculation.

9. **Extract Hour from 24-hour Time**

df['transaction_time_24hr'] = pd.to_datetime(df['transaction_time_24hr'], format='%H:%M:%S')

df['transaction_hour'] = df['transaction_time_24hr'].dt.hour

Converts transaction_time_24hr to datetime format and extracts the hour value to populate a new column, transaction_hour.

10. **Filter Transactions at Specific Hour**

transactions_at_14 = df[df['transaction_hour'] == 14]

Filters rows where transactions occurred at 2:00 PM (transaction_hour == 14).

11. **Reformat transaction_date and transaction_time**

df['transaction_date'] = pd.to_datetime(df['transaction_date']).dt.strftime('%d-%m-%Y')

df['transaction_time'] = pd.to_datetime(df['transaction_time']).dt.strftime('%H:%M:%S')

Ensures transaction_date is formatted as dd-mm-yyyy and transaction_time remains in a 24-hour format.

12. **Drop Unnecessary Columns**

df = df.drop(columns=['transaction_time_str', 'is_12_hour_format'])

Removes redundant columns (transaction_time_str, is_12_hour_format), which might no longer be needed for analysis.

13. **Convert transaction_time_24hr to Time Format**

df['transaction_time_24hr'] = pd.to_datetime(df['transaction_time_24hr']).dt.time

Extracts only the time part (without the date) from the transaction_time_24hr column.

14. **Filter Transactions for a Specific Hour**

df_15_hour = df[df['transaction_hour'] == 15]

Filters rows where transactions occurred at 3:00 PM (transaction_hour == 15).

15. **Drop the transaction_time_12hr Column**

df = df.drop(columns=['transaction_time_12hr'])

Removes the transaction_time_12hr column, which might be redundant.

16. **Round Off Amounts**

df['rounded_amt'] = df['amt'].round()

df = df.drop(columns=['amt'])

Rounds the amt column to the nearest integer and stores it in a new column, rounded_amt.

1. **Readme for Density Plot:**

   Prerequisites

1. Python installed on your system (preferably version 3.7 or above).
2. Required Python libraries:
   - pandas
   - seaborn
   - matplotlib

A CSV file named updated_dataset.csv containing the relevant data, structured with the following columns:

- job_categories: The category of the job associated with the transaction.
- rounded_amt: The transaction amount, rounded for simplicity.
- is_fraud: Binary flag indicating whether the transaction was fraudulent (1) or not (0).

2. Readme for Word Cloud:

Required Python libraries:

- pandas
- wordcloud
- matplotlib

Install them using the command:

pip install pandas wordcloud matplotlib

The word cloud is styled with:

- A white background.
- A viridis colormap for word coloring.
- A maximum of 200 words displayed.

A word cloud visualization will appear, showcasing the most common merchant categories in your dataset.

3. Readme for Correlation Matrix:

Required Python libraries:

- pandas
- seaborn
- matplotlib

Install them using the command:

pip install pandas seaborn matplotlib

Numeric Column Selection:

- Automatically selects numeric columns from the dataset using Pandas.
- Non-numeric columns are excluded to focus only on numerical correlations.

Correlation Matrix Calculation:

- Calculates pairwise correlations between numeric columns using the .corr() method.
- The resulting matrix shows values ranging from -1 (perfect negative correlation) to 1 (perfect positive correlation).

4. Readme for Dumbell Chart:

Required Python libraries:

- pandas
- matplotlib
- Numpy

  Focuses on transactions from selected states: NY, TX, OH, MO, FL, and PA.

  Ensures that only relevant data is analyzed.

Counting Unique Credit Card Numbers:

- Calculates the number of unique credit card numbers for both fraudulent and non-fraudulent transactions in each selected state.

Dumbbell Chart Creation:

- Uses Matplotlib to plot a dumbbell chart, showing the counts of unique credit card numbers for each state as two points connected by a line.
- Labels each point with its corresponding count for clarity.

5. Readme for KDE curve

Files in the Repository

1. main.py: Contains the Python script for analyzing fraudulent transaction amounts.
2. A1 Refined Dataset - Dataset.csv: The input dataset used for analysis (assumed to be available in the specified directory).

Libraries Used

- pandas: For data loading and manipulation.
- numpy: For numerical operations.
- matplotlib: For creating the visualization.
- scipy.stats: For kernel density estimation (KDE) and calculating mode.

Code: pip install pandas numpy matplotlib scipy

Next, filter Fraudulent Transactions: Filter the dataset to isolate rows where is_fraud equals 1, retaining only the amt column for further analysis.

Calculating Statistics:

mean_amt = fraud_transactions['amt'].mean()

median_amt = fraud_transactions['amt'].median()

mode_amt = fraud_transactions['amt'].mode()[0]

Output:

A KDE plot showing the distribution of fraudulent transaction amounts.

Vertical lines indicating:

- Mean (red, dashed line)
- Median (green, dashed line)
- Mode (purple, dashed line)

6. Readme for Covariance plots:

Installation and Setup

Prerequisites

- Python 3.7 or higher
- Required Python libraries:
    - pandas
    - seaborn
    - matplotlib
    - scikit-learn

Install the required libraries using:

pip install pandas seaborn matplotlib scikit-learn

Workflow:

1. **Data Loading**: Load the dataset using pandas.
2. **Data Cleaning**: Drop unnecessary columns like Unnamed, cc_num, unix_time, and zip.
3. **Categorical and Numeric Column Handling**:
    - Identify categorical and numeric columns.
    - Perform label encoding for high-cardinality columns.
    - Apply one-hot encoding for low-cardinality columns.
4. **Data Normalization**: Normalize numeric data using StandardScaler.
5. **Heatmap Visualization**: Use Seaborn's heatmap to display the covariance matrix with annotations and styling.

7. README for Covariance Matrix Heatmap of Selected Variables

Importing Libraries

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler, LabelEncoder

- **pandas**: For data manipulation and analysis.
- **seaborn**: For advanced data visualization.
- **matplotlib.pyplot**: For plotting graphs.
- **sklearn.preprocessing**: For data normalization using StandardScaler.

Selecting Relevant Columns

selected_columns = ['is_fraud', 'rounded_amt', 'transaction_hour', 'age', 'transaction_month']

data = data[selected_columns]

Filters the dataset to include only relevant columns for analysis:

- is_fraud: Indicates whether a transaction is fraudulent.
- rounded_amt: The transaction amount rounded to a specific value.
- transaction_hour: The hour at which the transaction occurred.
- age: Age of the customer.
- transaction_month: Month of the transaction.

Normalizing the Numeric Data

scaler = StandardScaler()

normalized_data = pd.DataFrame(scaler.fit_transform(data), columns=data.columns)

Normalizes the selected columns using StandardScaler to ensure they are on a similar scale for meaningful covariance calculation.

Calculating the Covariance Matrix

covariance_matrix = normalized_data.cov()

Computes the covariance matrix, showing the relationships between the normalized variables.

Visualizing the Covariance Matrix

sns.heatmap: Creates a heatmap for the covariance matrix.

- annot=True: Displays the covariance values in each cell.
- fmt=".2f": Rounds the covariance values to two decimal places.
- cmap="coolwarm": Uses a diverging colormap for visualization.
- xticklabels and yticklabels: Ensures labels are displayed for rows and columns.

Adjusting Label Orientation and Adding Title

- **plt.xticks**: Rotates x-axis labels for better readability.
- **plt.title**: Adds a title to the heatmap.
- **plt.tight_layout**: Ensures that all elements fit neatly within the figure.
- **plt.show**: Displays the heatmap.

8. README for Choropleth Map Visualization of Fraud Cases by State

Prerequisites

- **Python 3.7+**

Required libraries:
pip install pandas plotly

1. Importing Libraries

```
import pandas as pd

import plotly.express as px
```

- **pandas**: For data manipulation and analysis.
- **plotly.express**: For creating an interactive choropleth map.

2. Filtering Fraud Cases

```
fraud_data = data[data['is_fraud'] == 1]
```

Filters the dataset to include only rows where fraud cases (is_fraud == 1) are present.

3. Grouping Fraud Cases by State

```
fraud_by_state = fraud_data.groupby('state').size().reset_index(name='fraud_count')
```

Groups the filtered fraud data by the state column to calculate the number of fraud cases for each state.

4. Creating a Choropleth Map

- **Parameters**:
    - locations='state': Specifies the column containing state abbreviations (e.g., "NY" for New York).
    - locationmode='USA-states': Maps the data to the U.S. state boundaries.
    - color='fraud_count': Colors the states based on the number of fraud cases.
    - color_continuous_scale='Reds': Chooses a red color scale to represent fraud case intensity.
    - scope='usa': Restricts the map to the U.S. region.

- title: Sets the title of the map.
- hover_name: Displays state names in the tooltip when hovering.

5. Displaying the Map

fig.show()

The workflow produces an interactive U.S. map where:

- Each state is colored based on the number of fraud cases (fraud_count).
- A red gradient represents the intensity of fraud, with darker shades indicating higher fraud counts.
- Hovering over a state displays its name and fraud count.

9. README for Visualizing Fraud Occurrence by Gender in Selected U.S. States(Donut in geographic map)

1. Importing Required Libraries

```
import pandas as pd

import plotly.graph_objects as go
```

- **pandas**: Handles data processing.
- **plotly.graph_objects**: Creates interactive and customizable map visualizations.

2. Grouping Data by State and Gender

```
fraud_data = filtered_data.groupby(['state', 'gender']).size().reset_index(name='fraud_count')
```

- Groups data by state and gender to calculate the number of fraud cases for each gender in each state.

3. Calculating State Coordinates

```
state_coords = filtered_data.groupby('state')[['lat', 'long']].mean().reset_index()
```

Calculates the average latitude and longitude for each state to position the pie charts on the map.

4. Merging Fraud Data with Coordinates

fraud_data_with_coords = pd.merge(fraud_data, state_coords, on='state')

Merges the fraud data with the state coordinates to associate the fraud information with geographic locations.

5. Defining Pie Chart Placement

```
state_domains = {
   'TX': {'x': [0.40, 0.50], 'y': [0.30, 0.40]},
   'NY': {'x': [0.63, 0.73], 'y': [0.70, 0.80]},
   'PA': {'x': [0.69, 0.73], 'y': [0.55, 0.65]},
   'OH': {'x': [0.59, 0.70], 'y': [0.48, 0.58]},
   'FL': {'x': [0.62, 0.72], 'y': [0.20, 0.30]},
   'MO': {'x': [0.48, 0.58], 'y': [0.52, 0.63]},
}
```

Specifies the position of each pie chart for better placement over the corresponding states.

6. Adding Pie Charts and State Names to the Map

```
fig.add_trace(go.Pie(values=values, labels=labels, domain=state_domains[state],
name=f'Fraud in {state}',hole=0.3, textinfo='none',showlegend=True))
```

7. Customizing the Map Layout

```
fig.update_layout(  title_text='Fraud Occurrence by Gender in Selected
States',showlegend=True,
geo=dict(scope='usa',projection=go.layout.geo.Projection(type='albers usa'),
showland=True,  landcolor="rgb(250, 250, 250)"  ))
```

Sets the map title, ensures the legend is visible, and customizes the map's appearance.

8. Displaying the Interactive Map

fig.show()

Renders the map in an interactive environment.

Prerequisites

- **Python 3.7+**

Required Libraries:
pip install pandas plotly

10. README for Sunburst Chart Visualization of Fraud Cases in Selected U.S. States

1. Importing Required Libraries

import pandas as pd

import plotly.express as px

- **pandas**: Used for data manipulation.
- **plotly.express**: Simplifies the creation of interactive visualizations like Sunburst charts.

2. Creating the Sunburst Chart

```
fig = px.sunburst(
  filtered_data,
  path=['state', 'city', 'job_categories'],  # Hierarchical path: State > City > Job Categories
  values=None,
  title="Sunburst Chart: Job Categories by State and City of 2019(Fraud Cases in NY, TX, OH, PA, FL, MO)",
  color='state',
```

color_discrete_map=state_colors)

Hierarchical Path: Creates a Sunburst chart hierarchy:

- **state**: The outermost layer of the Sunburst.
- **city**: The middle layer, grouped under respective states.
- **job_categories**: The innermost layer, grouped under cities.

Segment Sizes: Automatically determined by the count of occurrences.

Color: Each state is assigned a unique pastel color based on the state_colors dictionary.

The script generates an interactive Sunburst chart with the following features:

1. **Hierarchy**:
   - **Outer Layer**: States.
   - **Middle Layer**: Cities within the selected states.
   - **Inner Layer**: Job categories associated with fraud cases.
2. **Color Scheme**:
   - Each state is represented by a custom pastel color.
3. **Interactivity**:
   - Hover over any segment to view details such as state, city, job category, and count of occurrences.
4. **Title**:
   - Describes the context: Fraud cases in the selected states from the 2019 dataset.

Prerequisites

- **Python 3.7+**

Required Libraries:
pip install pandas plotly

The Credit Card Transactions Dataset provides detailed records of transactions from 1 July 2019, to31 Dec, 2019. It contains over 1.85 million rows with key attributes that capture transaction time, amount, and associated demographic, geographic, and merchant details. For analysis purposes, only the last six months of 2019 are considered, balancing dataset manageability and relevance.

Data Processing-
This process was done at the time of preprocessing the dataset and is mentioned in the same file.

11. README: Violin Plot - Fraud Transaction Hour for High-Fraud Cities

This web application visualizes fraud transaction hours in high-fraud cities using a **Violin Plot**. The plot provides a distribution of transaction hours, offering insights into fraudulent behavior patterns.

1. **Violin Plot Visualization**:
    - Represents the distribution of transaction hours in high-fraud cities.
    - Cities included: Dallas, Lolita, Houston, Meridian, Roma.
    - Displays:
        - Meanline for the data distribution.
        - Points for individual transaction hours.
        - Box plot overlay for summary statistics.
2. **Interactive File Upload**:
    - Accepts CSV files as input.
    - Filters data based on is_fraud = 1 and the specified cities.
3. **Dynamic Error Handling**:
    - Ensures only valid CSV files are accepted.
    - Handles missing or incomplete data gracefully.

How to Use

1. **Upload a CSV File**:
    - Click the file input box and select a .csv file containing the required fields.
    - The CSV must include the following columns:
        - city: City names.
        - is_fraud: Indicator for fraudulent transactions (should contain "1" for fraud).
        - transaction_hour: The hour of the transaction.

Installation and Requirements

Dependencies

The project uses the following external libraries hosted online:

- **Plotly** for interactive plotting.
- **D3.js** for CSV parsing.

Code Explanation

HTML Structure

- **File Input**: Enables the user to upload a CSV file.
- **Error Message**: Displays errors if an invalid file is uploaded.
- **Plot Div**: Placeholder for rendering the Violin Plot.

JavaScript Workflow

1. **File Upload and Validation**:
    - Listens for file selection.
    - Validates file type to ensure it's a CSV.
2. **Data Parsing**:
    - Uses d3.csvParse to parse the uploaded CSV.
3. **Data Filtering**:
    - Filters data for high-fraud cities and transactions marked as fraud (is_fraud = 1).
4. **Plot Generation**:
    - Creates individual violin traces for each city.
    - Handles missing or incomplete data by representing it as empty distributions.
5. **Plot Rendering**:
    - Renders the Violin Plot using Plotly.

12. README: Violin Plot - Fraud Rounded Amount for High-Fraud Cities

This web application visualizes the distribution of fraud-related **Rounded Amounts** in high-fraud cities using a **Violin Plot**. The plot provides insights into the financial patterns of fraudulent transactions in selected cities.

1. **Violin Plot Visualization**:
    - Represents the distribution of **Rounded Amounts** for fraudulent transactions.
    - Cities included: Rock Glen, Vanderbilt, Clune, Corsica, Skytop.
    - Displays:
        - Meanline for data distribution.
        - Points for individual amounts.
        - Box plot overlay for summary statistics.
2. **Interactive File Upload**:
    - Accepts CSV files as input.
    - Filters data based on is_fraud = 1 and specified cities.
3. **Dynamic Error Handling**:
    - Ensures only valid CSV files are accepted.
    - Handles missing or incomplete data gracefully.

How to Use

1. **Upload a CSV File**:
    - Click the file input box and select a .csv file containing the required fields.
    - The CSV must include the following columns:
        - city: City names.
        - is_fraud: Indicator for fraudulent transactions (should contain "1" for fraud).
        - rounded_amt: The rounded transaction amount.
2. **View the Plot**:
    - Once the CSV file is uploaded, the Violin Plot will be rendered.
    - Each violin represents the distribution of rounded transaction amounts for a city.

Installation and Requirements

Dependencies

- **Plotly** for interactive plotting.
- **D3.js** for CSV parsing.

Code Explanation

HTML Structure

- **File Input**: Enables the user to upload a CSV file.
- **Error Message**: Displays errors if an invalid file is uploaded.
- **Plot Div**: Placeholder for rendering the Violin Plot.

JavaScript Workflow

1. **File Upload and Validation**:
    - Listens for file selection.
    - Validates file type to ensure it's a CSV.
2. **Data Parsing**:
    - Uses d3.csvParse to parse the uploaded CSV.
3. **Data Filtering**:
    - Filters data for high-fraud cities and transactions marked as fraud (is_fraud = 1).
4. **Plot Generation**:
    - Creates individual violin traces for each city.
    - Handles missing or incomplete data by representing it as empty distributions.

5. **Plot Rendering**:
    - Renders the Violin Plot using Plotly.

13. README: Parallel Coordinate Plot - Fraud Analysis for High-Fraud Cities

This web application visualizes the relationship between multiple variables in high-fraud cities through a **Parallel Coordinate Plot**. The plot helps in analyzing and comparing various dimensions of fraudulent transactions across selected cities.

1. **Parallel Coordinate Plot Visualization**:
    - Illustrates the relationships between multiple dimensions:
        - **City**: High-fraud cities in Pennsylvania.
        - **Is Fraud**: Indicates fraudulent (1) or non-fraudulent (0) transactions.
        - **Category**: Category of transactions.
        - **Generation**: Generational data associated with the transaction.
        - **Job Categories**: Job categories related to individuals involved in transactions.
    - Line coloring:
        - Blue for non-fraudulent transactions.
        - Yellow for fraudulent transactions.
2. **Interactive File Upload**:
    - Accepts a CSV file as input for dynamic data visualization.
    - Filters and processes data based on predefined criteria.
3. **Error Handling**:
    - Validates file type to ensure it is a CSV.
    - Alerts the user if the uploaded file has no valid data points.

How to Use

1. **Upload a CSV File**:
    - Click the file input box and select a .csv file containing the required fields.
    - The file should include the following columns:
        - state: The state where the transaction occurred (must include "PA").
        - city: The city where the transaction occurred (one of the high-fraud cities).
        - is_fraud: Indicates whether the transaction was fraudulent (1 for fraud).

        - category: The category of the transaction.
        - generation: The generational group (e.g., Gen Z, Millennial).
        - job_categories: The job category associated with the individual.
2. **View the Plot**:
    - Upon uploading, the Parallel Coordinate Plot will be displayed.
    - Each line represents a transaction, connecting points across all dimensions.
3. **Analyze Insights**:
    - Use the interactive plot to filter and explore trends in the data.
    - Hover over lines to see detailed values for each dimension.

Installation and Requirements

Dependencies

- **Plotly**: For rendering the Parallel Coordinate Plot.
- **D3.js**: For parsing the CSV data.

Code Explanation

HTML Structure

- **File Input**: Allows users to upload a CSV file.
- **Error Message**: Displays validation or data-related error messages.
- **Plot Div**: Placeholder for rendering the Parallel Coordinate Plot.

JavaScript Workflow

1. **File Upload and Validation**:
    - Listens for file input.
    - Validates file type and checks if data is present.
2. **Data Filtering**:
    - Filters data for:
        - state = "PA".
        - Cities in ["Rock Glen", "Vanderbilt", "Clune", "Corsica", "Skytop"].
        - Dimensions (category, generation, job_categories) are non-empty.
3. **Dynamic Dimension Assignment**:
    - Maps categorical data (e.g., cities, generations) to numeric indices for plotting.
4. **Plot Generation**:
    - Uses Plotly.parcoords to render the Parallel Coordinate Plot.
    - Customizes line color to highlight fraudulent transactions.
5. **Plot Rendering**:

- Displays the plot with appropriate labels, ticks, and scales.

How to run the code for PCP

1. Access the Google Sheet: Click on the provided Google Sheet link.
2. Download the Data: In Google Sheets, go to File > Download > Comma-separated values (.csv, current sheet). Ensure the file is downloaded in CSV format, as the code is designed to work only with CSV files.
3. Run the HTML Code: Once the CSV is downloaded, open the HTML file in your browser. This can be done by double-clicking the HTML file or right-clicking it and selecting Open with > Your preferred browser.
4. Select the CSV File: Upon opening the HTML page, you will see an option to choose a file.
5. Click the "Choose File" button and select the CSV file you just downloaded from Google Sheets.
6. View the Visualization: After selecting the CSV file, the treemap visualization will be displayed based on the data from the CSV file.

14. Readme for ML Classification:

Requirements

To run this project, you will need the following Python libraries:

- pandas
- numpy
- scikit-learn
- imbalanced-learn
- matplotlib
- seaborn
- joblib

You can install these libraries using the following command:

pip install -r requirements.txt

How to Use

Load the Dataset: Update the file_path variable with the path to your dataset CSV file.

```
file_path = r"C:\path\to\your\dataset.csv"
df = pd.read_csv(file_path)
```

Preprocessing: The data is preprocessed by scaling numerical features and encoding categorical features using one-hot encoding. You can customize the numerical_columns and categorical_columns lists to match the features in your dataset.

Model Training: The model is trained using a Random Forest classifier. The dataset is split into training and test sets (80% train, 20% test).

pipeline.fit(X_train, y_train)

Model Evaluation: The model is evaluated using various metrics, including accuracy, classification report, confusion matrix, ROC AUC, and Precision-Recall curves.

Threshold Adjustment: The decision threshold can be adjusted for predicting fraudulent transactions. You can change the threshold value to optimize precision and recall.

predictions = (probabilities > 0.7).astype(int)

The trained model can be saved for later use.

Here is how to save and load the model:

Save the Model:

```
import joblib
joblib.dump(pipeline, 'fraud_detection_model.pkl')
```

Load the Model:

```
pipeline = joblib.load('fraud_detection_model.pkl')
```

15. Readme for Forecasting

Requirements

To run this project, you will need the following Python libraries:

```
pip install pandas matplotlib seaborn statsmodels pmdarima scikit-learn
```

16. Readme for K to D (Data Processing)

Prerequisites

Python 3.x
Libraries:
- `pandas`
- `seaborn`

- `matplotlib`
- `sklearn`

Install the required libraries using:
pip install pandas seaborn matplotlib scikit-learn

File Structure

- Input File: updated_dataset.csv
- Output File: corrected_dataset.csv

Usage

Inspect Results:

Metrics such as fraud_victim_count, fraud_loss_amount, fraud_per_victim, and fraud_per_1000 will be added to the dataset.

The corrected dataset will be saved to the specified output file.

Outputs

- **Fraud Metrics**:
  - fraud_victim_count: Count of fraud transactions per victim.
  - fraud_loss_amount: Total amount lost due to fraud per victim.
  - fraud_per_victim: Ratio of fraudulent transactions to total transactions per victim.
  - fraud_per_1000: Number of fraud cases per 1,000 people per state.
- **Corrected Dataset**:
  - Contains all original columns plus the computed fraud metrics.

For plots : Change path in HTML file for CSS and Javascript link depending upon your downloads.