ASSIGNMENT 3 - REGRESSION

The dataset "California Housing.csv" contains various features related to houses in California, including geographical data and the corresponding median house prices. The primary goal of this project is to design and implement a comprehensive regression system that addresses key challenges in predicting house prices, such as missing values, feature scaling, and model selection. By applying effective regression algorithms, the objective is to analyze the median house price based on various features and enhance the overall quality, reliability, and usability of the data for further analysis and machine learning applications. This task will focus on implementing and comparing multiple regression techniques to find the best model for predicting house prices.

SOURCE

The California Housing dataset used for this project is available in the sklearn library. It can be loaded using the fetch_california_housing() function from sklearn.datasets.

IMPORTING MODULES

In [2]:
```
import pandas as pd
import numpy as np
```

```python
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
import sys
if not sys.warnoptions:
    warnings.simplefilter("ignore")
```

LOADING & PREPROCESSING

1. LOAD THE DATA AND CONVERT INTO DATA FRAME

In [15]:

```python
# LOAD THE DATASET
from sklearn.datasets import fetch_california_housing

data = fetch_california_housing()

# EXTRACT THE FEATURES (X) AND TARGET VARIABLE (Y)
X = pd.DataFrame(data.data, columns=data.feature_names)
Y = pd.Series(data.target)
```

2. DISPLAY FIRST & LAST ROWS

In [17]:

```python
# DISPLAY FIRST FEW ROWS TO UNDERSTAND THE STRUCTURE OF THE DATA
print(X.head())
print("\nTarget Variable (Prices) Sample:")
print(Y.head())
```

```
   MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0  8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88
1  8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86
2  7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85
3  5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85
4  3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85

   Longitude
0    -122.23
1    -122.22
2    -122.24
3    -122.25
4    -122.25

Target Variable (Prices) Sample:
0    4.526
1    3.585
2    3.521
3    3.413
4    3.422
dtype: float64
```

In [19]:

```
# DISPLAY LAST FEW ROWS TO UNDERSTAND THE STRUCTURE OF THE DATA
print(X.tail())
print("\nTarget Variable (Prices) Sample:")
print(Y.tail())
```

```
       MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
20635  1.5603      25.0  5.045455   1.133333       845.0  2.560606     39.48
20636  2.5568      18.0  6.114035   1.315789       356.0  3.122807     39.49
20637  1.7000      17.0  5.205543   1.120092      1007.0  2.325635     39.43
20638  1.8672      18.0  5.329513   1.171920       741.0  2.123209     39.43
20639  2.3886      16.0  5.254717   1.162264      1387.0  2.616981     39.37

       Longitude
20635    -121.09
20636    -121.21
20637    -121.22
20638    -121.32
20639    -121.24

Target Variable (Prices) Sample:
20635    0.781
20636    0.771
20637    0.923
20638    0.847
20639    0.894
dtype: float64
```

3. DATATYPE OF EACH COLUMN

In [38]:
```
# DISPLAY DATA TYPE OF EACH COLUMN
print("Dataset Info:")
X.info()
```

```
Dataset Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   MedInc      20640 non-null  float64
 1   HouseAge    20640 non-null  float64
 2   AveRooms    20640 non-null  float64
 3   AveBedrms   20640 non-null  float64
 4   Population  20640 non-null  float64
 5   AveOccup    20640 non-null  float64
 6   Latitude    20640 non-null  float64
 7   Longitude   20640 non-null  float64
dtypes: float64(8)
memory usage: 1.3 MB
```

In [36]:

```
# DISPLAY DATA TYPE OF EACH COLUMN
print("Dataset Info:")
Y.info()
```

```
Dataset Info:
<class 'pandas.core.series.Series'>
RangeIndex: 20640 entries, 0 to 20639
Series name: None
Non-Null Count  Dtype  
--------------  -----  
20640 non-null  float64
dtypes: float64(1)
memory usage: 161.4 KB
```

4. STATISTICAL SUMMARY OF DATA

In [34]:

```
# DISPLAY STATISTICAL SUMMARY
print("Statistical Summary:")
X.describe()
```

```
Statistical Summary:
```

Out[34]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
count	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000
mean	3.870671	28.639486	5.429000	1.096675	1425.476744	3.070655	35.631861	-119.569704
std	1.899822	12.585558	2.474173	0.473911	1132.462122	10.386050	2.135952	2.003532
min	0.499900	1.000000	0.846154	0.333333	3.000000	0.692308	32.540000	-124.350000
25%	2.563400	18.000000	4.440716	1.006079	787.000000	2.429741	33.930000	-121.800000
50%	3.534800	29.000000	5.229129	1.048780	1166.000000	2.818116	34.260000	-118.490000
75%	4.743250	37.000000	6.052381	1.099526	1725.000000	3.282261	37.710000	-118.010000
max	15.000100	52.000000	141.909091	34.066667	35682.000000	1243.333333	41.950000	-114.310000

In [42]:

```
# DISPLAY STATISTICAL SUMMARY
print("Statistical Summary:")
Y.describe()
```

```
Statistical Summary:
```

Out[42]:

```
count    20640.000000
mean         2.068558
std          1.153956
min          0.149990
25%          1.196000
50%          1.797000
75%          2.647250
max          5.000010
dtype: float64
```

5. DISPLAY ALL COLUMN NAMES

In [44]:

```
# DISPLAY PARTICULAR COLUMN
print("Columns of the dataset:")
X.columns
```

```
Columns of the dataset:
```

Out[44]:
```
Index(['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Population', 'AveOccup',
       'Latitude', 'Longitude'],
      dtype='object')
```

6. NULL / MISSING VALUES IN EACH COLUMN

In [51]:
```
# DISPLAY NULL VALUES IN EACH COLUMN
print("Null values in each column:")
print(X.isnull().sum())
```

```
Null values in each column:
MedInc        0
HouseAge      0
AveRooms      0
AveBedrms     0
Population    0
AveOccup      0
Latitude      0
Longitude     0
dtype: int64
```

7. DUPLICATE VALUES

In [163…
```
# FINDING THE TOTAL NO OF DUPLICATES
X.duplicated().sum()
```

Out[163…
```
0
```

8. FEATURE SCALING

In [67]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler() # OBJECT CREATION

X_scaled = scaler.fit_transform(X)
```

```
print("\nScaled Feature Data (First 5 rows):")
print(X_scaled[:5])
```

```
Scaled Feature Data (First 5 rows):
[[ 2.34476576  0.98214266  0.62855945 -0.15375759 -0.9744286  -0.04959654
   1.05254828 -1.32783522]
 [ 2.33223796 -0.60701891  0.32704136 -0.26333577  0.86143887 -0.09251223
   1.04318455 -1.32284391]
 [ 1.7826994   1.85618152  1.15562047 -0.04901636 -0.82077735 -0.02584253
   1.03850269 -1.33282653]
 [ 0.93296751  1.85618152  0.15696608 -0.04983292 -0.76602806 -0.0503293
   1.03850269 -1.33781784]
 [-0.012881    1.85618152  0.3447108  -0.03290586 -0.75984669 -0.08561576
   1.03850269 -1.33781784]]
```

9. SPLITTING THE DATA INTO TRAINING AND TESTING SET

In [167…

```
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X_scaled, Y, test_size=0.2, random_state=42)

print("\nTraining set shape of X: ", X_train.shape)
print("Test set shape of X:", X_test.shape)
print("\nTraining set shape of Y", Y_train.shape)
print("Test set shape of Y:", Y_test.shape)
```

```
Training set shape of X:  (16512, 8)
Test set shape of X: (4128, 8)

Training set shape of Y (16512,)
Test set shape of Y: (4128,)
```

Preprocessing steps with explanations:

1. *Load the Data*:

   - Used the `fetch_california_housing` function to load the dataset containing features (X) and target values (Y).

2. *Convert to DataFrame*:
   - Converted the data into pandas DataFrames for easier manipulation and analysis.
3. *Display First and Last Rows*:
   - Displayed the first few rows of X and Y to understand the data structure and confirm it loaded correctly.
4. *Check Data Types*:
   - Used `info()` to check the data types of the columns and ensure they are as expected (numerical values).
5. *Statistical Summary*:
   - Used `describe()` to view statistics (mean, min, max, etc.) of both features and target to understand their distribution.
6. *Display Column Names*:
   - Printed the column names of the features to know what variables we are working with.
7. *Check for Missing Values*:
   - Checked for missing values with `isnull().sum()` to ensure the dataset is complete.
8. *Find Duplicate Rows*:

- Checked for duplicate rows using `duplicated().sum()` to ensure there are no repeated records.

9. *Feature Scaling*:

   - Scaled the features using `StandardScaler` to ensure that all features are on the same scale, which is important for some machine learning models.

10. *Train-Test Split*:

    - Split the data into training and testing sets to evaluate the model's performance on unseen data.

These steps are necessary to clean and prepare the data for better model performance.

REGRESSION ALGORITHMS IMPLEMENTATION

1. LINEAR REGRESSION ALGORITHM

In [87]:
```
from sklearn.linear_model import LinearRegression

lr_model = LinearRegression()

lr_model.fit(X_train, Y_train)

y_pred = lr_model.predict(X_test)
y_pred
```

Out[87]:
```
array([0.71912284, 1.76401657, 2.70965883, ..., 4.46877017, 1.18751119,
       2.00940251])
```

Linear Regression works by finding the best-fit line that minimizes the difference between the actual and predicted values. It assumes a linear relationship between the target variable (house prices) and input features (such as average income, house age, etc.). This model is suitable for the California Housing dataset because factors like income and location likely have a linear influence on house prices, making it a good fit for predicting the target variable.

2. DECISION TREE REGRESSOR ALGORITHM

In [99]:

```
from sklearn.tree import DecisionTreeRegressor

dt_model = DecisionTreeRegressor(random_state=42)

dt_model.fit(X_train, Y_train)

y_pred_dt = dt_model.predict(X_test)
y_pred_dt
```

Out[99]:

```
array([0.414   , 1.203   , 5.00001, ..., 5.00001, 0.66    , 2.172   ])
```

The Decision Tree Regressor works by recursively splitting the data based on feature values to minimize the variance within each subset. It does not assume a linear relationship between the target variable (house prices) and the input features (such as average income, house age, etc.). This model is suitable for the California Housing dataset because it can capture non-linear relationships and complex interactions between features, like how income and location might jointly influence house prices in ways that a linear model cannot.

3. RANDOM FOREST REGRESSOR ALGORITHM

In [118…

```
from sklearn.ensemble import RandomForestRegressor

rf_model = RandomForestRegressor(n_estimators=100, random_state=42)

rf_model.fit(X_train, Y_train)

y_pred_rf = rf_model.predict(X_test)
y_pred_rf
```

Out[118…

```
array([0.5095   , 0.74161  , 4.9232571, ..., 4.7582187, 0.71443  ,
       1.65772  ])
```

The Random Forest Regressor is an ensemble learning method that builds multiple decision trees and combines their predictions to improve accuracy and reduce overfitting. It handles complex, non-linear relationships between features and target variables effectively. This makes it suitable for the California Housing dataset, as it can capture intricate interactions between factors like income, house age, and location while providing robust predictions and feature importance insights.

4. GRADIENT BOOSTING REGRESSOR ALGORITHM

In [126…

```
from sklearn.ensemble import GradientBoostingRegressor

gb_model = GradientBoostingRegressor(n_estimators=100, random_state=42)
```

```
gb_model.fit(X_train, Y_train)

y_pred_gb = gb_model.predict(X_test)
y_pred_gb
```

Out[126…

```
array([0.50518761, 1.09334601, 4.24570956, ..., 4.68181295, 0.85329537,
       1.96275219])
```

The Gradient Boosting Regressor is an ensemble learning technique that builds models sequentially, with each model correcting the errors of the previous one. It minimizes residual errors using gradient descent and combines predictions from multiple weak learners to improve accuracy. This method is well-suited for the California Housing dataset as it can effectively capture complex, non-linear relationships between features like income, house age, and location, making it highly effective for predicting house prices.

5. SUPPORT VECTOR REGRESSOR ALGORITHM

In [134…

```
from sklearn.svm import SVR

svr_model = SVR(kernel='rbf')

svr_model.fit(X_train, Y_train)

y_pred_svr = svr_model.predict(X_test)
y_pred_svr
```

Out[134…

```
array([0.52166189, 1.56843583, 3.58873947, ..., 4.80511008, 0.70878931,
       1.73587521])
```

The Support Vector Regressor (SVR) is a regression model that finds a function which fits the data within a specified margin of error, focusing on minimizing the error for key data points called support vectors. It can handle both linear and non-linear relationships by applying kernel functions, such as the Radial Basis Function (RBF). SVR is suitable for the California Housing dataset because it can capture complex, non-linear relationships between features like income, house age, and location, while also being robust to outliers and effective in high-dimensional spaces.

MODEL EVALUATION

1. LINEAR REGRESSION MODEL EVALUATION

In [141…

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(Y_test, y_pred)

# Calculate Mean Absolute Error (MAE)
mae = mean_absolute_error(Y_test, y_pred)

# Calculate R-squared (R²)
r2 = r2_score(Y_test, y_pred)

# Print the evaluation metrics
print(f"Linear Regression Model Evaluation:")
print(f"Mean Squared Error (MSE): {mse}")
```

```
print(f"Mean Absolute Error (MAE): {mae}")
print(f"R-squared (R²): {r2}")
```

```
Linear Regression Model Evaluation:
Mean Squared Error (MSE): 0.5558915986952442
Mean Absolute Error (MAE): 0.5332001304956566
R-squared (R²): 0.575787706032451
```

2. DECISION TREE REGRESSOR MODEL EVALUATION

In [143…

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Calculate Mean Squared Error (MSE)
mse_dt = mean_squared_error(Y_test, y_pred_dt)

# Calculate Mean Absolute Error (MAE)
mae_dt = mean_absolute_error(Y_test, y_pred_dt)

# Calculate R-squared (R²)
r2_dt = r2_score(Y_test, y_pred_dt)

# Print the evaluation metrics
print(f"Decision Tree Regressor Evaluation:")
print(f"Mean Squared Error (MSE): {mse_dt}")
print(f"Mean Absolute Error (MAE): {mae_dt}")
print(f"R-squared (R²): {r2_dt}")
```

```
Decision Tree Regressor Evaluation:
Mean Squared Error (MSE): 0.4942716777366763
Mean Absolute Error (MAE): 0.4537843265503876
R-squared (R²): 0.6228111330554302
```

3. RANDOM FOREST REGRESSOR MODEL EVALUATION

In [145…

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Calculate Mean Squared Error (MSE)
mse_rf = mean_squared_error(Y_test, y_pred_rf)

# Calculate Mean Absolute Error (MAE)
```

```python
mae_rf = mean_absolute_error(Y_test, y_pred_rf)

# Calculate R-squared (R²)
r2_rf = r2_score(Y_test, y_pred_rf)

# Print the evaluation metrics
print(f"Random Forest Regressor Evaluation:")
print(f"Mean Squared Error (MSE): {mse_rf}")
print(f"Mean Absolute Error (MAE): {mae_rf}")
print(f"R-squared (R²): {r2_rf}")
```

```
Random Forest Regressor Evaluation:
Mean Squared Error (MSE): 0.25549776668540763
Mean Absolute Error (MAE): 0.32761306601259704
R-squared (R²): 0.805024407701793
```

4. GRADIENT BOOSTING REGRESSOR MODEL EVALUATION

In [148…

```python
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Calculate Mean Squared Error (MSE)
mse_gb = mean_squared_error(Y_test, y_pred_gb)

# Calculate Mean Absolute Error (MAE)
mae_gb = mean_absolute_error(Y_test, y_pred_gb)

# Calculate R-squared (R²)
r2_gb = r2_score(Y_test, y_pred_gb)

# Print the evaluation metrics
print(f"Gradient Boosting Regressor Evaluation:")
print(f"Mean Squared Error (MSE): {mse_gb}")
print(f"Mean Absolute Error (MAE): {mae_gb}")
print(f"R-squared (R²): {r2_gb}")
```

```
Gradient Boosting Regressor Evaluation:
Mean Squared Error (MSE): 0.29399901242474274
Mean Absolute Error (MAE): 0.37165044848436773
R-squared (R²): 0.7756433164710084
```

5. SUPPORT VECTOR REGRESSOR MODEL EVALUATION

In [150…

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Calculate Mean Squared Error (MSE)
mse_svr = mean_squared_error(Y_test, y_pred_svr)

# Calculate Mean Absolute Error (MAE)
mae_svr = mean_absolute_error(Y_test, y_pred_svr)

# Calculate R-squared (R²)
r2_svr = r2_score(Y_test, y_pred_svr)

# Print the evaluation metrics
print(f"Support Vector Regressor Evaluation:")
print(f"Mean Squared Error (MSE): {mse_svr}")
print(f"Mean Absolute Error (MAE): {mae_svr}")
print(f"R-squared (R²): {r2_svr}")
```

```
Support Vector Regressor Evaluation:
Mean Squared Error (MSE): 0.3551984619989429
Mean Absolute Error (MAE): 0.397763096343787
R-squared (R²): 0.7289407597956454
```

Summary of Best and Worst-Performing Models

Best-Performing Model:

Random Forest Regressor is the best-performing model with the lowest MSE (0.2555), lowest MAE (0.3276), and the highest R^2 (0.8050). This indicates that it has the highest predictive accuracy and is able to explain the most variance in the target variable (house prices). It handles the complex, non-linear relationships in the dataset effectively.

Worst-Performing Model:

Linear Regression performs the worst among the models tested, with the highest MSE (0.5559) and MAE (0.5332). Its R^2 score of 0.5758 means it only explains 57.5% of the variation in the data, which is relatively low compared to the tree-based models. Linear Regression assumes a straight-line relationship between the features and the target, but this may not be enough for this dataset, as it likely has more complex, non-linear patterns.

CONCLUSION

The Random Forest Regressor is the best model for the California Housing dataset because it performs well and can handle complex patterns in the data. On the other hand, Linear Regression is the least effective model, likely because it assumes a simple linear relationship, which doesn't capture the complexity of the data.

In [ ]: