Machine Learning Lab

Week2:

Task: **DataPre-processing and visualization**

Required Python Libraries: pandas, matplotlib, seaborn

Problems:

Import Dataset [car_sales.csv]

Download File::

https://github.com/chandanverma07/DataSets/blob/master/Car_sales.csv

Installation of Seaborn Library

For Python environment :

pip install seaborn

For conda environment :

conda install seaborn

Need Data Pre-Processing?

Raw data often contains:

- Missing values (e.g., empty cells)
- Duplicate entries (same data appearing multiple times)
- Incorrect data types (e.g., text instead of numbers)
- Inconsistent formats (e.g., different date formats)
- Outliers (extremely high or low values affecting results)

By pre-processing the data, we ensure:

✓ Data is clean and structured

✓ Models perform better

✓ Accurate visualizations and insights

Pandas is used for data cleaning and transformation

Matplotlib creates basic plots for visualization

Seaborn helps in advanced and aesthetic statistical visualizations

Data Pre-Processing ensures data quality before visualization

Visualization techniques help understand relationships and detect patterns

Seaborn

- Built on top of Matplotlib
- **Advanced statistical visualizations** (heatmaps, box plots, violin plots)
- **Easier to use** with built-in themes and color schemes

Code:

```
import pandas as pd
df=pd.read_csv("car_sales.csv")
print(df.head())
```

- pd.read_csv("Car_sales.csv"): Reads the dataset into a Pandas DataFrame.
- df.head(): Displays the top five rows of the dataset to understand its structure.

print(df.head())

- Explicitly prints the DataFrame's first five rows as text.
- Used when running scripts in environments like command-line interfaces (CLI) or Python scripts, where implicit display doesn't happen.

df.head()

- Returns the first five rows of the DataFrame **as an object**.
- In interactive environments like Jupyter Notebook, the DataFrame is automatically displayed with proper formatting (e.g., styled tables).

code:

```
import pandas as pd
df = pd.read_csv("car_sales.csv")
num_attributes = df.shape[1]
print("Number of attributes:", num_attributes)
```

- df.shape returns a tuple (rows, columns).
- df.shape[1] gives the number of columns (attributes).

Code:

```
import pandas as pd
df = pd.read_csv("car_sales.csv")
print("DataFrame Info:")
print(df.info())
df.info()
```

- Provides an overview of the dataset, including the number of **non-null values** and **data types** of columns.

df.dtypes

- Lists the **data type** of each column.

Code:

```
import pandas as pd
df = pd.read_csv("car_sales.csv")
num_rows, num_columns = df.shape
print("Number of rows:", num_rows)
print("Number of columns:", num_columns)
```

- df.shape[0] → **Number of rows (records)**
- df.shape[1] → **Number of columns (attributes)**

Duplicates:

Code:

```
df_cleaned = df.drop_duplicates()
print("Number of rows before removing duplicates:", df.shape[0])
print("Number of rows after removing duplicates:",
df_cleaned.shape[0])
```

- **df.drop_duplicates()**
- Removes duplicate rows from the DataFrame.
- Returns a **new DataFrame** without modifying the original.
- **Checking the number of rows before and after**
- df.shape[0] → Number of rows **before** removing duplicates.

Print Summary Statistics for Numerical Variables

Code:

```
print("Summary Statistics for Numerical Variables:")
print(df.describe())
```

df.describe() provides statistical details of numerical columns, such as:

- **count** (number of values)
- **mean** (average)
- **std** (standard deviation)
- **min** (minimum value)
- **25%, 50%, 75%** (quartiles)
- **max** (maximum value)

Missing Values in Each Column

- df.isnull().sum() counts missing values in each column.

```
print("Number of missing values in each column:")
print(df.isnull().sum())
```

Drop the Column with Most Missing Values

- df.isnull().sum().idxmax() finds the column with the most missing values.

- df.drop(columns=[column_to_drop]) removes it.

Impute (Fill) Missing Numerical Values

in our dataset (car_sales.csv), some rows have missing values in the set

- Incomplete data entry
- Errors in data collection
- Data corruption

To check missing values in the dataset:

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

- in case occurring missing values in the data set

following this steps to fill the data **or** replaces missing values

df[" "].fillna(df[" "].mean(), inplace=True)

- **df[" "].mean()** → Calculates the mean (average) **example** of all available values.
- **fillna(df[" "].mean())** → Replaces missing values in the "**example**" column with this mean.
- **inplace=True** → Modifies the original df without creating a new copy.

Ex:

Code:

```
import pandas as pd

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

# Check missing values before imputation
print("Missing values before:\n", df.isnull().sum())

df["Sales in thousands"].fillna(df["Sales in thousands"].mean(), inplace=True)

print("Missing values after:\n", df.isnull().sum())

print(df.head())
```

Horsepower, Length, and Fuel Efficiency

- Function Definition (find_min_max): The function calculates the min and max values for a given column in the DataFrame and returns the corresponding rows.
- Calling the Function: We call find_min_max with "Horsepower" as the column name to get the min/max horsepower values and their corresponding rows

Code:

```
import pandas as pd
```

```
# Load the dataset
df = pd.read_csv("car_sales.csv")

# Function to find min and max values of any column
def find_min_max(df, column_name):
    min_value = df[column_name].min()
    max_value = df[column_name].max()

    # Find the rows with the min and max values
    min_row = df[df[column_name] == min_value]
    max_row = df[df[column_name] == max_value]

    return min_value, max_value, min_row, max_row

# Now call the function to find min and max for 'Horsepower'
hp_min, hp_max, hp_min_row, hp_max_row = find_min_max(df,
"Horsepower")

# Print results
print("\nHorsepower - Min:", hp_min, "\n", hp_min_row)
print("\nHorsepower - Max:", hp_max, "\n", hp_max_row)
```

the Function:

- **df[column_name].min()**: This finds the **minimum value** in the specified column (column_name).

- **df[column_name].max()**: This finds the **maximum value** in the specified column (column_name).
- **df[df[column_name] == min_value]**: This filters the DataFrame and returns the rows where the column value equals the **minimum value**.
- **df[df[column_name] == max_value]**: This filters the DataFrame and returns the rows where the column value equals the **maximum value**.

Probability density distribution of continuous numerical variable-length

```
plt.figure(figsize=(8, 5))
```

figsize=(8, 5): Creates a figure **8 inches wide and 5 inches tall**.

```
plt.hist(df[column], bins=30, density=True, alpha=0.6, color="green", label="Histogram")
```

Histogram: A bar graph representing the **frequency distribution** of df[column].

- **Parameters**:
  - df[column]: The numerical data to be plotted.
  - bins=30: Divides data into **30 bins (intervals)**.

- density=True: Normalizes the histogram so that the total area equals **1** (converts frequency to probability density).
- alpha=0.6: Sets **transparency** (0 = fully transparent, 1 = fully visible).
- color="green": Sets the **bar color** to green.
- label="Histogram": Adds a label for the legend

```
sns.kdeplot(df[column], fill=True, color="blue", alpha=0.5, label="KDE")
```

- **KDE (Kernel Density Estimation)**: A smooth curve that estimates the **probability density function (PDF)** of the data.
- **Parameters**:
  - df[column]: The numerical data to estimate the density.
  - fill=True: Fills the area under the KDE curve.
  - color="blue": Sets the KDE curve **color**.
  - alpha=0.5: Sets the transparency of the filled area.
  - label="KDE": Adds a label for the legend.

Ex:

```
plt.figure(figsize=(8, 5))
plt.hist(df[column], bins=30, density=True, alpha=0.6, color="green", label="Histogram")
sns.kdeplot(df[column], fill=True, color="blue", alpha=0.5, label="KDE")
plt.xlabel("column")
plt.ylabel("Density")
plt.title(f"PDF of {column} (Histogram + KDE)")
plt.legend()
```

```
plt.grid()
plt.show()
```

Count by category –group by manufacture

To count the number of cars by manufacturer (or any other categorical column) in your dataset, you can use the groupby() function in pandas along with size() or count(). Here's how you can do it:

Code:

```
import pandas as pd
df = pd.read_csv("car_sales.csv")
manufacturer_counts = df.groupby('Manufacturer').size().reset_index(name='Count')
print(manufacturer_counts)
```

1. **groupby('Manufacturer')**:
    - Groups the dataset by the Manufacturer column.
2. **size()**:
    - Counts the number of rows (cars) for each manufacturer.
3. **reset_index(name='Count')**:
    - Converts the grouped result into a DataFrame and renames the count column to Count.

Boxplot of other numerical variables w.r.t manufacture

- **Clean Column Names**:
  Spaces are trimmed from column names using df.columns.str.strip() to avoid mismatches.
- **Select Numerical Columns**:
  We use df.select_dtypes(include=['number']) to pick out only the numerical columns (e.g., Price, Sales, etc.) excluding Manufacturer.
- **Loop Through Numerical Columns**:
  For each numerical column (e.g., Price, Horsepower), a boxplot is generated with sns.boxplot().
- **X-axis**: Represents different **Manufacturers**.
- **Y-axis**: Represents the numerical variable being analyzed (e.g., Price, Sales).
- **Boxplot Visualization**:
  A boxplot is generated to visualize the distribution, including the **median**, **quartiles**, and potential **outliers** for each manufacturer.
- **Rotation of X-axis Labels**:
  The x-axis labels (Manufacturer names) are rotated for better readability using plt.xticks(rotation=45).

Code:

```
import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt


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


df.columns = df.columns.str.strip()
```

```
numerical_columns = df.select_dtypes(include=['number']).columns
numerical_columns = [col for col in numerical_columns if col != 'Manufacturer']


if 'Manufacturer' in df.columns:
    # Create boxplots for each numerical column with respect to 'Manufacturer'
    for col in numerical_columns:
        plt.figure(figsize=(10, 6))
        sns.boxplot(x='Manufacturer', y=col, data=df)
        plt.title(f'Boxplot of {col} by Manufacturer')
        plt.xlabel('Manufacturer')
        plt.ylabel(col)
        plt.xticks(rotation=45)
        plt.show()
else:
    print("Error: 'Manufacturer' column not found in the dataset.")
```

We aim to create **boxplots** for each numerical variable (like Price, Sales, Horsepower, etc.) with respect to the **Manufacturer**.4

Divide the data

1. **Check Column Names**:
    - The column names are printed to help you verify the exact name of the sales column.
    - A case-insensitive search is performed to find any column containing the word 'sales'.
2. **Handle Missing Column**:

- If no column related to 'sales' is found, a KeyError is raised with a helpful message.

3. **Extract y (Output)**:
   - The first matching column is used as the sales column, and its values are assigned to y.
4. **Extract X (Input)**:
   - The sales column is dropped from the dataset, and the remaining columns are assigned to X.
5. **Display Shapes and Sample Data**:
   - The shapes of X and y are printed to confirm the split.
   - The first few rows of X and y are displayed for verification.

Code:

```
import pandas as pd
df = pd.read_csv('car_sales.csv')
print("Column names in the dataset:")
print(df.columns)
sales_column = [col for col in df.columns if 'sales' in col.lower()]
if not sales_column:
    raise KeyError("No column related to 'sales' found in the dataset. Please check the column names.")
sales_column_name = sales_column[0]
print(f"Using column '{sales_column_name}' as the sales column.")
```

```
y = df[sales_column_name]
X = df.drop(columns=[sales_column_name])

print("\nShape of X (input):", X.shape)
print("Shape of y (output):", y.shape)

print("\nInput (X):")
print(X.head())
print("\nOutput (y):")
print(y.head())
```

Encode other categorical variables using label encoder

Label Encoding is a technique used to convert **categorical variables** into **numeric form** so that they can be used in machine learning models. The idea is to **assign a unique integer** to each category in a given feature (column).

How It Works:

- Each **unique category** in the categorical column is mapped to an **integer** value.
- This mapping typically follows the order of appearance or a lexicographical order of the categories.

For example, consider the following dataset:

Color
Red
Blue

Color
Green
Blue
Red

Label Encoding assigns an integer to each color:

- **Red** → 0
- **Blue** → 1
- **Green** → 2

After encoding:

Color _encoded

Red 0

Blue 1

Green 2

Blue 1

Red 0

Advantages of Label Encoding:

1. **Simplicity**: It's easy to implement and is very fast, making it useful for datasets with smaller categorical variables.
2. **Memory-Efficient**: It uses a single column for encoding, reducing the number of columns required, unlike One-Hot Encoding.
3. **Maintains Information**: It preserves the **order of the data** if the categories have a natural ranking or hierarchy (for example, "Low", "Medium", "High").

Example:

```
import pandas as pd

from sklearn.preprocessing import LabelEncoder

data = {'Color': ['Red', 'Blue', 'Green', 'Blue', 'Red', 'Yellow']}

df = pd.DataFrame(data)

encoder = LabelEncoder()

df['Color_encoded'] = encoder.fit_transform(df['Color'])
```

```
print(df)
```

Example:

Code:

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder

df = pd.read_csv('car_sales.csv')
print("Column names in the dataset:")
print(df.columns)
sales_column = [col for col in df.columns if 'sales' in col.lower()]
if not sales_column:
    raise KeyError("No column related to 'sales' found in the dataset. Please check the column names.")
sales_column_name = sales_column[0]
print(f"Using column '{sales_column_name}' as the sales column.")
y = df[sales_column_name]
X = df.drop(columns=[sales_column_name])
categorical_columns = X.select_dtypes(include=['object', 'category']).columns
label_encoders = {}
for col in categorical_columns:
    le = LabelEncoder()
    X[col] = le.fit_transform(X[col])
```

```
    label_encoders[col] = le
print("\nShape of X (input):", X.shape)
print("Shape of y (output):", y.shape)
print("\nInput (X) with encoded categorical variables:")
print(X.head())
print("\nOutput (y):")
print(y.head())
for col in categorical_columns:
    print(f"\nMapping for {col}:")
    print(dict(zip(label_encoders[col].classes_,
label_encoders[col].transform(label_encoders[col].classes_))))
```

Explanation of the Code:

1. **Identify Categorical Columns**:
    - X.select_dtypes(include=['object', 'category']) identifies columns with categorical data (e.g., strings or categories).
2. **Encode Categorical Columns**:
    - A LabelEncoder is applied to each categorical column.
    - The fit_transform method converts categorical values into numerical labels.
    - The encoders are stored in a dictionary (label_encoders) for potential inverse transformation later.

Encoding Techniques (Color Example)

Method	Description	Example (Color)
Label Encoding	Assigns a unique integer to each category.	"Red" → 2, "Blue" → 0, "Green" → 1, "Yellow" → 3
One-Hot Encoding	Creates binary columns for each category.	"Red" → 0, "Blue" → 1, "Green" → 0, "Yellow" → 1 (for each column)

Method	Description	Example (Color)
Ordinal Encoding	Converts categories into integers based on a predefined order.	"Red" → 0, "Yellow" → 1, "Green" → 2, "Blue" → 3
Binary Encoding	Converts categories into binary digits.	"Red" → 1 1 0, "Blue" → 0 0 1, etc.
Frequency Encoding	Replaces categories with the frequency of that category.	"Red" → 2, "Blue" → 2, "Green" → 1, "Yellow" → 1
Target Encoding	Replaces categories with the mean of the target variable.	"Red" → 20500, "Blue" → 24000, etc.

One-Hot Encoding

One-Hot Encoding is a technique used to convert **categorical variables** into a **binary (0 or 1)** format, where each category is represented by a separate column. For each category in the original feature, a new column is created, and that column is filled with **0s** and **1s**, where **1** indicates the presence of that category in a given record, and **0** indicates its absence.

How It Works:

1. For each unique category in the column, a **new binary column** is created.
2. Each column corresponds to one category.
3. If a record has that category, the corresponding column is marked as **1**, otherwise, it is **0**.

applying **One-Hot Encoding**, we would get:

Color_Red	Color_Blue	Color_Green
1	0	0
0	1	0
0	0	1
0	1	0
1	0	0

- The original `Color` column is split into three new binary columns: `Color_Red`, `Color_Blue`, and `Color_Green`.
- **Red** is represented as `1` in the `Color_Red` column and `0` in the others.
- **Blue** is represented as `1` in the `Color_Blue` column and `0` in the others.
- **Green** is represented as `1` in the `Color_Green` column and `0` in the others.

Use One-Hot Encoding:

- When the categorical variable is **nominal** (no intrinsic order).
- When the number of unique categories is **manageable** and doesn't lead to a huge number of new columns.

Example:

```
import pandas as pd

data = {'Color': ['Red', 'Blue', 'Green', 'Blue', 'Red']}

df = pd.DataFrame(data)

df_encoded = pd.get_dummies(df, columns=['Color'])

print(df_encoded)
```

- **Color_Blue** column is 1 when the color is blue, and 0 otherwise.
- **Color_Green** column is 1 when the color is green, and 0 otherwise.
- **Color_Red** column is 1 when the color is red, and 0 otherwise.

One-Hot Encoding:

Method	Description	When to Use
One-Hot Encoding	Converts each category in a feature into a new binary column.	For **nominal** data (unordered categories).
Result	Generates multiple binary columns where each category becomes a column.	Best for **low cardinality** categories.

Encode categorical variable vehicle type using one-hot Encoder

One-Hot Encoding (OHE) is a technique used to convert categorical data into numerical format by creating separate binary columns for each unique category.

1. **Machine Learning Models Need Numeric Input**
   - Many ML algorithms (**Linear Regression, Neural Networks**) don't understand text data.
   - One-Hot Encoding converts text into numbers that models can process.

Code:

```
import pandas as pd
df = pd.read_csv("car_sales.csv")
df.columns = df.columns.str.strip()
print("Available columns:", df.columns)


if 'Vehicle type' in df.columns:

    df = pd.get_dummies(df, columns=['Vehicle type'], prefix='Vehicle', drop_first=True)

    print(df.head())
else:
    print("Error: 'Vehicle type' column not found in the dataset.")
```

 df.columns = df.columns.str.strip()

- **Removes extra spaces** from all column names, ensuring they match exactly.

 print("Available columns:", df.columns)

- **Displays the column names** so you can check the correct spelling and formatting before using them.

 if 'Vehicle type' in df.columns:

- **Checks if the column exists** before applying One-Hot Encoding.

 pd.get_dummies(df, columns=['Vehicle type'], prefix='Vehicle', drop_first=True)

- **Converts the categorical column into numerical columns** (one column per category).
- **Uses drop_first=True** to avoid redundancy (removes one category to prevent multicollinearity).

Split the Dataset into Train (70%) and Test (10%

 train_test_split(X, y, test_size=0.30, random_state=42)

- Splits **30%** of data as X_temp, y_temp (this includes both test + unused data).
- Leaves **70%** as the X_train, y_train.

 train_test_split(X_temp, y_temp, test_size=2/3, random_state=42)

- Takes **10% of the original dataset** as X_test, y_test.
- The remaining **20% is ignored**, but you can use it as a **validation set** if needed.

 Random State (random_state=42)

- Ensures the split is **reproducible** (same results every time).

Code:

```
import pandas as pd
from sklearn.model_selection import train_test_split
df = pd.read_csv("car_sales.csv")
df.columns = df.columns.str.strip()

X = df.drop(columns=['Sales in thousands'])
y = df['Sales in thousands']
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.30, random_state=42)
X_test, _, y_test, _ = train_test_split(X_temp, y_temp, test_size=2/3, random_state=42)
print("Training set shape:", X_train.shape, y_train.shape)
print("Test set shape:", X_test.shape, y_test.shape)
```

Final Data Distribution

Dataset	Percentage
Training Set (`X_train`, `y_train`)	70%
Test Set (`X_test`, `y_test`)	10%
Unused Data (or Validation)	20%

- **If you also need a validation set**, just store the ignored **20% as `X_val, y_val`** instead of discarding it.

Data visualization with Seaborn Pairplot

Data Visualization is the presentation of data in pictorial format. It is extremely important for Data Analysis, primarily because of the fantastic ecosystem of data-centric Python packages. **Seaborn** is one of those packages that can make analyzing data much easier.

- It **automatically selects numerical columns** from the dataset.
- It **creates scatter plots** for every pair of numerical columns.
- It **plots histograms** (or KDE plots) on the diagonal to show individual distributions.

Code:

```
import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt


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

df.columns = df.columns.str.lower()

sns.pairplot(df)

plt.show()
```

- **Converts column names to lowercase** to avoid case mismatches.
- **Uses sns.pairplot(df)** to create pairwise scatter plots & histograms for all numerical columns.
- **Displays the plot** with plt.show().

Plot correlation of price and sales using scatterplot

- **Reads the dataset**: pd.read_csv("car_sales.csv")

 Handles case differences: Converts all column names to lowercase.

 Finds the correct columns:

- Searches for a column containing "price".
- Searches for a column containing "sales" or "units".

 Plots a scatter plot if both columns exist.

 Handles errors: If columns are missing, it prints an error message.

Code:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv("car_sales.csv")
df.columns = df.columns.str.lower()
price_col = next((col for col in df.columns if "price" in col), None)
sales_col = next((col for col in df.columns if "sales" in col or "units" in col), None)

if price_col and sales_col:
    plt.figure(figsize=(8, 5))
    sns.scatterplot(x=df[price_col], y=df[sales_col])
    plt.xlabel(price_col.capitalize())
    plt.ylabel(sales_col.capitalize())
    plt.title("Price vs Sales Scatter Plot")
```

```
    plt.show()
else:
    print("Error: Could not find 'price' and 'sales' columns.")
```

Boxplot of sales of different manufacturer

- Creates a sample dataset with "manufacturer" and "sales" columns.
- Uses sns.boxplot() to visualize the distribution of sales for each manufacturer.
- Adjusts figure size & labels for better readability

Code:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Sample dataset with 'manufacturer' and 'sales' columns
data = {
    "manufacturer": ["Toyota", "Ford", "BMW", "Toyota", "Ford", "BMW", "Toyota", "Ford", "BMW"],
    "sales": [500, 450, 520, 600, 490, 580, 620, 470, 590]
}

# Create DataFrame
df = pd.DataFrame(data)
```

```
# Create a boxplot of sales by manufacturer
plt.figure(figsize=(8, 5))
sns.boxplot(x=df["manufacturer"], y=df["sales"])

# Customize labels
plt.xlabel("Manufacturer")
plt.ylabel("Sales")
plt.title("Sales Distribution by Manufacturer")
plt.show()
```

.

- Compares sales distributions across Toyota, Ford, and BMW.
- Median (central line): Shows typical sales for each manufacturer.
- Interquartile range (IQR): Box represents the middle 50% of sales.
- Outliers (dots): Show any extreme values.

correlation coefficient value of price and sales

ex: sample Data:

- **Creates a sample dataset** with price and sales columns.
- **Calculates the correlation coefficient** using .corr(), which will give a value between -1 and 1.
- **Prints the correlation coefficient** between price and sales.

Code:

```
import pandas as pd
```

```
data = {
    "price": [20000, 25000, 30000, 35000, 40000, 45000, 50000],
    "sales": [250, 200, 150, 130, 120, 100, 90]
}
df = pd.DataFrame(data)
correlation = df["price"].corr(df["sales"])
print(f"The correlation coefficient between price and sales is: {correlation}")
```

he correlation coefficient will indicate the **relationship** between price and sales.

- A **negative correlation** (near -1) means that as price increases, sales decrease.
- A **positive correlation** (near 1) means that as price increases, sales also increase.
- A **value near 0** suggests no linear correlation.

Sort and Find Most & Least Expensive Cars

- **Sort the data** by price in **ascending** order using df.sort_values(by="price").
- **Find the least expensive car** by accessing the **first row** (iloc[0]) after sorting.
- **Find the most expensive car** by accessing the **last row** (iloc[-1]).
- **Print the details** of both the least and most expensive cars.

Code:

```
import pandas as pd

data = {
    "car": ["Toyota", "Ford", "BMW", "Honda", "Mercedes", "Audi", "Nissan"],
    "price": [20000, 25000, 30000, 35000, 40000, 45000, 50000],
    "sales": [250, 200, 150, 130, 120, 100, 90]
}

df = pd.DataFrame(data)
df_sorted = df.sort_values(by="price")

least_expensive_car = df_sorted.iloc[0]
most_expensive_car = df_sorted.iloc[-1]
print("Least Expensive Car:")
print(least_expensive_car)

print("\nMost Expensive Car:")
print(most_expensive_car)
```

For the sample dataset:

- The **least expensive car** will be the one with the lowest price.
- The **most expensive car** will be the one with the highest price.

find min and max values of any column

Find Min and Max Values:

1. Load the dataset.
2. Identify the column you want to analyze (for example, Horsepower or any other numerical column).
3. Use the .min() and .max() methods to find the min and max values.
4. Optionally, display the rows corresponding to the min and max values.

Example:

```
import pandas as pd

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

def find_min_max(df, column_name):

    min_value = df[column_name].min()

    max_value = df[column_name].max()



    min_row = df[df[column_name] == min_value]

    max_row = df[df[column_name] == max_value]


    return min_value, max_value, min_row, max_row

min_value, max_value, min_row, max_row = find_min_max(df,
'Horsepower')


# Display the results

print(f"Min value: {min_value}\n", min_row)

print(f"Max value: {max_value}\n", max_row)
```

Steps to Customize:

- Replace 'Horsepower' with the name of any other numerical column you want to find the min and max values for.
- Ensure that the column you're interested in is numerical (i.e., contains numeric values).

Handling Missing or Non-Numeric Data:

If your column contains missing or non-numeric data, you can clean the data before finding the min and max values. For example:

```
df['Horsepower'] = pd.to_numeric(df['Horsepower'], errors='coerce')
df = df.dropna(subset=['Horsepower'])

min_value, max_value, min_row, max_row = find_min_max(df, 'Horsepower')
print(f"Min value: {min_value}\n", min_row)
print(f"Max value: {max_value}\n", max_row)
```

Plot Histograms for Multiple Continuous Variables

Code:

- **Load the Data**:
- We load the car_sales.csv dataset using pd.read_csv.
- **List of Columns**:
- The columns **price**, **sales**, **horsepower**, and **fuelefficiency** are stored in a list columns for easy iteration.
- **Subplots**:
- We set up a 2x2 grid of subplots using plt.subplots(2, 2, figsize=(12, 10)), where each plot will be placed in a specific position.

- **Plotting Histograms**:

- For each variable, we use sns.histplot() to plot a histogram. The kde=True option adds a smooth **Kernel Density Estimate** curve to the histogram, which gives a better sense of the distribution.

- **Customize Titles and Labels**:

- The set_title(), set_xlabel(), and set_ylabel() functions are used to customize the plot's title and labels.

- **Layout Adjustment**:

- plt.tight_layout() is used to adjust the layout to avoid any overlap between plots.

Code:

```
import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns


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

columns = ["Price in thousands", "Sales in thousands", "Horsepower", "Fuel efficiency"]

fig, axes = plt.subplots(2, 2, figsize=(12, 10))  # 2 rows and 2 columns



for i, column in enumerate(columns):

    # Determine row and column index for the subplot

    ax = axes[i // 2, i % 2]
```

```
    # Plot histogram
    sns.histplot(df[column], kde=True, ax=ax)


    ax.set_title(f"Histogram of {column.capitalize()}")
    ax.set_xlabel(column.capitalize())
    ax.set_ylabel("Frequency")


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

This code will produce 4 histograms:

1. Histogram of price
2. Histogram of sales
3. Histogram of horsepower
4. Histogram of fuelefficiency

Each histogram will show the frequency distribution of these continuous variables.

Apply feature scaling on numerical variables

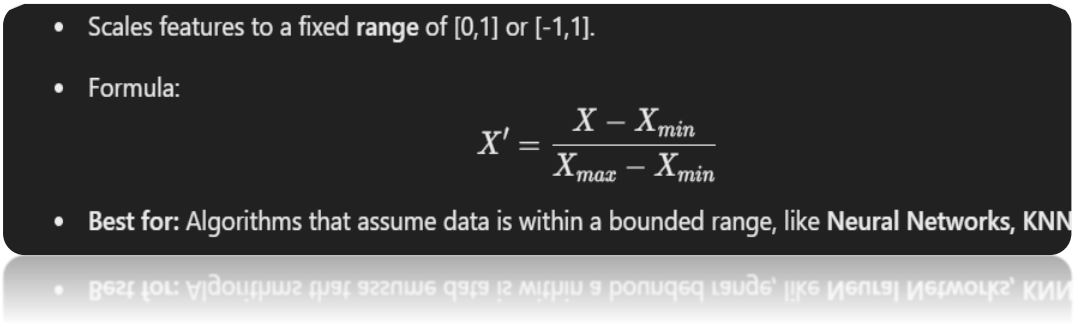
Feature Scaling Important?

1. **Improves Model Performance**: Algorithms like **KNN, SVM, Logistic Regression, and Neural Networks** perform better when data is scaled.

2. **Speeds Up Convergence**: Gradient-based optimization (e.g., in Deep Learning) converges faster with scaled data.
3. **Removes Bias from Large Values**: Models might assign higher importance to features with larger ranges.
4. **Essential for Distance-Based Algorithms**: Algorithms like **K-Means, KNN, and PCA** rely on distances (Euclidean or Manhattan) and need features on the same scale.

Min-Max Scaling (Normalization)

- Scales features to a fixed **range** of [0,1] or [-1,1].
- Formula



Example:

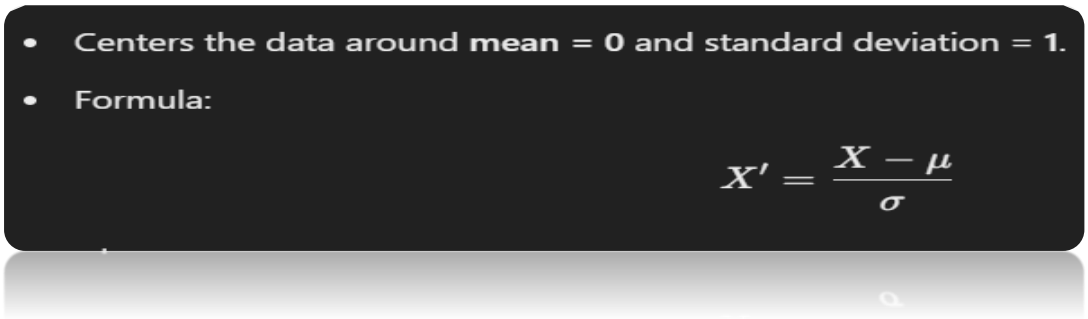
```
from sklearn.preprocessing import MinMaxScaler
import pandas as pd


# Sample DataFrame
df = pd.DataFrame({'Horsepower': [100, 150, 200, 250, 300]})


# Initialize the scaler
scaler = MinMaxScaler()


# Fit and transform the data
```

df['Horsepower_Scaled'] = scaler.fit_transform(df[['Horsepower']])

print(df)

Standardization (Z-Score Scaling)

- Centers the data around **mean = 0** and standard deviation = **1**



 where:

- μ\muμ = Mean of the feature
- σ\sigmaσ = Standard deviation

 Best for: Algorithms assuming normally distributed data (**Logistic Regression, Linear Regression, PCA, SVM, K-Means**).

Example:

from sklearn.preprocessing import StandardScaler

Initialize the scaler

scaler = StandardScaler()

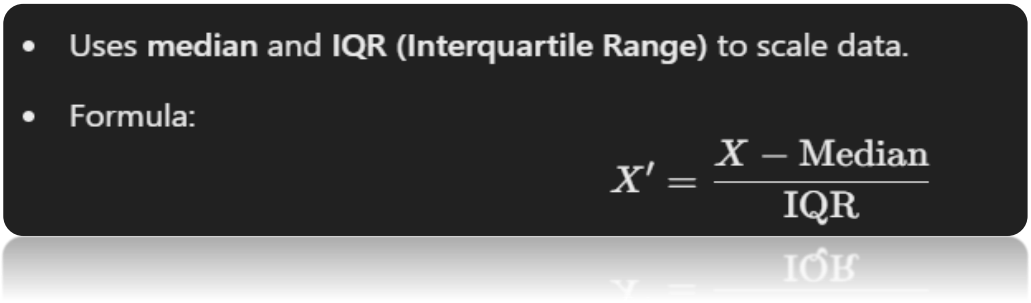
Fit and transform the data

df['Horsepower_Standardized'] = scaler.fit_transform(df[['Horsepower']])

print(df)

Robust Scaling (for Outliers)

- Uses **median** and **IQR (Interquartile Range)** to scale data.
-



 where **IQR = Q3 - Q1** (75th percentile - 25th percentile).

 Best for: Datasets with **outliers** (since mean-based scaling is sensitive to extreme values).

Example:

from sklearn.preprocessing import RobustScaler

Initialize the scaler

scaler = RobustScaler()

Fit and transform the data

df['Horsepower_Robust'] = scaler.fit_transform(df[['Horsepower']])

print(df)

Code:

```
import pandas as pd
from sklearn.preprocessing import StandardScaler, MinMaxScaler



df = pd.read_csv("car_sales.csv")
columns = ["Price in thousands", "Sales in thousands",
"Horsepower", "Fuel efficiency"]
df[columns] = df[columns].apply(pd.to_numeric, errors='coerce')
df_cleaned = df.dropna(subset=columns)
scaler = StandardScaler()
df_standardized = df_cleaned.copy()
df_standardized[columns] =
scaler.fit_transform(df_cleaned[columns])


normalizer = MinMaxScaler()


df_normalized = df_cleaned.copy()
df_normalized[columns] =
normalizer.fit_transform(df_cleaned[columns])


print("Standardized Data:")
```

```
print(df_standardized[columns].head())

print("\nNormalized Data:")
print(df_normalized[columns].head())
```

- **Data Cleaning**: Any invalid numeric data (like .) is converted to NaN and handled.
- **Standardized Data**: The numerical columns will be standardized to have a **mean of 0** and a **standard deviation of 1**.
- **Normalized Data**: The numerical columns will be scaled between **0** and **1**.

Exercise Questions:

1. Read top five values
2. Print dataframe info_data types of each column
3. Print number of rows and columns
4. Drop duplicate rows_if any
5. Print number of rows and columns after dropping duplicates
6. Print summary statistics for numerical variables
7. Print number of missing values in each column
8. Drop the column with most missing values
9. Drop the rows with categorical missing values
10. Import the rows with numerical missing values
11. Sort the data w.r.t price_find the details of the most and the least expensive cars

12. Write a function to find min and max values of any column

13. Call the above function to find min max of horse power, length, fuel efficiency

14. Plot histogram of continuous numerical variable : price,sales,hoursepower,fuelefficiency

15. Probability density distribution of continuous numerical variable-length

16. Count by category –group by manufacture

17. Select all numerical variables

18. Print correlation coefficient value of price and sales

19. Plot correlation of price and sales using scatterplot

20.Pair plot

21. Boxplot of sales of different manufacturer

22. Boxplot of other numerical variables w.r.t manufacture

23. Divide the data into input and output y=sales in thousand, x=all other variables

24. Encode other categorical variables using label encoder

25. Encode categorical variable vehicle type using one-hot encoder

26.Split the data set into train and test set 70% train set 10% test set

27. Apply feature scaling on numerical variables