

Lakshya Gurung

```
[1] 0s
import pandas as pd
import numpy as np
```

- To - Do - 1:

1. Read and Observe the Dataset.
2. Print top(5) and bottom(5) of the dataset {Hint: pd.head and pd.tail}.
3. Print the Information of Datasets. {Hint: pd.info}.
4. Gather the Descriptive info about the Dataset. {Hint: pd.describe}
5. Split your data into Feature (X) and Label (Y).

```
[3] 0s
▶ df = pd.read_csv('/content/Houseprice_(1).csv');
print(df.head())
df.tail()
```

	HouseAge	HouseFloor	HouseArea	HousePrice
0	52	2	112.945574	543917.179841
1	93	1	174.312126	817740.124828
2	15	4	125.219577	387992.503019
3	72	4	121.210124	240840.742388
4	61	4	59.221737	277273.386525

	HouseAge	HouseFloor	HouseArea	HousePrice
95	85	1	120.193091	540347.535683
96	80	1	56.078992	222539.252789
97	82	2	211.368074	715211.774907

3	72	4	121.210124	240840.742388
4	61	4	59.221737	277273.386525

	HouseAge	HouseFloor	HouseArea	HousePrice
95	85	1	120.193091	540347.535683
96	80	1	56.078992	222539.252789
97	82	2	211.368074	715211.774907
98	53	4	94.277670	210926.069798
99	24	4	285.114646	988329.950912

```
▶ df.info()
```

```
... <class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   HouseAge    100 non-null    int64  
 1   HouseFloor  100 non-null    int64  
 2   HouseArea   100 non-null    float64 
 3   HousePrice  100 non-null    float64 
dtypes: float64(2), int64(2)
memory usage: 3.3 KB
```

```

    df.isnull().sum() #NO empty data

...
      0
HouseAge  0
HouseFloor 0
HouseArea  0
HousePrice  0

dtype: int64

df.describe()

  HouseAge  HouseFloor  HouseArea  HousePrice
count  100.000000  100.000000  100.000000  1.000000e+02
mean   51.540000  2.910000  170.226332  5.790964e+05
std    29.425963  1.457097  71.298821  2.615097e+05
min    2.000000  1.000000  51.265396  1.493562e+05
25%   23.500000  1.000000  109.770201  3.905859e+05
50%   54.000000  3.000000  176.846510  5.391941e+05
75%   76.500000  4.000000  222.794276  7.479899e+05
max   100.000000  5.000000  296.412614  1.228712e+06

```

```

[7]  ✓ 0s
    Age = df["HouseAge"].to_numpy().ravel()
    Floor = df["HouseFloor"].to_numpy().ravel()
    Area = df["HouseArea"].to_numpy().ravel()
    Price = df["HousePrice"].to_numpy().ravel()

[8]  ✓ 0s
    X = df[['HouseAge', 'HouseFloor', 'HouseArea']]  # features
    Y = df['HousePrice'] #Label
    W = np.array([-1.5, 4, 6])

[9]  ✓ 0s
    def scale_features(X):
        """Scale features (excluding intercept column)"""
        X_scaled = X.copy()
        for col in range(1, X.shape[1]): # Skip intercept column (column 0)
            mean = np.mean(X[:, col])
            std = np.std(X[:, col])
            if std > 0: # Avoid division by zero
                X_scaled[:, col] = (X[:, col] - mean) / std
            else:
                X_scaled[:, col] = 0
        return X_scaled

[10] ✓ 0s
    def scale_target(Y):
        """Scale target variable"""
        mean = np.mean(Y)
        std = np.std(Y)
        Y_scaled = (Y - mean) / std
        return Y_scaled, mean, std

```

TODO 2:

```
[11] ✓ 0s
    W = np.array([1, 1, 1]).reshape(-1,1)
    print(W.shape)
    print(X.shape)
    result= np.dot(X,W)
    print(result[:5])

    ▾ (3, 1)
    (100, 3)
    [[166.94557396]
     [268.31212647]
     [144.21957745]
     [197.21012359]
     [124.22173684]]
```

TODO 3:

```
[12] ✓ 0s
    def split(X, Y, test_split=0.2, seed=42):
        np.random.seed(seed)
        indices = np.arange(X.shape[0])
        np.random.shuffle(indices)

        split_size = int(len(X) * test_split)

        test_indices = indices[:split_size]
        train_indices = indices[split_size:]

        X_train, X_test = X[train_indices], X[test_indices]
        Y_train, Y_test = Y[train_indices], Y[test_indices]

        return X_train, X_test, Y_train, Y_test
```

TODO 4:

```
[13] ✓ 0s
    ⏎ x0 = np.ones(len(Age))
    X4 = np.array([x0, Age, Floor, Area]).T
    W4 = np.array([0.0, 0.0, 0.0, 0.0])
    Y4 = Price.ravel()

    X4_scaled = scale_features(X4)
    Y4_scaled, Y_mean, Y_std = scale_target(Y4)

[14] ✓ 0s
    def cost_function(X, Y, W):
        """ Parameters:
            This function finds the Mean Square Error.
        Input parameters:
            X: Feature Matrix
            Y: Target Matrix
            W: Weight Matrix
        Output Parameters:
            J: accumulated mean square error.
        """
        m = len(Y)

        J = np.sum((X.dot(W) - Y) ** 2)/(2 * m)
        return J
```

```
[15] ✓ Os
    # Test case
    X_test = np.array([[1, 2],
                      [3, 4],
                      [5, 6]])  #3X2
    Y_test = np.array([3, 7, 11])  #1X3
    W_test = np.array([1, 1])  # 1X2

    cost = cost_function(X_test, Y_test, W_test)

    if cost == 0:
        print("Proceed Further")
    else:
        print("Something went wrong: Reimplement the cost function")
        print("Cost function output:", cost)

    Proceed Further

[16] ✓ Os
    initial_cost = cost_function(X4, Y4, W4)
    print(initial_cost)

    201528080199.132

[17] ✓ Os
    print(X4.shape)
    print(Y4.shape)
    print(W4.shape)

    (100, 4)
    (100,)
    (4,)
```

TODO 6:

```
[18] ✓ Os
def gradient_descent(X, Y, B, alpha, iterations):
    cost_history = [0] * iterations
    m = len(Y)

    for iteration in range(iterations):
        # Forward pass
        Y_pred = X.dot(B)
        loss = Y_pred - Y

        # Gradient calculation
        gradient = (X.T.dot(loss)) / m

        # Update weights
        B = B - alpha * gradient  # FIX: Update B, not create W_update

        # Calculate and store cost
        cost_history[iteration] = cost_function(X, Y, B)

    return B, cost_history
```

```
[19] ✓ 0s
def rmse(Y, Y_pred):
    """
    This Function calculates the Root Mean Squares.
    Input Arguments:
        Y: Array of actual(Target) Dependent Variables.
        Y_pred: Array of predicted Dependent Variables.
    Output Arguments:
        rmse: Root Mean Square.
    """
    rmse = np.sqrt(sum((Y-Y_pred)**2)/len(Y))
    return rmse

[20] ✓ 0s
▶ def r2(Y, Y_pred):
    """
    This Function calculates the R Squared Error.
    Input Arguments:
        Y: Array of actual(Target) Dependent Variables.
        Y_pred: Array of predicted Dependent Variables.
    Output Arguments:
        rsquared: R Squared Error.
    """
    mean_y = np.mean(Y)
    ss_tot = sum((Y - mean_y) ** 2)
    ss_res = sum((Y - Y_pred) ** 2)
    r2 = 1 - (ss_res / ss_tot)
    return r2
```

```
[21] ✓ 0s
def simple_normalize(arr):
    """Normalize array to [0, 1] range"""
    return (arr - np.min(arr)) / (np.max(arr) - np.min(arr))

X4_norm = X4.copy()
Y4_norm = simple_normalize(Y4)

[22] ✓ 0s
▶ for i in range(1, X4_norm.shape[1]):
    X4_norm[:, i] = simple_normalize(X4[:, i])

    print("== DATA NORMALIZED TO [0, 1] RANGE ==")
    print(f"X ranges: [{np.min(X4_norm[:, 1:]):.3f}, {np.max(X4_norm[:, 1:]):.3f}]")
    print(f"Y range: [{np.min(Y4_norm):.3f}, {np.max(Y4_norm):.3f}]")
    ...
    == DATA NORMALIZED TO [0, 1] RANGE ==
    X ranges: [0.000, 1.000]
    Y range: [0.000, 1.000]

[23] ✓ 0s
    initial_cost = cost_function(X4, Y4, W4)
    print(f"Initial cost: {initial_cost:.2f}")

    Initial cost: 201528080199.13
```

```
[24] 0s
▶ alpha = 0.000025
new_weights, cost_history = gradient_descent(X4, Y4, W4, alpha, 10000)

print(f"\nOptimized weights:")
print(f"Intercept (b0): {new_weights[0]:.4f}")
print(f"HouseAge coefficient (b1): {new_weights[1]:.4f}")
print(f"HouseFloor coefficient (b2): {new_weights[2]:.4f}")
print(f"HouseArea coefficient (b3): {new_weights[3]:.4f}")
print(f"Final cost: {cost_history[-1]:.2f}")

...
Optimized weights:
Intercept (b0): 1982.9551
HouseAge coefficient (b1): 472.6966
HouseFloor coefficient (b2): -4256.0767
HouseArea coefficient (b3): 3270.8719
Final cost: 12016392087.97

[25] 0s
Y_pred = X4.dot(new_weights)

print(f"\nModel Performance:")
print(f"RMSE: {rmse(Y4, Y_pred):.2f}")
print(f"R-squared: {r2(Y4, Y_pred):.4f}")

...
Model Performance:
RMSE: 155025.11
R-squared: 0.6450
```

```
[26] 0s
▶ print("\nFirst 5 predictions vs actual:")
for i in range(5):
    print(f"Actual: {Y4[i]:.2f}, Predicted: {Y_pred[i]:.2f}, Difference: {abs(Y4[i] - Y_pred[i]):.2f}")

...
First 5 predictions vs actual:
Actual: 543917.18, Predicted: 387481.53, Difference: 156435.65
Actual: 817740.12, Predicted: 611840.30, Difference: 205899.82
Actual: 387992.50, Predicted: 401626.30, Difference: 13633.79
Actual: 240840.74, Predicted: 415455.59, Difference: 174614.85
Actual: 277273.39, Predicted: 207499.86, Difference: 69773.53

[27] 0s
new_weights_scaled, cost_history = gradient_descent(
    X4_scaled, Y4_scaled, W4, alpha, 10000
)

[28] 0s
print(f"\nModel Performance:")
print(f"RMSE: {rmse(Y4, Y_pred):.2f}")
print(f"R-squared: {r2(Y4, Y_pred):.4f}")

...
Model Performance:
RMSE: 155025.11
R-squared: 0.6450
```

```
[30]  ✓ 0s
▶ print("\nFirst 5 predictions vs actual:")
for i in range(5):
    print(f"Actual: {Y4[i]:.2f}, Predicted: {Y_pred[i]:.2f}, Difference: {abs(Y4[i] - Y_pred[i]):.2f}")

# Also show cost reduction
print("\nCost reduction:")
print(f"Initial cost: {cost_history[0]:.4f}")
print(f"Final cost: {cost_history[-1]:.4f}")

...
First 5 predictions vs actual:
Actual: 543917.18, Predicted: 387481.53, Difference: 156435.65
Actual: 817740.12, Predicted: 611840.30, Difference: 205899.82
Actual: 387992.50, Predicted: 401626.30, Difference: 13633.79
Actual: 240840.74, Predicted: 415455.59, Difference: 174614.85
Actual: 277273.39, Predicted: 207499.86, Difference: 69773.53

Cost reduction:
Initial cost: 0.5000
Final cost: 0.3663
```