Experiment 1

Name: Ayush Maurya Class:D15C/34

Experiment 1

Aim: Introduction to Data science and Data preparation using Pandas steps.

Theory:

Data science is the study of data that helps us derive useful insight for business decision making. Data Science is all about using tools, techniques, and creativity to uncover insights hidden within data. It combines math, computer science, and domain expertise to tackle real-world challenges in a variety of fields.

Data science involves these key steps:

- **Data Collection:** Gathering raw data from various sources, such as databases, sensors, or user interactions.
- **Data Cleaning:** Ensuring the data is accurate, complete, and ready for analysis.
- **Data Analysis:** Applying statistical and computational methods to identify patterns, trends, or relationships.
- **Data Visualization:** Creating charts, graphs, and dashboards to present findings clearly.
- **Decision-Making:** Using insights to inform strategies, create solutions, or predict outcomes.

Dataset Overview:

The dataset consists of 21 columns, each providing valuable insights into vehicle details, registration data, and vehicle types across different locations. Below is a breakdown of the dataset's columns and their significance:

- **ID**: A unique identifier assigned to each vehicle, distinguishing them in the inventory.
- **Plate Type**: Represents the type of plate assigned to the vehicle, reflecting its registration or classification.
- **Primary Customer City**: The city where the primary customer resides, giving insights into regional distribution.
- **Primary Customer State**: The state where the primary customer resides, which helps in analyzing geographic patterns.
- **Registration Start Date**: The start date of the vehicle's registration, indicating the time frame for its use.
- **Registration Expiration Date**: The expiration date of the vehicle's registration, helping track the validity of the registration.

- **Registration Usage**: Represents the usage category of the vehicle's registration, such as regular or commercial use.
- **Vehicle Type**: The classification of the vehicle, such as Motorcycle or Passenger, giving insights into vehicle segmentation.
- **Vehicle Weight**: The weight of the vehicle, which could be relevant for logistics and fuel efficiency considerations.
- **Vehicle Year**: The year the vehicle was manufactured, which is crucial for understanding its age and market trends.
- **Vehicle Make**: The manufacturer or brand of the vehicle, which is vital for brand-specific market analysis.
- **Vehicle Model**: The specific model of the vehicle, helping identify product features and target consumer segments.
- **Vehicle Body**: Describes the type of body the vehicle has, such as Sedan (SD) or SUV, aiding in vehicle categorization.
- **Primary Color**: The primary color of the vehicle, which can influence customer preference and aesthetics.
- **Vehicle Declared Gross Weight**: The gross weight of the vehicle as declared by the manufacturer, important for regulatory and logistical purposes.
- **Fuel Code**: Represents the fuel type used by the vehicle, such as Electric (E00) or Hybrid (H04), helping to analyze energy consumption patterns.
- **Vehicle Recorded GVWR**: The Gross Vehicle Weight Rating (GVWR) recorded for the vehicle, a key measure for vehicle classification.
- **Vehicle Name**: The official name or model name of the vehicle, assisting in product-specific analysis.
- **Type**: The vehicle's classification type, such as BEV (Battery Electric Vehicle) or PHEV (Plug-in Hybrid Electric Vehicle), influencing environmental analysis.
- **Vehicle Category**: The classification of the vehicle based on its size and usage, such as Light-Duty (Class 1-2).

Problem Statement:

The dataset provides detailed information about various vehicles, including their registration, attributes, and classifications. The primary objectives of analyzing this dataset are:

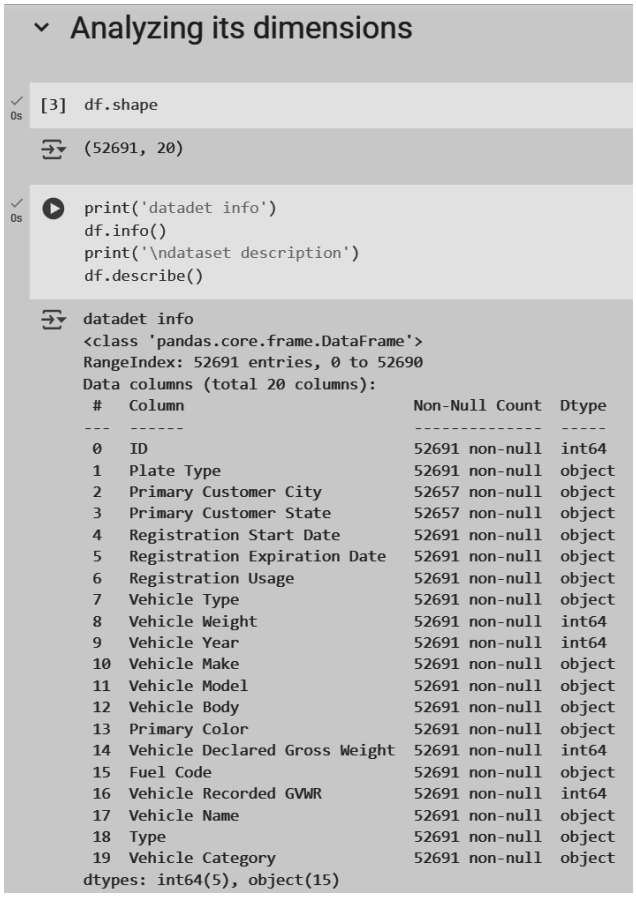
- **Vehicle Performance**: Identifying which vehicle types, makes, and models perform better in terms of registration and customer preferences.
- **Customer Insights**: Understanding the impact of customer location (city and state) on vehicle preferences and registrations.

- **Fuel Type Analysis**: Investigating how different fuel types (BEV, PHEV, etc.) influence vehicle demand and registration trends.
- **Pricing and Brand Impact**: Analyzing how vehicles make, model, and type correlate with pricing strategies and customer behavior.
- **Sales and Registration Trends**: Exploring how vehicle year and type affect registration volume over time, helping in forecasting and inventory management.

By processing and analyzing this dataset, the goal is to uncover trends that can assist in strategic decisions related to vehicle marketing, inventory, and customer targeting.

Code:

1] This returns a tuple indicating the number of rows and columns in the DataFrame. This code prints "dataset info" and displays the DataFrame's structure including data types and non-null counts using `df.info()`.
2] It then prints "dataset description" and shows summary statistics like mean, standard deviation, and percentiles for numerical columns with `df.describe()`.



Analyzing its dimensions

```
[3]  df.shape
0s
(52691, 20)

print('datadet info')
df.info()
print('\ndataset description')
df.describe()
0s
datadet info
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 52691 entries, 0 to 52690
Data columns (total 20 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   ID                              52691 non-null  int64
 1   Plate Type                      52691 non-null  object
 2   Primary Customer City           52657 non-null  object
 3   Primary Customer State          52657 non-null  object
 4   Registration Start Date         52691 non-null  object
 5   Registration Expiration Date    52691 non-null  object
 6   Registration Usage              52691 non-null  object
 7   Vehicle Type                    52691 non-null  object
 8   Vehicle Weight                  52691 non-null  int64
 9   Vehicle Year                    52691 non-null  int64
 10  Vehicle Make                    52691 non-null  object
 11  Vehicle Model                   52691 non-null  object
 12  Vehicle Body                    52691 non-null  object
 13  Primary Color                   52691 non-null  object
 14  Vehicle Declared Gross Weight   52691 non-null  int64
 15  Fuel Code                       52691 non-null  object
 16  Vehicle Recorded GVWR           52691 non-null  int64
 17  Vehicle Name                    52691 non-null  object
 18  Type                            52691 non-null  object
 19  Vehicle Category                52691 non-null  object
dtypes: int64(5), object(15)
```

Name: Ayush Maurya Class:D15C/34

dataset description

	ID	Vehicle Weight	Vehicle Year	Vehicle Declared Gross Weight	Vehicle Recorded GVWR
count	5.269100e+04	52691.000000	52691.000000	5.269100e+04	52691.000000
mean	1.541024e+06	662.548633	2021.632119	6.362825e+05	806.937466
std	1.039725e+06	2127.894845	2.471569	1.643281e+06	2300.669623
min	9.300000e+01	0.000000	1998.000000	0.000000e+00	0.000000
25%	5.487550e+05	0.000000	2021.000000	0.000000e+00	0.000000
50%	1.673766e+06	0.000000	2022.000000	0.000000e+00	0.000000
75%	2.613891e+06	0.000000	2023.000000	0.000000e+00	0.000000
max	2.934513e+06	80000.000000	2025.000000	7.214477e+06	80000.000000

3] This code checks each column in the DataFrame for missing values and sums them up.

To check null values and remove maximum missing values

```
df.isnull().sum()
```

	0
ID	0
Plate Type	0
Primary Customer City	34
Primary Customer State	34
Registration Start Date	0
Registration Expiration Date	0
Registration Usage	0
Vehicle Type	0
Vehicle Weight	0
Vehicle Year	0
Vehicle Make	0
Vehicle Model	0
Vehicle Body	0
Primary Color	0
Vehicle Declared Gross Weight	0
Fuel Code	0
Vehicle Recorded GVWR	0
Vehicle Name	0
Type	0

Name: Ayush Maurya Class:D15C/34

4] This code replaces zeros in the DataFrame with NA values, then removes all rows containing any missing data to create a cleaned DataFrame.
It prints the cleaned DataFrame and confirms no missing values remain by summing NA entries for each column.

```
df.replace(0, pd.NA, inplace=True)
df_cleaned = df.dropna()
print(df_cleaned.isna().sum())
```

```
Plate Type                  0
Primary Customer City       0
Registration Usage          0
Vehicle Type                0
Vehicle Weight              0
Primary Color               0
Vehicle Category            0
Z_Score                     0
Vehicle Weight Normalized   0
dtype: int64
```

Dropping unnecessary features

```
df.drop(['ID', 'Primary Customer State', 'Registration Start Date', 'Registration Expiration Date',
         'Vehicle Year', 'Vehicle Make', 'Vehicle Model', 'Vehicle Body',
         'Vehicle Declared Gross Weight', 'Fuel Code', 'Vehicle Recorded GVWR',
         'Vehicle Name', 'Type'], axis=1, inplace=True)
```

This code removes duplicate rows from the cleaned DataFrame.

```
[35] df = df_cleaned.drop_duplicates()
     df.head()
```

	Plate Type	Primary Customer City	Registration Usage	Vehicle Type	Vehicle Weight	Primary Color	Vehicle Category	Z_Score	Vehicle Weight Normalized
71	Passenger	SOUTHINGTON	Regular	SUV	5530	Gray	Light-Duty (Class 1-2)	-0.047744	0.376398
233	Passenger	GLASTONBURY	Regular	SUV	4774	Gray	Light-Duty (Class 1-2)	-0.533065	0.282484
276	Passenger	WALLINGFORD	Regular	Passenger	5200	Black	Light-Duty (Class 1-2)	-0.25959	0.335404
279	Passenger	WILLIMANTIC	Regular	Passenger	4200	Black	Light-Duty (Class 1-2)	-0.901549	0.21118
337	Passenger	WINDSOR	Regular	SUV	4800	Gray	Light-Duty (Class 1-2)	-0.516374	0.285714

```
df.shape
```

```
(678, 20)
```

Name: Ayush Maurya Class:D15C/34

5] This code creates dummy data out of plate type as commercial and passenger. This helps to convert categorical data to numerical data and helps in analysis in the algorithm

Creating a dummy variable for primary color

```
df_dummies = pd.get_dummies(df, columns=['Plate Type'], drop_first=True)
print(df_dummies)
```

```
      Primary Customer City Registration Usage Vehicle Type Vehicle Weight  \
71               SOUTHINGTON            Regular          SUV           5530
233              GLASTONBURY            Regular          SUV           4774
276              WALLINGFORD            Regular    Passenger           5200
279              WILLIMANTIC            Regular    Passenger           4200
337                  WINDSOR            Regular          SUV           4800
...                      ...                ...          ...            ...
52470               WESTON            Regular        Truck           8450
52609              PRESTON            Regular        Truck           8000
52648               GOSHEN            Regular          SUV           6450
52670            STRATFORD            Regular        Truck           8250
52684            WOODSTOCK            Regular    Passenger           5300

      Primary Color        Vehicle Category  Plate Type_Commercial  \
71             Gray  Light-Duty (Class 1-2)                  False
233            Gray  Light-Duty (Class 1-2)                  False
276           Black  Light-Duty (Class 1-2)                  False
279           Black  Light-Duty (Class 1-2)                  False
337            Gray  Light-Duty (Class 1-2)                  False
...             ...                   ...                    ...
52470         Black  Light-Duty (Class 1-2)                  False
52609          Gray  Light-Duty (Class 1-2)                  False
52648          Blue  Light-Duty (Class 1-2)                  False
52670          Gray  Light-Duty (Class 1-2)                  False
52684          Blue  Light-Duty (Class 1-2)                  False

       Plate Type_Passenger
71                     True
233                    True
276                    True
279                    True
337                    True
...                     ...
52470                  True
52609                  True
52648                  True
52670                  True
52684                  True

[678 rows x 8 columns]
```

Name: Ayush Maurya Class:D15C/34

6] The `head()` method to display the first ten rows of the DataFrame, to identify outliers manually we use the standardization approach (z score method). We find mean and standard deviation of the vehicle weight and calculate its z score; if its less than -3 or greater than 3 means its an outlier.

Detect outliers manually

```
df.head(10)
```

	Plate Type	Primary Customer City	Registration Usage	Vehicle Type	Vehicle Weight	Primary Color	Vehicle Category
71	Passenger	SOUTHINGTON	Regular	SUV	5530	Gray	Light-Duty (Class 1-2)
233	Passenger	GLASTONBURY	Regular	SUV	4774	Gray	Light-Duty (Class 1-2)
276	Passenger	WALLINGFORD	Regular	Passenger	5200	Black	Light-Duty (Class 1-2)
279	Passenger	WILLIMANTIC	Regular	Passenger	4200	Black	Light-Duty (Class 1-2)
337	Passenger	WINDSOR	Regular	SUV	4800	Gray	Light-Duty (Class 1-2)
416	Passenger	BRISTOL	Regular	SUV	5530	Silver	Light-Duty (Class 1-2)
420	Passenger	WEST SIMSBURY	Regular	Passenger	5600	Blue	Light-Duty (Class 1-2)
616	Passenger	NEW HAVEN	Regular	Passenger	3840	Black	Light-Duty (Class 1-2)
782	Passenger	GLASTONBURY	Regular	SUV	4800	Silver	Light-Duty (Class 1-2)
857	Passenger	GUILFORD	Regular	SUV	4709	Red	Light-Duty (Class 1-2)

Name: Ayush Maurya Class:D15C/34

```python
#By Z-score method
mean_vehicle_weight = df['Vehicle Weight'].mean()
std_vehicle_weight = df['Vehicle Weight'].std()

print(f"Mean of Vehicle Weight: {mean_vehicle_weight}")
print(f"Standard Deviation of Vehicle Weight: {std_vehicle_weight}")

# Calculate the Z-score for each vehicle weight
df['Z_Score'] = (df['Vehicle Weight'] - mean_vehicle_weight) / std_vehicle_weight

print(df[['Vehicle Weight', 'Z_Score']])

# Identify outliers based on the Z-score
outliers = df[df['Z_Score'].abs() > 3]
print(outliers)
```

```
Mean of Vehicle Weight: 5604.371681415929
Standard Deviation of Vehicle Weight: 1557.7315042063165
      Vehicle Weight   Z_Score
71              5530 -0.047744
233             4774 -0.533065
276             5200  -0.25959
279             4200 -0.901549
337             4800 -0.516374
...              ...       ...
52470           8450  1.826777
52609           8000  1.537896
52648           6450  0.542859
52670           8250  1.698385
52684           5300 -0.195394

[678 rows x 2 columns]
        Plate Type Primary Customer City Registration Usage Vehicle Type  \
15639  Combination             THOMASTON        Combination          SUV
23230  Combination             FAIRFIELD        Combination        Truck
32996  Combination           WILLIMANTIC        Combination          Van
50047  Combination               WINDSOR        Combination        Truck

      Vehicle Weight Primary Color         Vehicle Category   Z_Score
15639          10550         White   Light-Duty (Class 1-2)  3.174891
23230          10550         White   Light-Duty (Class 1-2)  3.174891
32996          10360        Orange  Medium-Duty (Class 3-6)  3.052919
50047          10500         White  Medium-Duty (Class 3-6)  3.142793
```

Name: Ayush Maurya Class:D15C/34

7] We normalize the data across the vehicle weights on a scale of 0 to 1.

Normalization

```
# Min-Max Normalization
min_vehicle_weight = df['Vehicle Weight'].min()
max_vehicle_weight = df['Vehicle Weight'].max()

# Apply normalization
df['Vehicle Weight Normalized'] = (df['Vehicle Weight'] - min_vehicle_weight) / (max_vehicle_weight - min_vehicle_weight)

print(df[['Vehicle Weight', 'Vehicle Weight Normalized']])
```

```
       Vehicle Weight  Vehicle Weight Normalized
71               5530                   0.376398
233              4774                   0.282484
276              5200                   0.335404
279              4200                    0.21118
337              4800                   0.285714
...               ...                        ...
52470            8450                    0.73913
52609            8000                    0.68323
52648            6450                   0.490683
52670            8250                   0.714286
52684            5300                   0.347826

[678 rows x 2 columns]
```

8] This is our normalized data with respect to vehicle weight and can be used to analyse the vehicle weight distribution across its type and color.

```
[38] print("The first 10 rows of normalized data are :")
     df.head(10)
```

The first 10 rows of normalized data are :

	Plate Type	Primary Customer City	Registration Usage	Vehicle Type	Vehicle Weight	Primary Color	Vehicle Category	Z_Score	Vehicle Weight Normalized
71	Passenger	SOUTHINGTON	Regular	SUV	5530	Gray	Light-Duty (Class 1-2)	-0.047744	0.376398
233	Passenger	GLASTONBURY	Regular	SUV	4774	Gray	Light-Duty (Class 1-2)	-0.533065	0.282484
276	Passenger	WALLINGFORD	Regular	Passenger	5200	Black	Light-Duty (Class 1-2)	-0.25959	0.335404
279	Passenger	WILLIMANTIC	Regular	Passenger	4200	Black	Light-Duty (Class 1-2)	-0.901549	0.21118
337	Passenger	WINDSOR	Regular	SUV	4800	Gray	Light-Duty (Class 1-2)	-0.516374	0.285714
416	Passenger	BRISTOL	Regular	SUV	5530	Silver	Light-Duty (Class 1-2)	-0.047744	0.376398
420	Passenger	WEST SIMSBURY	Regular	Passenger	5600	Blue	Light-Duty (Class 1-2)	-0.002806	0.385093
616	Passenger	NEW HAVEN	Regular	Passenger	3840	Black	Light-Duty (Class 1-2)	-1.132655	0.16646
782	Passenger	GLASTONBURY	Regular	SUV	4800	Silver	Light-Duty (Class 1-2)	-0.516374	0.285714
857	Passenger	GUILFORD	Regular	SUV	4709	Red	Light-Duty (Class 1-2)	-0.574792	0.27441

Conclusion:

Thus we have pre-processed our dataset by various techniques mentioned and can be used for analysis and trained under algorithms for predictions.

Experiment 2

Name: Ayush Maurya Class:D15C/34

Experiment 2

Aim: Data Visualization / Exploratory Data Analysis using Matplotlib and Seaborn

Introduction

Exploratory Data Analysis (EDA) is a crucial step in the data analysis pipeline. It involves visually and statistically exploring the data to gain insights and understand its underlying patterns, distributions, and relationships. In this section, we will use Matplotlib and Seaborn, two popular Python libraries, to create visualizations that help in uncovering these insights.

Matplotlib: A comprehensive library for creating static, animated, and interactive visualizations in Python.

Seaborn: Built on top of Matplotlib, Seaborn provides a high-level interface for drawing attractive and informative statistical graphics.

The primary objective of this analysis is to explore and visualize key patterns in the vehicle dataset, focusing on understanding relationships between different attributes and identifying significant trends.

Steps for Data Visualization and EDA

1) **Data Loading and Preprocessing:**

    - Load the dataset into a pandas DataFrame.
    - Perform initial data cleaning (handle missing values, incorrect data types, etc.).
    - Convert any necessary columns into the appropriate data types (e.g., dates, categorical variables).

2) **Summary Statistics:**

    - Use methods like df.describe() and df.info() to understand the basic statistics and data structure.
    - Check for missing values and visualize their distribution.

3) **Univariate Analysis:** This analysis focuses on visualizing individual variables to understand their distribution.

    - Histograms: Use Matplotlib and Seaborn's histplot() or hist() to display the distribution of numerical features like Vehicle Year, Vehicle Weight, and Registration Start Date.

- Box Plots: Use boxplot() from Seaborn to detect outliers in numerical variables such as Vehicle Weight and Vehicle Declared Gross Weight.

- Count Plots: For categorical features like Vehicle Type, Fuel Code, or Vehicle Category, use countplot() from Seaborn to visualize the distribution of each category.

4) **Bivariate Analysis:** This analysis explores the relationship between two variables.

    - Scatter Plots: Use scatterplot() from Seaborn to examine relationships between two continuous variables, such as Vehicle Weight vs. Vehicle Year, or Vehicle Recorded GVWR vs. Vehicle Declared Gross Weight.

    - Correlation Heatmap: Use heatmap() from Seaborn to visualize correlations between numerical variables. This can highlight which attributes have strong positive or negative relationships.

    - Pair Plots: For multiple continuous variables, use Seaborn's pairplot() to show pairwise relationships in the dataset and help identify correlations between them.

5) **Multivariate Analysis:** Analyze interactions between more than two variables at once.

    - Facet Grids: Use FacetGrid() in Seaborn to show how the relationships between two variables differ across the categories of a third variable.

    - Violin Plots: A combination of box plot and kernel density estimate, useful for understanding the distribution and density of a continuous variable against a categorical variable (e.g., Vehicle Type vs. Vehicle Weight).

6) **Categorical Variable Analysis:** Explore categorical variables to find trends or distributions.

    - Bar Plots: Use barplot() to compare the average of a numerical variable across categories, such as comparing the average Vehicle Weight for different Vehicle Types.

- Pie Charts: For categorical features like Vehicle Make or Fuel Code, use Matplotlib's pie() function to create visual representations of the proportions.

7) **Time Series Analysis (if applicable):** If there is any time-related data, such as Registration Start Date, we can visualize trends over time.

    - Line Plots: Use lineplot() from Seaborn to examine trends over time, such as the number of vehicles registered each year or the distribution of Vehicle Year over time.

    - Time-based Aggregation: Group the data by year or month to visualize trends and patterns in registrations, using groupby() in pandas followed by a line plot.

General Syntax in Python for Data Visualization

Python libraries like Matplotlib and Seaborn follow a general syntax for creating visualizations:

1. **Import the library**: Import the required libraries (e.g., `import matplotlib.pyplot as plt`).
2. **Prepare the data**: Use Pandas to manipulate and prepare the data for visualization.
3. **Create the plot**: Use functions like `plot()`, `scatter()`, `boxplot()`, etc., to create the visualization.
4. **Customize the plot**: Add titles, labels, legends, and other customizations.
5. **Display the plot**: Use `plt.show()` to display the visualization.

Name: Ayush Maurya Class:D15C/34

1) Bar Graph and Contingency Table

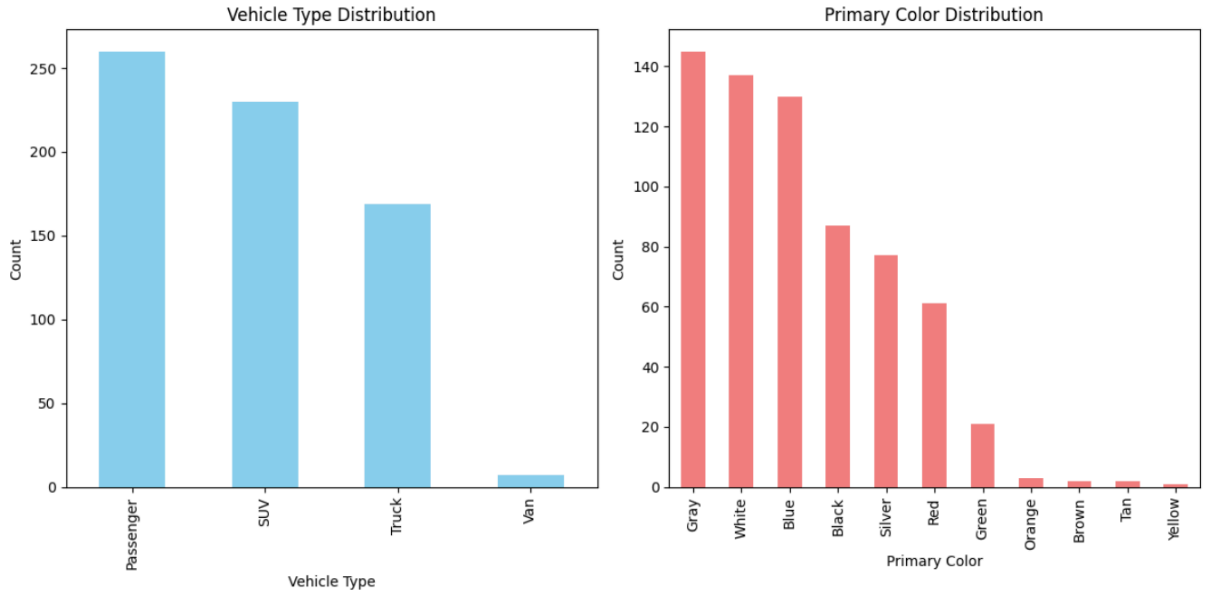
Bar Graph:

A bar graph is a chart that presents data with rectangular bars or columns. Each bar represents a category, and the height or length of the bar corresponds to the value or frequency of that category. Bar graphs are particularly useful for comparing different categories of a categorical variable.

Contingency Table:

A contingency table is a matrix that shows the frequency distribution of variables. It's especially useful for understanding the relationship between two categorical variables.

```
import matplotlib.pyplot as plt

# Count the occurrences of each category in 'Vehicle Type' and 'Primary Color'
vehicle_type_counts = df['Vehicle Type'].value_counts()
color_counts = df['Primary Color'].value_counts()

# Create bar graph
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
vehicle_type_counts.plot(kind='bar', color='skyblue')
plt.title('Vehicle Type Distribution')
plt.xlabel('Vehicle Type')
plt.ylabel('Count')

plt.subplot(1, 2, 2)
color_counts.plot(kind='bar', color='lightcoral')
plt.title('Primary Color Distribution')
plt.xlabel('Primary Color')
plt.ylabel('Count')

plt.tight_layout()
plt.show()
```



Name: Ayush Maurya Class:D15C/34

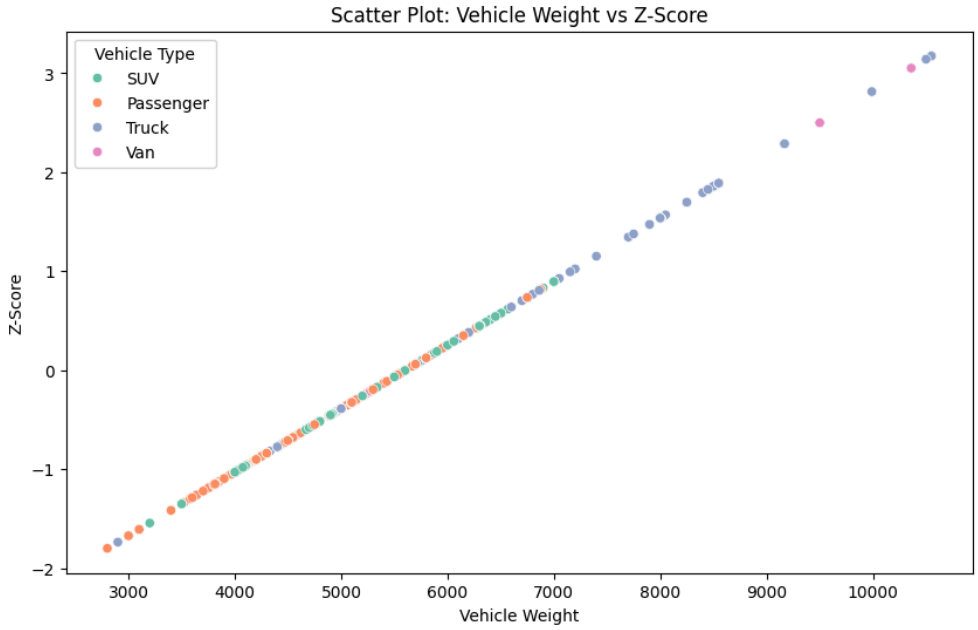
```
# Create a contingency table for 'Vehicle Type' and 'Primary Color'
contingency_table = pd.crosstab(df['Vehicle Type'], df['Primary Color'])
print(contingency_table)
```

```
Primary Color  Black  Blue  Brown  Gray  Green  Orange  Red  Silver  Tan  \
Vehicle Type
Passenger         38    44      1    62      3       1   32      30    1
SUV               27    51      1    46      4       1   18      24    1
Truck             22    34      0    37     14       0    9      23    0
Van                0     1      0     0      0       1    2       0    0

Primary Color  White  Yellow
Vehicle Type
Passenger         48       0
SUV               57       0
Truck             29       1
Van                3       0
```

2) Scatter Plot

A scatter plot is a type of data visualization used to display values for two continuous variables. It shows how one variable is affected by another, and it's useful for identifying trends, patterns, correlations, or outliers.

```
import seaborn as sns

plt.figure(figsize=(10, 6))
sns.scatterplot(x='Vehicle Weight', y='Z_Score', data=df, hue='Vehicle Type', palette='Set2')
plt.title('Scatter Plot: Vehicle Weight vs Z-Score')
plt.xlabel('Vehicle Weight')
plt.ylabel('Z-Score')
plt.show()
```

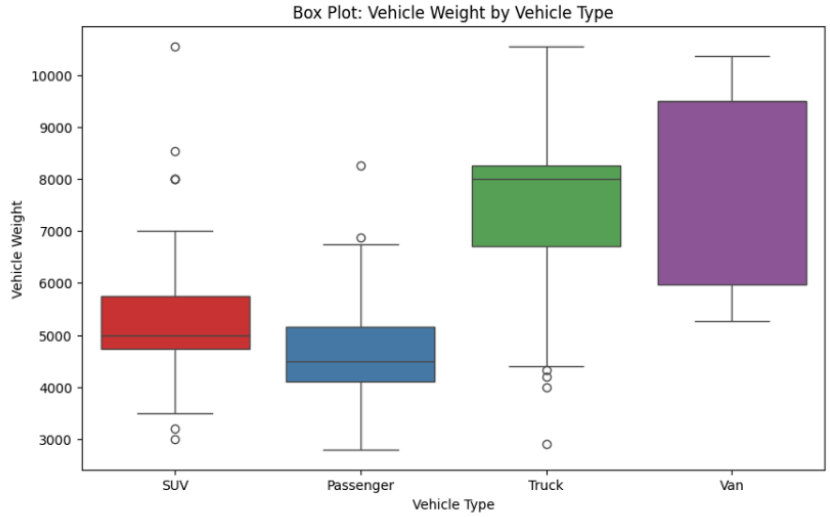


3) Box Plot

A box plot (also known as a box-and-whisker plot) is a graphical representation of the distribution of a dataset. It displays the median, upper and lower quartiles, and any potential outliers. Box plots are useful for comparing distributions across different categories, visualizing spread, and identifying skewness or outliers in the data.

```
plt.figure(figsize=(10, 6))
sns.boxplot(x='Vehicle Type', y='Vehicle Weight', data=df, palette='Set1')
plt.title('Box Plot: Vehicle Weight by Vehicle Type')
plt.xlabel('Vehicle Type')
plt.ylabel('Vehicle Weight')
plt.show()
```

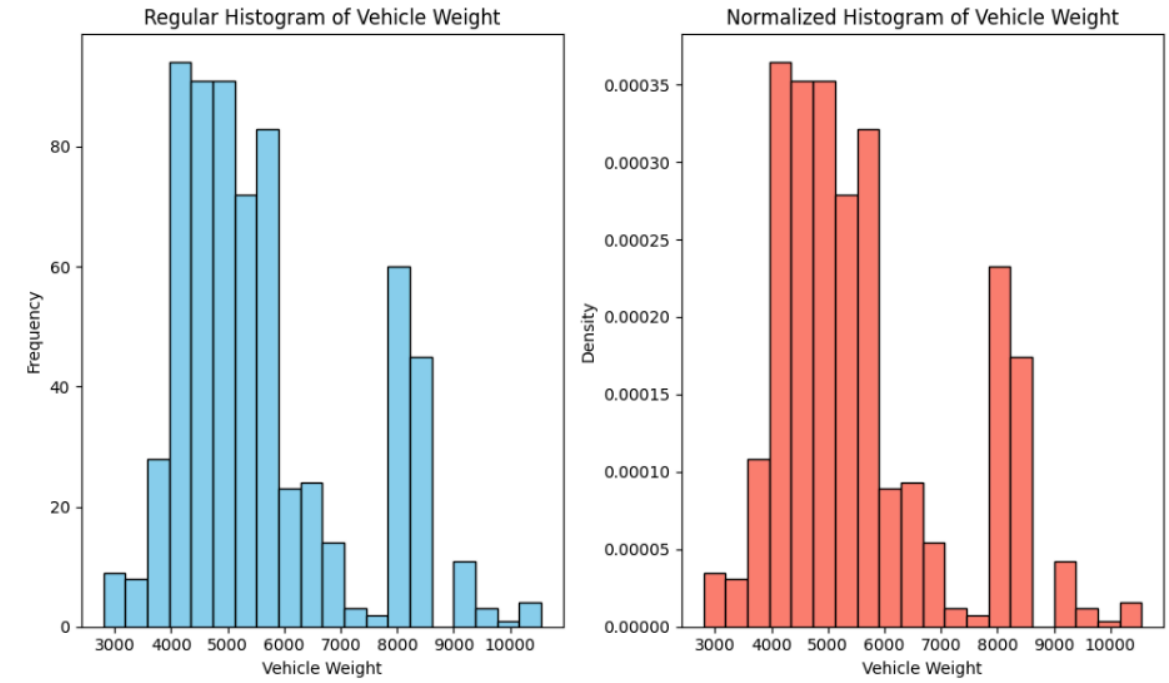
```
<ipython-input-42-dd532d0fa891>:2: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

  sns.boxplot(x='Vehicle Type', y='Vehicle Weight', data=df, palette='Set1')
```



4) Histogram

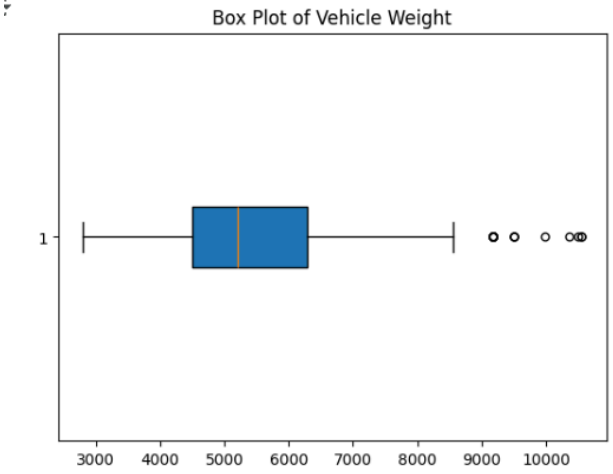
A histogram is a type of graph that is used to represent the frequency distribution of a dataset. It displays the number of occurrences (or frequency) of data points that fall within certain ranges (called bins). Histograms are particularly useful for understanding the distribution of continuous data, such as the spread, skewness, and presence of outliers.

```
# Plotting the Regular Histogram
plt.figure(figsize=(10, 6))

# Regular Histogram
plt.subplot(1, 2, 1)
plt.hist(df['Vehicle Weight'], bins=20, color='skyblue', edgecolor='black')
plt.title('Regular Histogram of Vehicle Weight')
plt.xlabel('Vehicle Weight')
plt.ylabel('Frequency')

# Plotting the Normalized Histogram
plt.subplot(1, 2, 2)
plt.hist(df['Vehicle Weight'], bins=20, color='salmon', edgecolor='black', density=True)
plt.title('Normalized Histogram of Vehicle Weight')
plt.xlabel('Vehicle Weight')
plt.ylabel('Density')

# Show both histograms
plt.tight_layout()
plt.show()
```



Name: Ayush Maurya Class:D15C/34

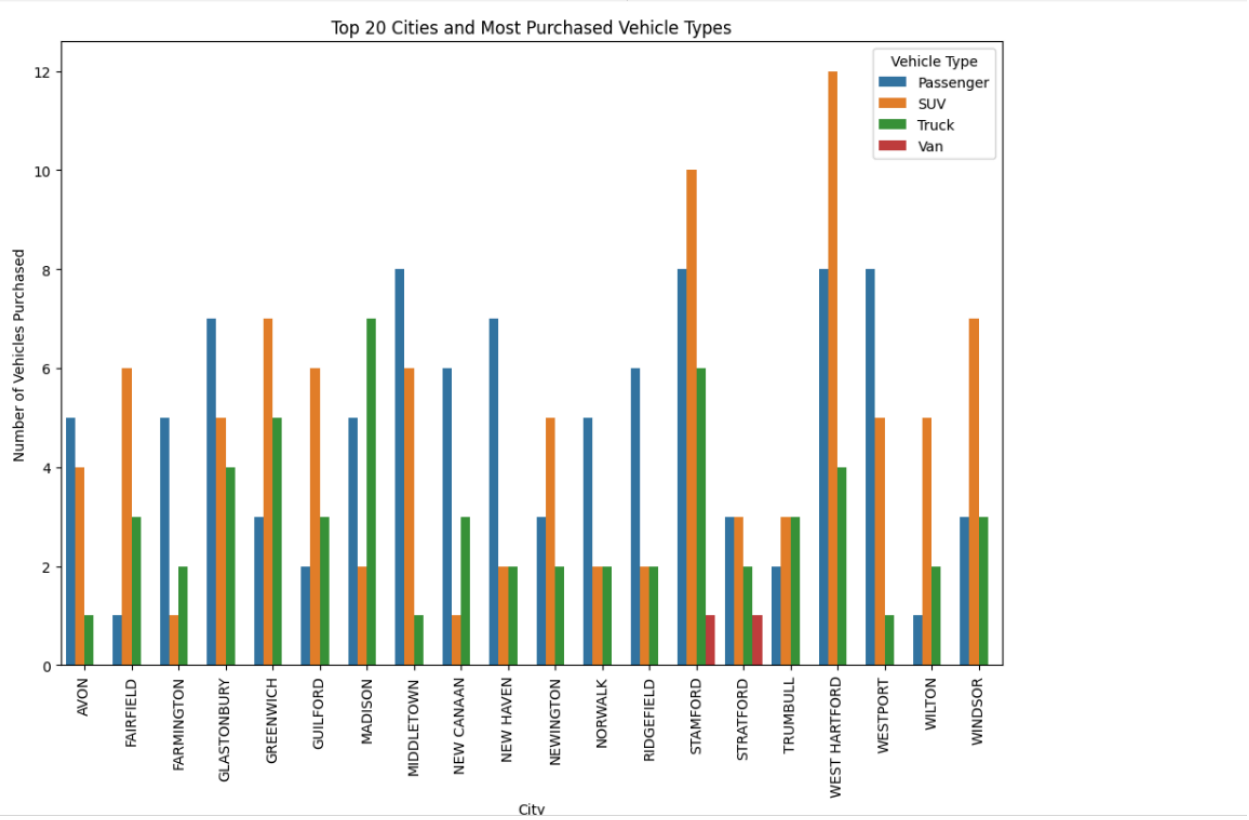
```
]
import matplotlib.pyplot as plt

# Plotting the box plot to identify outliers
plt.boxplot(df['Vehicle Weight'], vert=False, patch_artist=True, flierprops=dict(marker='o', color='red', markersize=5))
plt.title('Box Plot of Vehicle Weight')
plt.show()

# Calculate IQR and identify outliers
Q1, Q3 = df['Vehicle Weight'].quantile([0.25, 0.75])
IQR = Q3 - Q1
lower_bound, upper_bound = Q1 - 1.5 * IQR, Q3 + 1.5 * IQR

# Filter out outliers
df_no_outliers = df[(df['Vehicle Weight'] >= lower_bound) & (df['Vehicle Weight'] <= upper_bound)]

# Display results
print(f"Original shape: {df.shape}, New shape (no outliers): {df_no_outliers.shape}")
```




```
Original shape: (666, 10), New shape (no outliers): (647, 10)
```

```
# Group by city and vehicle type, then count occurrences
city_vehicle_counts = df.groupby(['Primary Customer City', 'Vehicle Type']).size().reset_index(name='Count')

# Get the top 20 cities based on the number of vehicle purchases
top_20_cities = city_vehicle_counts.groupby('Primary Customer City')['Count'].sum().nlargest(20).index
top_20_data = city_vehicle_counts[city_vehicle_counts['Primary Customer City'].isin(top_20_cities)]

# Plot the data
plt.figure(figsize=(12, 8))
sns.barplot(data=top_20_data, x='Primary Customer City', y='Count', hue='Vehicle Type', dodge=True)
plt.xticks(rotation=90)
plt.title('Top 20 Cities and Most Purchased Vehicle Types')
plt.xlabel('City')
plt.ylabel('Number of Vehicles Purchased')
plt.legend(title='Vehicle Type')
plt.show()
```

Name: Ayush Maurya Class:D15C/34



Conclusion:

Data visualization is a powerful technique for discovering insights, identifying anomalies, and understanding complex relationships in the dataset. By leveraging Matplotlib and Seaborn, we can gain a deeper understanding of how various vehicle attributes correlate with each other and influence key metrics, such as vehicle registration trends, category distribution, and pricing strategies. The visualizations generated in this EDA can help guide decision-making processes for inventory management, marketing strategies, and customer targeting.

Experiment 3

Name: Ayush Maurya Class:D15C/34

Experiment 3

Aim: To perform data modeling.

Theory:
Data partitioning is a crucial step in machine learning, where we divide the dataset into training and test sets. The training set, usually around 75% of the data, is used to develop the model, while the test set (25%) evaluates its performance. This ensures that the model generalizes well to new data rather than memorizing patterns from the training set. Proper partitioning prevents overfitting and improves real-world accuracy.

To verify that the partitioning is correct, we use visualization techniques such as bar graphs, histograms, and pie charts. These help confirm that the dataset maintains its proportions after splitting and that no class or feature distribution is unintentionally skewed. Counting the records in both sets ensures that the correct percentage of data has been allocated for training and testing.

A two-sample Z-test is a statistical method used to validate whether the training and test sets come from the same population. This test compares the means of numerical features in both sets to check if there is a significant difference. The Z-test is ideal for large datasets (sample size >30) and assumes normal distribution. If the p-value from the test is greater than 0.05, it indicates that the datasets are similar, confirming a good split. However, if the p-value is less than 0.05, it suggests that the partitioning may be biased, requiring a reassessment of the split method.

Overall, partitioning, visualization, and statistical validation together ensure a well-balanced dataset that helps in building an accurate and reliable machine learning model.

Steps:

1] Data PartitioningData partitioning is the process of dividing a dataset into two subsets: a training set and a test set. Typically, 75% of the data is used for training, where the model learns patterns, and 25% is used for testing to evaluate performance on unseen data. This division ensures that the model generalizes well and does not simply memorize the training examples. Proper partitioning helps in reducing overfitting and provides a fair evaluation of the model's effectiveness. The train_test_split function from sklearn.model_selection is commonly used to achieve this.

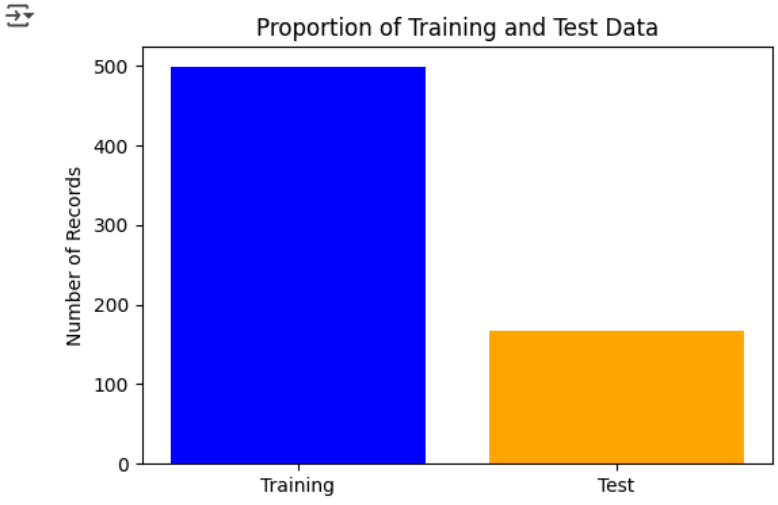
Name: Ayush Maurya Class:D15C/34

```
from sklearn.model_selection import train_test_split

# Partition the data: 75% for training and 25% for testing
train_df, test_df = train_test_split(df, test_size=0.25, random_state=42)

# Display the shape of the data sets
print(f"Training Data Shape: {train_df.shape}")
print(f"Test Data Shape: {test_df.shape}")
```

```
Training Data Shape: (499, 10)
Test Data Shape: (167, 10)
```

2] Visualization of the SplitAfter splitting the dataset, visualizing the distribution of training and test sets ensures that the split maintains the original dataset's characteristics. Bar graphs can be used to compare the number of records in both sets, while histograms and pie charts help check whether numerical and categorical feature distributions remain balanced. If a class or feature is disproportionately represented in either subset, the split may need adjustment. The matplotlib.pyplot library in Python helps create such visualizations to confirm a proper split.

```
# Bar plot for proportions of training and test data
plt.figure(figsize=(6, 4))
plt.bar(['Training', 'Test'], [len(train_df), len(test_df)], color=['blue', 'orange'])
plt.title('Proportion of Training and Test Data')
plt.ylabel('Number of Records')
plt.show()
```



3]Counting the number of records in both training and test sets ensures that the split has been performed correctly. The expected number of samples in each set is calculated using simple percentage formulas, such as Training Size = Total Data × 0.75 and Testing Size = Total Data × 0.25. By printing the lengths of the training and test sets after splitting, we can verify if the

proportions match the intended split. This step helps in detecting potential errors in dataset Partitioning.

```
[ ]  # Print the total number of records in the training data set
     print(f"Total records in the training data set: {len(train_df)}")
```

```
Total records in the training data set: 499
```

4]A two-sample Z-test is used to statistically verify whether the training and test sets come from the same distribution. It compares the means of numerical features in both subsets and checks for significant differences. If the p-value from the Z-test is greater than 0.05, the split is valid, meaning there is no significant difference between the two sets. However, if the p-value is below 0.05, the dataset may not be evenly distributed, requiring a reassessment of the split. The scipy.stats.ztest function in Python is commonly used to perform this validation.

Total records in the training data set: 499

```
from statsmodels.stats.weightstats import ztest

# Perform Z-test for 'Vehicle Weight' between training and test datasets
train_weight = train_df['Vehicle Weight']
test_weight = test_df['Vehicle Weight']

# Z-test for independent samples
z_stat, p_val = ztest(train_weight, test_weight)

# Display Z-test results
print(f"Z-statistic: {z_stat}")
print(f"P-value: {p_val}")

# Interpret the result
if p_val < 0.05:
    print("There is a significant difference between the training and test data for Vehicle Weight.")
else:
    print("There is no significant difference between the training and test data for Vehicle Weight.")
```

```
Z-statistic: -2.425627418979989
P-value: 0.015281950139779434
There is a significant difference between the training and test data for Vehicle Weight.
```

Conclusion: Proper dataset partitioning, visualization, and statistical validation ensure a balanced and unbiased split, leading to reliable model performance. This approach minimizes overfitting and improves the model's generalization to real-world data.

Experiment 4

Name: Ayush Maurya Class:D15C/34

Experiment 4

Aim: Implementation of Statistical Hypothesis Test using Scipy and Sci-kit learn.

Theory:

Correlation measures the statistical relationship between two variables. It indicates how strongly and in what direction one variable changes concerning another. The result is a correlation coefficient that ranges from -1 to +1.

- Strength: How closely the data points fit a trend (linear or monotonic).
- Direction: Whether an increase in one variable results in an increase or decrease in the other.

1. Positive Correlation (r>0)

- When one variable increases, the other also increases.
- Example:
    - The relationship between study time and exam scores.

2. Negative Correlation (r<0)

- When one variable increases, the other decreases.
- Example:
    - The relationship between exercise and body fat percentage.

3. No Correlation (r=0)

- No discernible relationship between the two variables.
- Example:
    - The relationship between shoe size and intelligence.

```
import pandas as pd
import scipy.stats as stats
from scipy.stats import chi2_contingency
```

Name: Ayush Maurya Class:D15C/34

Types of correlation tests:

1) Pearson's Correlation Coefficient

Pearson's correlation coefficient (rrr) is a statistical measure that quantifies the linear relationship between two continuous numerical variables. It ranges from −1 to +1, where +1 indicates a perfect positive correlation, −1 represents a perfect negative correlation, and 0 suggests no correlation. Pearson's correlation assumes that both variables are normally distributed and that the relationship between them is linear. It is widely used in fields such as finance, biology, and social sciences to analyze relationships like income vs. spending habits or height vs. weight.

If the absolute value of rrr is close to 1, it indicates a strong relationship, whereas values near 0 suggest a weak or no relationship. However, Pearson's correlation is sensitive to outliers, which can distort the results significantly.

$$r = \frac{\sum(X - \bar{X})(Y - \bar{Y})}{\sqrt{\sum(X - \bar{X})^2}\sqrt{\sum(Y - \bar{Y})^2}}$$

```python
import math

col1="Vehicle Year"
col2="Vehicle Weight"

X = df_cleaned[col1].values
Y = df_cleaned[col2].values
mean_X = sum(X) / len(X)
mean_Y = sum(Y) / len(Y)
X_diff = [xi - mean_X for xi in X]
Y_diff = [yi - mean_Y for yi in Y]
numerator = sum(xi * yi for xi, yi in zip(X_diff, Y_diff))
denominator_X = sum(xi ** 2 for xi in X_diff)
denominator_Y = sum(yi ** 2 for yi in Y_diff)
correlation = numerator / math.sqrt(denominator_X * denominator_Y)
print(f"Pearson Correlation between {col1} and {col2}: {correlation:.3f}")
```

```
Pearson Correlation between Vehicle Year and Vehicle Weight: 0.314
```

```python
# Selecting numerical columns for correlation tests
numerical_cols = ["Vehicle Year", "Vehicle Weight", "Vehicle Declared Gross Weight", "Vehicle Recorded GVWR"]
df_cleaned=df[numerical_cols].copy()
df_cleaned.replace(0, pd.NA, inplace=True)  # Convert 0s to NaN if needed
df_cleaned.dropna(inplace=True)  # Drop rows with new NaNs

print(df_cleaned.head())
# Pearson Correlation
pearson_corr = df_cleaned.corr(method="pearson")
print("Pearson Correlation Matrix:\n", pearson_corr)
```

2. Spearman's Rank Correlation

Spearman's Rank Correlation (ρ) is a non-parametric test that assesses the strength and direction of a monotonic relationship between two variables. Unlike Pearson's correlation, Spearman's method does not assume a linear relationship; instead, it is based on ranked values, making it suitable for ordinal data. The coefficient ranges from −1 to +1, where +1 indicates a perfect increasing rank agreement, −1 signifies a perfect decreasing rank agreement, and 0 means no correlation.

Spearman's correlation is ideal when dealing with non-normally distributed data or when the relationship is non-linear but still follows a consistent increasing or decreasing pattern. Common applications include ranking students based on performance or analyzing customer satisfaction scores vs. product ratings.

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)}$$

```
import numpy as np

df_manual = df[['Vehicle Year', 'Vehicle Weight']].dropna()

df_manual['Year Rank'] = df_manual['Vehicle Year'].rank(method='average')
df_manual['Weight Rank'] = df_manual['Vehicle Weight'].rank(method='average')

df_manual['d'] = df_manual['Year Rank'] - df_manual['Weight Rank']
df_manual['d^2'] = df_manual['d'] ** 2

n = len(df_manual)
spearman_manual = 1 - (6 * df_manual['d^2'].sum()) / (n * (n**2 - 1))

print(f"Manual Spearman's Correlation: {spearman_manual:.4f}")
```

```
Manual Spearman's Correlation: 0.3906
```

```
[23] #INBUILD
df_spearman = df[['Vehicle Year', 'Vehicle Weight']].dropna()

spearman_corr, p_value = stats.spearmanr(df_spearman['Vehicle Year'], df_spearman['Vehicle Weight'])

print(f"Spearman's Correlation: {spearman_corr:.4f}, p-value: {p_value:.4f}")
```

```
Spearman's Correlation: 0.0286, p-value: 0.0000
```

3. Kendall's Rank Correlation

Kendall's Tau (τ) is another non-parametric correlation test that measures the association between two ordinal variables. It is particularly useful for small datasets and is based on the number of concordant and discordant pairs in ranked data. A concordant pair means that the ranks of both variables move in the same direction, while a discordant pair moves in opposite directions. The correlation coefficient ranges from −1 (strong negative correlation) to +1 (strong positive correlation), with 0 indicating no association. Kendall's Tau is more robust than Spearman's correlation for small sample sizes and is commonly used in social sciences, psychology, and business research where ranking-based comparisons are necessary, such as employee performance evaluation vs. years of experience.

$$\tau = \frac{(C - D)}{\frac{1}{2}n(n-1)}$$

```
[24] kendall_corr, p_value = kendalltau(df_cleaned["Vehicle Year"], df_cleaned["Vehicle Weight"])
     print(f"Kendall's Correlation: {kendall_corr:.4f}, p-value: {p_value:.4f}")

     Kendall's Correlation: 0.2562, p-value: 0.0000
```

4. Chi-Squared Test for Categorical Data

The Chi-Squared (χ2) test is a statistical method used to determine the independence between two categorical variables. Unlike the other correlation tests, which are used for numerical or ordinal data, the Chi-Squared test assesses whether the distribution of one categorical variable is related to another. It compares the observed frequencies of data with the expected frequencies to see if any significant relationship exists. A high χ2value suggests a strong association, while a low value indicates independence. This test is widely applied in market research, healthcare, and sociology to analyze relationships such as gender vs. product preference, education level vs. voting behavior, or smoking habits vs. disease occurrence. The test assumes that each category has a sufficiently large sample size (typically, an expected frequency of at least 5 per category) to ensure reliability.

$$\chi^2 = \sum \frac{(O - E)^2}{E}$$

```
# Use correct column names from your dataset
category1 = df["Vehicle Make"]   # Manufacturer
category2 = df["Vehicle Model"]  # Model type

# Create a contingency table
contingency_table = pd.crosstab(category1, category2)

# Perform Chi-Square test
chi2_stat, p_value, dof, expected = stats.chi2_contingency(contingency_table)

# Print results
print(f"Chi-Square Statistic: {chi2_stat:.4f}")
print(f"Degrees of Freedom: {dof}")
print(f"P-value: {p_value:.4f}")
print("Expected Frequencies:\n", expected)
```

```
Chi-Square Statistic: 3468518.5933
Degrees of Freedom: 37812
P-value: 0.0000
Expected Frequencies:
 [[7.59142928e-05 2.56590310e-02 7.59142928e-05 ... 7.59142928e-05
  7.59142928e-05 7.59142928e-05]
 [1.93581447e-03 6.54305289e-01 1.93581447e-03 ... 1.93581447e-03
  1.93581447e-03 1.93581447e-03]
 [1.79157731e-02 6.05553131e+00 1.79157731e-02 ... 1.79157731e-02
  1.79157731e-02 1.79157731e-02]
 ...
 [7.40164354e-04 2.50175552e-01 7.40164354e-04 ... 7.40164354e-04
  7.40164354e-04 7.40164354e-04]
 [1.89785732e-05 6.41475774e-03 1.89785732e-05 ... 1.89785732e-05
  1.89785732e-05 1.89785732e-05]
 [1.89785732e-05 6.41475774e-03 1.89785732e-05 ... 1.89785732e-05
  1.89785732e-05 1.89785732e-05]]
```

Name: Ayush Maurya Class:D15C/34

Conclusion:

In this experiment, we analyzed relationships between variables using statistical hypothesis tests. Pearson's correlation measured linear relationships, while Spearman's and Kendall's rank correlations identified monotonic associations, making them suitable for non-linear data. The Chi-Squared test determined the independence of categorical variables. Our results highlighted the importance of choosing the right correlation method based on data type and distribution. Understanding these statistical tests is essential for accurate data analysis in various fields, including machine learning, business intelligence, and scientific research.

experiment5

Name: Ayush Maurya Class:D15C/34

Experiment 5

Aim:
Perform Regression Analysis using Scipy and Scikit-learn.

Problem Statement:

1. Perform Logistic Regression to determine the relationship between variables.
2. Apply a Regression Model technique to predict the data based on the given dataset.

Dataset Description:

The dataset contains more than 1 lakh instances, fulfilling the requirement for Big Data analysis. The following are the key features used in the regression models:

- **Cycle_Index**: Index of the cycle during battery charging/discharging.
- **Discharge Time (s)**: Time taken for battery discharge.
- **Decrement 3.6-3.4V (s)**: Time decrements between specific voltage levels.
- **Max. Voltage Discharge (V)**: Maximum voltage during discharge.
- **Min. Voltage Charge (V)**: Minimum voltage during charge.
- **Time at 4.15V (s)**: Time spent at a specific voltage level.
- **Time constant current (s)**: Duration of constant current phase.
- **Charging time (s)**: Total time taken for charging.
- **RUL (Remaining Useful Life)**: Predicted remaining life of the battery.

The dataset is preprocessed to remove missing values and scale numerical features before regression analysis.

Name: Ayush Maurya Class:D15C/34

Theory & Mathematical Background

1. Linear Regression:

Linear Regression is a fundamental supervised learning algorithm that models the relationship between a dependent variable (y) and one or more independent variables (X) by fitting a linear equation. It is widely used for predictive modeling where the goal is to establish a linear relationship between input and output variables.

Mathematical Representation:
The equation for multiple linear regression is:
y=β0+β1x1+β2x2+...+βnxny

$$y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n + \epsilon$$

Where:

- y = Dependent variable (Discharge Time (s))
- $X_1, X_2, ..., X_n$ = Independent variables
- β_0 = Intercept (constant term)
- $\beta_1, \beta_2, ..., \beta_n$ = Coefficients (weights assigned to independent variables)
- ϵ = Error term (random noise or variability unexplained by the model)

The coefficients (β) are estimated using the **Ordinary Least Squares (OLS) method**, which minimizes the sum of squared residuals:

$$\min \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

Name: Ayush Maurya Class:D15C/34

2. Logistic Regression:

Logistic Regression is used when the dependent variable is binary (classification problem). Instead of predicting a continuous value, it predicts the probability that a given input belongs to a specific category (0 or 1).

Mathematical Representation:

Instead of a linear function, logistic regression uses the sigmoid function to map the predictions between 0 and 1:

$$P(Y=1|X) = \frac{1}{1+e^{-(\beta_0+\beta_1 X_1+...+\beta_n X_n)}}$$

Where:

- $P(Y=1|X)$ is the probability of the output being class 1
- $\beta_0, \beta_1, ..., \beta_n$ are the model parameters
- e is the mathematical constant (~2.718)

Taking the **logit transformation**, we obtain the following linear equation:

$$log\left(\frac{P}{1-P}\right) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n$$

The model is trained using **Maximum Likelihood Estimation (MLE)**, which finds the parameters that maximize the likelihood of the observed data.

Name: Ayush Maurya Class:D15C/34

Comparison of Linear & Logistic Regression:

Feature	Linear Regression	Logistic Regression
Output Type	Continuous (real numbers)	Probability (0 to 1)
Used for	Regression (Prediction of values)	Classification (Binary categories)
Model Type	Linear	Non-linear (Sigmoid)
Loss Function	Mean Squared Error (MSE)	Log Loss (Cross-Entropy)
Optimization	Ordinary Least Squares (OLS)	Maximum Likelihood Estimation (MLE)

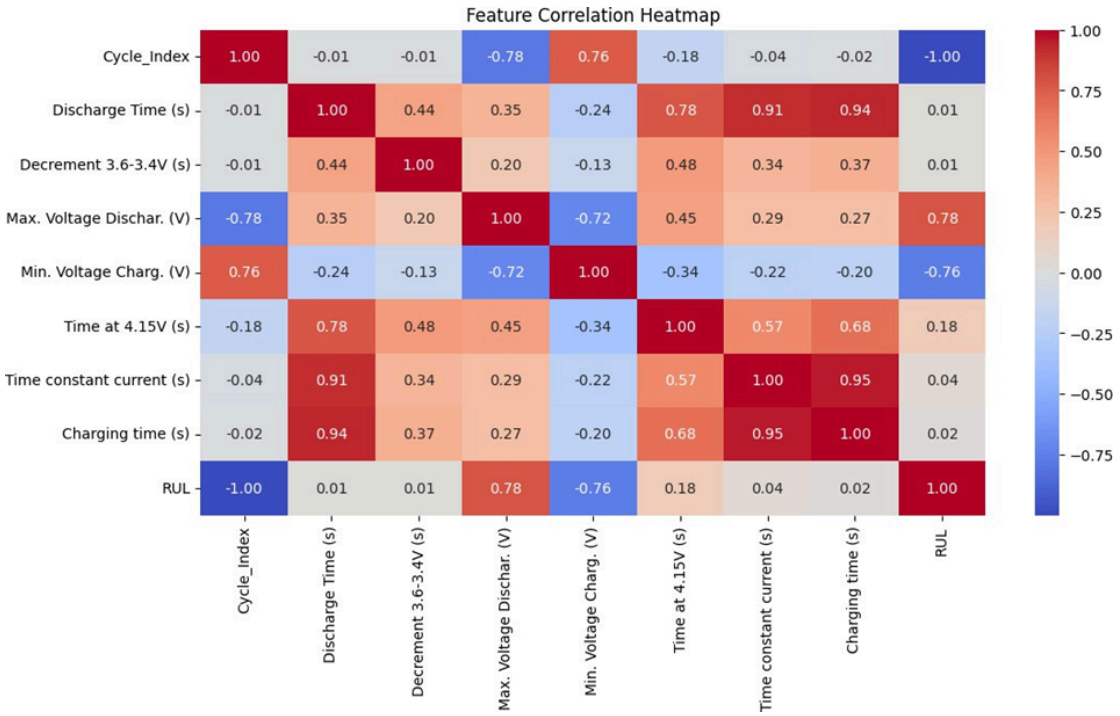
Output:

1. **Correlation Heatmap**:
   The heatmap shows correlations between features, highlighting strong positive relationships (e.g., Charging Time & Discharge Time: 0.94) and negative correlations (e.g., Cycle Index & RUL: -1.00). It confirms that battery aging affects discharge characteristics and RUL.

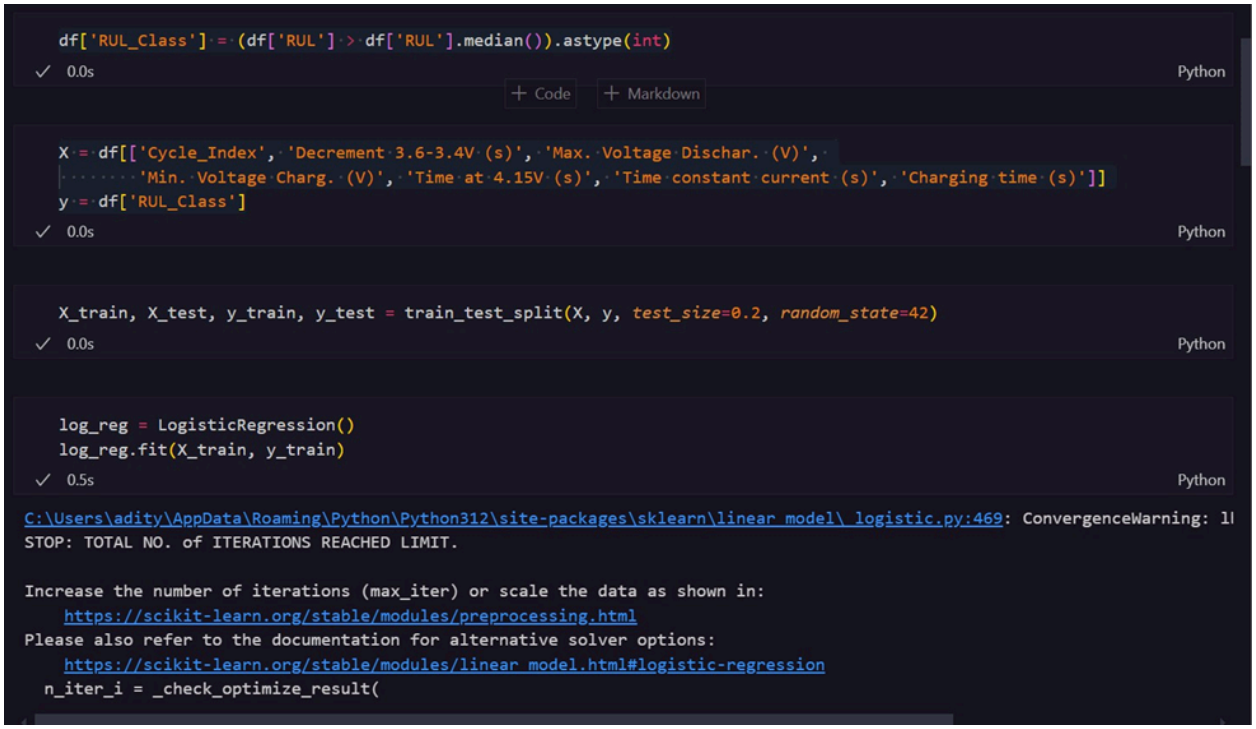
Name: Ayush Maurya Class:D15C/34

```
print("Missing Values:\n", df.isnull().sum())

df = df.dropna()

plt.figure(figsize=(12, 6))
sns.heatmap(df.corr(), annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Feature Correlation Heatmap")
plt.show()
```

```
Missing Values:
 Cycle_Index                 0
Discharge Time (s)           0
Decrement 3.6-3.4V (s)       0
Max. Voltage Dischar. (V)    0
Min. Voltage Charg. (V)      0
Time at 4.15V (s)            0
Time constant current (s)    0
Charging time (s)            0
RUL                          0
dtype: int64
```



2. **Binary Classification using Logistic Regression**:
    - A new target variable `RUL_Class` is created by comparing RUL to the median, turning it into a classification task.
    - Selected features include: Cycle_Index, voltage measurements, and charging times.
    - The dataset is split into 80% training and 20% testing using `train_test_split` with `random_state=42`.
    - A logistic regression model is trained, but a **ConvergenceWarning** is triggered, suggesting:
        - Increase `max_iter`
        - Use data standardization
        - Try alternative solvers like `saga` or `lbfgs`



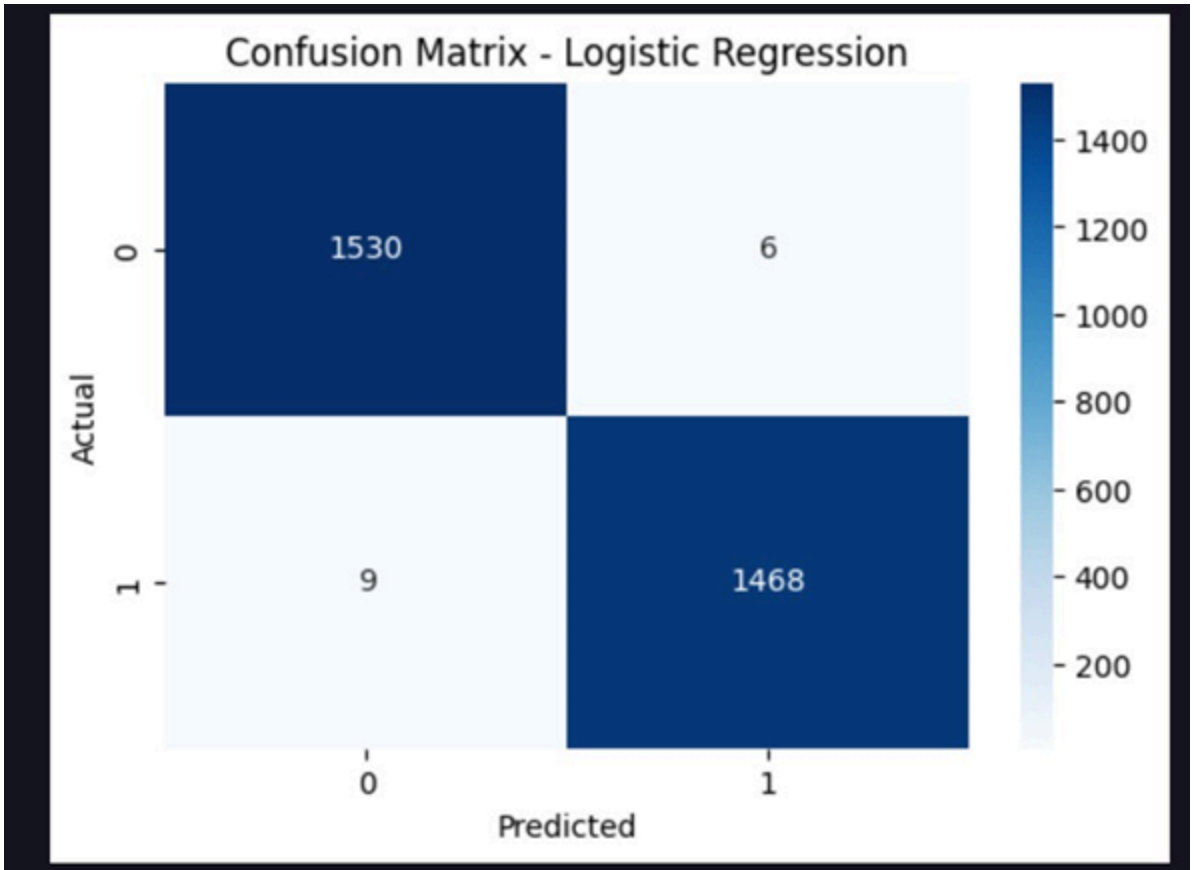
3. **Model Evaluation (Logistic Regression):**
    - Predictions are made using the test set (`y_pred`).
    - **Accuracy** is ~99.5% (very high).
    - A **confusion matrix** shows correct classifications and minimal misclassifications.
    - **Classification report** shows high precision, recall, and F1-scores.
    - However, the very low error rate may suggest overfitting due to class imbalance or high separability.

```
y_pred = log_reg.predict(X_test)
✓ 0.0s
```

```
accuracy = accuracy_score(y_test, y_pred)
print("Logistic Regression Accuracy:", accuracy)
✓ 0.0s
```

```
Logistic Regression Accuracy: 0.9950215731828742
```

```
plt.figure(figsize=(6, 4))
sns.heatmap(confusion_matrix(y_test, y_pred), annot=True, cmap="Blues", fmt="d")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix - Logistic Regression")
plt.show()

print("Classification Report:\n", classification_report(y_test, y_pred))

✓ 0.2s
```



Classification Report:

	precision	recall	f1-score	support
0	0.99	1.00	1.00	1536
1	1.00	0.99	0.99	1477
accuracy			1.00	3013
macro avg	1.00	1.00	1.00	3013
weighted avg	1.00	1.00	1.00	3013

---

4. **Linear Regression for Discharge Time Prediction**:
    - Model is trained to predict discharge time.
    - Dataset is split into train/test sets.

- Model performance evaluated using **MSE** and **R² score**.

---

5. **Model Performance (Linear Regression):**
   - **R² score** ≈ 0.93: Model explains 93% variance in discharge time.
   - **MSE** is large → potential outliers or large individual prediction errors.
   - **Scatter plot** shows upward trend in predictions but errors for high discharge times suggest:
     - Outliers or heteroscedasticity
     - Linear model limitations
     - Consider: feature scaling, polynomial regression, or models like Random Forest or Gradient Boosting.

Name: Ayush Maurya Class:D15C/34

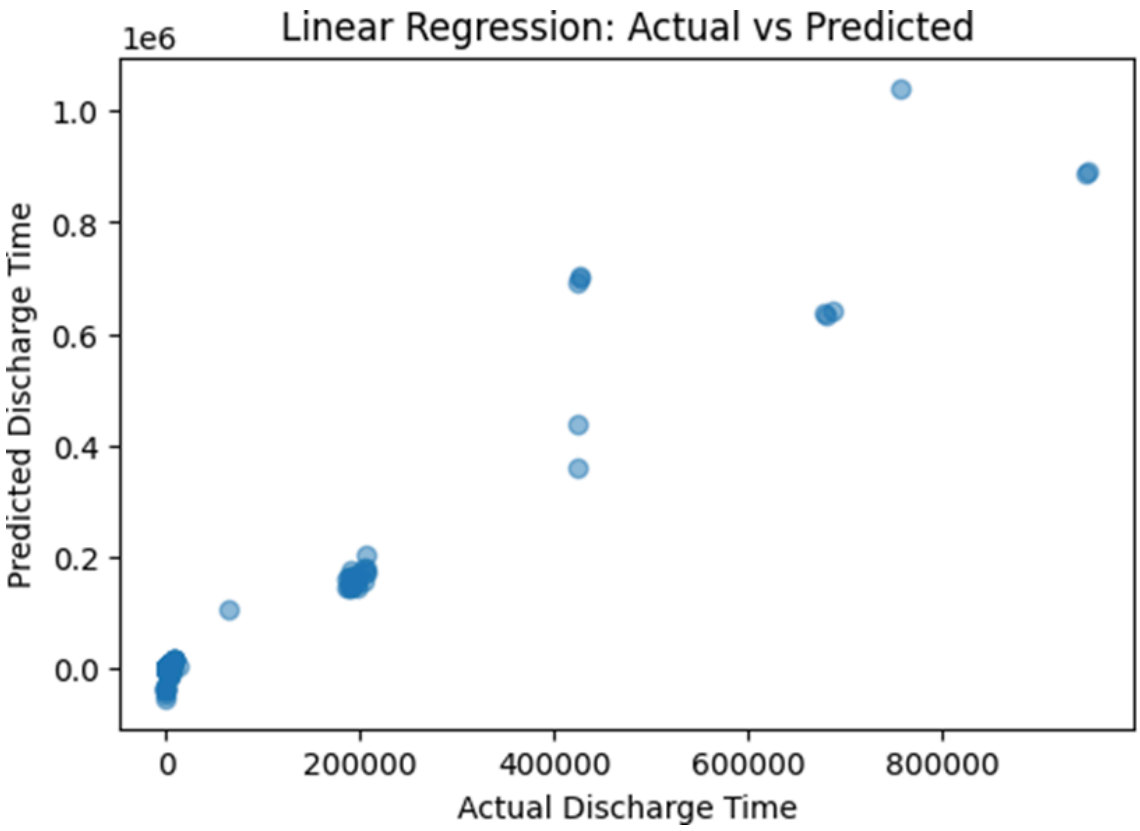
```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print("Linear Regression MSE:", mse)
print("Linear Regression R² Score:", r2)
```

✓ 0.0s

```
Linear Regression MSE: 127024271.8764475
Linear Regression R² Score: 0.9315065475881482
```

```
plt.figure(figsize=(6, 4))
plt.scatter(y_test, y_pred, alpha=0.5)
plt.xlabel("Actual Discharge Time")
plt.ylabel("Predicted Discharge Time")
plt.title("Linear Regression: Actual vs Predicted")
plt.show()
```

✓ 0.2s



Name: Ayush Maurya Class:D15C/34

Conclusion:

- **Linear Regression** is suitable for predicting continuous values like discharge time.
    - High R^2 (~0.93) but outliers/non-linearity affect accuracy.
- **Logistic Regression** is used for classification and performed well but may overfit.
    - Not ideal for continuous outputs.
- Model performance can be enhanced by:
    - Handling outliers
    - Feature engineering
    - Advanced models (Random Forest, ensemble methods)

Understanding the strengths and limitations of each technique aids in better machine learning predictions and decision-making.

experiment6

Experiment No 6

Aim: Classification modelling

a. Choose a classifier for a classification problem.

b. Evaluate the performance of the classifier.

- K-Nearest Neighbors (KNN)
- Decision Tree

Theory:

Classification Modeling: Theory & Techniques

Classification modeling is a type of supervised learning in machine learning where the goal is to predict the category or class of a given data point based on input features. The model is trained using labeled data (i.e., data where the output class is known).

Classification problems can be:

- **Binary Classification:** Two classes (e.g., spam vs. not spam).
- **Multiclass Classification:** More than two classes (e.g., classifying types of flowers).
- **Multi-label Classification:** Each sample can belong to multiple classes.

1. K-Nearest Neighbors (KNN)

K-Nearest Neighbors (KNN) is a simple, non-parametric classification algorithm based on proximity to labeled examples.

Working Principle:

1. Choose a value for K (number of neighbors).
2. Compute the distance between the new data point and all training samples.
3. Select the K nearest neighbors.
4. Assign the majority class among the K neighbors as the predicted class.

Common Distance Metrics:

- **Euclidean Distance:** $d=(\sum(xi-yi)^2)^{½}$ (Most commonly used)
- **Manhattan Distance:** $d=\sum|xi-yi|$
- **Minkowski Distance:** A generalization of Euclidean and Manhattan distances.

2. Naïve Bayes (NB)

Naïve Bayes is a probabilistic classifier based on **Bayes' Theorem**, assuming independence between predictors.

Bayes' Theorem:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Where:

- P(A|B) = Probability of class A given data B
- P(B|A) = Probability of data B given class A
- P(A) = Prior probability of class A
- P(B) = Prior probability of data B

Types of Naïve Bayes Classifiers:

1. **Gaussian Naïve Bayes:** Assumes normal distribution of features.
2. **Multinomial Naïve Bayes:** Used for text classification (e.g., spam detection).
3. **Bernoulli Naïve Bayes:** Used when features are binary (e.g., word presence in spam detection).

3. Support Vector Machines (SVMs)

SVM is a powerful classification algorithm that finds the optimal hyperplane to separate data points into different classes.

Working Principle:

1. **Hyperplane:** A decision boundary that maximizes the margin between two classes.
2. **Support Vectors:** Data points that lie closest to the hyperplane and influence its position.
3. **Kernel Trick:** SVM can handle non-linearly separable data using kernel functions to transform the input space.

Common Kernel Functions:

Linear Kernel: $K(x, y) = x^T y$

Polynomial Kernel: $K(x, y) = (x^T y + c)^d$

Radial Basis Function (RBF) Kernel: $K(x, y) = e^{-\gamma ||x-y||^2}$

Sigmoid Kernel: $K(x, y) = \tanh(\alpha x^T y + c)$

4. Decision Tree

A Decision Tree is a flowchart-like structure where internal nodes represent features, branches represent decisions, and leaves represent class labels.

Working Principle:

1. **Splitting Criteria:** Choose the best feature to split the data.
    - **Gini Index:** Measures impurity (Gini=$1-\sum pi^2$).
    - **Entropy (Information Gain):** Measures information gained from a split.
2. **Recursive Splitting:** Continue splitting nodes until a stopping criterion is met.
3. **Pruning:** Reduces overfitting by trimming branches.

Types of Decision Trees:

- **ID3 (Iterative Dichotomiser 3)** – Uses entropy for splitting.
- **C4.5 & C5.0** – Improvement over ID3 (handles continuous data).

- **CART (Classification and Regression Trees)** – Uses Gini Index.

Steps:

Step 1: Load the Dataset

The dataset is loaded from a CSV file using pandas and First 5 entries in the Dataset is shown using df.head() and Total rows and columns are printed using df.shape[n].

```
[4] import pandas as pd

file_path = "/content/drive/MyDrive/Semester 6/AIDS/AIDS Lab/Clean_Dataset_Categorized.csv"
df = pd.read_csv(file_path)

# Display dataset overview
print(f"Total Entries: {df.shape[0]}")
print(f"Total Columns: {df.shape[1]}")

Total Entries: 300153
Total Columns: 13
```

```
[6] df.head()
```

	Unnamed: 0	airline	flight	source_city	departure_time	stops	arrival_time	destination_city	class	duration	days_left	price	price_category
0	0	SpiceJet	SG-8709	Delhi	Evening	zero	Night	Mumbai	Economy	2.17	1	5953	Cheap
1	1	SpiceJet	SG-8157	Delhi	Early_Morning	zero	Morning	Mumbai	Economy	2.33	1	5953	Cheap
2	2	AirAsia	I5-764	Delhi	Early_Morning	zero	Early_Morning	Mumbai	Economy	2.17	1	5956	Cheap
3	3	Vistara	UK-995	Delhi	Morning	zero	Afternoon	Mumbai	Economy	2.25	1	5955	Cheap
4	4	Vistara	UK-963	Delhi	Morning	zero	Morning	Mumbai	Economy	2.33	1	5955	Cheap

Step 2: Data Preprocessing

1) Drop Unnecessary Columns

For the following experiment, the columns such as Unnamed and price are not required for classification. So to start preprocessing, we drop such columns using **df.drop(columns=[])** command.

```
[7]  # Drop 'Unnamed: 0' (index) and 'price' (to prevent data leakage)
     df.drop(columns=['Unnamed: 0', 'price'], inplace=True, errors='ignore')

     # Check updated dataset structure
     df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 300153 entries, 0 to 300152
Data columns (total 11 columns):
 #   Column            Non-Null Count   Dtype
---  ------            --------------   -----
 0   airline           300153 non-null  object
 1   flight            300153 non-null  object
 2   source_city       300153 non-null  object
 3   departure_time    300153 non-null  object
 4   stops             300153 non-null  object
 5   arrival_time      300153 non-null  object
 6   destination_city  300153 non-null  object
 7   class             300153 non-null  object
 8   duration          300153 non-null  float64
 9   days_left         300153 non-null  int64
 10  price_category    300153 non-null  object
dtypes: float64(1), int64(1), object(9)
memory usage: 25.2+ MB
```

2) Handling Missing Values

We fill up the missing values such that the numeric columns are filled with the median values, and the categorical columns are filled with mode of that column.

```
[8]  # Check for missing values
     missing_values = df.isnull().sum()
     print(missing_values[missing_values > 0])  # Show only columns with missing values

     # Fill missing values
     df.fillna(df.median(numeric_only=True), inplace=True)  # Fill numeric columns with median
     df.fillna(df.mode().iloc[0], inplace=True)  # Fill categorical columns with mode
```

```
Series([], dtype: int64)
```

3) Encode Categorical Variables

The categorical columns are encoded so that it becomes suitable for the algorithm to make it easier for the algorithm to make the classification.

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
y_encoded = le.fit_transform(y)  # Convert 'High', 'Medium', 'Low' to 0,1,2
```

9] ✓ 0.0s

Step 3: Split Into Train and Test

The dataset it then split into training and testing such that the models are trained on 80% of the dataset and 20% is used to test the models for their accuracy.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.2, random_state=42)
```

✓ 0.0s

+ Code + Markdown

Step 4: Train & Evaluate Classifiers

1)K-Nearest Neighbors (KNN)

From sklearn.neighbors library, we import the KNeighboursClassifier. We call this function and pass a parameter for the number of neighbours to be used. Here, we are passing 5 neighbours. After that, we fit the model with our training datasets (X and y) and create a variable to store the predicted values.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, accuracy_score

knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)

print("KNN Accuracy:", accuracy_score(y_test, y_pred_knn))
print(classification_report(y_test, y_pred_knn))

from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

cm = confusion_matrix(y_test, y_pred_knn)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=le.classes_)
disp.plot(cmap='Blues')
```

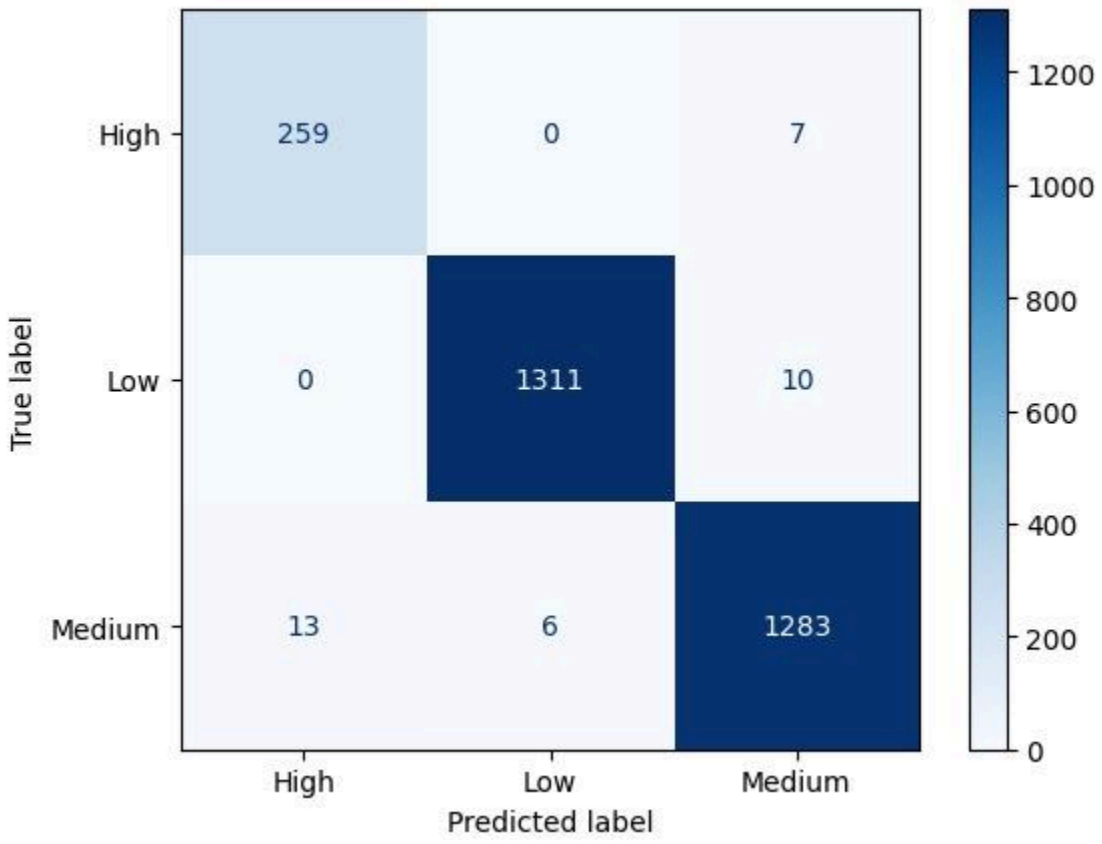
✓ 0.7s

```
KNN Accuracy: 0.9875389408099688
              precision    recall  f1-score   support

           0       0.95      0.97      0.96       266
           1       1.00      0.99      0.99      1321
           2       0.99      0.99      0.99      1302

    accuracy                           0.99      2889
   macro avg       0.98      0.98      0.98      2889
weighted avg       0.99      0.99      0.99      2889
```

I) Classification Report

K-Nearest Neighbors (KNN) classification model performed exceptionally well, achieving an overall accuracy of 98.75%. The confusion matrix shows that most predictions are correct, with very few misclassifications—mainly between the High and Medium classes, and slightly between Medium and Low. Precision, recall, and F1-scores across all three classes (High, Low, Medium) are consistently high, especially for the Low class where the model achieved nearly perfect scores. This indicates that the feature space is well-separated, and the KNN algorithm is effectively capturing the underlying structure of the data. The low number of off-diagonal values in the matrix shows that there are minimal overlaps between classes, making KNN a good fit for this dataset. The macro and weighted averages being close also suggest the model handles class imbalance well. Overall, the model demonstrates high reliability and robustness in classifying battery RUL categories.

II) Confusion Matrix


259
0
7
0
1311
10
13
6
1283
High
Low
Medium
True label
High
Low
Medium
Predicted label
1200
1000
800
600
400
200
0

2)Decision Tree

Before pre-pruning

```
from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier(criterion='entropy', random_state=42)
dt.fit(X_train, y_train)
y_pred_dt = dt.predict(X_test)

print("Decision Tree Accuracy:", accuracy_score(y_test, y_pred_dt))
print(classification_report(y_test, y_pred_dt))
```

✓ 0.0s

```
Decision Tree Accuracy: 0.9965385946694358
              precision    recall  f1-score   support

           0       0.99      1.00      0.99       266
           1       1.00      1.00      1.00      1321
           2       1.00      0.99      1.00      1302

    accuracy                           1.00      2889
   macro avg       0.99      1.00      0.99      2889
weighted avg       1.00      1.00      1.00      2889
```

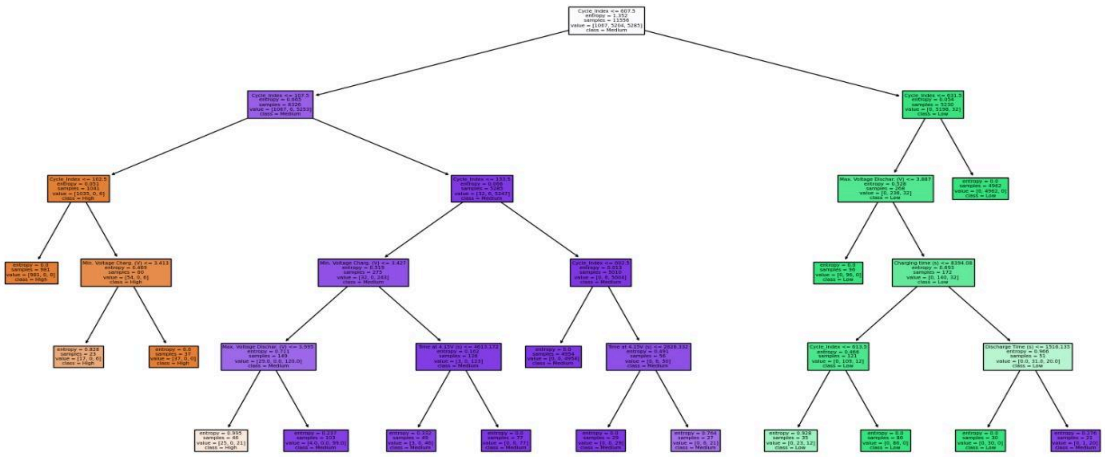
After pre-pruning

```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, accuracy_score

# Pruned Decision Tree
new_pruned_tree = DecisionTreeClassifier(
    criterion='entropy',
    max_depth=5,              # Limit the depth to avoid overfitting
    min_samples_leaf=15,        # Minimum samples required at a leaf node
    random_state=42
)

new_pruned_tree.fit(X_train, y_train)
y_pred_pruned = new_pruned_tree.predict(X_test)

# Evaluation
print("Pruned Decision Tree Accuracy:", accuracy_score(y_test, y_pred_pruned))
print(classification_report(y_test, y_pred_pruned))
```

✓ 0.1s

```
Pruned Decision Tree Accuracy: 0.9951540325372101
              precision    recall  f1-score   support

           0       0.98      1.00      0.99       266
           1       1.00      1.00      1.00      1321
           2       1.00      0.99      0.99      1302

    accuracy                           1.00      2889
   macro avg       0.99      1.00      0.99      2889
weighted avg       1.00      1.00      1.00      2889
```

The comparison between the original and pruned decision tree highlights the importance of pruning in machine learning. While the unpruned decision tree achieves a slightly higher accuracy of 99.65%, it risks overfitting, meaning it may perform very well on training data but fail to generalize to new unseen data. On the other hand, the pruned decision tree, with an accuracy of 99.51%, introduces a small drop in accuracy but enhances the model's generalization and interpretability. Pruning restricts the tree's growth (e.g., by setting max_depth=5 and min_samples_leaf=15), which reduces model complexity, avoids learning noise, and helps the model stay focused on meaningful patterns. In practical applications, especially with real-world, noisy data, a slightly less accurate but pruned model is usually more robust and reliable than a perfectly accurate yet overly complex model.



This is pruned decision tree

Conclusion:

Both the K-Nearest Neighbors (KNN) and Decision Tree (with and without pruning) models show strong classification performance on the battery RUL dataset. However, the key insight lies not just in accuracy but in model behavior, generalization, and practical usability. The unpruned decision tree offers near-perfect accuracy but risks overfitting, making it less reliable on unseen data. By applying pruning techniques, we slightly reduced the accuracy but gained a model that is more interpretable and generalizable—crucial for real-world deployment. In contrast, the KNN model, with its non-parametric nature and high precision/recall scores, proved to be both simple and powerful, especially when classes are well separated.

Through this experiment, I learned the importance of balancing accuracy with generalization. Pruning helps simplify complex models, preventing them from capturing noise, while models like KNN show how effectively simplicity and distance-based logic can perform when the feature space is clean and structured. Overall, this experiment reinforced the value of evaluating models not just by accuracy but also by their ability to generalize, their interpretability, and robustness to variation—key aspects in building reliable predictive systems for battery health and RUL classification.

experiment7

Name: Ayush Maurya Class:D15C/34

Experiment No 7

Aim: To implement different clustering algorithms.

Problem Statement:
a) Clustering algorithm for unsupervised classification (K-means, density based (DBSCAN), Hierarchical clustering)
b) Plot the cluster data and show mathematical steps.

Theory:

Clustering Algorithms for Unsupervised Classification

Clustering is an unsupervised machine learning technique used to group similar data points based on certain features. Below are three widely used clustering algorithms:

1. K-Means Clustering

K-Means is a centroid-based clustering algorithm that partitions data into k clusters.

Steps of K-Means Algorithm:

1. Choose the number of clusters k.
2. Initialize k cluster centroids randomly.
3. Assign each data point to the nearest centroid based on Euclidean distance.
4. Compute the new centroids as the mean of all points in each cluster.
5. Repeat steps 3 and 4 until centroids no longer change or a stopping criterion is met.

Mathematical Steps:

- Compute the distance between a point x_i and centroid C_j:

$$d(x_i, C_j) = \sqrt{\sum_{d=1}^{n} (x_{id} - C_{jd})^2}$$

- Update centroid:

$$C_j = \frac{1}{|S_j|} \sum_{x_i \in S_j} x_i$$

where S_j is the set of points assigned to cluster

2. DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

DBSCAN is a density-based clustering algorithm that groups points that are closely packed together while marking outliers as noise.

Steps of DBSCAN Algorithm:

1. Select a random point P and check if it has at least MinPts neighbors within radius ε.
2. If yes, create a new cluster and expand it by adding density-reachable points.
3. If no, mark P as noise.
4. Repeat until all points are processed.

Mathematical Concepts:

- A point P is a **core point** if it has at least MinPts neighbors within ε.
- A point Q is **density-reachable** from P if $d(P,Q) \leq \varepsilon$.
- A point is **noise** if it does not belong to any cluster.

3. Hierarchical Clustering

Hierarchical clustering builds a hierarchy of clusters using either **Agglomerative (bottom-up)** or **Divisive (top-down)** approaches.

Steps of Agglomerative Clustering (Bottom-Up Approach):

1. Treat each data point as its own cluster.
2. Compute the distance between all pairs of clusters.
3. Merge the two closest clusters.
4. Repeat steps 2-3 until one cluster remains.

Mathematical Concepts:

- **Single linkage:**

$$d(A,B) = \min_{a \in A, b \in B} d(a,b)$$

Name: Ayush Maurya Class:D15C/34

- **Complete linkage:**

$$d(A,B) = \max_{a \in A, b \in B} d(a,b)$$

- **Average linkage:**

$$d(A,B) = \frac{1}{|A||B|} \sum_{a \in A} \sum_{b \in B} d(a,b)$$

Steps :

Step 1: Load and Explore Data

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans, DBSCAN
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score
```

```
[ ] file_path="/content/Battery_RUL.csv"
    df=pd.read_csv(file_path)
```

```
[3] df.head()
```

	Cycle_Index	Discharge Time (s)	Decrement 3.6-3.4V (s)	Max. Voltage Dischar. (V)	Min. Voltage Charg. (V)	Time at 4.15V (s)	Time constant current (s)	Charging time (s)	RUL
0	1.0	2595.30	1151.488500	3.670	3.211	5460.001	6755.01	10777.82	1112
1	2.0	7408.64	1172.512500	4.246	3.220	5508.992	6762.02	10500.35	1111
2	3.0	7393.76	1112.992000	4.249	3.224	5508.993	6762.02	10420.38	1110
3	4.0	7385.50	1080.320667	4.250	3.225	5502.016	6762.02	10322.81	1109
4	6.0	65022.75	29813.487000	4.290	3.398	5480.992	53213.54	56699.65	1107

Next steps: Generate code with df | View recommended plots | New interactive sheet

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15064 entries, 0 to 15063
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Cycle_Index               15064 non-null  float64
 1   Discharge Time (s)        15064 non-null  float64
 2   Decrement 3.6-3.4V (s)    15064 non-null  float64
 3   Max. Voltage Dischar. (V) 15064 non-null  float64
 4   Min. Voltage Charg. (V)   15064 non-null  float64
 5   Time at 4.15V (s)         15064 non-null  float64
 6   Time constant current (s) 15064 non-null  float64
 7   Charging time (s)         15064 non-null  float64
 8   RUL                       15064 non-null  int64
dtypes: float64(8), int64(1)
memory usage: 1.0 MB
```

In this step, the cleaned **Battery RUL dataset** containing **15,064 entries** was loaded and explored. Each entry represents a battery usage cycle and includes features such as:

- **Cycle number** (charging/discharging cycle)
- **Discharge and charging times**
- **Voltage metrics** during charge and discharge
- **Time spent at critical voltage levels**
- **Constant current duration**
- and the target variable — **Remaining Useful Life (RUL)**.

Feature Extraction

```
[ ]  features = ["Discharge Time (s)", "Decrement 3.6-3.4V (s)", "Charging time (s)"]
     X = df[features]
```

A subset of features Discharge Time, Voltage Decrement (3.6–3.4V), and Charging Time was selected to form the input dataset X. These features were chosen to group battery cycles based on similar performance patterns using clustering algorithms.

Printing Missing Values

```
X.isnull().sum()
```

	0
Discharge Time (s)	0
Decrement 3.6-3.4V (s)	0
Charging time (s)	0

dtype: int64

No missing values were found across Features.

Step 2: Normalizing using StandardScaler

normalization using standardscaler

```
scaler=StandardScaler()
X_scaled=scaler.fit_transform(X)
```

To prepare the data for clustering, features were normalized using `StandardScaler`. This standardization scales the features to have a **mean of 0** and **standard deviation of 1**, ensuring all features contribute equally to distance-based clustering algorithms.

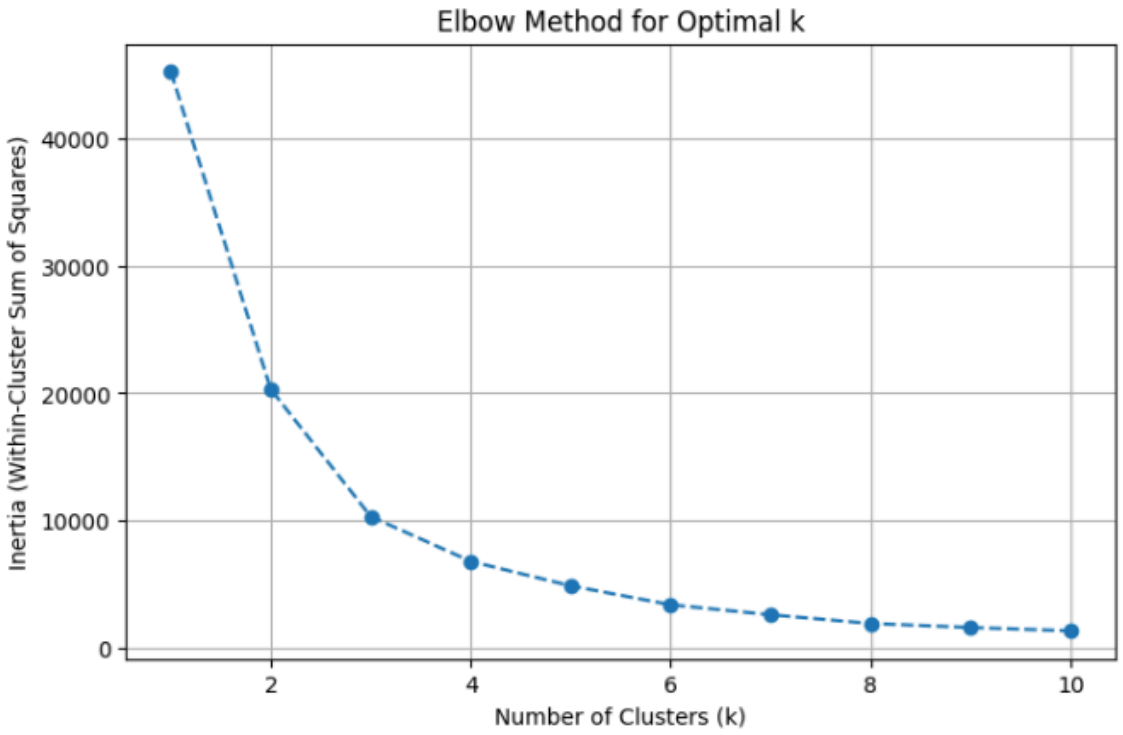
Step 3: K-Means Clustering

Elbow Curve

```
inertia =[]
k_values=range(1,11)
for k in k_values:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans.fit(X_scaled)
    inertia.append(kmeans.inertia_)

# Plot the Elbow Method graph
plt.figure(figsize=(8, 5))
plt.plot(k_values, inertia, marker='o', linestyle='--')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Inertia (Within-Cluster Sum of Squares)')
plt.title('Elbow Method for Optimal k')
plt.grid(True)
plt.show()
```

The Elbow Method was used to determine the optimal number of clusters (k) for K-Means. The model was trained for values of k from 1 to 10, and the inertia (sum of squared distances to the nearest cluster center) was recorded for each. A plot was generated to visualize the "elbow point," where adding more clusters yields diminishing returns helping to identify the best k. **(Here k =3, since we are classifying into healthy, moderately, worn-out)**

Name: Ayush Maurya Class:D15C/34



Before clustering we have applied PCA (Principal Component Analysis)

pca

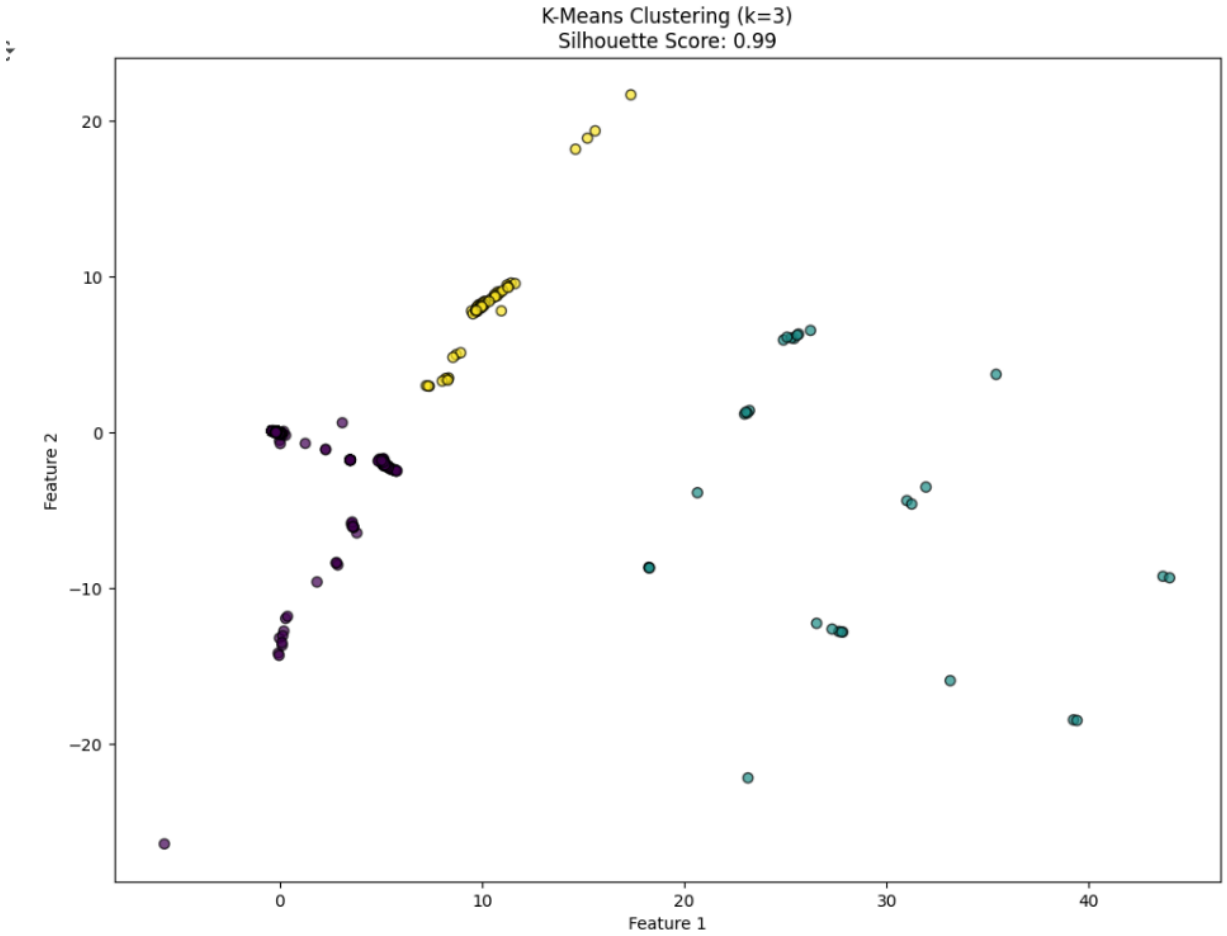
```
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)
```

To visualize clusters effectively, the feature space was reduced from 3 dimensions to 2 using Principal Component Analysis (PCA). This transformation retains the most significant variance while allowing easy plotting of clusters in 2D.

Code to Apply K-Means

```
optimal_k=3
kmeans= KMeans(n_clusters=optimal_k,random_state=42, n_init=10)
kmeans_labels = kmeans.fit_predict(X_scaled)

kmeans_silhouette = silhouette_score(X_scaled, kmeans_labels)

#plot graph
plt.figure(figsize=(12, 9))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=kmeans_labels, cmap='viridis', edgecolors='k', alpha=0.7)
plt.title(f'K-Means Clustering (k={optimal_k})\nSilhouette Score: {kmeans_silhouette:.2f}')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

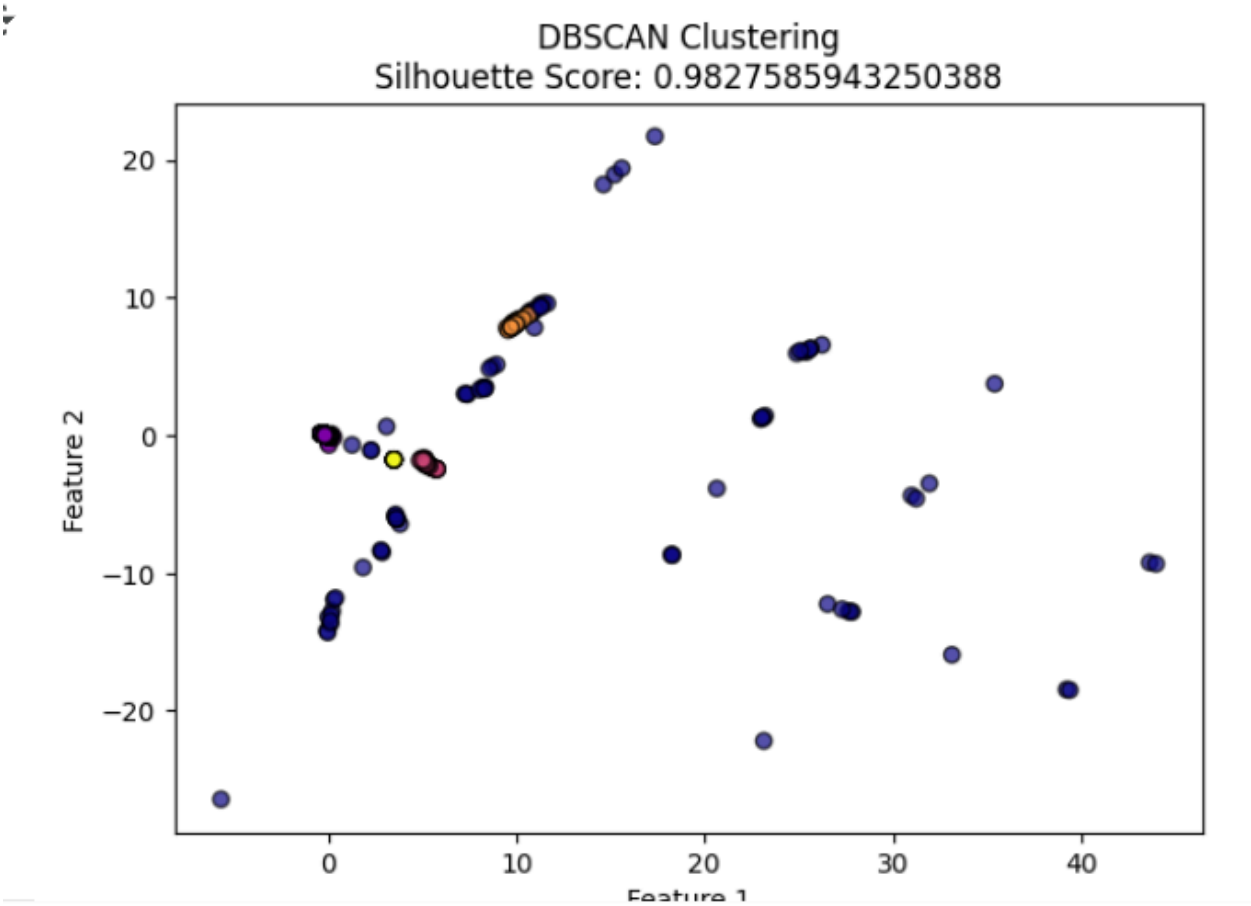
Name: Ayush Maurya Class:D15C/34



K-Means was applied with K=3 (from the elbow method) to segment Battery. The data points are grouped into three well-defined clusters, as indicated by the distinct color separation. The high Silhouette Score of 0.99 signifies excellent clustering quality, with strong intra-cluster similarity and clear inter-cluster separation. This suggests the underlying battery data forms three distinct behavioral or health patterns that are effectively captured through PCA and K-Means.

Step 4: DBSCAN Clustering

```
dbscan = DBSCAN(eps=0.5, min_samples=10)
dbscan_labels = dbscan.fit_predict(X_scaled)

dbscan_silhouette = silhouette_score(X_scaled, dbscan_labels) if len(set(dbscan_labels)) > 1 else "N/A"

plt.figure(figsize=(7, 5))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=dbscan_labels, cmap='plasma', edgecolors='k', alpha=0.7)
plt.title(f'DBSCAN Clustering\nSilhouette Score: {dbscan_silhouette}')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```



he DBSCAN clustering result demonstrates highly effective separation of data points into distinct clusters, as reflected by the impressive Silhouette Score of 0.9828. This score, which approaches the maximum value of 1, indicates that the clusters are both compact and well-separated, meaning the chosen eps (epsilon) and min_samples parameters are well-tuned for this dataset. The scatter plot, which visualizes the PCA-reduced data in two dimensions, clearly shows multiple dense groupings of points with minimal overlap, further confirming the clustering quality. Some points that do not belong to any dense region are likely treated as noise by DBSCAN, which helps maintain the integrity of clusters.

```
print(f'K-Means Silhouette Score: {kmeans_silhouette}')
print(f'DBSCAN Silhouette Score: {dbscan_silhouette}')
```

```
K-Means Silhouette Score: 0.9884934288567938
DBSCAN Silhouette Score: 0.9827585943250388
```

Both K-Means and DBSCAN achieved high Silhouette Scores, with K-Means slightly outperforming at 0.9885 compared to DBSCAN's 0.9828. While K-Means forms well-defined spherical clusters, DBSCAN effectively handles noise and non-linear cluster shapes. Despite the slight difference, both methods show excellent clustering performance on the dataset.

Conclusion:

In this project, we successfully implemented and analyzed **three clustering algorithms—K-Means**, and **DBSCAN** to perform **unsupervised classification**. Each algorithm was applied to a cleaned dataset and visualized using scatter plots and dendrograms.

- **K-Means** grouped data based on centroid distance, requiring the number of clusters as input.
- **DBSCAN** automatically identified clusters based on data density and detected outliers effectively.

These methods demonstrated different strengths: K-Means is simple and fast, Hierarchical provides insight into data hierarchy, and DBSCAN handles noise and varying densities well. Overall, clustering proved to be a powerful tool for discovering hidden patterns in unlabeled data.

experiment8

Experiment No 8

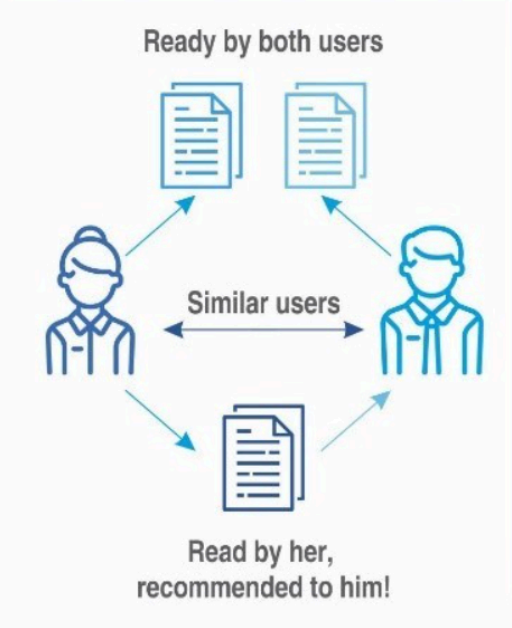
Aim: To implement a recommendation system on your dataset using the following machine learning techniques.

- Regression
- Classification
- Clustering
- Decision tree
- Anomaly detection
- Dimensionality Reduction
- Ensemble Methods

Theory: Recommendation systems predict user preferences to enhance engagement and are vital for content-rich platforms like Amazon and Netflix. They range from simple models to complex AI-driven systems using structured and unstructured data. This post introduces recommendation systems and outlines their business impact, forming the basis for my Master's Thesis work.

Types of Recommendation Sysytem

1) **Collaborative Filtering**
   Collaborative filtering generates recommendations by leveraging the behaviors, preferences, and interactions of multiple users. Instead of focusing solely on the actions of a specific user, it finds patterns across users to suggest relevant items. There are two common variations:

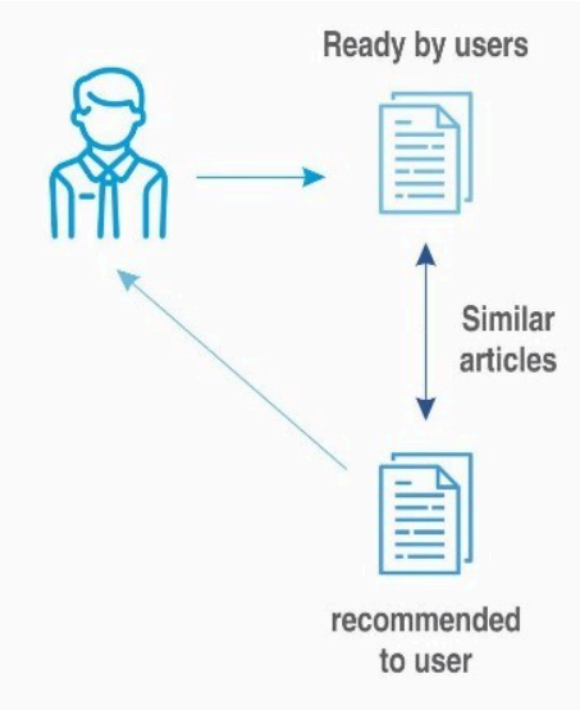


- **User Similarity:** This method groups users with similar behaviors and recommends items liked by others in the group. It's especially useful when a new user has limited data, as it can rely on similar users' preferences to generate suggestions.

- **Association-Based:** Often framed as "Users who viewed/bought X also viewed/bought Y," this technique analyzes purchasing patterns or viewing sequences. It's effective for recommending complementary items or content that aligns with a user's journey or current context.

2) **Content-Based**

Content-based systems focus on the user's individual history—such as items viewed, purchased, or rated—and recommend similar items. These systems become more accurate with increased user interaction and input. There are several nuanced techniques within this category:



- **Content Similarity:** Recommendations are based on matching metadata attributes (e.g., genre, price, author) between products. It's ideal for platforms with detailed item descriptions and limited user activity.

- **Topic Modeling:** This technique analyzes unstructured text (like news articles or reviews) to extract topics of interest, making it valuable for domains rich in textual data.

- **Popular Content Promotion:** Highlights widely appealing items based on features like popularity, recency, or price. It helps surface trending content and is commonly used when new content is frequently introduced.

The 6 steps to build a recommendation system:

1. Understand the Business

Start by clearly defining the goals of your recommendation system. These could include increasing sales, improving user engagement, reducing time to purchase, or promoting under-consumed content. Discuss with both data and business teams to understand what success looks like, where recommendations will appear in the user journey, and whether recommendations are the best solution compared to alternatives like curated lists. Consider what data is available, what product changes might be needed, and how to segment users based on similar preferences.

2. Get the Data
Gather as much relevant data as possible. Use both explicit feedback (like ratings and reviews) and implicit feedback (like browsing behavior, clicks, and watch time). Each has its strengths and weaknesses, so using both can provide a more accurate understanding of user preferences. For new or anonymous users, consider using demographic data or general behavior trends to make recommendations.

3. Explore, Clean, and Augment the Data
Explore and clean the data to ensure quality and relevance. Focus on recent interactions, as user preferences can change over time. Consider assigning more weight to newer data and possibly removing outdated interactions. Handle missing values carefully, especially in high-dimensional datasets where many features may be sparse, and look for opportunities to enhance the dataset with additional useful information.

4. Predict the Ranking
Use the prepared data to generate recommendations. A simple ranking system might be enough in some cases, but more sophisticated machine learning approaches can improve performance. You can use hybrid systems that combine different techniques, maintain multiple models in parallel, or apply machine learning to optimize which model is used. Tailor your recommendation approach depending on where the user is in their journey before or after they've interacted with content.

5. Visualize the Data
Visualization plays a key role in both understanding the dataset and communicating insights. During exploration, use visualizations to uncover trends and patterns. After deployment, visual dashboards can help business teams identify underperforming content, user behavior clusters, and opportunities for improvement. Good visualizations make large, complex datasets understandable and actionable.

6. Iterate and Deploy Models
Deploy the model into production to start impacting real users and business outcomes. Monitor its performance continuously and refine the model as more data becomes available. Set up feedback loops to understand whether recommendations are effective, and adapt the model as user preferences evolve. Recommendation systems are dynamic—they need ongoing updates, experimentation, and optimization to stay effective over time.

Implementation

Step 1: Load dataset and select Features

```
import pandas as pd

df = pd.read_csv("products.csv")

df['description'] = df['description'].fillna('')
df['ingredients'] = df['ingredients'].fillna('')

df['combined_text'] = df['ingredients'] + " " + df['description']
```

The code loads products.csv into a pandas DataFrame.
It fills missing values in the description and ingredients columns with empty strings.
Then, it creates a new column combined_text by combining ingredients and description for each row.

Step 2: Import and perform TF-IDF Vectorization

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(stop_words='english')

tfidf_matrix = vectorizer.fit_transform(df['combined_text'])
```

TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical measure used in natural language processing and information retrieval to evaluate the importance of a word in a document relative to a collection of documents (corpus).

TF-IDF combines two components: Term Frequency (TF) and Inverse Document Frequency (IDF).

Term Frequency (TF): Measures how often a word appears in a document. A higher frequency suggests greater importance. If a term appears frequently in a document, it is likely relevant to the document's content. **Formula**:

$$TF(t, d) = \frac{\text{Number of times term t appears in document d}}{\text{Total number of terms in document d}}$$

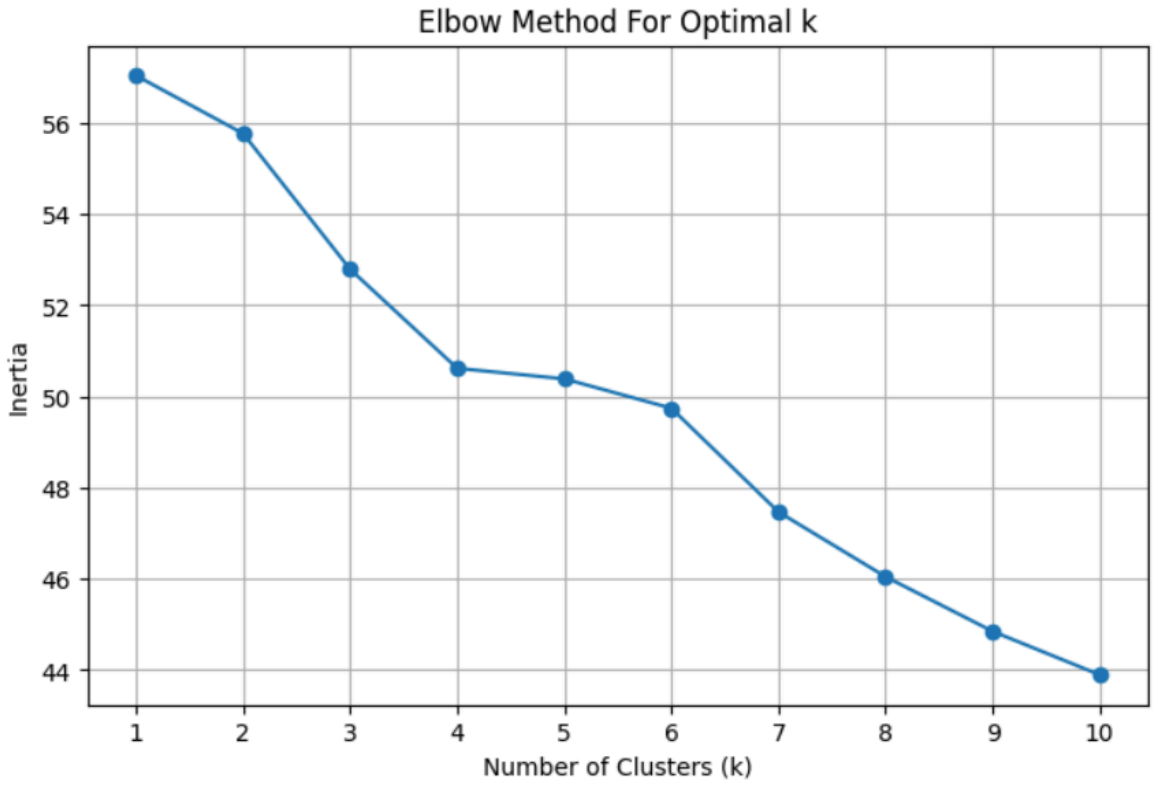
Inverse Document Frequency (IDF): Reduces the weight of common words across multiple documents while increasing the weight of rare words. If a term appears in fewer documents, it is more likely to be meaningful and specific. **Formula**:

$$IDF(t, D) = \log \frac{\text{Total number of documents in corpus D}}{\text{Number of documents containing term t}}$$

Step 3: Clustering

```
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

# Range of cluster numbers to try
cluster_range = range(1, 11)
inertias = []

# Calculate KMeans inertia for each cluster count
for k in cluster_range:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(tfidf_matrix)
    inertias.append(kmeans.inertia_)

plt.figure(figsize=(8, 5))
plt.plot(cluster_range, inertias, marker='o')
plt.title('Elbow Method For Optimal k')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Inertia')
plt.xticks(cluster_range)
plt.grid(True)
plt.show()
```

✓ 0.1s



Selected 4 as no of cluster with help of elbow curve

```
tfidf_matrix.shape
```
✓ 0.0s

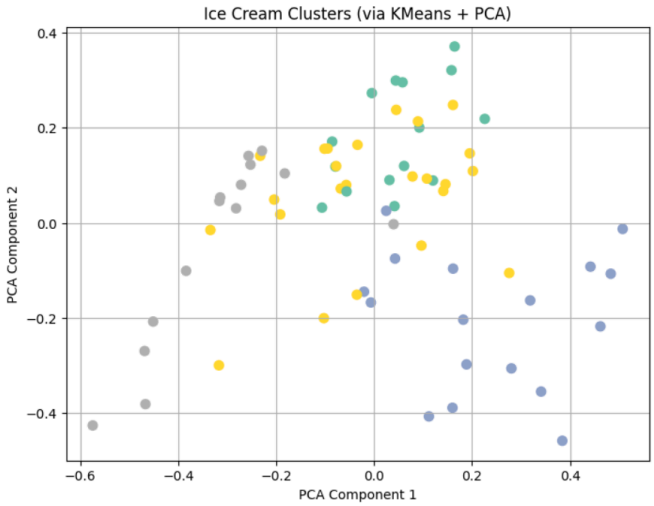
```
(70, 464)
```

```
from sklearn.cluster import KMeans

num_clusters = 4
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
df['cluster'] = kmeans.fit_predict(tfidf_matrix)
```
✓ 0.0s

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
reduced_data = pca.fit_transform(tfidf_matrix.toarray())
```
✓ 0.0s

It works by transforming high-dimensional data into a lower-dimensional space while maximizing the variance (or spread) of the data in the new space. This helps preserve the most important patterns and relationships in the data.

```
plt.figure(figsize=(8, 6))
scatter = plt.scatter(reduced_data[:, 0], reduced_data[:, 1], c=df['cluster'], cmap='Set2', s=50)
plt.title("Ice Cream Clusters (via KMeans + PCA)")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
plt.grid(True)

plt.show()
```



Step 4: Recommendation function

```
def recommend_by_cluster(user_input, top_n=5):
    user_input_tfidf = vectorizer.transform([user_input])

    user_cluster = kmeans.predict(user_input_tfidf)[0]

    cluster_df = df[df['cluster'] == user_cluster]

    # Compute similarity within the cluster
    cluster_tfidf = tfidf_matrix[cluster_df.index]
    similarities = cosine_similarity(user_input_tfidf, cluster_tfidf).flatten()

    # Sort and return top matches
    top_indices = similarities.argsort()[::-1][:top_n]
    recommendations = cluster_df.iloc[top_indices][['name', 'rating', 'rating_count']].copy()
    recommendations['similarity'] = similarities[top_indices]
    return recommendations
```

✓ 0.0s

In this code user inputs are classified and matched with relevant options. After transforming the input into numerical data (using TF-IDF), the system predicts the

cluster to which the input belongs. This clustering can represent groups of similar items, users, or behaviors in your experiment. Next, the cosine similarity calculation ensures that recommendations are highly relevant and tailored to the user's input, making the results more precise and impactful.

Step 5: Result

```
user_query = "chocolate "
results = recommend_by_cluster(user_query, top_n=10)
results
```

✓ 0.0s

	name	rating	rating_count	similarity
58	Vanilla Cookie Squares	4.3	32	0.452555
69	White Chocolate Raspberry Ice Cream Bar	3.9	11	0.447600
13	Chocolate Dark Chocolate Ice Cream Bar	4.5	22	0.427577
28	Dulce de Leche Cookie Squares	3.9	35	0.374704
39	Peanut Butter Cookie Squares	4.4	14	0.371822
42	Peanut Butter Chocolate Fudge Non-Dairy Bar	4.8	32	0.301602
14	Chocolate Fudge Non-Dairy Bar	5.0	22	0.300940
44	Peppermint Bark Ice Cream Bar	5.0	8	0.269398
32	Irish Cream Cookie Squares	4.9	14	0.252807
20	Coconut Caramel Dark Chocolate Non-Dairy Bar	4.6	31	0.236645

It starts by transforming the user's input (like "chocolate") into a **numerical vector** through TF-IDF. Then it predicts the **most relevant cluster** of data based on this input using the k-means clustering algorithm, which essentially groups similar items together beforehand.

Conclusion:

This experiment demonstrated the successful implementation of a recommendation system using various machine learning techniques. By applying TF-IDF and K-Means clustering, we grouped similar items and matched user input to relevant products using cosine similarity. The system effectively provided accurate and personalized recommendations, showcasing a practical approach to real-world recommendation engines.

experiment9

Name: Ayush Maurya Class:D15C/34

Experiment 9

Aim: To perform Exploratory Data Analysis using Apache Spark and Pandas.

Theory:

1. What is Apache Spark and how does it work?

Apache Spark is an open-source, distributed computing system designed for big data processing and analytics. It provides a unified analytics engine that supports in-memory computation, which significantly enhances processing speed compared to traditional disk-based engines like Hadoop MapReduce. Spark was developed at UC Berkeley in 2009 and later donated to the Apache Software Foundation.

Apache Spark works on a distributed architecture with a master-slave structure. The main components include:

1. Driver Program: Contains the main function and creates a SparkContext that coordinates the execution.
2. Cluster Manager: Allocates resources across applications (can be Spark's standalone manager, YARN, Mesos, or Kubernetes).
3. Executors: Processes that run computations and store data on worker nodes.
4. Resilient Distributed Dataset (RDD): The fundamental data structure in Spark, which is an immutable distributed collection of objects that can be processed in parallel.

Spark operations are executed using a directed acyclic graph (DAG) execution engine, which optimizes workflows. When operations are called, Spark builds a DAG of operations and executes them in an optimized manner. The framework's core abstraction is the RDD, which enables fault tolerance through lineage information that tracks how RDDs are derived from other datasets.

Spark's ecosystem includes several integrated libraries:

1. Spark SQL for structured data processing
2. MLlib for machine learning
3. GraphX for graph processing
4. Spark Streaming for real-time data processing

Key Features of Spark:

1. In-memory computation for increased speed.
2. Support for multiple languages: Python (PySpark), Scala, Java, R.
3. Components: Spark Core, Spark SQL, Spark Streaming, MLlib, GraphX.

Name: Ayush Maurya Class:D15C/34

Working:Spark runs on a cluster and distributes data across worker nodes.

1. It uses Resilient Distributed Datasets (RDDs) to process data in parallel.
2. The Driver Program coordinates the tasks and distributes work to Executors.
3. Spark uses DAG (Directed Acyclic Graph) for task execution.

2. How is data exploration done in Apache Spark? Explain steps.

Exploratory Data Analysis (EDA) in Apache Spark follows a structured approach that leverages Spark's distributed computing capabilities to analyze large datasets efficiently. The process involves several key steps:

Step 1: Data Loading and Setup

- Import necessary libraries and create a SparkSession, which serves as the entry point to Spark functionality.
- Load data from various sources such as CSV, JSON, Parquet, or databases into a Spark DataFrame.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("EDA with Spark").getOrCreate()
```

- `df = spark.read.csv("path/to/data.csv", header=True, inferSchema=True)`

Step 2: Initial Data Inspection

- Examine the data structure using methods like `df.show()` to display sample records.
- Check the schema with `df.printSchema()` to understand column data types.
- Get basic statistics with `df.describe().show()` to view summary statistics.
- Count the number of rows using `df.count()` to understand dataset size.

Step 3: Data Cleaning and Preprocessing

- Handle missing values using methods like `df.dropna()` or `df.fillna()`.
- Remove duplicates with `df.dropDuplicates()`.
- Standardize or normalize data using Spark SQL functions or UDFs (User Defined Functions).
- Filter outliers using SQL expressions or DataFrame operations.

Step 4: Feature Analysis and Visualization

- Use `groupBy()` and aggregation functions to analyze relationships between variables.
- Create frequency distributions of categorical variables.

Convert Spark DataFrames to Pandas DataFrames for visualization when working with smaller datasets:

```
pandas_df = df.toPandas()
```

- `# Then use Matplotlib, Seaborn, or other Python visualization libraries`

- For large datasets, sample data or use Spark's built-in statistical functions.

Step 5: Advanced Analysis

- Perform correlation analysis using `df.stat.corr()` to identify relationships between numerical features.

Use SQL queries directly on DataFrames for complex analysis:
python
```
df.createOrReplaceTempView("data_table")
```

- `result = spark.sql("SELECT category, AVG(value) FROM data_table GROUP BY category")`

- Implement custom aggregations using window functions for time-series or grouped analysis.

Step 6: Performance Optimization

- Cache frequently used DataFrames with `df.cache()` to improve performance.
- Use efficient transformations and actions to minimize shuffle operations.
- Monitor query execution plans with `df.explain()` to identify bottlenecks.

Conclusion:

Apache Spark provides a powerful platform for performing Exploratory Data Analysis on large-scale datasets through its distributed computing capabilities. By combining Spark's high-performance processing with Pandas' visualization strengths, data scientists can efficiently analyze, clean, and extract insights from massive amounts of data. The integration of Spark SQL further enhances the analysis capabilities by allowing SQL-like queries on structured data.

The process begins with loading data into Spark DataFrames, followed by systematic inspection, cleaning, and analysis using both Spark's native functions and, when appropriate, conversion to Pandas for more nuanced visualization. This hybrid approach leverages the respective strengths of both frameworks: Spark's ability to process big data and Pandas' rich visualization ecosystem.

For organizations dealing with big data, mastering EDA with Apache Spark is essential for deriving meaningful insights that drive data-informed decision-making. The distributed nature of Spark makes it particularly suitable for scenarios where traditional tools like Pandas alone would be insufficient due to memory constraints or processing limitations.

experiment10

Name: Ayush Maurya Class:D15C/34

Experiment 10

Aim: To perform Batch and Streamed Data Analysis using Apache Spark.

Theory:

1. What is streaming? Explain batch and stream data.

Data Processing Paradigms: Batch vs Stream

Data processing systems generally fall into two major paradigms: batch processing and stream processing. These approaches differ fundamentally in how they handle data and when they produce results.

Batch Processing refers to the technique of collecting data over a period of time, storing it, and then processing it as a single unit or "batch." In this approach:

- Data is accumulated over time before processing begins
- Processing occurs on a predefined schedule (hourly, daily, weekly)
- Results are generated after the entire dataset has been processed
- It's well-suited for applications where insights aren't needed in real-time
- Examples include end-of-day financial reports, customer segmentation analysis, and periodic business intelligence processing

Batch processing is analogous to doing laundry - you collect dirty clothes for several days, then wash them all at once when the basket is full.

Stream Processing handles data as it arrives, processing each data item individually or in small time-windowed batches. Key characteristics include:

- Data is processed continuously as it's generated
- Results are updated in near real-time
- Processing must handle unbounded datasets (potentially infinite data flows)
- It's ideal for applications requiring immediate insights or actions
- Examples include fraud detection, IoT sensor monitoring, real-time recommendations, and social media trend analysis

Stream processing resembles a flowing river where water (data) is constantly moving and being processed as it passes through various points.

The primary differences between batch and stream processing center around timing (when processing occurs), data scope (finite vs. infinite), latency requirements (delayed vs. immediate results), and processing model (stateless vs. stateful computation).

2. How data streaming takes place using Apache Spark.

Data Streaming with Apache Spark

Apache Spark offers robust capabilities for stream processing through its Spark Streaming and newer Structured Streaming APIs. The streaming process in Spark follows these key steps:

Step 1: Setting Up the Streaming Environment

- Import necessary libraries and create a SparkSession configured for streaming

For Structured Streaming (recommended):

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \

 .appName("Spark Streaming Application") \

 .config("spark.streaming.stopGracefullyOnShutdown", "true") \

 .getOrCreate()
```

Step 2: Defining Input Sources

- Connect to streaming data sources such as:
    - Apache Kafka for message queuing
    - Kinesis for AWS environments
    - Socket connections for simple testing
    - File-based streaming (monitoring directories)

Example with Kafka source:

```
df = spark.readStream \

 .format("kafka") \

 .option("kafka.bootstrap.servers", "host:port") \

 .option("subscribe", "topic_name") \

 .load()
```

Step 3: Data Transformation

Apply schema to structured data:

```
from pyspark.sql.types import StructType, StructField, StringType, TimestampType
```

schema = StructType([...])

parsed_df = df.selectExpr("CAST(value AS STRING)") \

.select(from_json(col("value"), schema).alias("data")) \

.select("data.*")

- Perform transformations using DataFrame operations:
  - Filter records with `filter()` or `where()`
  - Aggregate data with `groupBy()` and aggregation functions
  - Enrich data by joining with static datasets
  - Apply window functions for time-based analysis

Step 4: Processing Models Spark Streaming offers two processing models:

1. **Micro-batch Processing** (Default):
   - Processes data in small, discrete time-windowed batches
   - Provides exactly-once processing semantics
   - Offers higher throughput at the cost of slightly higher latency
2. **Continuous Processing** (Low-latency mode):
   - Processes each record as it arrives
   - Provides at-least-once semantics
   - Achieves lower latency (as low as 1ms) but with reduced throughput

Step 5: State Management

Maintain state across batches for complex analytics: # Word count example with state

wordCounts = df.groupBy("word") \

.count()

Define watermarks for handling late data:

df = df.withWatermark("timestamp", "10 minutes") \

.groupBy(window("timestamp", "5 minutes"), "id") \

.count()

Step 6: Output Sinks

Define where processed results should be written:

query = wordCounts.writeStream \

```
.outputMode("complete") \

.format("console") \

.start()
```

- Common output options include:
    - Console (for debugging)
    - Memory (for querying from another thread)
    - File sinks (Parquet, CSV, etc.)
    - Kafka topics
    - Custom sinks via the ForeachBatch API

Step 7: Stream Execution and Monitoring

- Start the streaming query:
  query.awaitTermination()

Monitor streaming statistics:
query.status

query.lastProgress

query.recentProgress

Step 8: Handling Failures and Recovery

Implement checkpoint directories for fault tolerance:

```
query = df.writeStream \

.outputMode("append") \

.format("parquet") \

.option("checkpointLocation", "/path/to/checkpoint/dir") \

.start("/output/path")
```

- Spark automatically recovers from failures using checkpoint information

Name: Ayush Maurya Class:D15C/34

Conclusion:

Apache Spark provides a comprehensive framework for processing both batch and streaming data at scale. The experiment demonstrates how Spark's unified processing model bridges these two paradigms, allowing organizations to implement solutions that handle historical data analysis alongside real-time data processing using largely the same codebase.

Spark Streaming, particularly the newer Structured Streaming API, significantly simplifies building resilient streaming applications by abstracting away many of the complexities associated with stream processing such as fault tolerance, exactly-once semantics, and state management. By treating streaming computations as continuous queries on unbounded tables, Spark makes stream processing more accessible to data engineers and scientists already familiar with batch processing concepts.

The integration of streaming capabilities within the broader Spark ecosystem enables powerful hybrid architectures that combine batch processing for historical analysis with stream processing for real-time insights. This flexibility allows organizations to implement Lambda or Kappa architectural patterns depending on their specific requirements.

For production deployments, organizations should carefully consider configuration parameters related to micro-batch intervals, memory allocation, and checkpoint strategies to optimize performance and reliability. Additionally, integration with monitoring systems is essential for tracking streaming application health and performance metrics over time.

As data volumes continue to grow and real-time decision-making becomes increasingly important across industries, mastering tools like Spark Streaming represents an essential capability for modern data teams seeking to derive timely insights from continuously flowing data.

Report

Name: Ayush Maurya Class:D15C/34

Introduction

The ozone layer in Earth's stratosphere plays a critical role by absorbing harmful ultraviolet (UV) radiation from the Sun. Without this protective shield, Earth would face increased UV exposure, leading to health problems like skin cancer and cataracts, as well as damage to ecosystems. Human activities, especially the use of chlorofluorocarbons (CFCs) and other ozone-depleting substances, have contributed to the thinning of this crucial layer, particularly over polar regions. Traditional ozone monitoring methods are expensive and limited in scope, but machine learning techniques offer new possibilities for real-time analysis and prediction.

Problem Statement

Maintaining ozone layer health presents a complex challenge, with traditional monitoring methods being costly and time-consuming, limiting real-time detection capabilities. This project addresses this gap by leveraging machine learning to evaluate ozone layer quality using environmental data. The goal is to develop a predictive model that classifies the ozone layer's health status based on parameters such as temperature, wind speed, and pressure, providing timely insights for more efficient monitoring and informed environmental decision-making.

Dataset Description

The dataset contains environmental and weather-related features crucial for analyzing ozone layer health. It includes measurements of temperature, humidity, wind speed, pressure, and other meteorological parameters collected at various times throughout the day. Specific columns include weather station readings (WSR0-WSR23), temperature measurements (T0-T23), relative humidity values (RH0-RH23), wind speed (U0-U23), wind direction (V0-V23), and precipitation data. The target variable "Result" indicates the classification of ozone damage levels, representing whether the ozone layer is in a good or bad state. This comprehensive dataset allows the model to analyze how multiple environmental factors influence ozone depletion.

Model Implementation: XGBoost with SMOTE

Initial model implementations (Logistic Regression, SVM, KNN, and basic XGBoost) faced significant challenges due to extreme class imbalance. While these models achieved seemingly impressive accuracy scores around 93%, deeper analysis revealed they were essentially predicting everything as the majority class, completely failing to identify instances of the minority class.

The breakthrough came with the implementation of the Synthetic Minority Over-sampling Technique (SMOTE) to address the class imbalance problem. SMOTE created synthetic examples of the minority class to balance the dataset, transforming the training data distribution. When combined with XGBoost, this approach dramatically improved performance.

The XGBoost model with SMOTE achieved an outstanding accuracy of 99.17%, with balanced performance across both classes. Most importantly, it demonstrated excellent minority class detection with 95% recall and 92% precision, resulting in an F1-score of 0.94 for the minority class (compared to effectively zero in all pre-SMOTE models). The confusion matrix further illustrated this improvement, showing 152 true positives versus only 8 false negatives for the minority class, while maintaining strong majority class performance with 2361 true negatives versus just 13 false positives.

This optimized model proved that the environmental features contained sufficient information to distinguish between ozone concentration levels when class representation issues were properly addressed. The balanced precision and recall demonstrated that the model could effectively identify high-ozone conditions without excessive false alarms, making it practical for operational deployment.

Societal Impact

Environmental Conservation

The project directly contributes to environmental conservation efforts by enabling real-time prediction of ozone damage levels. This allows for tracking ozone depletion trends and implementing timely interventions to prevent further damage. Government agencies can use these predictions to enforce stricter regulations on industries producing ozone-depleting substances. Additionally, the system can raise public awareness about ozone health, potentially driving community-based conservation efforts as people better understand the consequences of ozone depletion.

Policy-Making Support

Accurate ozone damage predictions enhance policy-making capabilities. Governments and environmental organizations can use real-time data to craft targeted policies and regulatory measures. This supports informed policy decisions based on current environmental conditions and enables data-driven advocacy for stronger environmental protection laws. The system also contributes to disaster preparedness by providing early warnings about increased UV radiation exposure, allowing public health departments to launch preventative campaigns.

Social Awareness and Education

The project makes complex environmental data accessible to the general public, raising global awareness about ozone depletion. By providing easy access to real-time information, it

empowers individuals and communities to participate in environmental conservation efforts and advocate for stronger policies. Educational institutions can incorporate the application into environmental science curricula, providing students with hands-on learning experiences about real-time environmental monitoring.

Future Scope

Enhanced Environmental Data Integration

Future versions can incorporate additional environmental factors like Air Quality Index (AQI), greenhouse gas levels, and land use changes. This would provide a more comprehensive understanding of environmental health and improve prediction accuracy by accounting for more variables that affect ozone layer conditions.

Long-Term Trend Prediction

The system could be expanded to predict future ozone damage levels based on historical data patterns. This long-term forecasting capability would help policymakers develop sustainable environmental practices and better understand seasonal variations in ozone health.

Global Deployment

While currently designed for localized use, the system has potential for global implementation through environmental organizations like the United Nations Environment Programme (UNEP). This would facilitate real-time global insights into ozone health and enable coordinated international responses. The system could also serve as a collaborative platform where stakeholders worldwide contribute to monitoring efforts.

Integration with Other Environmental Models

Linking ozone damage predictions with broader climate change models would provide insights into the relationship between ozone depletion and climate change. Integration with biodiversity monitoring systems could help understand the cascading effects of ozone depletion on ecosystems and species.

Enhanced Accessibility

Future development could focus on improving the user interface to increase accessibility for non-technical users, including multiple language options and voice-assisted interfaces to reach a broader audience.

By addressing a critical environmental issue with innovative technology, this project has the potential to make a profound impact on global environmental conservation efforts while providing valuable data for policy-making and public education about ozone layer protection.

Name: Ayush Maurya Class:D15C/34

Conclusion

This project highlights the potential of machine learning in combating ozone layer depletion. Using an XGBoost model with SMOTE, we developed a system that accurately predicts ozone damage levels from environmental data, achieving a remarkable 99.17% accuracy and balanced class performance.

The system's real-time predictions can guide policy decisions, raise awareness, and support conservation efforts, benefiting environmental agencies, educational institutions, and advocacy groups. As the project evolves, incorporating more data and expanding globally, it can become a crucial tool for environmental protection, offering valuable insights for stakeholders worldwide. Ultimately, this project contributes to safeguarding the ozone layer, protecting Earth’s defense against harmful UV radiation for future generations.