

Predicting Crime Trends using Machine Learning Models

Table of Contents

Table of Figures.....	i
Abstract:.....	iv
Introduction	5
1.0 Problem Statement.....	6
1.1 Data Understanding:.....	6
1.2 Research Questions:	7
2.0 Data Preparation:.....	8
2.1 Data Loading and Merging:.....	8
2.2 Dataset Integrity Check:.....	11
2.3 Data Cleansing with Pandas:.....	15
2.5 Handling Invalid Entries:	17
3.0 Queries.....	23
3.1 SQL Queries.....	23
3.2 Hive Queries.....	27
4.0 MapReduce	29
5.0 Machine Learning.....	32
5.1 Data Pre-processing.....	32
5.1.1 Initial Data Analysis.....	32
5.1.2 Feature Selection	32
5.1.3 Dropping Irrelevant Columns.....	33
5.1.4 Data Transformation.....	33
5.1.5 Splitting Train and Test	34
5.2 Prediction Algorithms	35
5.2.1 Linear Regression:	36
5.2.2 Linear SVR (Support Vector Regression):.....	37

5.2.3 RBF SVR (Radial Basis Function Support Vector Regression):.....	37
5.2.4 Polynomial SVR:	37
5.2.5 Decision Tree:.....	38
5.2.6 Random Forest:.....	38
5.2.7 Evaluation of Models by using MSE:	39
5.2.8 Result:	40
5.3 Data Visualization	42
6.0 Conclusion:.....	44

Table of Figures

Figure 1:Create DataBase.....	9
Figure 2:merge tables	9
Figure 3: Merge and detect not-force areas.....	10
Figure 4: Removed Non-Police Force Areas.....	10
Figure 5: Detect missing values.....	11
Figure 6: Identify duplicate records.....	12
Figure 7: check null values.	12

Figure 8: check space values	13
Figure 9: Delete space values.	14
Figure 10: recheck duplicate values.....	15
Figure 11: last count.....	15
Figure 12: Install pandas scikit-learn.....	16
Figure 13: Import library, load database	16
Figure 14: Detect Missing Values.....	16
Figure 15: Detect duplicates.	17
Figure 16: check data types	17
Figure 17: check trailing spaces.	18
Figure 18: Check non-numeric and NaN values.....	18
Figure 19: Try to strip the object column.	19
Figure 20: Convert object type to string type and then strip.	19
Figure 21: Convert non-numeric values.....	20
Figure 22: check comma for numbers.	20
Figure 23: Functions on the numeric column.	20
Figure 24: Recheck data types.	21
Figure 25: check Nan values after changing type.	21
Figure 26: check non-numeric values.	21
Figure 27: check negative values.	21
Figure 28: count each column.....	22
Figure 29: Save changes.....	22
Figure 30: Five common offence descriptions.....	23
Figure 31: shows the number of offence descriptions according to the years.	24
Figure 32: shows the relation between crime and force name.....	25
Figure 33: found relation between offences in each quarter.....	26
Figure 34: lowest rate of criminal activity.	27
Figure 35: Force name with highest criminal rates.....	27
Figure 36: Hive Queries_q1.....	28
Figure 37: result Hive queries_q1.	28
Figure 38: Hive Queries_q2.....	29
Figure 39: result Hive queries_q2.	29
Figure 40: install tools and libraries before running MapReduce.....	30
Figure 41: Response of MapReduce by PySpark.....	31
Figure 42: Show result in powerBI.....	31
Figure 43: summary statistics	32
Figure 44: summary statistics for a numeric column.....	32
Figure 45: drop irrelevant columns.....	33
Figure 46: Label Encoding.	33
Figure 47: year Column Modification.	33
Figure 48: save the transformed file.....	34
Figure 49: split dataset to train and test.....	35
Figure 50: count of rows for train and test.....	35
Figure 51: define X and Y vectors.....	36
Figure 52: Linear regression train algorithm.....	36
Figure 53: Linear regression prediction.	37
Figure 54: Linear SVR algorithm.....	37
Figure 55: RBF SVR algorithm.....	37

Figure 56: Polynomial SVR train algorithm.	37
Figure 57: prediction of different types of SVR.	37
Figure 58: Decision Tree algorithm.	38
Figure 59: prediction Decision Tree.	38
Figure 60: Random Forest algorithms.	38
Figure 61: Prediction of different types of Random Forests.	39
Figure 62: MSE formula.	39
Figure 63: calculate MSE.	40
Figure 64: Show MSE results.	40
Figure 65: Show diagram code.	41
Figure 66: Comparison Diagram.	41
Figure 67: Dashboard of powerBI.	42

Abstract:

This study focuses on machine learning's capacity to obtain insights without the need for human interaction in criminal analysis. It uses artificial intelligence and big data to achieve this. We navigate the complexities of crime data using the CRISP-DM paradigm, from early understanding to real-world implementation. Large datasets may be sophisticatedly analysed with this approach, which is essential for creating effective, data-driven strategies for the detection and prevention of criminal patterns. The report is divided into two sections. The first part presents several models for predicting the number of offences, focusing on their relevant attributes, and compares the accuracy of these models. The second segment examines how big data and machine learning technologies are impacting criminal analysis. It illustrates the value of ongoing innovation in this dynamic subject by demonstrating how various technologies impact crime pattern analysis and contribute to its evolution.

Keywords: Big Data, Machine Learning, Data Analytics, Spark, Power BI

Introduction

In this introduction, we will explore the intricacies of crime analysis utilising the CRISP-DM framework, with a focus on comprehensive data preparation and analysis. The method begins with tackling the subtle issues in crime data interpretation, using resources such as the "Police recorded crime open data tables." During this stage, extensive data collection, cleaning, and exploratory analysis are done using Python, Pandas, and PySpark. The report is divided into two sections. The first half delves into predictive modelling techniques for crime investigation, while the second section looks at the contributions and implications of big data and machine learning in this emerging field. With the help of our approach, crime trends can be better understood, which is crucial for enhancing law enforcement procedures and developing efficient crime prevention strategies.

1.0 Problem Statement

This study aims to analyse crime data to understand and predict crime trends. Our objective is to dissect and understand crime data, exploring its multifaceted nature to uncover underlying patterns and trends. This analysis is pivotal in understanding the complexities and variations in crime occurrences, their distribution across different regions, and their evolution over time. Leveraging tools like SQL, MapReduce, and machine learning, this study aims to provide a comprehensive view of crime dynamics, potentially revealing insights into effective crime prevention and policy-making. Leveraging tools like SQL, MapReduce, and machine learning, this study aims to provide a comprehensive view of crime dynamics, potentially revealing insights into effective crime prevention and policy-making. By investigating these areas, the study seeks to gain a deeper understanding of crime occurrences and distributions, aiding in the development of informed strategies for crime prevention and law enforcement.

1.1 Data Understanding:

This scholarly investigation was prepared by a thorough analysis of the 'Police recorded crime open data tables' dataset, which covers the period from March 2013 to June 2023. This study began with a thorough analysis of the user manual and pertinent web resources to obtain a better knowledge of the historical background and structure of the dataset. Important columns like Financial Year, Quarter, Force Name, Offence Description, Code, and Number of Offences are included in the collection, which is divided into yearly segments. Important details regarding the type, date, and classification of crimes are included in these columns. Understanding the relationships between these columns is crucial to our study because they highlight trends and patterns in criminal activity. The inquiry uncovered disparities in data reporting between law enforcement agencies and notable changes to crime recording guidelines after 2017. The basis for in-depth data analysis and machine learning applications is this thorough comprehension of the dataset.

This study's primary focus is on using machine learning techniques to anticipate and prevent crimes. This in-depth understanding is crucial for developing effective crime prevention policies because it has a direct impact on the choices that policymakers and law enforcement make.

1.2 Research Questions:

After gaining a solid understanding of the dataset, this study will use machine learning techniques, SQL queries, MapReduce, or Spark to tackle research problems:

We'll utilise SQL queries for:

1. Identify the five most common crime types.
2. How has the frequency of specific crimes changed annually?
3. Examine the connection between crime types and police force areas.
4. Examine how the crime rate varies with the quarters.
5. Find out which places have the highest and lowest rates of criminal activity.

Hive:

1. How do crime rates vary across different financial quarters?
2. Which police force regions have seen the biggest shifts in crime rates over the course of various fiscal quarters?

MapReduce:

1. How many crimes happen each year?

Machine Learning:

1. Predicting future criminal trends

These questions try to optimise the value of the data by assisting in the establishment of a comprehensive understanding of crime patterns for effective prevention and law enforcement efforts.

2.0 Data Preparation:

The crime dataset will be meticulously organised for analysis in this report part. Cleaning and arranging the data is a crucial step in making sure it is accurate and suitable for our study. To keep the dataset intact, outliers—such as negative data points—must be recognised and dealt with. To maintain consistency, we will deal with duplicate data and missing values in addition to standardising the data format. This preparation is required for the successful use of SQL queries, machine learning techniques, and other analytical tools. Our goal is to establish a robust foundation for answering research questions and generating valuable insights into crime patterns and trends.

2.1 Data Loading and Merging:

During this critical phase, we focused on getting the dataset ready to align with crime trends from 2019 to 2023 to improve its accuracy and relevance. This time period was used to quantify offences more accurately across various Police Force areas. To correctly identify these places, we combined our own data with the 'recrime-geo-pfa' dataset. This was significant because our data's 'Force Name' column combines data on Police Force Areas with information from agencies like Action Fraud, CIFAS, and Financial Fraud UK. Initial steps in data preparation:

Make use of a spreadsheet programme to convert the required sheets from .ods to .csv format.

Creation of a new database named 'OffencesDB' in SQLite Studio.

Import of each .csv file into SQLite Studio, and make sure that the right data types are selected for each column.

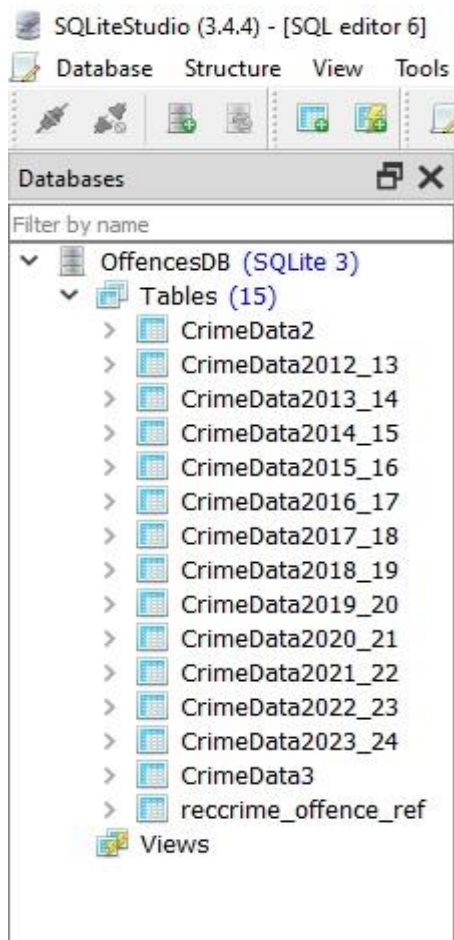


Figure 1:Create DataBase.

Merge Tables with SQL:

After importing, merge the tables into a single table. The SQL query for merging might look like this:

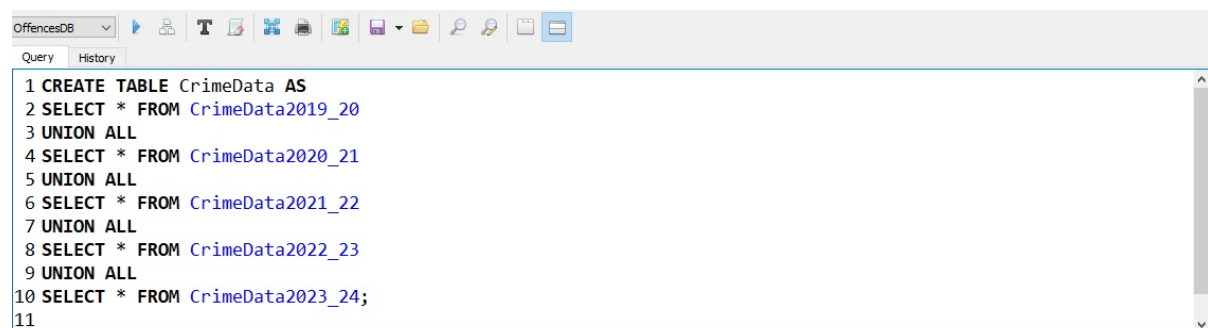


Figure 2:merge tables

Detect Police Force Areas:

It has included some steps because the column Force Name combines Police Force Area and some information for fraud offences from organisations. First, we merge the CrimeData table with the recrime-geo-pfa reference table, in this table we have the exact Police Force Area and define which data are not in the recrime-geo-pfa reference table:

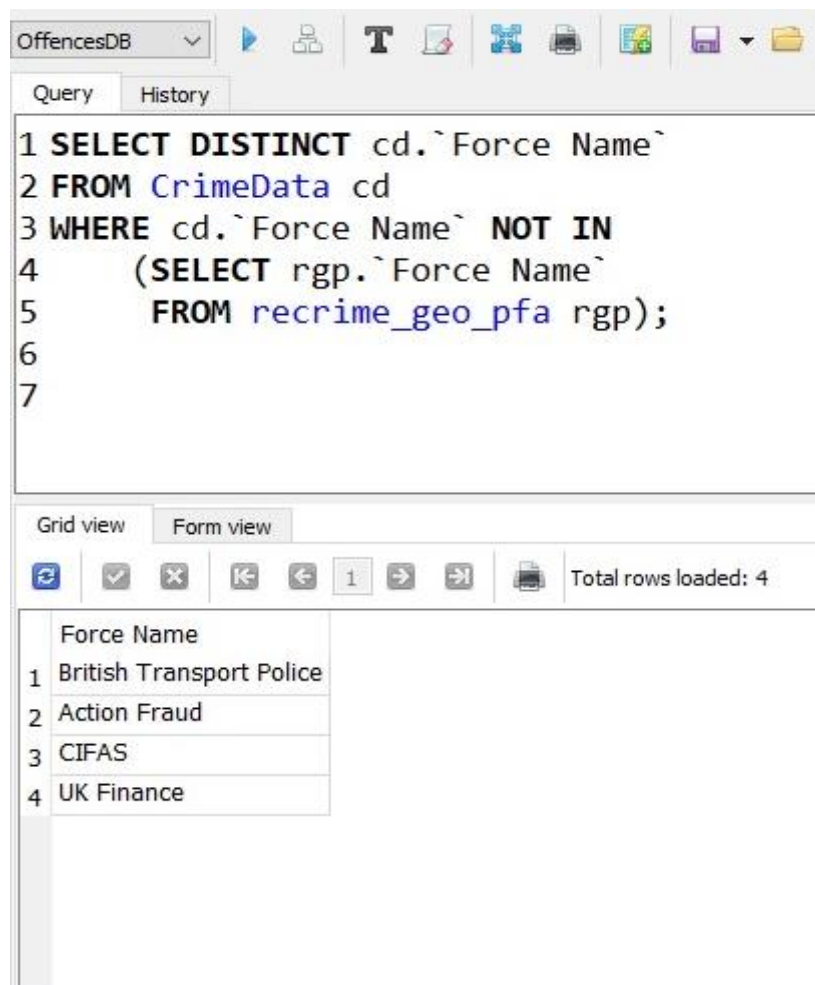


Figure 3: Merge and detect not-force areas.

Then decided to delete them from the CrimeData Table:

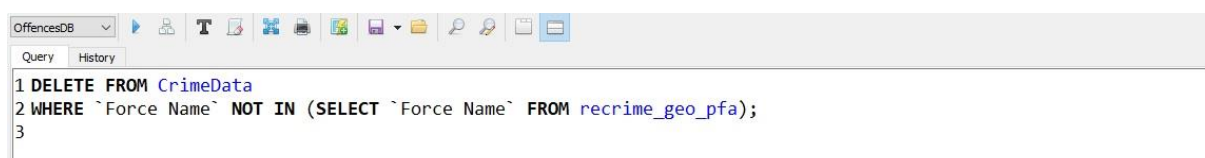


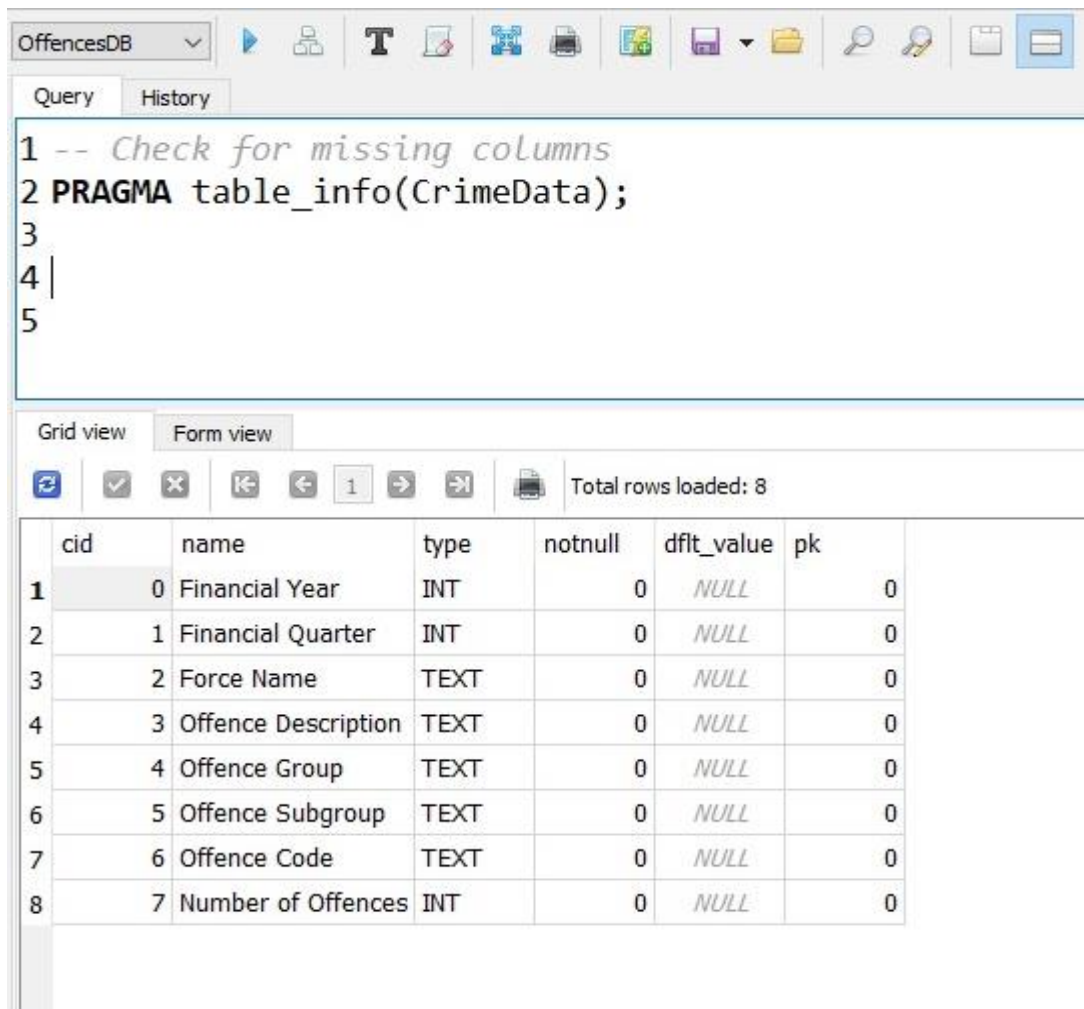
Figure 4: Removed Non-Police Force Areas.

This process efficiently consolidates our data, making it ready for the subsequent integrity check and data cleansing steps.

2.2 Dataset Integrity Check:

We verify the accuracy and completeness of our merged dataset in this phase. The two primary SQL queries are these two:

Checking for Missing Columns:



The screenshot shows a database query tool interface. The top toolbar includes icons for database operations. Below the toolbar, there are tabs for 'Query' and 'History'. The 'Query' tab is active, displaying a SQL query:

```
1 -- Check for missing columns
2 PRAGMA table_info(CrimeData);
3
4 |
5
```

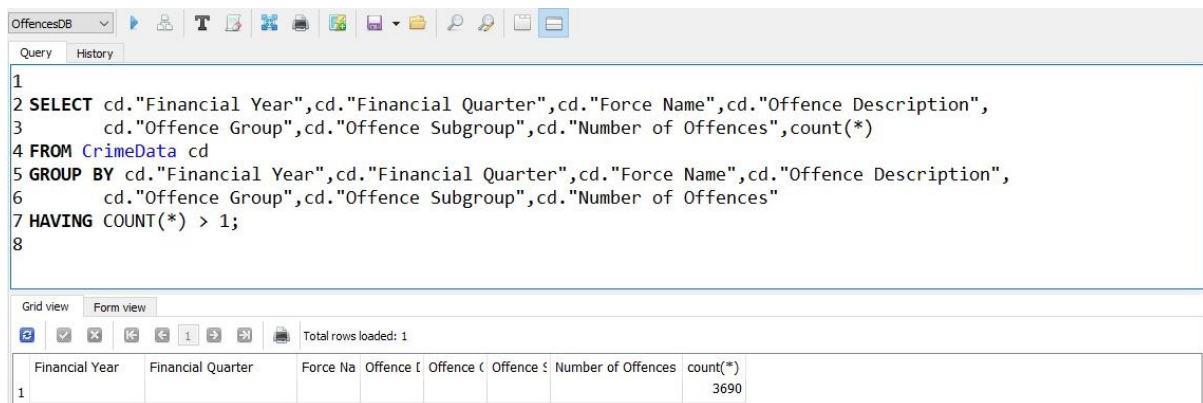
Below the query editor, there are tabs for 'Grid view' and 'Form view'. The 'Grid view' tab is active, showing a table of results. The table has 8 rows and 7 columns: 'cid', 'name', 'type', 'notnull', 'dflt_value', and 'pk'. The 'Total rows loaded: 8' is displayed on the right side of the grid view toolbar.

	cid	name	type	notnull	dflt_value	pk
1	0	Financial Year	INT	0	NULL	0
2	1	Financial Quarter	INT	0	NULL	0
3	2	Force Name	TEXT	0	NULL	0
4	3	Offence Description	TEXT	0	NULL	0
5	4	Offence Group	TEXT	0	NULL	0
6	5	Offence Subgroup	TEXT	0	NULL	0
7	6	Offence Code	TEXT	0	NULL	0
8	7	Number of Offences	INT	0	NULL	0

Figure 5: Detect missing values.

We use a PRAGMA statement to examine the structure of our combined dataset and confirm that all necessary columns are there.

Identifying Duplicate Records:



```
1
2 SELECT cd."Financial Year",cd."Financial Quarter",cd."Force Name",cd."Offence Description",
3        cd."Offence Group",cd."Offence Subgroup",cd."Number of Offences",count(*)
4 FROM CrimeData cd
5 GROUP BY cd."Financial Year",cd."Financial Quarter",cd."Force Name",cd."Offence Description",
6          cd."Offence Group",cd."Offence Subgroup",cd."Number of Offences"
7 HAVING COUNT(*) > 1;
8
```

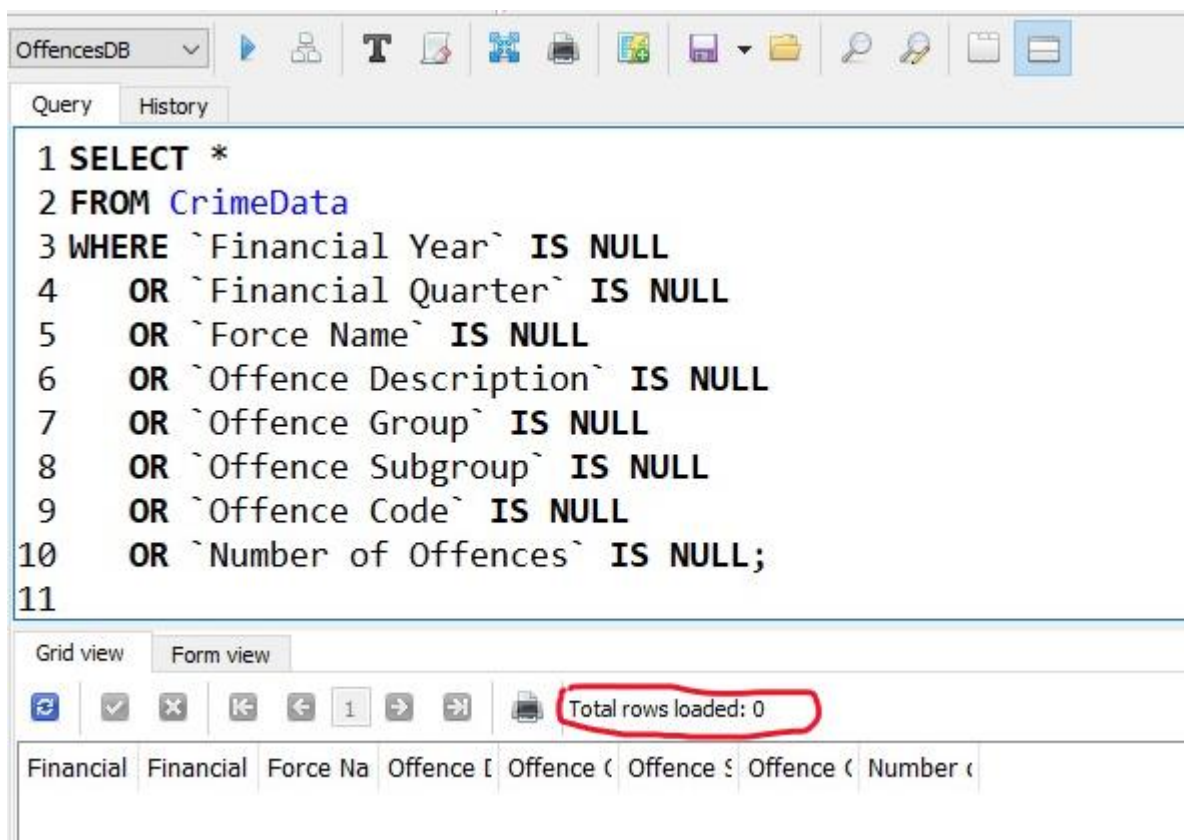
Grid view Form view

Total rows loaded: 1

	Financial Year	Financial Quarter	Force Na	Offence t	Offence C	Offence S	Offence C	Number of Offences	count(*)
1									3690

Figure 6: Identify duplicate records.

A SELECT query, grouped by all columns, is executed to count and identify any duplicate rows in our data. For the result, we see that detect 3690 records that all columns are null for the next step we check the null value in our database:



```
1 SELECT *
2 FROM CrimeData
3 WHERE `Financial Year` IS NULL
4      OR `Financial Quarter` IS NULL
5      OR `Force Name` IS NULL
6      OR `Offence Description` IS NULL
7      OR `Offence Group` IS NULL
8      OR `Offence Subgroup` IS NULL
9      OR `Offence Code` IS NULL
10     OR `Number of Offences` IS NULL;
11
```

Grid view Form view

Total rows loaded: 0

Financial	Financial	Force Na	Offence t	Offence C	Offence S	Offence C	Number C
-----------	-----------	----------	-----------	-----------	-----------	-----------	----------

Figure 7: check null values.

But we do not have any null values, so we decided to check space values :

The screenshot shows a database query tool interface. At the top, there's a toolbar with various icons. Below it, a tab labeled 'Query' is active. The query editor contains the following SQL code:

```
2 FROM CrimeData
3 WHERE TRIM(`Financial Year`) = ''
4 OR TRIM(`Financial Quarter`) = ''
5 OR TRIM(`Force Name`) = ''
6 OR TRIM(`Offence Description`) = ''
7 OR TRIM(`Offence Group`) = ''
8 OR TRIM(`Offence Subgroup`) = ''
9 OR TRIM(`Offence Code`) = ''
10 OR TRIM(`Number of Offences`) = '';
11
```

Below the query editor, there are tabs for 'Grid view' and 'Form view'. The 'Grid view' tab is selected. Below the tabs, there's a toolbar with icons for refresh, check, close, and navigation. A status bar at the bottom right of the toolbar indicates 'Total rows loaded: 3690', which is highlighted with a red box. Below the toolbar, a grid view of the query results is shown. The grid has 8 columns: 'Financial', 'Financial', 'Force Na', 'Offence I', 'Offence (', 'Offence §', 'Offence (', and 'Number ('. The first column is numbered 1 to 8, corresponding to the rows. The grid is currently empty, showing only the headers and row numbers.

	Financial	Financial	Force Na	Offence I	Offence (Offence §	Offence (Number (
1								
2								
3								
4								
5								
6								
7								
8								

Figure 8: check space values

So, after that, we decide to delete these values by trimming the command:

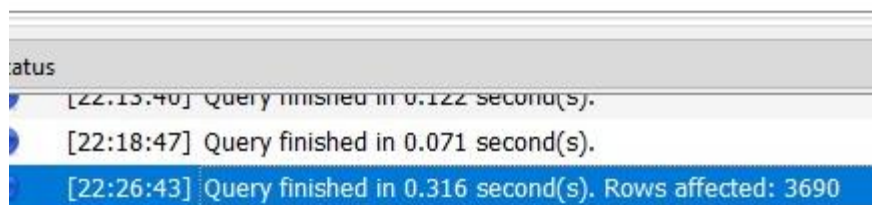
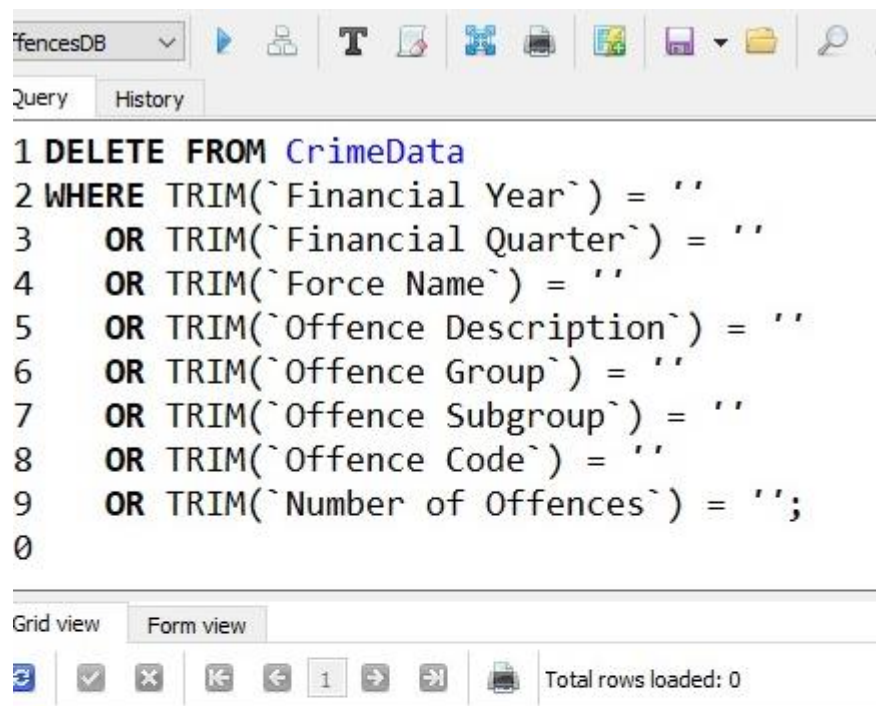


Figure 9: Delete space values.

After that, we recheck the duplicate values:

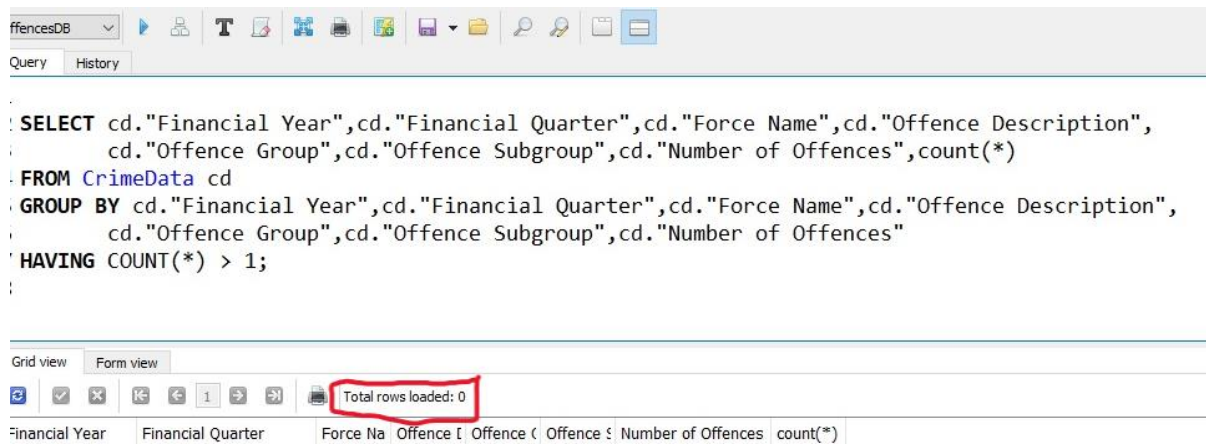


Figure 10: recheck duplicate values.

And we see the result was 0. Finally, our table has 98777 rows and 8 columns:

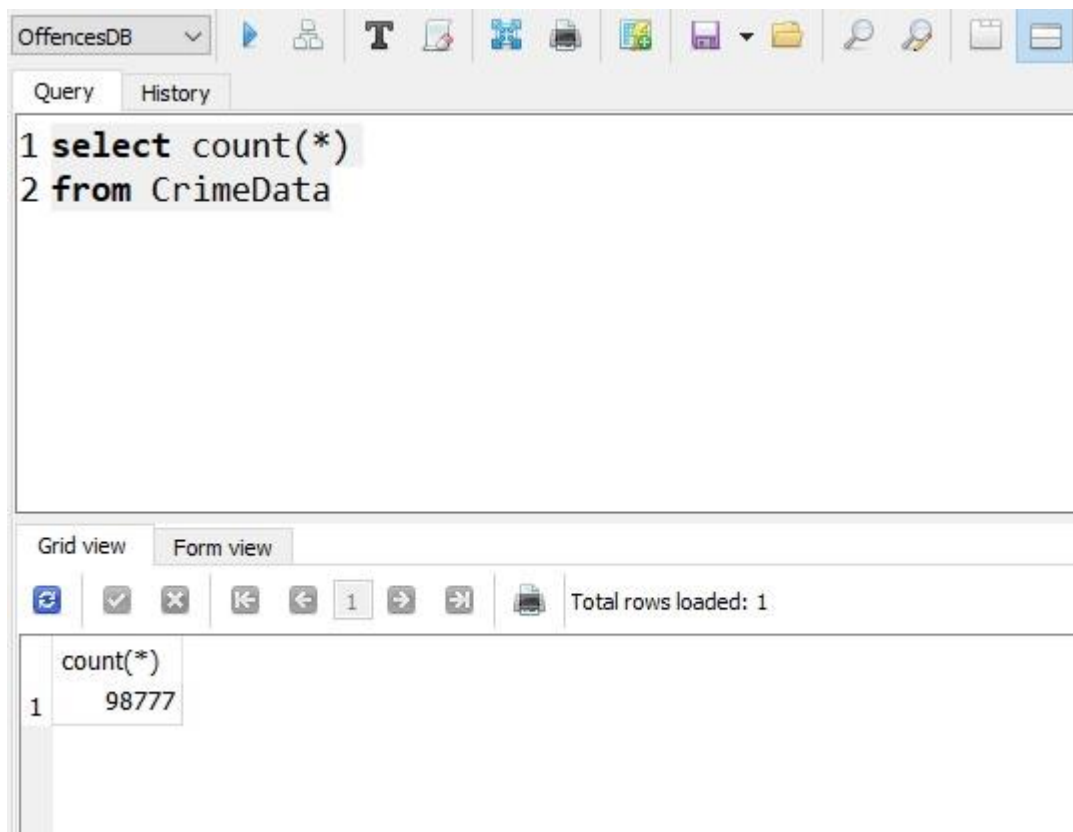
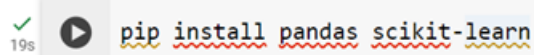


Figure 11: last count

2.3 Data Cleansing with Pandas:

Python's Pandas library is used to improve the quality of the dataset, which is required for effective analysis. After that, any leftover data inconsistencies are thoroughly examined and fixed. This meticulous process includes:

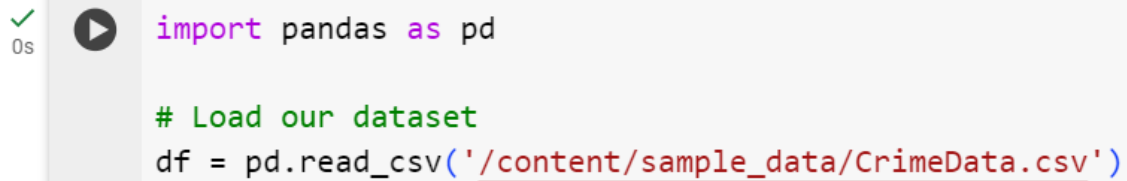
First, we install every necessary library:



```
✓ 19s ▶ pip install pandas scikit-learn
```

Figure 12: Install pandas scikit-learn.

Then, load our database:



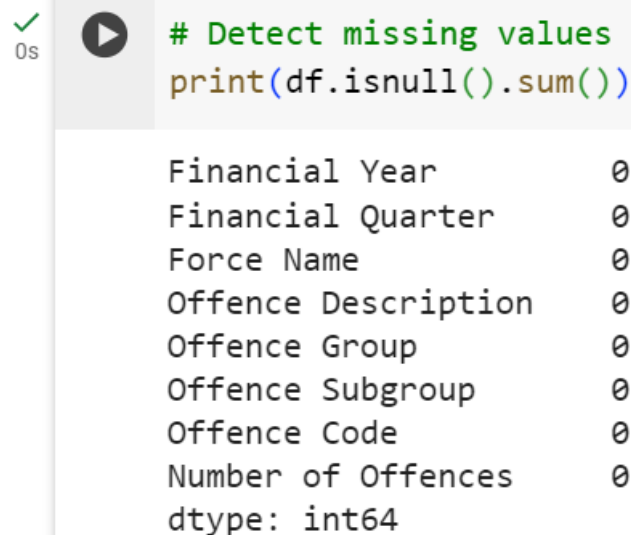
```
✓ 0s ▶ import pandas as pd

# Load our dataset
df = pd.read_csv('/content/sample_data/CrimeData.csv')
```

Figure 13: Import library, load database

Null Value Management:

At the beginning of this part, first, we must determine the missing values:



```
✓ 0s ▶ # Detect missing values
print(df.isnull().sum())
```

Financial Year	0
Financial Quarter	0
Force Name	0
Offence Description	0
Offence Group	0
Offence Subgroup	0
Offence Code	0
Number of Offences	0
dtype:	int64

Figure 14: Detect Missing Values

So, because we do not have any missing values passed this step.

Duplicate Record Resolution:

```
df = pd.read_csv('/content/sample_data/CrimeData.csv')

# Find duplicate rows based on all columns
duplicate_rows = df[df.duplicated()]
count_duplicates_all = duplicate_rows.shape[0]

# If you want to check duplicates based on specific columns
duplicate_rows_specific = df[df.duplicated(['Financial Year', 'Financial Quarter', 'Force Name', 'Offence Description'])]
count_duplicates_specific = duplicate_rows_specific.shape[0]

print(f"Total Duplicate Rows (All Columns): {count_duplicates_all}")
print(duplicate_rows)
print(f"\nTotal Duplicate Rows (Specific Columns): {count_duplicates_specific}")
print(duplicate_rows_specific)
```

Total Duplicate Rows (All Columns): 0
Empty DataFrame
Columns: [Financial Year, Financial Quarter, Force Name, Offence Description, Offence Group, Offence Subgroup, Offence Code, Number of Offences]
Index: []

Total Duplicate Rows (Specific Columns): 0
Empty DataFrame
Columns: [Financial Year, Financial Quarter, Force Name, Offence Description, Offence Group, Offence Subgroup, Offence Code, Number of Offences]
Index: []

<ipython-input-11-577a822e5c85>:4: DtypeWarning: Columns (7) have mixed types. Specify dtype option on import or set low_memory=False.
df = pd.read_csv('/content/sample_data/CrimeData.csv')

Figure 15: Detect duplicates.

Our investigation did not reveal any duplicate data during our checks. We focused on these four columns since they are crucial for forecasting crime. These columns have to be distinct since they let us distinguish between different acts and spot criminal tendencies.

2.5 Handling Invalid Entries:

To check invalid data, first, we check the data type of columns:

```
print(df.dtypes)
```

Financial Year	object
Financial Quarter	int64
Force Name	object
Offence Description	object
Offence Group	object
Offence Subgroup	object
Offence Code	object
Number of Offences	object
dtype:	object

Figure 16: check data types

So, the above figure shows the type of column 'Number of Offences' is an object, but we need numeric, so we check another parameter for this column. We check the space at the beginning and end of each value:

```
import pandas as pd

df = pd.read_csv('/content/sample_data/CrimeData.csv')

# Function to count values with leading or trailing spaces in a column
def count_leading_trailing_spaces(column):
    if df[column].dtype == 'object':
        # Convert to string and count values with leading/trailing spaces
        return df[column].astype(str).str.contains(r'^\s+|\s+$', regex=True).sum()
    else:
        # If not an object (string) type, return 0
        return 0

# Apply the function to each column and print the counts
for col in df.columns:
    count = count_leading_trailing_spaces(col)
    print(f"Column '{col}' has {count} values with leading or trailing spaces.")
```

<ipython-input-32-95e095556705>:3: DtypeWarning: Columns (7) have mixed types. Specify dtype option on import or set low_memory=False.
Column 'Financial Year' has 0 values with leading or trailing spaces.
Column 'Financial Quarter' has 0 values with leading or trailing spaces.
Column 'Force Name' has 0 values with leading or trailing spaces.
Column 'Offence Description' has 0 values with leading or trailing spaces.
Column 'Offence Group' has 0 values with leading or trailing spaces.
Column 'Offence Subgroup' has 0 values with leading or trailing spaces.
Column 'Offence Code' has 0 values with leading or trailing spaces.
Column 'Number of Offences' has 1313 values with leading or trailing spaces.

Figure 17: check trailing spaces.

It shows that only 'Number of Offences', has space that the nationality of this column must be numbered. But before dropping these spaces for this column we check another thing. In the below, we see that some of the other values for this column are detected as a text value.

```
#Count Non-numeric Values
non_numeric_before = df['Number of Offences'].apply(lambda x: isinstance(x, str))
print("Number of non-numeric values before conversion:", non_numeric_before.sum())
print("Unique non-numeric values:", df['Number of Offences'][non_numeric_before].unique())
```

Number of non-numeric values before conversion: 65536
Unique non-numeric values: ['9' '5' '0' ... '2084' '1768' '1996']

```
#Count NaN Values
nan_count_before = df['Number of Offences'].isna().sum()
print("Number of NaN values before conversion:", nan_count_before)
```

Number of NaN values before conversion: 0

Figure 18: Check non-numeric and NaN values.

After that, we strip the space for specific columns, and again evaluate:

```
import pandas as pd

df = pd.read_csv('/content/sample_data/CrimeData.csv')

# Strip leading and trailing spaces from the specified column
df['Number of Offences'] = df['Number of Offences'].str.strip()

# Optional: Save the cleaned data
# df.to_csv('cleaned_data.csv', index=False)
```

<ipython-input-34-038b4e3aae61>:3: DtypeWarning: Columns (7) have mixed types. Specify dtype option on import or set low_memory=False.
df = pd.read_csv('/content/sample_data/CrimeData.csv')

```
[35] #Count Non-numeric Values
non_numeric_before = df['Number of Offences'].apply(lambda x: isinstance(x, str))
print("Number of non-numeric values before conversion:", non_numeric_before.sum())
print("Unique non-numeric values:", df['Number of Offences'][non_numeric_before].unique())
```

Number of non-numeric values before conversion: 65536
Unique non-numeric values: ['9' '5' '0' ... '2084' '1768' '1996']

```
#Count NaN Values
nan_count_before = df['Number of Offences'].isna().sum()
print("Number of NaN values before conversion:", nan_count_before)
```

Number of NaN values before conversion: 33241

Figure 19: Try to strip the object column.

Now we have 33241 NaN values that are not acceptable, so again try but this time first convert the object to string type for it then do strip and then check NaN values again, if it has 0 NaN that means this process is true:

```
[43] df = pd.read_csv('/content/sample_data/CrimeData.csv')

# Strip leading and trailing spaces from the specified column
df['Number of Offences'] = df['Number of Offences'].astype(str)

<ipython-input-43-dbf284ed03e3>:3: DtypeWarning: Columns (7) have mixed types. Specify dtype option on import or set low_memory=False.  
df = pd.read_csv('/content/sample_data/CrimeData.csv')
```

```
df['Number of Offences'] = df['Number of Offences'].str.strip()
# Optional: Save the cleaned data
# df.to_csv('cleaned_data.csv', index=False)
```

```
[46] #Count Non-numeric Values
non_numeric_before = df['Number of Offences'].apply(lambda x: isinstance(x, str))
print("Number of non-numeric values before conversion:", non_numeric_before.sum())
print("Unique non-numeric values:", df['Number of Offences'][non_numeric_before].unique())
```

Number of non-numeric values before conversion: 98777
Unique non-numeric values: ['9' '5' '0' ... '8414' '5283' '1498']

```
#Count NaN Values
nan_count_before = df['Number of Offences'].isna().sum()
print("Number of NaN values before conversion:", nan_count_before)
```

Number of NaN values before conversion: 0

Figure 20: Convert object type to string type and then strip.

We check non-numeric values and NaN values for this column. As we can see 65536 data were non-numeric values then we tried to convert them to numeric values.

```

✓ [24] #Convert to Numeric and Count NaNs
0s df['Number of Offences'] = pd.to_numeric(df['Number of Offences'], errors='coerce')
nan_count_after = df['Number of Offences'].isna().sum()
print("Number of NaN values after conversion:", nan_count_after)

```

Number of NaN values after conversion: 1313

Figure 21: Convert non-numeric values.

But after converting, again we have 1313 NaN values, the other thing that we can check for the type of number is ',' for upper than a thousand.

```

✓ [60] has_commas = df['Number of Offences'].astype(str).str.contains(',')
0s count_rows_with_commas = has_commas.sum()
print("Number of rows with commas:", count_rows_with_commas)

```

Number of rows with commas: 1313

Figure 22: check comma for numbers.

It has exactly 1313 numbers that have ',' in their numbers. So again, first convert object to str then drop comma then strip space them and finally convert to integer type.

```

✓ [62] #Finally solution:
0s df = pd.read_csv('/content/sample_data/CrimeData.csv')
#1.Convert Object to String
df['Number of Offences'] = df['Number of Offences'].astype(str)

<ipython-input-62-74f27b78aa57>:2: DtypeWarning: Columns (7) have mixed types. Specify dtype option on import or set low_memory=False.
df = pd.read_csv('/content/sample_data/CrimeData.csv')

✓ [63] #Remove Commas
0s df['Number of Offences'] = df['Number of Offences'].str.replace(',', '')

✓ [64] #Strip Spaces
0s df['Number of Offences'] = df['Number of Offences'].str.strip()

✓ [65] #Convert to Numeric type
0s df['Number of Offences'] = pd.to_numeric(df['Number of Offences'], errors='coerce')

```

Figure 23: Functions on the numeric column.

After that, we check it again for the type and Nan values:

```
✓ 0s ▶ print(df.dtypes)
```

Financial Year	object
Financial Quarter	int64
Force Name	object
Offence Description	object
Offence Group	object
Offence Subgroup	object
Offence Code	object
Number of Offences	int64
dtype:	object

Figure 24: Recheck data types.

The above picture shows the type of columns is correct. The type of 'Financial Year' is the object because the values are a mix of numbers and a character like '2019/20'. In continuation, we have some other checks.

```
✓ 0s ▶ nan_count_after = df['Number of Offences'].isna().sum()
print("Number of NaN values before conversion:", nan_count_after)
```

Number of NaN values before conversion: 0

Figure 25: check Nan values after changing type.

```
✓ 0s [17] #Count Non-numeric Values
non_numeric_before = df['Number of Offences'].apply(lambda x: isinstance(x, str))
print("Number of non-numeric values before conversion:", non_numeric_before.sum())
print("Unique non-numeric values:", df['Number of Offences'][non_numeric_before].unique())
```

Number of non-numeric values before conversion: 0
Unique non-numeric values: []

Figure 26: check non-numeric values.

```
✓ 0s [18] negative_count = (df['Number of Offences'] < 0).sum()
print("Number of negative values in 'Number of Offences':", negative_count)
```

Number of negative values in 'Number of Offences': 0

Figure 27: check negative values.

```
✓ [19] # Count of non-null, non-NaN values in each column  
0s non_null_counts = df.notna().sum()  
print(non_null_counts)
```

```
Financial Year      98777  
Financial Quarter   98777  
Force Name          98777  
Offence Description  98777  
Offence Group       98777  
Offence Subgroup    98777  
Offence Code        98777  
Number of Offences  98777  
dtype: int64
```

Figure 28: count each column.

And finally, we save changes into a CSV file.

```
✓ 1s df.to_csv('/content/sample_data/cleaned_data.csv', index=False)
```

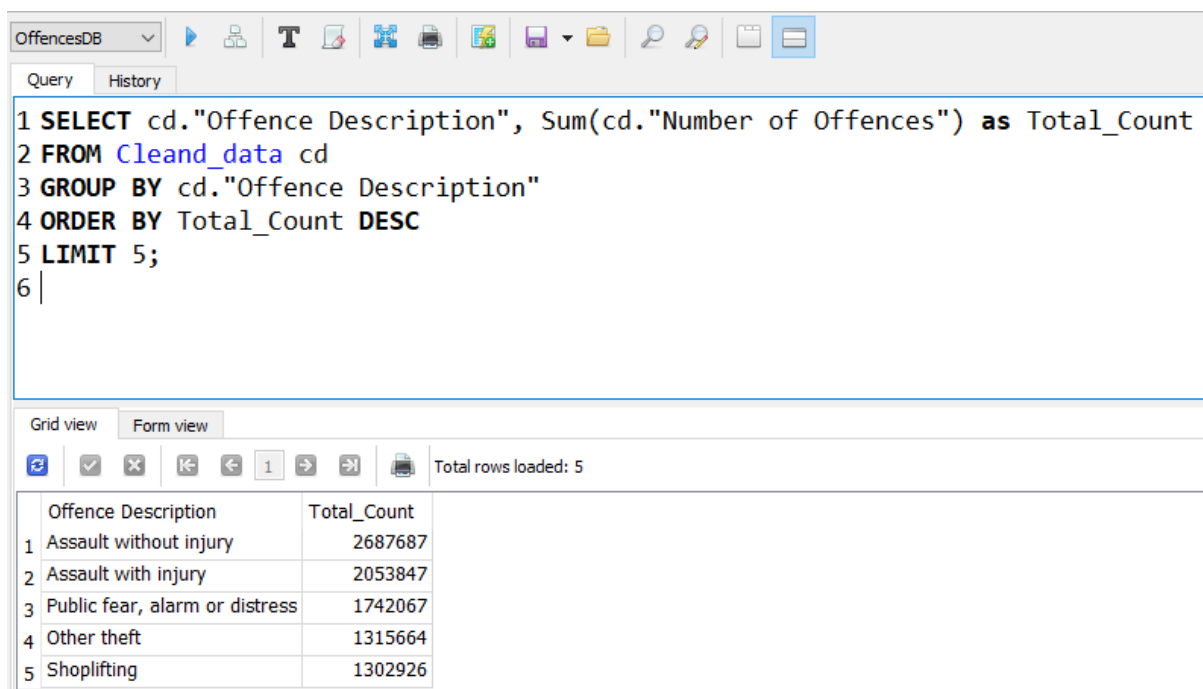
Figure 29: Save changes.

3.0 Queries

3.1 SQL Queries

In this section, we'll use SQL (Structured Query Language) to extract valuable data from our crime dataset. Due to its reputation for efficiency when working with huge datasets, SQL is a useful tool for managing and analysing data. Our main goal is to use SQL queries to solve the following research questions:

1. Identify the five most common crime types.



The screenshot shows a SQL query interface with a toolbar at the top. The database is 'OffencesDB'. The query is as follows:

```
1 SELECT cd."Offence Description", Sum(cd."Number of Offences") as Total_Count
2 FROM Cleand_data cd
3 GROUP BY cd."Offence Description"
4 ORDER BY Total_Count DESC
5 LIMIT 5;
6 |
```

Below the query editor, there are tabs for 'Grid view' and 'Form view'. The 'Grid view' is selected, showing a table with 5 rows and 2 columns: 'Offence Description' and 'Total_Count'. The table data is as follows:

	Offence Description	Total_Count
1	Assault without injury	2687687
2	Assault with injury	2053847
3	Public fear, alarm or distress	1742067
4	Other theft	1315664
5	Shoplifting	1302926

The interface also includes a toolbar with various icons and a status bar indicating 'Total rows loaded: 5'.

Figure 30: Five common offence descriptions.

This query groups the records by crime description, sums the number of offences, and then orders them in descending order to get the top 5 most common crime types across recent years in our specified range.

2. How has the frequency of specific crimes changed annually?

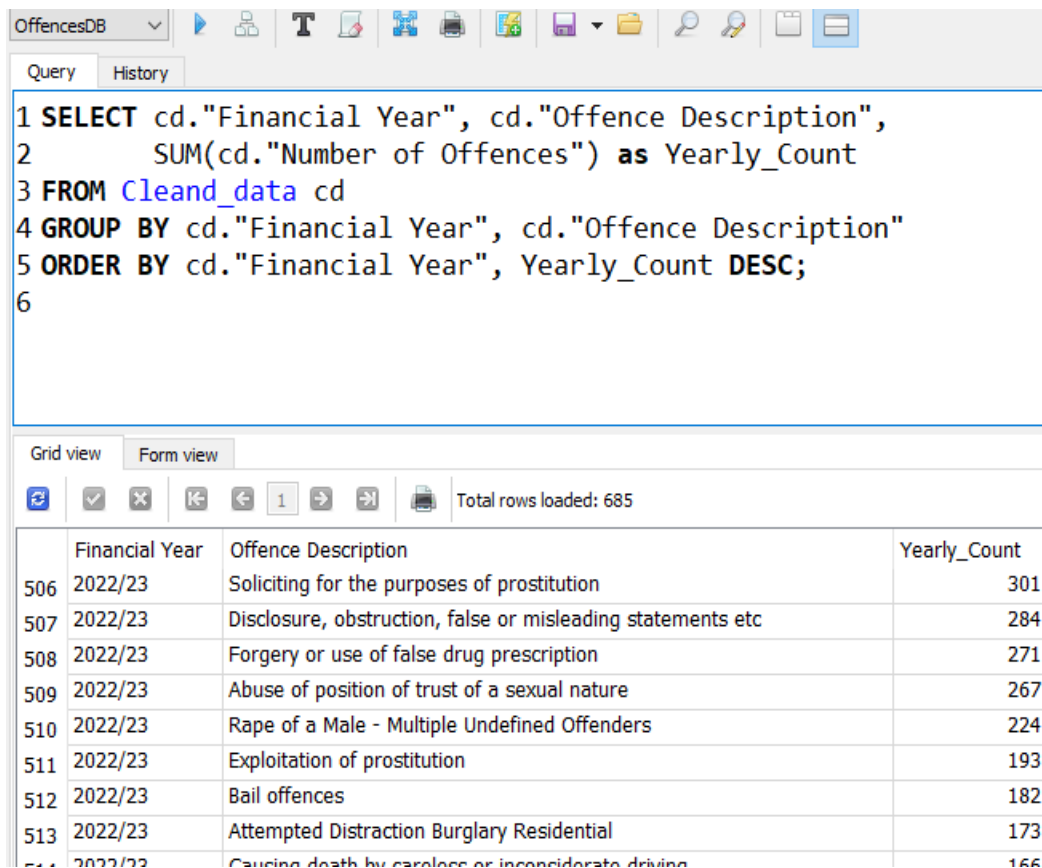
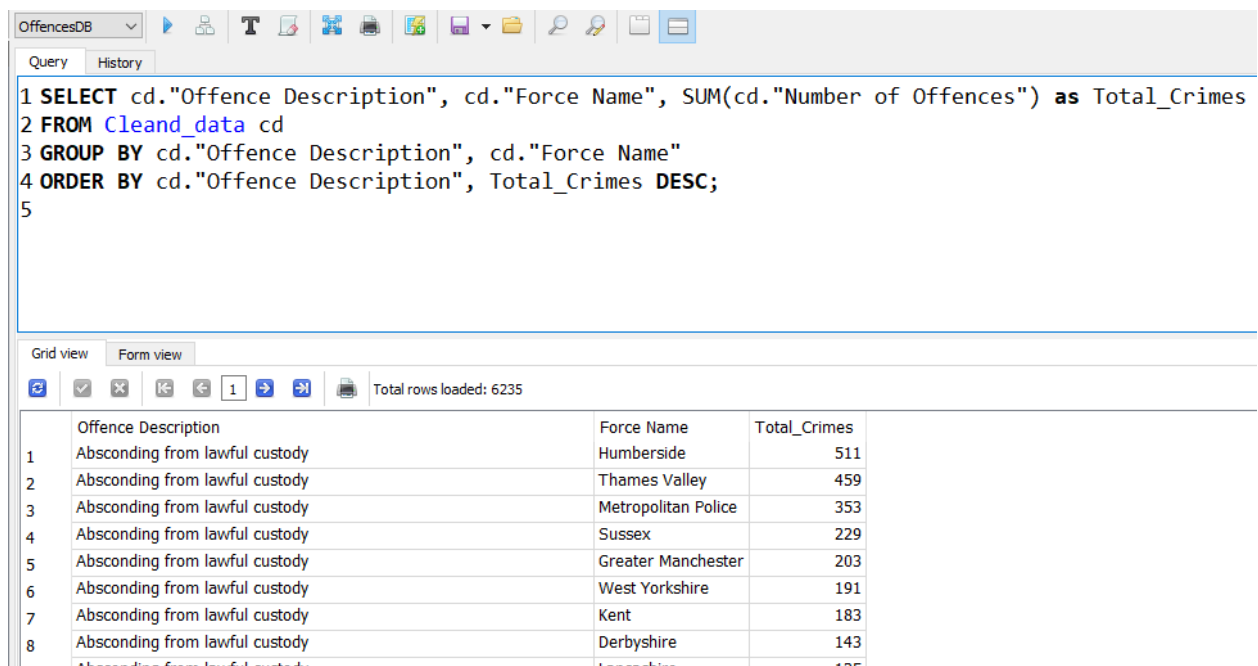


Figure 31: shows the number of offence descriptions according to the years.

This query groups our data by year and crime type, counts the occurrences for each year and type, and then orders the results by year and count. It shows how the frequency of each crime type has changed annually over the specified range.

3. Examine the connection between crime types and police force areas.



The screenshot shows a database query tool interface. At the top, there's a toolbar with various icons. Below it, a tab labeled 'Query' is active, displaying a SQL query. The query is as follows:

```
1 SELECT cd."Offence Description", cd."Force Name", SUM(cd."Number of Offences") as Total_Crimes
2 FROM Cleand_data cd
3 GROUP BY cd."Offence Description", cd."Force Name"
4 ORDER BY cd."Offence Description", Total_Crimes DESC;
5
```

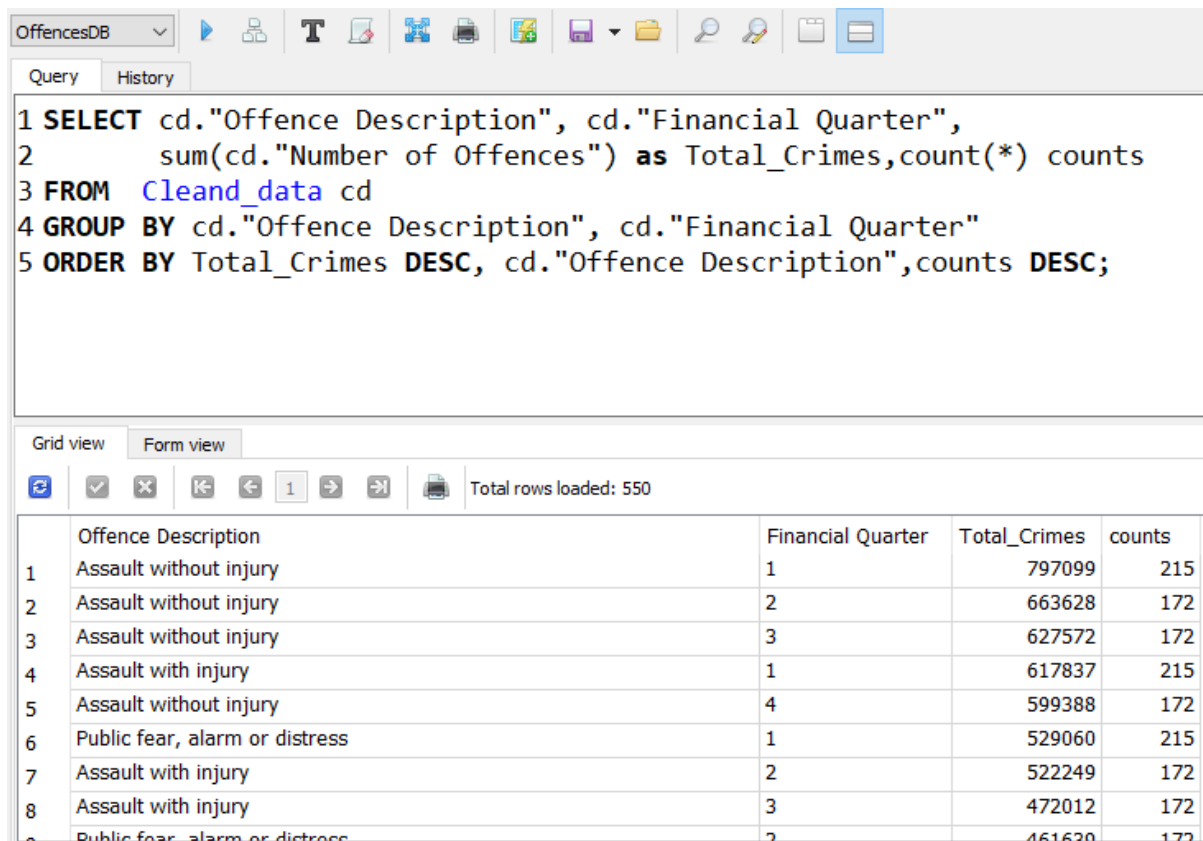
Below the query editor, there's a 'Form view' tab. Underneath it, a toolbar contains icons for grid view, form view, and other navigation functions. A status bar indicates 'Total rows loaded: 6235'. The main area displays a grid view of the query results, which is a table with three columns: 'Offence Description', 'Force Name', and 'Total_Crimes'. The first eight rows of the table are shown, with the first column truncated for rows 2 through 8.

	Offence Description	Force Name	Total_Crimes
1	Absconding from lawful custody	Humberside	511
2	Absconding from lawful custody	Thames Valley	459
3	Absconding from lawful custody	Metropolitan Police	353
4	Absconding from lawful custody	Sussex	229
5	Absconding from lawful custody	Greater Manchester	203
6	Absconding from lawful custody	West Yorkshire	191
7	Absconding from lawful custody	Kent	183
8	Absconding from lawful custody	Derbyshire	143

Figure 32: shows the relation between crime and force name.

This query groups data by crime type (Offence Description) and location (Force Name) and counts the occurrences.

4. Examine how the crime rate varies with the quarters.



The screenshot shows a database query tool interface. At the top, there's a toolbar with various icons. Below it, a 'Query' tab is active, displaying a SQL query. The query is as follows:

```
1 SELECT cd."Offence Description", cd."Financial Quarter",
2       sum(cd."Number of Offences") as Total_Crimes, count(*) counts
3 FROM   Cleand_data cd
4 GROUP BY cd."Offence Description", cd."Financial Quarter"
5 ORDER BY Total_Crimes DESC, cd."Offence Description", counts DESC;
```

Below the query editor, there's a 'Grid view' tab. It shows a table with 5 columns: 'Offence Description', 'Financial Quarter', 'Total_Crimes', and 'counts'. The table contains 9 rows of data. The first row is 'Assault without injury' in quarter 1 with 797099 total crimes and 215 counts. The second row is 'Assault without injury' in quarter 2 with 663628 total crimes and 172 counts. The third row is 'Assault without injury' in quarter 3 with 627572 total crimes and 172 counts. The fourth row is 'Assault with injury' in quarter 1 with 617837 total crimes and 215 counts. The fifth row is 'Assault without injury' in quarter 4 with 599388 total crimes and 172 counts. The sixth row is 'Public fear, alarm or distress' in quarter 1 with 529060 total crimes and 215 counts. The seventh row is 'Assault with injury' in quarter 2 with 522249 total crimes and 172 counts. The eighth row is 'Assault with injury' in quarter 3 with 472012 total crimes and 172 counts. The ninth row is 'Public fear, alarm or distress' in quarter 2 with 461620 total crimes and 172 counts. The interface also shows 'Total rows loaded: 550'.

	Offence Description	Financial Quarter	Total_Crimes	counts
1	Assault without injury	1	797099	215
2	Assault without injury	2	663628	172
3	Assault without injury	3	627572	172
4	Assault with injury	1	617837	215
5	Assault without injury	4	599388	172
6	Public fear, alarm or distress	1	529060	215
7	Assault with injury	2	522249	172
8	Assault with injury	3	472012	172
9	Public fear, alarm or distress	2	461620	172

Figure 33: found relation between offences in each quarter.

This query identifies seasonal trends in crime occurrences.

5. Find out which places have the highest and lowest rates of criminal activity.

We divided this question into two queries:

OffencesDB

Query History

```

1 SELECT cd."Force Name",
2       SUM(cd."Number of Offences") AS Total_Crimes,
3       COUNT(DISTINCT cd."Financial Year") AS Years_Counted,
4       ROUND((SUM(cd."Number of Offences") / COUNT(DISTINCT cd."Financial Year")), 2)
5       AS Avg_Crimes_Per_Year
6 FROM cleand_data cd
7 GROUP BY cd."Force Name"
8 ORDER BY Total_Crimes
9 limit 5;
10

```

Grid view Form view

Total rows loaded: 5

	Force Name	Total_Crimes	Years_Counted	Avg_Crimes_Per_Year
1	London, City of	27942	5	5588
2	Cumbria	154957	5	30991
3	Dyfed-Powys	162666	5	32533
4	Warwickshire	170988	5	34197
5	Wiltshire	176859	5	35371

Figure 34: lowest rate of criminal activity.

OffencesDB

Query History

```

1 SELECT cd."Force Name",
2       SUM(cd."Number of Offences") AS Total_Crimes,
3       COUNT(DISTINCT cd."Financial Year") AS Years_Counted,
4       ROUND((SUM(cd."Number of Offences") / COUNT(DISTINCT cd."Financial Year")), 2)
5       AS Avg_Crimes_Per_Year
6 FROM cleand_data cd
7 GROUP BY cd."Force Name"
8 ORDER BY Total_Crimes Desc
9 limit 5;
10

```

Grid view Form view

Total rows loaded: 5

	Force Name	Total_Crimes	Years_Counted	Avg_Crimes_Per_Year
1	Metropolitan Police	3591984	5	718396
2	Greater Manchester	1390132	5	278026
3	West Midlands	1346698	5	269339
4	West Yorkshire	1226618	5	245323
5	Kent	757818	5	151563

Figure 35: Force name with highest criminal rates.

3.2 Hive Queries

The use of Apache Hive for criminal data analysis is examined in this section, emphasising the capability of HiveQL to handle and query massive datasets. Hive is a crucial component of big

data analytics in criminal justice because it enables the speedy analysis of enormous amounts of crime data and helps identify patterns and trends that are crucial for both anticipating and preventing crime. These queries are intended to extract critical insights, such as crime hotspots and temporal patterns, demonstrating Hive's capabilities for improving data-driven crime prevention initiatives. Our main goal is to use Hive queries to solve the following research questions:

1. How do crime rates vary across different financial quarters?

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sum

# Initialize Spark Session
spark = SparkSession.builder \
    .appName("Crime Analysis") \
    .enableHiveSupport() \
    .getOrCreate()

df = spark.read.csv("/content/sample_data/cleaned_data.csv", header=True, inferSchema=True)

# Group by "Financial Quarter" and sum "Number of Offences"
df_summed = df.groupBy("Financial Quarter").agg(sum("Number of Offences").alias("total_offences"))

# Order by the sum of "Number of Offences" in descending order
df_summed_ordered = df_summed.orderBy(col("total_offences"), ascending=False)

# Show the result
df_summed.show()

# Stop the Spark session
spark.stop()
```

Figure 36: Hive Queries_q1.

Financial Quarter	total_offences
1	6459000
3	5154518
4	4983585
2	5397100

Figure 37: result Hive queries_q1.

2. Which police force areas have seen the biggest shifts in crime rates over the course of various fiscal quarters?

```

from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sum

# Initialize Spark Session
spark = SparkSession.builder \
    .appName("Crime Analysis") \
    .enableHiveSupport() \
    .getOrCreate()

df = spark.read.csv("/content/sample_data/cleaned_data.csv", header=True, inferSchema=True)

# Group by "Financial Quarter" and "Force Name", and sum "Number of Offences"
df_summed = df.groupBy("Financial Quarter", "Force Name").agg(sum("Number of Offences").alias("total_offences"))

# Order the results by "Financial Quarter" and the sum of "Number of Offences" in descending order
df_summed_ordered = df_summed.orderBy("Financial Quarter", col("total_offences"), ascending=False)

# Show the result
df_summed_ordered.show()

# Stop the Spark session
spark.stop()

```

Figure 38: Hive Queries_q2.

Financial Quarter	Force Name	total_offences
4	Metropolitan Police	812573
4	Greater Manchester	322915
4	West Midlands	319177
4	West Yorkshire	280508
4	Kent	166679
4	Thames Valley	163953
4	Hampshire	155365
4	Essex	153834
4	Merseyside	147251
4	South Yorkshire	144707
4	Northumbria	135237
4	Avon and Somerset	129948
4	Lancashire	125331
4	Sussex	114299
4	South Wales	99271
4	Nottinghamshire	97465
4	Leicestershire	94445
4	Humberside	92667
4	Devon and Cornwall	91325
4	Cheshire	84718

only showing top 20 rows

Figure 39: result Hive queries_q2.

4.0 MapReduce

The first step in the MapReduce segment involves installing the necessary libraries and tools to facilitate MapReduce operations. This stage involves configuring the computational

environment required to execute PySpark, a popular large data processing tool. This basic configuration ensures that the ensuing data analysis will be carried out effectively and efficiently.

```
✓ [1] !apt-get install openjdk-8-jdk-headless -qq > /dev/null
36s !wget -q http://archive.apache.org/dist/spark/spark-3.1.1/spark-3.1.1-bin-hadoop3.2.tgz
!tar xf spark-3.1.1-bin-hadoop3.2.tgz
!pip install -q findspark

✓ [2] import os
0s os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.1.1-bin-hadoop3.2"

✓ [3] import findspark
7s findspark.init()
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[*]").getOrCreate()
spark.conf.set("spark.sql.repl.eagerEval.enabled", True) # Property used to format output tables better
```

Figure 40: install tools and libraries before running MapReduce.

We construct a tuple structure within the context of a data analytics project, containing the necessary data items to answer the question, "How many crimes happen each year?" This tuple consists of the year and the total number of crimes. Following that, the analytical method consists of two critical stages: mapping, which converts data into this tuple format, and reduction, which adds up these tuples to get the total number of crimes committed annually. This technique can be used to process and analyse data efficiently using the MapReduce paradigm.

```
from pyspark import SparkConf, SparkContext

def parseCrime(line):
    fields = line.split(',')
    year = fields[0]
    offences = int(fields[-1])
    return (year, offences)

if __name__ == "__main__":
    conf = SparkConf().setAppName("CrimeAnalysis")
    sc = SparkContext.getOrCreate(conf=conf)

    lines = sc.textFile("/content/sample_data/cleaned_data.csv")

    # Filter out the header row
    header = lines.first()
    filteredLines = lines.filter(lambda line: line != header)
```

```

# Map and Reduce
crimeData = filteredLines.map(parseCrime)
crimeCounts = crimeData.reduceByKey(lambda x, y: x + y)

# Collect and print results
results = crimeCounts.collect()
for result in results:
    print(f"Year: {result[0]}, Total Offences: {result[1]}")

```

and the result is:

```

Year: 2019/20, Total Offences: 5233676
Year: 2021/22, Total Offences: 5281533
Year: 2020/21, Total Offences: 4570453
Year: 2022/23, Total Offences: 5516196
Year: 2023/24, Total Offences: 1392345

```

Figure 41: Response of MapReduce by PySpark.

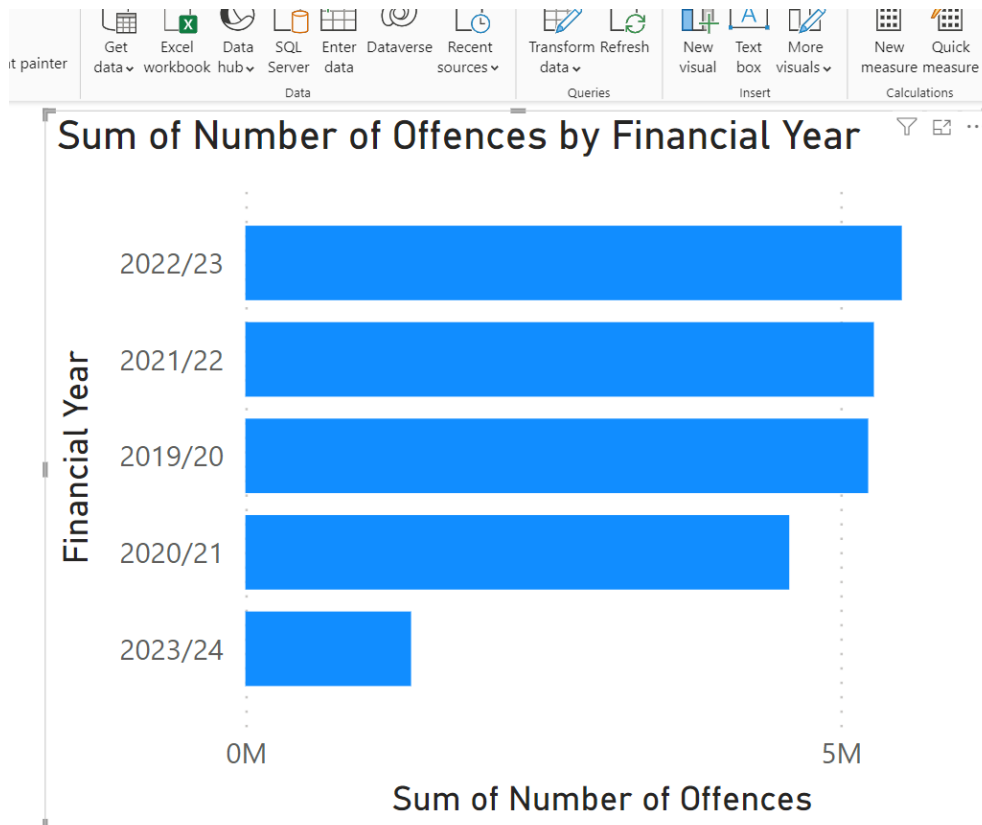


Figure 42: Show result in powerBI.

The observed decline in the crime rate for 2023–2024 may be explained by the fact that the data only covers a quarter of the year. Due to insufficient data collection, the total crime number for 2023–2024 is lower; nevertheless, this decline could not accurately reflect annual crime patterns.

5.0 Machine Learning

5.1 Data Pre-processing

5.1.1 Initial Data Analysis

We started by running a statistical analysis on our dataset using Python's Pandas package. This meant calculating several metrics for each column, such as the standard deviation and mean.



Figure 43: summary statistics

The 'Number of Offences' column receives particular attention because it includes numerical data. Since this column provides unique viewpoints that are critical to our study, it is examined independently. Since the summary numbers in this column provide a clear, quantitative indicator of criminal events, they are particularly important.

summary	Number of Offences
count	98777
mean	222.66522571043868
stddev	861.6207697187683
min	0
max	29480

Figure 44: summary statistics for a numeric column.

This step gave insight into the characteristics of the data, which aided in feature selection.

5.1.2 Feature Selection

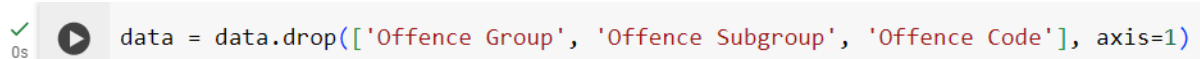
Selected columns for the project:

1. Financial Year
2. Financial Quarter
3. Force Name

4. Offence Description
5. Number of Offences

5.1.3 Dropping Irrelevant Columns

We simplified our dataset by eliminating the fields 'Offence Group', 'Offence Subgroup', and 'Offence Code'.



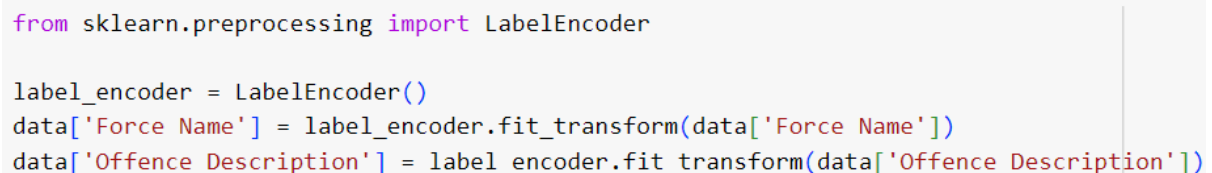
```
data = data.drop(['Offence Group', 'Offence Subgroup', 'Offence Code'], axis=1)
```

Figure 45: drop irrelevant columns.

5.1.4 Data Transformation

Since we have decided to use a regression technique for predictive modelling, all column types must be converted to numerical format. To ensure that all of the data meet the numerical requirements of the regression model, this transformation makes use of many strategies, including further adjustments to the year column and label encoding for category text columns.

1. Label Encoding: Applied to 'Force Name' and 'Offence Description' using scikit-learn:

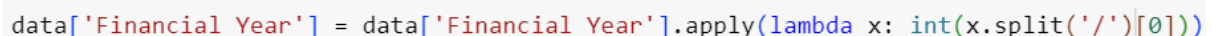


```
from sklearn.preprocessing import LabelEncoder

label_encoder = LabelEncoder()
data['Force Name'] = label_encoder.fit_transform(data['Force Name'])
data['Offence Description'] = label_encoder.fit_transform(data['Offence Description'])
```

Figure 46: Label Encoding.

2. Year Column Modification: Transforming 'Financial Year' to numeric:



```
data['Financial Year'] = data['Financial Year'].apply(lambda x: int(x.split('/')[0]))
```

Figure 47: year Column Modification.

After that, we save the last changes in a new file and recheck duplicate rows.

```
✓ [10] # Save the transformed DataFrame to a new CSV file
0s      data.to_csv('transformed_dataset.csv', index=False)

✓ [11] duplicate_rows = data[data.duplicated()]
0s      print("Number of duplicate rows: ", duplicate_rows.shape[0])

      Number of duplicate rows:  0
```

Figure 48: save the transformed file.

5.1.5 Splitting Train and Test

5.1.5.1 The Reasons to Select Sequential Data Splitting:

Forecasting crime rates using force names, year, quarter, and crime statistics is the aim of this study. Because our data is time-sensitive, we adopt a sequential splitting technique for the following reasons:

1. Temporal Relevance: The features in our dataset are time-dependent, meaning that over the course of quarters and years, crime trends may change. Sequential splitting makes sure the model can predict future trends before utilising the knowledge gleaned from historical data—years and quarters.
2. Realistic Evaluation: Using training data from 2019 to 2022 (up to the first quarter) and testing data from the remaining quarters of 2022 and the first quarter of 2023, we reproduce a realistic scenario in which the model predicts future events based on past observations.

We want to improve the prediction accuracy and reliability of our model for future crime rates by leveraging temporal trends seen in past data.

5.1.5.2 perform a sequential split:

The training and test datasets are split using Pandas according to the selected years and quarters so that we can segment our dataset sequentially for Python time-series forecasting:

```
import pandas as pd

data = pd.read_csv('transformed_dataset.csv')

# Define training and testing periods
train_end_year = 2022
train_end_quarter = 1

# Split the dataset into training and test sets based on the condition
train_data = data[(data['Financial Year'] < train_end_year) | ((data['Financial Year'] == train_end_year) & (data['Financial Quarter'] <= train_end_quarter))]
test_data = data[(data['Financial Year'] > train_end_year) | ((data['Financial Year'] == train_end_year) & (data['Financial Quarter'] > train_end_quarter))]

# Optionally, save the split datasets
train_data.to_csv('train_dataset.csv', index=False)
test_data.to_csv('test_dataset.csv', index=False)
```

Figure 49: split dataset to train and test.

```
Number of records in the training dataset: 75127
Number of records in the test dataset: 23650
```

Figure 50: count of rows for train and test.

5.2 Prediction Algorithms

Our project's primary goal was to estimate crime rates using regression analysis. Given the structure of our dataset, regression models are useful for determining how different factors influence crime trends over time.

We selected multiple regression models because we were aware that different algorithms might capture different data aspects. This variant offered a comprehensive understanding of the fundamental patterns. Before using any prediction methods, it is necessary to identify the dependent and independent variables in our dataset. Assigning the 'y' vector to the dependent variable 'Number of Offences' in the column is where this project stands. The purpose of our research, which is to forecast crime rates, drove this decision. As a result, the remaining selected columns, which represent various driving factors, are put on the 'x' vector and make up the collection of independent variables that we are studying.

```

import pandas as pd
DF_train=pd.read_csv('/content/sample_data/train_dataset.csv')
DF_test=pd.read_csv('/content/sample_data/test_dataset.csv')

DF_train_final = DF_train.loc[:,['Financial Year', 'Financial Quarter','Force Name','Offence Description','Number of Offences']]
DF_test_final = DF_test.loc[:,['Financial Year', 'Financial Quarter','Force Name','Offence Description','Number of Offences']]
#DF_train_final
x_train = DF_train_final.loc[:,['Financial Year', 'Financial Quarter','Force Name','Offence Description']]
#x_train

y_train = DF_train_final.loc[:,['Number of Offences']]
#y_train
x_test = DF_test_final.loc[:,['Financial Year', 'Financial Quarter','Force Name','Offence Description']]
#x_train

y_test = DF_test_final.loc[:,['Number of Offences']]
#y_test

```

Figure 51: define X and Y vectors.

.ravel() Method:

Both the Random Forest methodology with a configurable number of estimators and Support Vector Regression (SVR) with varying kernels were implemented using the .ravel() method. Since these techniques assume that the target variable (y_train) is one-dimensional, this method was required. However, the y_train in our dataset was initially stored in a two-dimensional DataFrame type. y_train was effectively converted into a one-dimensional NumPy array by using .ravel(), which matches the required input structure of these methods and guarantees the compliance and utility of the model training process.

In the below, we discuss each model:

5.2.1 Linear Regression:

A basic regression technique for building a performance baseline. A linear approach is used to illustrate the relationship between a dependent variable and one or more independent variables.

```

from sklearn.linear_model import LinearRegression

model_LR = LinearRegression()
model_LR.fit(x_train,y_train)

```

▼ LinearRegression
LinearRegression()

Figure 52: Linear regression train algorithm.

```
# prediction Linear Regression
y_pred_LR = model_LR.predict(x_test)
```

Figure 53: Linear regression prediction.

5.2.2 Linear SVR (Support Vector Regression):

An adaptation of Support Vector Machines (SVM) for regression tasks. It functions well for high-dimensional datasets.

```
from sklearn.svm import SVR

model_linear= SVR(kernel = 'linear')
model_linear.fit (x_train,y_train.values.ravel())
```

Figure 54: Linear SVR algorithm.

5.2.3 RBF SVR (Radial Basis Function Support Vector Regression):

This SVR version is suitable for complex datasets since it models non-linear correlations in the data using the Radial Basis Function kernel.

```
model_RBF= SVR(kernel = 'rbf')
model_RBF.fit (x_train,y_train.values.ravel())
```

Figure 55: RBF SVR algorithm.

5.2.4 Polynomial SVR:

It offers a method for handling nonlinear patterns by simulating and exploring polynomial relationships inside data using polynomial kernels in SVR.

```
model_poly= SVR(kernel = 'poly')
model_poly.fit (x_train,y_train.values.ravel())
```

Figure 56: Polynomial SVR train algorithm.

```
# Prediction different type of SVR
y_pred_linear_SVR = model_linear.predict(x_test)
y_pred_RBF_SVR = model_RBF.predict(x_test)
y_pred_poly_SVR = model_poly.predict(x_test)
```

Figure 57: prediction of different types of SVR.

5.2.5 Decision Tree:

A model based on trees that splits data into branches to make predictions and is particularly good at identifying relationships between variables and nonlinear patterns.

```
from sklearn.tree import DecisionTreeRegressor

model_DT = DecisionTreeRegressor()
model_DT.fit(x_train, y_train)
```

▼ DecisionTreeRegressor
DecisionTreeRegressor()

Figure 58: Decision Tree algorithm.

```
# Prediction Decision Tree
y_pred_DT = model_DT.predict(x_test)
```

Figure 59: prediction Decision Tree.

5.2.6 Random Forest:

An ensemble approach that reduces the risk of overfitting while maximising prediction accuracy and robustness through the employment of many decision trees.

```
from sklearn.ensemble import RandomForestRegressor

model_RF_50 = RandomForestRegressor(n_estimators = 50)
model_RF_50.fit(x_train, y_train.values.ravel())
model_RF_100 = RandomForestRegressor(n_estimators = 100)
model_RF_100.fit(x_train, y_train.values.ravel())
model_RF_200 = RandomForestRegressor(n_estimators = 200)
model_RF_200.fit(x_train, y_train.values.ravel())
```

▼ RandomForestRegressor
RandomForestRegressor(n_estimators=200)

Figure 60: Random Forest algorithms.

```
# prediction different type of Random Forest
y_pred_RF_50 = model_RF_50.predict(x_test)
y_pred_RF_100 = model_RF_100.predict(x_test)
y_pred_RF_200 = model_RF_200.predict(x_test)
```

Figure 61: Prediction of different types of Random Forests.

5.2.7 Evaluation of Models by using MSE:

In our project which aims at crime trend prediction, we utilized Mean Squared Error (MSE) as a metric to compare and assess the accuracy of several predictive models. Mean square error, or MSE, is the average of the squares of the errors between the values that were expected and those that were observed. It is computed using the following formula:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Figure 62: MSE formula.

Where y_i represents the actual values, \hat{y}_i the predicted values, and n the number of observations.

This metric was crucial in our study to compare and evaluate the performance of the various algorithms used. The average squared difference (MSE) provides a simple, quantitative evaluation of the model's accuracy by calculating the difference between the observed actual outcomes and the model's predictions. Using Mean Squared Error (MSE), we were able to objectively assess the prediction performances of different models, such as Random Forest with a varied number of estimators and SVR with different kernels. Based on this comparison, we were able to identify the model that predicted crime rates the most accurately.


```

from sklearn import metrics
import numpy as np
mse_LR = metrics.mean_squared_error(y_test,y_pred_LR)
mse_linear = metrics.mean_squared_error(y_test,y_pred_linear_SVR)
mse_rbf = metrics.mean_squared_error(y_test,y_pred_RBF_SVR)
mse_poly = metrics.mean_squared_error(y_test,y_pred_poly_SVR)

mse_DT = metrics.mean_squared_error(y_test,y_pred_DT)
mse_RF_50 = metrics.mean_squared_error(y_test,y_pred_RF_50)
mse_RF_100 = metrics.mean_squared_error(y_test,y_pred_RF_100)
mse_RF_200 = metrics.mean_squared_error(y_test,y_pred_RF_200)

```

Figure 63: calculate MSE.

5.2.8 Result:

While each model provided distinct insights, the Best Prediction Model is the Decision Tree model that produced the lowest MSE. This result indicates that it was the most accurate at forecasting crime rates in our dataset. Its ability to handle complicated, non-linear relationships in data may have contributed to its performance; additionally, the model stood out for its predictive capacity, giving useful information for analysing and anticipating crime trends.

```

print("MSE (Linear Regression):", mse_LR)
print("MSE (Linear SVR):", mse_linear)
print("MSE (RBF SVR):", mse_rbf)
print("MSE (Polynomial SVR):", mse_poly)
print("MSE (Decision Tree):", mse_DT)
print("MSE (Random Forest 50):", mse_RF_50)
print("MSE (Random Forest 100):", mse_RF_100)
print("MSE (Random Forest 200):", mse_RF_200)

```

```

MSE (Linear Regression): 842079.6658624114
MSE (Linear SVR): 890789.3457830012
MSE (RBF SVR): 891626.8903137272
MSE (Polynomial SVR): 891232.4989417826
MSE (Decision Tree): 17901.481437632134
MSE (Random Forest 50): 23045.840141547567
MSE (Random Forest 100): 23119.563504190275
MSE (Random Forest 200): 21797.212839216703

```

Figure 64: Show MSE results.

```
import matplotlib.pyplot as plt
import numpy as np

# vector x
x = np.array(["LR", "Linear", "RBF", "POLY", "DT", "RF_50", "RF_100", "RF_200"])

# vector y
y = np.array([mse_LR, mse_linear, mse_rbf, mse_poly, mse_DT, mse_RF_50, mse_RF_100, mse_RF_200])

# plt.bar(x, y)
plt.xlabel('Models')
plt.ylabel('MSE')
plt.title('MSE Comparison Among Different Models')

plt.bar(x,y)
plt.show()
```

Figure 65: Show diagram code.

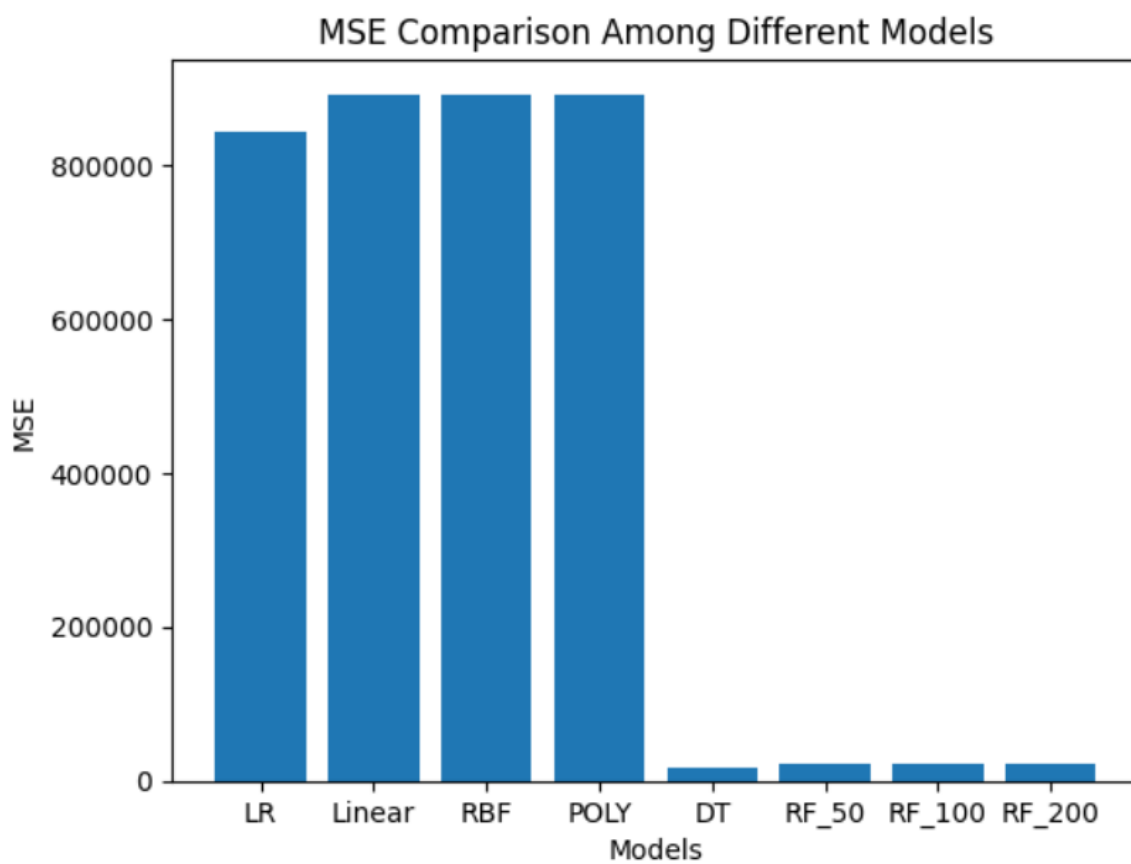


Figure 66: Comparison Diagram.

5.3 Data Visualization

Dashboard PowerBI

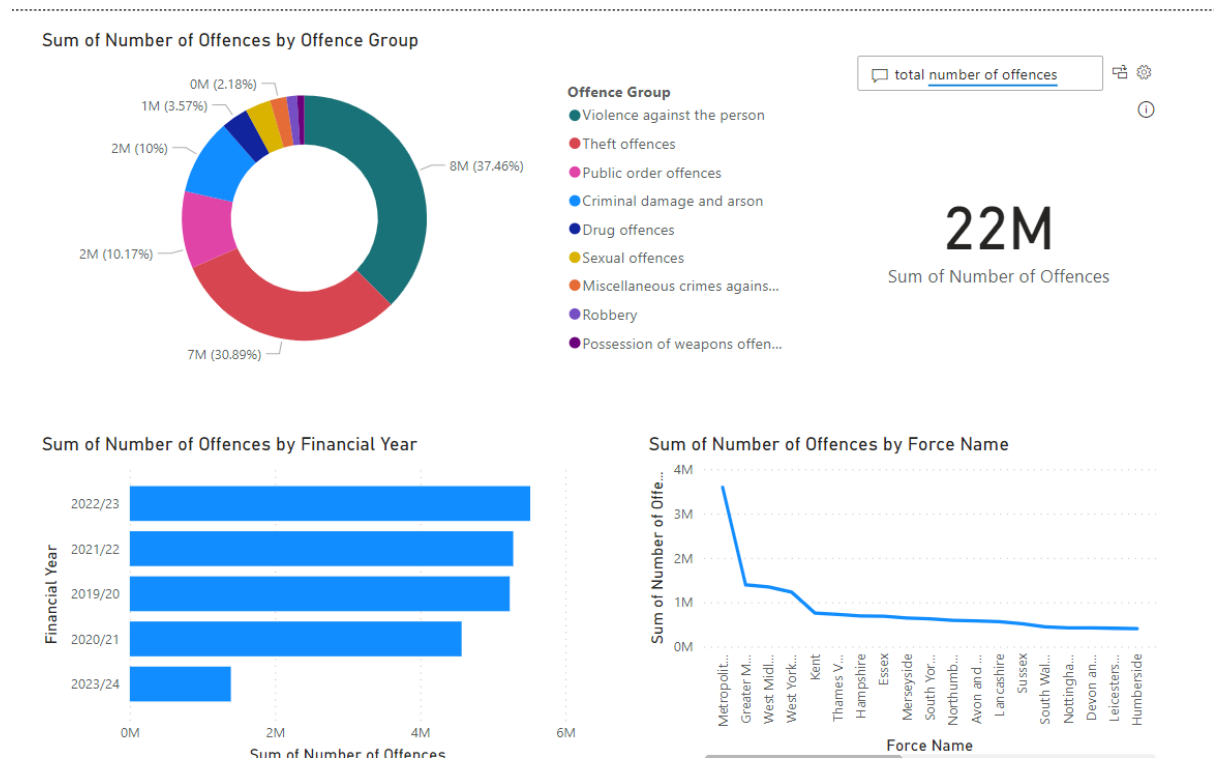


Figure 67: Dashboard of powerBI.

Pie Chart: Sum of Number of Offences by Offence Group

This pie chart delineates the proportion of total recorded offences categorized by offence groups. Each segment of the pie represents a distinct category of crime, with the segment size proportional to the frequency of the offences. The largest segment indicates the most frequent offence group, while smaller segments represent less frequent categories. This visualization aids in quickly identifying which types of crimes are most prevalent, providing a visual summary of the crime landscape.

Bar Chart: Sum of Number of Offences by Financial Year

The bar chart illustrates the total number of offences recorded annually, with bars arranged according to the financial year. The height of each bar corresponds to the total offences recorded for that year, offering a clear representation of the trends and changes in crime

rates over time. This chart allows viewers to observe how the incidence of crime has evolved and possibly correlate these trends with policy changes or enforcement strategies.

Line Chart: Sum of Number of Offences by Force Name

The line chart depicts the variation in the total number of offences across different police forces. Each point on the line represents a police force, and its vertical position reflects the total offences recorded by that force over a specified period. This visualization helps examine and compare the performance of various police forces in addressing crimes and can identify regions with the highest and lowest crime rates.

KPI Metric (Text Beside Charts): Total Sum of Number of Offences

The large numeric display adjacent to the charts provides a cumulative summary of the offences represented in the other visualizations. The figure highlights the overall total number of offences within the examined timeframe, serving as a reference point for comparing the data displayed in the other charts.

6.0 Conclusion:

We created a strong basis for predicting crime trends in this area by carrying out a thorough examination and organising the data. During this time, we concentrated on meticulously gathering and evaluating the data. Using this approach, we cleaned the data, chose key properties, and made any necessary modifications to ensure the data were in the proper format for our research. Understanding and organising our data took time and effort, but in the end, we built a solid basis for the predictive models that came after. After making this preparation, we used the Mean Squared Error (MSE) metric to test and assess the effectiveness of multiple predictive algorithms. This approach allowed us to identify the best algorithms for effectively predicting crime rates.