[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Loading and Preprocessing (2 marks):

Load the California Housing dataset using the fetch_california_housing function from sklearn. Convert the dataset into a pandas DataFrame for easier handling. Handle missing values (if any) and perform necessary feature scaling (e.g., standardization). Explain the preprocessing steps you performed and justify why they are necessary for this dataset.

[5]:
```
#Load the California Housing dataset using the fetch_california_housing function from sklearn.
from sklearn.datasets import fetch_california_housing
```

[7]:
```
data = fetch_california_housing()
data
```

[7]:
```
{'data': array([[   8.3252    ,   41.        ,    6.98412698, ...,    2.55555556,
          37.88      , -122.23      ],
        [   8.3014    ,   21.        ,    6.23813708, ...,    2.10984183,
          37.86      , -122.22      ],
        [   7.2574    ,   52.        ,    8.28813559, ...,    2.80225989,
          37.85      , -122.24      ],
        ...,
        [   1.7       ,   17.        ,    5.20554273, ...,    2.3256351 ,
          39.43      , -121.22      ],
        [   1.8672    ,   18.        ,    5.32951289, ...,    2.12320917,
          39.43      , -121.32      ],
        [   2.3886    ,   16.        ,    5.25471698, ...,    2.61698113,
          39.37      , -121.24      ]]),
 'target': array([4.526, 3.585, 3.521, ..., 0.923, 0.847, 0.894]),
 'frame': None,
 'target_names': ['MedHouseVal'],
 'feature_names': ['MedInc',
  'HouseAge',
  'AveRooms',
  'AveBedrms',
  'Population',
  'AveOccup',
  'Latitude',
  'Longitude'],
 'DESCR': '.. _california_housing_dataset:\n\nCalifornia Housing dataset\n--------------------------\n\n**Data Set Characteristics:**\n\n:Number of Instances: 20640\n\n:Number of Attributes: 8 numeric, predictive attributes and the target\n\n:Attribute Information:\n    - MedInc        median income in block group\n    - HouseAge      median house age in block group\n    - AveRooms      average number of rooms per household\n    - AveBedrms     average number of bedrooms per household\n    - Population    block group population\n    - AveOccup      average number of household members\n    - Latitude      block group latitude\n    - Longitude     block group longitude\n\n:Missing Attribute Values: None\n\nThis dataset was obtained from the StatLib repository.\nhttps://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html\n\nThe target variable is the median house value for California districts,\nexpressed in hundreds of thousands of dollars ($100,000).\n\nThis dataset was derived from the 1990 U.S. census, using one row per census\nblock group. A block group is the smallest geographical unit for which the U.S.\nCensus Bureau publishes sample data (a block group typically has a population\nof 600 to 3,000 people).\n\nA household is a group of people residing within a home. Since the average\nnumber of rooms and bedrooms in this dataset are provided per household, these\ncolumns may take surprisingly large values for block groups with few households\nand many empty houses, such as vacation resorts.\n\nIt can be downloaded/loaded using the\n:func:`sklearn.datasets.fetch_california_housing` function.\n\n.. topic:: References\n\n    - Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions,\n      Statistics and Probability Letters, 33 (1997) 291-297\n'}
```

[9]:
```
df = pd.DataFrame(data.data, columns=data.feature_names)
df
```

[9]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25
...	...	...	...	...	...	...	...	...
20635	1.5603	25.0	5.045455	1.133333	845.0	2.560606	39.48	-121.09
20636	2.5568	18.0	6.114035	1.315789	356.0	3.122807	39.49	-121.21
20637	1.7000	17.0	5.205543	1.120092	1007.0	2.325635	39.43	-121.22
20638	1.8672	18.0	5.329513	1.171920	741.0	2.123209	39.43	-121.32
20639	2.3886	16.0	5.254717	1.162264	1387.0	2.616981	39.37	-121.24

20640 rows × 8 columns

[11]:
```
# to get the basic information about the dataset(datatype,no:of columns,rows etc)
df.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
```

[13]:
```
# to find the duplicates in the data set.
df.duplicated().sum()
```

[13]:
```
0
```

There is no duplicated value in this dataset.

[16]:
```
# to check the misssing values in the dataset.
df.isnull().sum()
```

[16]:
```
MedInc        0
HouseAge      0
AveRooms      0
AveBedrms     0
Population    0
AveOccup      0
Latitude      0
Longitude     0
dtype: int64
```

The data do not have any missing values.

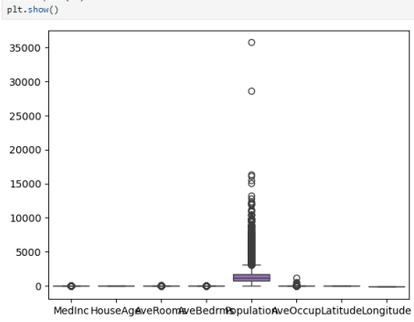
[19]:
```
#to analyse the statistical measures of the dataset.
df.describe()
```

[19]:

	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

we can not see major difference between "mean" and "median" of different columns except 'Population' and 'AveOccup'. All others have minor difference between mean and median.The difference may be the indication of 'outliers'.

[22]:
```
# to check the presence of outliers.
sns.boxplot(df)
plt.show()
```



[24]:
```
# to check the skewness of the data
skewness=df.skew()
skewness
```

[24]:
```
MedInc         1.646657
HouseAge       0.060331
AveRooms      20.697869
AveBedrms     31.316956
Population     4.935858
AveOccup      97.639561
Latitude       0.465953
Longitude     -0.297801
dtype: float64
```

The data is possitively skewed in most of the columns.

[27]:
```
# to find the outliers using IQR method.
numericals=df.select_dtypes("number")
for column in numericals:
    Q1=df[column].quantile(0.25)
    Q3=df[column].quantile(0.75)
    IQR=Q3-Q1
    lower_whisker=Q1-1.5*IQR
    upper_whisker=Q3+1.5*IQR
    outliers=df[(df[column]<lower_whisker) | (df[column]>upper_whisker)]

print(outliers)
```

```
Empty DataFrame
Columns: [MedInc, HouseAge, AveRooms, AveBedrms, Population, AveOccup, Latitude, Longitude]
Index: []
```

The data do not contain any outliers. that is why we got an empty dataframe for outliers.

[30]:
```
from sklearn.model_selection import train_test_split
```

[34]:
```
data.target # To get the 'target' column
```

[34]:
```
array([4.526, 3.585, 3.521, ..., 0.923, 0.847, 0.894])
```

[36]:
```
df["Target"]=data.target
df
```

[36]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	Target
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	3.413
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25	3.422
...	...	...	...	...	...	...	...	...	...
20635	1.5603	25.0	5.045455	1.133333	845.0	2.560606	39.48	-121.09	0.781
20636	2.5568	18.0	6.114035	1.315789	356.0	3.122807	39.49	-121.21	0.771
20637	1.7000	17.0	5.205543	1.120092	1007.0	2.325635	39.43	-121.22	0.923
20638	1.8672	18.0	5.329513	1.171920	741.0	2.123209	39.43	-121.32	0.847
20639	2.3886	16.0	5.254717	1.162264	1387.0	2.616981	39.37	-121.24	0.894

20640 rows × 9 columns

[38]:
```
df.isnull().sum()
```

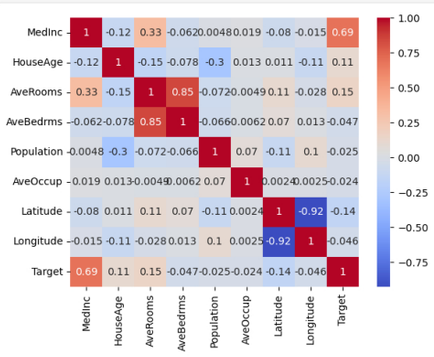
[38]:
```
MedInc        0
HouseAge      0
AveRooms      0
AveBedrms     0
Population    0
AveOccup      0
Latitude      0
Longitude     0
Target        0
dtype: int64
```

[40]:
```
#to find the correlation between the target and other features
corr=df.corr()
corr
```

[40]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	Target
MedInc	1.000000	-0.119034	0.326895	-0.062040	0.004834	0.018766	-0.079809	-0.015176	0.688075
HouseAge	-0.119034	1.000000	-0.153277	-0.077747	-0.296244	0.013191	0.011173	-0.108197	0.105623
AveRooms	0.326895	-0.153277	1.000000	0.847621	-0.072213	-0.004852	0.106389	-0.027540	0.151948
AveBedrms	-0.062040	-0.077747	0.847621	1.000000	-0.066197	-0.006181	0.069721	0.013344	-0.046701
Population	0.004834	-0.296244	-0.072213	-0.066197	1.000000	0.069863	-0.108785	0.099773	-0.024650
AveOccup	0.018766	0.013191	-0.004852	-0.006181	0.069863	1.000000	0.002366	0.002476	-0.023737
Latitude	-0.079809	0.011173	0.106389	0.069721	-0.108785	0.002366	1.000000	-0.924664	-0.144160
Longitude	-0.015176	-0.108197	-0.027540	0.013344	0.099773	0.002476	-0.924664	1.000000	-0.045967
Target	0.688075	0.105623	0.151948	-0.046701	-0.024650	-0.023737	-0.144160	-0.045967	1.000000

[42]:
```
# drawing a heatmap based on correlation
sns.heatmap(corr,annot=True,cmap="coolwarm")
plt.show()
```



Target has a strong possitive correlation with 'MedInc' compared to other features.

[45]:
```
X=df.drop("Target",axis=1) #features
X
```

[45]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25
...	...	...	...	...	...	...	...	...
20635	1.5603	25.0	5.045455	1.133333	845.0	2.560606	39.48	-121.09
20636	2.5568	18.0	6.114035	1.315789	356.0	3.122807	39.49	-121.21
20637	1.7000	17.0	5.205543	1.120092	1007.0	2.325635	39.43	-121.22
20638	1.8672	18.0	5.329513	1.171920	741.0	2.123209	39.43	-121.32
20639	2.3886	16.0	5.254717	1.162264	1387.0	2.616981	39.37	-121.24

20640 rows × 8 columns

[47]:
```
y=df["Target"]
y
```

[47]:
```
0        4.526
1        3.585
2        3.521
3        3.413
4        3.422
         ...
20635    0.781
20636    0.771
20637    0.923
20638    0.847
20639    0.894
Name: Target, Length: 20640, dtype: float64
```

[49]:
```
# feature scaling using standard scaler
from sklearn.preprocessing import StandardScaler
std_scaler=StandardScaler()
```

[51]:
```
X_scaled=std_scaler.fit_transform(X)
X_scaled
```

[51]:
```
array([[ 2.34476576,  0.98214266,  0.62855945, ..., -0.04959654,
         1.05254828, -1.32783522],
       [ 2.33223796, -0.60701891,  0.32704136, ..., -0.09251223,
```

```
  1.04318455, -1.32284391],
[ 1.7826994 ,  1.85618152,  1.15562047, ..., -0.02584253,
  1.03850269, -1.33282653],
...,
[-1.14259331, -0.92485123, -0.09031802, ..., -0.0717345 ,
  1.77823747, -0.8237132 ],
[-1.05458292, -0.84539315, -0.04021111, ..., -0.09122515,
  1.77823747, -0.87362627],
[-0.78012947, -1.00430931, -0.07044252, ..., -0.04368215,
  1.75014627, -0.83369581]])
```

[53]:
```
X_train,X_test,y_train,y_test=train_test_split(X_scaled,y)
```

[55]:
```
X_train.shape
```

[55]:
```
(15480, 8)
```

[57]:
```
X_test.shape
```

[57]:
```
(5160, 8)
```

[59]:
```
y_train.shape
```

[59]:
```
(15480,)
```

[61]:
```
y_test.shape
```

[61]:
```
(5160,)
```

Regression Algorithm Implementation (5 marks):

Implement the following regression algorithms:

1.Linear Regression 2.Decision Tree Regressor 3.Random Forest Regressor 4.Gradient Boosting Regressor 5.Support Vector Regressor (SVR) For each algorithm:

*Provide a brief explanation of how it works. *Explain why it might be suitable for this dataset.

[66]:
```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor,GradientBoostingRegressor
from sklearn.svm import SVR
```

[68]:
```
#.Linear Regression
lnreg=LinearRegression()
lnreg.fit(X_train,y_train)
```

[68]:
```
LinearRegression()
```

How it Works:Linear Regression models the relationship between the independent variables (features) and the dependent variable (target) using a straight-line equation.

Suitability :Provides a simple and interpretable model.Useful if the relationship between features and housing prices is linear.Computationally efficient for large datasets.However, it may underperform if relationships are non-linear or interactions between features are significant.

[72]:
```
lnreg_y_pred=lnreg.predict(X_test)
lnreg_y_pred
```

[72]:
```
array([2.13270208, 1.55677004, 1.82974354, ..., 2.3542956 , 4.02391977,
       3.38337452])
```

[74]:
```
# Decision Tree Regressor
dtreg=DecisionTreeRegressor()
dtreg.fit(X_train,y_train)
```

[74]:
```
DecisionTreeRegressor()
```

How it Work :A Decision Tree is a supervised learning algorithm used for classification and regression tasks.The structure resembles a tree, with nodes representing decisions, branches representing choices, and leaves representing outcomes or predictions.

Suitability:Handles non-linearity well, unlike Linear Regression. Can automatically capture interactions between features. Prone to overfitting if not pruned or regularized.

[78]:
```
dtreg_y_pred=dtreg.predict(X_test)
dtreg_y_pred
```

[78]:
```
array([2.063  , 0.675  , 1.22   , ..., 3.181  , 5.00001, 4.093  ])
```

[80]:
```
# Random Forest Regressor
rfreg=RandomForestRegressor()
rfreg.fit(X_train,y_train)
```

[80]:
```
RandomForestRegressor()
```

How it Works: Random Forest is an ensemble learning method that builds multiple Decision Trees and averages their predictions to improve accuracy and reduce overfitting. Each tree is built on a random subset of data and features.

Suitability:Reduces overfitting compared to a single Decision Tree. Handles non-linear relationships effectively.

[84]:
```
rfreg_y_pred=rfreg.predict(X_test)
rfreg_y_pred
```

[84]:
```
array([2.17264  , 1.1128   , 1.26062  , ..., 2.38108  , 4.8604985,
       3.7244207])
```

[86]:
```
# Gradient Boosting Regressor
gbreg=GradientBoostingRegressor()
gbreg.fit(X_train,y_train)
```

[86]:
```
GradientBoostingRegressor()
```

How it Works: Gradient Boosting is another ensemble method that builds Decision Trees sequentially, where each new tree corrects the errors of the previous one. It minimizes the loss function by iteratively improving predictions.

Suitability:More accurate than Random Forest in many cases. Works well with complex and structured datasets.

[90]:
```
gbreg_y_pred=gbreg.predict(X_test)
gbreg_y_pred
```

[90]:
```
array([2.41796587, 1.39275421, 1.42734544, ..., 2.09793304, 4.99754178,
       3.92351385])
```

[92]:
```
# Support Vector Regressor (SVR)
svreg=SVR()
svreg.fit(X_train,y_train)
```

[92]:
```
SVR()
```

How it Works: SVC uses a set of labeled training examples to find a decision boundary that separates the data points into different classes. The decision boundary is represented as a linear function, and the goal is to find the boundary that maximizes the separation between the classes.

Suitability:SVC is commonly used in image recognition, text classification, and medical diagnosis. It's effective for complex or non-linear datasets.

[99]:
```
svreg_y_pred=svreg.predict(X_test)
svreg_y_pred
```

[99]:
```
array([2.07366577, 1.42998729, 1.3034285 , ..., 2.32657629, 5.08941866,
       3.60146616])
```

[101]:
```
y_test
```

[101]:
```
449      2.12000
16433    2.02800
20583    1.37500
2438     0.88000
18955    1.29800
          ...
15261    3.69800
16501    1.69400
20223    1.86800
5349     5.00001
17001    4.04000
Name: Target, Length: 5160, dtype: float64
```

Model Evaluation and Comparison (2 marks):¶

Evaluate the performance of each algorithm using the following metrics:

Mean Squared Error (MSE,) Mean Absolute Error (MA,E) R-squared Score (R^2) Compare the results of all models and identi The best-performing algorithm with justification.

The worst-performing algorithm with reasoning.

[105]:
```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
```

[107]:
```
# Mean Squared Error (MSE)
lnreg_MSE=mean_squared_error(y_test,lnreg_y_pred)
dtreg_MSE=mean_squared_error(y_test,dtreg_y_pred)
rfreg_MSE=mean_squared_error(y_test,rfreg_y_pred)
gbreg_MSE=mean_squared_error(y_test,gbreg_y_pred)
svreg_MSE=mean_squared_error(y_test,svreg_y_pred)

print("Mean Squared Error-Linear Regression:",lnreg_MSE)
print("Mean Squared Error-Decision Tree Regressor:",dtreg_MSE)
print("Mean Squared Error-Random Forest Regressor:",rfreg_MSE)
print("Mean Squared Error-Gradient Boosting Regressor:",gbreg_MSE)
print("Mean Squared Error-Support Vector Regressor:",svreg_MSE)
```

```
Mean Squared Error-Linear Regression: 0.525175438026496
Mean Squared Error-Decision Tree Regressor: 0.5100158514343217
Mean Squared Error-Random Forest Regressor: 0.2483110796844896
Mean Squared Error-Gradient Boosting Regressor: 0.2867795153558999
Mean Squared Error-Support Vector Regressor: 0.348102536057333
```

[109]:
```
# Mean Absolute Error (MAE)
lnreg_MAE=mean_absolute_error(y_test,lnreg_y_pred)
dtreg_MAE=mean_absolute_error(y_test,dtreg_y_pred)
rfreg_MAE=mean_absolute_error(y_test,rfreg_y_pred)
gbreg_MAE=mean_absolute_error(y_test,gbreg_y_pred)
svreg_MAE=mean_absolute_error(y_test,svreg_y_pred)

print("Mean Absolute Error-Linear Regression:",lnreg_MAE)
print("Mean Absolute Error-Decision Tree Regressor:",dtreg_MAE)
print("Mean Absolute Errorr-Random Forest Regressor:",rfreg_MAE)
print("Mean Absolute Error-Gradient Boosting Regressor:",gbreg_MAE)
print("Mean Absolute Error-Support Vector Regressor:",svreg_MAE)
```

```
Mean Absolute Error-Linear Regression: 0.5299249681701015
Mean Absolute Error-Decision Tree Regressor: 0.45934110658914723
Mean Absolute Errorr-Random Forest Regressor: 0.32561199616279085
Mean Absolute Error-Gradient Boosting Regressor: 0.3700616944057887
Mean Absolute Error-Support Vector Regressor: 0.3934159051040744
```

[111]:
```
#  R-squared Score ( R**2)
lnreg_r2_score=r2_score(y_test,lnreg_y_pred)
dtreg_r2_score=r2_score(y_test,dtreg_y_pred)
rfreg_r2_score=r2_score(y_test,rfreg_y_pred)
gbreg_r2_score=r2_score(y_test,gbreg_y_pred)
svreg_r2_score=r2_score(y_test,svreg_y_pred)

print("R-squared Score-Linear Regression:",lnreg_r2_score)
print("R-squared Score-Decision Tree Regressor:",dtreg_r2_score)
print("R-squared Score-Random Forest Regressor:",rfreg_r2_score)
print("R-squared Score-Gradient Boosting Regressor:",gbreg_r2_score)
print("R-squared Score-Support Vector Regressor:",svreg_r2_score)
```

```
R-squared Score-Linear Regression: 0.605262388530077
R-squared Score-Decision Tree Regressor: 0.6166567885133538
R-squared Score-Random Forest Regressor: 0.813361944601781
R-squared Score-Gradient Boosting Regressor: 0.7844479145188457
R-squared Score-Support Vector Regressor: 0.7383556928209546
```

[113]:
```
# Compare the results of all models and identify:
predictions={
    "Linear Regression": lnreg_y_pred,
    "Decision Tree Regressor":dtreg_y_pred,
    "Random Forest Regressor":rfreg_y_pred,
    "Gradient Boosting Regressor":gbreg_y_pred,
    "Support Vector Regresso":svreg_y_pred
}


# Initialize a dictionary to store metrics
results={
    "Model":[],
    "Mean Squared Error":[],
    "Mean Absolute Error":[],
    "R-squared Score":[]

}

# Compute metrics for each model
for model_name, y_pred in predictions.items():
    results["Model"].append(model_name)
    results["Mean Squared Error"].append(mean_squared_error(y_test,y_pred))
    results["Mean Absolute Error"].append(mean_absolute_error(y_test,y_pred))
    results["R-squared Score"].append(r2_score(y_test,y_pred))
results
```

[113]:
```
{'Model': ['Linear Regression',
  'Decision Tree Regressor',
  'Random Forest Regressor',
  'Gradient Boosting Regressor',
  'Support Vector Regresso'],
 'Mean Squared Error': [0.525175438026496,
  0.5100158514343217,
  0.2483110796844896,
  0.2867795153558999,
  0.348102536057333],
 'Mean Absolute Error': [0.5299249681701015,
  0.45934110658914723,
  0.32561199616279085,
  0.3700616944057887,
  0.3934159051040744],
 'R-squared Score': [0.605262388530077,
  0.6166567885133538,
  0.813361944601781,
  0.7844479145188457,
  0.7383556928209546]}
```

[115]:
```
#Convert results to a DataFrame
Results=pd.DataFrame(results)
Results
```

[115]:

	Model	Mean Squared Error	Mean Absolute Error	R-squared Score
0	Linear Regression	0.525175	0.529925	0.605262
1	Decision Tree Regressor	0.510016	0.459341	0.616657
2	Random Forest Regressor	0.248311	0.325612	0.813362
3	Gradient Boosting Regressor	0.286780	0.370062	0.784448
4	Support Vector Regresso	0.348103	0.393416	0.738356

[117]:
```
#sort by R-squared Score
Results=Results.sort_values(by="R-squared Score",ascending=False)
Results
```

[117]:

	Model	Mean Squared Error	Mean Absolute Error	R-squared Score
2	Random Forest Regressor	0.248311	0.325612	0.813362
3	Gradient Boosting Regressor	0.286780	0.370062	0.784448
4	Support Vector Regresso	0.348103	0.393416	0.738356
1	Decision Tree Regressor	0.510016	0.459341	0.616657
0	Linear Regression	0.525175	0.529925	0.605262

Mean Squared Error (MSE) – Lower values indicate better performance.

Mean Absolute Error (MAE) – Lower values indicate better accuracy.

R-squared Score (R^2) – Higher values indicate better model fit.

From the results table, we can conclude that "Random Forest Regressor" is the best-performing algorithm. Because Random Forest Regressor model has the highest R-squared Score. It's Mean Squared Error and Mean Absolute Error values are also the lowest compared to other models, which indicates better performance and accuracy.

The worst-performing algorithm is "Linear Regression model". Linear Regression has the lowest R-squared Score and highest Mean Squared Error and Mean Absolute Error values, which indicates poor performance and poor accuracy.

[ ]: