INFO-531 Data Warehousing and Analytics In The Cloud

Final Project Report

Team Members: Hari Dave & Naitik Shah

Project Option : OPTION -3 (ML Model)

Course Title: INFO 531/ISTA 431: Data Warehousing and Analytics in the Cloud
Term name and year: Fall 2024
Submission Week: Week 16
Assignment Instructor's Name: Nayem Rahman, PhD
Date of Submission: 12/18/2024

Table Of Contents:

1. Problem Statement

This project aims to develop a predictive model that can estimate the price of diamonds based on their physical and quality characteristics. By leveraging machine learning techniques, the goal is to provide a tool that assists stakeholders in making informed decisions regarding diamond valuation. This model will utilize a dataset comprising several attributes of diamonds, with the primary challenge being to discern the intricate patterns that dictate pricing dynamics.

The objectives of this project are:

1. To preprocess and clean the dataset to facilitate effective model training.
2. To identify and select significant features that impact diamond prices using exploratory data analysis and feature correlation.
3. To implement and tune machine learning models that can predict diamond prices with high accuracy, thereby serving as a decision support tool for pricing diamonds in the market.
4. To evaluate the model's performance using appropriate metrics and refine the approach based on the results to enhance prediction accuracy.

Through this project, we expect to derive a clear understanding of the key determinants of diamond prices and develop a reliable predictive model that can be used in real-world scenarios to estimate the prices of diamonds accurately.

2. Data Preparation

First, let’s learn about the data. What we have in our dataset?

Carat: Weight of the diamond measured in carats, a key indicator of size and price.

Cut: Quality of the diamond's cut affecting its brilliance, categorized from Fair to Ideal.

Color: Color rating of the diamond from D (colorless) to Z (light color), impacting its value.

Clarity: Measure of the diamond's clarity from Flawless to Included, indicating the presence of blemishes or inclusions.

Depth: Height of the diamond as a percentage of its average girdle diameter, influencing its optical properties.

Table: Width of the diamond's top facet as a percentage of its average diameter, important for its light reflection.

Price: Market price of the diamond in US dollars, serving as the prediction target in this project.

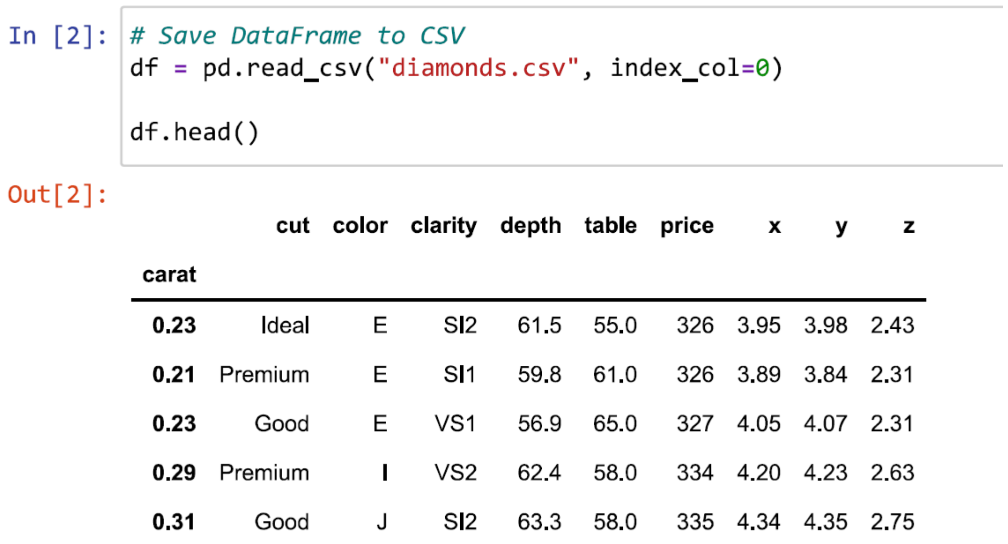
X (Length): Length of the diamond in millimeters, the longest measure of its dimensions.

Y (Width): Width of the diamond in millimeters, perpendicular to the length.

Z (Depth): Depth measurement of the diamond in millimeters, from the culet to the table.

These are the columns in our dataset.

Here we have imported the data and displayed the head of the table.

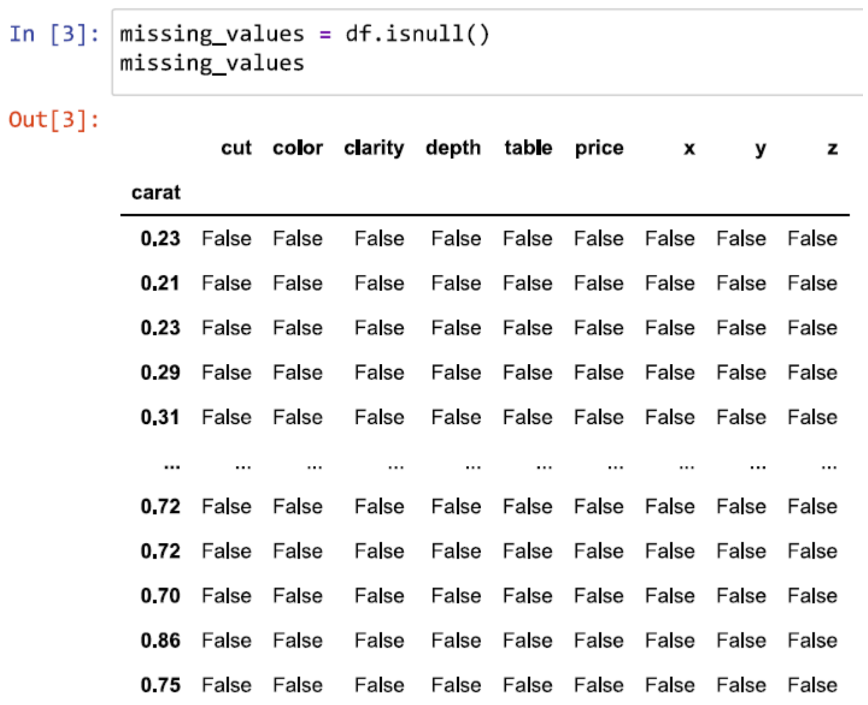


In [2]:
```
# Save DataFrame to CSV
df = pd.read_csv("diamonds.csv", index_col=0)

df.head()
```

Out[2]:

	cut	color	clarity	depth	table	price	x	y	z
carat									
0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

Head of the Table



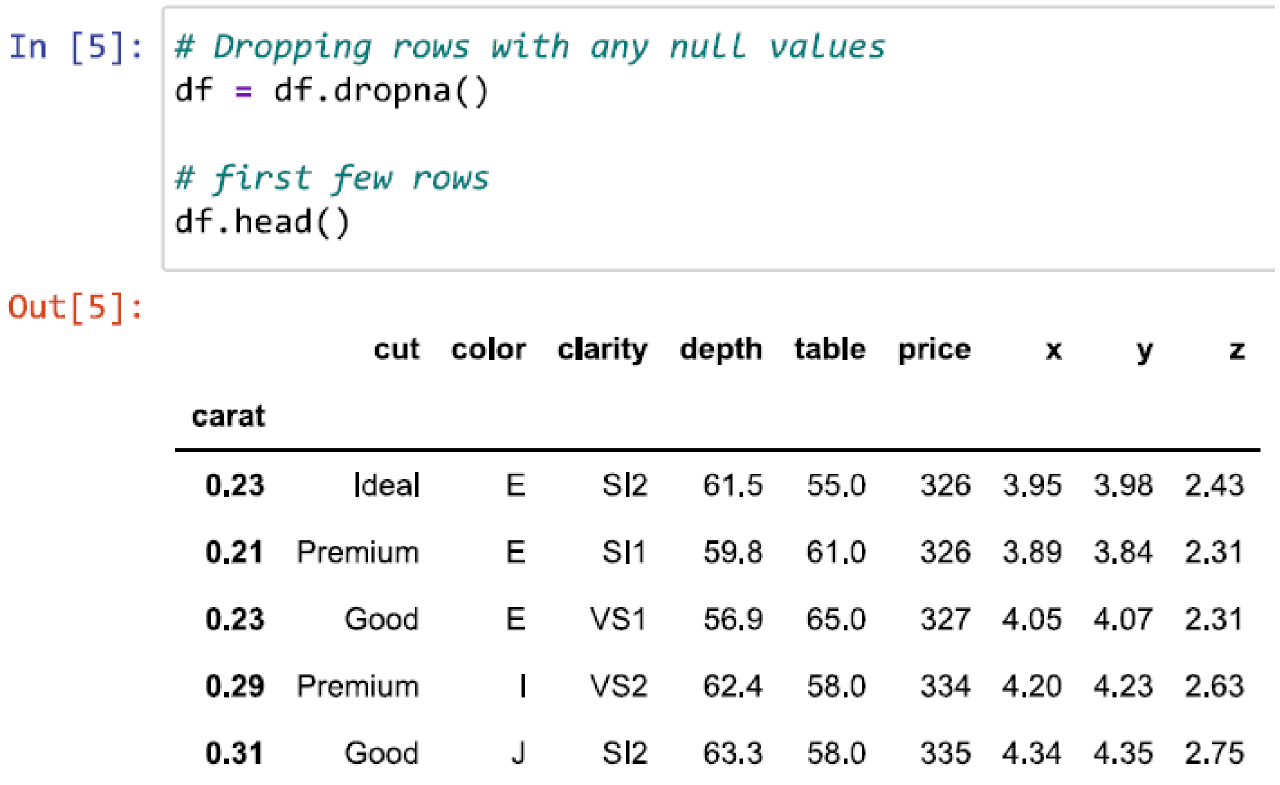
In [3]:
```
missing_values = df.isnull()
missing_values
```

Out[3]:

	cut	color	clarity	depth	table	price	x	y	z
carat									
0.23	False	False	False	False	False	False	False	False	False
0.21	False	False	False	False	False	False	False	False	False
0.23	False	False	False	False	False	False	False	False	False
0.29	False	False	False	False	False	False	False	False	False
0.31	False	False	False	False	False	False	False	False	False
...	...	...	...	...	...	...	...	...	...
0.72	False	False	False	False	False	False	False	False	False
0.72	False	False	False	False	False	False	False	False	False
0.70	False	False	False	False	False	False	False	False	False
0.86	False	False	False	False	False	False	False	False	False
0.75	False	False	False	False	False	False	False	False	False

The above code snippet checks for missing values across all columns of the dataset. The isnull() function was applied to each column, returning a DataFrame of the same shape as df, where each entry is either True if the corresponding entry in df is NaN, or False otherwise. The output clearly demonstrates that there are no missing values in any of the columns ('cut', 'color', 'clarity', 'depth', 'table', 'price', 'x', 'y', 'z'), as all cells in the resulting DataFrame are marked False.

3. Data Exploration:



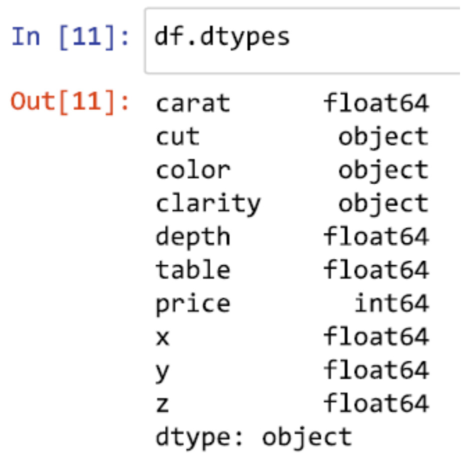
In [5]:
```python
# Dropping rows with any null values
df = df.dropna()

# first few rows
df.head()
```

Out[5]:

carat	cut	color	clarity	depth	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

Here we drop the null values and again check the table.

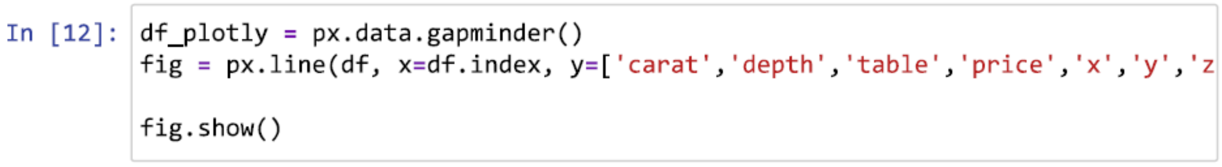
This comprehensive check ensures that the dataset is complete with no null entries, eliminating the need for further data cleaning related to missing values. This is significant as it allows us to proceed with confidence that the quality of our data will not adversely affect the predictive accuracy of our models.



In [11]:
```python
df.dtypes
```

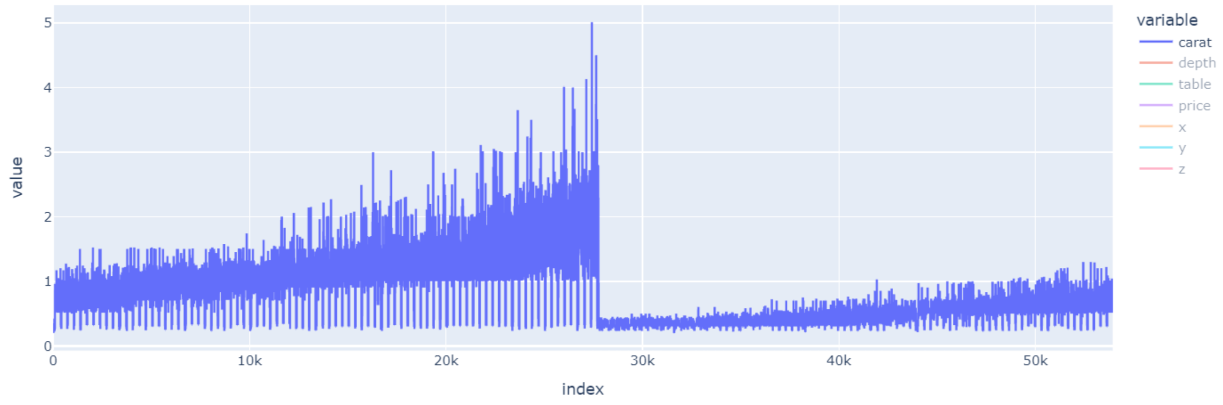
Out[11]:
```
carat       float64
cut          object
color        object
clarity      object
depth       float64
table       float64
price         int64
x           float64
y           float64
z           float64
dtype: object
```

Here, we can see the datatypes of the columns in the table.

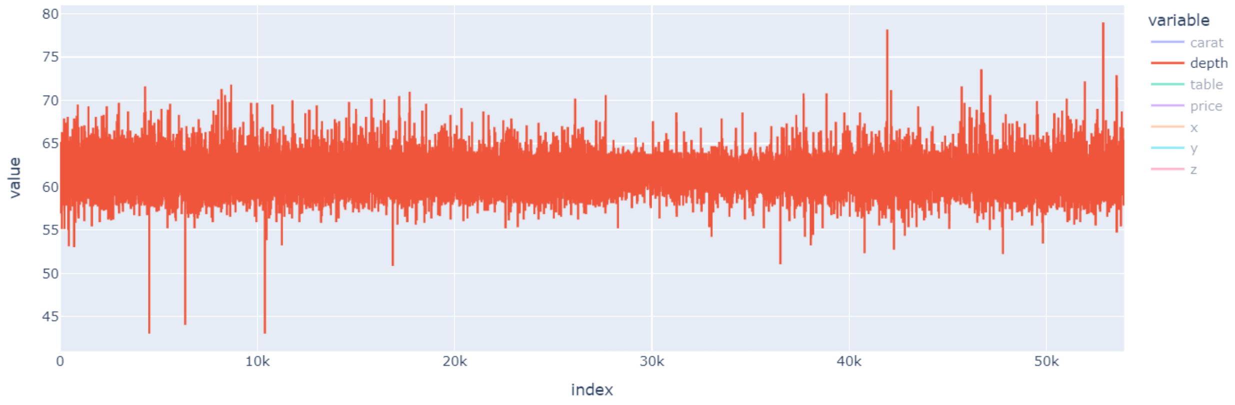


```
In [12]: df_plotly = px.data.gapminder()
         fig = px.line(df, x=df.index, y=['carat','depth','table','price','x','y','z

         fig.show()
```

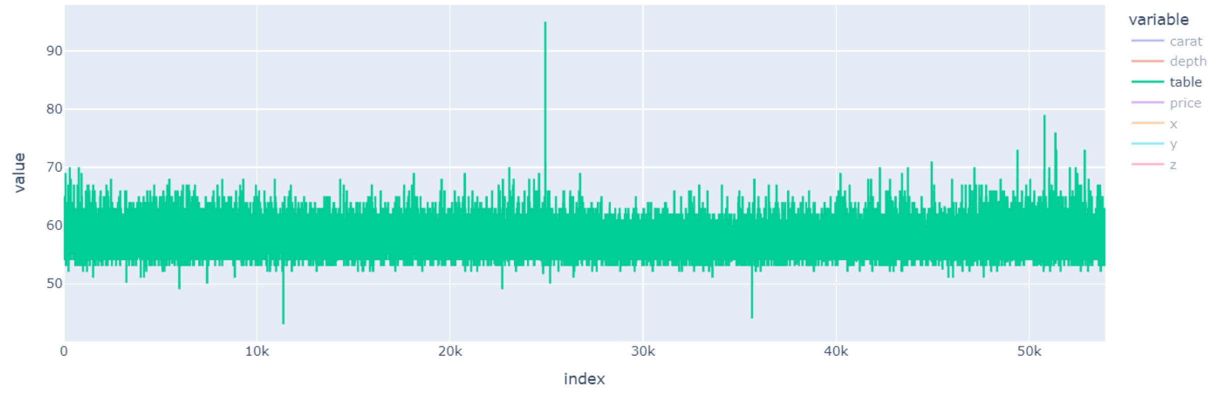
Plots for each variable are below shown:



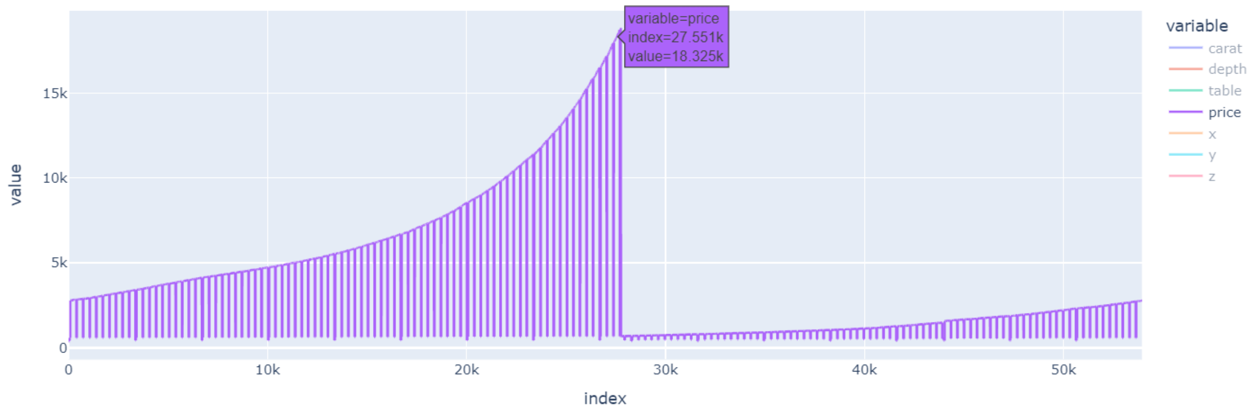
For Carat



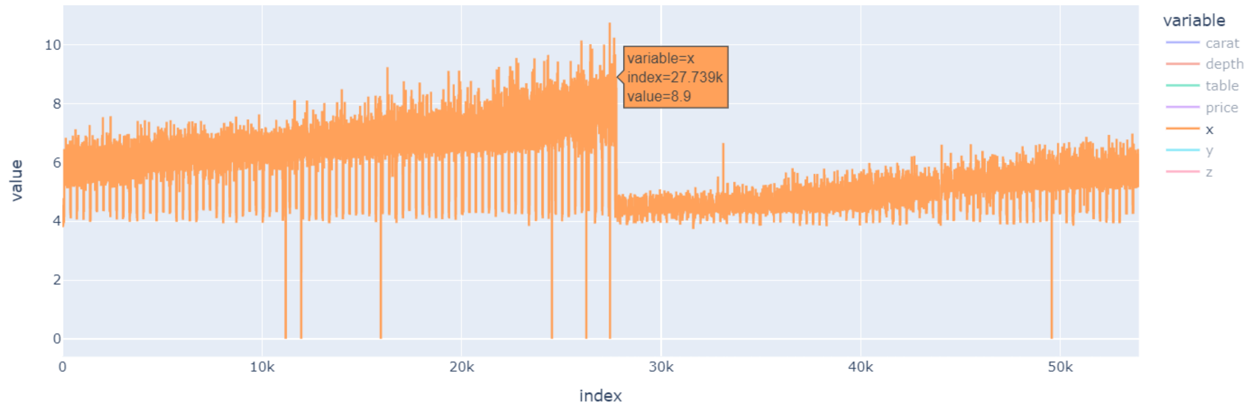
For Depth



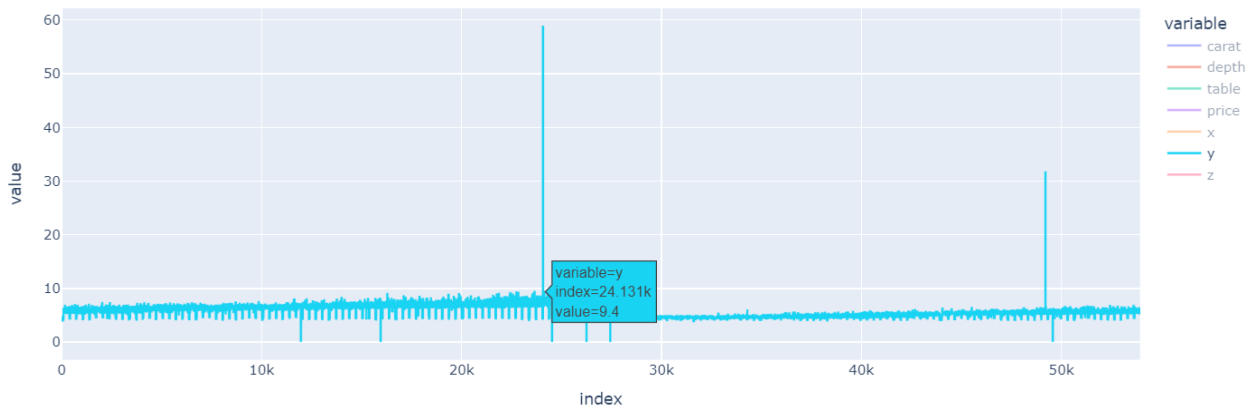
For Table



For Price

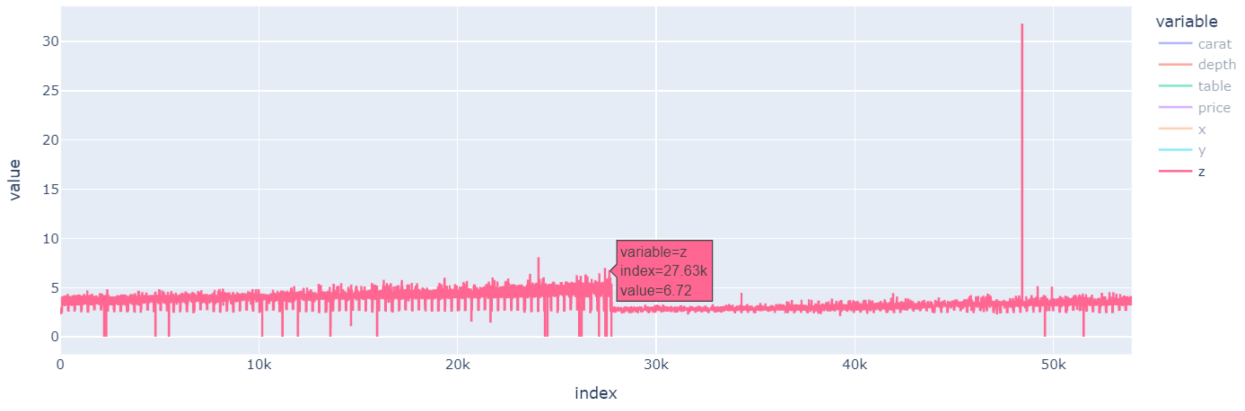


For X



variable
carat
depth
table
price
x
y
z
variable=y
index=24.131k
value=9.4
value
index
0
10
20
30
40
50
60
0
10k
20k
30k
40k
50k

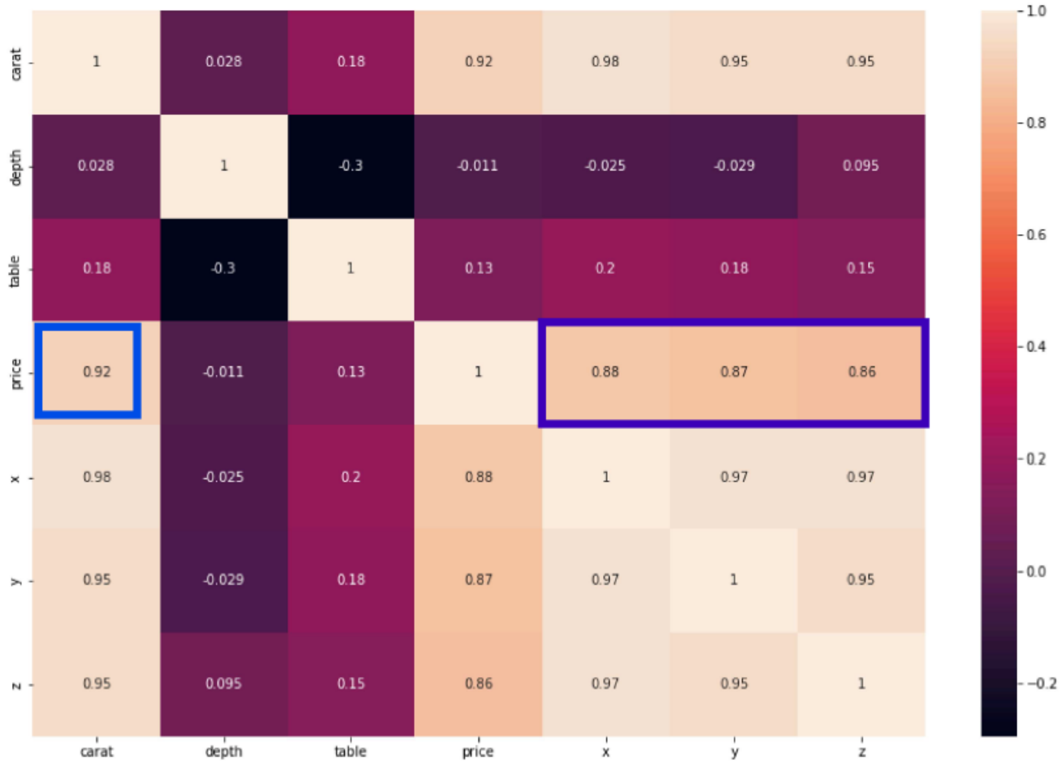
For Y



variable
carat
depth
table
price
x
y
z
variable=z
index=27.63k
value=6.72
value
index
0
5
10
15
20
25
30
0
10k
20k
30k
40k
50k

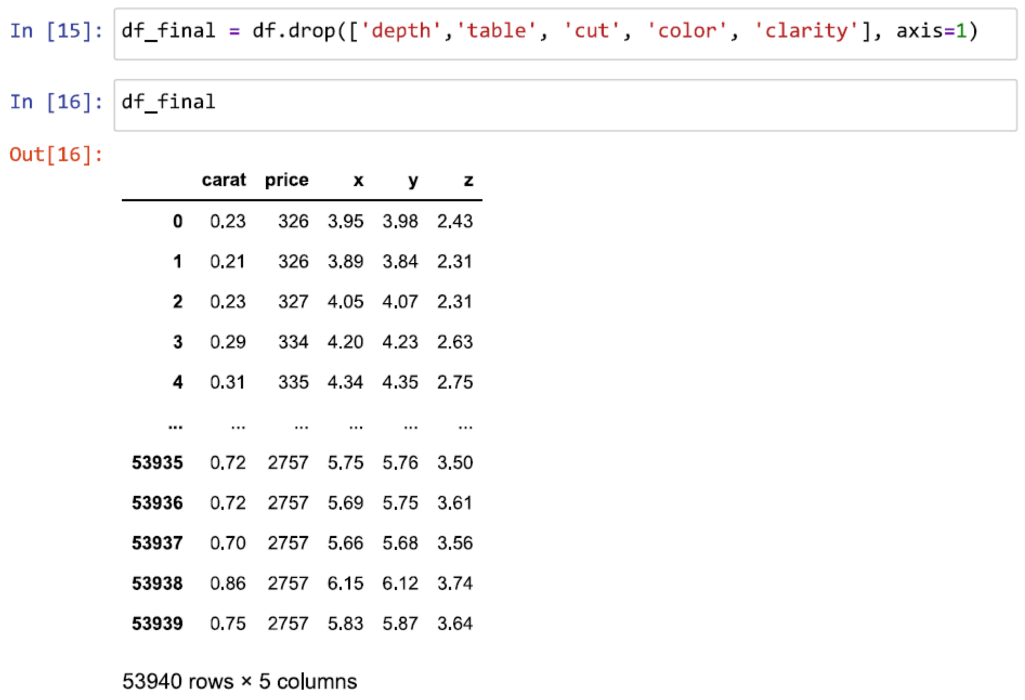
For Z

```
In [14]: plt.figure(figsize = (15, 10))
         sns.heatmap(df.corr(),annot=True)

Out[14]: <Axes: >
```



	carat	depth	table	price	x	y	z
carat	1	0.028	0.18	0.92	0.98	0.95	0.95
depth	0.028	1	-0.3	-0.011	-0.025	-0.029	0.095
table	0.18	-0.3	1	0.13	0.2	0.18	0.15
price	0.92	-0.011	0.13	1	0.88	0.87	0.86
x	0.98	-0.025	0.2	0.88	1	0.97	0.97
y	0.95	-0.029	0.18	0.87	0.97	1	0.95
z	0.95	0.095	0.15	0.86	0.97	0.95	1

- The variables carat, x (length), y (width), and z (depth) are strongly correlated to each other (coefficients above 0.95), which is intuitive since larger diamonds are generally heavier.
- The variable price is strongly correlated with carat (0.92) and moderately with x, y, and z (around 0.88 to 0.87). This indicates that as the size and weight of the diamond increase, its price tends to increase significantly.
- depth and table show very little correlation with price, suggesting they might be less impactful in predicting the price compared to carat or dimensions.

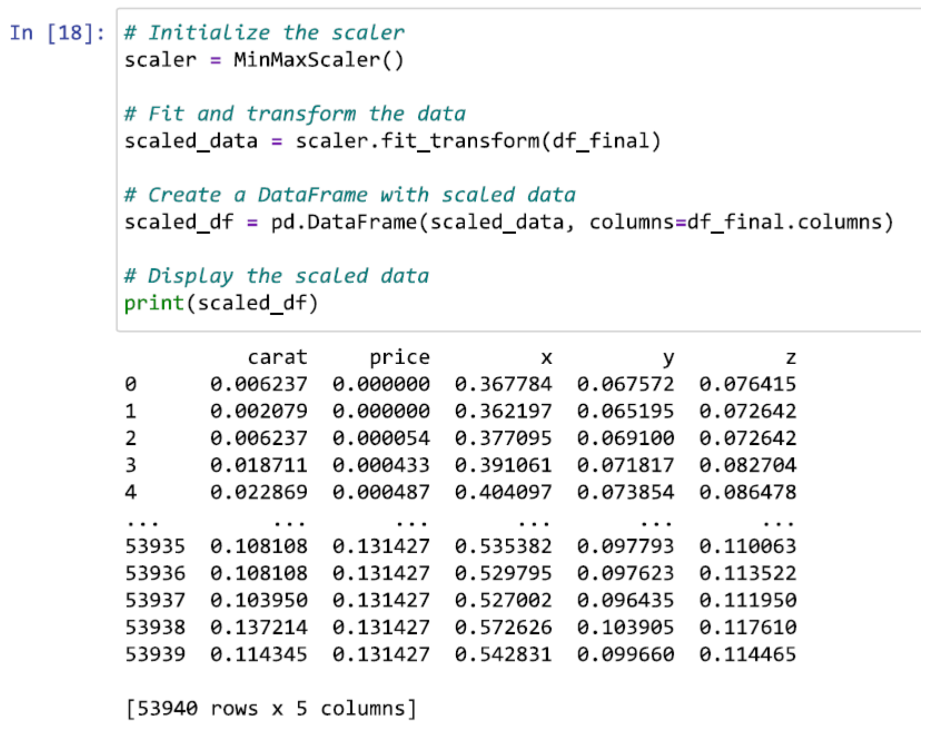


```
In [15]: df_final = df.drop(['depth','table', 'cut', 'color', 'clarity'], axis=1)

In [16]: df_final

Out[16]:
```

	carat	price	x	y	z
0	0.23	326	3.95	3.98	2.43
1	0.21	326	3.89	3.84	2.31
2	0.23	327	4.05	4.07	2.31
3	0.29	334	4.20	4.23	2.63
4	0.31	335	4.34	4.35	2.75
...	...	...	...	...	...
53935	0.72	2757	5.75	5.76	3.50
53936	0.72	2757	5.69	5.75	3.61
53937	0.70	2757	5.66	5.68	3.56
53938	0.86	2757	6.15	6.12	3.74
53939	0.75	2757	5.83	5.87	3.64

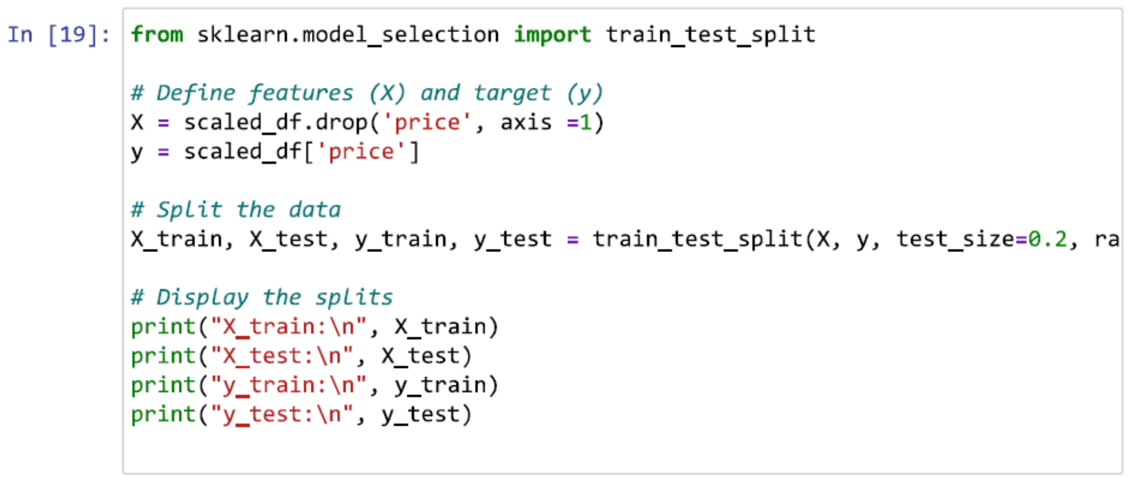
53940 rows × 5 columns

Here we have dropped the rest of the columns which are not co-related. And for further model we will be using these 4 parameters and 1 target (price).

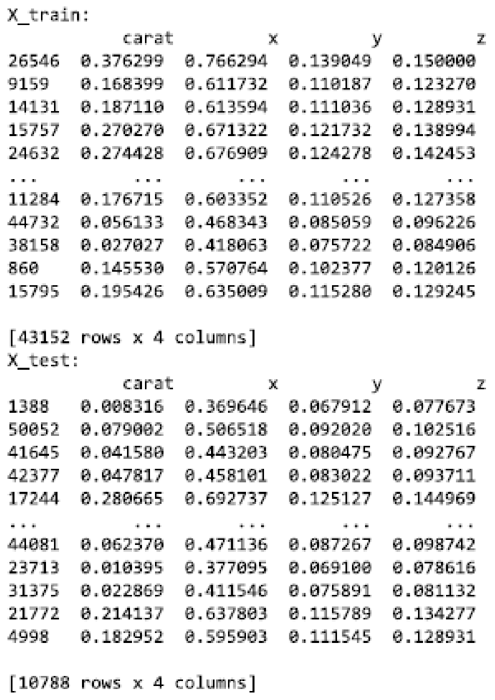


```
In [18]: # Initialize the scaler
         scaler = MinMaxScaler()

         # Fit and transform the data
         scaled_data = scaler.fit_transform(df_final)

         # Create a DataFrame with scaled data
         scaled_df = pd.DataFrame(scaled_data, columns=df_final.columns)

         # Display the scaled data
         print(scaled_df)

            carat     price         x         y         z
0        0.006237  0.000000  0.367784  0.067572  0.076415
1        0.002079  0.000000  0.362197  0.065195  0.072642
2        0.006237  0.000054  0.377095  0.069100  0.072642
3        0.018711  0.000433  0.391061  0.071817  0.082704
4        0.022869  0.000487  0.404097  0.073854  0.086478
...           ...       ...       ...       ...       ...
53935    0.108108  0.131427  0.535382  0.097793  0.110063
53936    0.108108  0.131427  0.529795  0.097623  0.113522
53937    0.103950  0.131427  0.527002  0.096435  0.111950
53938    0.137214  0.131427  0.572626  0.103905  0.117610
53939    0.114345  0.131427  0.542831  0.099660  0.114465

[53940 rows x 5 columns]
```

By using the Min-Max Scaler, data are transformed to fall within the 0 to 1 range, maintaining the distribution and relationships of the original data while normalizing the scale. This method is particularly beneficial when the data are known not to have outliers, as outliers can skew the maximum and minimum values significantly, leading to less effective scaling.

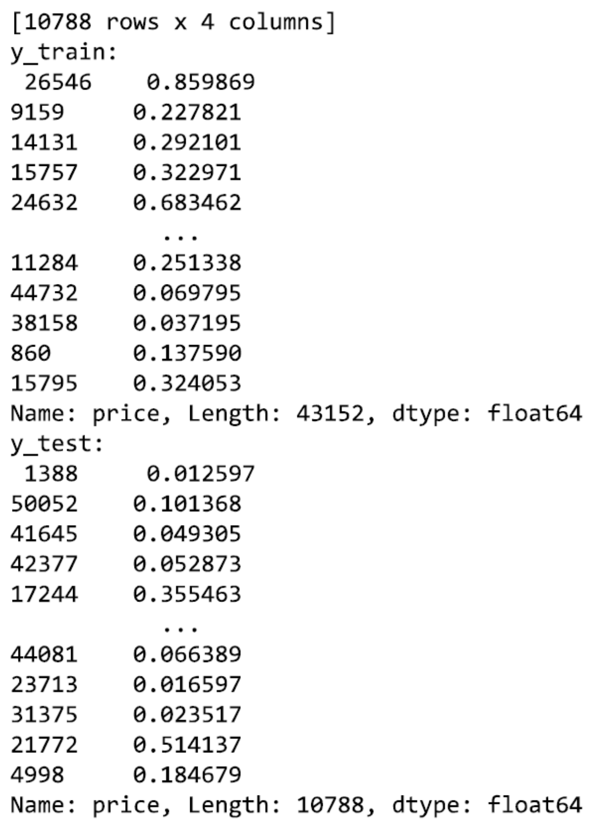


```
In [19]: from sklearn.model_selection import train_test_split

         # Define features (X) and target (y)
         X = scaled_df.drop('price', axis =1)
         y = scaled_df['price']

         # Split the data
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ra

         # Display the splits
         print("X_train:\n", X_train)
         print("X_test:\n", X_test)
         print("y_train:\n", y_train)
         print("y_test:\n", y_test)
```



```
X_train:
          carat         x         y         z
26546  0.376299  0.766294  0.139049  0.150000
9159   0.168399  0.611732  0.110187  0.123270
14131  0.187110  0.613594  0.111036  0.128931
15757  0.270270  0.671322  0.121732  0.138994
24632  0.274428  0.676909  0.124278  0.142453
...         ...       ...       ...       ...
11284  0.176715  0.603352  0.110526  0.127358
44732  0.056133  0.468343  0.085059  0.096226
38158  0.027027  0.418063  0.075722  0.084906
860    0.145530  0.570764  0.102377  0.120126
15795  0.195426  0.635009  0.115280  0.129245

[43152 rows x 4 columns]
X_test:
          carat         x         y         z
1388   0.008316  0.369646  0.067912  0.077673
50052  0.079002  0.506518  0.092020  0.102516
41645  0.041580  0.443203  0.080475  0.092767
42377  0.047817  0.458101  0.083022  0.093711
17244  0.280665  0.692737  0.125127  0.144969
...         ...       ...       ...       ...
44081  0.062370  0.471136  0.087267  0.098742
23713  0.010395  0.377095  0.069100  0.078616
31375  0.022869  0.411546  0.075891  0.081132
21772  0.214137  0.637803  0.115789  0.134277
4998   0.182952  0.595903  0.111545  0.128931

[10788 rows x 4 columns]
```

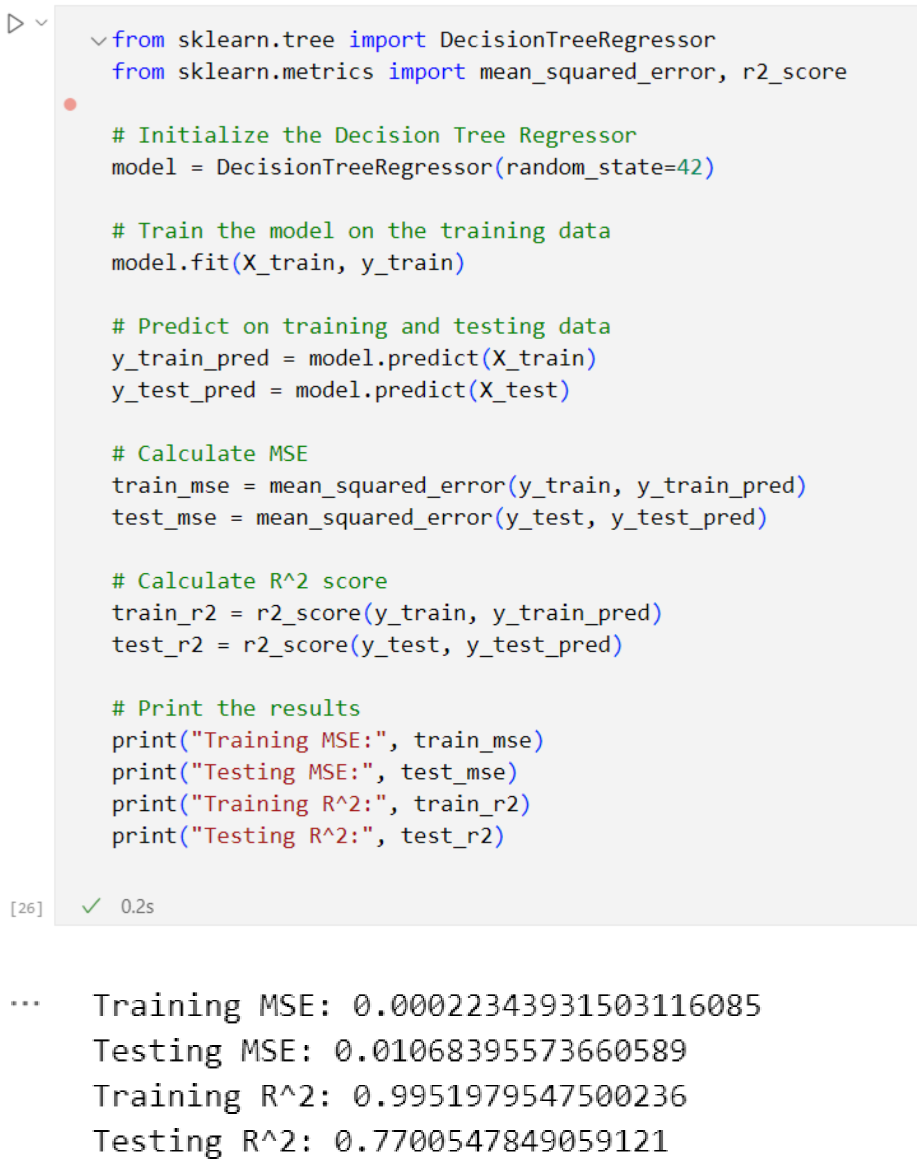
Here we can see that we have split the data into training and testing data.



```
[10788 rows x 4 columns]
y_train:
 26546    0.859869
9159     0.227821
14131    0.292101
15757    0.322971
24632    0.683462
           ...
11284    0.251338
44732    0.069795
38158    0.037195
860      0.137590
15795    0.324053
Name: price, Length: 43152, dtype: float64
y_test:
 1388     0.012597
50052    0.101368
41645    0.049305
42377    0.052873
17244    0.355463
           ...
44081    0.066389
23713    0.016597
31375    0.023517
21772    0.514137
4998     0.184679
Name: price, Length: 10788, dtype: float64
```

4. ML Techniques:

We began our analysis by implementing a **Decision Tree Regressor**, aiming to predict the target variable using a set of explanatory variables. The decision tree was chosen for its straightforwardness and ease of interpretation, which allows us to see how decisions are made at each node.



```python
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error, r2_score

# Initialize the Decision Tree Regressor
model = DecisionTreeRegressor(random_state=42)

# Train the model on the training data
model.fit(X_train, y_train)

# Predict on training and testing data
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

# Calculate MSE
train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)

# Calculate R^2 score
train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)

# Print the results
print("Training MSE:", train_mse)
print("Testing MSE:", test_mse)
print("Training R^2:", train_r2)
print("Testing R^2:", test_r2)
```

[26] ✓ 0.2s

```
Training MSE: 0.00022343931503116085
Testing MSE: 0.01068395573660589
Training R^2: 0.9951979547500236
Testing R^2: 0.7700547849059121
```

Performance Evaluation:

To evaluate the model's performance, we calculated two key metrics:

- **Mean Squared Error (MSE)**, which measures the average of the squares of the errors between actual and predicted values. Lower MSE values indicate better model performance.

- **R² Score**, which measures the proportion of variance in the dependent variable that is predictable from the independent variables. An R² Score closer to 1 suggests a high level of predictive accuracy.

Conclusion:The simple Decision Tree model demonstrated excellent performance on the training data, as indicated by the low training MSE and high R² Score. However, the performance on the testing data showed a notable decline. This pattern suggests that while the Decision Tree was very effective at learning the training data, it struggled to maintain this performance on the testing data, potentially indicating overfitting to the training set.

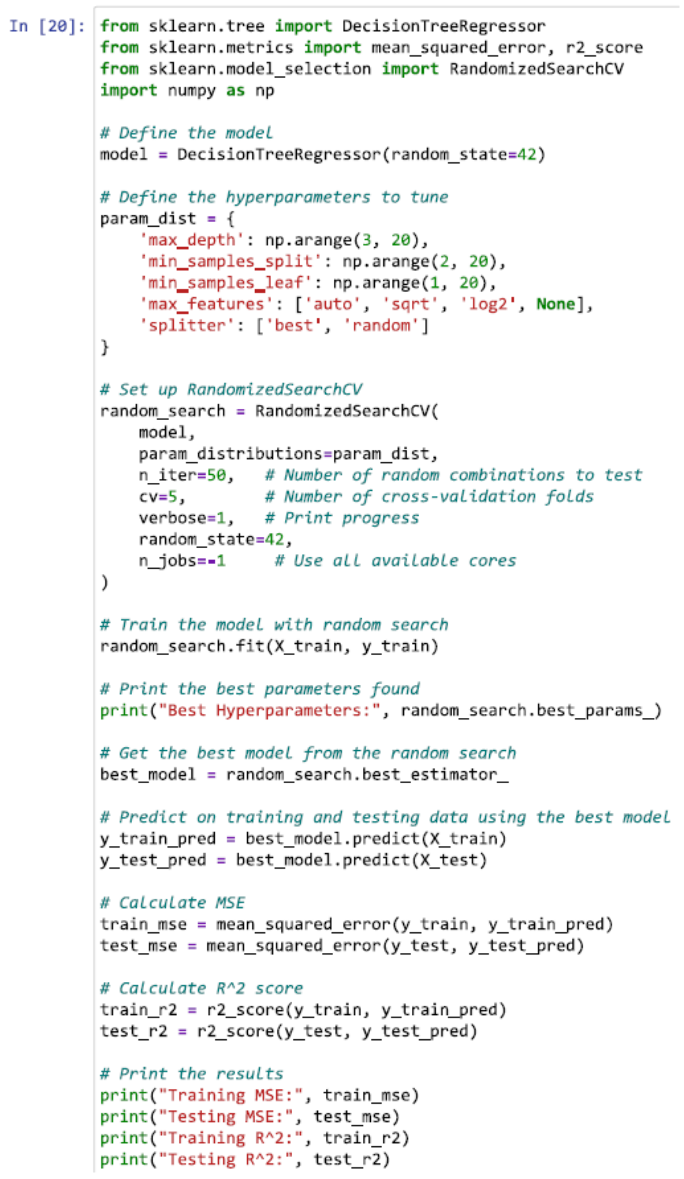
This leads to our next model:

Decision Tree with Hyperparameters.

Why Hyperparameters matters?

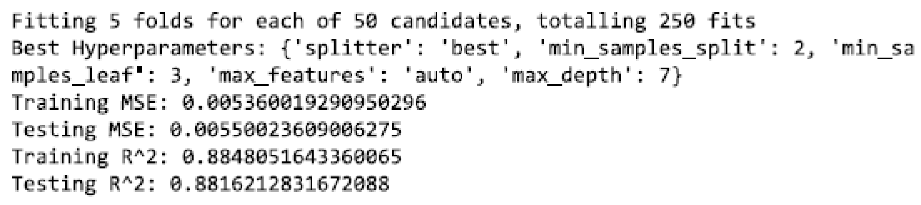
Hyperparameters are crucial in machine learning models because they control the behavior of the training algorithm and directly influence the performance of the model. Unlike model parameters that are learned during training, hyperparameters are set before the training process and need careful calibration. For Decision Tree models, hyperparameters determine how deep the tree grows, how many samples each node splits, and other essential aspects that can drastically affect both the model's complexity and its ability to generalize to new data.

To identify the optimal hyperparameters for the Decision Tree Regressor, we used RandomizedSearchCV from Scikit-Learn, which performs a random search over specified parameter values. This method is not only efficient but also effective in exploring a wide range of values and combinations.



```
In [20]: from sklearn.tree import DecisionTreeRegressor
         from sklearn.metrics import mean_squared_error, r2_score
         from sklearn.model_selection import RandomizedSearchCV
         import numpy as np

         # Define the model
         model = DecisionTreeRegressor(random_state=42)

         # Define the hyperparameters to tune
         param_dist = {
             'max_depth': np.arange(3, 20),
             'min_samples_split': np.arange(2, 20),
             'min_samples_leaf': np.arange(1, 20),
             'max_features': ['auto', 'sqrt', 'log2', None],
             'splitter': ['best', 'random']
         }

         # Set up RandomizedSearchCV
         random_search = RandomizedSearchCV(
             model,
             param_distributions=param_dist,
             n_iter=50,   # Number of random combinations to test
             cv=5,        # Number of cross-validation folds
             verbose=1,   # Print progress
             random_state=42,
             n_jobs=-1     # Use all available cores
         )

         # Train the model with random search
         random_search.fit(X_train, y_train)

         # Print the best parameters found
         print("Best Hyperparameters:", random_search.best_params_)

         # Get the best model from the random search
         best_model = random_search.best_estimator_

         # Predict on training and testing data using the best model
         y_train_pred = best_model.predict(X_train)
         y_test_pred = best_model.predict(X_test)

         # Calculate MSE
         train_mse = mean_squared_error(y_train, y_train_pred)
         test_mse = mean_squared_error(y_test, y_test_pred)

         # Calculate R^2 score
         train_r2 = r2_score(y_train, y_train_pred)
         test_r2 = r2_score(y_test, y_test_pred)

         # Print the results
         print("Training MSE:", train_mse)
         print("Testing MSE:", test_mse)
         print("Training R^2:", train_r2)
         print("Testing R^2:", test_r2)
```

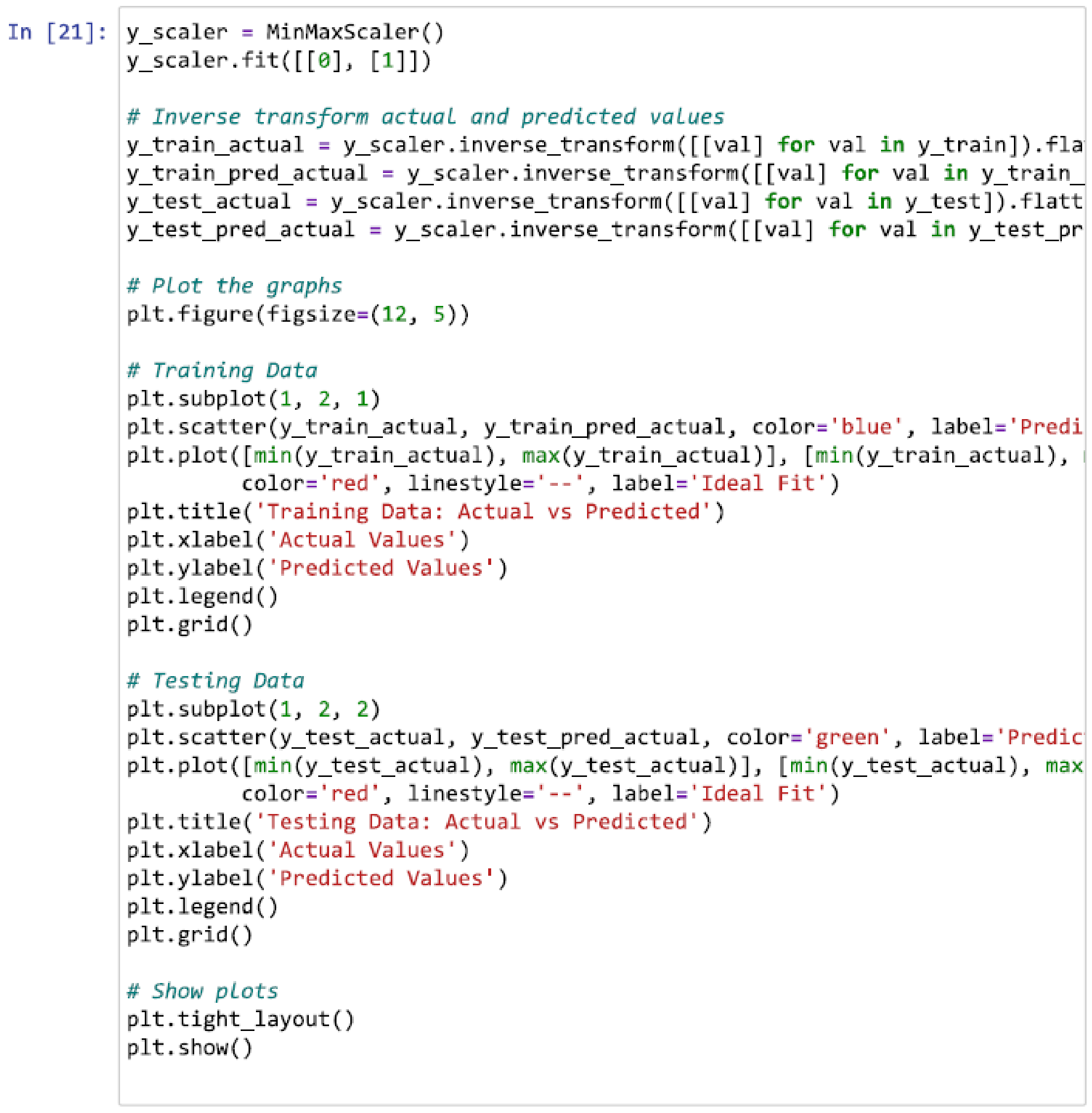
Model Results:



```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
Best Hyperparameters: {'splitter': 'best', 'min_samples_split': 2, 'min_sa
mples_leaf': 3, 'max_features': 'auto', 'max_depth': 7}
Training MSE: 0.005360019290950296
Testing MSE: 0.00550023609006275
Training R^2: 0.8848051643360065
Testing R^2: 0.8816212831672088
```

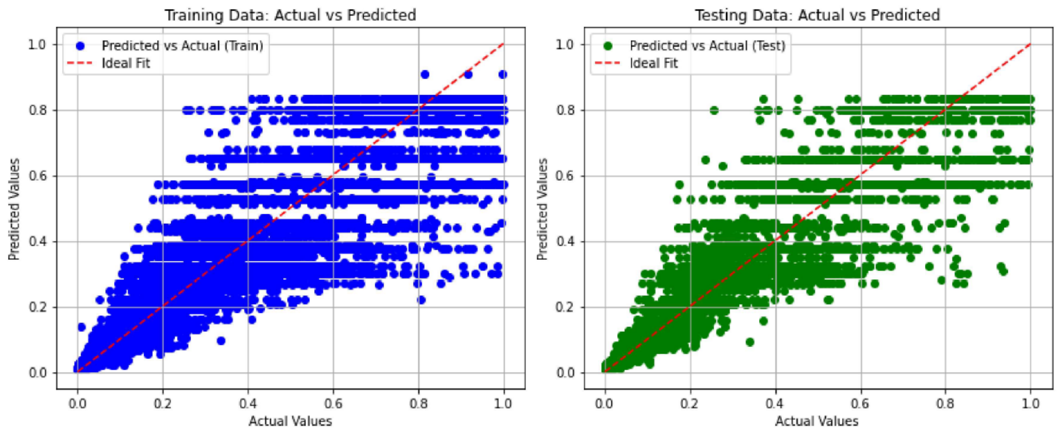
The use of hyperparameter tuning significantly improved the Decision Tree's generalizability and performance, achieving a balance between fitting the training data and performing well on unseen data. This process is vital in developing robust predictive models that perform consistently across different data samples.

Now, after developing a predictive model, it's essential to evaluate how well the model's predictions match the actual values. This step involved visualizing the relationship between the actual and predicted outcomes from both the training and testing datasets. By plotting these values against each other, we can visually assess the model's performance, including its accuracy and consistency across different data sets.



```
In [21]: y_scaler = MinMaxScaler()
         y_scaler.fit([[0], [1]])

         # Inverse transform actual and predicted values
         y_train_actual = y_scaler.inverse_transform([[val] for val in y_train]).fla
         y_train_pred_actual = y_scaler.inverse_transform([[val] for val in y_train_
         y_test_actual = y_scaler.inverse_transform([[val] for val in y_test]).flatt
         y_test_pred_actual = y_scaler.inverse_transform([[val] for val in y_test_pr

         # Plot the graphs
         plt.figure(figsize=(12, 5))

         # Training Data
         plt.subplot(1, 2, 1)
         plt.scatter(y_train_actual, y_train_pred_actual, color='blue', label='Predi
         plt.plot([min(y_train_actual), max(y_train_actual)], [min(y_train_actual),
                  color='red', linestyle='--', label='Ideal Fit')
         plt.title('Training Data: Actual vs Predicted')
         plt.xlabel('Actual Values')
         plt.ylabel('Predicted Values')
         plt.legend()
         plt.grid()

         # Testing Data
         plt.subplot(1, 2, 2)
         plt.scatter(y_test_actual, y_test_pred_actual, color='green', label='Predic
         plt.plot([min(y_test_actual), max(y_test_actual)], [min(y_test_actual), max
                  color='red', linestyle='--', label='Ideal Fit')
         plt.title('Testing Data: Actual vs Predicted')
         plt.xlabel('Actual Values')
         plt.ylabel('Predicted Values')
         plt.legend()
         plt.grid()

         # Show plots
         plt.tight_layout()
         plt.show()
```

Model Results:



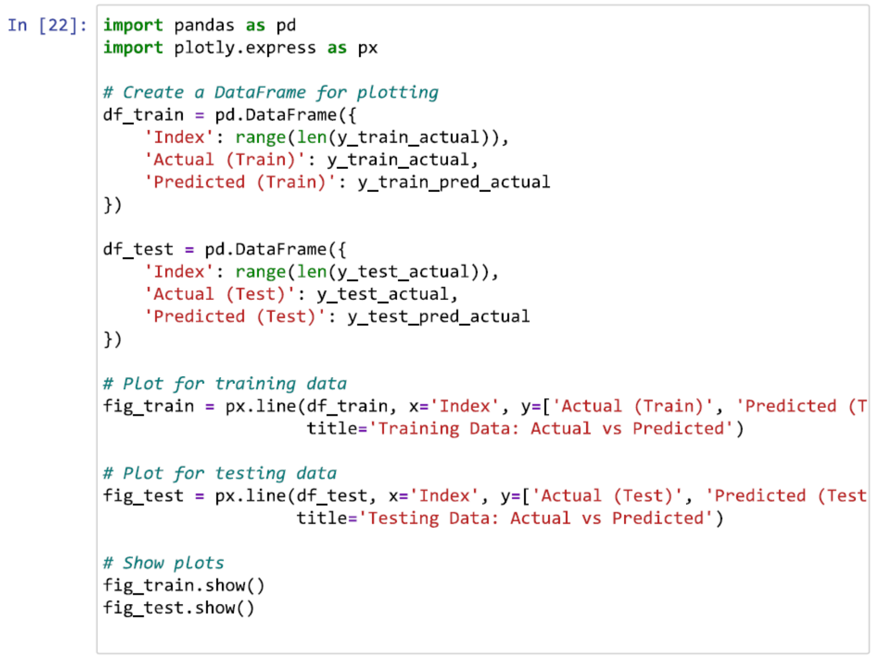
Training Data Plot (Blue): Most of the points cluster around the ideal fit line but spread as the values increase. The density of points near the line indicates that the model predicts the training data relatively well, although the spread suggests variance in the accuracy of predictions, especially at higher value ranges.

Testing Data Plot (Green): The spread in the testing data is wider than in the training data, suggesting that the model's predictions are less consistent when faced with new data. However, a substantial number of points still cluster near the ideal fit line, indicating that the model maintains a reasonable level of predictive accuracy on unseen data.

The visual analysis through scatter plots is a critical step in model evaluation as it provides a clear, intuitive way to understand the model's performance. It highlights areas where the model performs well and areas where improvements may be needed.

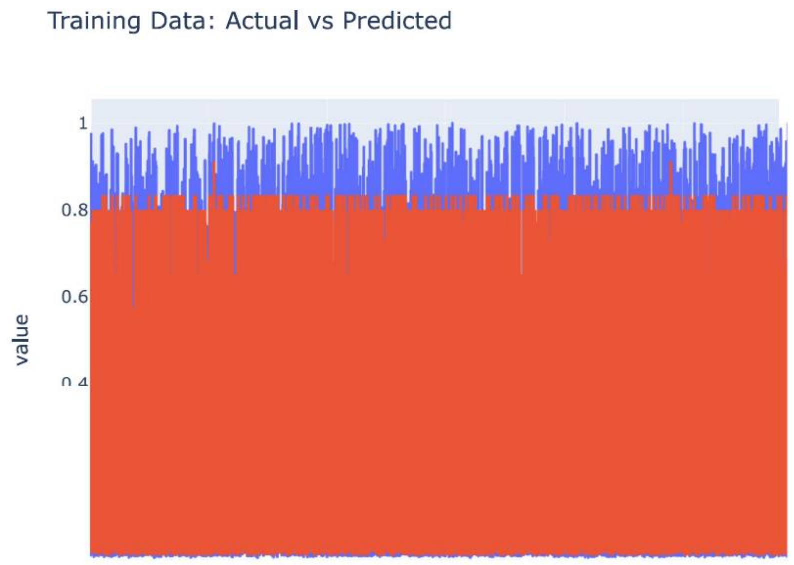
From the plots, while the model performs adequately, the visible spread especially in the testing data suggests potential overfitting to the training data or the need for further tuning to enhance the model's generalization capabilities.

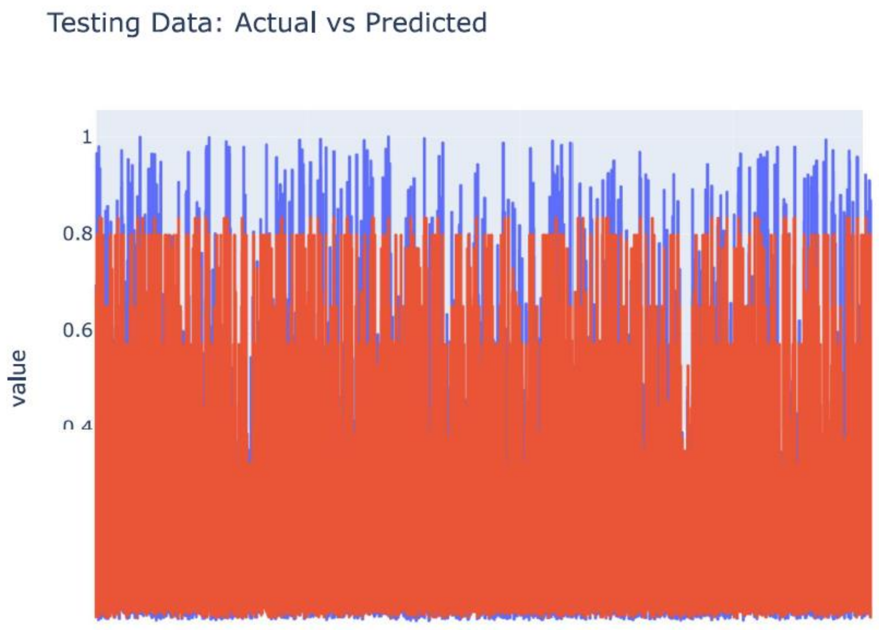
Line Plot:



```
In [22]: import pandas as pd
         import plotly.express as px

         # Create a DataFrame for plotting
         df_train = pd.DataFrame({
             'Index': range(len(y_train_actual)),
             'Actual (Train)': y_train_actual,
             'Predicted (Train)': y_train_pred_actual
         })

         df_test = pd.DataFrame({
             'Index': range(len(y_test_actual)),
             'Actual (Test)': y_test_actual,
             'Predicted (Test)': y_test_pred_actual
         })

         # Plot for training data
         fig_train = px.line(df_train, x='Index', y=['Actual (Train)', 'Predicted (T
                             title='Training Data: Actual vs Predicted')

         # Plot for testing data
         fig_test = px.line(df_test, x='Index', y=['Actual (Test)', 'Predicted (Test
                            title='Testing Data: Actual vs Predicted')

         # Show plots
         fig_train.show()
         fig_test.show()
```

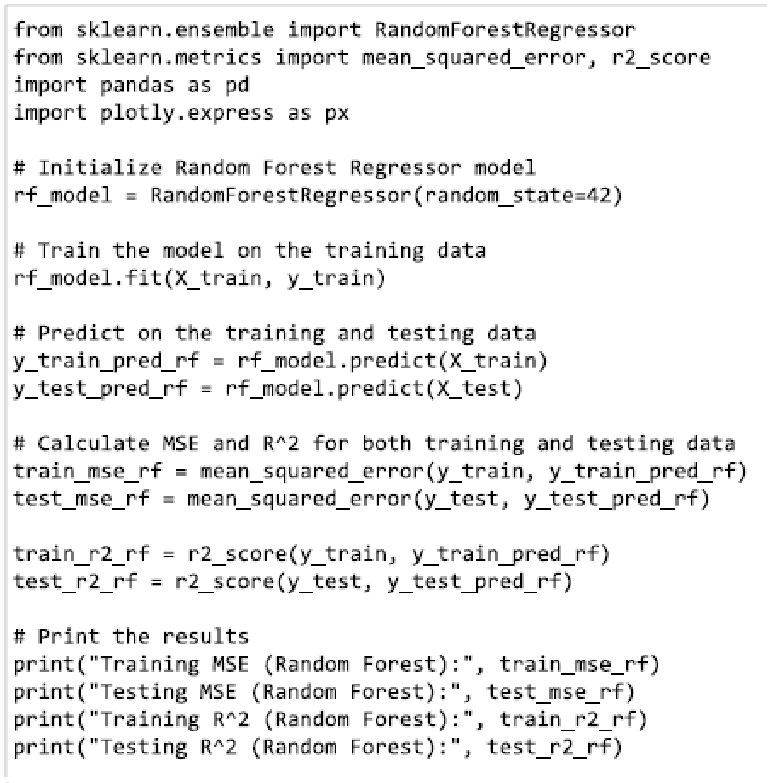
Model Results:



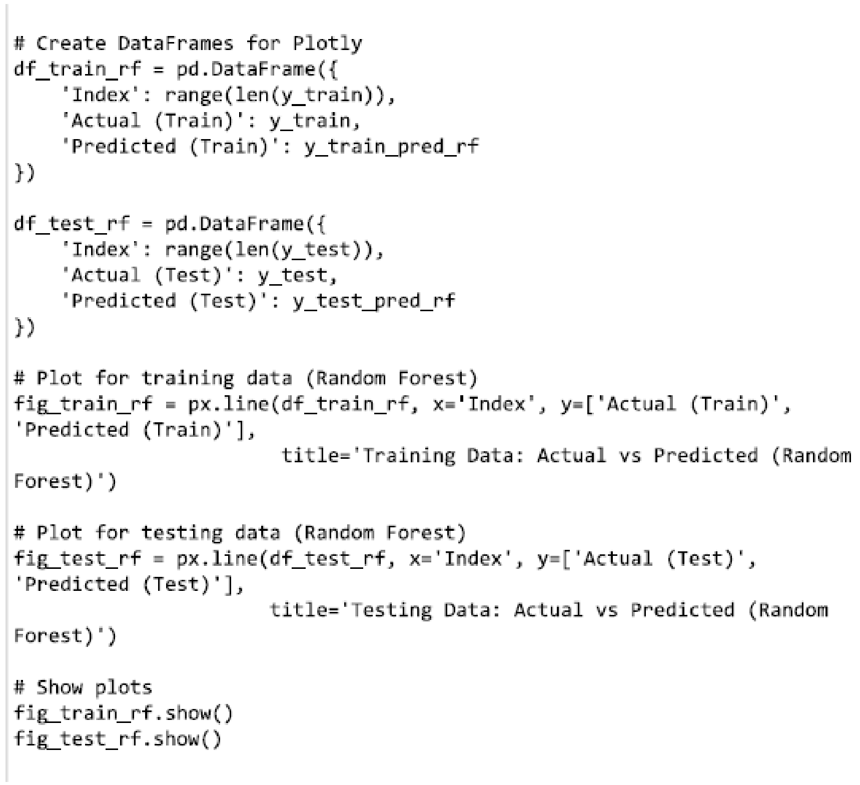


The line plots provide a clear visual confirmation that the model has learned to predict with high precision on the training data and maintains a reasonable level of accuracy on the testing data.

Now we perform RandomForest Regressor Model:

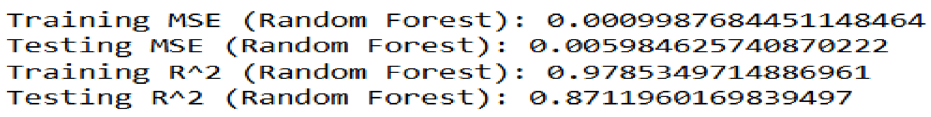


```python
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
import pandas as pd
import plotly.express as px

# Initialize Random Forest Regressor model
rf_model = RandomForestRegressor(random_state=42)

# Train the model on the training data
rf_model.fit(X_train, y_train)

# Predict on the training and testing data
y_train_pred_rf = rf_model.predict(X_train)
y_test_pred_rf = rf_model.predict(X_test)

# Calculate MSE and R^2 for both training and testing data
train_mse_rf = mean_squared_error(y_train, y_train_pred_rf)
test_mse_rf = mean_squared_error(y_test, y_test_pred_rf)

train_r2_rf = r2_score(y_train, y_train_pred_rf)
test_r2_rf = r2_score(y_test, y_test_pred_rf)

# Print the results
print("Training MSE (Random Forest):", train_mse_rf)
print("Testing MSE (Random Forest):", test_mse_rf)
print("Training R^2 (Random Forest):", train_r2_rf)
print("Testing R^2 (Random Forest):", test_r2_rf)
```

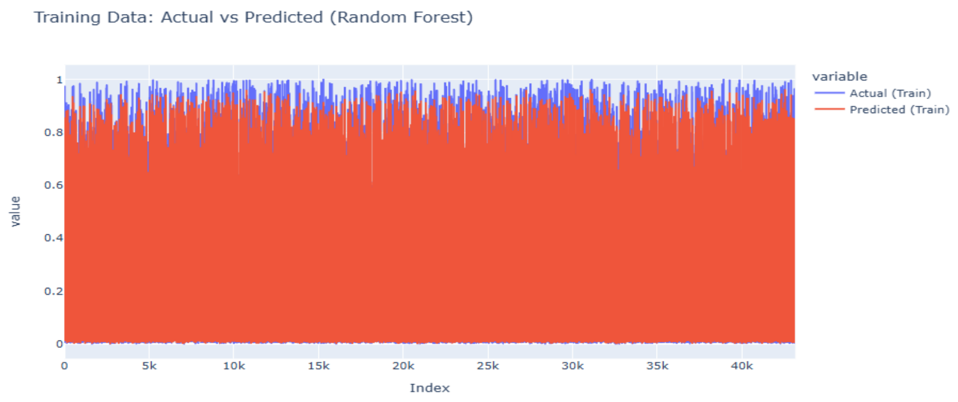


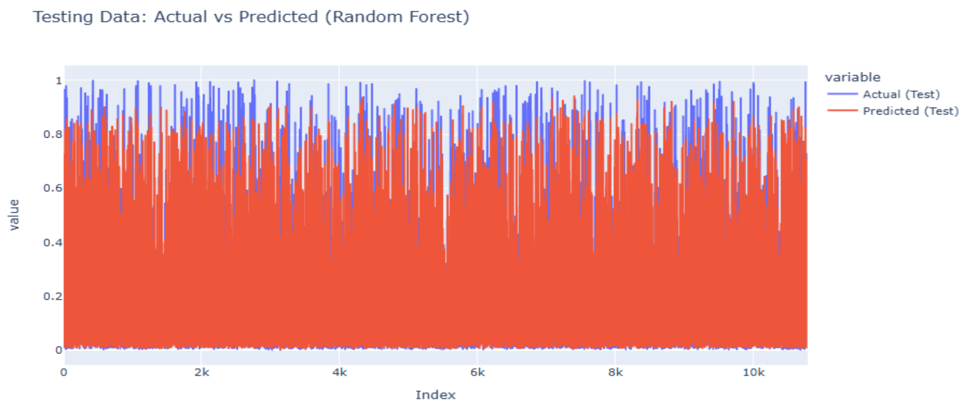
```python
# Create DataFrames for Plotly
df_train_rf = pd.DataFrame({
    'Index': range(len(y_train)),
    'Actual (Train)': y_train,
    'Predicted (Train)': y_train_pred_rf
})

df_test_rf = pd.DataFrame({
    'Index': range(len(y_test)),
    'Actual (Test)': y_test,
    'Predicted (Test)': y_test_pred_rf
})

# Plot for training data (Random Forest)
fig_train_rf = px.line(df_train_rf, x='Index', y=['Actual (Train)',
'Predicted (Train)'],
                       title='Training Data: Actual vs Predicted (Random
Forest)')

# Plot for testing data (Random Forest)
fig_test_rf = px.line(df_test_rf, x='Index', y=['Actual (Test)',
'Predicted (Test)'],
                      title='Testing Data: Actual vs Predicted (Random
Forest)')

# Show plots
fig_train_rf.show()
fig_test_rf.show()
```

Model Results:

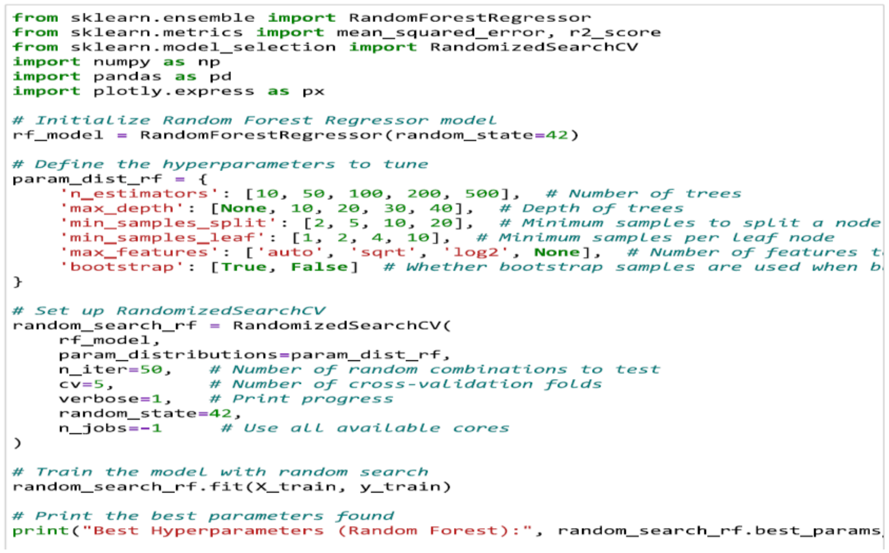


```
Training MSE (Random Forest): 0.0009987684451148464
Testing MSE (Random Forest): 0.005984625740870222
Training R^2 (Random Forest): 0.9785349714886961
Testing R^2 (Random Forest): 0.8711960169839497
```

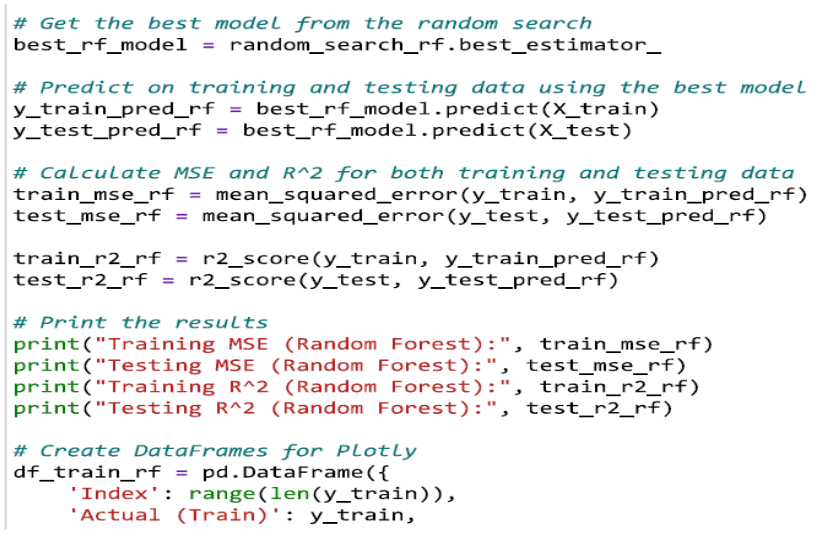




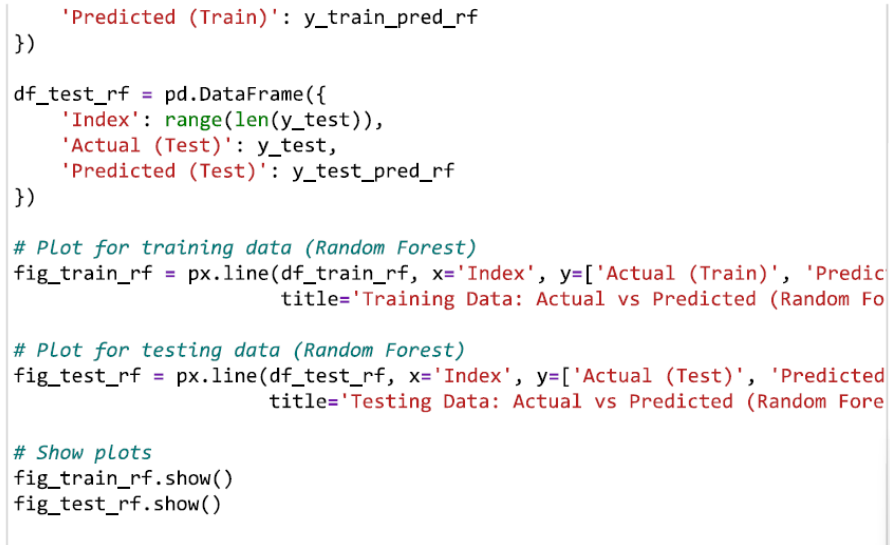
Now trying with Hyperparameters.



```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import RandomizedSearchCV
import numpy as np
import pandas as pd
import plotly.express as px

# Initialize Random Forest Regressor model
rf_model = RandomForestRegressor(random_state=42)

# Define the hyperparameters to tune
param_dist_rf = {
    'n_estimators': [10, 50, 100, 200, 500],  # Number of trees
    'max_depth': [None, 10, 20, 30, 40],  # Depth of trees
    'min_samples_split': [2, 5, 10, 20],  # Minimum samples to split a node
    'min_samples_leaf': [1, 2, 4, 10],  # Minimum samples per leaf node
    'max_features': ['auto', 'sqrt', 'log2', None],  # Number of features t
    'bootstrap': [True, False]  # Whether bootstrap samples are used when b
}

# Set up RandomizedSearchCV
random_search_rf = RandomizedSearchCV(
    rf_model,
    param_distributions=param_dist_rf,
    n_iter=50,    # Number of random combinations to test
    cv=5,         # Number of cross-validation folds
    verbose=1,    # Print progress
    random_state=42,
    n_jobs=-1      # Use all available cores
)

# Train the model with random search
random_search_rf.fit(X_train, y_train)

# Print the best parameters found
print("Best Hyperparameters (Random Forest):", random_search_rf.best_params
```

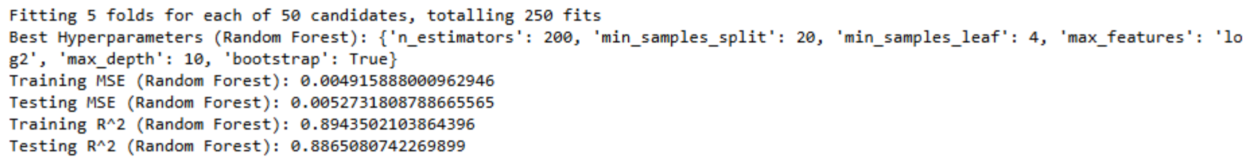


```
# Get the best model from the random search
best_rf_model = random_search_rf.best_estimator_

# Predict on training and testing data using the best model
y_train_pred_rf = best_rf_model.predict(X_train)
y_test_pred_rf = best_rf_model.predict(X_test)

# Calculate MSE and R^2 for both training and testing data
train_mse_rf = mean_squared_error(y_train, y_train_pred_rf)
test_mse_rf = mean_squared_error(y_test, y_test_pred_rf)

train_r2_rf = r2_score(y_train, y_train_pred_rf)
test_r2_rf = r2_score(y_test, y_test_pred_rf)

# Print the results
print("Training MSE (Random Forest):", train_mse_rf)
print("Testing MSE (Random Forest):", test_mse_rf)
print("Training R^2 (Random Forest):", train_r2_rf)
print("Testing R^2 (Random Forest):", test_r2_rf)

# Create DataFrames for Plotly
df_train_rf = pd.DataFrame({
    'Index': range(len(y_train)),
    'Actual (Train)': y_train,
```

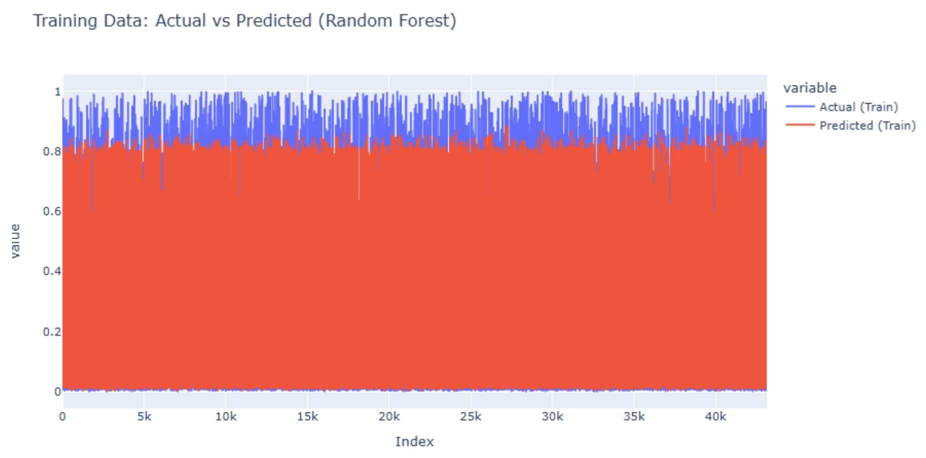


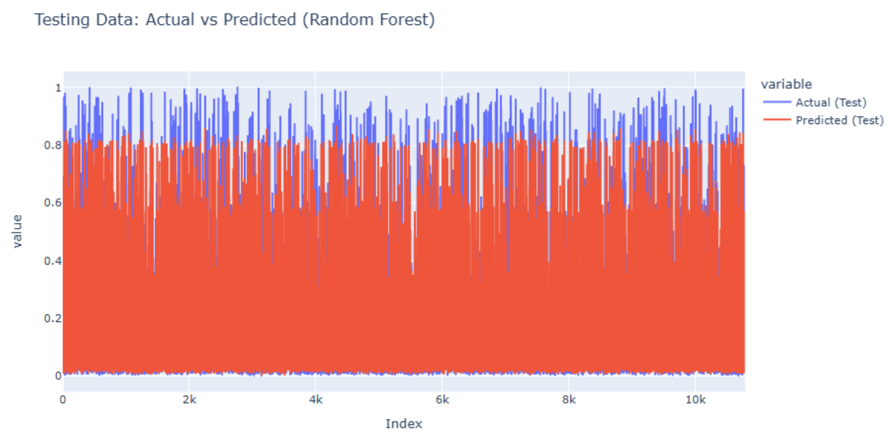
```
    'Predicted (Train)': y_train_pred_rf
})

df_test_rf = pd.DataFrame({
    'Index': range(len(y_test)),
    'Actual (Test)': y_test,
    'Predicted (Test)': y_test_pred_rf
})

# Plot for training data (Random Forest)
fig_train_rf = px.line(df_train_rf, x='Index', y=['Actual (Train)', 'Predic
                       title='Training Data: Actual vs Predicted (Random Fo

# Plot for testing data (Random Forest)
fig_test_rf = px.line(df_test_rf, x='Index', y=['Actual (Test)', 'Predicted
                      title='Testing Data: Actual vs Predicted (Random Fore

# Show plots
fig_train_rf.show()
fig_test_rf.show()
```

Model Results:



```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
Best Hyperparameters (Random Forest): {'n_estimators': 200, 'min_samples_split': 20, 'min_samples_leaf': 4, 'max_features': 'lo
g2', 'max_depth': 10, 'bootstrap': True}
Training MSE (Random Forest): 0.004915888000962946
Testing MSE (Random Forest): 0.0052731808788665565
Training R^2 (Random Forest): 0.8943502103864396
Testing R^2 (Random Forest): 0.8865080742269899
```





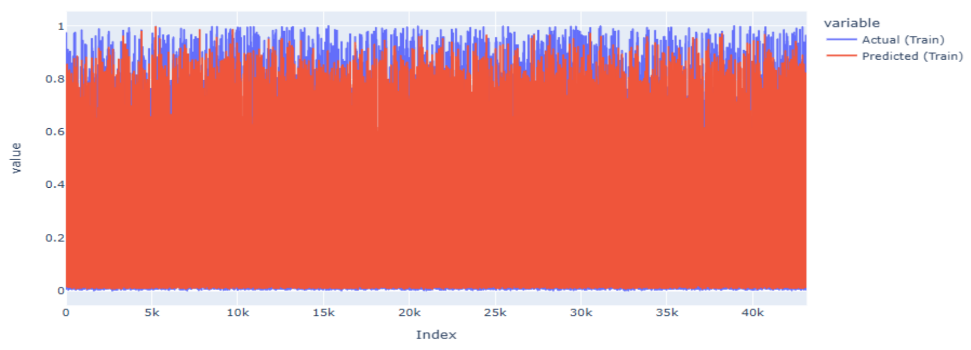
Also we implement a boosting algorithm called XG boost algorithm.

```
import xgboost as xgb
from sklearn.metrics import mean_squared_error, r2_score
import pandas as pd
import plotly.express as px

# Initialize XGBoost Regressor model
xgb_model = xgb.XGBRegressor(random_state=42)

# Train the model on the training data
xgb_model.fit(X_train, y_train)

# Predict on the training and testing data
y_train_pred_xgb = xgb_model.predict(X_train)
y_test_pred_xgb = xgb_model.predict(X_test)

# Calculate MSE and R^2 for both training and testing data
train_mse_xgb = mean_squared_error(y_train, y_train_pred_xgb)
test_mse_xgb = mean_squared_error(y_test, y_test_pred_xgb)

train_r2_xgb = r2_score(y_train, y_train_pred_xgb)
test_r2_xgb = r2_score(y_test, y_test_pred_xgb)

# Print the results
print("Training MSE (XGBoost):", train_mse_xgb)
print("Testing MSE (XGBoost):", test_mse_xgb)
print("Training R^2 (XGBoost):", train_r2_xgb)
print("Testing R^2 (XGBoost):", test_r2_xgb)

# Create DataFrames for Plotly
df_train_xgb = pd.DataFrame({
    'Index': range(len(y_train)),
    'Actual (Train)': y_train,
    'Predicted (Train)': y_train_pred_xgb
})

df_test_xgb = pd.DataFrame({
    'Index': range(len(y_test)),
    'Actual (Test)': y_test,
    'Predicted (Test)': y_test_pred_xgb
})

# Plot for training data (XGBoost)
fig_train_xgb = px.line(df_train_xgb, x='Index', y=['Actual (Train)',
'Predicted (Train)'],
                        title='Training Data: Actual vs Predicted
(XGBoost)')

# Plot for testing data (XGBoost)
fig_test_xgb = px.line(df_test_xgb, x='Index', y=['Actual (Test)',
'Predicted (Test)'],
                       title='Testing Data: Actual vs Predicted
(XGBoost)')

# Show plots
fig_train_xgb.show()
fig_test_xgb.show()
```

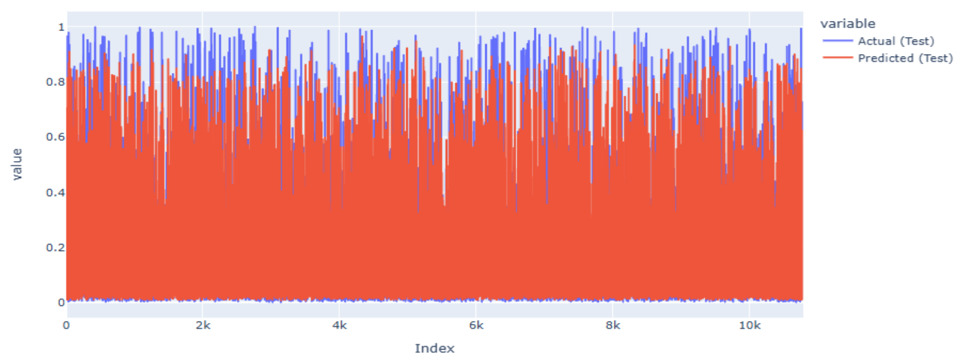
Model Results & Plots:

```
Training MSE (XGBoost): 0.003727518624951219
Testing MSE (XGBoost): 0.005408694084310784
Training R^2 (XGBoost): 0.9198900466345892
Testing R^2 (XGBoost): 0.8835914940817541
```

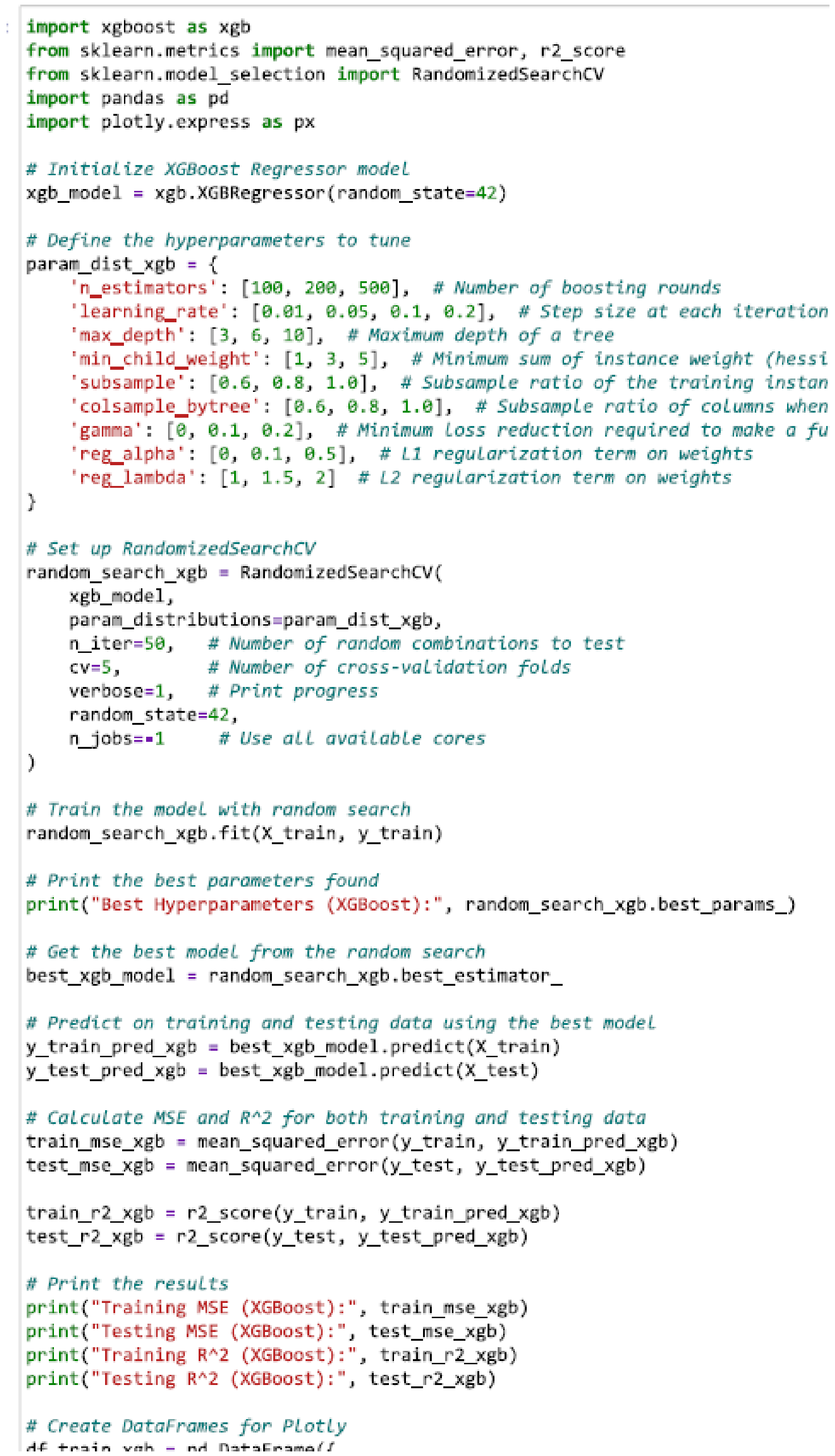
Training Data: Actual vs Predicted (XGBoost)



Testing Data: Actual vs Predicted (XGBoost)



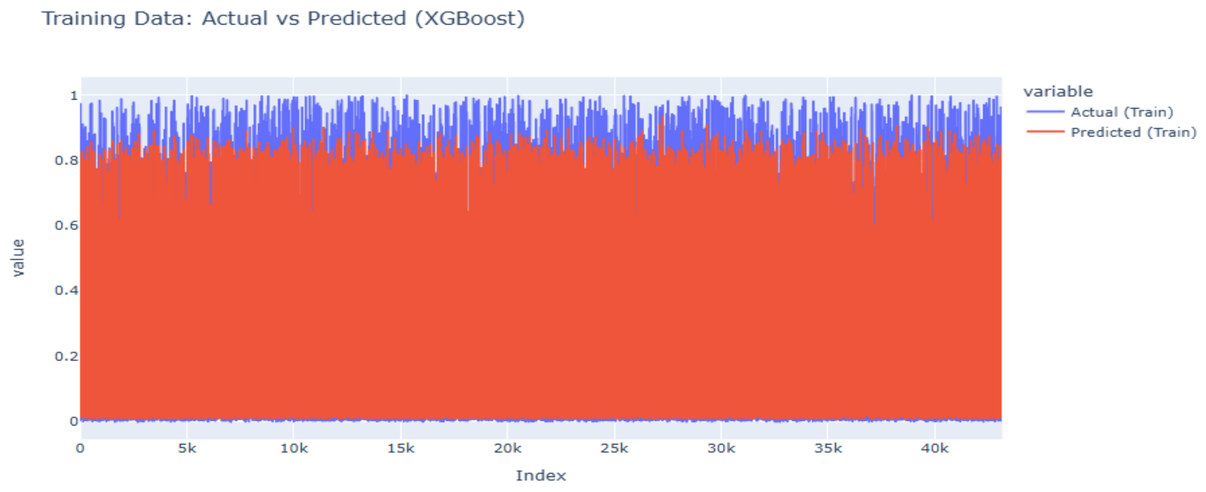
Now we try XG Boost algorithm with Hyperparameters:

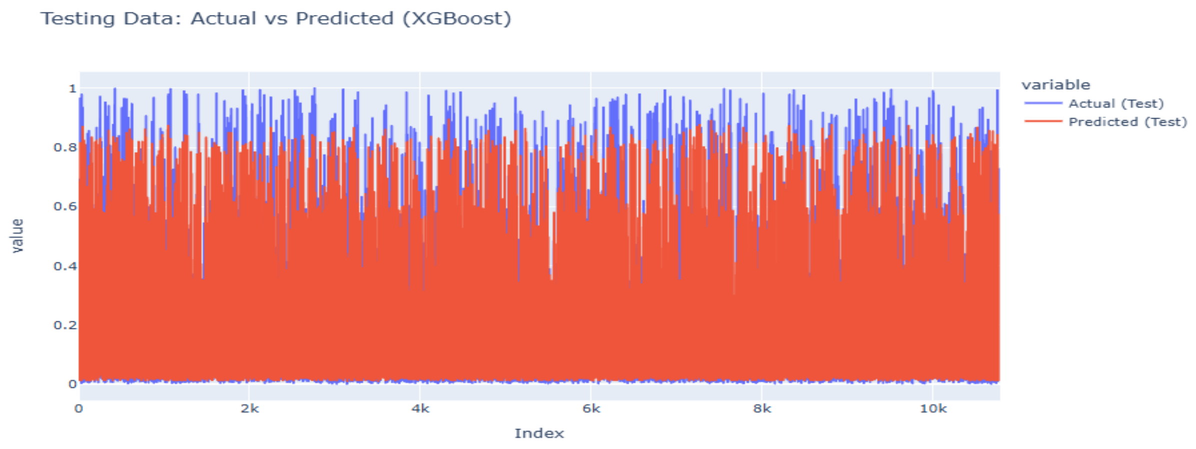


```
import xgboost as xgb
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import RandomizedSearchCV
import pandas as pd
import plotly.express as px

# Initialize XGBoost Regressor model
xgb_model = xgb.XGBRegressor(random_state=42)

# Define the hyperparameters to tune
param_dist_xgb = {
    'n_estimators': [100, 200, 500],  # Number of boosting rounds
    'learning_rate': [0.01, 0.05, 0.1, 0.2],  # Step size at each iteration
    'max_depth': [3, 6, 10],  # Maximum depth of a tree
    'min_child_weight': [1, 3, 5],  # Minimum sum of instance weight (hessi
    'subsample': [0.6, 0.8, 1.0],  # Subsample ratio of the training instan
    'colsample_bytree': [0.6, 0.8, 1.0],  # Subsample ratio of columns when
    'gamma': [0, 0.1, 0.2],  # Minimum loss reduction required to make a fu
    'reg_alpha': [0, 0.1, 0.5],  # L1 regularization term on weights
    'reg_lambda': [1, 1.5, 2]  # L2 regularization term on weights
}

# Set up RandomizedSearchCV
random_search_xgb = RandomizedSearchCV(
    xgb_model,
    param_distributions=param_dist_xgb,
    n_iter=50,    # Number of random combinations to test
    cv=5,         # Number of cross-validation folds
    verbose=1,    # Print progress
    random_state=42,
    n_jobs=-1      # Use all available cores
)

# Train the model with random search
random_search_xgb.fit(X_train, y_train)

# Print the best parameters found
print("Best Hyperparameters (XGBoost):", random_search_xgb.best_params_)

# Get the best model from the random search
best_xgb_model = random_search_xgb.best_estimator_

# Predict on training and testing data using the best model
y_train_pred_xgb = best_xgb_model.predict(X_train)
y_test_pred_xgb = best_xgb_model.predict(X_test)

# Calculate MSE and R^2 for both training and testing data
train_mse_xgb = mean_squared_error(y_train, y_train_pred_xgb)
test_mse_xgb = mean_squared_error(y_test, y_test_pred_xgb)

train_r2_xgb = r2_score(y_train, y_train_pred_xgb)
test_r2_xgb = r2_score(y_test, y_test_pred_xgb)

# Print the results
print("Training MSE (XGBoost):", train_mse_xgb)
print("Testing MSE (XGBoost):", test_mse_xgb)
print("Training R^2 (XGBoost):", train_r2_xgb)
print("Testing R^2 (XGBoost):", test_r2_xgb)

# Create DataFrames for Plotly
df_train_xgb = pd.DataFrame({
    'Index': range(len(y_train)),
    'Actual (Train)': y_train,
    'Predicted (Train)': y_train_pred_xgb
})

df_test_xgb = pd.DataFrame({
    'Index': range(len(y_test)),
    'Actual (Test)': y_test,
    'Predicted (Test)': y_test_pred_xgb
})

# Plot for training data (XGBoost)
fig_train_xgb = px.line(df_train_xgb, x='Index', y=['Actual (Train)', 'Pred
                        title='Training Data: Actual vs Predicted (XGBoost)

# Plot for testing data (XGBoost)
fig_test_xgb = px.line(df_test_xgb, x='Index', y=['Actual (Test)', 'Predict
                       title='Testing Data: Actual vs Predicted (XGBoost)')

# Show plots
fig_train_xgb.show()
fig_test_xgb.show()
```

Model Results & Plots :

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
Best Hyperparameters (XGBoost): {'subsample': 0.8, 'reg_lambda': 2, 'reg_alpha': 0.1, 'n_estimators': 100, 'min_child_weight':
5, 'max_depth': 6, 'learning_rate': 0.1, 'gamma': 0, 'colsample_bytree': 0.8}
Training MSE (XGBoost): 0.004742346479959888
Testing MSE (XGBoost): 0.005272774446695709
Training R^2 (XGBoost): 0.8980798773722618
Testing R^2 (XGBoost): 0.8865168216549323
```





5. Summary of the Project

Introduction to Modeling Techniques:

Our analysis began with the implementation of a Decision Tree Regressor to predict a target variable using various explanatory variables. The Decision Tree was selected for its simplicity and clarity in decision-making at each node, making it highly interpretable.

Performance Evaluation of Decision Tree:

The performance of the Decision Tree was evaluated using:

- **Mean Squared Error (MSE)**: Lower values indicate better performance.
- **R^2 Score**: Closer to 1 suggests high predictive accuracy.

While the Decision Tree model showed excellent performance on the training data, its performance declined on the testing data. This highlighted potential overfitting, where the model excelled in learning training data but underperformed with new, unseen data.

Hyperparameter Tuning with RandomizedSearchCV:

To enhance the Decision Tree's ability to generalize, we employed RandomizedSearchCV for hyperparameter tuning. This process effectively balanced the tree's depth and node splits, improving its generalizability and performance across unseen data samples.

Visualization of Model Predictions:

Post-tuning, the model's predictions were visually compared with actual outcomes using scatter plots for both training and testing datasets:

- **Training Data Plot**: Showed a high density of points near the ideal fit line, indicating accurate predictions, though with some spread at higher value ranges.
- **Testing Data Plot**: Displayed a wider spread, suggesting less consistency in predictions when faced with new data but still maintaining reasonable accuracy.

Advancement to Ensemble Methods – Random Forest Regressor:

Given the single tree's tendency for overfitting, we transitioned to an ensemble method, the Random Forest Regressor, known for its robustness and better performance due to averaging multiple decision trees trained on different parts of the same data set.

Implementation and Results of Random Forest:

The Random Forest model further stabilized the predictions, demonstrated by improved MSE and R^2 scores for both training and testing datasets. This model's ability to perform consistently well underscored the benefits of ensemble learning in handling complex datasets.

Introduction of Boosting Algorithm – XGBoost:

To explore the advantages of boosting algorithms, we implemented XGBoost, which is renowned for its performance and speed. This method focuses on sequentially correcting the mistakes of weak learners to improve overall prediction accuracy.

XGBoost Performance with and without Hyperparameters:

Both the vanilla and hyperparameter-tuned versions of XGBoost were tested. The hyperparameter tuning further refined the model, enhancing its accuracy and efficiency, as reflected in the comparative analysis of MSE and R^2 scores before and after tuning.

Conclusion:

Throughout our series of model implementations and enhancements—from simple trees to advanced ensemble and boosting methods—our analysis demonstrated a progressive improvement in predictive accuracy and model robustness. The visual and quantitative assessments provided clear insights into each model's strengths and limitations, guiding our future choices towards more sophisticated algorithms and tuning strategies to achieve optimal performance in predictive tasks. This systematic approach highlighted the importance of model selection, hyperparameter tuning, and the potential of ensemble and boosting methods in achieving superior predictive modeling outcomes.