

# Module 13 - DATA ANALYSIS

Welcome to Module 13! This module is your gateway into the fascinating world of Data Analysis. Data analysis is the process of inspecting, cleaning, transforming, and modeling data with the goal of discovering useful information, informing conclusions, and supporting decision-making.

We'll cover essential tools, techniques, and statistical concepts that data analysts use daily.

---

## Chapter 1: Data Analysis

### 1.1 What is Data Analysis?

- **Definition:** Data analysis is a methodical approach to extract insights from raw data. It involves applying statistical and logical techniques to describe, illustrate, condense, recap, and evaluate data.
- **Purpose:** To make sense of data, identify trends, solve problems, test hypotheses, and support informed decision-making.
- **It's an iterative process:** You often go back and forth between different stages.

### 1.2 Why is Data Analysis Important?

- **Informed Decision Making:** Moves decisions from intuition to data-backed evidence.
- **Pattern Recognition:** Helps uncover hidden patterns and relationships within data.
- **Predictive Capabilities:** Enables forecasting future trends and outcomes.
- **Efficiency and Optimization:** Identifies areas for improvement in processes and operations.
- **Problem Solving:** Pinpoints root causes of issues.

### 1.3 Steps in Data Analysis (Overview)

While the exact steps can vary, a typical data analysis workflow includes:

1. **Data Collection:** Gathering data from various sources (databases, files, APIs, web scraping).
2. **Data Cleaning/Wrangling:** Handling missing values, outliers, inconsistencies, and errors.
3. **Exploratory Data Analysis (EDA):** Summarizing main characteristics of the data, often with visual methods, to uncover patterns and anomalies.
4. **Data Transformation:** Formatting, normalizing, binning data to make it suitable for analysis or modeling.
5. **Modeling (Statistical Analysis / Machine Learning):** Applying statistical methods or machine learning algorithms to derive insights or make predictions.
6. **Interpretation of Results:** Understanding what the findings mean in the context of the problem.

7. **Data Visualization:** Presenting findings clearly and effectively using charts and graphs.
  8. **Communication:** Sharing insights with stakeholders.
- 

## Chapter 2: Setting Up Your Environment - Installing Anaconda

For data analysis in Python, **Anaconda** is the recommended platform.

### 2.1 What is Anaconda?

- Anaconda is a free and open-source distribution of the Python and R programming languages for scientific computing, data science, machine learning, and large-scale data processing.
- It simplifies package management and deployment.

### 2.2 Why Use Anaconda?

- **Bundled Libraries:** Comes pre-packaged with over 150 popular data science packages (e.g., NumPy, Pandas, Scikit-learn, Matplotlib, SciPy), saving you the hassle of installing them individually.
- **conda Package Manager:** A powerful package and environment management system that goes beyond `pip`. It can manage packages for any language, and it handles non-Python dependencies.
- **Virtual Environments:** Provides robust tools for creating isolated environments for different projects, preventing dependency conflicts (similar to `venv` but more comprehensive).
- **Anaconda Navigator:** A desktop GUI that allows you to launch applications and easily manage `conda` packages, environments, and channels without using command-line commands.

### 2.3 Installation Steps (General Overview)

1. **Download:** Go to the official Anaconda Distribution website ([www.anaconda.com/download](https://www.anaconda.com/download)).
2. **Select Your OS:** Choose the installer for your operating system (Windows, macOS, or Linux).
3. **Download Installer:** Download the graphical installer (usually the latest Python 3.x version).
4. **Run Installer:** Follow the instructions provided by the installer.
  - **Important:**
    - For "Installation Type," choose "Just Me" unless you need it for all users.
    - For "Advanced Installation Options," it's generally recommended to **leave "Add Anaconda to my PATH environment variable" UNCHECKED** unless you know what you are doing (it can interfere with other Python installations). Instead, rely on the Anaconda Prompt (Windows) or terminal after installation.

- **Check "Register Anaconda as my default Python 3.x"** (this is usually enabled by default).
5. **Complete Installation:** The installer will copy files and configure your system.

## 2.4 Verifying Installation

- **Anaconda Navigator:** Search for "Anaconda Navigator" in your applications/Start Menu and launch it. If it opens successfully, your installation is working.
- **Command Line:**
  - **Windows:** Open "Anaconda Prompt" (from your Start Menu).
  - **macOS / Linux:** Open your regular terminal.
  - Type:

Bash

```
conda --version  
python --version
```

- You should see version numbers for both, indicating that Anaconda's `conda` and Python are accessible.
- 

## Chapter 3: Introduction to Jupyter Notebook

After installing Anaconda, you now have access to Jupyter Notebook, an incredibly popular tool for interactive data science.

### 3.1 What is Jupyter Notebook?

- **Definition:** Jupyter Notebook is an open-source web application that allows you to create and share documents containing live code, equations, visualizations, and narrative text.
- **Interactive Environment:** It provides an interactive computing environment where you can execute code cell by cell, see immediate results, and iterate quickly on your analysis.
- **Document Format:** A Jupyter Notebook (`.ipynb` file) combines code, output, and explanatory text in a single, shareable document, making it excellent for documenting your data analysis workflow.

### 3.2 Launching Jupyter Notebook from Anaconda

There are two primary ways to launch Jupyter Notebook after installing Anaconda:

1. **Using Anaconda Navigator (Recommended for beginners):**
  - Open **Anaconda Navigator** from your Start Menu (Windows) or Applications folder (macOS).
  - In the Navigator window, you will see a list of applications. Find and click the **"Launch"** button under **Jupyter Notebook**.

- This will open a new tab in your web browser (usually your default browser) that displays the Jupyter file explorer interface.
- 2. **Using the Command Line:**
  - Open your **Anaconda Prompt** (Windows) or regular Terminal (macOS/Linux).
  - Navigate to the directory where you want to store your notebook files using `cd` commands (e.g., `cd Documents/MyDataAnalysis`).
  - Type the command: `jupyter notebook`
  - Press Enter. This will start the Jupyter server and open a new tab in your web browser.

Once the Jupyter interface opens in your browser, you can:

- Navigate through your file system.
- Click **"New"** (usually top-right) and select **"Python 3 (ipykernel)"** to create a new blank notebook.
- Click on an existing `.ipynb` file to open it.

### 3.3 Key Features & Usage Tips for Jupyter Notebook

- **Cells:**
  - Notebooks are made up of individual cells.
  - **Code cells:** Contain executable Python code.
  - **Markdown cells:** Contain text formatted using Markdown, useful for explanations, headings, and notes.
  - You can change the cell type using the dropdown menu in the toolbar.
- **Execution:**
  - To run a code cell, select it and press `Shift + Enter` or click the "Run" button in the toolbar.
  - The output of the code (results, print statements, plots) will appear directly below the cell.
- **Inline Plots (`%matplotlib inline`):**
  - For plots generated using Matplotlib or Seaborn to display directly within your notebook output (rather than in a separate window), always include the following "magic command" at the very beginning of your notebook or before your first plotting code:

Python

```
%matplotlib inline
```

- **Saving:** Your notebook is automatically saved periodically, but you can manually save by clicking the floppy disk icon or going to `File -> Save and Checkpoint`.
- **Kernel:** The "kernel" is the computational engine that executes the code in your notebook. If your code gets stuck or you want to clear variables, you can restart the kernel (`Kernel -> Restart`).

## Chapter 4: Essential Python Libraries for Data Analysis

Anaconda comes pre-installed with the most crucial libraries for data analysis. Let's briefly introduce them and their core functionalities.

### 4.1 Introduction to Key Libraries

#### 1. NumPy (Numerical Python):

- **Purpose:** The fundamental package for numerical computation in Python. Provides support for large, multi-dimensional arrays and matrices, along with a collection of high-level mathematical functions to operate on these arrays.
- **Why important:** Forms the backbone for many other data science libraries (Pandas is built on NumPy). Highly optimized for performance.

#### 2. Pandas:

- **Purpose:** A powerful library for data manipulation and analysis. It introduces two primary data structures: `Series` (1D labeled array) and `DataFrame` (2D labeled table-like structure).
- **Why important:** Makes working with tabular data (like spreadsheets or SQL tables) incredibly easy and efficient. Ideal for data cleaning, transformation, and exploration.

#### 3. Matplotlib:

- **Purpose:** A comprehensive library for creating static, animated, and interactive visualizations in Python.
- **Why important:** Provides the foundational tools for plotting various types of charts (line plots, scatter plots, histograms, bar charts, etc.).

#### 4. Seaborn:

- **Purpose:** A statistical data visualization library based on Matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.
- **Why important:** Simplifies the creation of complex statistical plots (e.g., heatmaps, violin plots, regression plots) and often produces more aesthetically pleasing results than raw Matplotlib.

#### 5. SciPy (Scientific Python):

- **Purpose:** Builds on NumPy and provides a collection of algorithms and tools for scientific computing, including optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, and more.
- **Why important:** Used for advanced statistical functions and scientific computations.

#### 6. Scikit-learn (sklearn):

- **Purpose:** A robust and comprehensive library for machine learning in Python. It provides simple and efficient tools for data mining and data analysis.
  - **Why important:** Offers a wide range of supervised and unsupervised learning algorithms, including regression, classification, clustering, dimensionality reduction, and model selection.
-

## Chapter 5: Loading and Viewing Data (Pandas Basics)

The first step in any data analysis project is to load your data into a suitable structure and then inspect it. Pandas `DataFrame` is the go-to structure for tabular data.

### 5.1 Importing CSV File

CSV (Comma Separated Values) is a very common format for sharing tabular data. Pandas provides a simple function to read them.

- **Example Code (Jupyter Cell):**

Python

```
import pandas as pd
import numpy as np # Needed for creating NaN values for demonstration
import matplotlib.pyplot as plt
import seaborn as sns

# To ensure plots appear inline in Jupyter Notebook
%matplotlib inline

# Read the CSV file into a Pandas DataFrame
# Make sure 'cars.csv' is in the same directory as your Python
script/notebook,
# or provide the full path to the file.
df = pd.read_csv('cars.csv')
print("CSV file loaded successfully!")
print(df)
```

```
CSV file loaded successfully!
   Make  Model  Year  Mileage  Price  FuelType
0  Toyota  Camry  2020   50000  25000   Petrol
1   Honda  Civic  2021   30000  22000   Petrol
2   Ford   F-150  2019   70000  35000   Diesel
3    BMW     X5   2022   20000  60000   Petrol
4   Audi    A4   2021   45000  48000   Diesel
5 Mercedes C-Class  2023   15000  55000   Petrol
6   Nissan  Altima  2020   60000  18000   Petrol
7  Hyundai  Elantra  2022   25000  28000   Petrol
8 Chevrolet  Malibu  2018   85000  15000   Petrol
9 Volkswagen  Jetta  2019   40000  19000   Petrol
```

### 5.2 Methods to View Data

Once loaded, you'll want to inspect your `DataFrame` to understand its structure and content.

- **`df.head(n)`**: Displays the first `n` rows of the `DataFrame` (default is 5). Useful for a quick peek.

Python

```
print("--- df.head() ---")
```

`df.head()` # In Jupyter, the last expression is displayed automatically

```
[21]: df.head()
```

```
[21]:
```

	Make	Model	Year	Mileage	Price	FuelType
0	Toyota	Camry	2020	50000	25000	Petrol
1	Honda	Civic	2021	30000	22000	Petrol
2	Ford	F-150	2019	70000	35000	Diesel
3	BMW	X5	2022	20000	60000	Petrol
4	Audi	A4	2021	45000	48000	Diesel

- **`df.tail(n)`**: Displays the last `n` rows of the DataFrame (default is 5). Useful for checking the end of the data.

Python

```
print("\n--- df.tail(2) ---")
df.tail(2) # Display last 2 rows
```

```
[15]: df.tail(2)
```

```
[15]:
```

	Make	Model	Year	Mileage	Price	FuelType
8	Chevrolet	Malibu	2018	85000	15000	Petrol
9	Volkswagen	Jetta	2019	40000	19000	Petrol

- **`df.info()`**: Provides a concise summary of the DataFrame, including the number of non-null entries, data types of each column, and memory usage.

Python

```
print("\n--- df.info() ---")
df.info()
```

```
[17]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 6 columns):
#   Column      Non-Null Count  Dtype
---  ---
0   Make        10 non-null    object
1   Model       10 non-null    object
2   Year        10 non-null    int64
3   Mileage     10 non-null    int64
4   Price       10 non-null    int64
5   FuelType    10 non-null    object
dtypes: int64(3), object(3)
memory usage: 612.0+ bytes
```

- **`df.describe()`**: Generates descriptive statistics of numerical columns (count, mean, std, min, max, quartiles).



Python

```
print("\n--- df.describe() ---")
df.describe()
```

```
[19]: df.describe()
```

```
[19]:
```

	Year	Mileage	Price
<b>count</b>	10.000000	10.000000	10.000000
<b>mean</b>	2020.500000	44000.000000	32500.000000
<b>std</b>	1.581139	22705.848488	16311.209506
<b>min</b>	2018.000000	15000.000000	15000.000000
<b>25%</b>	2019.250000	26250.000000	19750.000000
<b>50%</b>	2020.500000	42500.000000	26500.000000
<b>75%</b>	2021.750000	57500.000000	44750.000000
<b>max</b>	2023.000000	85000.000000	60000.000000

- To include categorical columns, use `df.describe(include='all')`.

Python

```
print("\n--- df.describe(include='all') ---")
df.describe(include='all')
```

```
[23]: df.describe(include='all')
```

```
[23]:
```

	Make	Model	Year	Mileage	Price	FuelType
<b>count</b>	10	10	10.000000	10.000000	10.000000	10
<b>unique</b>	10	10	NaN	NaN	NaN	2
<b>top</b>	Toyota	Camry	NaN	NaN	NaN	Petrol
<b>freq</b>	1	1	NaN	NaN	NaN	8
<b>mean</b>	NaN	NaN	2020.500000	44000.000000	32500.000000	NaN
<b>std</b>	NaN	NaN	1.581139	22705.848488	16311.209506	NaN
<b>min</b>	NaN	NaN	2018.000000	15000.000000	15000.000000	NaN
<b>25%</b>	NaN	NaN	2019.250000	26250.000000	19750.000000	NaN
<b>50%</b>	NaN	NaN	2020.500000	42500.000000	26500.000000	NaN
<b>75%</b>	NaN	NaN	2021.750000	57500.000000	44750.000000	NaN
<b>max</b>	NaN	NaN	2023.000000	85000.000000	60000.000000	NaN

- **df.shape:** Returns a tuple representing the dimensions of the DataFrame (rows, columns).

Python

```
print("\n--- df.shape ---")
df.shape
```



```
[25]: df.shape
[25]: (10, 6)
```

- **df.columns:** Returns a list of column labels.

Python

```
print("\n--- df.columns ---")
df.columns
```

- **Selecting a Column (Series):** You can access individual columns, which are Pandas Series objects.

Python

```
print("\n--- Selecting 'Price' column (first 5 values) ---")
df['Price'].head() # Using dictionary-like access
```

Python

```
print("\n--- Selecting 'Year' column (first 5 values) using
attribute-like access ---")
df.Year.head() # Attribute-like access (if column name is a valid
Python identifier)
```

```
[27]: df.Year.head()
[27]: 0    2020
      1    2021
      2    2019
      3    2022
      4    2021
      Name: Year, dtype: int64
```

---

## Chapter 6: Data Cleaning (Handling Missing Data)

Raw data is rarely perfect. Data cleaning is the process of fixing or removing incorrect, corrupted, incorrectly formatted, duplicate, or incomplete data within a dataset. Missing values are one of the most common issues.

### 6.1 What is Data Cleaning?

- Ensuring data quality and reliability.
- Addressing inconsistencies, typos, outliers, and missing information.
- Crucial step before analysis, as "Garbage In, Garbage Out" applies.

### 6.2 Identifying Missing Values

Missing values are typically represented as `NaN` (Not a Number) in Pandas.

- **df.isnull() / df.isna() :** Returns a DataFrame of boolean values indicating `True` for missing values and `False` otherwise.

Python

```
# Let's intentionally introduce some missing values for demonstration
df_copy = df.copy()
df_copy.loc[1, 'Mileage'] = np.nan # Make Mileage for Honda Civic missing
df_copy.loc[3, 'FuelType'] = np.nan # Make FuelType for BMW X5 missing
df_copy.loc[6, 'Price'] = np.nan # Make Price for Nissan Altima missing

print("--- df_copy.isnull() for missing values ---")
df_copy.isnull()
```

```
[29]: df_copy = df.copy()
df_copy.loc[1, 'Mileage'] = np.nan # Make Mileage for Honda Civic missing
df_copy.loc[3, 'FuelType'] = np.nan # Make FuelType for BMW X5 missing
df_copy.loc[6, 'Price'] = np.nan # Make Price for Nissan Altima missing

print("--- df_copy.isnull() for missing values ---")
df_copy.isnull()
```

--- df\_copy.isnull() for missing values ---

```
[29]:
```

	Make	Model	Year	Mileage	Price	FuelType
0	False	False	False	False	False	False
1	False	False	False	True	False	False
2	False	False	False	False	False	False
3	False	False	False	False	False	True
4	False	False	False	False	False	False
5	False	False	False	False	False	False
6	False	False	False	False	True	False
7	False	False	False	False	False	False
8	False	False	False	False	False	False
9	False	False	False	False	False	False

- **df.isnull().sum() :** Counts the number of missing values per column. Very useful for quickly seeing where the problems are.

Python

```
print("\n--- Count of missing values per column ---")
df_copy.isnull().sum()
```

### 6.3 Handling Missing Values

Once identified, missing values need to be dealt with. Common strategies include:

#### 1. Dropping Data:

- **df.dropna(axis=0) :** Drops rows that contain any `NaN` values. `axis=0` means rows.

Python

```
print("\n--- df_copy.dropna(axis=0) (drops rows with NaN) ---")
df_dropped_rows = df_copy.dropna(axis=0)
print("Original shape:", df_copy.shape)
print("Shape after dropping rows:", df_dropped_rows.shape)
df_dropped_rows
```

- **df.dropna(axis=1):** Drops columns that contain any NaN values. axis=1 means columns.

Python

```
print("\n--- df_copy.dropna(axis=1) (drops columns with NaN) ---")
df_dropped_cols = df_copy.dropna(axis=1)
print("Original shape:", df_copy.shape)
print("Shape after dropping columns:", df_dropped_cols.shape)
df_dropped_cols
```

- **When to drop:** When only a few rows have missing data, or when a column has a very high percentage of missing data and is not critical.

## 2. Replacing (Imputation):

- **df.fillna(value):** Fills NaN values with a specified value.
- **Replacing with a specific value:**

Python

```
print("\n--- Replacing NaN in 'FuelType' with 'Unknown' ---")
df_filled_specific = df_copy.copy()
df_filled_specific['FuelType'].fillna('Unknown', inplace=True)
print("Missing values after filling 'FuelType':\n",
df_filled_specific.isnull().sum())
df_filled_specific
```

- **Replacing with the mean/median/mode:**
  - **Mean:** For numerical data.

Python

```
print("\n--- Replacing NaN in 'Mileage' with its mean ---")
df_filled_mean = df_copy.copy()
avg_mileage = df_filled_mean['Mileage'].mean()
df_filled_mean['Mileage'].fillna(avg_mileage,
inplace=True)
print("Missing values after filling 'Mileage':\n",
df_filled_mean.isnull().sum())
df_filled_mean
```

- **Mode:** For categorical data.

Python

```

print("\n--- Replacing NaN in 'FuelType' with its mode ---")
df_filled_mode = df_copy.copy()
# idxmax() gets the index (value) of the max count
mode_fueltype = df_filled_mode['FuelType'].mode()[0]
df_filled_mode['FuelType'].fillna(mode_fueltype,
inplace=True)
print("Missing values after filling 'FuelType' with
mode:\n", df_filled_mode.isnull().sum())
df_filled_mode

```

- **Median:** Often preferred over mean for skewed numerical data as it's less sensitive to outliers.

Python

```

print("\n--- Replacing NaN in 'Price' with its median ---")
df_filled_median = df_copy.copy()
median_price = df_filled_median['Price'].median()
df_filled_median['Price'].fillna(median_price,
inplace=True)
print("Missing values after filling 'Price' with
median:\n", df_filled_median.isnull().sum())
df_filled_median

```

- **Forward Fill (ffill) or Backward Fill (bfill):** Fills missing values with the previous (ffill) or next (bfill) valid observation. Useful for time series data.

Python

```

print("\n--- Replacing NaN with ffill (forward fill) on
'Mileage' ---")
df_filled_ffill = df_copy.copy()
df_filled_ffill['Mileage'].fillna(method='ffill', inplace=True)
print("Missing values after ffill on 'Mileage':\n",
df_filled_ffill.isnull().sum())
df_filled_ffill

```

- **When to replace:** When dropping too many rows/columns would lead to significant data loss, and when a reasonable imputation strategy can be applied.

---

## Chapter 7: Data Transformation (Formatting, Normalization, Binning)

After cleaning, data often needs to be transformed to be in a suitable format for analysis or modelling.

### 7.1 Data Formatting

Ensuring data is in a consistent and correct format.

- **Consistent Data Types:** Converting columns to appropriate data types (int, float, object for strings, datetime).

Python

```
print("--- Data Types before formatting ---")
df.dtypes

# Convert 'Year' to integer if it's not already (it's already int
from dummy data)
df['Year'] = df['Year'].astype(int)

# Convert 'Price' to float if it's not already (it's float from dummy
data)
df['Price'] = df['Price'].astype(float)

# Convert 'FuelType' to 'category' for efficiency if it has few
unique values
df['FuelType'] = df['FuelType'].astype('category')

print("\n--- Data Types after formatting ---")
df.dtypes
```

- **Renaming Columns:** For clarity or consistency.

Python

```
print("\n--- Renaming 'Mileage' to 'Kilometers' ---")
df_renamed = df.rename(columns={'Mileage': 'Kilometers'})
df_renamed.columns
```

- **Standardizing Units:** Converting units to a common scale (e.g., miles to kilometers, USD to INR).

Python

```
print("\n--- Standardizing units (Mileage from assumed miles to
kilometers) ---")
# Assuming 'Mileage' was in miles and we want kilometers (1 mile =
1.60934 km)
df_units = df.copy()
df_units['Mileage_km'] = df_units['Mileage'] * 1.60934
df_units[['Mileage', 'Mileage_km']].head()
```

## 7.2 Data Normalization (Scaling)

Rescaling features to a standard range, which helps many machine learning algorithms perform better (e.g., gradient descent-based algorithms, distance-based algorithms like KNN).

- **Purpose:**
  - Prevents features with larger values from dominating the learning process.
  - Helps algorithms converge faster.
- **Method 1: Min-Max Scaling (Data Normalization)**
  - Rescales values to a range between 0 and 1.
  - Formula:  $X_{\text{normalized}} = (X - X_{\text{min}}) / (X_{\text{max}} - X_{\text{min}})$

## Python

```
print("\n--- Min-Max Scaling on 'Price' ---")
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
# Reshape the column for the scaler (needs 2D array)
df_normalized = df.copy()
df_normalized['Price_Normalized'] =
scaler.fit_transform(df_normalized[['Price']])
print("Min Normalized Price:",
df_normalized['Price_Normalized'].min())
print("Max Normalized Price:",
df_normalized['Price_Normalized'].max())
df_normalized[['Price', 'Price_Normalized']].head()
```

```
[33]: print("\n--- Min-Max Scaling on 'Price' ---")
      from sklearn.preprocessing import MinMaxScaler

      scaler = MinMaxScaler()
      # Reshape the column for the scaler (needs 2D array)
      df_normalized = df.copy()
      df_normalized['Price_Normalized'] = scaler.fit_transform(df_normalized[['Price']])
      print("Min Normalized Price:", df_normalized['Price_Normalized'].min())
      print("Max Normalized Price:", df_normalized['Price_Normalized'].max())
      df_normalized[['Price', 'Price_Normalized']].head()
```

```
--- Min-Max Scaling on 'Price' ---
Min Normalized Price: 0.0
Max Normalized Price: 1.0
```

```
[33]:
```

	Price	Price_Normalized
0	25000	0.222222
1	22000	0.155556
2	35000	0.444444
3	60000	1.000000
4	48000	0.733333

- **Method 2: Z-score Standardization (Data Normalization 2)**
  - Rescales values to have a mean of 0 and a standard deviation of 1.
  - Formula:  $X_{\text{standardized}} = (X - \mu) / \sigma$  (where  $\mu$  is mean,  $\sigma$  is standard deviation)

## Python

```
print("\n--- Z-score Standardization on 'Mileage' ---")
from sklearn.preprocessing import StandardScaler

scaler_z = StandardScaler()
df_standardized = df.copy()
df_standardized['Mileage_Standardized'] =
scaler_z.fit_transform(df_standardized[['Mileage']])
print("Mean Standardized Mileage (approx 0):",
df_standardized['Mileage_Standardized'].mean())
print("Std Dev Standardized Mileage (approx 1):",
df_standardized['Mileage_Standardized'].std())
df_standardized[['Mileage', 'Mileage_Standardized']].head()
```

```
[35]: print("\n--- Z-score Standardization on 'Mileage' ---")
      from sklearn.preprocessing import StandardScaler

      scaler_z = StandardScaler()
      df_standardized = df.copy()
      df_standardized['Mileage_Standardized'] = scaler_z.fit_transform(df_standardized[['Mileage']])
      print("Mean Standardized Mileage (approx 0):", df_standardized['Mileage_Standardized'].mean())
      print("Std Dev Standardized Mileage (approx 1):", df_standardized['Mileage_Standardized'].std())
      df_standardized[['Mileage', 'Mileage_Standardized']].head()
```

```
--- Z-score Standardization on 'Mileage' ---
Mean Standardized Mileage (approx 0): -2.2204460492503132e-17
Std Dev Standardized Mileage (approx 1): 1.0540925533894598
```

```
[35]:
```

	Mileage	Mileage_Standardized
0	50000	0.278543
1	30000	-0.649934
2	70000	1.207020
3	20000	-1.114172
4	45000	0.046424

- **When to use which:**
  - **Min-Max:** Good when you know the approximate upper and lower bounds of your data, or when the data is not normally distributed. Sensitive to outliers.
  - **Z-score:** Good for data that follows a Gaussian (normal) distribution. Less sensitive to outliers than Min-Max.

### 7.3 Binning

Transforming continuous numerical variables into discrete categorical "bins."

- **Purpose:**
  - Simplifies data, making it easier to analyse and interpret.
  - Reduces the impact of small fluctuations or noise in continuous data.
  - Can be useful for converting numerical features into categorical ones for certain models or visualizations.
- **Method 1: `pd.cut()` (Equal-width or custom bins)**
  - Used to segment and sort data values into bins. You can define the number of bins or specify custom bin edges.

Python

```
print("\n--- Binning 'Price' into 3 equal-width bins ---")
# Define bin labels
bins_labels = ['Low', 'Medium', 'High']
# Cut the 'Price' column into 3 bins
df_binned = df.copy()
df_binned['Price_Category'] = pd.cut(df_binned['Price'], bins=3,
labels=bins_labels, include_lowest=True)
print("\nValue counts for Price_Category:")
print(df_binned['Price_Category'].value_counts())
df_binned[['Price', 'Price_Category']].head()
```



```
[39]: print("\n--- Binning 'Price' into 3 equal-width bins ---")
# Define bin labels
bins_labels = ['Low', 'Medium', 'High']
# Cut the 'Price' column into 3 bins
df_binned = df.copy()
df_binned['Price_Category'] = pd.cut(df_binned['Price'], bins=3, labels=bins_labels, include_lowest=True)
print("\nValue counts for Price_Category:")
print(df_binned['Price_Category'].value_counts())
df_binned[['Price', 'Price_Category']].head()
```

--- Binning 'Price' into 3 equal-width bins ---

Value counts for Price\_Category:

```
Price_Category
Low          6
High         3
Medium       1
Name: count, dtype: int64
```

```
[39]:
```

	Price	Price_Category
0	25000	Low
1	22000	Low
2	35000	Medium
3	60000	High
4	48000	High

- **Method 2: `pd.qcut()` (Equal-frequency/quantile bins) (Binning 2)**
  - Used to cut data into  $q$  equal-sized bins based on quantiles. Each bin will have roughly the same number of observations.

Python

```
print("\n--- Binning 'Mileage' into 4 equal-frequency (quantile) bins ---")
df_qcut_binned = df.copy()
# qcut will try to make sure each bin has roughly the same number of data points
df_qcut_binned['Mileage_Quartile'] =
pd.qcut(df_qcut_binned['Mileage'], q=4, labels=['Q1', 'Q2', 'Q3', 'Q4'])
print("\nValue counts for Mileage_Quartile:")
print(df_qcut_binned['Mileage_Quartile'].value_counts())
df_qcut_binned[['Mileage', 'Mileage_Quartile']].head()
```

```
[41]: print("\n--- Binning 'Mileage' into 4 equal-frequency (quantile) bins ---")
df_qcut_binned = df.copy()
# qcut will try to make sure each bin has roughly the same number of data points
df_qcut_binned['Mileage_Quartile'] = pd.qcut(df_qcut_binned['Mileage'], q=4, labels=['Q1', 'Q2', 'Q3', 'Q4'])
print("\nValue counts for Mileage_Quartile:")
print(df_qcut_binned['Mileage_Quartile'].value_counts())
df_qcut_binned[['Mileage', 'Mileage_Quartile']].head()
```

```
--- Binning 'Mileage' into 4 equal-frequency (quantile) bins ---

Value counts for Mileage_Quartile:
Mileage_Quartile
Q1      3
Q4      3
Q2      2
Q3      2
Name: count, dtype: int64
```

```
[41]:
```

	Mileage	Mileage_Quartile
0	50000	Q3
1	30000	Q2
2	70000	Q4
3	20000	Q1
4	45000	Q3

## Chapter 8: Exploratory Data Analysis (EDA) - Basic Statistics & Visualization

EDA is a critical step to understand your data, identify patterns, and detect anomalies.

### 8.1 Value Counts

For categorical (or discrete) data, `value_counts()` is essential for understanding the distribution of unique values.

- **`Series.value_counts()`**: Returns a Series containing counts of unique values.

Python

```
print("\n--- Value counts for 'FuelType' ---")
df['FuelType'].value_counts()
```

- Can also return as a DataFrame for better presentation:

Python

```
print("\n--- Value counts as DataFrame ---")
fuel_type_counts = df['FuelType'].value_counts().to_frame()
fuel_type_counts.index.name = 'FuelType'
fuel_type_counts.columns = ['Count']
fuel_type_counts
```

```
[43]: print("\n--- Value counts as DataFrame ---")
fuel_type_counts = df['FuelType'].value_counts().to_frame()
fuel_type_counts.index.name = 'FuelType'
fuel_type_counts.columns = ['Count']
fuel_type_counts
```

--- Value counts as DataFrame ---

```
[43]:      Count
FuelType
Petrol    8
Diesel    2
```

```
[49]: df.head(10)
```

```
[49]:      Make  Model  Year  Mileage  Price  FuelType
0   Toyota  Camry  2020   50000  25000   Petrol
1   Honda   Civic  2021   30000  22000   Petrol
2    Ford   F-150  2019   70000  35000   Diesel
3    BMW     X5    2022   20000  60000   Petrol
4   Audi    A4    2021   45000  48000   Diesel
5  Mercedes  C-Class  2023   15000  55000   Petrol
6   Nissan  Altima  2020   60000  18000   Petrol
7  Hyundai  Elantra  2022   25000  28000   Petrol
8  Chevrolet  Malibu  2018   85000  15000   Petrol
9 Volkswagen  Jetta  2019   40000  19000   Petrol
```

## 8.2 Visualizing Relationships

Visualizations are powerful for understanding distributions and relationships.

- **Boxplot:**
  - **Purpose:** Displays the distribution of a numerical variable across different categories. Shows median, quartiles, and potential outliers.
  - **Syntax:** `seaborn.boxplot(x=categorical_col, y=numerical_col, data=df)`

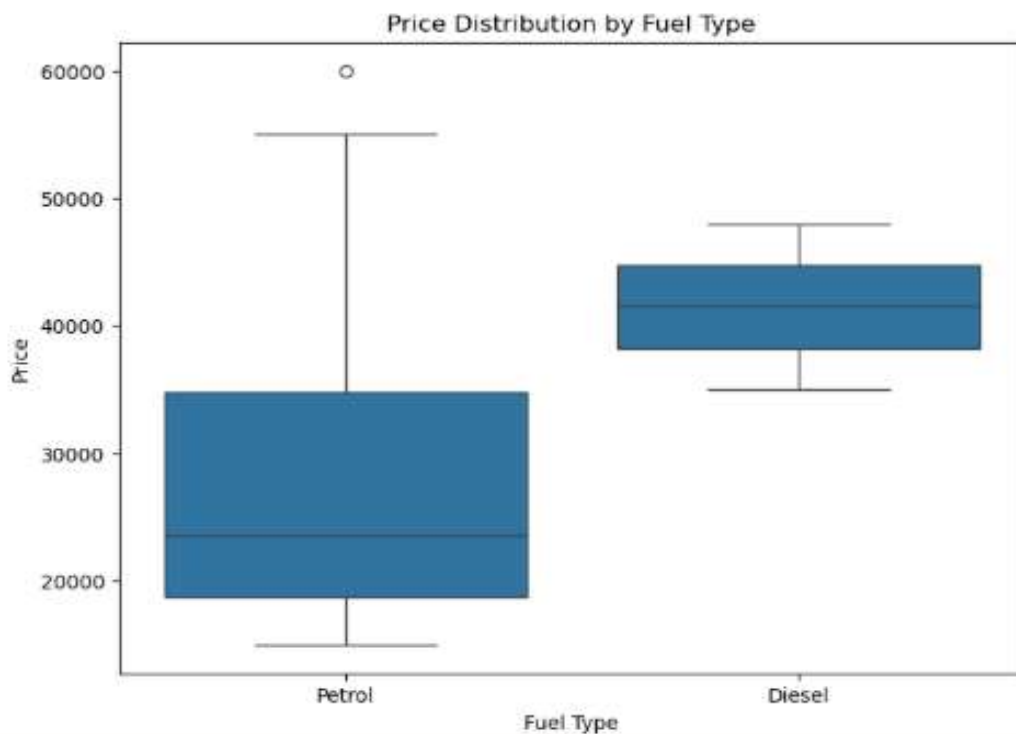
Python

```
# Ensure you have these imports at the top of your notebook or
before this section
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np # Needed if you're using the dummy data
creation
```

```
# --- Your DataFrame 'df' should be loaded ---
df = pd.read_csv('cars.csv')
# To ensure plots appear inline in Jupyter Notebook (if not
already set)
%matplotlib inline
# Now, the plotting code should run without a NameError
```

```
plt.figure(figsize=(8, 6))
sns.boxplot(x='FuelType', y='Price', data=df)
plt.title('Price Distribution by Fuel Type')
plt.xlabel('Fuel Type')
plt.ylabel('Price')
plt.show() # In Jupyter, .show() is often optional but good
practice for clarity
```

```
[10]: import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
path = "C:/Users/k.sanjay/Downloads/cars.csv"
df = pd.read_csv(path)
%matplotlib inline
plt.figure(figsize=(8, 6))
sns.boxplot(x='FuelType', y='Price', data=df)
plt.title('Price Distribution by Fuel Type')
plt.xlabel('Fuel Type')
plt.ylabel('Price')
plt.show()
```



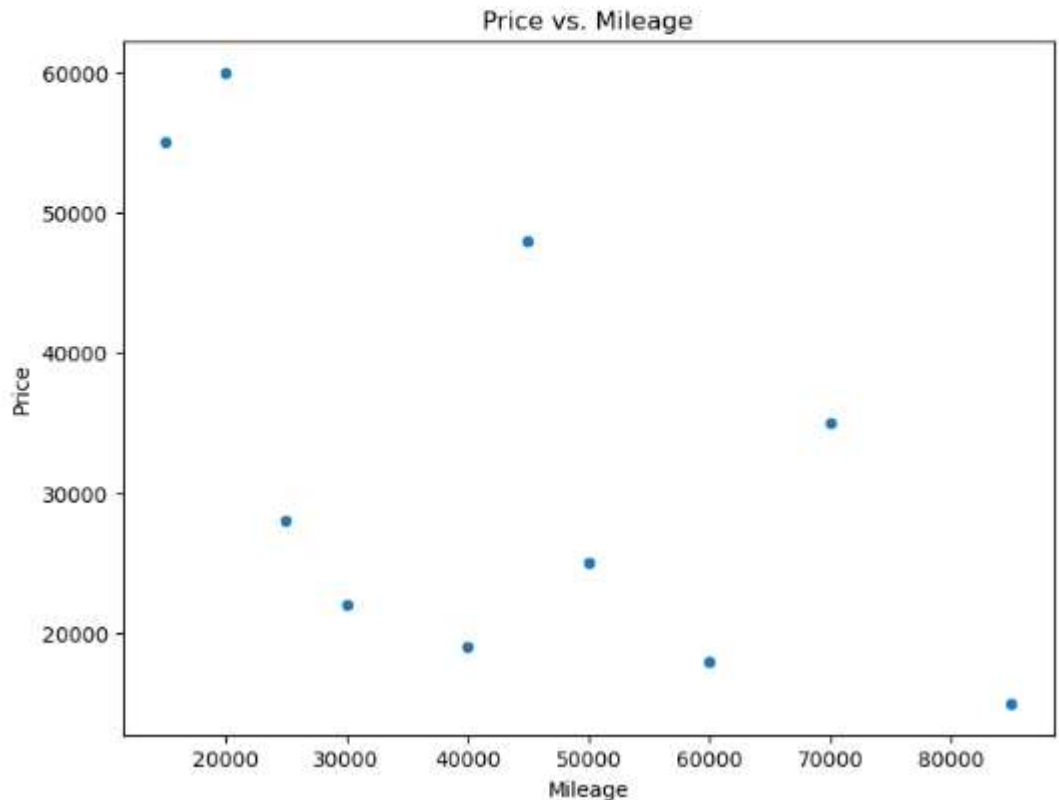
- **Scatterplot:**

- **Purpose:** Visualizes the relationship between two numerical variables. Each point represents an observation.
- **Syntax:** `seaborn.scatterplot(x=numerical_col1, y=numerical_col2, data=df)` OR `matplotlib.pyplot.scatter(x, y)`

Python

```
plt.figure(figsize=(8, 6))
sns.scatterplot(x='Mileage', y='Price', data=df)
plt.title('Price vs. Mileage')
plt.xlabel('Mileage')
plt.ylabel('Price')
plt.show()
```

```
[14]: plt.figure(figsize=(8, 6))
sns.scatterplot(x='Mileage', y='Price', data=df)
plt.title('Price vs. Mileage')
plt.xlabel('Mileage')
plt.ylabel('Price')
plt.show()
```



## Chapter 9: Grouping and Aggregation

Grouping data allows you to perform calculations on subsets of your DataFrame, providing insights into different categories or segments.

### 9.1 Groupby

- **Purpose:** Splitting the data into groups based on some criteria, applying a function to each group independently, and combining the results into a data structure. This is often described as the "split-apply-combine" strategy.
- **Syntax:** `df.groupby('column_name')` followed by an aggregation function (e.g., `mean()`, `sum()`, `count()`, `min()`, `max()`).

Python

```
print("\n--- Grouping by 'FuelType' and calculating mean 'Price' and 'Mileage' ---")
# Group by FuelType and calculate the mean of numerical columns
grouped_data = df.groupby('FuelType').mean(numeric_only=True)
```

grouped\_data

- **Applying Multiple Aggregation Functions (.agg()):** You can apply different aggregation functions to different columns, or multiple functions to the same column.

Python

```
print("\n--- Grouping by 'FuelType' with multiple aggregations ---")
multi_agg = df.groupby('FuelType').agg(
    Average_Price=('Price', 'mean'),
    Max_Mileage=('Mileage', 'max'),
    Num_Cars=('Make', 'count') # Count items in 'Make' column for
    each group
)
multi_agg
```

## 9.2 Pivot Tables

- **Purpose:** A powerful tool for summarizing and reorganizing data, similar to pivot tables in Excel. They create a new table that summarizes data from another table based on selected columns.
- **Syntax:** `df.pivot_table(values=..., index=..., columns=..., aggfunc=...)`
  - o values: Column(s) to aggregate.
  - o index: Column(s) to place on the new table's index (rows).
  - o columns: Column(s) to place on the new table's columns.
  - o aggfunc: Aggregation function (e.g., 'mean', 'sum', 'count'). Default is 'mean'.

Python

```
print("\n--- Pivot Table: Average Price by Fuel Type and Year ---")
# Ensure 'Year' is not a categorical type if you want it as a
numerical index/column in pivot
df['Year'] = df['Year'].astype(int)

pivot_table = df.pivot_table(
    values='Price',
    index='FuelType',
    columns='Year',
    aggfunc='mean'
)
pivot_table
```

```
[16]: print("\n--- Pivot Table: Average Price by Fuel Type and Year ---")
# Ensure 'Year' is not a categorical type if you want it as a numerical index/column in pivot
df['Year'] = df['Year'].astype(int)

pivot_table = df.pivot_table(
    values='Price',
    index='FuelType',
    columns='Year',
    aggfunc='mean'
)
pivot_table
```

--- Pivot Table: Average Price by Fuel Type and Year ---

```
[16]:
```

	Year	2018	2019	2020	2021	2022	2023
FuelType							
Diesel		NaN	35000.0	NaN	48000.0	NaN	NaN
Petrol		15000.0	19000.0	21500.0	22000.0	44000.0	55000.0

---

## Chapter 10: Correlation and Causation

Understanding the relationship between variables is crucial for modeling and insights.

### 10.1 Correlation

- **Definition:** Correlation is a statistical measure that quantifies the extent to which two variables tend to move in tandem. It expresses the strength and direction of a **linear relationship** between them. It's vital to remember that **correlation does NOT imply causation!**
- **Types of Linear Correlation:**
  - **Positive Correlation:** When one variable increases, the other variable also tends to increase. They move in the same direction.
    - *Example:* As the number of hours studied increases, exam scores tend to increase.
  - **Negative Correlation:** When one variable increases, the other variable tends to decrease. They move in opposite directions.
    - *Example:* As the outdoor temperature increases, heating bill costs tend to decrease.
  - **No Correlation (or Zero Correlation):** There is no consistent linear relationship between the two variables. Changes in one variable do not predict changes in the other in a linear fashion.
    - *Example:* A person's height and their favorite color typically have no correlation.
- **Correlation Coefficient (Pearson's r):**
  - The most common measure of linear correlation is the **Pearson product-moment correlation coefficient**, denoted by  $r$ .
  - **Range:** The value of  $r$  always falls between -1 and +1, inclusive.
    - **$r=+1$ :** Indicates a perfect positive linear correlation. All data points lie on a straight line with a positive slope.
    - **$r=-1$ :** Indicates a perfect negative linear correlation. All data points lie on a straight line with a negative slope.
    - **$r=0$ :** Indicates no linear correlation. There is no straight-line relationship between the variables. (Note: this does not mean there's *no* relationship at all, just no *linear* one. There could still be a strong non-linear relationship).
  - **Interpreting Strength:** The closer  $r$  is to +1 or -1, the stronger the linear relationship. The closer  $r$  is to 0, the weaker the linear relationship.
    - **$|r| \approx 0.0$  to  $0.1$ :** Negligible or very weak linear correlation.
    - **$|r| \approx 0.1$  to  $0.3$ :** Weak or low linear correlation.
    - **$|r| \approx 0.3$  to  $0.5$ :** Moderate linear correlation.
    - **$|r| \approx 0.5$  to  $1.0$ :** Strong or high linear correlation. (Note: These ranges are general guidelines and interpretation can depend on the field of study.)
- **Syntax (Pandas):** `df.corr()` calculates the pairwise Pearson correlation between all numerical columns in your DataFrame.

Python



```
print("\n--- Correlation Matrix ---")
# Select only numerical columns for correlation calculation
# This is important because correlation is a numerical measure.
numerical_df = df.select_dtypes(include=['number'])
print(numerical_df.corr())
```

```
[22]: print("\n--- Correlation Matrix ---")
# Select only numerical columns for correlation calculation
# This is important because correlation is a numerical measure.
numerical_df = df.select_dtypes(include=['number'])
print(numerical_df.corr())
```

```
--- Correlation Matrix ---
      Year  Mileage  Price
Year    1.000000 -0.897527  0.708708
Mileage -0.897527  1.000000 -0.582017
Price    0.708708 -0.582017  1.000000
```

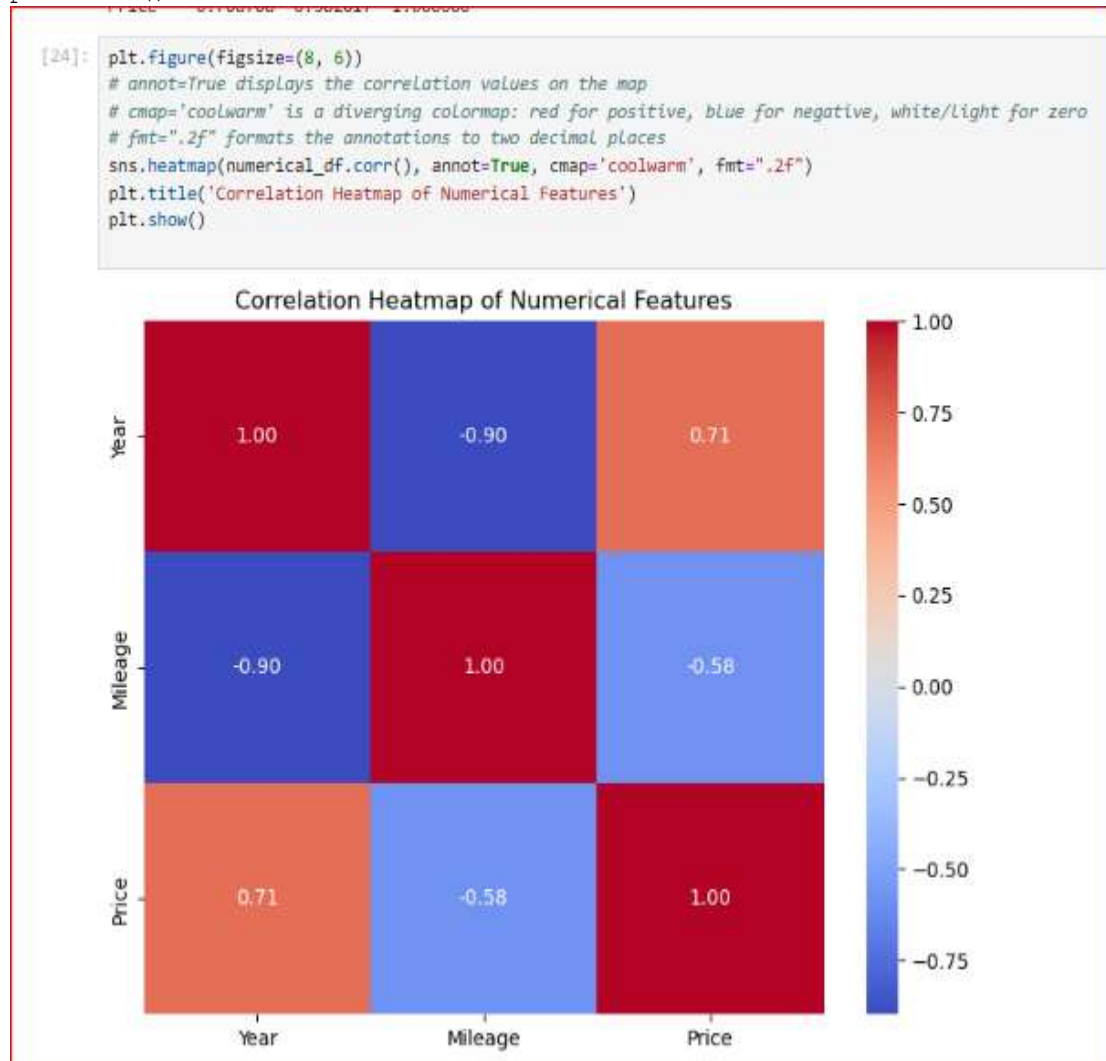
- **Output Interpretation:** The output is a square table (matrix) where both rows and columns represent your numerical variables.
  - The diagonal values are always 1.00, as a variable is perfectly correlated with itself.
  - The matrix is symmetrical: the correlation between A and B is the same as between B and A.

## 10.2 Heat Map (Visualizing Correlation)

- **Purpose:** While the correlation matrix table provides the exact numerical coefficients, a heatmap offers a powerful and intuitive visual representation of these correlations. It uses a color gradient to quickly highlight the strength and direction of relationships, making patterns and strong correlations jump out visually.
- **How to Interpret a Correlation Heatmap:**
  - **Colors:** A color scale (or `cmap`) is used to represent the magnitude and direction of the correlation coefficients.
    - **Warm colors (e.g., reds, oranges):** Typically indicate **positive correlation**. Darker shades of red mean stronger positive correlation (closer to +1).
    - **Cool colors (e.g., blues, purples):** Typically indicate **negative correlation**. Darker shades of blue mean stronger negative correlation (closer to -1).
    - **Neutral color (e.g., white, light grey):** Indicates **weak or no linear correlation** (closer to 0).
  - **Numerical Annotations:** Often, the correlation coefficient itself is displayed within each cell (`annot=True`), providing precise values alongside the visual cue.
  - **Diagonal:** The diagonal from top-left to bottom-right will always be the strongest positive correlation (1.00), representing a variable correlated with itself.
  - **Symmetry:** The heatmap is usually symmetrical along its diagonal, as `corr(A, B)` is the same as `corr(B, A)`.
- **Heat map 1: Basic Heatmap**
  - **Syntax:** `seaborn.heatmap(data)`

## Python

```
plt.figure(figsize=(8, 6))
# annot=True displays the correlation values on the map
# cmap='coolwarm' is a diverging colormap: red for positive, blue for negative, white/light for zero
# fmt=".2f" formats the annotations to two decimal places
sns.heatmap(numerical_df.corr(), annot=True, cmap='coolwarm',
            fmt=".2f")
plt.title('Correlation Heatmap of Numerical Features')
plt.show()
```



- **Heat map 2: Customizing the Heatmap**
  - The example above already includes useful customizations (`annot`, `cmap`, `fmt`). You can explore other `cmap` options (e.g., 'viridis', 'plasma', 'RdBu') to see different color schemes. The choice of `cmap` is important for clarity, especially for diverging data like correlations.

### 10.3 Causation vs. Correlation

- **Crucial Distinction:** This is one of the most fundamental principles in data analysis and statistics. **Correlation does NOT imply causation.**

- Just because two variables move together (are correlated) doesn't mean that one directly causes the other to change. There are several reasons why correlation might exist without causation:
  1. **Third Variable (Confounding Factor):** A hidden or unobserved variable might be influencing both correlated variables.
    - *Example:* Ice cream sales and drowning incidents often increase simultaneously during summer. They are positively correlated. However, ice cream sales do not cause drownings. The third variable, **temperature**, causes both.
  2. **Reverse Causation:** It's possible that Y causes X, instead of X causing Y.
  3. **Coincidence:** The correlation might be purely by chance, especially in large datasets.
- **To establish causation, you generally need:**
  - **Controlled Experiments:** (e.g., A/B testing) where you manipulate one variable and observe the effect on another while holding other factors constant.
  - **Strong theoretical backing and expert knowledge** in the domain.
  - **Advanced statistical methods** designed to infer causal relationships (beyond the scope of this basic module).

Always be cautious when interpreting correlations. They are excellent for identifying potential relationships that warrant further investigation, but they rarely provide definitive proof of cause and effect.

---

## Chapter 11: Introduction to Regression Analysis

Regression analysis is a statistical method used to model the relationship between a dependent variable (what you want to predict) and one or more independent variables (predictors).

### 11.1 What is Regression?

- **Purpose:** To understand how the value of the dependent variable changes when any one of the independent variables is varied, while the other independent variables are held fixed. It's often used for forecasting and prediction.
- **Dependent Variable (Y):** The outcome or response variable (e.g., Car Price).
- **Independent Variables (X):** The predictor or explanatory variables (e.g., Mileage, Year, Make).

### 11.2 Simple Linear Regression

- **Definition:** Models the linear relationship between one dependent variable (Y) and one independent variable (X).
- **Equation:**  $Y = b_0 + b_1X + \epsilon$ 
  - Y: Dependent variable
  - X: Independent variable
  - $b_0$ : Y-intercept (the value of Y when X is 0)
  - $b_1$ : Slope (the change in Y for a one-unit change in X)

- epsilon: Error term (represents the irreducible error not explained by the model)
- **Assumptions (Key ones):**
  - **Linearity:** The relationship between X and Y is linear.
  - **Independence of Errors:** Residuals (errors) are independent of each other.
  - **Homoscedasticity:** The variance of the residuals is constant across all levels of X.
  - **Normality of Errors:** Residuals are normally distributed.

### 11.3 Multiple Linear Regression

- **Definition:** Models the linear relationship between one dependent variable (Y) and *two or more* independent variables ( $X_1, X_2, \dots, X_n$ ).
- **Equation:**  $Y = b_0 + b_1X_1 + b_2X_2 + \dots + b_nX_n + \text{epsilon}$ 
  - $b_0$ : Y-intercept
  - $b_i$ : Coefficients (slopes) for each independent variable  $X_i$ , representing the change in Y for a one-unit change in  $X_i$ , holding other X's constant.
- **Assumptions:** Same as Simple Linear Regression, plus:
  - **No Multicollinearity:** Independent variables should not be too highly correlated with each other.

### 11.4 ANOVA (Analysis of Variance)

- **Purpose:** ANOVA is a statistical test used to determine whether there are any statistically significant differences between the means of two or more independent (unrelated) groups.
- **When it's Used:**
  - Often used to assess the significance of categorical independent variables in predicting a numerical dependent variable.
  - For example, comparing the mean prices of cars from different 'Makes' to see if there's a significant difference.
- **Hypotheses:**
  - **Null Hypothesis ( $H_0$ ):** The means of all groups are equal.
  - **Alternative Hypothesis ( $H_1$ ):** At least one group mean is different from the others.
- **F-statistic:** ANOVA produces an F-statistic and a p-value. A low p-value (typically < 0.05) indicates that you can reject the null hypothesis, suggesting a significant difference between group means.
- **Conceptual Overview:** ANOVA works by partitioning the total variance in the data into different components (variance between groups and variance within groups) and comparing them.
- **Implementation (using `scipy.stats` or `statsmodels`):**

Python

```
from scipy import stats
import statsmodels.api as sm
from statsmodels.formula.api import ols

# Let's perform ANOVA to see if 'FuelType' has a significant effect
on 'Price'
```

```
# First, ensure 'FuelType' is treated as a categorical variable
df['FuelType'] = df['FuelType'].astype('category')

# Create a formula for OLS (Ordinary Least Squares) model
# 'C(FuelType)' tells statsmodels to treat 'FuelType' as a
categorical variable
model = ols('Price ~ C(FuelType)', data=df).fit()

# Perform ANOVA
anova_table = sm.stats.anova_lm(model, typ=2) # typ=2 for Type II sum
of squares
print("\n--- ANOVA Table (Price by FuelType) ---")
print(anova_table)

# Interpretation: Look at the 'PR(>F)' column (p-value) for
'C(FuelType)'.
# If p-value < 0.05, there's a statistically significant difference
in mean prices across fuel types.
```

```
[26]: from scipy import stats
import statsmodels.api as sm
from statsmodels.formula.api import ols

# Let's perform ANOVA to see if 'FuelType' has a significant effect on 'Price'
# First, ensure 'FuelType' is treated as a categorical variable
df['FuelType'] = df['FuelType'].astype('category')

# Create a formula for OLS (Ordinary Least Squares) model
# 'C(FuelType)' tells statsmodels to treat 'FuelType' as a categorical variable
model = ols('Price ~ C(FuelType)', data=df).fit()

# Perform ANOVA
anova_table = sm.stats.anova_lm(model, typ=2) # typ=2 for Type II sum of squares
print("\n--- ANOVA Table (Price by FuelType) ---")
print(anova_table)

# Interpretation: Look at the 'PR(>F)' column (p-value) for 'C(FuelType)'.
# If p-value < 0.05, there's a statistically significant difference in mean prices across fuel types.
```

```
--- ANOVA Table (Price by FuelType) ---
              sum_sq  df      F    PR(>F)
C(FuelType)  2.025000e+08  1.0  0.739051  0.414982
Residual    2.192000e+09  8.0      NaN      NaN
```

## Chapter 12: Implementing Regression in Python

We'll use the `scikit-learn` library to build our regression models.

### 12.1 Simple Linear Regression using Python

Let's predict Price using Mileage.

Python

```
from sklearn.linear_model import LinearRegression
# from sklearn.model_selection import train_test_split # Often used, but
for simplicity, we'll use all data for now
```

```

# Assume df is our DataFrame from previous steps
# Define independent variable (X) and dependent variable (Y)
X = df[['Mileage']] # X must be a 2D array (DataFrame with one column)
Y = df['Price']      # Y can be a 1D Series

# Create a Linear Regression model object
lm_simple = LinearRegression()

# Train the model using the .fit() method
lm_simple.fit(X, Y)

# Get the coefficients
b0 = lm_simple.intercept_ # Intercept
b1 = lm_simple.coef_[0]   # Coefficient for Mileage

print(f"\n--- Simple Linear Regression Model (Price ~ Mileage) ---")
print(f"Equation: Price = {b0:.2f} + ({b1:.2f} * Mileage)")
print(f"Intercept (b0): {b0:.2f}")
print(f"Coefficient for Mileage (b1): {b1:.2f}")

# Make predictions
predictions_simple = lm_simple.predict(X)
print(f"First 5 predictions: {np.round(predictions_simple[:5], 2)}")
print(f"Actual first 5 prices: {Y.head().values}")

```

```

[28]: from sklearn.linear_model import LinearRegression
# from sklearn.model_selection import train_test_split # Often used, but for simplicity, we'll use all data for now

# Assume df is our DataFrame from previous steps
# Define independent variable (X) and dependent variable (Y)
X = df[['Mileage']] # X must be a 2D array (DataFrame with one column)
Y = df['Price']     # Y can be a 1D Series

# Create a Linear Regression model object
lm_simple = LinearRegression()

# Train the model using the .fit() method
lm_simple.fit(X, Y)

# Get the coefficients
b0 = lm_simple.intercept_ # Intercept
b1 = lm_simple.coef_[0]   # Coefficient for Mileage

print(f"\n--- Simple Linear Regression Model (Price ~ Mileage) ---")
print(f"Equation: Price = {b0:.2f} + ({b1:.2f} * Mileage)")
print(f"Intercept (b0): {b0:.2f}")
print(f"Coefficient for Mileage (b1): {b1:.2f}")

# Make predictions
predictions_simple = lm_simple.predict(X)
print(f"First 5 predictions: {np.round(predictions_simple[:5], 2)}")
print(f"Actual first 5 prices: {Y.head().values}")

```

```

--- Simple Linear Regression Model (Price ~ Mileage) ---
Equation: Price = 50896.55 + (-0.42 * Mileage)
Intercept (b0): 50896.55
Coefficient for Mileage (b1): -0.42
First 5 predictions: [29991.38 38353.45 21629.31 42534.48 32081.9 ]
Actual first 5 prices: [25000 22000 35000 60000 48000]

```

```
[ ]:
```



## 12.2 Multiple Linear Regression using Python

Now, let's predict Price using Mileage and Year.

Python

```
# Define independent variables (X) and dependent variable (Y)
X_multi = df[['Mileage', 'Year']] # X is a DataFrame with multiple columns
Y_multi = df['Price']

# Create a Multiple Linear Regression model object
lm_multi = LinearRegression()

# Train the model
lm_multi.fit(X_multi, Y_multi)

# Get coefficients
b0_multi = lm_multi.intercept_
b_mileage = lm_multi.coef_[0]
b_year = lm_multi.coef_[1]

print(f"\n--- Multiple Linear Regression Model (Price ~ Mileage + Year) ---")
print(f"Equation: Price = {b0_multi:.2f} + ({b_mileage:.2f} * Mileage) + ({b_year:.2f} * Year)")
print(f"Intercept (b0): {b0_multi:.2f}")
print(f"Coefficient for Mileage: {b_mileage:.2f}")
print(f"Coefficient for Year: {b_year:.2f}")

# Make predictions
predictions_multi = lm_multi.predict(X_multi)
print(f"First 5 predictions: {np.round(predictions_multi[:5], 2)}")
```

```
[30]: # Define independent variables (X) and dependent variable (Y)
X_multi = df[['Mileage', 'Year']] # X is a DataFrame with multiple columns
Y_multi = df['Price']

# Create a Multiple Linear Regression model object
lm_multi = LinearRegression()

# Train the model
lm_multi.fit(X_multi, Y_multi)

# Get coefficients
b0_multi = lm_multi.intercept_
b_mileage = lm_multi.coef_[0]
b_year = lm_multi.coef_[1]

print(f"\n--- Multiple Linear Regression Model (Price ~ Mileage + Year) ---")
print(f"Equation: Price = {b0_multi:.2f} + ({b_mileage:.2f} * Mileage) + ({b_year:.2f} * Year)")
print(f"Intercept (b0): {b0_multi:.2f}")
print(f"Coefficient for Mileage: {b_mileage:.2f}")
print(f"Coefficient for Year: {b_year:.2f}")

# Make predictions
predictions_multi = lm_multi.predict(X_multi)
print(f"First 5 predictions: {np.round(predictions_multi[:5], 2)}")

--- Multiple Linear Regression Model (Price ~ Mileage + Year) ---
Equation: Price = -19950374.88 + (0.20 * Mileage) + (9885.71 * Year)
Intercept (b0): -19950374.88
Coefficient for Mileage: 0.20
Coefficient for Year: 9885.71
First 5 predictions: [28755.67 34646.31 22865.02 42534.48 37642.61]
```



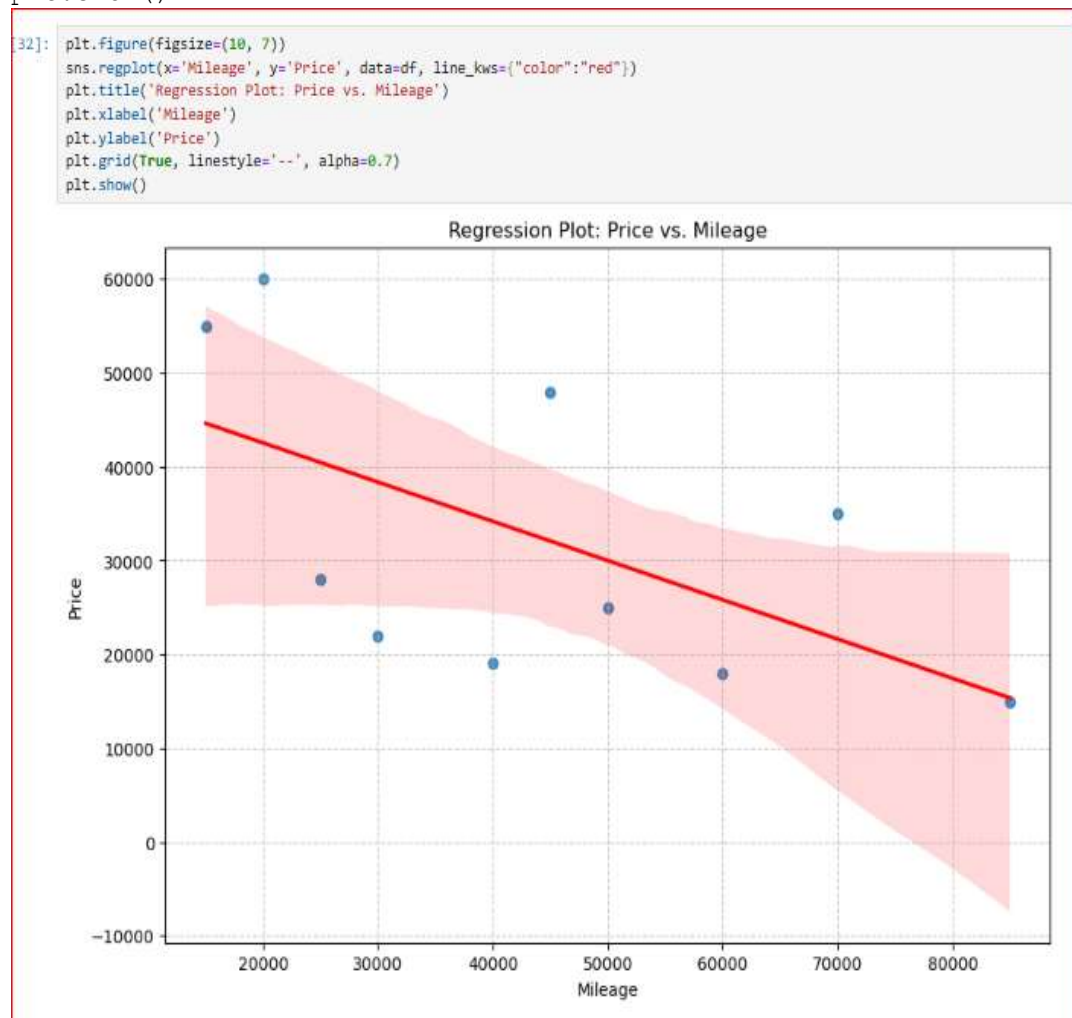
## 12.3 Regression Plot

A regression plot (or `regplot`) combines a scatterplot with a linear regression model fit.

- **Syntax:** `seaborn.regplot(x=independent_var, y=dependent_var, data=df)`

Python

```
plt.figure(figsize=(10, 7))
sns.regplot(x='Mileage', y='Price', data=df,
            line_kws={"color": "red"})
plt.title('Regression Plot: Price vs. Mileage')
plt.xlabel('Mileage')
plt.ylabel('Price')
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()
```



## 12.4 Residual Plot

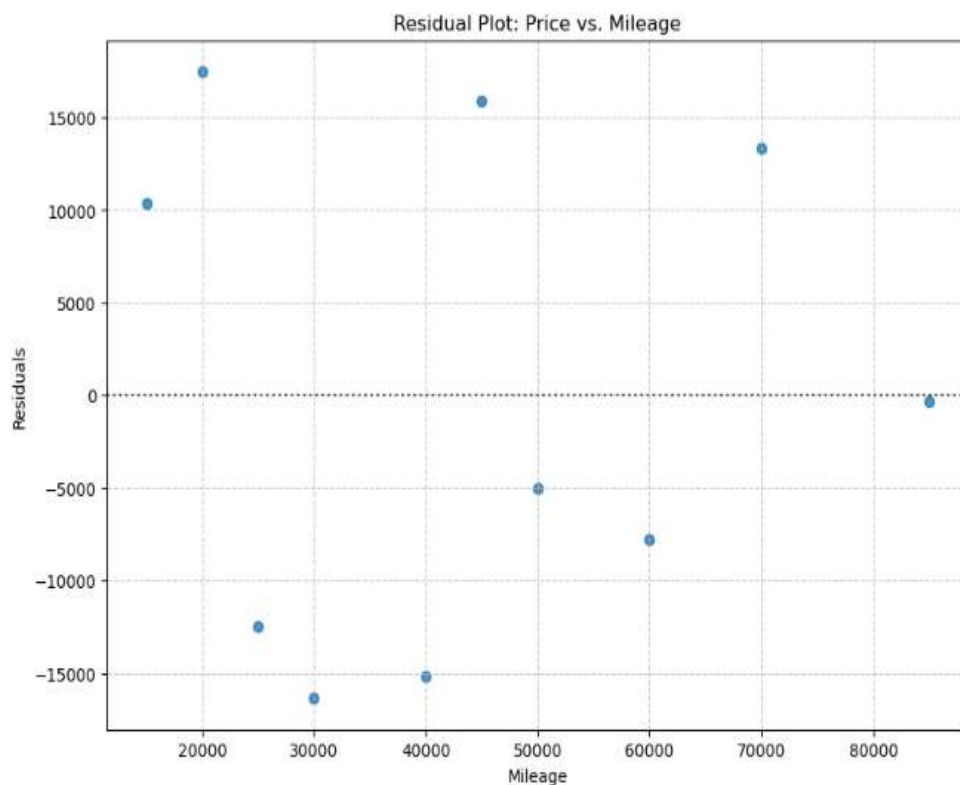
A residual plot shows the residuals (the difference between actual and predicted values) on the y-axis and the independent variable (or predicted values) on the x-axis.

- **Purpose:** To check the linearity, homoscedasticity, and independence of errors assumptions. Ideally, residuals should be randomly scattered around zero with no discernible pattern.
- **Syntax:** `seaborn.residplot(x=independent_var, y=dependent_var, data=df)`

Python

```
plt.figure(figsize=(10, 7))
sns.residplot(x='Mileage', y='Price', data=df)
plt.title('Residual Plot: Price vs. Mileage')
plt.xlabel('Mileage')
plt.ylabel('Residuals')
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()
```

```
[34]: plt.figure(figsize=(10, 7))
      sns.residplot(x='Mileage', y='Price', data=df)
      plt.title('Residual Plot: Price vs. Mileage')
      plt.xlabel('Mileage')
      plt.ylabel('Residuals')
      plt.grid(True, linestyle='--', alpha=0.7)
      plt.show()
```



## 12.5 Distribution Plot

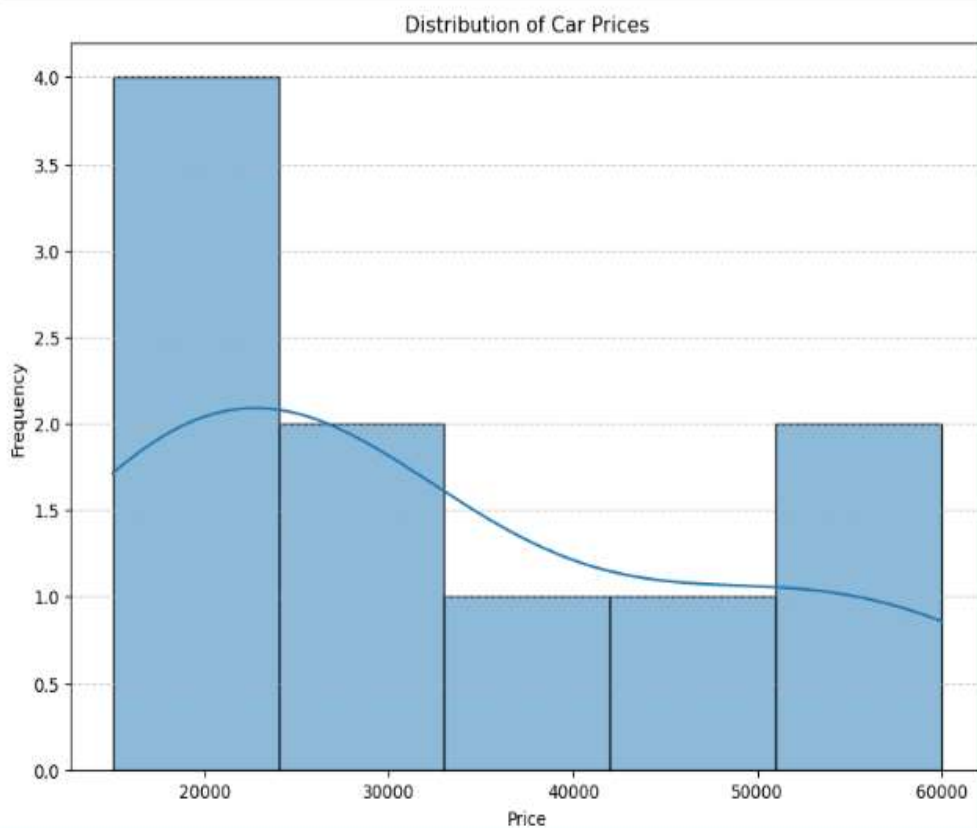
Distribution plots help visualize the probability distribution of a single variable.

- **Distribution Plot 1: Histogram/KDE Plot for a variable**
  - **Purpose:** Shows the frequency distribution of a numerical variable.
  - **Syntax:** `seaborn.histplot(data=df, x=column_name, kde=True)` or `seaborn.kdeplot(data=df, x=column_name)`

## Python

```
plt.figure(figsize=(10, 7))
sns.histplot(df['Price'], kde=True, bins=5) # bins argument for histogram bars
plt.title('Distribution of Car Prices')
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

```
[36]: plt.figure(figsize=(10, 7))
      sns.histplot(df['Price'], kde=True, bins=5) # bins argument for histogram bars
      plt.title('Distribution of Car Prices')
      plt.xlabel('Price')
      plt.ylabel('Frequency')
      plt.grid(axis='y', linestyle='--', alpha=0.7)
      plt.show()
```



- **Distribution Plot 2: Distribution Plot for Residuals**

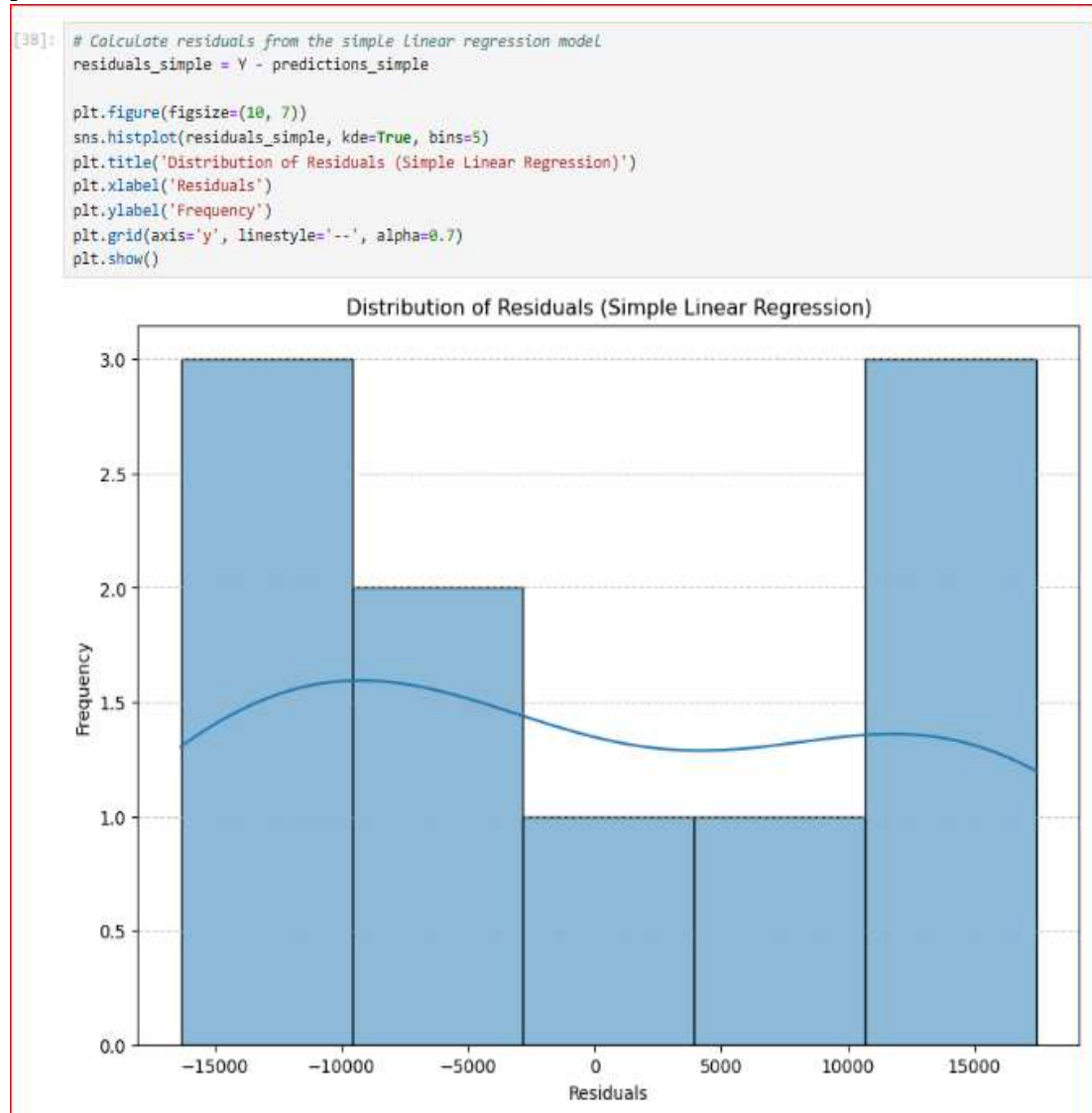
- **Purpose:** To check if the residuals of a regression model are normally distributed (a key assumption).
- Calculate residuals:  $\text{residuals} = Y - \text{predictions}$ .
- Plot residuals.

## Python

```
# Calculate residuals from the simple linear regression model
residuals_simple = Y - predictions_simple

plt.figure(figsize=(10, 7))
```

```
sns.histplot(residuals_simple, kde=True, bins=5)
plt.title('Distribution of Residuals (Simple Linear Regression)')
plt.xlabel('Residuals')
plt.ylabel('Frequency')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```



## 12.6 Polynomial Regression

- **Polynomial Regression 1: When Linear isn't Enough**
  - Sometimes, the relationship between variables is not strictly linear. Polynomial regression models the relationship as an nth degree polynomial.
  - Equation (e.g., 2nd degree):  $Y = b_0 + b_1X + b_2X^2 + \epsilon$
  - This is still a form of linear regression because it's linear in the coefficients ( $b_0, b_1, b_2$ ). We transform the independent variable(s) into polynomial features.
- **Polynomial Regression 2: Implementation in Python**
  - Use `PolynomialFeatures` from `sklearn.preprocessing` to create polynomial terms.
  - Then, apply `LinearRegression` on these transformed features.

## Python

```
from sklearn.preprocessing import PolynomialFeatures

# Assuming 'Mileage' has a non-linear relationship with 'Price'
X_poly = df[['Mileage']]
Y_poly = df['Price']

# Create polynomial features (e.g., degree 2)
poly = PolynomialFeatures(degree=2)
X_poly_transformed = poly.fit_transform(X_poly)

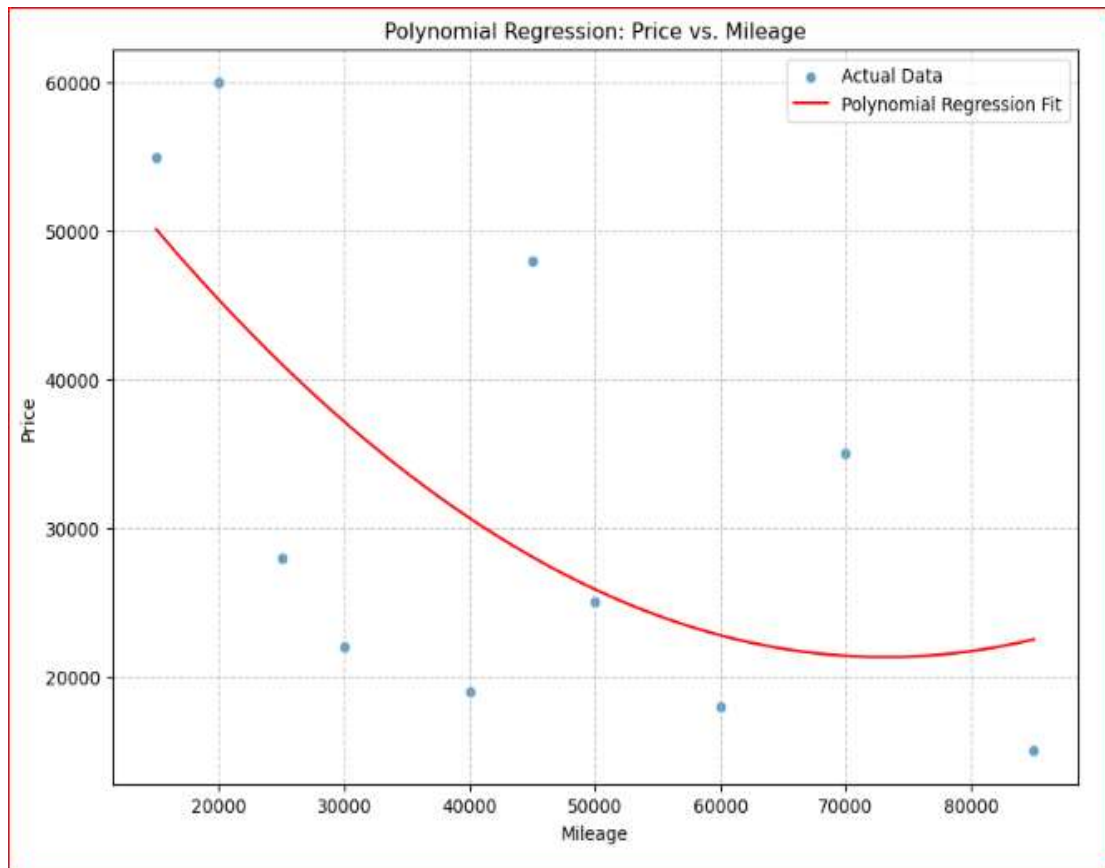
# Train a Linear Regression model on the transformed features
poly_lm = LinearRegression()
poly_lm.fit(X_poly_transformed, Y_poly)

# Make predictions
predictions_poly = poly_lm.predict(X_poly_transformed)

print(f"\n--- Polynomial Regression Model (Price ~ Mileage +
Mileage^2) ---")
# The coefficients correspond to the constant (b0), X (b1), X^2 (b2),
etc.
# The first coefficient is for the constant term (intercept), which
is handled by poly_lm.intercept_
# The remaining coefficients are for the polynomial features
print(f"Intercept: {poly_lm.intercept_:.2f}")
print(f"Coefficients (for X, X^2, ...): {np.round(poly_lm.coef_[1:],
2)}") # Exclude the first coef (for constant term)

# Plotting Polynomial Regression (requires sorting for a smooth
curve)
# Create new data points for plotting the curve
# Ensure X_plot covers the full range of X_poly
X_plot = np.linspace(X_poly.min(), X_poly.max(), 100).reshape(-1, 1)
X_plot_transformed = poly.transform(X_plot)
Y_plot_predictions = poly_lm.predict(X_plot_transformed)

plt.figure(figsize=(10, 7))
sns.scatterplot(x='Mileage', y='Price', data=df, label='Actual Data',
alpha=0.7)
plt.plot(X_plot, Y_plot_predictions, color='red', label='Polynomial
Regression Fit')
plt.title('Polynomial Regression: Price vs. Mileage')
plt.xlabel('Mileage')
plt.ylabel('Price')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()
```



## Chapter 13: Model Evaluation

After building a regression model, it's crucial to evaluate its performance.

### 13.1 In-Sample Evaluation

- **Definition:** Evaluating how well the model fits the data it was trained on. This gives an idea of how well the model has learned the patterns in the existing data.
- **Caution:** High in-sample evaluation metrics don't guarantee good performance on unseen data (can indicate overfitting). For robust evaluation, one uses train-test split (which is typically introduced after these metrics).

### 13.2 Mean Squared Error (MSE)

- **Definition:** The average of the squares of the errors (residuals). It measures the average squared difference between the estimated values and the actual value.
- **Interpretation:**
  - A lower MSE indicates a better fit.
  - It's sensitive to outliers because errors are squared.
  - The unit of MSE is the square of the unit of the dependent variable.
- **Syntax:** `sklearn.metrics.mean_squared_error(y_true, y_pred)`

Python

```

from sklearn.metrics import mean_squared_error

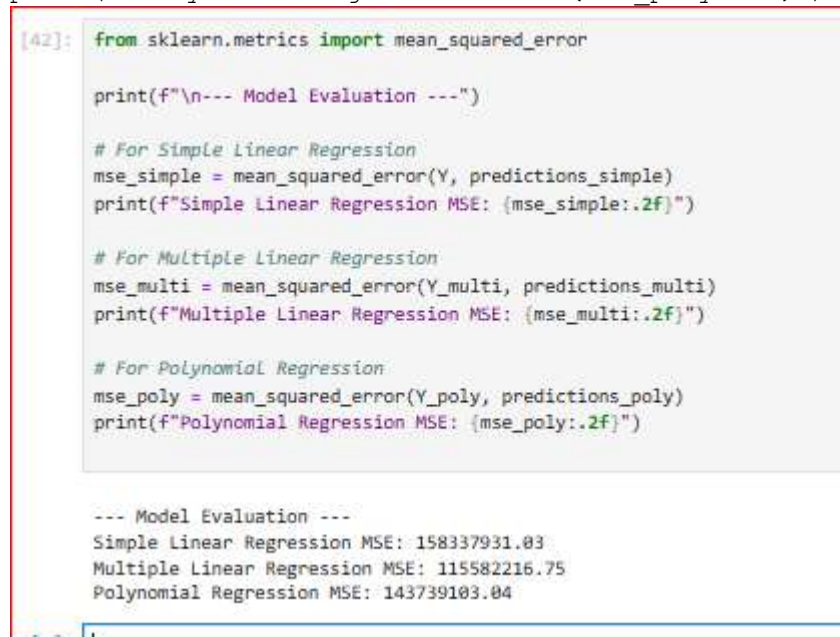
print(f"\n--- Model Evaluation ---")

# For Simple Linear Regression
mse_simple = mean_squared_error(Y, predictions_simple)
print(f"Simple Linear Regression MSE: {mse_simple:.2f}")

# For Multiple Linear Regression
mse_multi = mean_squared_error(Y_multi, predictions_multi)
print(f"Multiple Linear Regression MSE: {mse_multi:.2f}")

# For Polynomial Regression
mse_poly = mean_squared_error(Y_poly, predictions_poly)
print(f"Polynomial Regression MSE: {mse_poly:.2f}")

```



```

[42]: from sklearn.metrics import mean_squared_error

print(f"\n--- Model Evaluation ---")

# For Simple Linear Regression
mse_simple = mean_squared_error(Y, predictions_simple)
print(f"Simple Linear Regression MSE: {mse_simple:.2f}")

# For Multiple Linear Regression
mse_multi = mean_squared_error(Y_multi, predictions_multi)
print(f"Multiple Linear Regression MSE: {mse_multi:.2f}")

# For Polynomial Regression
mse_poly = mean_squared_error(Y_poly, predictions_poly)
print(f"Polynomial Regression MSE: {mse_poly:.2f}")

--- Model Evaluation ---
Simple Linear Regression MSE: 158337931.03
Multiple Linear Regression MSE: 115582216.75
Polynomial Regression MSE: 143739103.04

```

### 13.3 R-squared ( $R^2$ )

- **R Squared 1: Definition**
  - Also known as the coefficient of determination.
  - It's a statistical measure that represents the proportion of the variance in the dependent variable that is predictable from the independent variable(s).
  - It indicates how well the model fits the observed data.
- **R Squared 2: Calculation and Interpretation**
  - **Range:** R-squared typically ranges from 0 to 1 (can be negative for poor models or non-linear regression, but usually interpreted in the 0-1 range).
  - **Interpretation:**
    - **0:** The model explains none of the variability of the response data around its mean.
    - **1:** The model explains all the variability of the response data around its mean.
    - Higher R-squared values generally indicate a better fit (a larger proportion of variance explained).
  - **Syntax:** `sklearn.metrics.r2_score(y_true, y_pred)` or `model.score(X, Y)` (for LinearRegression objects).



## Python

```
from sklearn.metrics import r2_score

# For Simple Linear Regression
r2_simple = r2_score(Y, predictions_simple)
print(f"Simple Linear Regression R-squared: {r2_simple:.2f}")

# For Multiple Linear Regression
r2_multi = r2_score(Y_multi, predictions_multi)
print(f"Multiple Linear Regression R-squared: {r2_multi:.2f}")

# Using model.score() method directly for R-squared (alternative for
scikit-learn models)
r2_simple_score_method = lm_simple.score(X, Y)
print(f"Simple Linear Regression R-squared (model.score()):
{r2_simple_score_method:.2f}")

# For Polynomial Regression
r2_poly = r2_score(Y_poly, predictions_poly)
print(f"Polynomial Regression R-squared: {r2_poly:.2f}")
```

```
[44]: from sklearn.metrics import r2_score

# For Simple Linear Regression
r2_simple = r2_score(Y, predictions_simple)
print(f"Simple Linear Regression R-squared: {r2_simple:.2f}")

# For Multiple Linear Regression
r2_multi = r2_score(Y_multi, predictions_multi)
print(f"Multiple Linear Regression R-squared: {r2_multi:.2f}")

# Using model.score() method directly for R-squared (alternative for scikit-learn models)
r2_simple_score_method = lm_simple.score(X, Y)
print(f"Simple Linear Regression R-squared (model.score()): {r2_simple_score_method:.2f}")

# For Polynomial Regression
r2_poly = r2_score(Y_poly, predictions_poly)
print(f"Polynomial Regression R-squared: {r2_poly:.2f}")

Simple Linear Regression R-squared: 0.34
Multiple Linear Regression R-squared: 0.52
Simple Linear Regression R-squared (model.score()): 0.34
Polynomial Regression R-squared: 0.40
```

---

## Appendix: Accessing Data for Analysis: Popular Websites

For practical data analysis, you'll need datasets. Here are some widely used platforms for accessing them:

### 1. Kaggle.com

- **What it is:** Kaggle is the world's largest online community of data scientists and machine learning engineers. It's a fantastic resource for learning, competing, and finding data.
- **What it offers:**
  - **Datasets:** A vast repository of publicly available datasets covering almost any topic imaginable, uploaded by users, organizations, or as part of competitions.

You can find datasets on anything from movie ratings to economic indicators, health data, and more.

- **Competitions:** Kaggle is famous for its machine learning competitions, where data scientists compete to build the best models for a given problem. These competitions often come with rich datasets.
- **Code (Notebooks/Kernels):** Users can share their analytical code (often in Jupyter Notebooks, called "Kaggle Kernels" or "Kaggle Notebooks") directly on the platform, providing excellent examples and learning opportunities.
- **How to access data:**
  - **Direct Download:** Most datasets can be downloaded directly as CSV, JSON, or other file formats.
  - **Kaggle API:** For programmatic access, especially if you're participating in competitions or frequently downloading data, you can use the Kaggle API.
- **Good for:** Practicing with diverse real-world data, seeing how others approach problems, and finding inspiration.

## 2. Google Cloud Public Datasets

- **What it is:** Google Cloud Public Datasets is a program that hosts a variety of interesting datasets that are publicly available for analysis and integrated with Google Cloud Platform services.
- **What it offers:**
  - **Large-Scale Data:** Often includes very large datasets that might be challenging to process on a local machine without cloud resources (e.g., genomic data, weather data, public traffic data, open-source code).
  - **Integration with BigQuery:** Many datasets are hosted in Google BigQuery, a serverless, highly scalable, and cost-effective cloud data warehouse. This allows you to query massive datasets using SQL directly in the cloud, often without needing to download them.
- **How to access data:**
  - **BigQuery:** The primary way to access most datasets here is via Google BigQuery, using SQL queries. You'll need a Google Cloud account (they offer free tiers/credits).
  - **Download:** Some smaller datasets might offer direct download options.
- **Good for:** Exploring very large datasets, learning cloud data warehousing concepts, and working with specialized data types.

## 3. data.world

- **What it is:** data.world describes itself as the most meaningful place to work with data, offering a platform for data collaboration.
- **What it offers:**
  - **Community and Collaboration:** It's built around the idea of a social network for data, allowing users to discover, share, and collaborate on data projects.
  - **Diverse Datasets:** Hosts a wide variety of datasets from individuals, organizations, and governments (e.g., open data initiatives).
  - **Tools for Data Projects:** Provides features for version control, data lineage, and even running SQL queries directly on hosted datasets.
- **How to access data:**
  - **Download:** Datasets can be downloaded in various formats.

- **API:** data.world provides an API for programmatic access.
  - **Direct Integration:** They also offer integrations with popular tools and languages, allowing you to connect directly to datasets.
- **Good for:** Discovering curated datasets, engaging with a data community, and seeing how data projects are structured and shared.