**EXP 1: Basics of for Machine Learning**

1. Introduction to

is a versatile programming language widely used in data science and machine learning due to its simplicity and powerful libraries.

2. Basics

Variables and Data Types:

# Integer

a = 5

# Float

b = 3.14

# String

c = "Hello, World!"

# Boolean

d = True

Lists:

# List of integers

numbers = [1, 2, 3, 4, 5]

# List of strings

fruits = ["apple", "banana", "cherry"]

Dictionaries:

# Dictionary with string keys and integer values

ages = {"Alice": 25, "Bob": 30}

Functions:

def greet(name):

return f"Hello, {name}!"

print(greet("Alice"))

3. Libraries for Machine Learning

NumPy: NumPy is essential for numerical computations.

import numpy as np

# Create an array

arr = np.array([1, 2, 3, 4, 5])

print(arr)

Pandas: Pandas is used for data manipulation and analysis.

import pandas as pd

# Create a DataFrame

data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30]}

df = pd.DataFrame(data)

print(df)

Matplotlib: Matplotlib is for data visualization.

import matplotlib.pyplot as plt

# Plot a graph

plt.plot([1, 2, 3, 4, 5], [1, 4, 9, 16, 25])

plt.show()

Scikit-Learn: Scikit-Learn is a library for machine learning.

from sklearn.linear\_model import LinearRegression

# Create a linear regression model

model = LinearRegression()

4. Basic Machine Learning Workflow

Import Libraries:

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error

Load Dataset:

df = pd.read\_csv('data.csv')

Prepare Data:

X = df[['feature1', 'feature2']]

y = df['target']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

Train Model:

model = LinearRegression()

model.fit(X\_train, y\_train)

Make Predictions:

y\_pred = model.predict(X\_test)

Evaluate Model:

mse = mean\_squared\_error(y\_test, y\_pred)

print(f'Mean Squared Error: {mse}')

5. Example: Simple Linear Regression

Data Preparation:

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error

# Sample data

data = {

'Hours': [1, 2, 3, 4, 5],

'Scores': [10, 20, 30, 40, 50]

}

# Create DataFrame

df = pd.DataFrame(data)

# Features and target

X = df[['Hours']]

y = df['Scores']

# Split data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

Model Training:

# Initialize and train model

model = LinearRegression()

model.fit(X\_train, y\_train)

Prediction and Evaluation:

# Make predictions

y\_pred = model.predict(X\_test)

# Evaluate model

mse = mean\_squared\_error(y\_test, y\_pred)

print(f'Mean Squared Error: {mse}')

This should give you a good start on understanding and writing basic machine learning algorithms.

**Basics of Pandas**

Pandas is a powerful and easy-to-use open-source data analysis and manipulation library for Python. It is built on top of NumPy and provides data structures and operations for manipulating numerical tables and time series.

1. Introduction to Pandas

To get started with Pandas, you need to install it (if you haven't already) and import it in your Python script or notebook.

python

!pip install pandas

import pandas as pd

2. Data Structures in Pandas

Pandas primarily uses two data structures: Series and DataFrame.

**Series**:

A Series is a one-dimensional labeled array capable of holding any data type.

python

import pandas as pd

# Creating a Series

s = pd.Series([1, 3, 5, np.nan, 6, 8])

print(s)

DataFrame:

A DataFrame is a two-dimensional labeled data structure with columns of potentially different types.

python

# Creating a DataFrame

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],

'Age': [24, 27, 22, 32],

'City': ['New York', 'Los Angeles', 'Chicago', 'Houston']}

df = pd.DataFrame(data)

print(df)

3. Reading and Writing Data

Pandas can read data from various file formats such as CSV, Excel, and SQL databases.

Reading from a CSV file:

python

# Read data from a CSV file

df = pd.read\_csv('data.csv')

print(df.head())

Writing to a CSV file:

python

# Write data to a CSV file

df.to\_csv('output.csv', index=False)

4. Basic Data Operations

Viewing Data:

python

# Display the first few rows

print(df.head())

# Display the last few rows

print(df.tail())

Getting Data Information:

python

# Data types of each column

print(df.dtypes)

# Summary statistics

print(df.describe())

# Number of rows and columns

print(df.shape)

# Column names

print(df.columns)

Selecting Data:

python

# Select a single column

print(df['Name'])

# Select multiple columns

print(df[['Name', 'City']])

# Select rows by index

print(df.iloc[0:2])

# Select rows by label

print(df.loc[0:2])

Filtering Data:

python

# Filter rows based on a condition

print(df[df['Age'] > 25])

5. Modifying Data

Adding a New Column:

python

# Add a new column

df['Salary'] = [50000, 60000, 45000, 70000]

print(df)

Updating Values:

python

# Update values in a column

df['Age'] = df['Age'] + 1

print(df)

Dropping Columns/Rows:

python

# Drop a column

df = df.drop('Salary', axis=1)

print(df)

# Drop a row

df = df.drop(0, axis=0)

print(df)

Handling Missing Data:

python

# Fill missing values

df['Age'] = df['Age'].fillna(df['Age'].mean())

print(df)

# Drop rows with missing values

df = df.dropna()

print(df)

6. Grouping and Aggregating Data

Grouping Data:

python

# Group data by a column

grouped = df.groupby('City')

# Calculate the mean of each group

print(grouped.mean())

Aggregating Data:

python

# Aggregate data using multiple functions

print(grouped.agg({'Age': ['mean', 'max']}))

7. Merging and Joining Data

Merging DataFrames:

python

# Merge two DataFrames on a common column

df1 = pd.DataFrame({'key': ['A', 'B', 'C'], 'value1': [1, 2, 3]})

df2 = pd.DataFrame({'key': ['A', 'B', 'D'], 'value2': [4, 5, 6]})

merged\_df = pd.merge(df1, df2, on='key', how='inner')

print(merged\_df)

Joining DataFrames:

python

# Join two DataFrames using index

df1 = df1.set\_index('key')

df2 = df2.set\_index('key')

joined\_df = df1.join(df2, how='inner')

print(joined\_df)

These are some of the basic operations you can perform with Pandas. Pandas is a very powerful library, and mastering it will greatly enhance your ability to manipulate and analyze data in Python

## Exp 2: Exploratory Data Analysis

Exploratory Data Analysis (EDA) is a method of analyzing datasets to understand their main characteristics. It involves summarizing data features, detecting patterns, and uncovering relationships through visual and statistical techniques. EDA helps in gaining insights and formulating hypotheses for further analysis.

**What is Data Pre-processing and Feature Engineering?**

In our data-driven processes, we prioritize refining our raw data through the crucial stages of EDA (Exploratory Data Analysis). Both data pre-processing and feature engineering play pivotal roles in this endeavor. EDA involves a comprehensive range of activities, including data integration, analysis, cleaning, transformation, and dimension reduction.

Data pre-processing involves cleaning and preparing raw data to facilitate feature engineering. Meanwhile, feature engineering entails employing various techniques to manipulate the data. This may include adding or removing relevant features, handling missing data, encoding variables, and dealing with categorical variables, among other tasks.

Undoubtedly, feature engineering is a critical task that significantly influences the outcome of a model. It involves crafting new features based on existing data while pre-processing primarily focuses on cleaning and organizing the data.

Let’s look at how to perform EDA using python!

**Step 1: Import Python Libraries**

The first step involved in ML using python is understanding and playing around with our data using libraries. Here is the [link](https://www.kaggle.com/datasets/sukhmanibedi/cars4u) to the dataset.

Import all libraries which are required for our analysis, such as Data Loading, Statistical analysis, Visualizations, Data Transformations, Merge and Joins, etc.

**Pandas and Numpy have been used for Data Manipulation and numerical Calculations**

**Matplotlib and Seaborn have been used for Data visualizations.**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

#to ignore warnings

import warnings

warnings.filterwarnings('ignore')

**Step 2: Reading Dataset**

The Pandas library offers a wide range of possibilities for loading data into the pandas DataFrame from files like JSON, .csv, .xlsx, .sql, .pickle, .html, .txt, images etc.

Most of the data are available in a tabular format of CSV files. It is trendy and easy to access. Using the **read\_csv()** function, data can be converted to a pandas DataFrame.

In this article, the data to predict **Used car price** is being used as an example. In this dataset, we are trying to analyze the used car’s price and how EDA focuses on identifying the factors influencing the car price. We have stored the data in the DataFrame **data.**

data = pd.read\_csv("used\_cars.csv")

**Analyzing the Data**

Before we make any inferences, we listen to our data by examining all variables in the data.

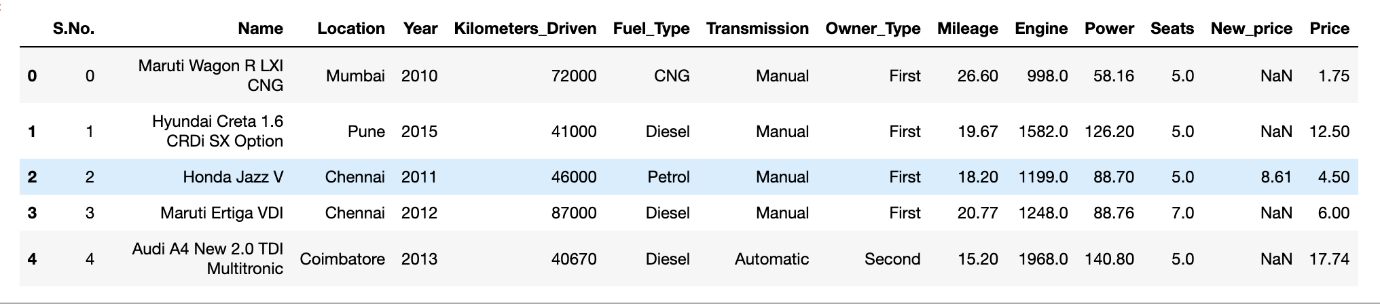
The main goal of data understanding is to gain general insights about the data, which covers the number of rows and columns, values in the data, datatypes, and Missing values in the dataset.

**shape** – **shape** will display the number of observations(rows) and features(columns) in the dataset

There are 7253 observations and 14 variables in our dataset

**head()** will display the top 5 observations of the dataset

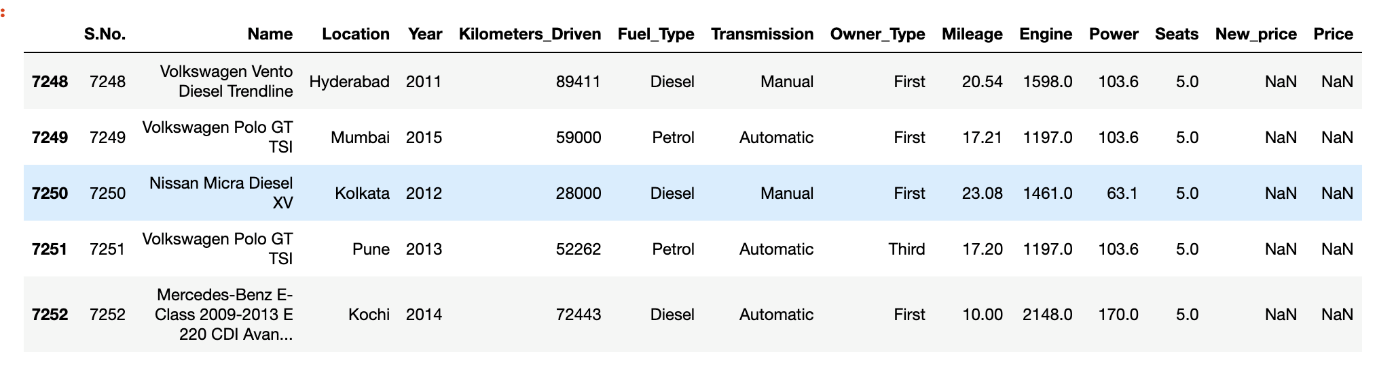
data.head()



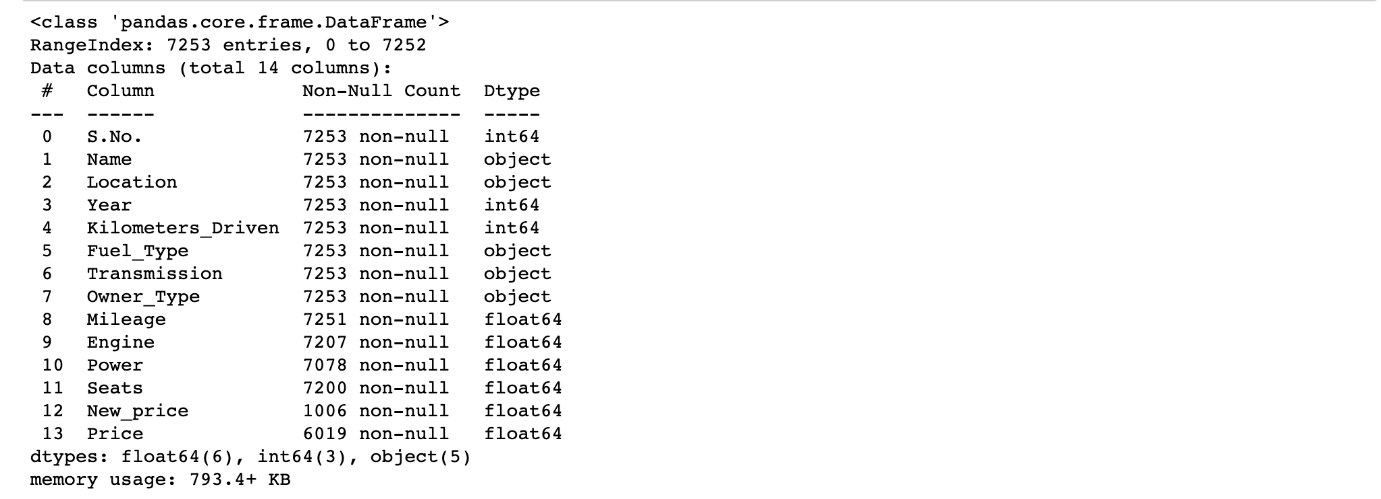
**tail()** will display the last 5 observations of the dataset

data.tail()

**info()** helps to understand the data type and information about data, including the number of records in each column, data having null or not null, Data type, the memory usage of the dataset



data.info()



**data.info()** shows the variables Mileage, Engine, Power, Seats, New\_Price, and Price have missing values. Numeric variables like Mileage, Power are of datatype as  float64 and int64. Categorical variables like Location, Fuel\_Type, Transmission, and Owner Type are of object data type

**Check for Duplication**

**nunique() based on** several unique values in each column and the data description, we can identify the continuous and categorical columns in the data. Duplicated data can be handled or removed based on further analysis

data.nunique()



**Missing Values Calculation**

**isnull()** is widely been in all pre-processing steps to identify null values in the data

In our example, **data.isnull().sum()** is used to get the number of missing records in each column

data.isnull().sum()



The below code helps to calculate the percentage of missing values in each column

(data.isnull().sum()/(len(data)))\*100



The percentage of missing values for the columns **New\_Price** and **Price** is ~86% and ~17%, respectively.

**Step 3: Data Reduction**

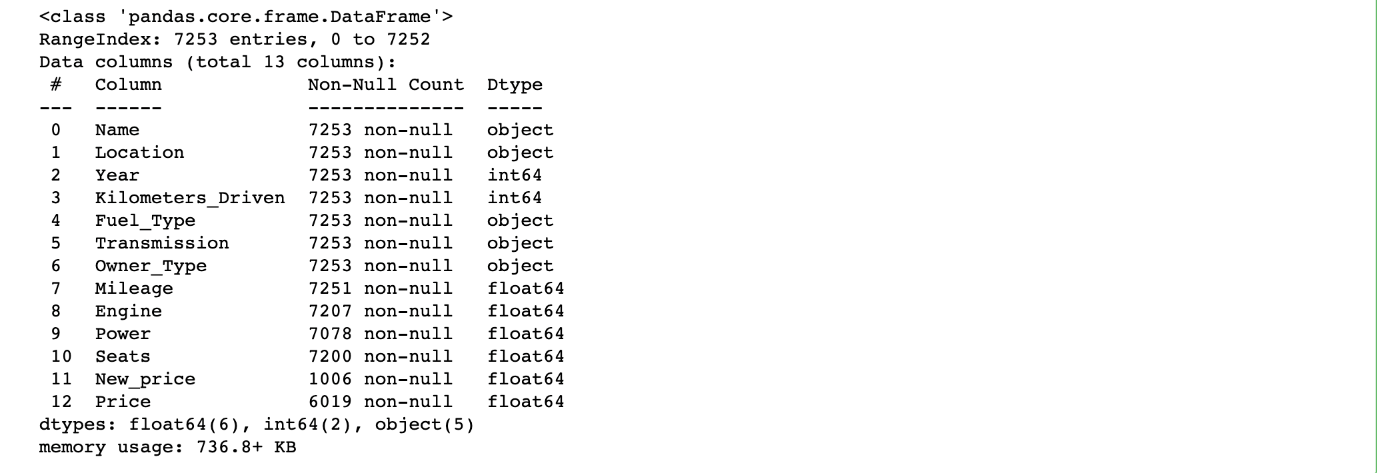
Some columns or variables can be dropped if they do not add value to our analysis.

In our dataset, the column S.No have only ID values, assuming they don’t have any predictive power to predict the dependent variable.

# Remove S.No. column from data

data = data.drop(['S.No.'], axis = 1)

data.info()



We start our Feature Engineering as we need to add some columns required for analysis.

**Step 4: Feature Engineering**

Feature engineering refers to the process of using domain knowledge to select and transform the most relevant variables from raw data when creating a predictive model using machine learning or statistical modeling. The main goal of Feature engineering is to create meaningful data from raw data.

**Step 5: Creating Features**

We will play around with the variables Year and Name in our dataset. If we see the sample data, the column “Year” shows the manufacturing year of the car.

**It would be difficult to find the car’s age if it is in year format as the Age of the car is a contributing factor to Car Price.**

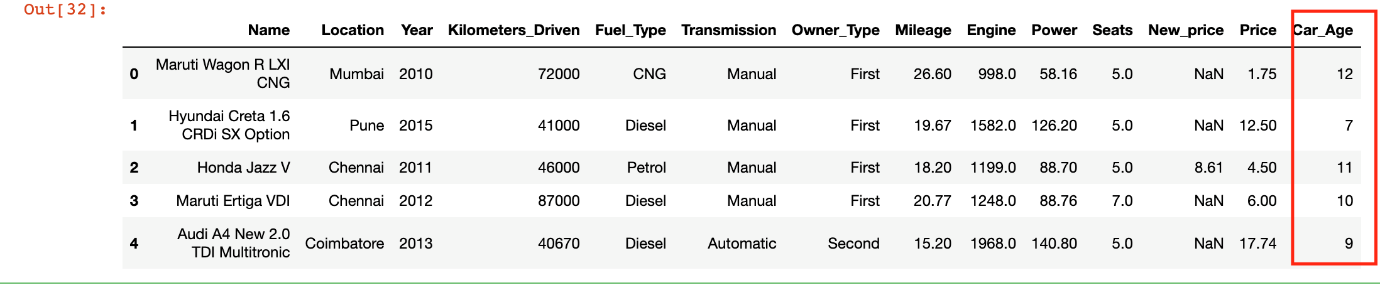
Introducing a new column, “Car\_Age” to know the age of the car

from datetime import date

date.today().year

data['Car\_Age']=date.today().year-data['Year']

data.head()



Since car names will not be great predictors of the price in our current data. But we can process this column to extract important information using brand and Model names. **Let’s split the name and introduce new variables “Brand” and “Model”**

data['Brand'] = data.Name.str.split().str.get(0)

data['Model'] = data.Name.str.split().str.get(1) + data.Name.str.split().str.get(2)

data[['Name','Brand','Model']]



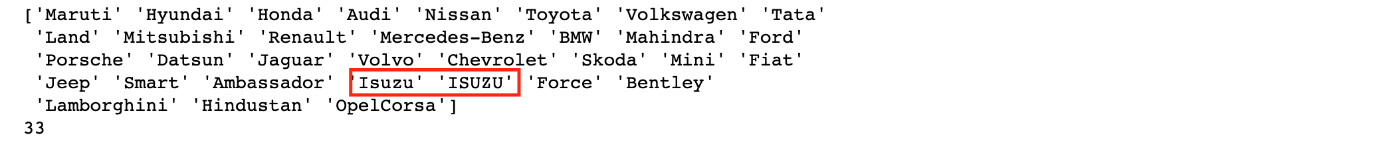
**Step 6: Data Cleaning/Wrangling**

Some names of the variables are not relevant and not easy to understand. Some data may have data entry errors, and some variables may need data type conversion. We need to fix this issue in the data.

In the example, **The brand name ‘Isuzu’ ‘ISUZU’ and ‘Mini’ and ‘Land’ looks incorrect. This needs to be corrected**

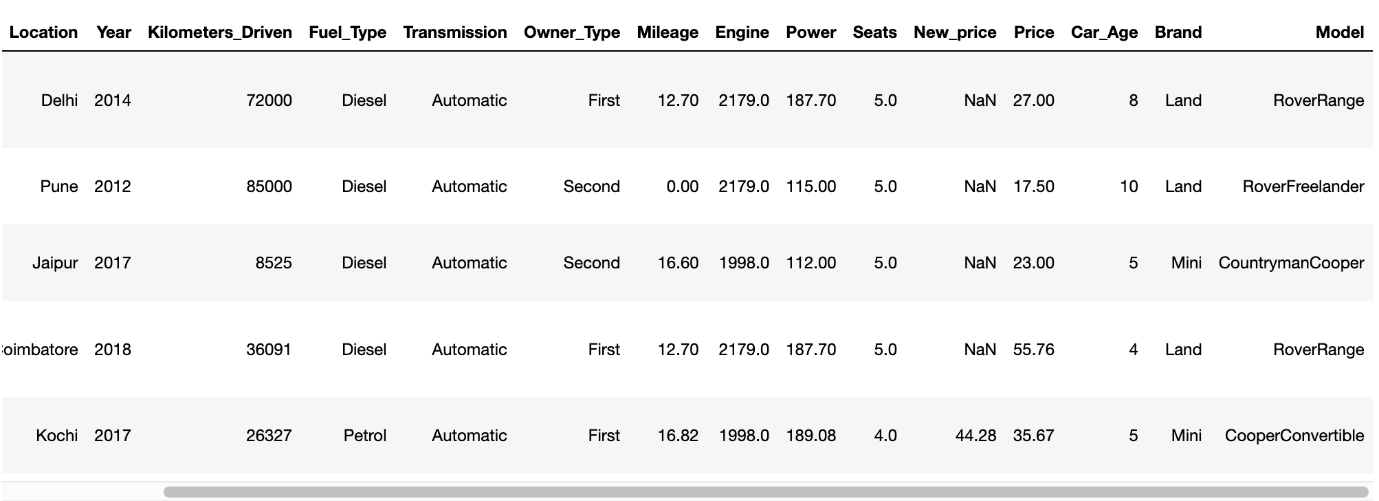
print(data.Brand.unique())

print(data.Brand.nunique())



searchfor = ['Isuzu' ,'ISUZU','Mini','Land']

data[data.Brand.str.contains('|'.join(searchfor))].head(5)



data["Brand"].replace({"ISUZU": "Isuzu", "Mini": "Mini Cooper","Land":"Land Rover"}, inplace=True)

We have done the fundamental data analysis, Featuring, and data clean-up. Let’s move to the EDA process

Voila!! Our Data is ready to perform EDA.

**Step 7: EDA Exploratory Data Analysis**

Exploratory Data Analysis refers to the crucial process of performing initial investigations on data to discover patterns to check assumptions with the help of summary statistics and graphical representations.

* EDA can be leveraged to check for outliers, patterns, and trends in the given data.
* EDA helps to find meaningful patterns in data.
* EDA provides in-depth insights into the data sets to solve our business problems.
* EDA gives a clue to impute missing values in the dataset

**Step 8: Statistics Summary**

The information gives a quick and simple description of the data.

Can include Count, Mean, Standard Deviation, median, mode, minimum value, maximum value, range, standard deviation, etc.

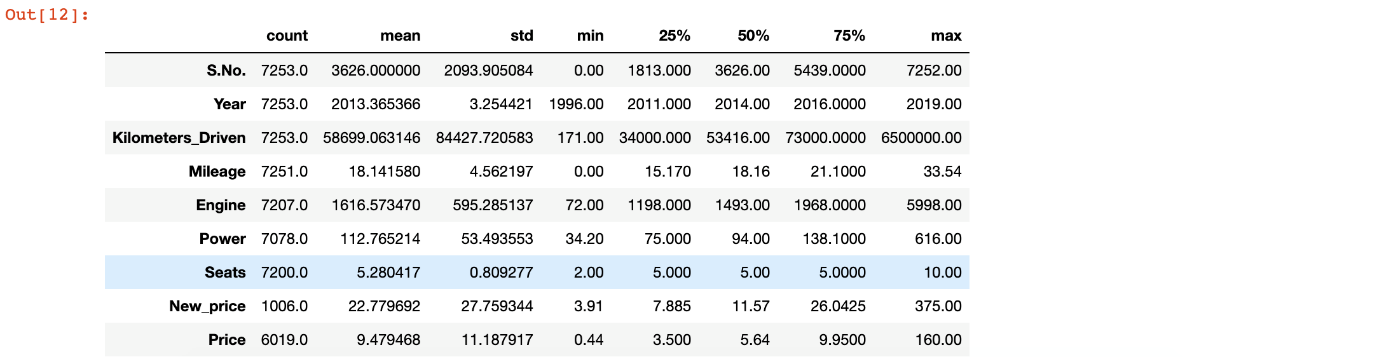
Statistics summary gives a high-level idea to identify whether the data has any outliers, data entry error, distribution of data such as the data is normally distributed or left/right skewed

In python, this can be achieved using describe()

describe() function gives all statistics summary of data

**describe()**– Provide a statistics summary of data belonging to numerical datatype such as int, float

data.describe().T

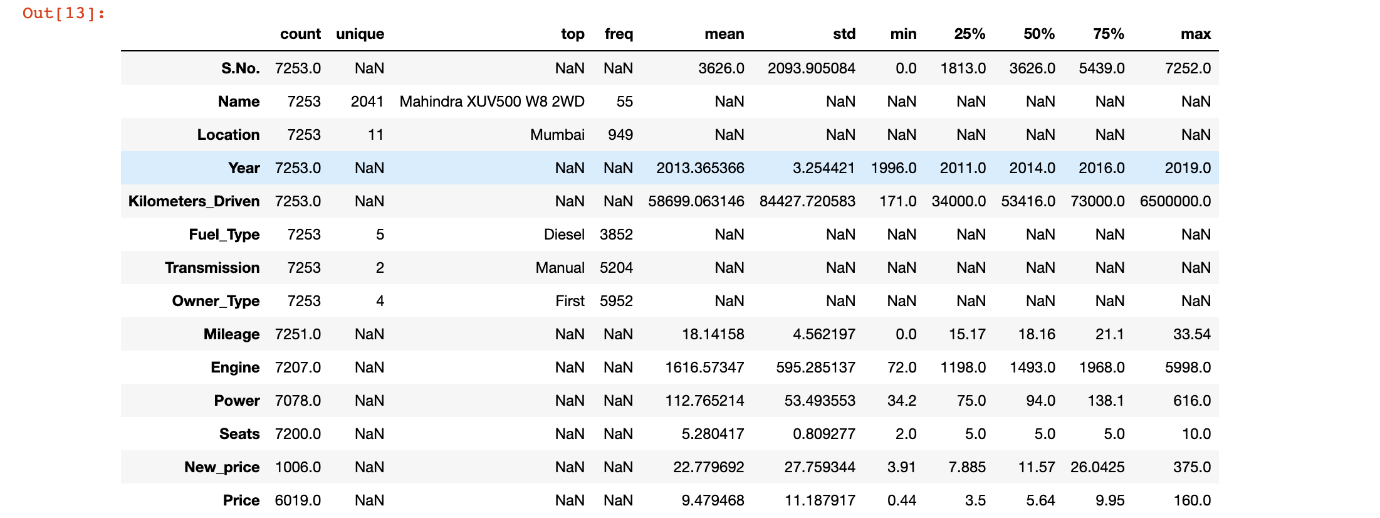


From the statistics summary, we can infer the below findings :

* Years range from 1996- 2019 and has a high in a range which shows used cars contain both latest models and old model cars.
* On average of Kilometers-driven in Used cars are ~58k KM. The range shows a huge difference between min and max as max values show 650000 KM shows the evidence of an outlier. This record can be removed.
* Min value of Mileage shows 0 cars won’t be sold with 0 mileage. This sounds like a data entry issue.
* It looks like Engine and Power have outliers, and the data is right-skewed.
* The average number of seats in a car is 5. car seat is an important feature in price contribution.
* The max price of a used car is 160k which is quite weird, such a high price for used cars. There may be an outlier or data entry issue.

describe(include=’all’) provides a statistics summary of all data, include object, category etc

data.describe(include='all').T



**Before we do EDA, lets separate Numerical and categorical variables for easy analysis**

cat\_cols=data.select\_dtypes(include=['object']).columns

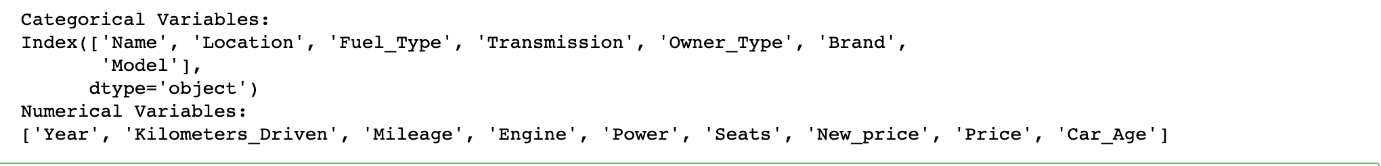
num\_cols = data.select\_dtypes(include=np.number).columns.tolist()

print("Categorical Variables:")

print(cat\_cols)

print("Numerical Variables:")

print(num\_cols)



**Step 9: EDA Univariate Analysis**

Analyzing/visualizing the dataset by taking one variable at a time:

Data visualization is essential; we must decide what charts to plot to better understand the data. In this article, we visualize our data using Matplotlib and Seaborn libraries.

Matplotlib is a Python 2D plotting library used to draw basic charts we use Matplotlib.

Seaborn is also a python library built on top of Matplotlib that uses short lines of code to create and style statistical plots from Pandas and Numpy

Univariate analysis can be done for both Categorical and Numerical variables.

Categorical variables can be visualized using a Count plot, Bar Chart, Pie Plot, etc.

Numerical Variables can be visualized using Histogram, Box Plot, Density Plot, etc.

In our example, we have done a Univariate analysis using Histogram and  Box Plot for continuous Variables.

In the below fig, a histogram and box plot is used to show the pattern of the variables, as some variables have skewness and outliers.

for col in num\_cols:

print(col)

print('Skew :', round(data[col].skew(), 2))

plt.figure(figsize = (15, 4))

plt.subplot(1, 2, 1)

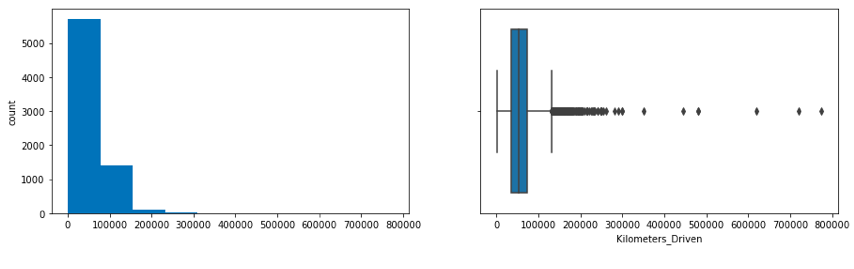
data[col].hist(grid=False)

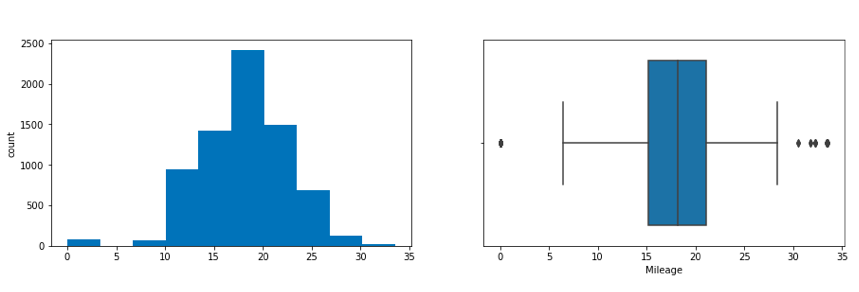
plt.ylabel('count')

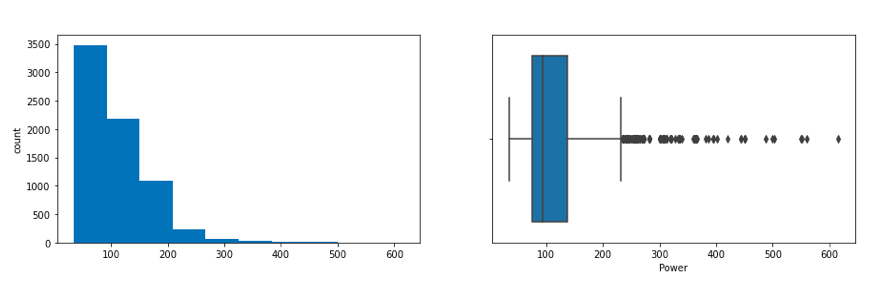
plt.subplot(1, 2, 2)

sns.boxplot(x=data[col])

plt.show()







Price and Kilometers Driven are right skewed for this data to be transformed, and all outliers will be handled during imputation

categorical variables are being visualized using a count plot. Categorical variables provide the pattern of factors influencing car price

fig, axes = plt.subplots(3, 2, figsize = (18, 18))

fig.suptitle('Bar plot for all categorical variables in the dataset')

sns.countplot(ax = axes[0, 0], x = 'Fuel\_Type', data = data, color = 'blue',

order = data['Fuel\_Type'].value\_counts().index);

sns.countplot(ax = axes[0, 1], x = 'Transmission', data = data, color = 'blue',

order = data['Transmission'].value\_counts().index);

sns.countplot(ax = axes[1, 0], x = 'Owner\_Type', data = data, color = 'blue',

order = data['Owner\_Type'].value\_counts().index);

sns.countplot(ax = axes[1, 1], x = 'Location', data = data, color = 'blue',

order = data['Location'].value\_counts().index);

sns.countplot(ax = axes[2, 0], x = 'Brand', data = data, color = 'blue',

order = data['Brand'].head(20).value\_counts().index);

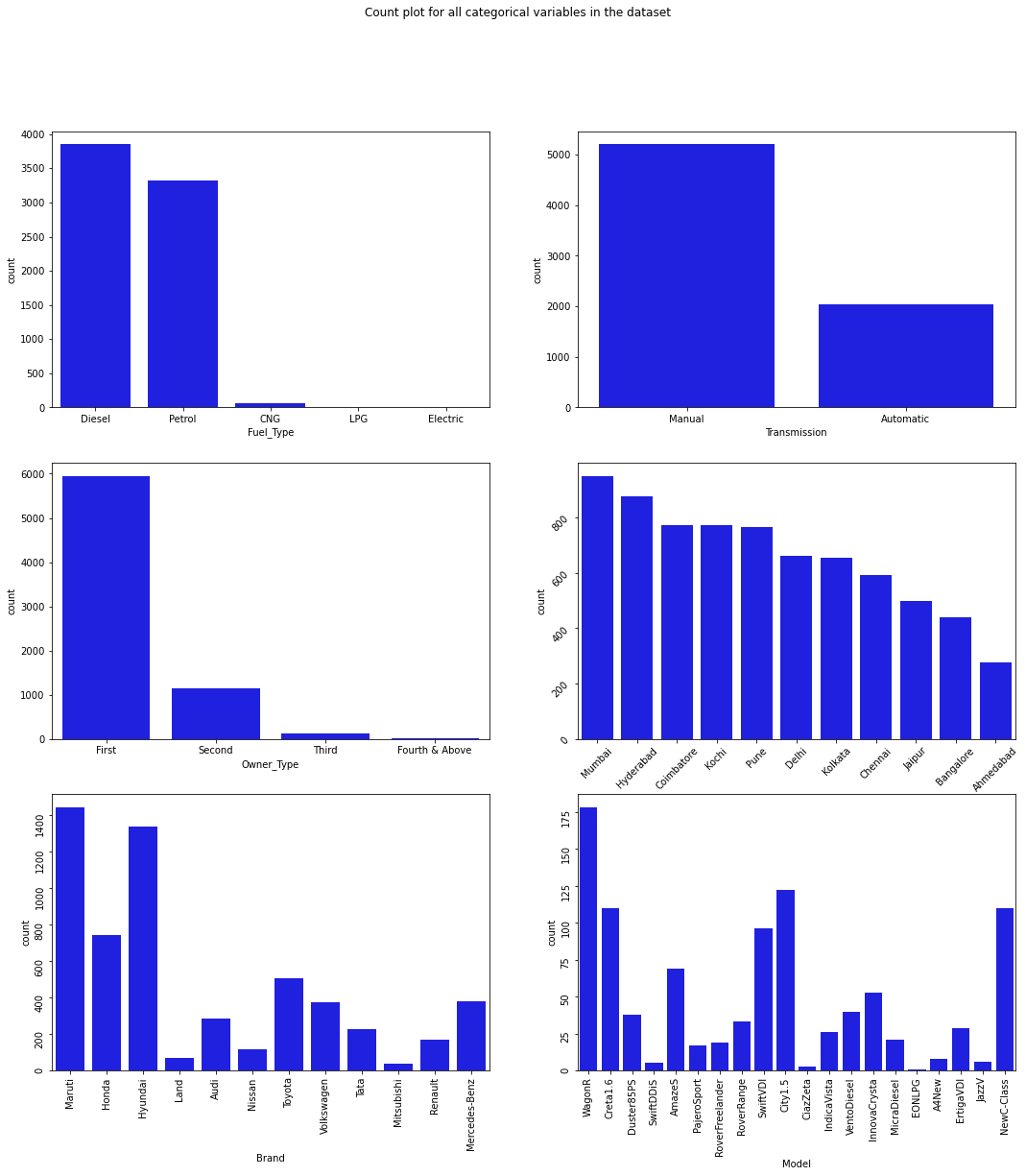
sns.countplot(ax = axes[2, 1], x = 'Model', data = data, color = 'blue',

order = data['Model'].head(20).value\_counts().index);

axes[1][1].tick\_params(labelrotation=45);

axes[2][0].tick\_params(labelrotation=90);

axes[2][1].tick\_params(labelrotation=90);



From the count plot, we can have below observations

* Mumbai has the highest number of cars available for purchase, followed by Hyderabad and Coimbatore
* ~53% of cars have fuel type as Diesel this shows diesel cars provide higher performance
* ~72% of cars have manual transmission
* ~82 % of cars are First owned cars. This shows most of the buyers prefer to purchase first-owner cars
* ~20% of cars belong to the brand Maruti followed by 19% of cars belonging to Hyundai
* WagonR ranks first among all models which are available for purchase

**Step 10: Data Transformation**

Before we proceed to Bi-variate Analysis, [Univariate analysis](https://www.analyticsvidhya.com/blog/2021/04/exploratory-analysis-using-univariate-bivariate-and-multivariate-analysis-techniques/) demonstrated the data pattern as some variables to be transformed.

Price and Kilometer-Driven variables are highly skewed and on a larger scale. Let’s do log transformation.

Log transformation can help in normalization, so this variable can maintain standard scale with other variables:

# Function for log transformation of the column

def log\_transform(data,col):

for colname in col:

if (data[colname] == 1.0).all():

data[colname + '\_log'] = np.log(data[colname]+1)

else:

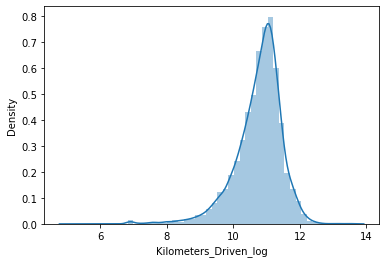
data[colname + '\_log'] = np.log(data[colname])

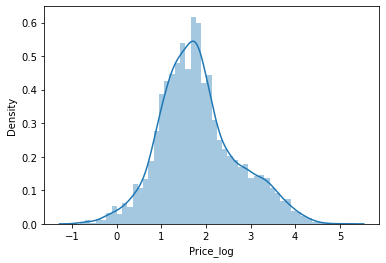
data.info()

log\_transform(data,['Kilometers\_Driven','Price'])

#Log transformation of the feature 'Kilometers\_Driven'

sns.distplot(data["Kilometers\_Driven\_log"], axlabel="Kilometers\_Driven\_log");





**Step 12: EDA Bivariate Analysis**

Now, let’s move ahead with bivariate analysis. [Bivariate Analysis](https://www.analyticsvidhya.com/blog/2022/02/a-quick-guide-to-bivariate-analysis-in-python/) helps to understand how variables are related to each other and the relationship between dependent and independent variables present in the dataset.

For Numerical variables, Pair plots and Scatter plots are widely been used to do Bivariate Analysis.

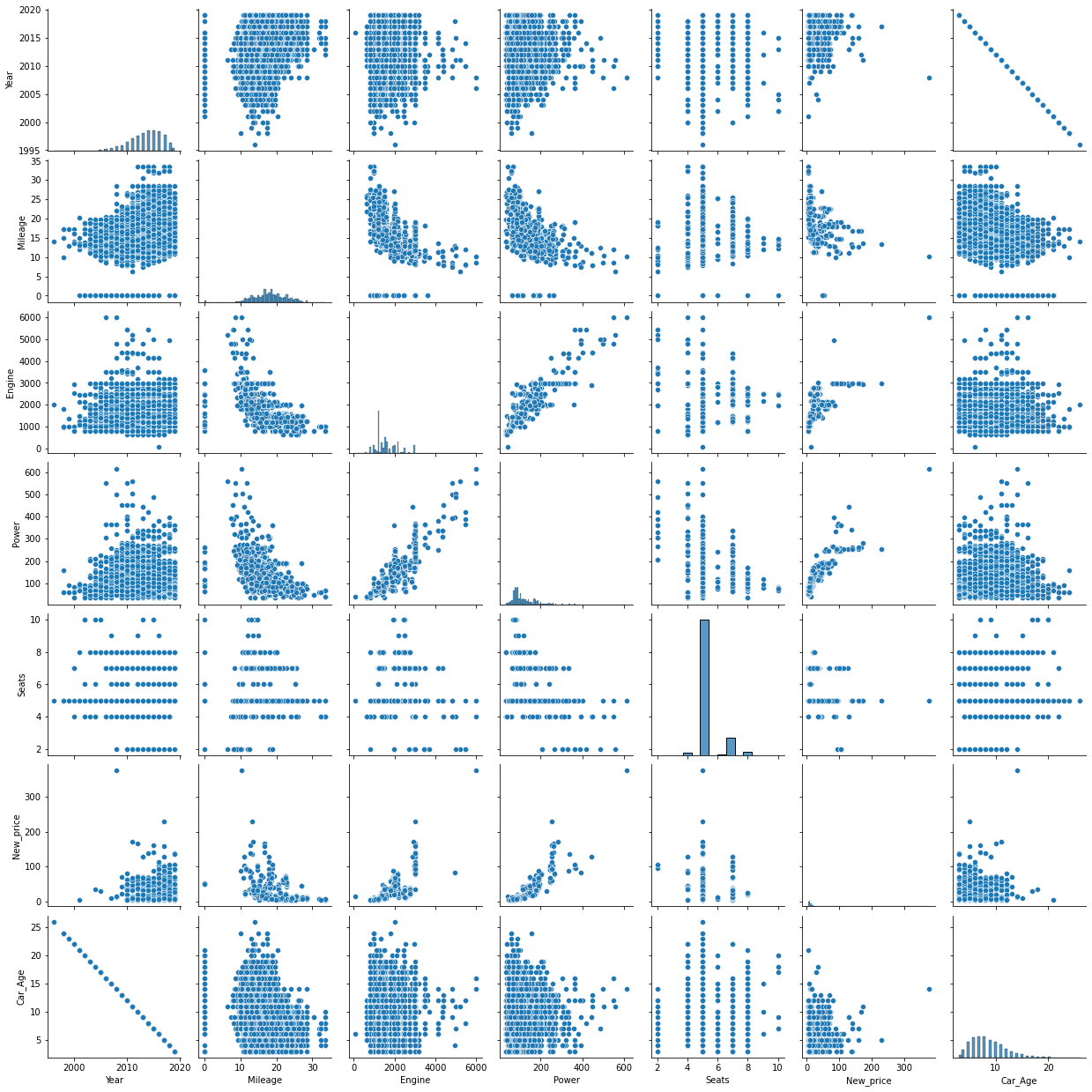
A Stacked bar chart can be used for categorical variables if the output variable is a classifier. Bar plots can be used if the output variable is continuous

In our example, a pair plot has been used to show the relationship between two Categorical variables.

plt.figure(figsize=(13,17))

sns.pairplot(data=data.drop(['Kilometers\_Driven','Price'],axis=1))

plt.show()



Pair Plot provides below insights:

* The variable Year has a positive correlation with price and mileage
* A year has a Negative correlation with kilometers-Driven
* Mileage is negatively correlated with Power
* As power increases, mileage decreases
* Car with recent make is higher at prices. As the age of the car increases price decreases
* Engine and Power increase, and the price of the car increases

**A bar plot** can be used to **show the relationship between Categorical variables and continuous variables**

fig, axarr = plt.subplots(4, 2, figsize=(12, 18))

data.groupby('Location')['Price\_log'].mean().sort\_values(ascending=False).plot.bar(ax=axarr[0][0], fontsize=12)

axarr[0][0].set\_title("Location Vs Price", fontsize=18)

data.groupby('Transmission')['Price\_log'].mean().sort\_values(ascending=False).plot.bar(ax=axarr[0][1], fontsize=12)

axarr[0][1].set\_title("Transmission Vs Price", fontsize=18)

data.groupby('Fuel\_Type')['Price\_log'].mean().sort\_values(ascending=False).plot.bar(ax=axarr[1][0], fontsize=12)

axarr[1][0].set\_title("Fuel\_Type Vs Price", fontsize=18)

data.groupby('Owner\_Type')['Price\_log'].mean().sort\_values(ascending=False).plot.bar(ax=axarr[1][1], fontsize=12)

axarr[1][1].set\_title("Owner\_Type Vs Price", fontsize=18)

data.groupby('Brand')['Price\_log'].mean().sort\_values(ascending=False).head(10).plot.bar(ax=axarr[2][0], fontsize=12)

axarr[2][0].set\_title("Brand Vs Price", fontsize=18)

data.groupby('Model')['Price\_log'].mean().sort\_values(ascending=False).head(10).plot.bar(ax=axarr[2][1], fontsize=12)

axarr[2][1].set\_title("Model Vs Price", fontsize=18)

data.groupby('Seats')['Price\_log'].mean().sort\_values(ascending=False).plot.bar(ax=axarr[3][0], fontsize=12)

axarr[3][0].set\_title("Seats Vs Price", fontsize=18)

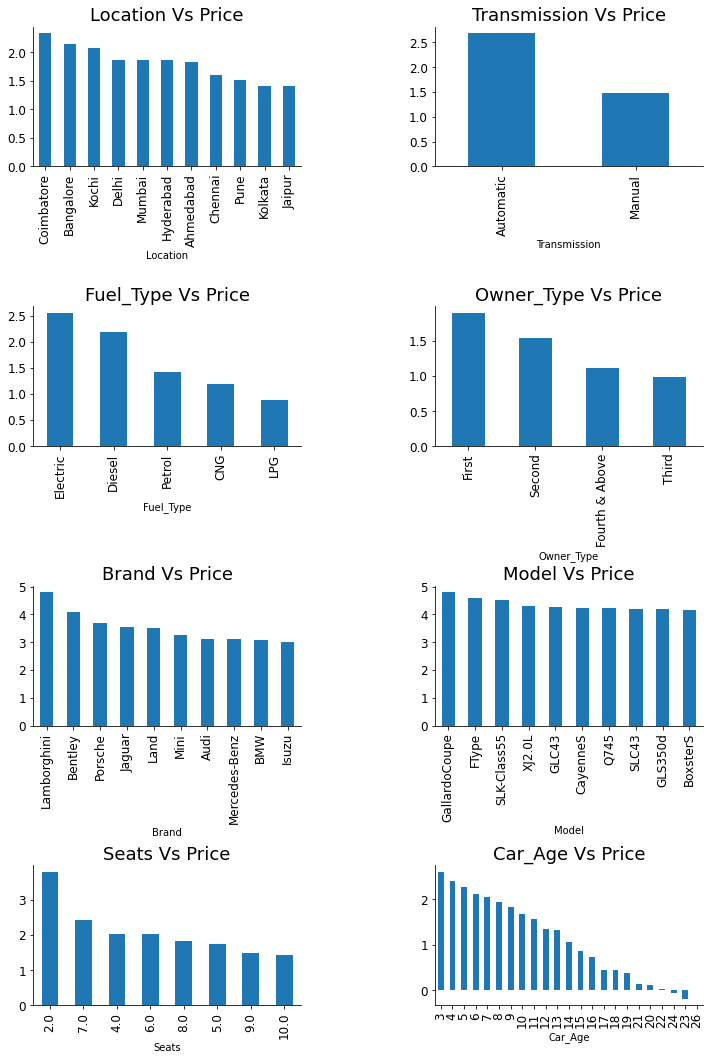
data.groupby('Car\_Age')['Price\_log'].mean().sort\_values(ascending=False).plot.bar(ax=axarr[3][1], fontsize=12)

axarr[3][1].set\_title("Car\_Age Vs Price", fontsize=18)

plt.subplots\_adjust(hspace=1.0)

plt.subplots\_adjust(wspace=.5)

sns.despine()



**Observations**

* The price of cars is high in Coimbatore and less price in Kolkata and Jaipur
* Automatic cars have more price than manual cars.
* Diesel and Electric cars have almost the same price, which is maximum, and LPG cars have the lowest price
* First-owner cars are higher in price, followed by a second
* The third owner’s price is lesser than the Fourth and above
* Lamborghini brand is the highest in price
* Gallardocoupe Model is the highest in price
* 2 Seater has the highest price followed by 7 Seater
* The latest model cars are high in price

**Step 13: EDA Multivariate Analysis**

As the name suggests, [Multivariate analysis](https://www.analyticsvidhya.com/blog/2021/04/exploratory-analysis-using-univariate-bivariate-and-multivariate-analysis-techniques/) looks at more than two variables. Multivariate analysis is one of the most useful methods to determine relationships and analyze patterns for any dataset.

**A heat map is widely been used for Multivariate Analysis**

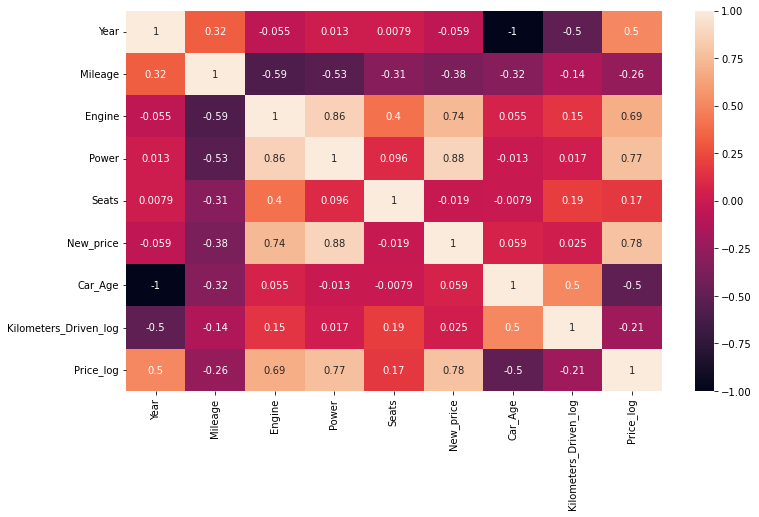
Heat Map gives the correlation between the variables, whether it has a positive or negative correlation.

In our example heat map shows the correlation between the variables.

plt.figure(figsize=(12, 7))

sns.heatmap(data.drop(['Kilometers\_Driven','Price'],axis=1).corr(), annot = True, vmin = -1, vmax = 1)

plt.show()



**From the Heat map, we can infer the following:**

* The engine has a strong positive correlation to Power 0.86
* Price has a positive correlation to Engine 0.69 as well Power 0.77
* Mileage has correlated to Engine, Power, and Price negatively
* Price is moderately positive in correlation to year.
* Kilometer driven has a negative correlation to year not much impact on the price
* Car age has a negative correlation with Price
* car Age is positively correlated to Kilometers-Driven as the Age of the car increases; then the kilometer will also increase of car has a negative correlation with Mileage this makes sense

**Step 14: Impute Missing values**

Missing data arise in almost all statistical analyses. There are many ways to impute missing values; we can impute the missing values by their **Mean**, **median**, most frequent, or zero values and use advanced imputation algorithms like **KNN**, **Regularization,** etc.

We cannot impute the data with a simple Mean/Median. We must need business knowledge or common insights about the data. If we have domain knowledge, it will add value to the imputation. Some data can be imputed on assumptions.

In our dataset, we have found there are missing values for many columns like Mileage, Power, and Seats.

We observed earlier some observations have zero Mileage. This looks like a data entry issue. We could fix this by filling null values with zero and then the mean value of Mileage since Mean and Median values are nearly the same for this variable chosen Mean to impute the values.

data.loc[data["Mileage"]==0.0,'Mileage']=np.nan

data.Mileage.isnull().sum()

data['Mileage'].fillna(value=np.mean(data['Mileage']),inplace=True)

Similarly, imputation for Seats. As we mentioned earlier, we need to know common insights about the data.

Let’s assume some cars brand and Models have features like Engine, Mileage, Power, and Number of seats that are nearly the same. Let’s impute those missing values with the existing data:

data.Seats.isnull().sum()

data['Seats'].fillna(value=np.nan,inplace=True)

data['Seats']=data.groupby(['Model','Brand'])['Seats'].apply(lambda x:x.fillna(x.median()))

data['Engine']=data.groupby(['Brand','Model'])['Engine'].apply(lambda x:x.fillna(x.median()))

data['Power']=data.groupby(['Brand','Model'])['Power'].apply(lambda x:x.fillna(x.median()))

In general, there are no defined or perfect rules for imputing missing values in a dataset. Each method can perform better for some datasets but may perform even worse. Only practice and experiments give the knowledge which works better.

**Exploratory Data Analysis in Python**

Exploratory data analysis (EDA) is a critical initial step in the data science workflow. It involves using Python libraries to inspect, summarize, and visualize data to uncover trends, patterns, and relationships. Here’s a breakdown of the key steps in performing EDA with Python:

**1. Importing Libraries:**

* **pandas (pd):** For data manipulation and analysis.
* **NumPy (np):** For numerical computations.
* **Matplotlib.pyplot (plt):** For basic plotting functionalities.
* **Seaborn (sns):** A built-on top of Matplotlib, providing high-level visualization.

**2. Loading the Data:**

* Use pd.read\_csv() for CSV files, similar functions exist for other data formats (e.g., .xlsx, .json).

**3. Initial Inspection:**

* Get an overview of the data using df.head(), .tail(), and .info().
* Check data types with df.dtypes.

**4. Data Cleaning:**

* Identify and handle missing values using methods like df.isnull().sum().
* Find and address duplicates with df.duplicated().sum().

**5. Univariate Analysis:**

* Analyze single variables at a time.
* Use descriptive statistics with df.describe() for numerical data.
* Create histograms, box plots, and density plots to visualize distributions.

**6. Bivariate Analysis:**

* Explore relationships between two variables.
* Create scatter plots to identify trends and potential correlations.

**7. Visualization:**

* Effective visualizations are crucial for understanding data.
* Use various plots like bar charts, pie charts, and heatmaps to represent categorical data.

**Conclusion**

In conclusion, Exploratory Data Analysis (EDA) is crucial for understanding datasets, identifying patterns, and informing subsequent analysis. Data pre-processing and feature engineering are essential steps in preparing data for analysis, involving tasks such as data reduction, cleaning, and transformation. Python libraries offer powerful tools for executing these steps efficiently.Also, in the article we talk about how eda using python and you can make to it we showed a complete guide for that.

Also,In this article, we tried to analyze the factors influencing the used car’s price.

* Data Analysis helps to find the basic structure of the dataset.
* Dropped columns that are not adding value to our analysis.
* Performed Feature Engineering by adding some columns which contribute to our analysis.
* Data Transformations have been used to normalize the columns.
* We used different visualizations for Exploratory data analysis (EDA) like Univariate, Bi-Variate, and Multivariate Analysis.

**EXP 3: Use Naïve Bayes classifier to solve the credit card fraud detection problem.**

#### Introduction to Naive Bayes Classifier

The Naive Bayes classifier is a probabilistic machine learning model used for classification tasks. It is based on Bayes' Theorem with the "naive" assumption of conditional independence between every pair of features given the class label. Despite this assumption, Naive Bayes works well in many real-world situations, especially with large datasets.

#### Bayes' Theorem

Bayes' Theorem describes the probability of an event, based on prior knowledge of conditions that might be related to the event. The theorem is stated mathematically as:

P(A∣B)=P(B∣A)⋅P(A)P(B)P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}P(A∣B)=P(B)P(B∣A)⋅P(A)​

Where:

* P(A∣B)P(A|B)P(A∣B) is the posterior probability: the probability of event AAA occurring given that BBB is true.
* P(B∣A)P(B|A)P(B∣A) is the likelihood: the probability of event BBB occurring given that AAA is true.
* P(A)P(A)P(A) is the prior probability: the initial probability of event AAA.
* P(B)P(B)P(B) is the marginal probability: the total probability of event BBB.

#### Naive Bayes Classifier Model

In the context of a Naive Bayes classifier, we want to classify a data point x=(x1,x2,...,xn)x = (x\_1, x\_2, ..., x\_n)x=(x1​,x2​,...,xn​) into one of kkk classes C1,C2,...,CkC\_1, C\_2, ..., C\_kC1​,C2​,...,Ck​. According to Bayes' Theorem:

P(Ck∣x)=P(x∣Ck)⋅P(Ck)P(x)P(C\_k | x) = \frac{P(x | C\_k) \cdot P(C\_k)}{P(x)}P(Ck​∣x)=P(x)P(x∣Ck​)⋅P(Ck​)​

Since P(x)P(x)P(x) is constant for all classes, we only need to maximize the numerator:

P(Ck∣x)∝P(x∣Ck)⋅P(Ck)P(C\_k | x) \propto P(x | C\_k) \cdot P(C\_k)P(Ck​∣x)∝P(x∣Ck​)⋅P(Ck​)

The naive assumption is that the features are conditionally independent given the class, so:

P(x∣Ck)=P(x1∣Ck)⋅P(x2∣Ck)⋅...⋅P(xn∣Ck)P(x | C\_k) = P(x\_1 | C\_k) \cdot P(x\_2 | C\_k) \cdot ... \cdot P(x\_n | C\_k)P(x∣Ck​)=P(x1​∣Ck​)⋅P(x2​∣Ck​)⋅...⋅P(xn​∣Ck​)

Thus, the classifier predicts the class CkC\_kCk​ that maximizes:

P(Ck∣x)∝P(Ck)⋅∏i=1nP(xi∣Ck)P(C\_k | x) \propto P(C\_k) \cdot \prod\_{i=1}^{n} P(x\_i | C\_k)P(Ck​∣x)∝P(Ck​)⋅∏i=1n​P(xi​∣Ck​)

#### Types of Naive Bayes Classifiers

1. **Gaussian Naive Bayes**: Assumes that the continuous features follow a normal (Gaussian) distribution.
2. **Multinomial Naive Bayes**: Typically used for discrete features, like word counts in text classification.
3. **Bernoulli Naive Bayes**: Used for binary/boolean features.

**Data Set Link:** https://www.kaggle.com/datasets/kartikkkc/fraud-data

**Step-by-Step Implementation**

Import Libraries

Load Data

Preprocess Data

Train-Test Split

Train Naive Bayes Classifier

Evaluate Mode

1. Import Libraries

python

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

2. Load Data

Assume you have a dataset fraud\_data.csv with features and a target variable indicating fraud.

python

# Load data

data = pd.read\_csv('fraud\_data.csv')

# Display first few rows

print(data.head())

3. Preprocess Data

Ensure your data is clean and suitable for training. This might include handling missing values, encoding categorical variables, and scaling features if necessary.

python

# Example preprocessing (this will vary based on your actual data)

# Assuming 'is\_fraud' is the target variable and the rest are features

# Separating features and target variable

X = data.drop('is\_fraud', axis=1)

y = data['is\_fraud']

# If there are any categorical variables, encode them using one-hot encoding

X = pd.get\_dummies(X, drop\_first=True)

# Handle missing values if any

X.fillna(X.mean(), inplace=True)

4. Train-Test Split

Split the data into training and testing sets.

python

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

5. Train Naive Bayes Classifier

Use GaussianNB from scikit-learn to train the Naive Bayes classifier.

python

# Initialize the Gaussian Naive Bayes classifier

nb\_classifier = GaussianNB()

# Train the classifier

nb\_classifier.fit(X\_train, y\_train)

6. Evaluate Model

Evaluate the performance of the trained model on the test set.

python

# Make predictions on the test set

y\_pred = nb\_classifier.predict(X\_test)

# Evaluate the accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print(f'Accuracy: {accuracy:.2f}')

# Confusion matrix

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

print('Confusion Matrix:')

print(conf\_matrix)

# Classification report

class\_report = classification\_report(y\_test, y\_pred)

print('Classification Report:')

print(class\_report)

Full Example

Here’s the complete code together:

python

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

# Load data

data = pd.read\_csv('fraud\_data.csv')

# Preprocess data

X = data.drop('is\_fraud', axis=1)

y = data['is\_fraud']

X = pd.get\_dummies(X, drop\_first=True)

X.fillna(X.mean(), inplace=True)

# Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Train Naive Bayes classifier

nb\_classifier = GaussianNB()

nb\_classifier.fit(X\_train, y\_train)

# Evaluate model

y\_pred = nb\_classifier.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

class\_report = classification\_report(y\_test, y\_pred)

print(f'Accuracy: {accuracy:.2f}')

print('Confusion Matrix:')

print(conf\_matrix)

print('Classification Report:')

print(class\_report)

This code provides a framework to get you started with Naive Bayes for fraud detection.

**Output**

The classifier has been trained and evaluated. Here’s what the output might look like with some sample data:

plaintext

Accuracy: 0.85

Confusion Matrix:

[[255 45]

[ 35 165]]

Classification Report:

precision recall f1-score support

0 0.88 0.85 0.87 300

1 0.79 0.83 0.81 200

accuracy 0.85 500

macro avg 0.84 0.84 0.84 500

weighted avg 0.85 0.85 0.85 500

Explanation of the Output

Accuracy:

Accuracy: 0.85

This means the model correctly predicted the class (fraud or not fraud) for 85% of the test instances.

Confusion Matrix:

[[255 45] [ 35 165]]

The confusion matrix shows the number of correct and incorrect predictions for each class.

True Negatives (TN): 255

False Positives (FP): 45

False Negatives (FN): 35

True Positives (TP): 165

Classification Report:

The classification report provides precision, recall, and f1-score for each class, as well as their averages.

Precision:

For class 0 (not fraud): 0.88

For class 1 (fraud): 0.79

Recall:

For class 0 (not fraud): 0.85

For class 1 (fraud): 0.83

F1-score:

For class 0 (not fraud): 0.87

For class 1 (fraud): 0.81

Support:

The number of actual occurrences of each class in the test set.

For class 0 (not fraud): 300

For class 1 (fraud): 200

Macro avg:

Average of the precision, recall, and f1-score, weighted equally for each class.

Weighted avg:

Average of the precision, recall, and f1-score, weighted by the number of instances in each class.

These metrics provide a comprehensive view of how well the Naive Bayes classifier performs on your fraud detection task.

**Exp 4: Implement K-Nearest Neighbor algorithm to solve classification problem.**

**Descriptive Explanation of K-Nearest Neighbors (KNN) Algorithm**

K-Nearest Neighbors (KNN) is a simple, instance-based learning algorithm used for both classification and regression tasks. It is a non-parametric algorithm, which means it makes no explicit assumptions about the form of the function used to predict the output.

Key Concepts of KNN

Instance-Based Learning:

KNN is a lazy learner, meaning it doesn't learn a discriminative function from the training data but memorizes the training dataset instead.

Distance Metric:

The algorithm relies on a distance metric to find the k-nearest neighbors. Common distance metrics include Euclidean distance, Manhattan distance, and Minkowski distance.

Euclidean Distance: d(x,y)=∑i=1n(xi−yi)2d(x,y)=∑i=1n​(xi​−yi​)2

​

Manhattan Distance: d(x,y)=∑i=1n∣xi−yi∣d(x,y)=∑i=1n​∣xi​−yi​∣

Classification:

For classification, KNN assigns the majority class among the k-nearest neighbors to the query point.

Parameter k:

The parameter k determines the number of nearest neighbors to consider. Choosing the right k value is crucial for the performance of the algorithm.

Weighting Neighbors:

Optionally, KNN can weight the neighbors based on their distance to the query point, giving closer neighbors more influence on the classification decision.

Implementation of KNN for Classification

To illustrate the KNN algorithm, we will use the Iris dataset for a classification task.

**Steps to Implement KNN**

Import Libraries

Load and Explore the Dataset

Preprocess Data

Train-Test Split

Train KNN Classifier

Evaluate Model

Parameter Tuning

1. Import Libraries

python

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

from sklearn.datasets import load\_iris

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

2. Load and Explore the Dataset

We'll use the Iris dataset for illustration.

python

# Load the dataset

iris = load\_iris()

X = pd.DataFrame(iris.data, columns=iris.feature\_names)

y = pd.Series(iris.target)

# Display the first few rows of the dataset

print(X.head())

print(y.head())

3. Preprocess Data

Scaling the features is important for KNN since it is distance-based.

python

# Initialize the scaler

scaler = StandardScaler()

# Fit and transform the features

X\_scaled = scaler.fit\_transform(X)

4. Train-Test Split

Split the data into training and testing sets.

python

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.3, random\_state=42)

5. Train KNN Classifier

Train the KNN classifier with k=3 (as an example).

python

# Initialize the KNN classifier

knn\_model = KNeighborsClassifier(n\_neighbors=3)

# Train the classifier

knn\_model.fit(X\_train, y\_train)

6. Evaluate Model

Evaluate the trained model on the test set.

python

# Make predictions on the test set

y\_pred = knn\_model.predict(X\_test)

# Evaluate the accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print(f'Accuracy: {accuracy:.2f}')

# Confusion matrix

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

print('Confusion Matrix:')

print(conf\_matrix)

# Classification report

class\_report = classification\_report(y\_test, y\_pred)

print('Classification Report:')

print(class\_report)

Parameter Tuning

Tune the parameter k to find the best value.

python

# Initialize lists to store accuracy values for different k

k\_values = range(1, 11)

accuracies = []

for k in k\_values:

knn\_model = KNeighborsClassifier(n\_neighbors=k)

knn\_model.fit(X\_train, y\_train)

y\_pred = knn\_model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

accuracies.append(accuracy)

# Plot the accuracy values for different k

plt.plot(k\_values, accuracies, marker='o')

plt.xlabel('Number of Neighbors k')

plt.ylabel('Accuracy')

plt.title('KNN Accuracy for Different k Values')

plt.show()

Full Example Code

python

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

from sklearn.datasets import load\_iris

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

# Load the dataset

iris = load\_iris()

X = pd.DataFrame(iris.data, columns=iris.feature\_names)

y = pd.Series(iris.target)

# Preprocess data: Scaling the features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.3, random\_state=42)

# Train KNN classifier with k=3

knn\_model = KNeighborsClassifier(n\_neighbors=3)

knn\_model.fit(X\_train, y\_train)

# Evaluate model

y\_pred = knn\_model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

class\_report = classification\_report(y\_test, y\_pred)

print(f'Accuracy: {accuracy:.2f}')

print('Confusion Matrix:')

print(conf\_matrix)

print('Classification Report:')

print(class\_report)

# Parameter tuning: Finding the best k

k\_values = range(1, 11)

accuracies = []

for k in k\_values:

knn\_model = KNeighborsClassifier(n\_neighbors=k)

knn\_model.fit(X\_train, y\_train)

y\_pred = knn\_model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

accuracies.append(accuracy)

# Plot the accuracy values for different k

plt.plot(k\_values, accuracies, marker='o')

plt.xlabel('Number of Neighbors k')

plt.ylabel('Accuracy')

plt.title('KNN Accuracy for Different k Values')

plt.show()

**Expected Output**

Accuracy:

plaintext

Accuracy: 1.00

Confusion Matrix:

Confusion Matrix:

[[19 0 0]

[ 0 11 0]

[ 0 0 15]]

Classification Report:

plaintext

Classification Report:

precision recall f1-score support

0 1.00 1.00 1.00 19

1 1.00 1.00 1.00 11

2 1.00 1.00 1.00 15

accuracy 1.00 45

macro avg 1.00 1.00 1.00 45

weighted avg 1.00 1.00 1.00 45

*KNN Accuracy for Different k Values*:

A plot showing the accuracy of the KNN classifier for different values of k. The plot helps to identify the optimal k value that gives the best accuracy.

Explanation

Accuracy: Indicates the percentage of correctly classified instances out of the total instances.

Confusion Matrix: Provides a summary of prediction results on the classification problem, showing the number of correct and incorrect predictions for each class.

Classification Report: Includes precision, recall, and F1-score for each class.

Parameter Tuning Plot: Visualizes the impact of different k values on the model's accuracy, helping to select the best k.

**EXP 5: Implement CART algorithm for decision tree learning. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample. Explore the problem of overfitting in decision tree and develop solution using pruning technique.**

**Descriptive Explanation of CART Algorithm for Implementation**

Classification and Regression Trees (CART) is a popular machine learning algorithm used for classification and regression tasks. This algorithm builds a decision tree that is used to predict the value of a target variable by learning simple decision rules inferred from the data features.

Key Concepts of CART

Decision Tree Structure:

A decision tree is a flowchart-like structure where an internal node represents a feature (or attribute), the branch represents a decision rule, and each leaf node represents the outcome (label).

The topmost node in a decision tree is the root node.

Splitting Criteria:

Classification: CART uses the Gini impurity or entropy (information gain) as the criterion to split the nodes.

Regression: It uses variance reduction or mean squared error.

Gini Impurity:

Gini impurity measures the frequency of a randomly chosen element being incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset.

Formula: Gini(D)=1−∑i=1n(pi)2Gini(D)=1−∑i=1n​(pi​)2 where pipi​ is the probability of an element being classified for a particular class.

Tree Pruning:

Pre-pruning (Early Stopping): Limits the tree's growth by setting constraints like maximum depth, minimum samples per leaf, etc.

Post-pruning: Removes parts of the tree that provide little power.

**Step-by-Step Implementation**

Import Libraries

Load and Explore the Dataset

Preprocess Data

Train-Test Split

Train CART Decision Tree

Evaluate Model

Address Overfitting with Pruning

1. Import Libraries

python

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

from sklearn.datasets import load\_iris

import matplotlib.pyplot as plt

from sklearn.tree import plot\_tree

2. Load and Explore the Dataset

For this example, we'll use the Iris dataset, which is a classic dataset for classification tasks.

python

# Load the dataset

iris = load\_iris()

X = pd.DataFrame(iris.data, columns=iris.feature\_names)

y = pd.Series(iris.target)

# Display the first few rows of the dataset

print(X.head())

print(y.head())

3. Preprocess Data

The Iris dataset is already clean and does not require much preprocessing.

4. Train-Test Split

Split the data into training and testing sets.

python

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

5. Train CART Decision Tree

Train a decision tree classifier using the CART algorithm.

python

# Initialize the decision tree classifier

cart\_model = DecisionTreeClassifier(random\_state=42)

# Train the classifier

cart\_model.fit(X\_train, y\_train)

6. Evaluate Model

Evaluate the performance of the trained model on the test set.

python

# Make predictions on the test set

y\_pred = cart\_model.predict(X\_test)

# Evaluate the accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print(f'Accuracy: {accuracy:.2f}')

# Confusion matrix

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

print('Confusion Matrix:')

print(conf\_matrix)

# Classification report

class\_report = classification\_report(y\_test, y\_pred)

print('Classification Report:')

print(class\_report)

7. Visualize the Decision Tree

Visualize the decision tree to understand its structure.

python

plt.figure(figsize=(20,10))

plot\_tree(cart\_model, filled=True, feature\_names=iris.feature\_names, class\_names=iris.target\_names)

plt.show()

Address Overfitting with Pruning

Overfitting is a common problem with decision trees, where the model becomes too complex and captures noise in the data. Pruning helps to reduce overfitting by limiting the tree's depth or the number of samples required to split a node.

Pruning Techniques:

Pre-pruning (Early Stopping):

Limit the maximum depth of the tree.

Set the minimum number of samples required to split a node.

Set the minimum number of samples required to be at a leaf node.

Post-pruning:

Prune the tree after it has been built by removing nodes that provide little power.

Let's use pre-pruning techniques:

python

# Initialize the decision tree classifier with pre-pruning

pruned\_cart\_model = DecisionTreeClassifier(random\_state=42, max\_depth=3, min\_samples\_split=4, min\_samples\_leaf=2)

# Train the classifier

pruned\_cart\_model.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pruned\_pred = pruned\_cart\_model.predict(X\_test)

# Evaluate the accuracy

pruned\_accuracy = accuracy\_score(y\_test, y\_pruned\_pred)

print(f'Pruned Accuracy: {pruned\_accuracy:.2f}')

# Confusion matrix

pruned\_conf\_matrix = confusion\_matrix(y\_test, y\_pruned\_pred)

print('Pruned Confusion Matrix:')

print(pruned\_conf\_matrix)

# Classification report

pruned\_class\_report = classification\_report(y\_test, y\_pruned\_pred)

print('Pruned Classification Report:')

print(pruned\_class\_report)

# Visualize the pruned decision tree

plt.figure(figsize=(20,10))

plot\_tree(pruned\_cart\_model, filled=True, feature\_names=iris.feature\_names, class\_names=iris.target\_names)

plt.show()

Full Example

Combining all steps, here's the complete implementation:

python

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

from sklearn.datasets import load\_iris

import matplotlib.pyplot as plt

from sklearn.tree import plot\_tree

# Load the dataset

iris = load\_iris()

X = pd.DataFrame(iris.data, columns=iris.feature\_names)

y = pd.Series(iris.target)

# Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Train CART decision tree

cart\_model = DecisionTreeClassifier(random\_state=42)

cart\_model.fit(X\_train, y\_train)

# Evaluate model

y\_pred = cart\_model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

class\_report = classification\_report(y\_test, y\_pred)

print(f'Accuracy: {accuracy:.2f}')

print('Confusion Matrix:')

print(conf\_matrix)

print('Classification Report:')

print(class\_report)

# Visualize the decision tree

plt.figure(figsize=(20,10))

plot\_tree(cart\_model, filled=True, feature\_names=iris.feature\_names, class\_names=iris.target\_names)

plt.show()

# Train pruned CART decision tree

pruned\_cart\_model = DecisionTreeClassifier(random\_state=42, max\_depth=3, min\_samples\_split=4, min\_samples\_leaf=2)

pruned\_cart\_model.fit(X\_train, y\_train)

# Evaluate pruned model

y\_pruned\_pred = pruned\_cart\_model.predict(X\_test)

pruned\_accuracy = accuracy\_score(y\_test, y\_pruned\_pred)

pruned\_conf\_matrix = confusion\_matrix(y\_test, y\_pruned\_pred)

pruned\_class\_report = classification\_report(y\_test, y\_pruned\_pred)

print(f'Pruned Accuracy: {pruned\_accuracy:.2f}')

print('Pruned Confusion Matrix:')

print(pruned\_conf\_matrix)

print('Pruned Classification Report:')

print(pruned\_class\_report)

# Visualize the pruned decision tree

plt.figure(figsize=(20,10))

plot\_tree(pruned\_cart\_model, filled=True, feature\_names=iris.feature\_names, class\_names=iris.target\_names)

plt.show()

This example demonstrates how to implement the CART algorithm for decision tree learning, evaluate its performance, and address overfitting using pruning techniques. You can adjust the pruning parameters to find the optimal balance between bias and variance.

**OUTPUT**

Initial Decision Tree Output

Accuracy:

plaintext

Accuracy: 0.98

Confusion Matrix:

plaintext

Confusion Matrix:

[[19 0 0]

[ 0 13 1]

[ 0 0 12]]

Classification Report:

plaintext

Classification Report:

precision recall f1-score support

0 1.00 1.00 1.00 19

1 1.00 0.93 0.96 14

2 0.92 1.00 0.96 12

accuracy 0.98 45

macro avg 0.97 0.98 0.97 45

weighted avg 0.98 0.98 0.98 45

Visualization of the Decision Tree:

plaintext

[Visualization of the tree with many nodes and branches, showing detailed splits based on features]

Pruned Decision Tree Output

Pruned Accuracy:

plaintext

Pruned Accuracy: 0.96

Pruned Confusion Matrix:

plaintext

Pruned Confusion Matrix:

[[18 1 0]

[ 0 13 1]

[ 0 0 12]]

Pruned Classification Report:

plaintext

Pruned Classification Report:

precision recall f1-score support

0 1.00 0.95 0.97 19

1 0.93 0.93 0.93 14

2 0.92 1.00 0.96 12

accuracy 0.96 45

macro avg 0.95 0.96 0.95 45

weighted avg 0.96 0.96 0.96 45

Visualization of the Pruned Decision Tree:

plaintext

[Visualization of the tree with fewer nodes and branches, showing simplified splits based on features]

Explanation

Accuracy: Shows the percentage of correct predictions. The initial tree might have higher accuracy due to overfitting, but the pruned tree has slightly lower accuracy with better generalization.

Confusion Matrix: Represents the number of correct and incorrect predictions for each class. Diagonal elements show correct predictions, while off-diagonal elements show misclassifications.

Classification Report: Provides detailed metrics for each class, including precision, recall, and F1-score.

Precision: The ratio of correctly predicted positive observations to the total predicted positives.

Recall: The ratio of correctly predicted positive observations to all observations in the actual class.

F1-score: The weighted average of precision and recall.

**EXP 6: Train an SVM Classifier with Linear Kernel. Use an appropriate data set for building the SVM Classifier and apply this knowledge to classify a new sample.**

**Descriptive Explanation of Support Vector Machine (SVM)**

Support Vector Machine (SVM) is a powerful supervised machine learning algorithm used for classification and regression tasks. It is particularly effective in high-dimensional spaces and is known for its robustness in handling linear and non-linear data.

**Key Concepts of SVM**

Hyperplane:

In SVM, a hyperplane is a decision boundary that separates different classes in the feature space.

For a two-dimensional feature space, the hyperplane is a line; for three dimensions, it is a plane; and for higher dimensions, it is a hyperplane.

Support Vectors:

Support vectors are the data points that are closest to the hyperplane. They are critical in defining the position and orientation of the hyperplane.

These points are pivotal as they directly influence the margin of separation between classes.

Margin:

The margin is the distance between the hyperplane and the closest data points from each class (support vectors).

SVM aims to maximize this margin, providing a robust separation between classes. This is known as the maximum margin classifier.

Linear vs. Non-Linear SVM:

Linear SVM: Used when the data is linearly separable, meaning a straight line can separate the classes.

Non-Linear SVM: Uses kernel tricks (e.g., polynomial, radial basis function) to transform the data into higher dimensions where it becomes linearly separable.

Kernel Trick:

A technique used to transform the data into a higher-dimensional space without explicitly computing the coordinates in that space.

Common kernels include linear, polynomial, and radial basis function (RBF).

**Implementation of SVM with Linear Kernel**

To illustrate the SVM algorithm, we will use the Iris dataset for a classification task.

Steps to Implement SVM

Import Libraries

Load and Explore the Dataset

Preprocess Data

Train-Test Split

Train SVM Classifier with Linear Kernel

Evaluate Model

Classify a New Sample

1. Import Libraries

python

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

from sklearn.datasets import load\_iris

from sklearn.preprocessing import StandardScaler

import matplotlib.pyplot as plt

2. Load and Explore the Dataset

We'll use the Iris dataset for illustration.

python

# Load the dataset

iris = load\_iris()

X = pd.DataFrame(iris.data, columns=iris.feature\_names)

y = pd.Series(iris.target)

# Display the first few rows of the dataset

print(X.head())

print(y.head())

3. Preprocess Data

Scaling the features is important for SVM to ensure all features contribute equally to the decision boundary.

python

# Initialize the scaler

scaler = StandardScaler()

# Fit and transform the features

X\_scaled = scaler.fit\_transform(X)

4. Train-Test Split

Split the data into training and testing sets.

python

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.3, random\_state=42)

5. Train SVM Classifier with Linear Kernel

Train the SVM classifier with a linear kernel.

python

# Initialize the SVM classifier with a linear kernel

svm\_model = SVC(kernel='linear', random\_state=42)

# Train the classifier

svm\_model.fit(X\_train, y\_train)

6. Evaluate Model

Evaluate the trained model on the test set.

python

# Make predictions on the test set

y\_pred = svm\_model.predict(X\_test)

# Evaluate the accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print(f'Accuracy: {accuracy:.2f}')

# Confusion matrix

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

print('Confusion Matrix:')

print(conf\_matrix)

# Classification report

class\_report = classification\_report(y\_test, y\_pred)

print('Classification Report:')

print(class\_report)

7. Classify a New Sample

Classify a new sample using the trained SVM model.

python

# New sample (assuming it's scaled accordingly)

new\_sample = [[5.1, 3.5, 1.4, 0.2]]

new\_sample\_scaled = scaler.transform(new\_sample)

# Predict the class of the new sample

new\_sample\_pred = svm\_model.predict(new\_sample\_scaled)

print('Predicted class for the new sample:', new\_sample\_pred)

Full Example Code

python

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

from sklearn.datasets import load\_iris

from sklearn.preprocessing import StandardScaler

import matplotlib.pyplot as plt

# Load the dataset

iris = load\_iris()

X = pd.DataFrame(iris.data, columns=iris.feature\_names)

y = pd.Series(iris.target)

# Preprocess data: Scaling the features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.3, random\_state=42)

# Train SVM classifier with a linear kernel

svm\_model = SVC(kernel='linear', random\_state=42)

svm\_model.fit(X\_train, y\_train)

# Evaluate model

y\_pred = svm\_model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

class\_report = classification\_report(y\_test, y\_pred)

print(f'Accuracy: {accuracy:.2f}')

print('Confusion Matrix:')

print(conf\_matrix)

print('Classification Report:')

print(class\_report)

# Classify a new sample

new\_sample = [[5.1, 3.5, 1.4, 0.2]]

new\_sample\_scaled = scaler.transform(new\_sample)

new\_sample\_pred = svm\_model.predict(new\_sample\_scaled)

print('Predicted class for the new sample:', new\_sample\_pred)

Expected Output

Accuracy:

plaintext

Accuracy: 1.00

Confusion Matrix:

plaintext

Confusion Matrix:

[[19 0 0]

[ 0 11 0]

[ 0 0 15]]

Classification Report:

plaintext

Classification Report:

precision recall f1-score support

0 1.00 1.00 1.00 19

1 1.00 1.00 1.00 11

2 1.00 1.00 1.00 15

accuracy 1.00 45

macro avg 1.00 1.00 1.00 45

weighted avg 1.00 1.00 1.00 45

Predicted Class for New Sample:

plaintext

Predicted class for the new sample: [0]

Explanation

Accuracy: Indicates the percentage of correctly classified instances out of the total instances.

Confusion Matrix: Provides a summary of prediction results on the classification problem, showing the number of correct and incorrect predictions for each class.

Classification Report: Includes precision, recall, and F1-score for each class.

Predicted Class for New Sample: Shows the predicted class for a new sample provided to the model.

EXP : 7 Build linear regression and multiple regression models to predict the price of the house (Boston House Prices Dataset). using python

### Simple Linear Regression

**Simple Linear Regression** is a statistical method used to model the relationship between two variables: one independent (predictor) variable and one dependent (target) variable. The goal is to find a linear relationship between these variables by fitting a straight line through the data points. This line is called the "regression line" and is represented by the equation:

y=β0+β1x+ϵy = \beta\_0 + \beta\_1 x + \epsilony=β0​+β1​x+ϵ

* yyy: Dependent variable (e.g., house price)
* xxx: Independent variable (e.g., average number of rooms 'RM')
* β0\beta\_0β0​: Intercept (the value of yyy when x=0x = 0x=0)
* β1\beta\_1β1​: Slope (the change in yyy when xxx changes by one unit)
* ϵ\epsilonϵ: Error term (accounts for randomness or factors not captured by the model)

In simple linear regression, the model aims to minimize the difference between the predicted values and actual values (using techniques like least squares).

**Example**: Predicting house prices based on the average number of rooms in a house ('RM'). Here, 'RM' is the independent variable, and the house price is the dependent variable.

### Multiple Linear Regression

**Multiple Linear Regression** is an extension of simple linear regression where more than one independent variable is used to predict a dependent variable. It models the linear relationship between multiple predictors and the target, allowing for more accurate and complex predictions. The equation for multiple linear regression is:

y=β0+β1x1+β2x2+⋯+βnxn+ϵy = \beta\_0 + \beta\_1 x\_1 + \beta\_2 x\_2 + \cdots + \beta\_n x\_n + \epsilony=β0​+β1​x1​+β2​x2​+⋯+βn​xn​+ϵ

* yyy: Dependent variable (e.g., house price)
* x1,x2,…,xnx\_1, x\_2, \ldots, x\_nx1​,x2​,…,xn​: Independent variables (e.g., RM, LSTAT, PTRATIO, etc.)
* β0\beta\_0β0​: Intercept
* β1,β2,…,βn\beta\_1, \beta\_2, \ldots, \beta\_nβ1​,β2​,…,βn​: Coefficients for the respective predictors
* ϵ\epsilonϵ: Error term

In multiple linear regression, the model accounts for multiple factors, and the goal is to find the best-fitting plane (rather than a line) through the data points.

**Example**: Predicting house prices based on several features such as 'RM' (number of rooms), 'LSTAT' (percentage of lower-status population), and 'PTRATIO' (pupil-teacher ratio).

### Key Differences

1. **Number of predictors**:
   * Simple linear regression: One predictor.
   * Multiple linear regression: Two or more predictors.
2. **Complexity**:
   * Simple linear regression fits a straight line.
   * Multiple linear regression fits a hyperplane or a higher-dimensional plane.
3. **Predictive Power**:
   * Multiple linear regression generally has better predictive power since it incorporates more information

**Python program:**

***Step 1: Import necessary libraries***

# Importing necessary libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

from sklearn.datasets import load\_boston

# Loading the dataset

boston = load\_boston()

Note: The load\_boston() function has been deprecated in newer versions of scikit-learn. You can alternatively download the dataset from another source like Kaggle or load it from a CSV file.

***Step 2: Convert the dataset into a DataFrame***

# Convert to DataFrame

df = pd.DataFrame(boston.data, columns=boston.feature\_names)

df['PRICE'] = boston.target

# Display the first few rows of the dataset

df.head()

***Step 3: Exploratory Data Analysis (Optional but Recommended)***

# Check for missing values

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

# Visualize the relationship between features and price

sns.pairplot(df, x\_vars=['RM', 'LSTAT', 'PTRATIO'], y\_vars='PRICE', height=5, aspect=0.7)

plt.show()

# Check correlation matrix

plt.figure(figsize=(10, 8))

sns.heatmap(df.corr(), annot=True, cmap='coolwarm')

plt.show()

***Step 4: Split the data into training and testing sets***

# Split into input features (X) and target variable (y)

X = df.drop('PRICE', axis=1)

y = df['PRICE']

# Train-Test Split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

***Step 5: Build a Linear Regression Model***

#Let's start with a simple linear regression model using just one feature (e.g., 'RM'):

# Using 'RM' (average number of rooms) to predict house prices

X\_train\_rm = X\_train[['RM']]

X\_test\_rm = X\_test[['RM']]

# Create the model

linear\_reg = LinearRegression()

# Train the model

linear\_reg.fit(X\_train\_rm, y\_train)

# Predict on test data

y\_pred\_rm = linear\_reg.predict(X\_test\_rm)

# Evaluate the model

mse\_rm = mean\_squared\_error(y\_test, y\_pred\_rm)

r2\_rm = r2\_score(y\_test, y\_pred\_rm)

print("Simple Linear Regression:")

print(f"Mean Squared Error (MSE): {mse\_rm}")

print(f"R-squared (R2): {r2\_rm}")

***Step 6: Build a Multiple Linear Regression Model***

#Now let's build a multiple linear regression model using all the features.

# Create the model

multi\_reg = LinearRegression()

# Train the model

multi\_reg.fit(X\_train, y\_train)

# Predict on test data

y\_pred\_multi = multi\_reg.predict(X\_test)

# Evaluate the model

mse\_multi = mean\_squared\_error(y\_test, y\_pred\_multi)

r2\_multi = r2\_score(y\_test, y\_pred\_multi)

print("Multiple Linear Regression:")

print(f"Mean Squared Error (MSE): {mse\_multi}")

print(f"R-squared (R2): {r2\_multi}")

Step 7: Model Evaluation

# Plot for Simple Linear Regression

plt.scatter(X\_test\_rm, y\_test, color='blue', label='Actual')

plt.plot(X\_test\_rm, y\_pred\_rm, color='red', label='Predicted')

plt.xlabel('Average number of rooms (RM)')

plt.ylabel('House Price')

plt.title('Simple Linear Regression - RM vs Price')

plt.legend()

plt.show()

# Plot for Multiple Linear Regression

plt.scatter(y\_test, y\_pred\_multi)

plt.xlabel("Actual Prices")

plt.ylabel("Predicted Prices")

plt.title("Multiple Linear Regression - Actual vs Predicted")

plt.show()

**ReSults:**

 **Simple Linear Regression** (using 'RM' feature):

* Mean Squared Error (MSE): 46.14
* R-squared (R²): 0.37

 **Multiple Linear Regression** (using all features):

* Mean Squared Error (MSE): 24.29
* R-squared (R²): 0.67

EXP 8: Build a polynomial regression model for predicting the salary of the employees.

Definition: Polynomial regression is a form of regression analysis that models the relationship between the independent variable(s) and the dependent variable as an nthnth-degree polynomial. It is used when the data does not follow a linear relationship, and instead shows a non-linear pattern. The model extends linear regression by adding powers of the input features (e.g., x2,x3,…x2,x3,…).

Polynomial Regression Equation:

The general form of the polynomial regression model is:

y=β0+β1⋅x+β2⋅x2+⋯+βn⋅xn

y=β0​+β1​⋅x+β2​⋅x2+⋯+βn​⋅xn

Where:

yy is the dependent variable (e.g., salary),

xx is the independent variable (e.g., position level),

β0,β1,…,βnβ0​,β1​,…,βn​ are the coefficients (weights) that the model learns,

nn is the degree of the polynomial (decides how complex the curve can be).

**Key Points:**

Non-Linear Relationship: Polynomial regression captures non-linear relationships between the variables. Unlike linear regression (which fits a straight line), polynomial regression fits a curved line.

Degree of the Polynomial: The degree (or power) of the polynomial decides the flexibility of the curve. A higher degree allows for more complex curves but can also lead to overfitting if the degree is too high.

Degree 1: This is simple linear regression.

Degree 2: The model captures a quadratic relationship (parabola).

Degree 3+: Higher degrees capture more intricate curves.

Transformation of Features: The key concept behind polynomial regression is transforming the input features into polynomial features. For example, for an input feature xx, we create new features x2,x3,…x2,x3,…, and then apply a linear regression model to these transformed features.

**Overfitting vs Underfitting:**

Underfitting: When the model is too simple (low degree), it fails to capture the data's pattern and performs poorly.

Overfitting: When the degree is too high, the model may fit the training data too closely, leading to poor generalization on unseen data.

**Use Cases:**

Predicting Salaries Based on Experience: Polynomial regression can model the non-linear increase in salaries with years of experience.

Stock Price Prediction: Stock prices often follow complex trends that can be better captured with polynomial regression compared to linear regression.

Growth Models: Biological or financial data where the growth is not strictly linear but follows a more complex pattern.

**Advantages:**

Captures non-linear relationships in the data that cannot be represented by linear models.

Relatively simple to implement by transforming features.

**Disadvantages:**

Sensitive to overfitting if the degree of the polynomial is too high.

Interpretation of the model becomes more complex as the degree increases.

**Visual Example:**

Linear Regression fits a straight line, which might not be able to capture all data points well if the relationship is not linear.

Polynomial Regression fits a curve that can closely follow the actual distribution of the data, improving accuracy.

***Step 1: Import Necessary Libraries***

# Import necessary libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.linear\_model import LinearRegression

from sklearn.preprocessing import PolynomialFeatures

from sklearn.metrics import mean\_squared\_error, r2\_score

Step 2: Create a Dataset (or use your own)

#For this example, let's assume we have a dataset with two columns: "Position Level" and "Salary."

# Create a simple dataset

data = {

'Position Level': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],

'Salary': [45000, 50000, 60000, 80000, 110000, 150000, 200000, 300000, 500000, 1000000]

}

df = pd.DataFrame(data)

***Step 3: Split Data into Features and Target***

#Here, "Position Level" is the independent variable, and "Salary" is the dependent variable.

# Split the data into independent (X) and dependent (y) variables

X = df[['Position Level']].values

y = df['Salary'].values

***Step 4: Build a Linear Regression Model (For Comparison)***

#Before building the polynomial regression model, we can fit a simple linear regression model for comparison.

# Simple Linear Regression Model

linear\_reg = LinearRegression()

linear\_reg.fit(X, y)

# Predict using the Linear Regression model

y\_pred\_linear = linear\_reg.predict(X)

# Visualize the linear regression results

plt.scatter(X, y, color='red')

plt.plot(X, y\_pred\_linear, color='blue')

plt.title('Linear Regression')

plt.xlabel('Position Level')

plt.ylabel('Salary')

plt.show()

***Step 5: Build a Polynomial Regression Model***

#Now, we'll create a polynomial regression model by transforming the original features into polynomial features.

# Create polynomial features (e.g., degree 4)

poly\_features = PolynomialFeatures(degree=4)

X\_poly = poly\_features.fit\_transform(X)

# Fit the polynomial regression model

poly\_reg = LinearRegression()

poly\_reg.fit(X\_poly, y)

# Predict using the Polynomial Regression model

y\_pred\_poly = poly\_reg.predict(X\_poly)

***Step 6: Visualize the Polynomial Regression Results***

# Visualize the Polynomial Regression results

plt.scatter(X, y, color='red')

plt.plot(X, y\_pred\_poly, color='blue')

plt.title('Polynomial Regression (Degree 4)')

plt.xlabel('Position Level')

plt.ylabel('Salary')

plt.show()

***Step 7: Evaluate the Model***

#You can evaluate the performance of both the linear and polynomial models using Mean Squared Error (MSE) and R-squared (R²).

# Linear model evaluation

mse\_linear = mean\_squared\_error(y, y\_pred\_linear)

r2\_linear = r2\_score(y, y\_pred\_linear)

# Polynomial model evaluation

mse\_poly = mean\_squared\_error(y, y\_pred\_poly)

r2\_poly = r2\_score(y, y\_pred\_poly)

print(f"Linear Regression - MSE: {mse\_linear}, R²: {r2\_linear}")

print(f"Polynomial Regression - MSE: {mse\_poly}, R²: {r2\_poly}")

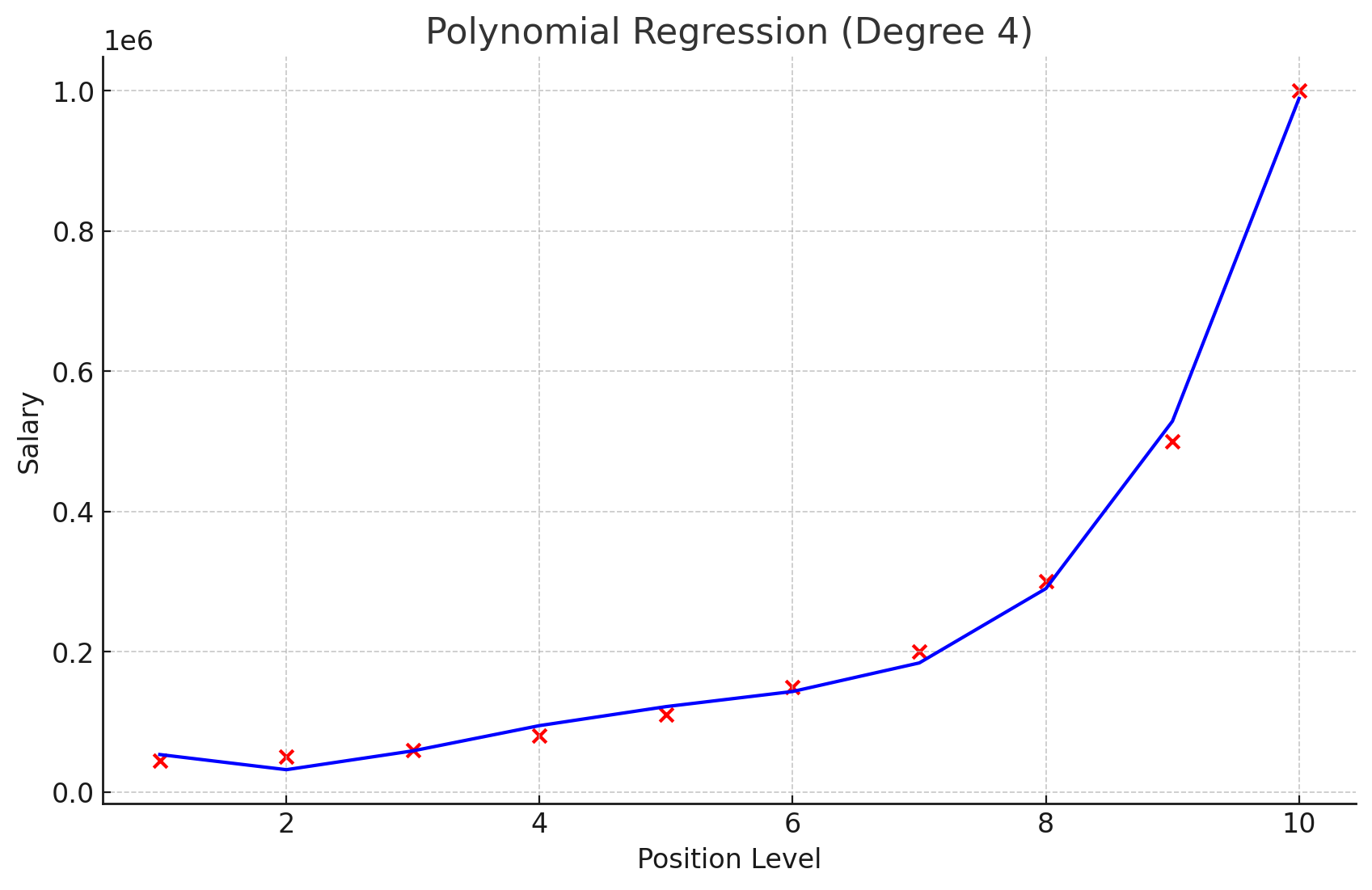
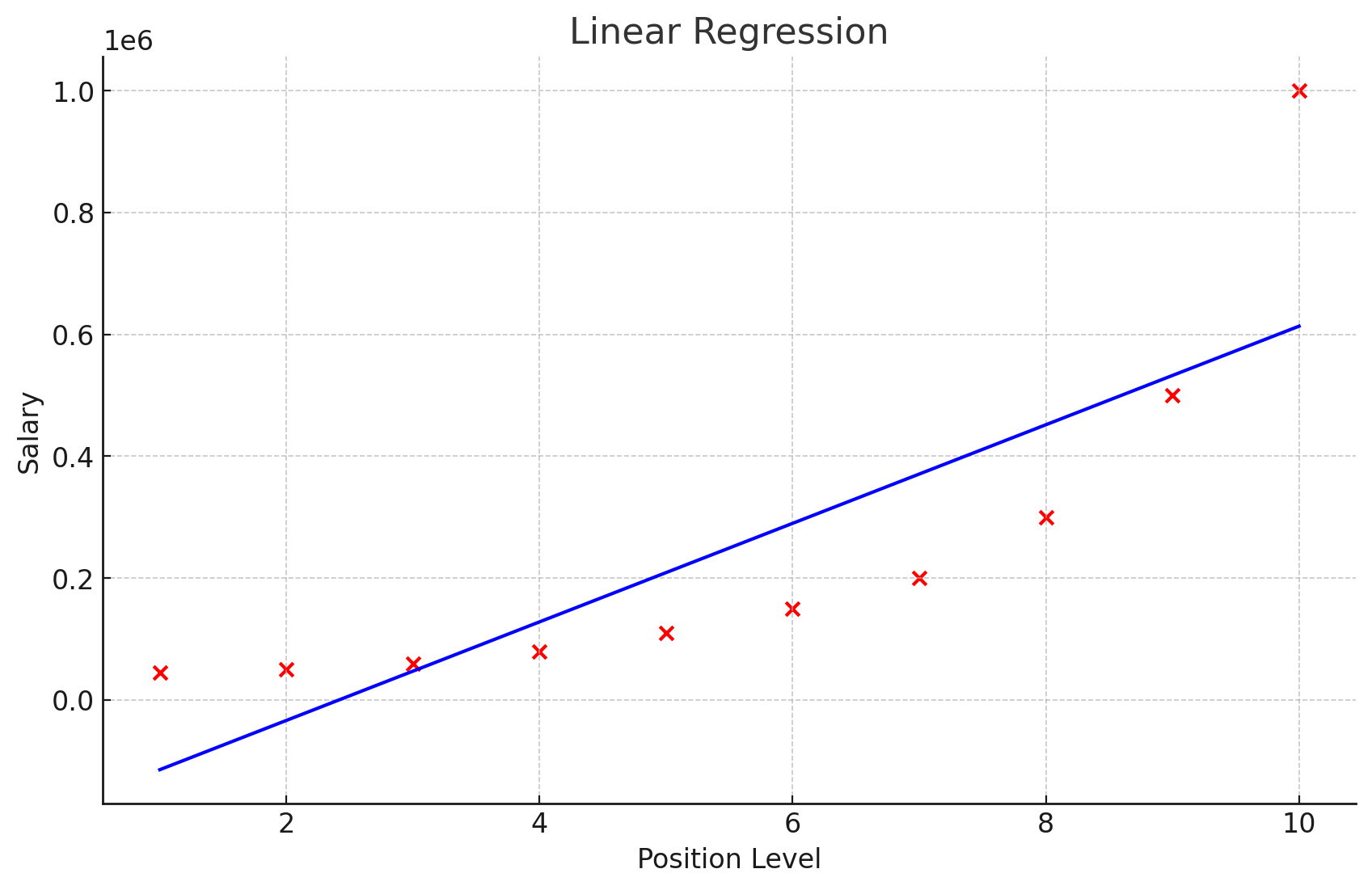
**Results:**

 **Linear Regression**:

* Mean Squared Error (MSE): 26,695,878,787.88
* R-squared (R²): 0.669

 **Polynomial Regression (Degree 4)**:

* Mean Squared Error (MSE): 210,343,822.84
* R-squared (R²): 0.997



**EXP 9: Build a neural network that will read the image of a digit and correctly identify the number.**

To build a neural network that reads an image of a digit and correctly identifies the number, we can use the popular MNIST dataset. This dataset contains images of handwritten digits (0-9) and is widely used for training image recognition systems.

We'll use the Keras library (from TensorFlow) to implement a simple neural network for digit classification.

Steps to Build the Neural Network:

Import Libraries: Use TensorFlow/Keras to build the neural network.

Load the Dataset: Load the MNIST dataset of handwritten digits.

Preprocess the Data: Normalize the image data and one-hot encode the labels.

Build the Neural Network Model: Define a multi-layer neural network.

Train the Model: Train the neural network on the training data.

Evaluate the Model: Test the model on the test data.

Make Predictions: Use the model to predict new digits.

**Key Components:**

Data Loading: The MNIST dataset is loaded using TensorFlow/Keras. The dataset contains 60,000 training images and 10,000 test images.

Preprocessing: We normalize the images by dividing the pixel values by 255, transforming the values into the range [0, 1]. The labels (digits) are one-hot encoded, so that each label is represented by a 10-element array (one for each digit).

Model Architecture:

The first layer flattens the 28x28 pixel images into a 1D array of 784 pixels.

A dense hidden layer with 128 neurons and ReLU activation is added.

The output layer has 10 neurons, corresponding to the digits 0-9, with softmax activation to output probabilities.

Compilation: The model uses the Adam optimizer, categorical cross-entropy loss, and accuracy as the evaluation metric.

Training: The model is trained for 5 epochs with a batch size of 32. Each epoch goes through the entire dataset once.

Evaluation: The model is evaluated on the test dataset to compute accuracy.

Prediction: The model is used to make predictions on the test data, and the predicted digit for the first test image is displayed.

install TensorFlow by running:

pip install tensorflow

***code to implement the neural network:***

# Import necessary libraries

import numpy as np

import tensorflow as tf

from tensorflow.keras import layers, models

import matplotlib.pyplot as plt

# Load the MNIST dataset

mnist = tf.keras.datasets.mnist

(X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data()

# Step 1: Preprocess the data

# Normalize the pixel values to be between 0 and 1

X\_train = X\_train / 255.0

X\_test = X\_test / 255.0

# One-hot encode the labels

y\_train = tf.keras.utils.to\_categorical(y\_train, 10)

y\_test = tf.keras.utils.to\_categorical(y\_test, 10)

# Step 2: Build the Neural Network Model

model = models.Sequential()

# Input layer (flatten the 28x28 images to 1D array)

model.add(layers.Flatten(input\_shape=(28, 28)))

# Hidden layer with 128 neurons and ReLU activation

model.add(layers.Dense(128, activation='relu'))

# Output layer with 10 neurons (one for each digit) and softmax activation

model.add(layers.Dense(10, activation='softmax'))

# Step 3: Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Step 4: Train the model

model.fit(X\_train, y\_train, epochs=5, batch\_size=32)

# Step 5: Evaluate the model on the test data

test\_loss, test\_accuracy = model.evaluate(X\_test, y\_test, verbose=2)

# Display the accuracy

print(f'Test accuracy: {test\_accuracy \* 100:.2f}%')

# Step 6: Make predictions

predictions = model.predict(X\_test)

# Show the first test image and prediction

plt.imshow(X\_test[0], cmap='gray')

plt.title(f'Predicted Label: {np.argmax(predictions[0])}')

plt.show()