Linear Regression

---

1. Introduction

Linear Regression is a **supervised learning** algorithm used for **predicting continuous values**. It assumes a linear relationship between the dependent variable (Y) and one or more independent variables (X). The goal is to find the best-fitting line that minimizes the error between the predicted and actual values.

---

2. Types of Linear Regression

A. Simple Linear Regression

It models the relationship between **one** independent variable (X) and a dependent variable (Y) using the equation:

$$Y = \beta_0 + \beta_1 X + \epsilon$$

where:

- Y = Dependent variable (Target)
- X = Independent variable (Feature)
- β_0 = Intercept (Bias)
- β_1 = Coefficient (Slope of the line)
- ϵ = Error term (Residual)

Example: Predicting house prices based on area (sq ft).

Finding the Best Line (Using Ordinary Least Squares - OLS)

The best-fit line minimizes the **Sum of Squared Errors (SSE)**:

$$SSE = \sum (Y_i - \hat{Y}_i)^2$$

where Y_i is the actual value and $\hat{Y}_i$ is the predicted value.

The optimal values of β_0 and β_1 are given by:

$$\beta_1 = \frac{\sum (X_i - \bar{X})(Y_i - \bar{Y})}{\sum (X_i - \bar{X})^2}$$

$$\beta_0 = \bar{Y} - \beta_1 \bar{X}$$

where $\bar{X}$ and $\bar{Y}$ are the means of X and Y.

B. Multiple Linear Regression

It extends simple regression by considering **multiple independent variables**.

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_n X_n + \epsilon$$

Example: Predicting house prices based on area, number of rooms, and location.

To estimate the coefficients β, we use **matrix notation**:

$$\mathbf{Y} = \mathbf{X}\beta + \epsilon$$

where:

- $\mathbf{Y}$ = Output vector
- $\mathbf{X}$ = Feature matrix
- β = Coefficients vector
- ϵ = Error term

The optimal solution is found using the **Normal Equation**:

$$\beta = (X^T X)^{-1} X^T Y$$

or by using **Gradient Descent** (iterative optimization technique).

---

3. Assumptions of Linear Regression

For the model to be valid, certain assumptions must hold:

1. **Linearity**: The relationship between X and Y is linear.
2. **Independence**: Observations are independent of each other.

3. **Homoscedasticity**: The variance of residuals remains constant across all values of X.
4. **Normality**: The residuals should follow a normal distribution.
5. **No Multicollinearity**: Independent variables should not be highly correlated.

---

4. Performance Metrics

To evaluate how well the model fits the data, we use the following metrics:

1. Mean Squared Error (MSE)

$MSE = \frac{1}{n} \sum (Y_i - \hat{Y}_i)^2$ Measures the average squared difference between actual and predicted values.

2. Root Mean Squared Error (RMSE)

$RMSE = \sqrt{MSE}$ Gives an interpretable error in the original units.

3. Mean Absolute Error (MAE)

$MAE = \frac{1}{n} \sum |Y_i - \hat{Y}_i|$ Measures the average absolute difference between actual and predicted values.

4. R-Squared (R^2)

$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$

where:

- $SS_{res} = \sum (Y_i - \hat{Y}_i)^2$ (Residual Sum of Squares)
- $SS_{tot} = \sum (Y_i - \bar{Y})^2$ (Total Sum of Squares)

R^2 represents the proportion of variance explained by the model.
$0 \leq R^2 \leq 1$, where **higher values** indicate a **better fit**.

Example: If $R^2 = 0.85$, it means 85% of the variation in Y is explained by X.

---

5. Advantages & Disadvantages

Advantages:

- Simple and interpretable
- Computationally efficient
- Works well when assumptions hold

Disadvantages:

- Assumes a **linear** relationship
- Sensitive to **outliers**
- Performs poorly with **correlated predictors** (multicollinearity)

---

6. Applications of Linear Regression

Business & Finance: Predicting stock prices, sales revenue
Healthcare: Estimating disease progression
Engineering: Forecasting energy consumption
Economics: Analyzing economic trends

Conclusion

Linear Regression is a **powerful** and **easy-to-use** technique for modeling relationships between variables. However, ensuring the assumptions hold is crucial for accurate predictions. When dealing with **non-linear** relationships or **correlated features**, advanced techniques like **Polynomial Regression**, **Ridge Regression**, or **Neural Networks** may be needed.

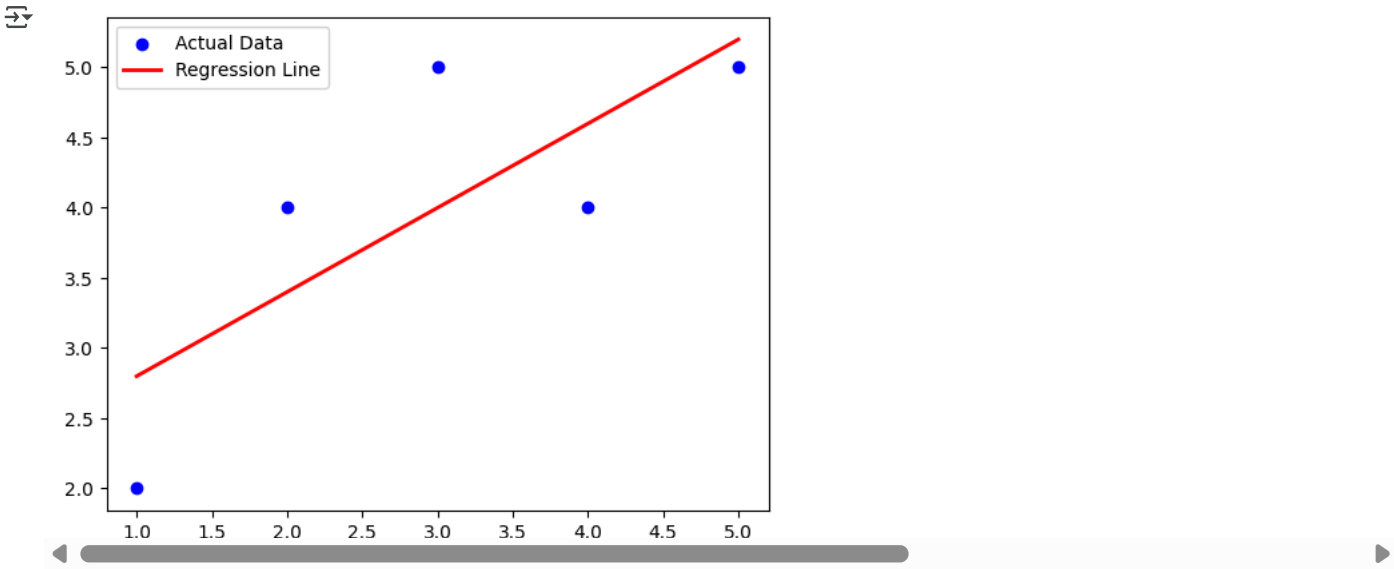
Simple Linear Regression Examples

Example 1

Surrogate data (small Dataset)

```
### **A. Using Scikit-Learn**
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Sample dataset
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
Y = np.array([2, 4, 5, 4, 5])

# Model training
model = LinearRegression()
model.fit(X, Y)

# Predictions
Y_pred = model.predict(X)

# Plotting
plt.scatter(X, Y, color='blue', label="Actual Data")
plt.plot(X, Y_pred, color='red', linewidth=2, label="Regression Line")
plt.legend()
plt.show()
```



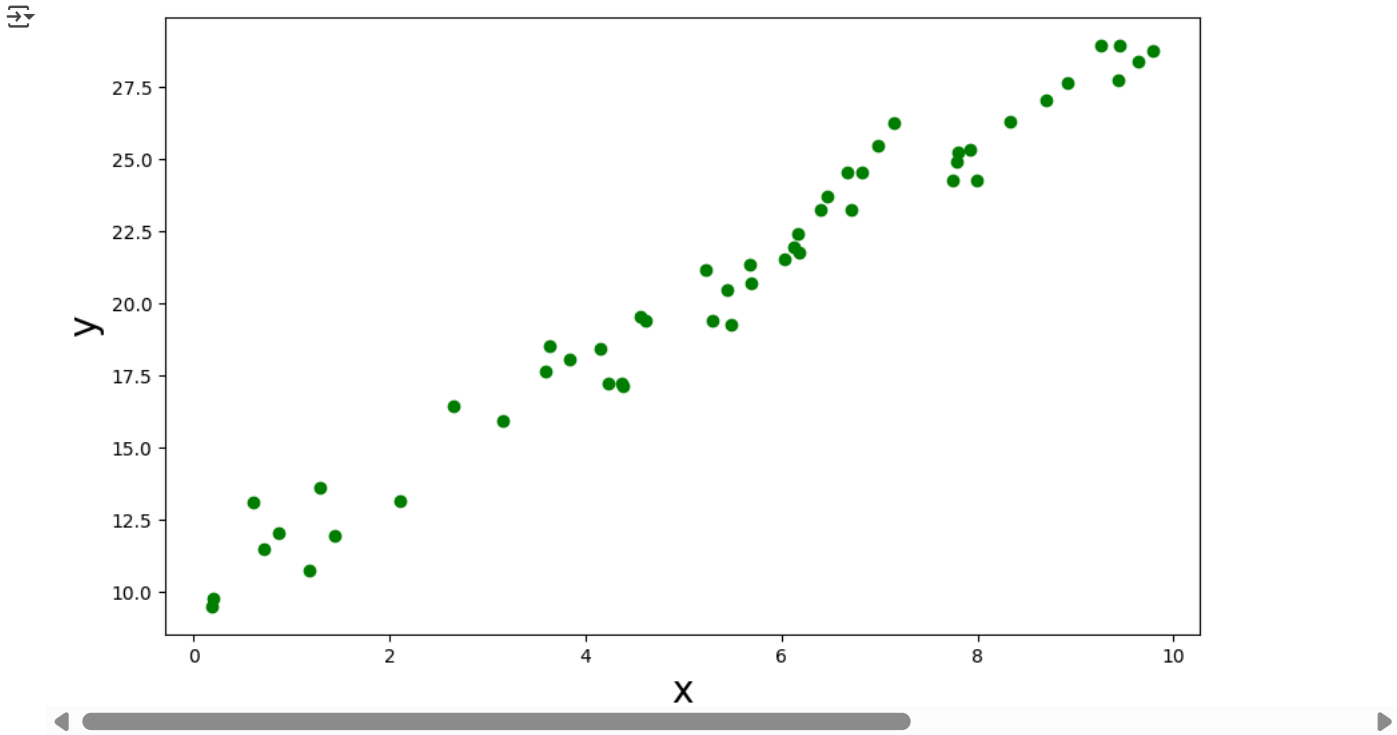
Example 2

Implementation of the Gradient Descent Algorithm

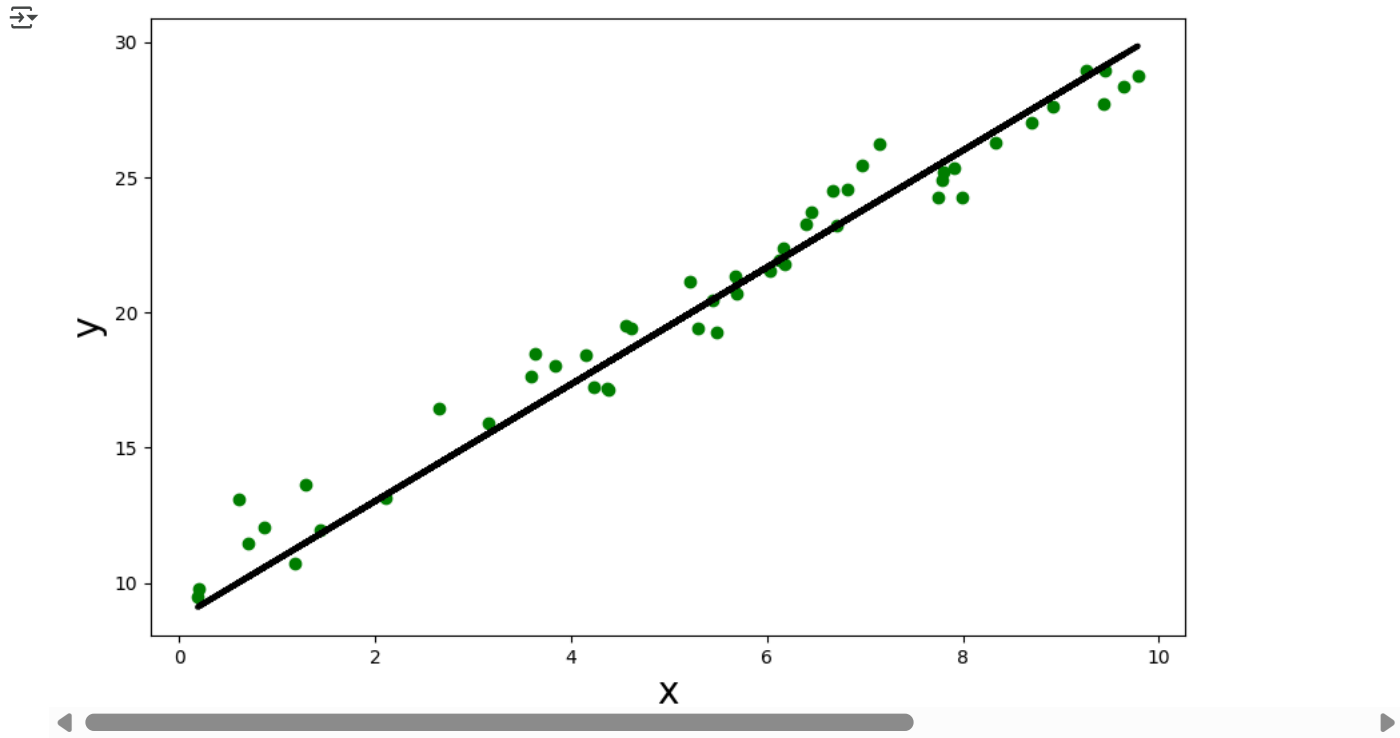
```
#Import required modules
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
#Defining the class
class LinearRegression:
    def __init__(self, x , y):
        self.data = x # Input (independent variable)
        self.label = y# Output (dependent variable)
        self.w = 0# Weight (slope), initialized to 0
        self.b = 0# Bias (y-intercept), initialized to 0
        self.m = len(x)# Number of data points

    def fit(self , epochs , lr):

        #Implementing Gradient Descent
        for i in range(epochs):
            y_pred = self.w * self.data + self.b

            #Calculating derivatives w.r.t Parameters
            D_w = (-2/self.m)*sum(self.data * (self.label - y_pred))
            D_b = (-1/self.m)*sum(self.label-y_pred)

            #Updating Parameters
            self.w = self.w - lr * D_w
            self.b = self.b - lr * D_b

    def predict(self , inp):
```

```
        y_pred = self.w * inp + self.b
        return y_pred
```

```
rng = np.random.RandomState(0)
X = 10 * rng.rand(50)
Y = 2 * X + 10 + rng.randn(50)

plt.figure(figsize = (10,6))
plt.scatter(X,Y , color = 'green')
plt.xlabel('x' , size = 20)
plt.ylabel('y', size = 20)
plt.show()
```




```
#Creating the class object
x = np.array(X)
y = np.array(Y)
regressor = LinearRegression(X,Y)

#Training the model with .fit method
regressor.fit(1000 , 0.01) # epochs-1000 , learning_rate - 0.0001

#Prediciting the values
y_pred = regressor.predict(x)

#Plotting the results
plt.figure(figsize = (10,6))
plt.scatter(x,y , color = 'green')
plt.plot(x , y_pred , color = 'k' , lw = 3)
plt.xlabel('x' , size = 20)
plt.ylabel('y', size = 20)
plt.show()
```



Example 3

Input as Catagorical Variables

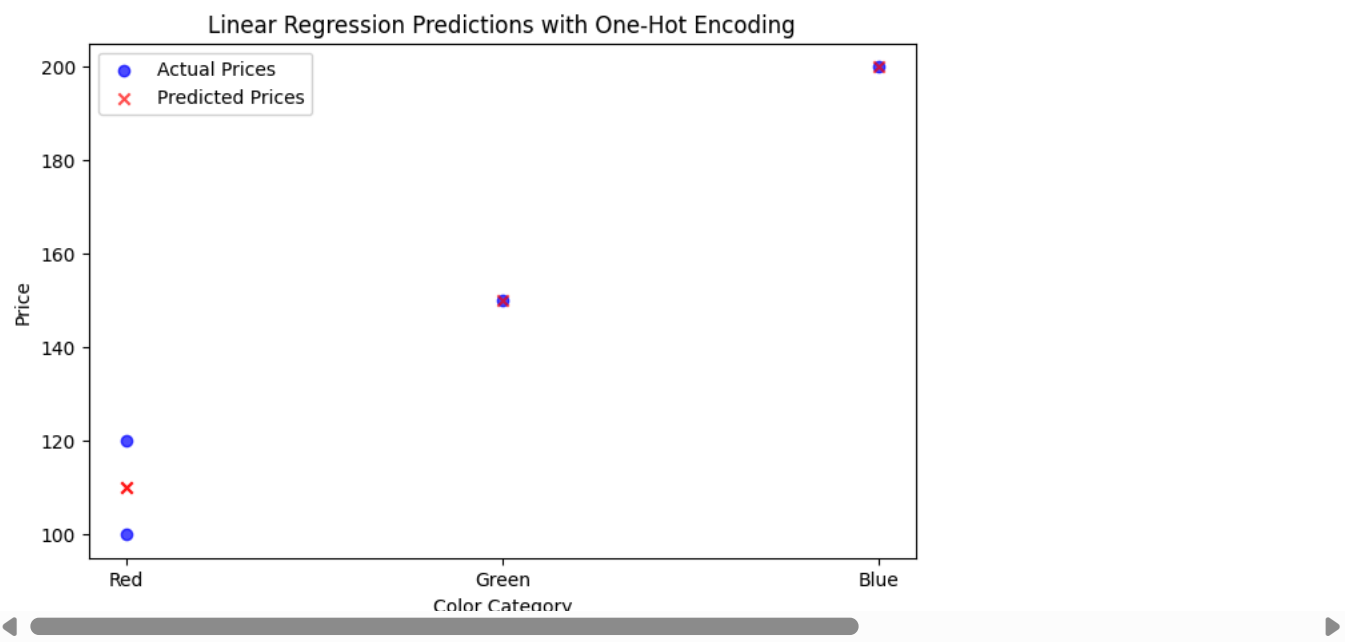
```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import OneHotEncoder

# Sample Data
data = pd.DataFrame({'Color': ['Red', 'Green', 'Blue', 'Red'], 'Price': [100, 150, 200, 120]})

# Apply One-Hot Encoding
# drop='first' → Avoids dummy variable trap by dropping one category. Ensures the output is an array, not a sparse matrix.
encoder = OneHotEncoder(drop='first', sparse_output=False)
encoded_features = encoder.fit_transform(data[['Color']])
```

```
# Convert to DataFrame
encoded_df = pd.DataFrame(encoded_features, columns=encoder.get_feature_names_out(['Color']))
print(encoded_df)
# Combine with Original Data
final_data = pd.concat([encoded_df, data['Price']], axis=1)
print(final_data)
# Train Linear Regression Model
X = final_data.drop('Price', axis=1)
y = final_data['Price']
model = LinearRegression()
model.fit(X, y)

# Print Coefficients
print("Coefficients:", model.coef_)
```

```
   Color_Green  Color_Red
0          0.0        1.0
1          1.0        0.0
2          0.0        0.0
3          0.0        1.0
   Color_Green  Color_Red  Price
0          0.0        1.0    100
1          1.0        0.0    150
2          0.0        0.0    200
3          0.0        1.0    120
Coefficients: [-50. -90.]
```

```
# Predict prices
predictions = model.predict(X)

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

```
plt.scatter(data['Color'], y, color='blue', label="Actual Prices", alpha=0.7)
plt.scatter(data['Color'], predictions, color='red', marker='x', label="Predicted Prices", alpha=0.7)
plt.xlabel("Color Category")
plt.ylabel("Price")
plt.title("Linear Regression Predictions with One-Hot Encoding")
plt.legend()
plt.show()
```




```
from sklearn.metrics import r2_score, mean_squared_error

y_pred = model.predict(X)
r2 = r2_score(y, y_pred)
mse = mean_squared_error(y, y_pred)

print(f"R² Score: {r2:.4f}")
print(f"Mean Squared Error: {mse:.2f}")
```

```
R² Score: 0.9648
Mean Squared Error: 50.00
```

Example 4

Bigger Surrogate dataset

```
import pandas as pd
import numpy as np
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.preprocessing import OneHotEncoder
from sklearn.metrics import mean_squared_error, r2_score

# Creating a larger dataset with multiple features
np.random.seed(42)
colors = np.random.choice(['Red', 'Green', 'Blue', 'Yellow'], size=50)  # Categorical feature
sizes = np.random.randint(1, 10, size=50)  # Continuous feature (Size)
weights = np.random.randint(50, 200, size=50)  # Continuous feature (Weight)
prices = 50 + 20*sizes + 3*weights + np.random.randint(-20, 20, size=50)  # Target variable

# Create a DataFrame
data = pd.DataFrame({'Color': colors, 'Size': sizes, 'Weight': weights, 'Price': prices})
data.describe()
```

	Size	Weight	Price
count	50.000000	50.000000	50.000000
mean	5.440000	121.980000	524.300000
std	2.643127	45.284358	152.037489
min	1.000000	50.000000	243.000000
25%	3.000000	82.500000	407.000000
50%	5.000000	113.500000	517.500000
75%	8.000000	167.250000	633.750000
max	9.000000	192.000000	801.000000

```
# Apply One-Hot Encoding to the "Color" column
# In summary, discarding one dummy variable when representing categorical variables ...
# in regression analysis helps to avoid multicollinearity, reduce redundancy, ...
# improve model interpretability, and avoid potential issues such as the dummy variable trap.
# It is a common practice in regression modeling when dealing with categorical data.

encoder = OneHotEncoder(drop='first', sparse_output=False)
encoded_features = encoder.fit_transform(data[['Color']])
```

```
# Convert to DataFrame
encoded_df = pd.DataFrame(encoded_features, columns=encoder.get_feature_names_out(['Color']))

# Combine Encoded Features with Numerical Features
final_data = pd.concat([encoded_df, data[['Size', 'Weight', 'Price']]], axis=1)

# Split into Training and Testing Data
X = final_data.drop('Price', axis=1)
y = final_data['Price']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train Multiple Linear Regression Model
model = LinearRegression()
model.fit(X_train, y_train)

# Predictions
y_pred_train = model.predict(X_train)
y_pred_test = model.predict(X_test)
```

```
# Model Performance Evaluation
train_mse = mean_squared_error(y_train, y_pred_train)
test_mse = mean_squared_error(y_test, y_pred_test)
train_r2 = r2_score(y_train, y_pred_train)
test_r2 = r2_score(y_test, y_pred_test)

# Print R² values
print(f"Train R²: {train_r2:.2f}")
print(f"Test R²: {test_r2:.2f}")
```

```
Train R²: 0.99
Test R²: 0.99
```

Interpretation of R² Values

$R^2 = 1$ → Perfect prediction (model explains 100% variance in y).

$R^2 > 0.7$ → Strong predictive power.

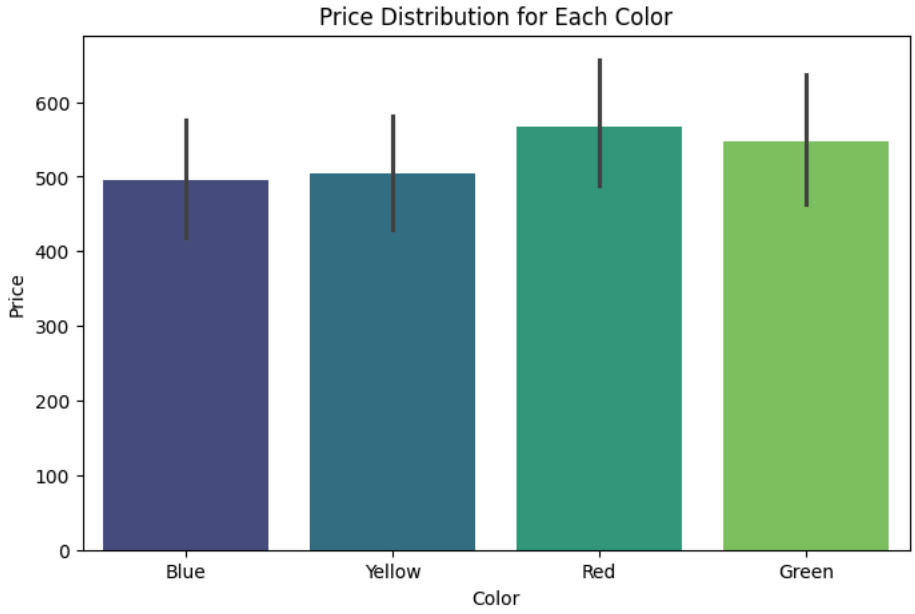
R^2 between 0.4 and 0.7 → Moderate prediction accuracy.

$R^2 < 0.4$ → Weak prediction power, model may need improvements.

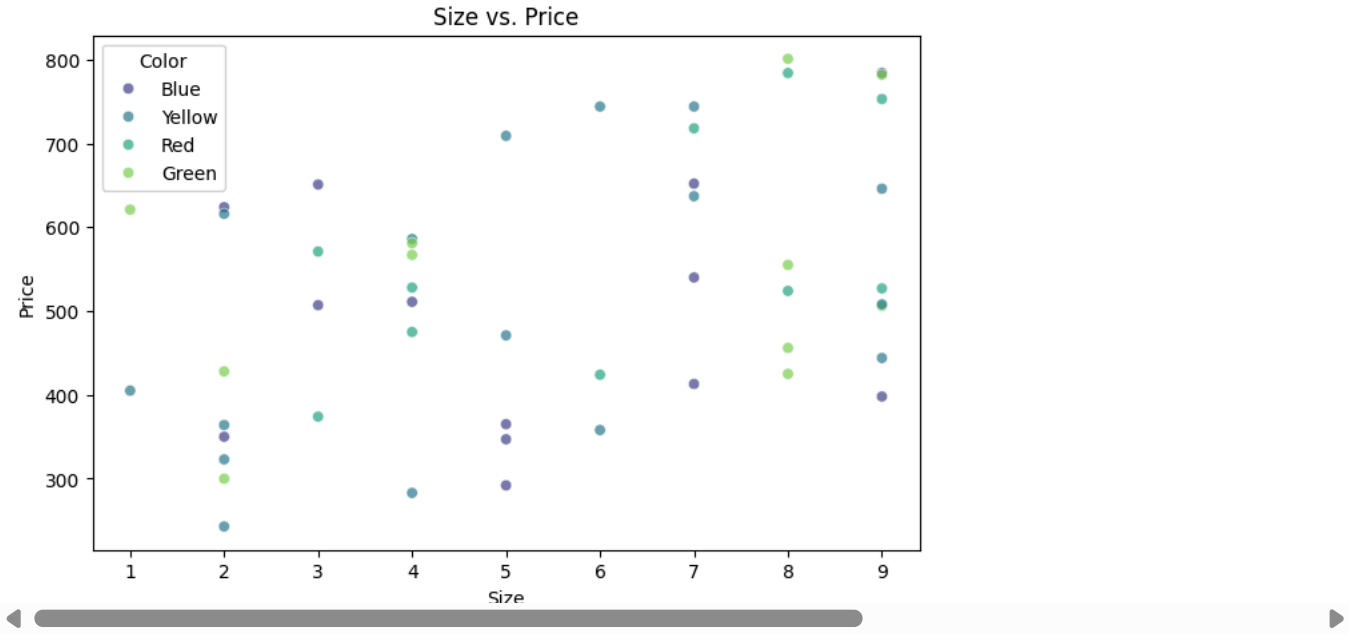
$R^2 < 0$ → Model performs worse than just predicting the mean value.

```
# ---------------------- PLOTTING ----------------------
# Bar Plot - Average Price per Color
plt.figure(figsize=(8, 5))
sns.barplot(x=data['Color'], y=data['Price'], palette="viridis", errorbar='ci')
plt.xlabel("Color")
plt.ylabel("Price")
plt.title("Price Distribution for Each Color")
plt.show()
```

```
<ipython-input-5-09908dfae241>:4: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `le

  sns.barplot(x=data['Color'], y=data['Price'], palette="viridis", errorbar='ci')
```



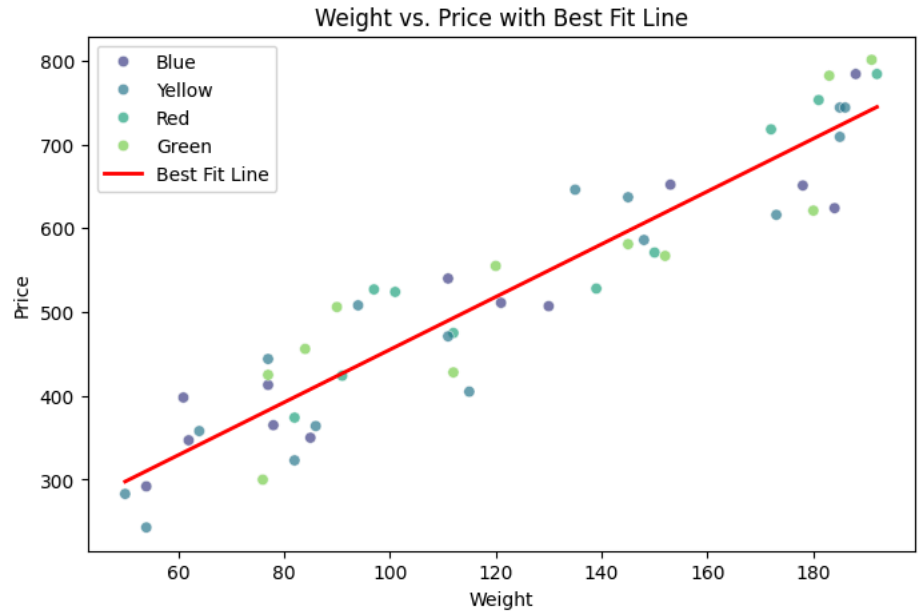

```
# Scatter Plot - Size vs. Price
plt.figure(figsize=(8, 5))
sns.scatterplot(x=data["Size"], y=data["Price"], hue=data["Color"], palette="viridis", alpha=0.7)
plt.xlabel("Size")
plt.ylabel("Price")
plt.title("Size vs. Price")
plt.show()
```




```
# Fit a linear regression model using Weight as the predictor
from sklearn.linear_model import LinearRegression

# Extract only Weight for simple linear regression
X_weight = data[['Weight']]  # Independent variable
y_price = data['Price']       # Dependent variable

# Train the model
weight_model = LinearRegression()
weight_model.fit(X_weight, y_price)

# Generate predictions (Best Fit Line)
weight_range = np.linspace(X_weight.min(), X_weight.max(), 100).reshape(-1, 1)  # Evenly spaced values for smooth line
price_pred = weight_model.predict(weight_range)  # Predict Price
```

```
# Scatter Plot with Best Fit Line
plt.figure(figsize=(8, 5))
sns.scatterplot(x=data["Weight"], y=data["Price"], hue=data["Color"], palette="viridis", alpha=0.7)
plt.plot(weight_range, price_pred, color='red', linewidth=2, label="Best Fit Line")  # Best Fit Line

# Labels and Title
plt.xlabel("Weight")
plt.ylabel("Price")
plt.title("Weight vs. Price with Best Fit Line")
plt.legend()
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/utils/validation.py:2739: UserWarning: X does not have valid feature names, but Line
  warnings.warn(
```



Example 5

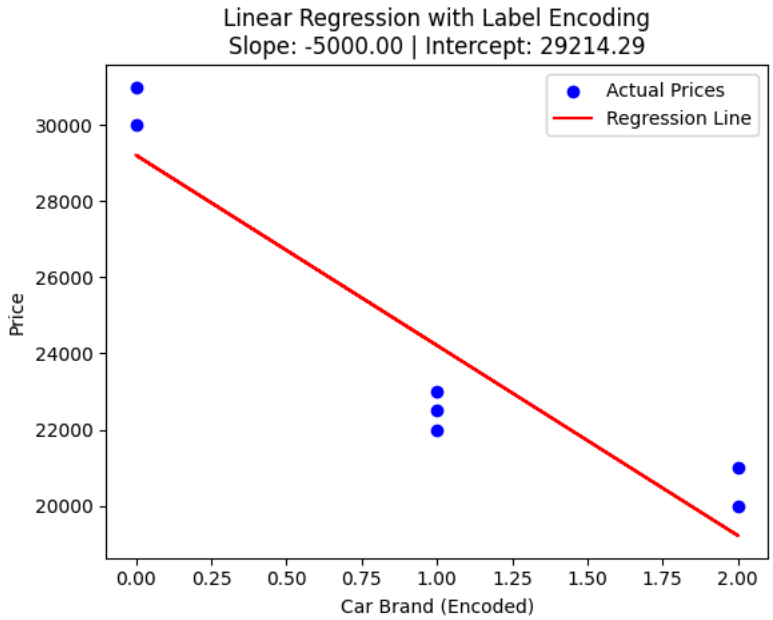
Real life dataset with Catagorial variable as input

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import LabelEncoder

# 1. Sample Data (with categorical and numerical variables)
data = {
    'Car_Brand': ['Toyota', 'Ford', 'BMW', 'Toyota', 'Ford', 'BMW', 'Ford'],
    'Price': [20000, 22000, 30000, 21000, 23000, 31000, 22500]
}
df = pd.DataFrame(data)

# 2. Label Encoding the Categorical Variable 'Car_Brand'
label_encoder = LabelEncoder()
df['Car_Brand_Encoded'] = label_encoder.fit_transform(df['Car_Brand'])

# Print the encoded DataFrame
print("Encoded DataFrame:")
display(df)

# 3. Define independent variable (X) and dependent variable (y)
X = df[['Car_Brand_Encoded']]  # Independent variable (label encoded column)
y = df['Price']  # Dependent variable

# 4. Perform Linear Regression
model = LinearRegression()
model.fit(X, y)

# 5. Get the regression line's coefficients
slope = model.coef_[0]
intercept = model.intercept_

# 6. Predict the target variable
```

```
y_pred = model.predict(X)

# 7. Visualize the results
plt.scatter(X, y, color='blue', label='Actual Prices')  # Actual data points
plt.plot(X, y_pred, color='red', label='Regression Line')  # Predicted regression line
plt.xlabel('Car Brand (Encoded)')
plt.ylabel('Price')
plt.legend()
plt.title(f'Linear Regression with Label Encoding\nSlope: {slope:.2f} | Intercept: {intercept:.2f}')
plt.show()

# Display the model's coefficients
print(f'Slope: {slope}')
print(f'Intercept: {intercept}')

mse = mean_squared_error(y, y_pred)
r2 = r2_score(y, y_pred)
print(f'\nMean Squared Error: {mse:.2f}')
print(f'R² Score: {r2:.4f}')
```

Encoded DataFrame:

	Car_Brand	Price	Car_Brand_Encoded
0	Toyota	20000	2
1	Ford	22000	1
2	BMW	30000	0
3	Toyota	21000	2
4	Ford	23000	1
5	BMW	31000	0
6	Ford	22500	1




```
Slope: -5000.0
Intercept: 29214.285714285714

Mean Squared Error: 2418367.35
```

```
Start coding or generate with AI.
```

Example 6

Sales Prediction (Real Dataset)

The aim is to build a model which predicts sales based on the money spent on different platforms such as TV, radio, and newspaper for marketing.

```
#Importing the libraries
import pandas as pd
```

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

```
#Reading the dataset
dataset = pd.read_csv("https://raw.githubusercontent.com/hirdeshiitkgp/Data/refs/heads/main/advertising.csv")
dataset.shape
```

```
(200, 4)
```

```
dataset.describe()
```

	TV	Radio	Newspaper	Sales
count	200.000000	200.000000	200.000000	200.000000
mean	147.042500	23.264000	30.554000	15.130500
std	85.854236	14.846809	21.778621	5.283892
min	0.700000	0.000000	0.300000	1.600000
25%	74.375000	9.975000	12.750000	11.000000
50%	149.750000	22.900000	25.750000	16.000000
75%	218.825000	36.525000	45.100000	19.050000
max	296.400000	49.600000	114.000000	27.000000

Data Pre-Processing

```
dataset.shape
```

```
(200, 4)
```

1. Checking for missing values

```
dataset.isna().sum()
```

	0
TV	0
Radio	0
Newspaper	0
Sales	0

Conclusion: The dataset does not have missing values

2. Checking for duplicate rows

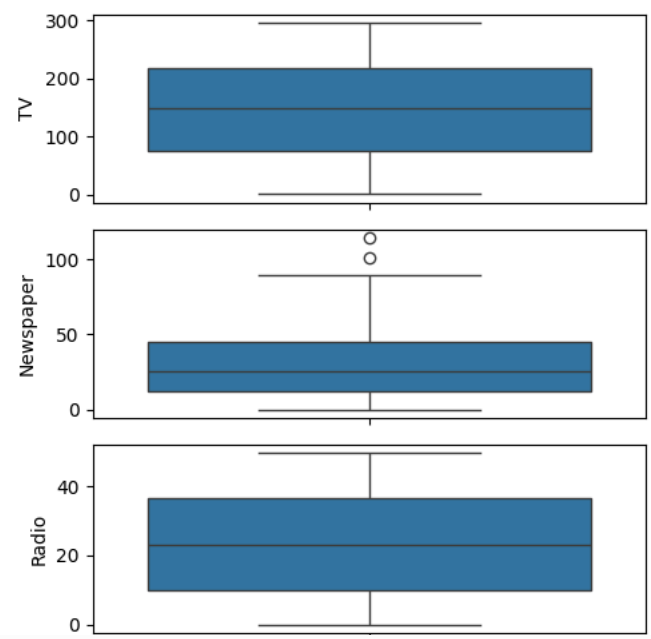
```
dataset.duplicated().any()
```

```
False
```

Conclusion: There are no duplicate rows present in the dataset

3. Checking for outliers

```
import seaborn as sns
fig, axs = plt.subplots(3, figsize = (5,5))
plt1 = sns.boxplot(dataset['TV'], ax = axs[0])
plt2 = sns.boxplot(dataset['Newspaper'], ax = axs[1])
plt3 = sns.boxplot(dataset['Radio'], ax = axs[2])
plt.tight_layout()
```



What is an Outlier?

In simple terms, an outlier is an extremely high or extremely low data point relative to the nearest data point and the rest of the neighboring co-existing values in a data graph or dataset you're working with.

Outliers are extreme values that stand out greatly from the overall pattern of values in a dataset or graph.

Conclusion: There are not that extreme values present in the dataset

```
sns.distplot(dataset['Newspaper'])
```

```
<ipython-input-8-91fb8808cc8c>:1: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(dataset['Newspaper'])
<Axes: xlabel='Newspaper', ylabel='Density'>
```

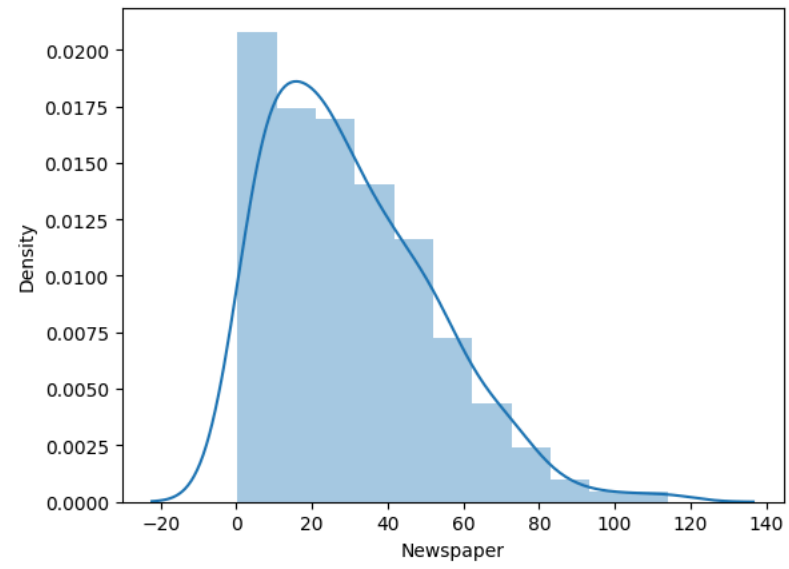


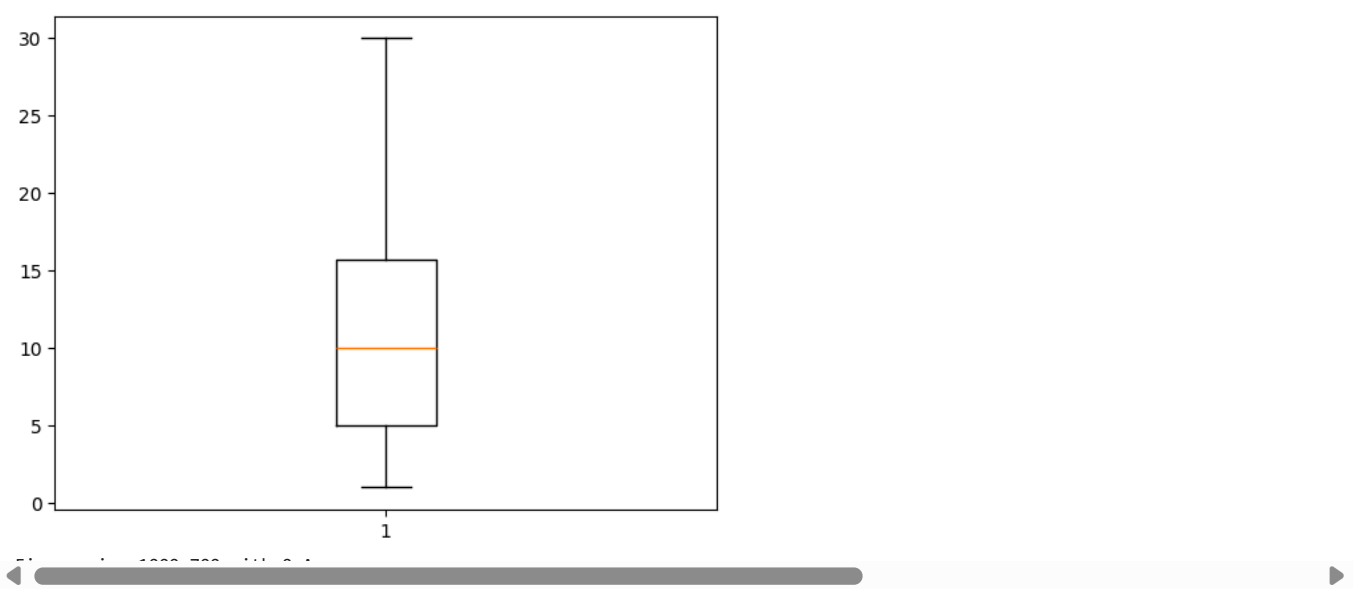
image.png

```
# Adding libraries
import numpy as np
```

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

# random integers between 1 to 20
arr = np.random.randint(1, 20, size=30)

# two outliers taken
arr1 = np.append(arr, [27, 30])

print('Thus the array becomes{}'.format(arr1))
```

```
Thus the array becomes[ 1  4 18 12  7  1 10 18 18 15  9  7 13 10  5  3 18 10  2  5 15  9 18  2
 13  3  5  9 15 18 27 30]
```

```
plt.boxplot(arr1)
fig = plt.figure(figsize =(10, 7))
plt.show()
```




```
# finding the 1st quartile
q1 = np.quantile(arr1, 0.25)

# finding the 3rd quartile
q3 = np.quantile(arr1, 0.75)
med = np.median(arr1)

# finding the iqr interquartile region
iqr = q3-q1

# finding upper and lower whiskers
# upper_bound = q3+(1.5*iqr)
# lower_bound = q1-(1.5*iqr)
print(iqr, q1,q3)
```

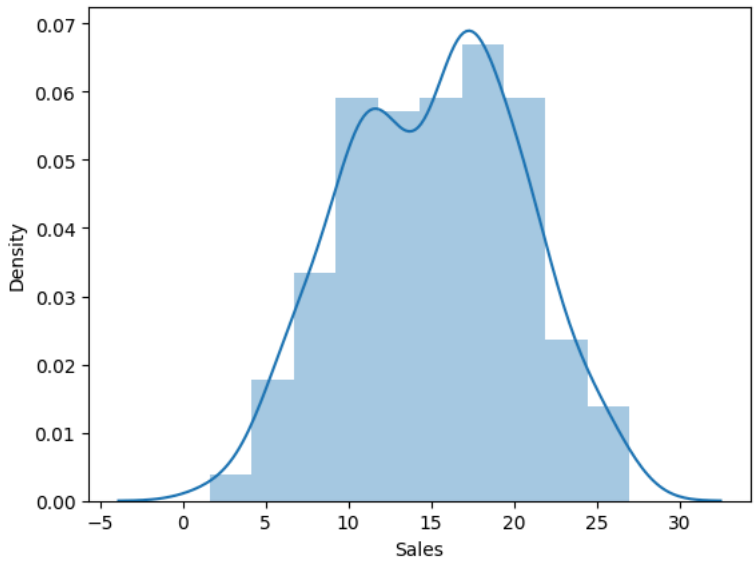
```
10.75 5.0 15.75
```

1. Distribution of the target variable

```
sns.distplot(dataset['Sales']);
```

```
<ipython-input-13-e26ae89dfd77>:1: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751
```

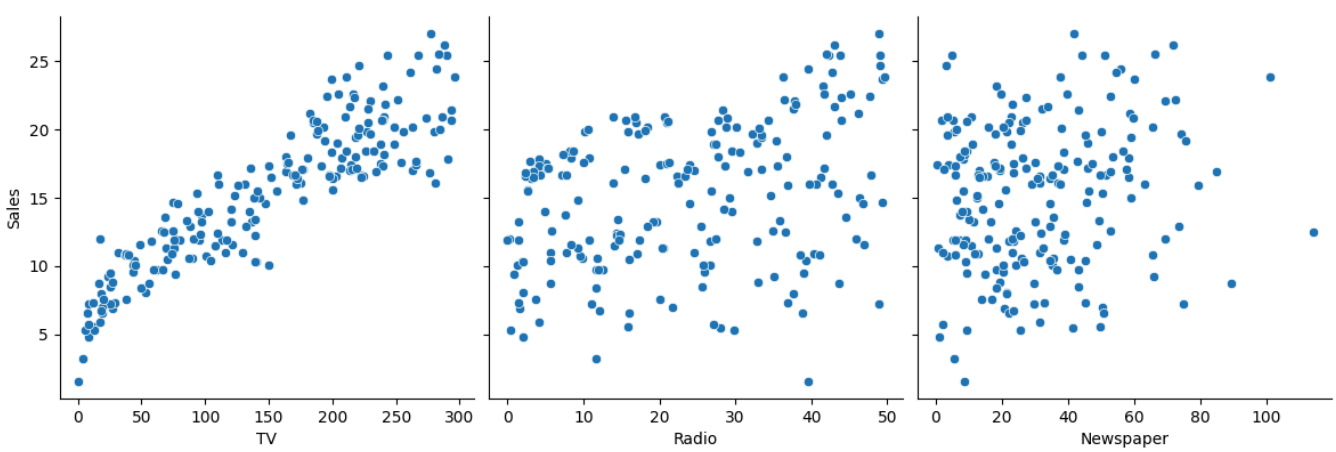
```
sns.distplot(dataset['Sales']);
```



Conclusion: It is normally distributed

2. How Sales are related with other variables

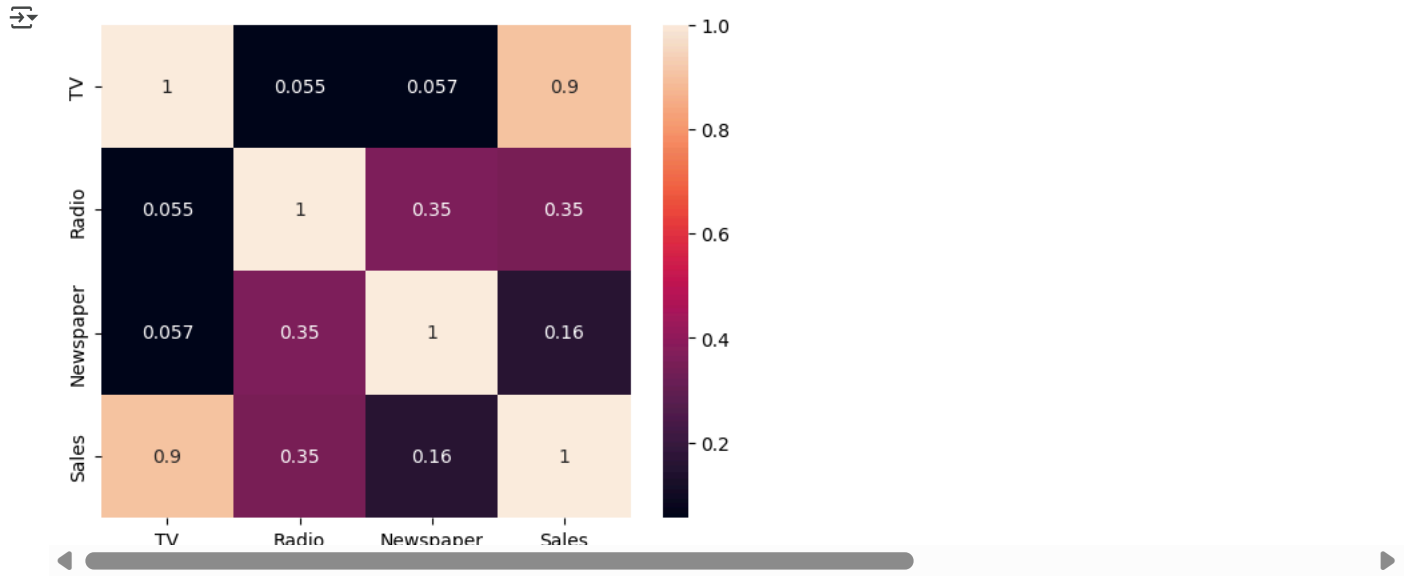
```
sns.pairplot(dataset, x_vars=['TV', 'Radio', 'Newspaper'], y_vars='Sales', height=4, aspect=1, kind='scatter')
plt.show()
```



Conclusion: TV is strongly, positively, linearly correlated with the target variable. Bu the Newspaper feature seems to be uncorrelated

3. Heatmap

```
sns.heatmap(dataset.corr(), annot = True)
plt.show()
```



Conclusion: TV seems to be most correlated with Sales as 0.9 is very close to 1

Model Definition, and Training

Defining The Problem

Defining the problem statement is the first step toward identifying what an ML model should achieve. “what is the main objective?”, “what is the input data?” and “what is the model trying to predict?” must be answered at this stage.

Data Collection

Data can be gathered from pre-existing databases or can be built from the scratch

Preparing The Data

The data preparation stage is when data is profiled, formatted and structured as needed to make it ready for training the model. Then data is categorized into two groups – one for training the ML model and the other for evaluating the model. Pre-processing of data by normalizing, eliminating duplicates and making error corrections is also carried out at this stage.

Assigning Appropriate Model / Protocols

Picking and assigning a model or protocol has to be done according to the objective that the ML model aims to achieve. There are several models to pick from, like linear regression, k-means, and bayesian depending on the type of data considered.

“The Model Training”

This is the stage where feeding datasets train the ML algorithm. This is the stage where the learning takes place. Consistent training can significantly improve the prediction rate of the ML model. The weights of the model must be initialized randomly. This way the algorithm will learn to adjust the weights accordingly.

Evaluating And Defining Measure Of Success

The machine model will have to be tested against the “validation dataset”. This helps assess the accuracy of the model. Identifying the measures of success based on what the model is intended to achieve is critical for justifying correlation.

Parameter Tuning

Selecting the correct parameter that will be modified to influence the ML model is key to attaining accurate correlation. The set of parameters that are selected based on their influence on the model architecture are called hyperparameters. The process of identifying the hyperparameters by tuning the model is called parameter tuning. The parameters for correlation should be clearly defined in a manner in which the point of diminishing returns for validation is as close to 100% accuracy as possible.

Model Building (Model Definition, and Training, Model Evaluation)

image.png

Linear Regression is a useful tool for predicting a quantitative response.

Prediction using: 1. Simple Linear Regression 2. Multiple Linear Regression

1. Simple Linear Regression

Simple linear regression has only one x and one y variable. It is an approach for predicting a quantitative response using a single feature.

It establishes the relationship between two variables using a straight line. Linear regression attempts to draw a line that comes closest to the data by finding the slope and intercept that define the line and minimize regression errors.

Formula: Y = β0 + β1X + e

```
Y = Dependent variable / Target variable
β0 = Intercept of the regression line
β1 = Slope of the regression lime which tells whether the line is increasing or decreasing
X = Independent variable / Predictor variable
e = Error
```

Equation: Sales = β0 + β1X; X = TV

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn import metrics
```

```
#Setting the value for X and Y
x = dataset[['TV']]
y = dataset['Sales']
```

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3, random_state = 105)
```

```
Start coding or generate with AI.
```

```
print(x_train.shape)
slr= LinearRegression()
slr.fit(x_train, y_train)
```

```
(140, 1)
▾ LinearRegression
LinearRegression()
```

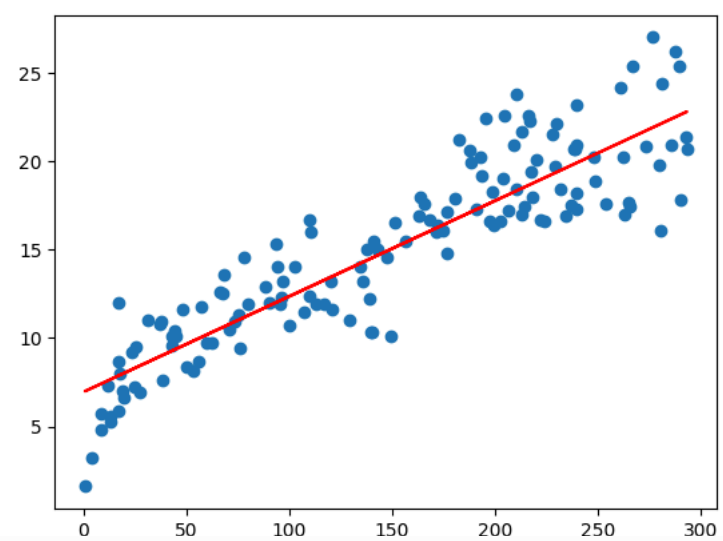
```
#Printing the model coefficients
print('Intercept: ', slr.intercept_)
print('Coefficient:', slr.coef_)
```

```
Intercept:  6.92719793182871
Coefficient: [0.05482896]
```

```
print('Regression Equation: Sales = 6.9 + 0.054 * TV')
```

```
Regression Equation: Sales = 6.948 + 0.054 * TV
```

```
#Line of best fit
plt.scatter(x_train, y_train)
plt.plot(x_train, 6.948 + 0.054*x_train, 'r')
plt.show()
```




```
#Prediction of Test and Training set result
y_pred_slr= slr.predict(x_test)
# x_pred_slr= slr.predict(x_train)
```

```
print("Prediction for test set: {}".format(y_pred_slr))
```

```
Prediction for test set: [18.73735667 11.0229215  20.26160185 17.76140112 16.07266904  8.30340492
 20.68378487 10.72136221 11.06678468 13.81919864 19.30757789 15.94656243
 17.06507328 19.00601859 11.70828355 11.04485309 22.47669198 18.27131048
  7.35486385 11.11064785 16.56064682  8.49530629 20.93051521 17.83267877
  7.32744937 19.12115941 16.25908752 11.11613074 12.27302187 19.38433844
 15.13509377  7.95249955  9.09294199  8.00184562 20.17935841  8.39661415
 14.11527505 14.14817242 19.4446503  13.67664334 19.94359386 17.06507328
 10.71039641  7.40420991 22.09837213 14.3400738  14.56487255  7.22327433
 23.17850271 18.97860411 11.84535596 13.53408803 22.51507225 12.66230751
 17.88202484 17.22956017 10.59525559 13.2873577  18.90732645 11.6260401 ]
```

```
#Actual value and the predicted value
slr_diff = pd.DataFrame({'Actual value': y_test, 'Predicted value': y_pred_slr})
slr_diff.head()
```

	Actual value	Predicted value
168	17.1	18.737357
37	14.7	11.022922
147	25.4	20.261602
96	16.7	17.761401
193	19.6	16.072669

```
#Predict for any value
slr.predict([[56]])
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/utils/validation.py:2739: UserWarning: X does not have valid feature names, but Line
  warnings.warn(
array([9.99761989])
```

Conclusion: The model predicted the Sales of 10.003 in that market

```
# print the R-squared value for the model
from sklearn.metrics import accuracy_score
print('R squared value of the model: {:.2f}'.format(slr.score(x,y)*100))
```

```
R squared value of the model: 81.14
```

Conclusion: 81.10% of the data fit the regression model

```
# 0 means the model is perfect. Therefore the value should be as close to 0 as possibl
meanAbErr = metrics.mean_absolute_error(y_test, y_pred_slr)
meanSqErr = metrics.mean_squared_error(y_test, y_pred_slr)
rootMeanSqErr = np.sqrt(metrics.mean_squared_error(y_test, y_pred_slr))
```

```
print('Mean Absolute Error:', meanAbErr)
print('Mean Square Error:', meanSqErr)
print('Root Mean Square Error:', rootMeanSqErr)
```

```
Mean Absolute Error: 1.748606163483661
Mean Square Error: 4.834209440005422
Root Mean Square Error: 2.1986835697765654
```

2. Multiple Linear Regression

Multiple linear regression has one y and two or more x variables. It is an extension of Simple Linear regression as it takes more than one predictor variable to predict the response variable.

Multiple Linear Regression is one of the important regression algorithms which models the linear relationship between a single dependent continuous variable and more than one independent variable.

Assumptions for Multiple Linear Regression: 1. A linear relationship should exist between the Target and predictor variables. 2. The regression residuals must be normally distributed. 3. MLR assumes little or no multicollinearity (correlation between the independent variable) in data.

Formula: Y = β0 + β1X1 + β2X2 + β3X3 + ... + βnXn + e

```
Y = Dependent variable / Target variable
β0 = Intercept of the regression line
β1, β2,..βn = Slope of the regression lime which tells whether the line is increasing or decreasing
X1, X2,..Xn = Independent variables / Predictor variables
e = Error
```

Equation: Sales = β0 + (β1 * TV) + (β2 * Radio) + (β3 * Newspaper)

```
#Setting the value for X and Y
x = dataset[['TV', 'Radio', 'Newspaper']]
y = dataset['Sales']
```

```
x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= 0.3, random_state=100)
```

```
mlr= LinearRegression()
mlr.fit(x_train, y_train)
```

```
LinearRegression
LinearRegression()
```

```
#Printing the model coefficients
print(mlr.intercept_)
# pair the feature names with the coefficients
list(zip(x, mlr.coef_))
```

```
4.334595861728431
[('TV', 0.053829108667250075),
 ('Radio', 0.11001224388558054),
 ('Newspaper', 0.0062899501461303325)]
```

```
#Predicting the Test and Train set result
y_pred_test= mlr.predict(x_test)
y_pred_train= mlr.predict(x_train)
```

```
Start coding or generate with AI.
```

```
# print the R-squared value for the model
print('R squared value of the model: {:.2f}'.format(mlr.score(x_train,y_train)*100))
```

```
R squared value of the model: 91.05
```