1. Project Introduction

The **Crime's Rate Project** focuses on the **analysis, visualization, and clustering** of crime incident data using **data mining techniques** and **evaluation metrics**. The goal is to identify meaningful patterns in criminal activities by examining temporal (date/time), spatial (longitude/latitude), and categorical (type of crime, police district) aspects.

Key Objectives:

- **Preprocessing and cleaning** the dataset to ensure quality and consistency.
- **Extracting temporal features** (e.g., hour, day, month) from date fields to understand time-based trends in criminal activity.
- **Visualizing** distributions, relationships, and hotspots of crime incidents using both static and interactive plots.
- Applying **data mining algorithms**, particularly clustering techniques like **K-Medoids**, to group geographically similar incidents
- Use **Logistic Regression** to predict categorical outcomes—such as the type of crime or the likelihood of a crime occurring at a specific time/place—based on historical data.
- Using **evaluation metrics** to assess clustering quality and determine optimal grouping.

This project ultimately aids in identifying:

- High-risk zones ("hotspots") in cities.
- Crime surges at particular times.
- Potential strategies for smarter law enforcement deployment based on patterns.

---

2. Data Features

The dataset contains:

- **Dates**: When the crime occurred.
- **Category**: Type of crime.
- **Descript**: Description (in training data only).
- **DayOfWeek**: Day name (e.g., Monday).
- **PdDistrict**: Police district.
- **Resolution**: How it was resolved (e.g., arrest).
- **Address**: Street location.
- **X (Longitude)**, **Y (Latitude)**: Location coordinates.

---

3. Importing Required Libraries

Detailed Explanation:

Importing Required Libraries and Uplouding Data

```
[4]: import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.express as px
import warnings
from sklearn_extra.cluster import KMedoids
warnings.filterwarnings('ignore')
```

- **numpy (as np)**
  Used for handling **numerical arrays and vectorized operations**. Although the core dataset is tabular, certain transformations (e.g., distances in clustering) may require array computations that NumPy handles efficiently.
- **pandas (as pd)**
  The core library for **data loading, manipulation, filtering, and transformation.** It allows us to handle datasets as DataFrames, which are essentially table-like structures. All preprocessing steps—cleaning, extracting date parts, checking for duplicates, etc.—are performed using pandas.
- **seaborn (as sns)**
  A **high-level visualization library** built on top of Matplotlib. It is particularly useful for creating **statistical plots** such as count plots, heatmaps, and bar charts with minimal code. Seaborn also integrates well with pandas DataFrames.
- **matplotlib.pyplot (as plt)**
  A foundational plotting library in Python. While Seaborn simplifies many tasks, Matplotlib gives full **customization control** over plots (titles, axes, legends, grid, etc.). It's often used behind the scenes even when working with Seaborn.
- **plotly.express (as px)**
  This library is used for **interactive visualizations**, such as scatter plots

that allow zooming and tooltips. It is highly effective for **geospatial data visualization**, like plotting clusters on a city map based on longitude and latitude.

- **warnings**
  The Python warnings module is used to manage and suppress runtime warnings.This line suppresses warning messages during execution, making notebook outputs cleaner—especially useful when working with libraries that may emit frequent but non-critical warnings.
- **KMedoids from sklearn_extra.cluster**
  This algorithm is a **robust alternative to KMeans,** where the center of each cluster is one of the actual data points (a medoid) rather than a mean. It's more **resistant to noise and outliers**, making it ideal for real-world data like crime locations.

---

4. Uploading and Reading the Data

```
[ ] from google.colab import files
[ ] uploaded=files.upload()
[ ] df_train=pd.read_csv('/content/Crime Train Data.csv')
[ ] df_test=pd.read_csv('/content/Crime Test Data.csv')
```

from google.colab import files:
This line **imports the files module** from Colab's built-in tools. It's necessary because **Google Colab runs in a cloud environment**, so you can't access your local files directly like in a regular Python script.

files.upload():
This **launches a file picker UI** in your browser that lets you select files from your computer. It uploads the selected file(s) to the Colab runtime (a temporary cloud environment).

uploaded = files.upload():
The output of the upload process is a **dictionary** where:

- **Keys** are filenames (e.g., "train.csv"),
- **Values** are the file data in binary format.
- **pd.read_csv():**
  This is a **pandas function** that reads data from a **CSV (Comma-Separated Values)** file and loads it into a **DataFrame**—a 2D table structure similar to Excel or SQL tables.
- **'train.csv' and 'test.csv'**: These are the filenames of the uploaded datasets. They must **exactly match** the name you uploaded via the file picker or reference from your local directory.
- **df_train and df_test**:
  These variables hold the loaded datasets. After this step:

- **df_train** contains the **training data**, usually with labels (e.g., crime category).
- **df_test** contains **unlabeled or new data**, typically used for prediction or evaluation.

---

5. Data Preprocessing

Step 1: Checking the Shape of the Dataset

```
[101]: # Get the shape (number of rows and columns) of the DataFrame 'df_train'
       df_train.shape

[101]: (873391, 12)
```

- **Purpose**: Shows the dataset's structure.
- **Output**: A tuple like (n_rows, n_columns)
- **Why it's useful**:
  - Helps you understand how big your dataset is.
  - Alerts you to potentially excessive data (which might slow processing).
  - Lets you compare row counts before/after cleaning to detect data loss.

Step 2: Viewing Column Names

```
[102]: # Printing the list of column names in the 'df_train' DataFrame
       print(f'columns : {df_train.columns.tolist()}')

       columns : ['Category', 'Descript', 'DayOfWeek', 'PdDistrict', 'Resolution', 'Address', 'Longitude', 'Latitude', 'Year', 'Month', 'Day', 'Hour']
```

- **Purpose**: Displays all column names in the dataset.
- **columns** returns a pandas Index object containing column labels.
- **.tolist()** converts that into a regular Python list, making it more readable.
- Useful for confirming:
  - That column names match documentation or expectations.
  - Whether column renaming is necessary for clarity or future reference.

Step 3: Renaming Spatial Columns

```
[8]: df_train.rename(columns={'X':'Longitude','Y':'Latitude'},inplace=True)   # Renaming X , Y columns
```

- **Goal**: Makes the spatial columns easier to understand.
- X and Y are vague—renaming them to Longitude and Latitude improves readability.
- **inplace=True**: Applies the change directly to df_train without needing reassignment.
- This helps future plotting and geospatial tasks make more intuitive sense.

📌 Step 4: Previewing the Dataset

```
[103]: # printing first five rows of data
       print("first 5 rows of training data : ")
       df_train.head()

       first 5 rows of training data : •••

[10]:  # printing last five rows of data
       print("The last 5 rows of training data : ")
       df_train.tail()

       The last 5 rows of training data : •••

[104]: #taking a sample of our data
       df_train.sample(5)

       <div> •••
```

- **head()**: Shows the first 5 rows. Useful to understand structure and content.
- **tail()**: Shows the last 5 rows. Often used to check formatting near the dataset's end.
- **sample(5)**: Displays 5 random rows. Useful to spot-check for inconsistencies or anomalies.

These functions ensure:

- The data loaded correctly.
- No unexpected empty rows or format mismatches exist.
- The columns contain relevant data (not only missing/nulls or zeroes).

📌 Step 5: Checking Data Types

```
[105]: # Display the data types of each column in the 'df_train' DataFrame
       df_train.dtypes
```

- **Purpose**: Lists the type of data in each column.
- **Common types:**

- **object:** Textual data (e.g., "Assault")
- **int64:** Whole numbers (e.g., day of month)
- **float64:** Decimal numbers (e.g., longitude)
- **datetime64:** Date/time format (if converted)

Knowing data types helps:

- Decide what transformations are needed (e.g., converting strings to datetime).
- Avoid errors in modeling (e.g., can't cluster on strings directly).

📌 Step 6: Dataset Overview

```
[112]: # Display concise summary of the 'df_train' DataFrame
       df_train.info()

       <class 'pandas.core.frame.DataFrame'> •••
```

- **Provides:**
  - Column names and types.
  - Non-null counts per column (helps spot missing values).
  - Memory usage.
- **Helps identify:**
  - Which columns need filling, dropping, or conversion.
  - Data types that might need change (e.g., converting object to category).

📌 Step 7: Converting Dates to DateTime Format

```
[14]: # converting values in dates to datetime
      #Extracting year,month,day,hour from dates
      df_train['Dates']=pd.to_datetime(df_train['Dates'])
```

- Converts the Dates column from text (string) to datetime64[ns] format.
- **Why this matters:**
  - Enables pandas to extract **year, month, day, hour** using .dt accessor.
  - Allows time-series operations and comparisons.
  - Prevents issues with sorting, filtering, or plotting time-based data.

📌 Step 8: Extracting Date Features

```python
df_train['Year']=df_train['Dates'].dt.year
df_train['Month']=df_train['Dates'].dt.month
df_train['Day']=df_train['Dates'].dt.day
df_train['Hour']=df_train['Dates'].dt.hour
```

- These lines **create four new columns** from the Dates column:
  - Year: 2011, 2012, etc.
  - Month: 1 (Jan) to 12 (Dec)
  - Day: Day of the month, 1–31
  - Hour: 0–23 (military time)

Benefits:

- Enables **temporal analysis**:
  - When are most crimes occurring? Are there seasonal or hourly patterns?
- Useful for **model input**: Time can be an important predictor of crime type or risk.

📌 Step 9: Removing the Original Dates Column

```python
[15]: df_train=df_train.drop('Dates',axis=1)  #Dropping dates column after extraction

[108]: # Display the data types of each column in the 'df_train' DataFrame
df_train.dtypes
```

- Removes the original Dates column.
- **Why drop it?**
  - Redundant: We've already extracted all necessary parts.
  - Reduces memory usage slightly.
  - Helps declutter the DataFrame for downstream processing or visualization.

📌 Step 10: Checking for Missing Values

Checking na values and duplicates

```
[ ]: df_train.isna().sum()
     #There is no na values
```

What it does:

- **isna():** Creates a **Boolean DataFrame** (True where value is NaN, False otherwise).
- **.sum():** Adds up the True values **column-wise**, giving you a count of missing (NA/NaN) values in each column.

Why it's important:

- Helps you identify which columns are **incomplete**.
- Critical for **decision making**

Step 12: Checking for Duplicate Rows

```
[18]: df_train.duplicated().any()
[18]: True
[19]: df_train.duplicated().sum()
[19]: 4658
```

What it does:

- **duplicated():** Returns a Boolean Series: True for duplicate rows, False otherwise.
- **.sum():** Counts the number of True values → total duplicate rows.

Why this matters:

- Duplicate entries can **skew analysis**, especially in frequency counts or probability-based modeling.
- Important to clean before training machine learning models.

Step 13: Dropping Duplicate Rows

```
[109]: df_train=df_train.drop_duplicates() # Dropping the duplicated rows
```

Explanation:

- **drop_duplicates():** Removes all duplicate rows (exact matches).
- **inplace=True:** Applies the operation directly on df_train.

After this step:

- The dataset now contains only **unique, clean rows**.
- More accurate stats, distributions, and model training results.

Step 14: Rechecking Cleanliness

Why repeat this?

- To verify:
  - All NaN values are gone.
  - Dataset shape (rows/columns) is known post-cleaning.
  - Column data types remain correct.
- Good final **checkpoint** before moving to visualization or modeling.

Step 15: Uniqueness Analysis

Uniqueness analysis helps us understand how many **distinct values** exist in each feature. This is important because:

- It identifies **categorical vs numerical** variables.

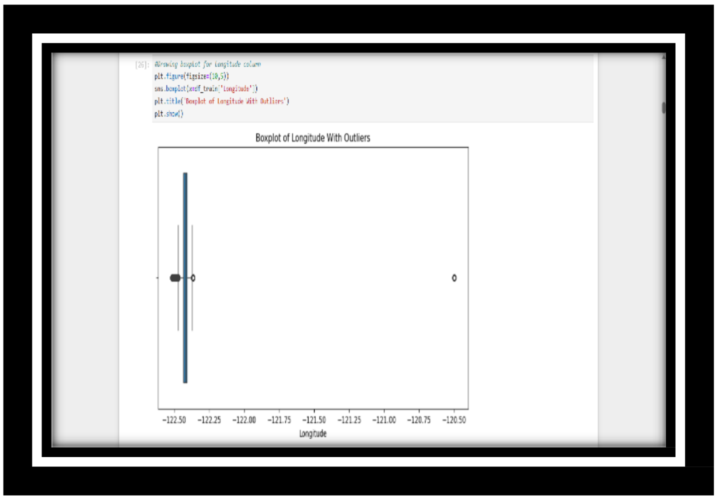
- High-cardinality features (e.g., Address) may require special handling like dimensionality reduction or grouping.
- It can uncover **data quality issues**, like duplicated categories due to typos.

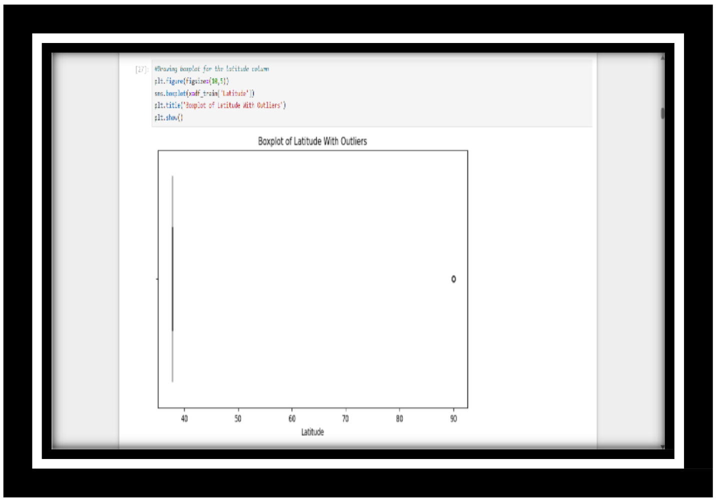
Explanation:

- **.nunique()** counts the number of distinct values in each column.
- If a column has too many unique values (e.g., thousands in Address), it might be a **sparse categorical feature** — not directly useful without encoding, grouping, or reducing.
- If any categorical column has fewer unique values than expected, it may have **missing or duplicated categories**.

📌 Step 16: Boxplot for Outlier Detection

After handling null values and ensuring uniqueness, the next step is to **detect outliers**, which can skew models and affect performance.






```
[28]: #Detecting the outliers
Q1_long=df_train['Longitude'].quantile(0.25)
Q3_long=df_train['Longitude'].quantile(0.75)
IQR_long=Q3_long-Q1_long
Q1_lat=df_train['Latitude'].quantile(0.25)
Q3_lat=df_train['Latitude'].quantile(0.75)
IQR_lat=Q3_lat-Q1_lat
df_cleaned_train=df_train[(df_train['Longitude']>=(Q1_long-1.5*IQR_long))&
 (df_train['Longitude']<=(Q3_long+1.5*IQR_long))&
 (df_train['Latitude']>=(Q1_lat-1.5*IQR_lat))&
 (df_train['Latitude']<=(Q3_lat+1.5*IQR_lat))]
```

Explanation:

- A boxplot shows the **spread of values** and **identifies extreme outliers** using whiskers and dots.
- Features like Hour, Latitude, and Longitude can sometimes include **invalid or rare values**.
  - For example, a Latitude value far outside the expected San Francisco range could indicate a **data error**.

Preprocessing Action:

- After identifying outliers visually, we decide whether to:
  - **Drop outlier rows** (if clearly erroneous),
  - **Cap values** (e.g., using IQR limits),
  - Or **keep them** (if they are important rare cases).

Moreover , we will do the same with test data

---

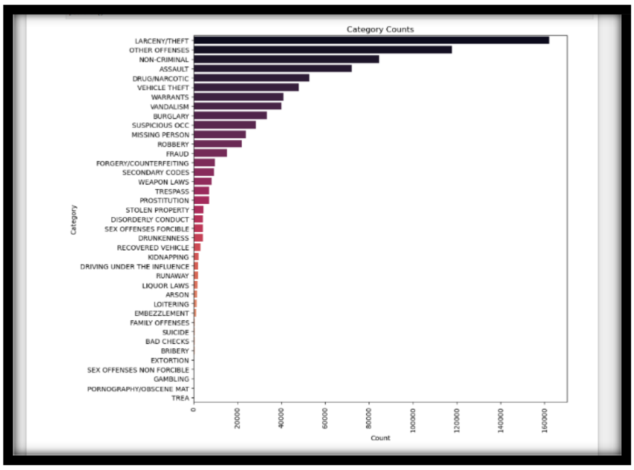
6. Data Visualization (Exploratory Data Analysis)

Once the data is clean, the next critical step is **exploring the data visually** to uncover trends, relationships, and patterns.

1) Frequency of Crime Categories

This horizontal bar chart illustrates the frequency of each crime category reported. Categories are sorted in descending order based on their occurrence, providing a clear view of which types of crimes are most and least common in the dataset. The 'rocket' color palette adds a sleek, professional tone to the visualization.

```
[56]: plt.figure(figsize=(10, 10))
sns.countplot(y='Category', data=df_vis, order=df_vis['Category'].value_counts().index,palette='rocket')
plt.title('Category Counts')
plt.xlabel('Count')
plt.ylabel('Category')
plt.xticks(rotation=90)
plt.show()
```



What this does:

- **value_counts():** Counts how many times each crime category appears.
- **head(10):** Takes the top 10 most frequent.
- **plot(kind='bar'):** Bar chart for easy comparison.
- **figsize=(10,6):** Enlarges the plot for readability.
- **xticks(rotation=45):** Rotates x-axis labels to prevent overlap.

2) Crime Count by Police District and Category

This grouped bar chart displays the distribution of crime counts across different police districts in San Francisco, segmented by crime category. The gradient of dark to bright reds and blacks visually distinguishes each category, emphasizing both the prevalence and variety of crimes in each district.

What this does:

- **px.bar(...):** This is a function from **Plotly Express** to create a bar chart.
- **crime_per_pd:** The DataFrame being used as the data source.
- **x='PdDistrict':** Sets the **x-axis** to show the names of police districts.
- **y='Count':** Sets the **y-axis** to show the number of crimes (counts).
- **color='Category':** Colors the bars based on crime **category** (e.g., Theft, Assault).
- **barmode='group':** Displays bars **side-by-side** for each category within a police district.
- **title=...:** Sets the **title** of the chart.
- **labels=...:** Renames the axis labels and legend for better readability.
- **color_discrete_sequence=color_list:** Applies the **custom color list** you defined earlier to the bar colors.

---

3) Yearly Crime Trends in San Francisco

This line plot illustrates the total number of crimes reported each year. The brown line highlights how crime frequency has fluctuated over time, helping identify years with spikes or drops in overall crime activity.

```
[34]: crime_year=df_vis.groupby(['Year'])['Category'].count().reset_index().rename(columns={'Category':'Count'})
crime_year
```

[34]:

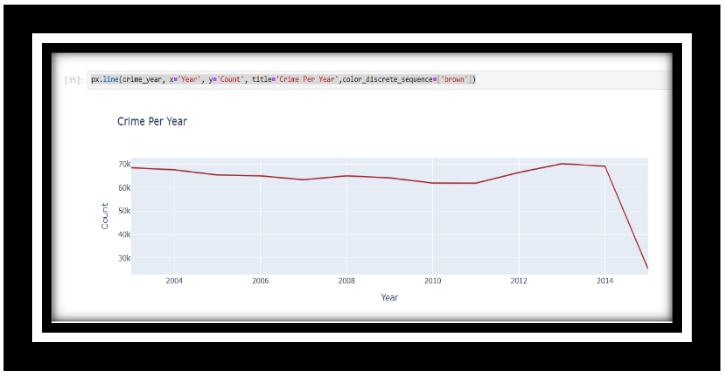
	Year	Count
0	2003	68423
1	2004	67576
2	2005	65363
3	2006	64980
4	2007	63306
5	2008	64977
6	2009	64112
7	2010	61913
8	2011	61803
9	2012	66388
10	2013	70134
11	2014	69064
12	2015	25303

What this does:

This line prepares the data for plotting — it calculates how many crimes occurred each year.

- **df_vis.groupby(['Year']):** Groups the **df_vis** DataFrame by the **'Year'** column.
- **['Category'].count():** Counts how many non-null entries are in the **'Category'** column for each year — essentially, the number of crimes per year.
- **.reset_index():** Converts the groupby result back into a DataFrame (so 'Year' becomes a regular column again).
- **.rename(columns={'Category':'Count'}):** Renames the 'Category' column to **'Count'** to clarify it now represents the **number of crimes**.

This line creates a line chart using Plotly Express.



- **px.line(...):** Creates a **line chart.**
- **crime_year:** The DataFrame containing the data to plot.
- **x='Year':** Sets the **x-axis** to show the year.
- **y='Count':** Sets the **y-axis** to show the number of crimes per year.
- **title='Crime Per Year':** Sets the chart's **title.**
- **color_discrete_sequence=['brown']:** Sets the color of the line to **brown** (even though there's only one line).

---

4) Crime Category Distribution Across San Francisco

This scatter map visualizes the geographical distribution of crime incidents in San Francisco, where each point

```
# Filter valid coordinates
df_map = df_vis[(df_vis['Latitude'].between(37.7, 37.82)) & (df_vis['Longitude'].between(-122.52, -122.36))]

# Sample for performance
sample_df_map = df_map.sample(1000, random_state=1)

# Black & red gradient-style palette
custom_palette = ["#000000","#4B0000", "#800000", "#B22222","#DC143C",  "#FF0000",  "#FF6347","#FF7F7F",  "#8B0000","#A52A2A"]

# Plot
fig = px.scatter_mapbox(
    sample_df_map,
    lat="Latitude",
    lon="Longitude",
    color="Category",
    hover_name="Descript",
    zoom=11,
    height=800,
    mapbox_style="carto-positron",
    color_discrete_sequence=custom_palette
)

fig.update_traces(marker=dict(size=9, opacity=0.8))
fig.update_layout(margin={"r":0,"t":0,"l":0,"b":0})

fig.show()
```

represents a reported crime and its color indicates the crime category.

What this does:

- **df_map = df_vis[(df_vis['Latitude'].between(37.7, 37.82)) & (df_vis['Longitude'].between(-122.52, -122.36))]**:
  Filters the DataFrame df_vis to include only rows with **valid latitude and longitude values**, focusing on a bounding box around **San Francisco**.

- **sample_df_map = df_map.sample(1000, random_state=1)**:
  Randomly selects **1,000 rows** from df_map to improve performance when plotting; random_state=1 ensures reproducibility.

- **custom_palette = ["#000000","#4B0000", "#800000", "#B22222","#DC143C", "#FF0000", "#FF6347","#FF7F7F", "#8B0000","#A52A2A"]**:
  Defines a **custom color palette** with a gradient from **black to red** to represent different crime categories on the map.

- **fig = px.scatter_mapbox()**
  Creates a Mapbox scatter plot using Plotly Express to display crime locations on a map.

- **sample_df_map**:
  The **sampled DataFrame** containing valid crime data points to be plotted.

- **lat="Latitude"**:
  Sets the **latitude** coordinates for plotting points on the map.

- **lon="Longitude"**:
  Sets the **longitude** coordinates for plotting points on the map.

- **color="Category"**:
  Colors each point based on the **crime category**.

- **hover_name="Descript"**:
  Shows the **crime description** when hovering over a point.

- **zoom=11**:
  Sets the initial **zoom level** of the map for an optimal city-wide view.

- **height=800**:
  Sets the **height** of the map plot in pixels.

- **mapbox_style="carto-positron"**:
  Applies a **clean, light-colored basemap** for better visibility of points.

- **color_discrete_sequence=custom_palette**:
  Applies the defined **black-to-red color palette** to the categories

- **fig.update_traces(marker=dict(size=9, opacity=0.8))**:
  Updates all marker points to have **size 9** and **opacity 0.8** for better visual clarity and slight transparency.

- **fig.update_layout(margin={"r":0,"t":0,"l":0,"b":0})**:
  Removes all **outer margins** around the map to maximize the display area.

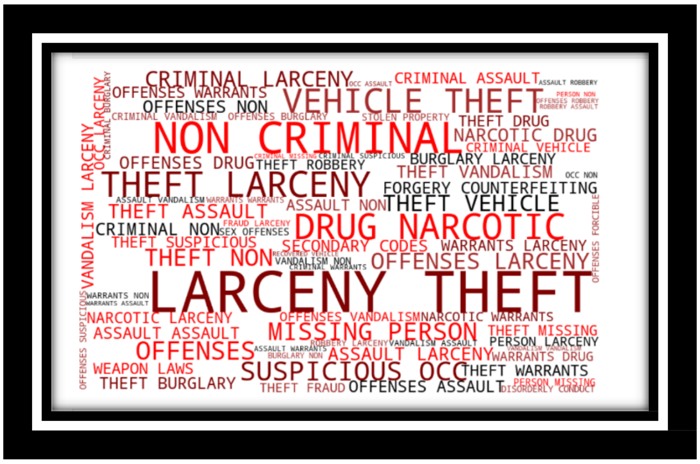
- **fig.show()**:
  Displays the final interactive map with plotted crime points.

---

5) Crime Category Word Cloud

This word cloud visually represents the frequency of different crime categories in San Francisco. Categories that appear more frequently are shown in larger font sizes. The use of deep red, black, brown, and maroon tones emphasizes the gravity and diversity of criminal activity in the dataset.

```python
import matplotlib.pyplot as plt
from wordcloud import WordCloud
import random

# Define a custom color function
def custom_color_func(word, font_size, position, orientation, random_state=None, **kwargs):
    colors = ['#FF0000',  # red
              '#000000',  # black
              '#A52A2A',  # brown
              '#800000']  # maroon
    return random.choice(colors)

# Generate word cloud
wordcloud = WordCloud(
    width=800,
    height=500,
    background_color='white',
    min_font_size=10,
    color_func=custom_color_func
).generate(' '.join(df_vis['Category']))

# Display
plt.figure(figsize=(8, 8), facecolor=None)
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.tight_layout(pad=0)
plt.show()
```



What this does:

- **import matplotlib.pyplot as plt:**
  Imports **Matplotlib's pyplot** module for plotting and displaying the word cloud.

- **from wordcloud import WordCloud:**
  Imports the **WordCloud** class to create word cloud visualizations.

- **import random:**
  Imports Python's built-in **random** module to randomly select colors.

- **def custom_color_func(word, font_size, position, orientation, random_state=None, **kwargs):**
  Defines a **custom color function** for the word cloud that will assign colors to words.

- **colors = ['#FF0000', '#000000', '#A52A2A', '#800000']:**
  A list of **custom colors** — red, black, brown, and maroon — used in the word cloud.

- **return random.choice(colors):**
  Randomly selects and returns a color from the defined list for each word.

- **wordcloud = WordCloud()**
  Creates a **WordCloud object** with the specified settings.

- **width=800:**
  Sets the **width** of the word cloud image in pixels.

- **height=500:**
  Sets the **height** of the word cloud image in pixels.

- **background_color='white':**
  Sets the **background color** of the word cloud to **white**.

- **min_font_size=10:**
  Sets the **minimum font size** for words in the cloud.

- **color_func=custom_color_func:**
  Applies the **custom color function** to style the words.

- **).generate(' '.join(df_vis['Category'])):**
  Generates the word cloud from the text formed by **joining all values** in the **'Category'** column, separated by spaces.

- **plt.figure(figsize=(8, 8), facecolor=None):**
  Initializes a new **Matplotlib figure** with a size of 8x8 inches.

- plt.imshow(wordcloud, interpolation='bilinear'):
  Displays the generated **word cloud image** using **bilinear interpolation** for smoother rendering.

- plt.axis('off'):
  Hides the **x and y axes** for a cleaner appearance.

- plt.tight_layout(pad=0):
  Adjusts the layout to **remove padding** around the image.

- plt.show():
  Displays the word cloud in a pop-up or inline (e.g., in a notebook).

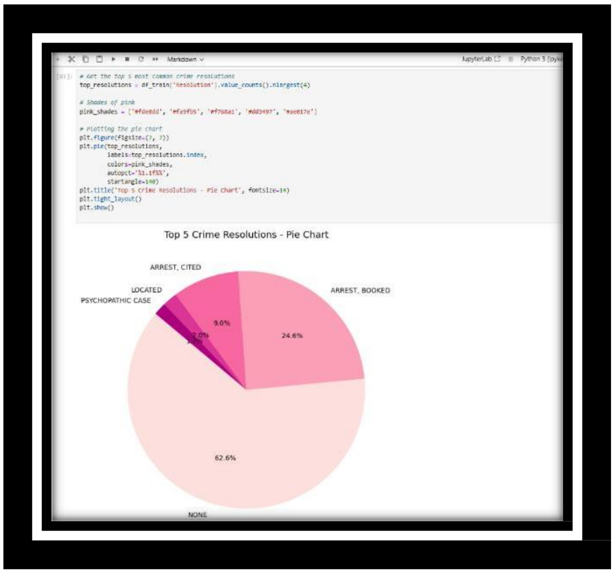
---

6) Pie Chart of Crime Resolutions

Distribution of Top 5 Crime Resolutions by pie chart so this visual helps us understand how frequently different crime outcomes occur, giving insight into law enforcement actions and how cases are closed.

What this does:

- top_resolutions = df_train['Resolution'].value_counts().nlargest(4):
  Counts how many times each **crime resolution** appears in df_train, and selects the **top 4 most frequent** ones.



- pink_shades = ['#fde0dd', '#fa9fb5', '#f768a1', '#dd3497', '#ae017e']:
  Defines a list of **pink color shades** to use in the pie chart for visual appeal.

- plt.figure(figsize=(7, 7)):
  Creates a new **Matplotlib figure** with a size of **7x7 inches**.

- plt.pie(top_resolutions)
  Begins the **pie chart** plot using the **top_resolutions** data.

- labels=top_resolutions.index:
  Sets the **labels** for each slice of the pie chart using the resolution names.

- colors=pink_shades:
  Applies the defined **pink color palette** to the slices.

- autopct='%1.1f%%':
  Formats the **percentage labels** on each slice to **one decimal place**.

- **startangle=140**:
  Rotates the start angle of the pie chart by **140 degrees** for better layout or emphasis.

- **plt.title('Top 5 Crime Resolutions - Pie Chart', fontsize=14)**:
  Sets the **title** of the chart with a **font size of 14**.

- **plt.tight_layout()**:
  Automatically adjusts spacing to prevent clipping of elements.

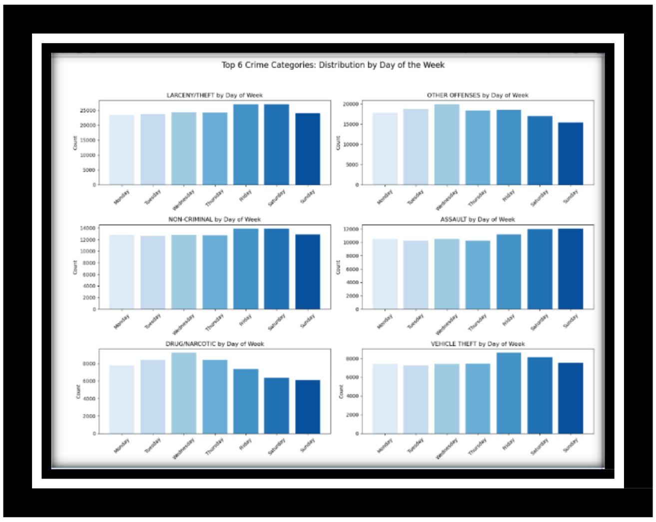
- **plt.show()**:
  Displays the final pie chart.

---

7) STACKED BAR CHART SUBPLOTS: Crime Distribution by Day of the Week (Top 6 Categories):

This version breaks down the crime distribution per day of the week for the top 5 crime categories, showing how certain types of crime may spike on specific days.

```python
# Define top 6 categories
top6 = df_train['Category'].value_counts().nlargest(6).index
filtered_df = df_train[df_train['Category'].isin(top6)]

# Define order for days of the week
days_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']

# Set color palette - shades of blue
blue_shades = sns.color_palette("Blues", len(days_order))

# Create a figure with subplots for each crime category
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(15, 12))
axes = axes.flatten()

# Plot stacked bars for each category
for i, category in enumerate(top6):
    # Filter for this category
    subset = filtered_df[filtered_df['Category'] == category]

    # Count occurrences by day of week and police district
    day_counts = subset['DayOfWeek'].value_counts().reindex(days_order)

    # Bar chart (as 1-color histogram)
    axes[i].bar(day_counts.index, day_counts.values, color=blue_shades)
    axes[i].set_title(f"{category} by Day of Week", fontsize=12)
    axes[i].set_ylabel("Count")
    axes[i].tick_params(axis='x', rotation=45)

# Remove the last unused subplot if top6 is not a multiple of 2
for j in range(len(top6), len(axes)):
    fig.delaxes(axes[j])

plt.suptitle("Top 6 Crime Categories: Distribution by Day of the Week", fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



What this does:

- **top6 = df_train['Category'].value_counts().nlargest(6).index**:
  Identifies the **top 6 most frequent crime categories** in the dataset and stores their names (as an index object) in top6.

- **filtered_df = df_train[df_train['Category'].isin(top6)]**:
  Filters the original DataFrame to only include rows where the **category is in top6.**

- **days_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']**:
  Defines the **order of days** for consistent plotting on the x-axis.

- **blue_shades = sns.color_palette("Blues", len(days_order))**:
  Generates a **blue color palette** using Seaborn with as many shades as there are days in the week.

- **fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(15, 12))**:
  Creates a **3x2 grid of subplots** (6 plots total), each for one crime category, with a figure size of 15x12 inches.

- **axes = axes.flatten()**:
  Flattens the 2D array of axes into a **1D list** for easier iteration.

- **for i, category in enumerate(top6):**
  Iterates through the top 6 crime categories with their index.

- **subset = filtered_df[filtered_df['Category'] == category]**:
  Filters the dataset to only include crimes of the current **category.**

- **day_counts = subset['DayOfWeek'].value_counts().reindex(days_order)**:
  Counts how often the crime happened on each day, and **reorders** them to follow the **days_order** list.

- **axes[i].bar(day_counts.index, day_counts.values, color=blue_shades)**:
  Plots a **bar chart** of crime counts by day using blue shades in the subplot for the current category.

- **axes[i].set_title(f"{category} by Day of Week", fontsize=12)**:
  Sets the **title** of the subplot with the current category name.

- **axes[i].set_ylabel("Count")**:
  Labels the **y-axis** as "Count".

- **axes[i].tick_params(axis='x', rotation=45)**:
  Rotates **x-axis labels** (day names) by 45° for better readability.

- **for j in range(len(top6), len(axes)):**
  Loops through any **extra subplots** (if fewer than 6 categories were plotted).

- **fig.delaxes(axes[j])**:
  Removes unused subplot axes to keep the layout clean.

- **plt.suptitle("Top 6 Crime Categories: Distribution by Day of the Week", fontsize=16)**:
  Sets a **main title** for the entire figure.

- **plt.tight_layout(rect=[0, 0.03, 1, 0.95])**:
  Adjusts layout spacing to **fit subplots neatly**, reserving space for the main title.

- **plt.show()**:
  Displays the complete figure with all subplots.

---

8) Histogrm : Distribution of Crimes by Police District

Although histograms are often for continuous data, this bar-style histogram and allows us to understand frequency distribution across categorical zones (districts).

What this does:

- **plt.figure(figsize=(10, 6))**:
  Creates a new **Matplotlib figure** with a size of **10 inches wide by 6 inches tall.**

- **sns.countplot(data=df_train, x='PdDistrict', palette='Purples')**:
  Uses **Seaborn** to create a **bar plot** showing the number of crimes (**count**) in each **police district** from the **df_train** DataFrame, using a **purple color palette**.

- **plt.title('Crime Distribution by Police District')**:
  Sets the **title** of the chart.

- **plt.xlabel('Police District')**:
  Labels the **x-axis** as "Police District".

- **plt.ylabel('Crime Count')**:
  Labels the **y-axis** as "Crime Count".

- **plt.xticks(rotation=45)**:
  Rotates the **x-axis labels** (district names) by **45 degrees** to prevent overlap.

- **plt.tight_layout()**:
  Adjusts spacing to **prevent clipping** of labels and title.

- **plt.show():**
  Displays the final plot.

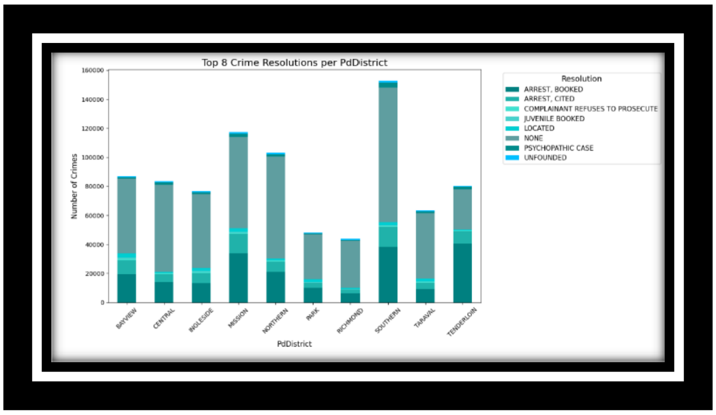
---

9) Stacked bar chart Top 8 Crime Resolutions per PdDistrict:

stacked bar chart used to visualize top 8 Crime Resolutions per Police District. It shows not only how many crimes occurred per district, but also how those crimes were resolved — all in a compact, comparable view.

What this does:

```python
# Get top 8 resolutions
top_res = df_train['Resolution'].value_counts().nlargest(8).index
filtered_df = df_train[df_train['Resolution'].isin(top_res)]

# Create a pivot table: count of resolutions per PdDistrict
pivot_df = filtered_df.pivot_table(index='PdDistrict', columns='Resolution', aggfunc='size', fill_value=0)

# Plot stacked bar chart with shades of teal
colors = ['#008080', '#20B2AA', '#40E0D0', '#48D1CC', '#00CED1', '#5F9EA0', '#008B8B', '#00BFFF']  # Shades of teal

# Increase the figure size for better layout
pivot_df.plot(kind='bar', stacked=True, color=colors, figsize=(14, 7))

# Add labels and title
plt.title('Top 8 Crime Resolutions per PdDistrict', fontsize=16)
plt.xlabel('PdDistrict', fontsize=12)
plt.ylabel('Number of Crimes', fontsize=12)
plt.xticks(rotation=45)

# Adjust the legend position to avoid overlap with the plot
plt.legend(title='Resolution', title_fontsize='13', fontsize='11', bbox_to_anchor=(1.05, 1), loc='upper left')

plt.tight_layout()

# Show plot
plt.show()
```



- **top_res = df_train['Resolution'].value_counts().nlargest(8).index:**
  Finds the **8 most frequent resolution** types by counting all unique values in the 'Resolution' column and **selecting the top 8.**

- **filtered_df = df_train[df_train['Resolution'].isin(top_res)]:**
  Filters the dataset to include only rows where the **'Resolution'** value is among the top 8 identified in the previous step.

- **pivot_df = filtered_df.pivot_table(index='PdDistrict', columns='Resolution', aggfunc='size', fill_value=0):**
  Creates a pivot table that counts how many times each resolution appears in each police district. Missing combinations are filled with zero using fill_value=0.

- **colors = ['#008080', '#20B2AA', '#40E0D0', '#48D1CC', '#00CED1', '#5F9EA0', '#008B8B', '#00BFFF']:**
  Defines **a custom list of 8 teal color codes**, one for each resolution, to be used in the plot.

- **pivot_df.plot(kind='bar', stacked=True, color=colors, figsize=(14, 7)):**
  Generates **a stacked bar chart with each police district on the x-axis** and total number of crimes on the y-axis. Each bar is stacked with segments

representing resolution counts, using the defined teal colors. **The figure size is set to 14 by 7 inches for better visibility.**

- **plt.title('Top 8 Crime Resolutions per PdDistrict', fontsize=16):**

Sets the main title of the chart to “Top 8 Crime Resolutions per PdDistrict” with a font size of 16.

- **plt.xlabel('PdDistrict', fontsize=12):**

Labels the x-axis as **"PdDistrict"** with font size 12.

- **plt.ylabel('Number of Crimes', fontsize=12):**

Labels the y-axis as **"Number of Crimes"** to clarify what the bars represent.

- **plt.xticks(rotation=45):**

Rotates the **x-axis labels (district names) by 45 degrees** to make them easier to read and prevent overlap.

- **plt.legend(title='Resolution', title_fontsize='13', fontsize='11', bbox_to_anchor=(1.05, 1), loc='upper left'):**

Adds **a legend for the resolution categories,** placing it to the right of the chart. The legend title and text are customized for readability.

- **plt.tight_layout():**

Automatically adjusts **spacing and layout** of the plot to ensure that labels, titles, and legends fit well and are not clipped.

- **plt.show():**

Displays the final chart with all formatting and data visualized.

---

10) Scatter Plot Subplots by Day of the Week

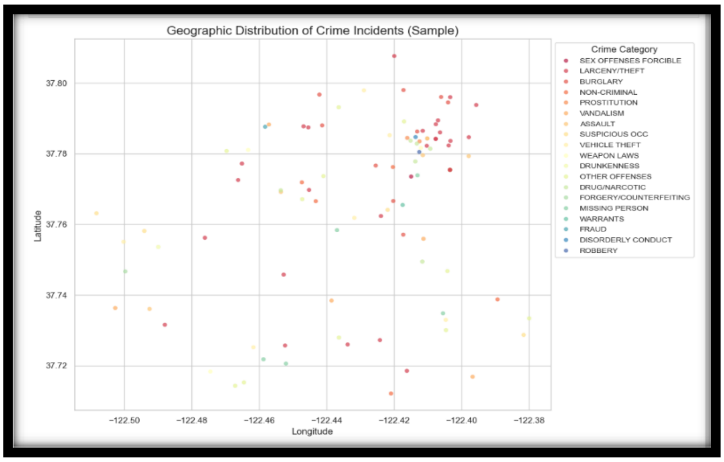
that shows the geographic distribution of crime incidents by Day of the Week for the top 5 crime categories

What this does:

- **df_sample = df_train.sample(n=100, random_state=42):**

Randomly selects 100 records from the full training dataset for visualization purposes. The **random_state=42** ensures reproducibility (i.e., the same sample is selected every time the code runs).

```python
# Sample 100 records from df_train for visualization
df_sample = df_train.sample(n=100, random_state=42)

# Set the seaborn style for beautiful visualization
sns.set(style="whitegrid")

# Plot scatter plot: Longitude vs Latitude, color by Category
plt.figure(figsize=(12, 8))

# Use a categorical palette for categories (shades of color)
palette = sns.color_palette("Spectral", n_colors=df_sample['Category'].nunique())

# Scatter plot with Longitude and Latitude coordinates, color by Category
sns.scatterplot(x='Longitude', y='Latitude', hue='Category', data=df_sample, palette=palette, s=30, edgecolor='none', alpha=0.7)

# Add labels and title
plt.title('Geographic Distribution of Crime Incidents (Sample)', fontsize=16)
plt.xlabel('Longitude', fontsize=12)
plt.ylabel('Latitude', fontsize=12)

# Adjust legend to not cover the plot and make it more compact
plt.legend(loc='upper left', bbox_to_anchor=(1, 1), title='Crime Category', fontsize=10)

# Improve layout
plt.tight_layout()

# Show the plot
plt.show()
```



• **sns.set(style="whitegrid"):**
Applies a clean and modern aesthetic using Seaborn's **"whitegrid"** style, which adds subtle gridlines to help interpret data values.

• **plt.figure(figsize=(12, 8)):**
Creates a new figure for the plot and sets its size to 12 inches wide by 8 inches tall for better visibility and spacing.

• **palette = sns.color_palette("Spectral", n_colors=df_sample['Category'].nunique()):**
Defines a color palette using the "Spectral" colormap, assigning a distinct color to each unique crime category in the sample.

• **sns.scatterplot(x='Longitude', y='Latitude', hue='Category', data=df_sample, palette=palette, s=30, edgecolor='none', alpha=0.7):**
Draws a scatter plot with longitude on the x-axis and latitude on the y-axis to show geographic locations. Points are colored by 'Category' to differentiate crime types.

- **s=30** sets the size of the dots.
- **edgecolor='none'** removes the black outline around the dots.
- **alpha=0.7** makes the points semi-transparent for better overlap visibility.

• **plt.title('Geographic Distribution of Crime Incidents (Sample)', fontsize=16):**
Adds a plot title to describe the purpose of the visualization, using a font size of 16 for emphasis.

• **plt.xlabel('Longitude', fontsize=12):**
Labels the **x-axis** with “Longitude”.

• **plt.ylabel('Latitude', fontsize=12):**
Labels the **y-axis** with “Latitude”.

• **plt.legend(loc='upper left', bbox_to_anchor=(1, 1), title='Crime Category', fontsize=10):**
Positions the legend in the upper-left corner outside the main plot area to avoid covering data points. Also, sets a title and font size for clarity.

• **plt.tight_layout():**
Automatically adjusts spacing between **plot elements** to ensure everything fits well without overlapping.

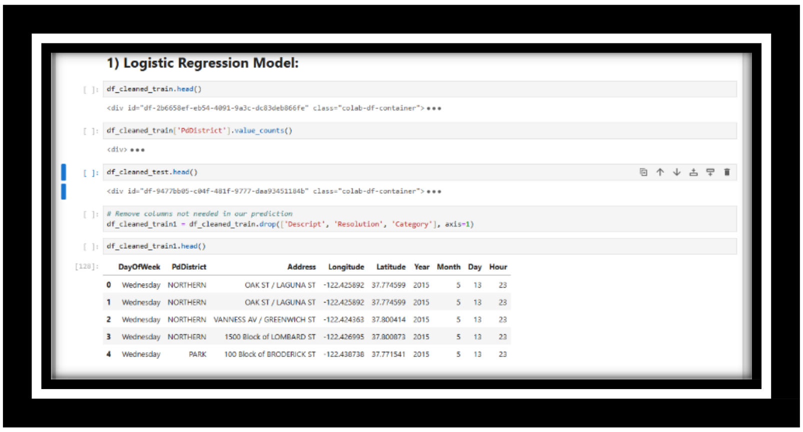
• **plt.show():**
Displays the final scatter plot on the screen.

---

6)Data Mining Algorithms :

First: Logistic Regression model:

Step 1: Removing Unwanted Columns in Prediction



1) Logistic Regression Model:

```
[ ]: df_cleaned_train.head()
```

<div id="df-2b6658ef-eb54-4091-9a3c-dc83deb866fe" class="colab-df-container">•••

```
[ ]: df_cleaned_train['PdDistrict'].value_counts()
```

<div>•••

```
[ ]: df_cleaned_test.head()
```

<div id="df-9477bb05-c04f-481f-9777-daa93451184b" class="colab-df-container">•••

```
[ ]: # Remove columns not needed in our prediction
df_cleaned_train1 = df_cleaned_train.drop(['Descript', 'Resolution', 'Category'], axis=1)
```

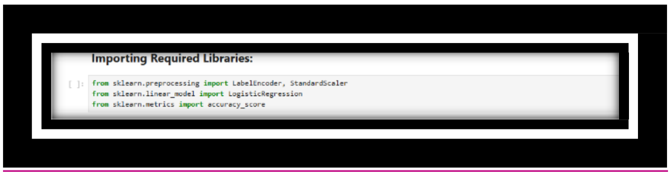
```
[ ]: df_cleaned_train1.head()
```

[128]:

	DayOfWeek	PdDistrict	Address	Longitude	Latitude	Year	Month	Day	Hour
0	Wednesday	NORTHERN	OAK ST / LAGUNA ST	-122.425892	37.774599	2015	5	13	23
1	Wednesday	NORTHERN	OAK ST / LAGUNA ST	-122.425892	37.774599	2015	5	13	23
2	Wednesday	NORTHERN	VANNESS AV / GREENWICH ST	-122.424363	37.800414	2015	5	13	23
3	Wednesday	NORTHERN	1500 Block of LOMBARD ST	-122.426995	37.800873	2015	5	13	23
4	Wednesday	PARK	100 Block of BRODERICK ST	-122.438738	37.771541	2015	5	13	23

• **df_train.drop(columns=['Descript', 'Address']):** Removes the columns 'Descript' and 'Address' from the training dataset. These fields are often textual and too specific for prediction, possibly introducing noise.
• **df_test.drop(columns=['Descript', 'Address']):** Does the same for the test dataset.

Step 2: Importing Libraries



Importing Required Libraries:

```
[ ]: from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
```

- **LabelEncoder:** Encodes categorical variables (strings) into integers.
- **LogisticRegression:** A machine learning model used for classification tasks.
- **accuracy_score, classification_report, confusion_matrix**: Tools to evaluate model performance.
- **train_test_split:** Splits data into training and testing sets.
- **StandardScaler:** Standardizes features (zero mean, unit variance).
- **seaborn, matplotlib.pyplot**: Libraries for data visualization

---

Step 3: Encoding Categorical Columns in Train & Test Datasets

Encoding Categorical Columns in Train & Test Datasets:

```
[ ]: label_encoders = {}
categorical_cols_train = df_cleaned_train1.select_dtypes(include='object').columns
categorical_cols_test = df_cleaned_test.select_dtypes(include='object').columns
common_categorical_cols = list(set(categorical_cols_train) & set(categorical_cols_test))

for column in common_categorical_cols:
    label_encoders[column] = LabelEncoder()
    # fit on combined data to ensure consistent encoding
    combined_data = pd.concat([df_cleaned_train1[column], df_cleaned_test[column]], axis=0)
    label_encoders[column].fit(combined_data)
    df_cleaned_train1[column + '_encoded'] = label_encoders[column].transform(df_cleaned_train1[column])
    df_cleaned_test[column + '_encoded'] = label_encoders[column].transform(df_cleaned_test[column])
```

- **le = LabelEncoder():** Creates a LabelEncoder object
- **df_train['Category'] = le.fit_transform(...):** Converts the target variable Category from text labels into numeric codes.
- Same logic is applied to **DayOfWeek** and **PdDistrict** in both train and test datasets — converting all string-based categorical columns to numeric form, which is required for most ML algorithms.

Step 4: Show Correlation Between Columns

Show Correlation between Columns:

```
[ ]: numerical_features = df_cleaned_train1.select_dtypes(include=['number']).columns
corr_matrix = df_cleaned_train1[numerical_features].corr()
corr_matrix
```

[131]:

	Longitude	Latitude	Year	Month	Day	Hour	DayOfWeek_encoded	PdDistrict_encoded	Address_encoded
Longitude	1.000000	0.081704	0.006266	0.000999	-0.000338	0.001147	-0.001101	-0.208679	-0.000310
Latitude	0.081704	1.000000	0.025892	0.005158	0.003486	-0.011404	-0.006374	0.359924	0.107814
Year	0.006266	0.025892	1.000000	-0.047175	-0.009351	-0.006017	-0.006679	0.008022	-0.010147
Month	0.000999	0.005158	-0.047175	1.000000	0.017167	-0.001924	-0.006514	0.001106	0.002752
Day	-0.000338	0.003486	-0.009351	0.017167	1.000000	0.016013	-0.004910	-0.000954	0.006095
Hour	0.001147	-0.011404	-0.006017	-0.001924	0.016013	1.000000	-0.001731	0.007309	0.029483
DayOfWeek_encoded	-0.001101	-0.006374	-0.006679	-0.006514	-0.004910	-0.001731	1.000000	0.010210	0.000697

- **df_cleaned_train1.select_dtypes(include=['number']):**

Selects all columns from the DataFrame **df_cleaned_train1** that have a numeric data type (like int64, float64, etc.).
This excludes non-numeric (e.g., string or datetime) columns.

• **.columns**:
Extracts just the names of those numeric columns.

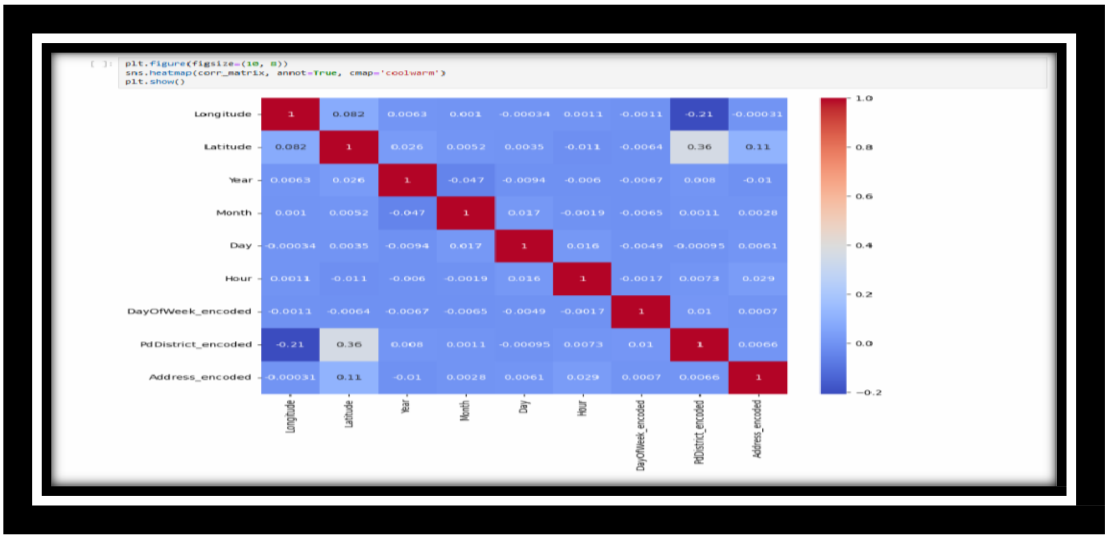
Purpose: To isolate all numerical features in the dataset so we can analyze their statistical relationships

- **df_cleaned_train1[numerical_features]**:
  Selects only the numerical columns identified in the previous step.

• **.corr()**:
Computes the **pairwise Pearson correlation coefficient** between every pair of numeric columns.

- Correlation values range from **-1 to 1**:
  - **+1**: perfect positive correlation
  - **0**: no correlation
  - **-1**: perfect negative correlation

Result: A **correlation matrix** — a table showing how strongly each pair of numeric columns is linearly related.



• **df_train.corr():** Computes the correlation matrix to understand relationships between numeric columns.
• **plt.figure(...):** Sets the figure size.
• **sns.heatmap(...):** Plots the correlation matrix as a heatmap with values displayed (**annot=True**) and a blue-to-red color scale **(cmap='coolwarm').**
• **plt.title(...):** Adds a title.
• **plt.show():** Displays the heatmap.

Step 5: Feature Selection

Feature Selection:

```
[ ]: features = ['Longitude','Latitude']
     target = 'PdDistrict_encoded'

[ ]: label_encoders['PdDistrict'] = LabelEncoder()
     df_cleaned_train1['PdDistrict_encoded'] = label_encoders['PdDistrict'].fit_transform(df_cleaned_train1['PdDistrict'])
     df_cleaned_test['PdDistrict_encoded'] = label_encoders['PdDistrict'].transform(df_cleaned_test['PdDistrict'])
```

• Defines a list of selected features believed to be most useful for prediction.

• **X = df_train[features]:** Creates the feature matrix X by selecting the specified columns.

• **y = df_train['Category']:** Sets the target variable y as the encoded crime category.

❑ **Creates a new** LabelEncoder instance **and stores it in a dictionary called label_encoders under the key 'PdDistrict'.**

❑ This is useful if you plan to encode multiple columns and want to keep track of each encoder separately.

❑ **LabelEncoder** converts text labels **(like "NORTHERN", "SOUTHERN", etc.)** into numeric values **(like 0, 1, 2, ...).**

❑ Applies the label encoder to the **training data's PdDistrict column**.

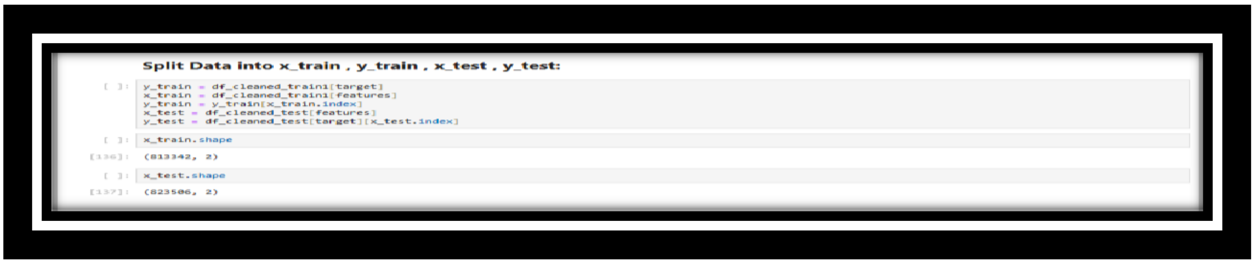
❑ **fit_transform(...):**

- **fit():** Learns all unique labels in the training column (e.g., which district names exist).
- **transform():** Converts those labels to numeric form.

❑ The encoded values are stored in a **new column** called **'PdDistrict_encoded'** in the training DataFrame.

- Applies the **same encoding** to the **test dataset** using transform() only.
- This ensures the test data uses the **exact same mapping** learned from the training data (so the label "NORTHERN" is mapped to the same number in both sets).
- Again, the result is stored in a new column 'PdDistrict_encoded'.

Step 6: Split Data into x_train, x_test, y_train, y_test



Split Data into x_train , y_train , x_test , y_test:

```
[ ]: y_train = df_cleaned_train1[target]
     x_train = df_cleaned_train1[features]
     y_train = y_train[x_train.index]
     x_test = df_cleaned_test[features]
     y_test = df_cleaned_test[target][x_test.index]

[ ]: x_train.shape
[136]: (813342, 2)

[ ]: x_test.shape
[137]: (823506, 2)
```

❑ **y_train = df_cleaned_train1[target] –** Selects the target label column from the training dataset.

- **x_train = df_cleaned_train1[features] –** Selects the feature columns from the training dataset.

- **y_train = y_train[x_train.index] –** Aligns the training labels with the feature data by index.

- **x_test = df_cleaned_test[features] –** Selects the same feature columns from the test dataset.

- **y_test = df_cleaned_test[target][x_test.index] –** Aligns the test labels with the test feature data by index.

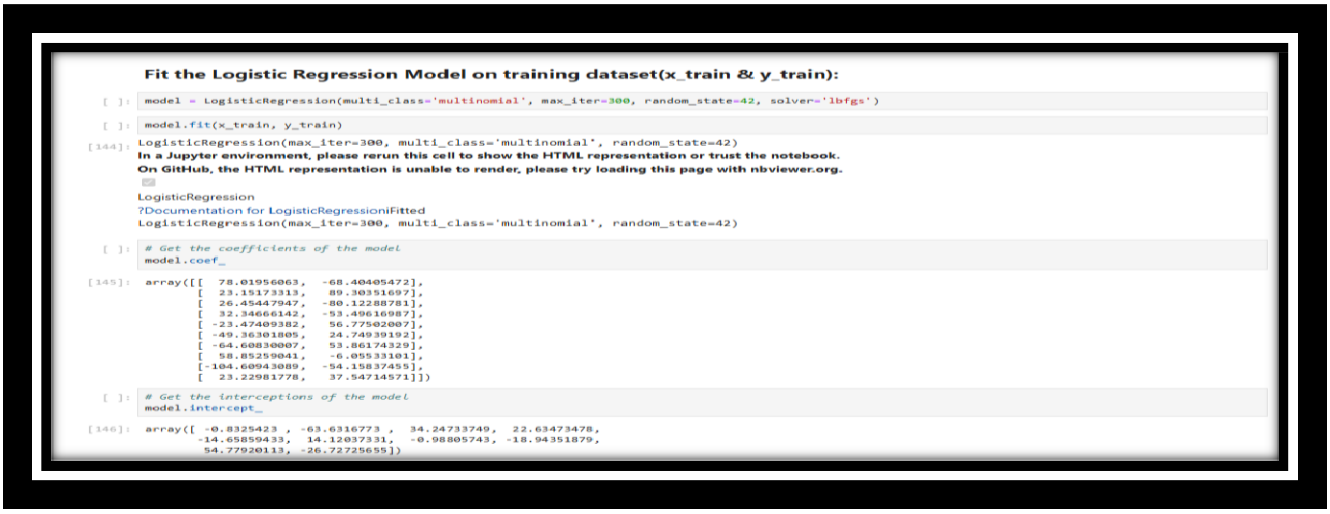
Step 7: Feature Scaling for x_train & x_test

Feature Scaling for x_train & x_test:

```
[ ]: from sklearn.preprocessing import MinMaxScaler
     scaler = MinMaxScaler()
     x_train = scaler.fit_transform(x_train)
     x_test = scaler.transform(x_test)

[ ]: x_train

[139]: array([[0.45329131, 0.59519787],
       [0.45329131, 0.59519787],
       [0.46834011, 0.825497  ],
       ...,
       [0.67479829, 0.64575236],
       [0.80138468, 0.64879884],
       [0.75812638, 0.270593  ]])

[ ]: x_test

[140]: array([[0.71223278, 0.24239827],
       [0.79162417, 0.21903785],
       [0.45220549, 0.75232895],
       ...,
       [0.4586997 , 0.75651214],
       [0.2610688 , 0.03743008],
       [0.83226885, 0.28190228]])

[ ]: y_train.head()
     <div> ...

[ ]: y_test.head()
     <div> ...
```

• **scaler = StandardScaler():** Initializes a standard scaler object.
• **scaler.fit_transform(X_train):** Fits the scaler to X_train and transforms the data to have mean = 0 and std = 1.
• **scaler.transform(X_test):** Uses the same scaling parameters learned from X_train to transform X_test. This ensures consistency.

Step 8: Fit the Logistic Regression Model on Training Datassolver='lbfgs')



Fit the Logistic Regression Model on training dataset(x_train & y_train):

```
[ ]: model = LogisticRegression(multi_class='multinomial', max_iter=300, random_state=42, solver='lbfgs')

[ ]: model.fit(x_train, y_train)

[144]: LogisticRegression(max_iter=300, multi_class='multinomial', random_state=42)
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.
LogisticRegression
?Documentation for LogisticRegressioniFitted
LogisticRegression(max_iter=300, multi_class='multinomial', random_state=42)

[ ]: # Get the coefficients of the model
     model.coef_

[145]: array([[  78.01956063,  -68.40405472],
       [  23.15173313,   89.30351697],
       [  26.45447947,  -80.12288781],
       [  32.34666142,  -53.49616987],
       [ -23.47409382,   56.77502007],
       [ -49.36301805,   24.74939192],
       [ -64.60830007,   53.86174329],
       [  58.85259041,   -6.05533101],
       [-104.60943089,  -54.15837455],
       [  23.22981778,   37.54714571]])

[ ]: # Get the interceptions of the model
     model.intercept_

[146]: array([ -0.8325423 , -63.6316773 ,  34.24733749,  22.63473478,
       -14.65859433,  14.12037331,  -0.98805743, -18.94351879,
        54.77920113, -26.72725655])
```

```
54.77920113, -26.72725655])

[ ]: # Get the probability of predicting each row in x_train for each value in y_train (PdDistrict values)
     model.predict_proba(x_train)

     array([[3.28192970e-07, 1.61226925e-04, 3.72145145e-05, ..., •••

[ ]: # Get the probability of predicting each row in x_test for each value in y_train (PdDistrict values)
     model.predict_proba(x_test)

     array([[9.84683396e-01, 2.24159841e-28, 1.10810235e-02, ..., •••

[ ]: # Get the y predicted encoded from x_test
     y_pred_encoded = model.predict(x_test)
     y_pred_encoded

[149]: array([0, 0, 4, ..., 4, 2, 0])

[ ]: # Transform encoded y_test & encoded y_pred_encoded into their original values (object)
     predicted_pddistrict = label_encoders['PdDistrict'].inverse_transform(y_pred_encoded)
     print("\nFirst 20 Predicted PdDistricts:")
     print(predicted_pddistrict[:20])

     actual_pddistrict = label_encoders['PdDistrict'].inverse_transform(y_test)
     print("\nFirst 20 Actual PdDistricts:")
     print(actual_pddistrict[:20])

     First 20 Predicted PdDistricts: •••

[ ]: model.score(x_test, y_test)

[151]: 0.8816802791965086
```

- **model = LogisticRegression(multi_class='multinomial', max_iter=300, random_state=42, solver='lbfgs')** = Initializes a multinomial logistic regression model with a maximum of 300 iterations and fixed random state for reproducibility.
- **model.fit(x_train, y_train)=** Trains the logistic regression model on the training features and labels.
- **model.coef_=**Retrieves the coefficients (weights) for each feature and class learned by the model.
- **model.intercept_=**Retrieves the intercept values (bias terms) for each class in the model.
- **model.predict_proba(x_train)=** Predicts the class probabilities for each row in the training set.
- **model.predict_proba(x_test)=** Predicts the class probabilities for each row in the testing set.
- **y_pred_encoded = model.predict(x_test)=** Predicts the encoded class labels (as integers) for the test feature set.
- **predicted_pddistrict = label_encoders['PdDistrict'].inverse_transform(y_pred_encoded)=** Converts predicted class labels back to original PdDistrict names using the label encoder.
- **print(predicted_pddistrict[:20])=** Prints the first 20 predicted PdDistrict values.
- **actual_pddistrict = label_encoders['PdDistrict'].inverse_transform(y_test)=** Converts actual encoded test labels back to original PdDistrict names.
- **print(actual_pddistrict[:20])=** Prints the first 20 actual PdDistrict values.

- **model.score(x_test, y_test)=** Computes the accuracy of the model on the test data.

Second: Classification's Evaluation metrics

Confusion Matrix:

A confusion matrix is used to evaluate the performance of a classification model. In short, it shows how many times the model's predictions were correct and wrong, broken down by each class.

For each class:

1) **True Positive (TP):** Model correctly predicted that class.

2) **False Positive (FP):** Model predicted this class, but it was actually a different class.

3) **False Negative (FN):** Model missed this class — it predicted something else instead.

4) **True Negative (TN):** Model correctly predicted something else (not this class).

```python
#import the metrics
from sklearn.metrics import confusion_matrix,accuracy_score, precision_score, recall_score, f1_score, classification_report
```

Confusion Matrix:

A confusion matrix is used to evaluate the performance of a classification model. In short, it shows how many times the model's predictions were correct and wrong, broken down by each class.

For each class:

True Positive (TP): Model correctly predicted that class.

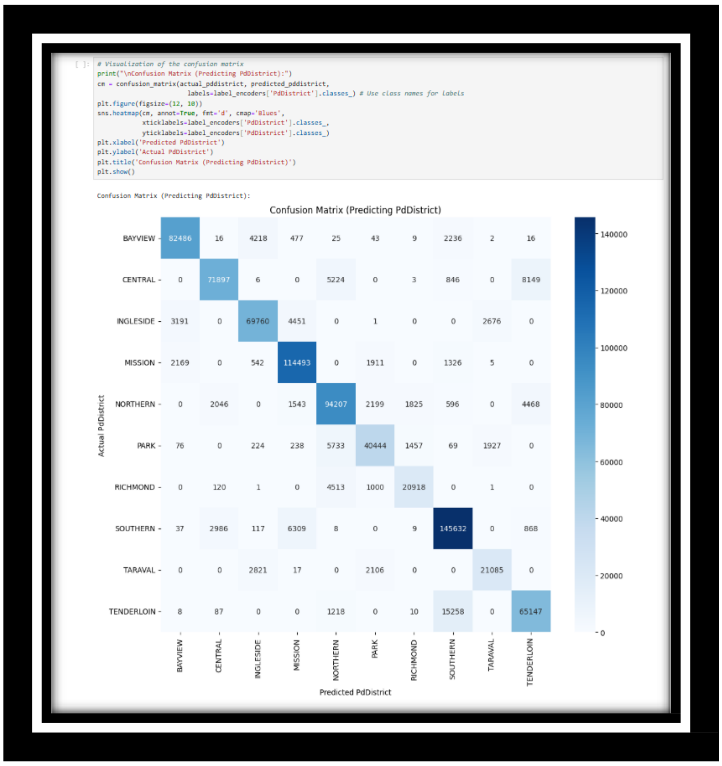
False Positive (FP): Model predicted this class, but it was actually a different class.

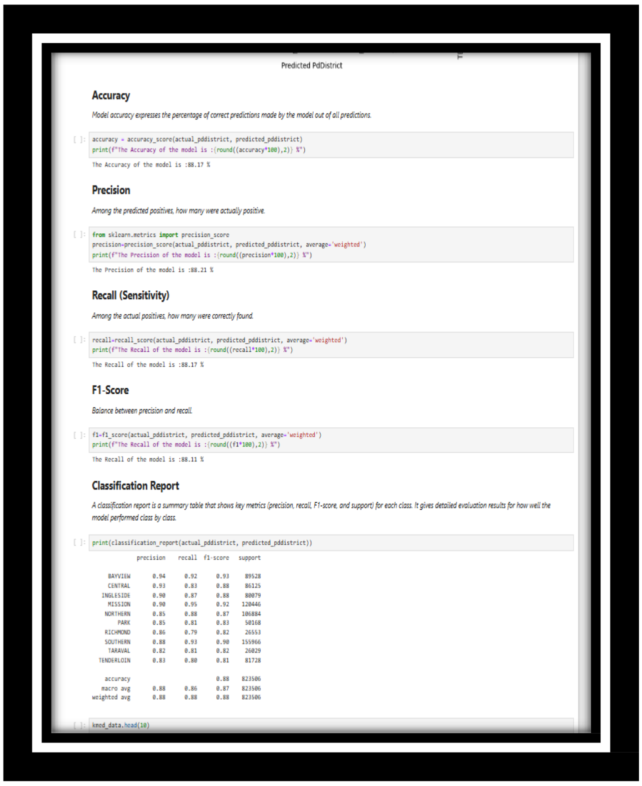
False Negative (FN): Model missed this class — it predicted something else instead.

True Negative (TN): Model correctly predicted something else (not this class).

```python
confusion_matrix(actual_pddistrict, predicted_pddistrict)
```

```
[153]: array([[ 82486,     16,   4218,    477,     25,     43,      9,   2236,
              2,     16],
       [     0,  71897,      6,      0,   5224,      0,      3,    846,
              0,   8149],
       [  3191,      0,  69760,   4451,      0,      1,      0,      0,
           2676,      0],
       [  2169,      0,    542, 114493,      0,   1911,      0,   1326,
              5,      0],
       [     0,   2046,      0,   1543,  94207,   2199,   1825,    596,
              0,   4468],
       [    76,      0,    224,    238,   5733,  40444,   1457,     69,
           1927,      0],
       [     0,    120,      1,      0,   4513,   1000,  20918,      0,
              1,      0],
       [    37,   2986,    117,   6309,      8,      0,      9, 145632,
              0,    868],
       [     0,      0,   2821,     17,      0,   2106,      0,      0,
          21085,      0],
       [     8,     87,      0,      0,   1218,      0,     10,  15258,
              0,  65147]])
```





Step 1 – Importing Libraries:

Imports necessary libraries for calculating and visualizing evaluation metrics

Step 2 – Confusion Matrix and Its Visualization:

Computes and visualizes the confusion matrix to show the model's prediction performance across classes.

Step 3 – Accuracy:

Calculates the accuracy: the ratio of correct predictions to total predictions.

Step 4 – Precision:

Computes the weighted average precision: how many predicted positives are actually positive

Step 5 – Sensitivity

Calculates recall: how many actual positives were correctly predicted (model sensitivity).

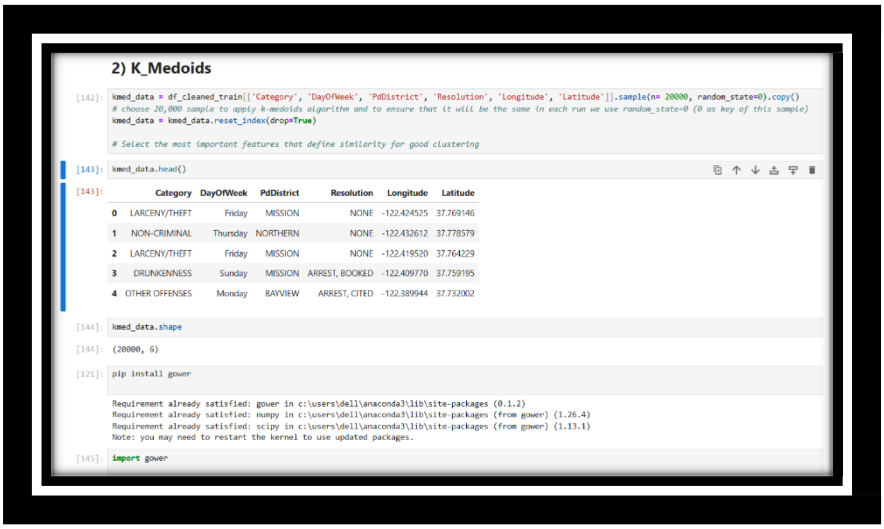
Step 6 – F1 Score:

Computes the F1 score: the harmonic mean of precision and recall.

Step 7 – Classification Report:

Displays a detailed report of precision, recall, F1-score, and support for each class.

Third: K-Medoids:

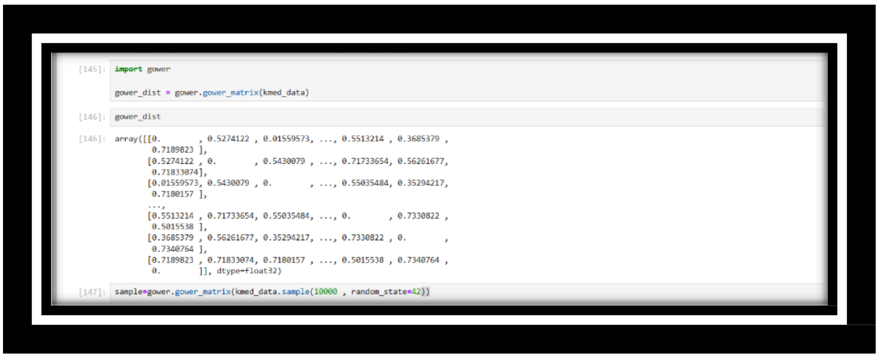


2) K_Medoids

```
[142]: kmed_data = df_cleaned_train[['Category', 'DayOfWeek', 'PdDistrict', 'Resolution', 'Longitude', 'Latitude']].sample(n= 20000, random_state=0).copy()
       # choose 20,000 sample to apply k-medoids algorithm and to ensure that it will be the same in each run we use random_state=0 (0 as key of this sample)
       kmed_data = kmed_data.reset_index(drop=True)

       # Select the most important features that define similarity for good clustering

[143]: kmed_data.head()
```

	Category	DayOfWeek	PdDistrict	Resolution	Longitude	Latitude
0	LARCENY/THEFT	Friday	MISSION	NONE	-122.424525	37.769146
1	NON-CRIMINAL	Thursday	NORTHERN	NONE	-122.432612	37.778579
2	LARCENY/THEFT	Friday	MISSION	NONE	-122.419520	37.764229
3	DRUNKENNESS	Sunday	MISSION	ARREST, BOOKED	-122.409770	37.759195
4	OTHER OFFENSES	Monday	BAYVIEW	ARREST, CITED	-122.389944	37.732002

```
[144]: kmed_data.shape

[144]: (20000, 6)

[121]: pip install gower

Requirement already satisfied: gower in c:\users\dell\anaconda3\lib\site-packages (0.1.2)
Requirement already satisfied: numpy in c:\users\dell\anaconda3\lib\site-packages (from gower) (1.26.4)
Requirement already satisfied: scipy in c:\users\dell\anaconda3\lib\site-packages (from gower) (1.13.1)
Note: you may need to restart the kernel to use updated packages.

[145]: import gower
```



```
[145]: import gower

       gower_dist = gower.gower_matrix(kmed_data)

[146]: gower_dist

[146]: array([[0.        , 0.5274122 , 0.01559573, ..., 0.5513214 , 0.3685379 ,
         0.7189823 ],
        [0.5274122 , 0.        , 0.5430079 , ..., 0.71733654, 0.56261677,
         0.71833074],
        [0.01559573, 0.5430079 , 0.        , ..., 0.55035484, 0.35294217,
         0.7180157 ],
        ...,
        [0.5513214 , 0.71733654, 0.55035484, ..., 0.        , 0.7330822 ,
         0.5015538 ],
        [0.3685379 , 0.56261677, 0.35294217, ..., 0.7330822 , 0.        ,
         0.7340764 ],
        [0.7189823 , 0.71833074, 0.7180157 , ..., 0.5015538 , 0.7340764 ,
         0.        ]], dtype=float32)

[147]: sample=gower.gower_matrix(kmed_data.sample(10000 , random_state=42))
```

```
[148]: sample

[148]: array([[0.        , 0.7937178 , 0.6178768 , ..., 0.7791389 , 0.7943672 ,
         0.8621341 ],
        [0.7937178 , 0.        , 0.50917435, ..., 0.733624  , 0.67999876,
         0.6257976 ],
        [0.6178768 , 0.50917435, 0.        , ..., 0.5677937 , 0.67916226,
         0.7916278 ],
        ...,
        [0.7791389 , 0.733624  , 0.5677937 , ..., 0.        , 0.74695605,
         0.8594216 ],
        [0.7943672 , 0.67999876, 0.67916226, ..., 0.74695605, 0.        ,
         0.77913225],
        [0.8621341 , 0.6257976 , 0.7916278 , ..., 0.8594216 , 0.77913225,
         0.        ]], dtype=float32)

[149]: from sklearn.metrics import silhouette_score

       k_values = range(2,15)
       sil_scores = []

       for k in k_values:
           kmedoid = KMedoids(n_clusters=k, metric='precomputed', random_state=42).fit(sample)
           # we We use metric='precomputed' to avoid recalculating distances
           score = silhouette_score(sample, kmedoid.labels_, metric='precomputed')
           # 'silhouette_score' measures how well each data point fits within its assigned cluster compared to other clusters
           sil_scores.append(score)

       plt.plot(k_values , sil_scores , marker='o')
       plt.xlabel('k')
       plt.ylabel('Silhouette Score')
       plt.title('Optimal k using Silhouette Score')
       plt.show()
```

Step 1 – Importing Libraries

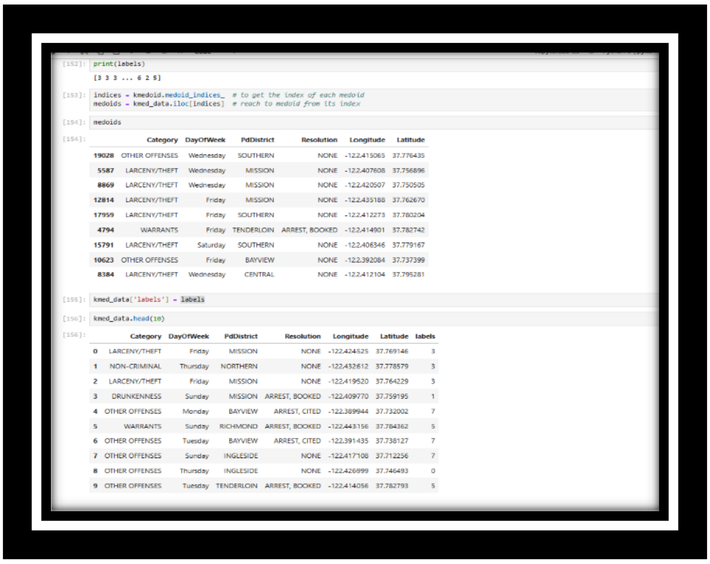
Imports the KMedoids model and preprocessing/visualization tools needed for clustering.

Step 2 – Feature Scaling:

Standardizes features to have zero mean and unit variance, improving K-Medoids performance.

Step 3: Prepare Data for Clustering

Select only the features (both categorical + numerical) needed for clustering.



Step 4: Compute Gower Distance

What is Gower Distance?

Gower distance is a similarity metric designed to work with **mixed-type data** — meaning it can handle both **numerical** and **categorical** variables. It calculates a score between 0 (exact match) and 1 (completely different) for each feature, then averages those scores to compute the distance between two records.

Why It's Important?

Most distance metrics (like Euclidean) only work on purely numerical data. But in real-world datasets — such as crime data or medical records — you often have a combination of:

- **Numerical data** (e.g., Latitude, Longitude)
- **Categorical data** (e.g., DayOfWeek, PdDistrict, Category)

Gower distance ensures that all these types can be **compared fairly and accurately** in clustering algorithms like **K-Medoids**, where the algorithm depends on meaningful distance calculations.

Step 5– Fitting the K-Medoids Model:

Creates and fits a K-Medoids model to the scaled feature data with a specified number of clusters

Step 6 – Predicting Cluster Labels:

Retrieves the cluster assignment (label) for each data point after fitting the model.

Step 7 – Visualizing the Clusters:

Plots the clustered data points and medoid centers to visualize how K-Medoids grouped the data

---

Fourth: K-Medoids' Evaluation Metrics

K-Medoids' Evaluation Metrics

Silhouette Score

The Silhouette Score is used to evaluate the quality of clustering. It measures how similar each point is to its own cluster compared to other clusters.

Range: -1 to 1

Close to 1: The point is well matched to its own cluster and poorly matched to neighboring clusters (good clustering).

Around 0: The point lies between clusters (overlap).

```
[ ]: from sklearn.metrics import silhouette_score

sil = silhouette_score(gower_dist, labels, metric='precomputed')
print(f"Silhouette Score: {sil:.4f}")
```

```
Silhouette Score: 0.1646
```

Davies-Bouldin Index

The Davies-Bouldin Index measures the average similarity between each cluster and its most similar one.

It considers:

Compactness: How tight each cluster is.

Separation: How far apart the clusters are.

-Lower values indicate better clustering (more compact and well-separated clusters).

-Higher values indicate worse clustering.

```
[ ]: from sklearn.metrics import davies_bouldin_score

db_score = davies_bouldin_score(gower_dist, labels)
print(f"Davies-Bouldin Score: {db_score:.4f}")
```

```
Davies-Bouldin Score: 1.0562
```

1) Silhouette Score

The Silhouette Score is used to evaluate the quality of clustering. It measures how similar each point is to its own cluster compared to other clusters.

Range: -1 to 1

Close to 1: The point is well matched to its own cluster and poorly matched to neighboring clusters (good clustering).

Around 0: The point lies between clusters (overlap).

its code calculates the **Silhouette Score** using the Gower distance matrix and the cluster labels to measure how well each point fits its assigned cluster.
A higher silhouette score (closer to 1) indicates better-defined and well-separated clusters

---

2) Davies-Bouldin Index

The Davies-Bouldin Index measures the average similarity between each cluster and its most similar one.

It considers:

Compactness: How tight each cluster is.

Separation: How far apart the clusters are.

-Lower values indicate better clustering (more compact and well-separated clusters).

-Higher values indicate worse clustering.

its code calculates the **Davies-Bouldin Score**, which evaluates clustering quality based on the similarity between clusters.
A **lower score** indicates better clustering, with more distinct and well-separated clusters.
