**INFOSYS MILESTONE-3**

**House Price Prediction**

**About 1st Dataset**

**Understanding the Data**

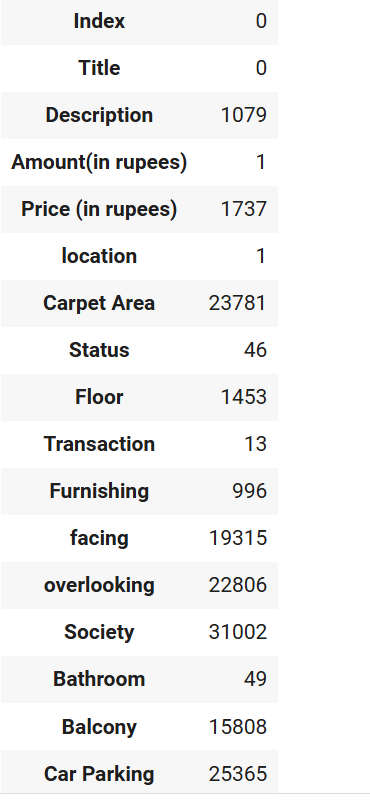
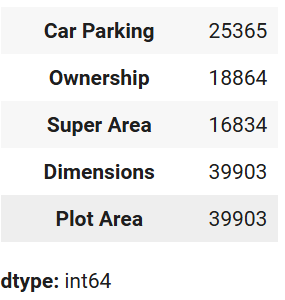
Based on the provided data, it seems to be a real estate listing for properties, primarily flats. The columns appear to represent:

1. **Index:** A unique identifier for each property.
2. **Title:** A brief description of the property.
3. **Description:** A detailed description of the property.
4. **Amount (in rupees):** The asking price of the property.
5. **Price (in rupees):** The price per square foot of the property.
6. **Location:** The city or locality where the property is located.
7. **Carpet Area:** The usable floor area of the property.
8. **Status:** The current status of the property (e.g., Ready to Move, Under Construction).
9. **Floor:** The floor number of the property.
10. **Transaction:** The type of transaction (e.g., Resale, New Property).
11. **Facing:** The direction the property faces.
12. **Overlooking:** The view from the property.
13. **Society:** The name of the society or complex where the property is located.
14. **Bathroom:** The number of bathrooms in the property.
15. **Balcony:** The number of balconies in the property.
16. **Car Parking:** The available car parking spaces.
17. **Ownership:** The type of ownership (e.g., Freehold, Leasehold).
18. **Super Area:** The total area of the property, including common areas.
19. **Dimensions:** The dimensions of the property.
20. **Plot Area:** The land area of the property.

**Code:** df.isnull().sum()

**Description:** .isnull().sum() method in pandas is used to calculate the number of missing values in each column of a DataFrame. It returns a Series containing the number of missing values for each column.

**Output:**

** **

**Code:** df['Price (in rupees)'] = df['Price (in rupees)'].fillna(df['Price (in rupees)'].mean())

**Description:** df['Price (in rupees)']: This part accesses the specific column named "Price (in rupees)" from the DataFrame df.

**.fillna(df['Price (in rupees)'].mean())**: This part fills the missing values (NaN) in the specified column with the mean value of that column.

**In simpler terms:** This line of code is used to handle missing values in the "Price (in rupees)" column of the DataFrame df. It replaces the missing values with the average price of all the available values in that column.

**Code:** df['Carpet Area'] = pd.to\_numeric(df['Carpet Area'].str.replace(' sqft', ''), errors='coerce')

**Description :**  The code df['Carpet Area'] = pd.to\_numeric(df['Carpet Area'].str.replace(' sqft', ''), errors='coerce') converts the 'Carpet Area' column from a text-based format (e.g., '500 sqft') to a numeric format (e.g., 500). It does this by removing the 'sqft' unit and then converting the remaining string to a number. The errors='coerce' argument handles potential errors during the conversion, setting invalid values to NaN.

**Code:** df['Carpet Area'] = df['Carpet Area'].fillna(df['Carpet Area'].median())

**Description** : This code fills missing values in the 'Carpet Area' column with the median value of that column. This is often used when there are outliers in the data, as the median is less sensitive to extreme values compared to the mean.

**Code:** df['location'] = df['location'].astype('category')

**Description :**  This code converts the 'location' column in the DataFrame df to a categorical data type. Categorical data types are often used to represent discrete categories or labels. By converting the 'location' column to a category, you can optimize memory usage and improve the performance of certain operations, especially when dealing with large datasets.

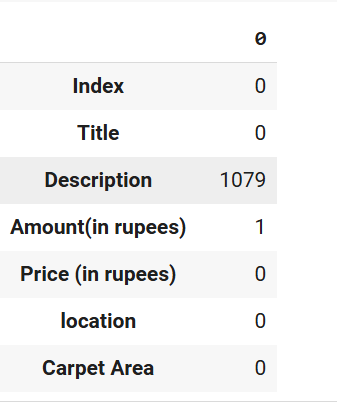
**Code:**  df['location'] = df['location'].fillna(df['location'].mode()[0])

**Description :** This code fills missing values in the 'location' column with the most frequent location (mode). The mode()[0] part selects the first mode in case there are multiple modes. This is a common technique for handling missing categorical data.

**Code**: df.isnull().sum()

**Description :** . It returns a Series containing the number of missing values for each column.

**Output:**

****

**Code:** import matplotlib.pyplot as plt

import seaborn as sns

# Calculate average price per location

average\_price\_by\_location = df.groupby('location', observed=False)['Price (in rupees)'].mean().reset\_index()

# Create the bar chart

plt.figure(figsize=(12, 6))  # Adjust figure size if needed

sns.barplot(x='location', y='Price (in rupees)', data=average\_price\_by\_location)

plt.title('Average House Price by Location')

plt.xlabel('Location')

plt.ylabel('Average Price (in rupees)')

plt.xticks(rotation=45, ha='right')  # Rotate x-axis labels for better readability

plt.tight\_layout()

plt.show()

**Description :**  Imports necessary libraries: matplotlib.pyplot and seaborn.

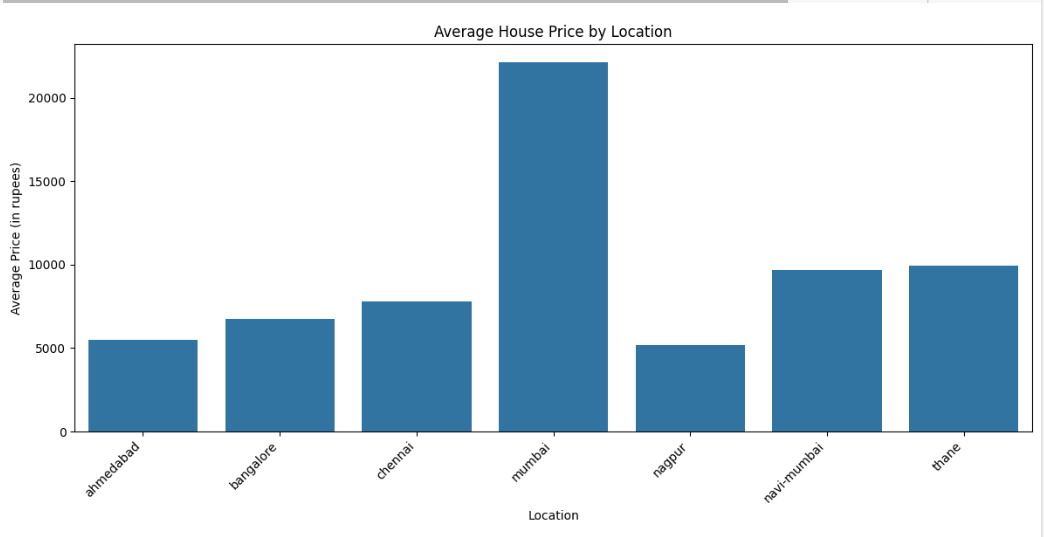
 Calculates average price per location.

 Creates a bar plot using Seaborn.

 Customizes the plot with title, labels, and x-axis rotation.

 Displays the plot.

**Output:**

****

**Code:** df['Price (in rupees)'] = df['Price (in rupees)'].astype(int)

**Description :** This code converts the data type of the 'Price (in rupees)' column from its current type (e.g., float, string) to an integer data type. This can be useful for certain calculations or operations that require integer values.

**Code:** # Convert 'Bathroom' to numeric, handling errors

df['Bathroom'] = pd.to\_numeric(df['Bathroom'], errors='coerce')

# Fill missing values in 'Bathroom' with the median

df['Bathroom'] = df['Bathroom'].fillna(df['Bathroom'].median())

**Description :** pd.to\_numeric(df['Bathroom'], errors='coerce'): This line attempts to convert the values in the 'Bathroom' column to numeric data type. If it encounters non-numeric values (e.g., text, symbols), it converts them to NaN (Not a Number).

df['Bathroom'].fillna(df['Bathroom'].median()): This line replaces the missing values (NaN) in the 'Bathroom' column with the median value of the non-missing values in that column. Using the median is often a robust approach to handle missing numerical data, as it's less sensitive to outliers compared to the mean.

**Code:** # Convert 'Balcony' to numeric, handling errors

df['Balcony'] = pd.to\_numeric(df['Balcony'], errors='coerce')

# Fill missing values in 'Balcony' with the median or mode

if df['Balcony'].dtype.kind in 'biufc':  # Check if numeric

    df['Balcony'] = df['Balcony'].fillna(df['Balcony'].median())  # Use median for numeric

else:

    df['Balcony'] = df['Balcony'].fillna(df['Balcony'].mode()[0])  # Use mode for non-numeric

**Description :** pd.to\_numeric(df['Bathroom'], errors='coerce'): This line attempts to convert the values in the 'Balcony' column to a numeric data type. If it encounters non-numeric values (e.g., text, symbols), it converts them to NaN.

The code checks the data type of the 'Balcony' column:

* If it's a numeric data type (integer, float, etc.), it fills missing values with the median using df['Balcony'].fillna(df['Balcony'].median()).
* If it's a non-numeric data type (e.g., categorical), it fills missing values with the most frequent value (mode) using df['Balcony'].fillna(df['Balcony'].mode()[0]).

**Code:** import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

**Description :** This code snippet imports essential libraries for machine learning and data analysis:

1. pandas:
   * For data manipulation and analysis. It's used to read, clean, and preprocess data.
2. sklearn.model\_selection:
   * For splitting data into training and testing sets. train\_test\_split is a function used to randomly split data into training and testing sets, which is crucial for model training and evaluation.
3. sklearn.linear\_model:
   * For linear regression models. LinearRegression is a class that implements linear regression, a statistical method used to model the relationship between a dependent variable and one or more independent variables.
4. sklearn.metrics:
   * For evaluating model performance.
     + mean\_squared\_error: Calculates the mean squared error, a common metric to assess the quality of predictions.
     + r2\_score: Calculates the coefficient of determination (R-squared), which measures the proportion of variance in the dependent variable explained by the independent variables.

Sources and related content

**Code: f**eatures = ['location', 'Carpet Area', 'Bathroom', 'Balcony']  # Removed 'Car Parking'

target = 'Price (in rupees)'

# One-hot encoding

X = pd.get\_dummies(df[features], columns=['location'], drop\_first=True)

y = df[target]

# Split data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  # Adjust test\_size and random\_state as needed

# Create and train the model

model = LinearRegression()

model.fit(X\_train, y\_train)

# Make predictions on the testing data

y\_pred = model.predict(X\_test)

# Evaluate the Model

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

# Print the evaluation metrics

print(f"Mean Squared Error: {mse}")

print(f"R-squared: {r2}")

**Description :** This code snippet defines features, targets, performs data preparation, trains a linear regression model, and evaluates its performance. Here's a breakdown:

1. Feature and Target Definition:

* features: Defines a list of features (location, Carpet Area, Bathroom, Balcony) to be used for prediction. Note that 'Car Parking' has been removed (potentially due to data quality issues or for other reasons).
* target: Defines the target variable (Price (in rupees)) that the model will predict.

2. Data Preparation:

* One-hot encoding:
  + X = pd.get\_dummies(df[features], columns=['location'], drop\_first=True):
    - Converts the categorical feature location into one-hot encoded features. This helps linear regression models handle categorical data effectively.
    - drop\_first=True avoids creating a dummy variable for the first category, reducing multicollinearity.
* Splitting data:
  + X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42):
    - Splits the data into training and testing sets for model training and evaluation.
    - test\_size=0.2 assigns 20% of the data to the testing set. You can adjust this value based on your needs.
    - random\_state=42 sets a seed for random splitting, ensuring reproducibility.

3. Model Training:

* model = LinearRegression(): Creates a linear regression model.
* model.fit(X\_train, y\_train): Trains the model using the training data (X\_train, y\_train).

4. Model Evaluation:

* y\_pred = model.predict(X\_test): Makes predictions on the testing data.
* mse = mean\_squared\_error(y\_test, y\_pred): Calculates the mean squared error (MSE) between the actual and predicted prices.
* r2 = r2\_score(y\_test, y\_pred): Calculates the R-squared score, which measures the proportion of variance explained by the model.
* print statements: Display the calculated MSE and R-squared values, providing insights into the model's performance.

5. Prediction for New Data (commented out):

* The commented section indicates the possibility of using the trained model to make predictions for new data. You would need to provide new data points with the same features (location, Carpet Area, Bathroom, Balcony) to make price predictions.

**Output:**

Mean Squared Error: 8756919.791605597

R-squared: 0.3968986132095417

**Code:** import pandas as pd

from sklearn.model\_selection import train\_test\_split, GridSearchCV

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import mean\_squared\_error, r2\_score

# ... (Data cleaning and feature engineering steps) ...

# Feature selection

features = ['location', 'Carpet Area', 'Bathroom', 'Balcony']  # Example features

target = 'Price (in rupees)'

# One-hot encoding

X = pd.get\_dummies(df[features], columns=['location'], drop\_first=True)

y = df[target]

# Split data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create and train the model

model = RandomForestRegressor(random\_state=42)

# Hyperparameter tuning (example using GridSearchCV)

param\_grid = {

    'n\_estimators': [100, 200, 300],

    'max\_depth': [None, 5, 10],

}

grid\_search = GridSearchCV(model, param\_grid, cv=5, scoring='neg\_mean\_squared\_error')

grid\_search.fit(X\_train, y\_train)

best\_model = grid\_search.best\_estimator\_

# Make predictions

y\_pred = best\_model.predict(X\_test)

# Evaluate the model

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print(f"Mean Squared Error: {mse}")

print(f"R-squared: {r2}")

# ... (Prediction for new data) ...

**Description :** The provided code implements a machine learning pipeline for real estate price prediction using a Random Forest Regressor. Here's a concise breakdown:

1. Data Preparation:
   * Feature Selection: Identifies relevant features like location, area, number of bathrooms, and balconies.
   * One-Hot Encoding: Converts categorical features (like location) into numerical representations suitable for the model.
   * Data Splitting: Divides the data into training and testing sets for model training and evaluation.
2. Model Selection and Training:
   * Random Forest Regressor: Chooses a Random Forest Regressor as the model.
   * Hyperparameter Tuning: Uses GridSearchCV to find the best hyperparameters for the model (number of trees, maximum depth, etc.).
   * Model Training: Trains the model on the training data.
3. Model Evaluation:
   * Prediction: Makes predictions on the testing data.
   * Evaluation Metrics: Calculates the Mean Squared Error (MSE) and R-squared to assess the model's performance.

This approach provides a robust and flexible framework for real estate price prediction, allowing for customization and experimentation with different features, models, and hyperparameters.

**Output:**

Mean Squared Error: 6104483.225397729

R-squared: 0.5795756514287511

**Code:** # Get the column order from the training data

column\_order = X\_train.columns

# Create a new data point for prediction (example)

new\_data = pd.DataFrame({

    'location\_bangalore': [1],  # Set to 1 for Bangalore, 0 otherwise

    'location\_chennai': [0],

    'location\_mumbai': [0],

    'location\_nagpur': [0],

    'location\_navi-mumbai': [0],

    'location\_thane': [0],

    'Carpet Area': [1200],

    'Bathroom': [2],

    'Balcony': [1],

    'Total Area': [1300],  # Include engineered features

    'Location Amenities': [0.8]  # Example value for location amenities

}, columns=column\_order)  # Specify column order

# Make predictions using the best model

predicted\_price = best\_model.predict(new\_data)

# Print the predicted price

print(f"Predicted Price: {predicted\_price[0]}")

**Description** : The code snippet demonstrates how to use a trained machine learning model to predict the price of a new property. Here's a concise breakdown:

1. Preserves column order: Ensures the new data has the same feature order as the training data.
2. Creates new data point: Defines a new data point with values for each feature, including one-hot encoded categorical features and numerical features.
3. Makes prediction: Uses the trained model to predict the price of the new property.
4. Prints the predicted price: Displays the predicted price.

**Output:**

Predicted Price: 6608.053270612152

**Colab\_link:** [**https://colab.research.google.com/drive/1HykNzK5ht42hDzCH1z1i-Z1YGewx-V9X?usp=sharing**](https://colab.research.google.com/drive/1HykNzK5ht42hDzCH1z1i-Z1YGewx-V9X?usp=sharing)

**About 2nd Dataset:**

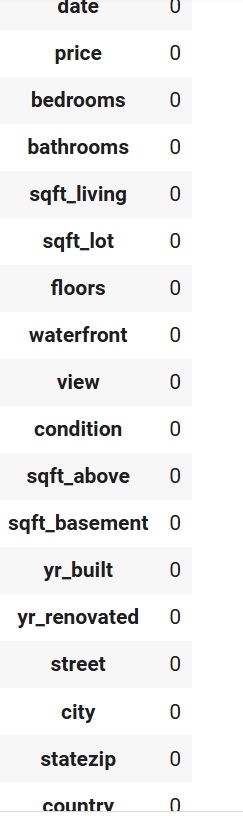
The provided data appears to be a portion of a real estate dataset. Each row represents a property with the following columns:

* **date:** Date the property was listed.
* **price:** The sale price of the property.
* **bedrooms:** Number of bedrooms.
* **bathrooms:** Number of bathrooms.
* **sqft\_living:** Square footage of the living space.
* **sqft\_lot:** Square footage of the lot.
* **floors:** Number of floors.
* **waterfront:** Whether the property has a waterfront view (binary: 0 or 1).
* **view:** View quality of the property (ordinal scale).
* **condition:** Condition of the property (ordinal scale).
* **sqft\_above:** Square footage of living space above ground level.
* **sqft\_basement:** Square footage of basement.
* **yr\_built:** Year the house was built.
* **yr\_renovated:** Year the house was renovated.
* **street:** Street address.
* **city:** City.
* **statezip:** State and ZIP code.
* **country:** Country.

**Code:** df.isnull().sum()

**Description**:  df.isnull():

* Checks each element in the DataFrame for missing values.
* Returns a DataFrame of the same size, with True for missing values and False for non-missing values.
* **Output:**

****

**Code:** import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.preprocessing import StandardScaler

**Description**: This code snippet imports essential libraries for data analysis and machine learning:

1. pandas:
   * A powerful data analysis and manipulation library. It's used for reading, cleaning, and transforming data.
2. sklearn.model\_selection:
   * Provides tools for model selection and evaluation.
     + train\_test\_split: Splits data into training and testing sets for model training and evaluation.
3. sklearn.linear\_model:
   * Contains various linear models, including logistic regression.
     + LogisticRegression: Implements logistic regression for classification tasks.
4. sklearn.preprocessing:
   * Offers preprocessing techniques for data.
     + StandardScaler: Standardizes features by removing the mean and scaling to unit variance. This is often crucial for machine learning algorithms, especially those that assume normally distributed features.

**Code**: import pandas as pd

import numpy as np

import statsmodels.api as sm

**Description:** This code snippet imports essential libraries for data analysis and statistical modeling:

1. pandas:
   * A powerful data analysis and manipulation library. It's used for reading, cleaning, and transforming data.
2. numpy:
   * A fundamental library for numerical computing in Python. It provides efficient array operations, mathematical functions, and 1 random number generation.
3. **statsmodels:**
   * A Python module that provides classes and functions for statistical modeling. It's used for statistical tests, hypothesis testing, and various statistical models, including linear regression, logistic regression, time series analysis, and more.

**Code**: df['date'] = pd.to\_datetime(df['date'])

**Description**: This code snippet converts the 'date' column in a pandas DataFrame df to a datetime data type. This is a crucial step in data analysis and manipulation, as it allows for various operations on dates and times.

**Code:** from statsmodels.tsa.stattools import adfuller

result = adfuller(df['price'])

print('ADF Statistic: %f' % result[0])

print('p-value: %f' % result[1])

print('Critical Values:')

for key, value in result[4].items():

    print('\t%s: %.3f' % (key, value))

# If the series is not stationary, apply differencing or other transformations

if result[1] > 0.05:

    df['price\_diff'] = df['price'].diff(1)

    df['price\_diff'].plot(figsize=(12, 6))

**Description:** The code tests for stationarity in a time series using the Augmented Dickey-Fuller (ADF) test.

1. ADF Test:
   * The adfuller function tests the null hypothesis that the time series has a unit root (non-stationary).
   * A low p-value (< 0.05) indicates rejection of the null hypothesis, suggesting stationarity.
2. Differencing:
   * If the series is non-stationary, differencing is applied to make it stationary.
   * Differencing involves subtracting the previous value from the current value.

**Output:** ADF Statistic: -22.542769

p-value: 0.000000

Critical Values:

1%: -3.432

5%: -2.862

10%: -2.567

**Code:** import pandas as pd

import statsmodels.api as sm

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

# Load the data (assuming you have it in a DataFrame called 'df')

# ...

# Convert 'date' to datetime objects and extract features

df['date'] = pd.to\_datetime(df['date'])

df['year'] = df['date'].dt.year

df['month'] = df['date'].dt.month

df['dayofweek'] = df['date'].dt.dayofweek

# Select features and target variable

features = ['year', 'month', 'dayofweek']

target = 'price'

X = df[features]

y = df[target]

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Scale the features (optional but often recommended)

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Add a constant term to the features (required for statsmodels)

X\_train = sm.add\_constant(X\_train)

X\_test = sm.add\_constant(X\_test)

**Description**: The code prepares a time series dataset for modeling:

1. Converts date to datetime format: Enables time-based analysis.
2. Extracts features: Creates features like year, month, and day of week from the date.
3. Splits data: Divides the data into training and testing sets.
4. Scales features (optional): Standardizes features for better model performance.
5. Adds constant term: Includes an intercept term for statistical models.

**Code**: # Create and train the OLS (Ordinary Least Squares) model

model = sm.OLS(y\_train, X\_train)

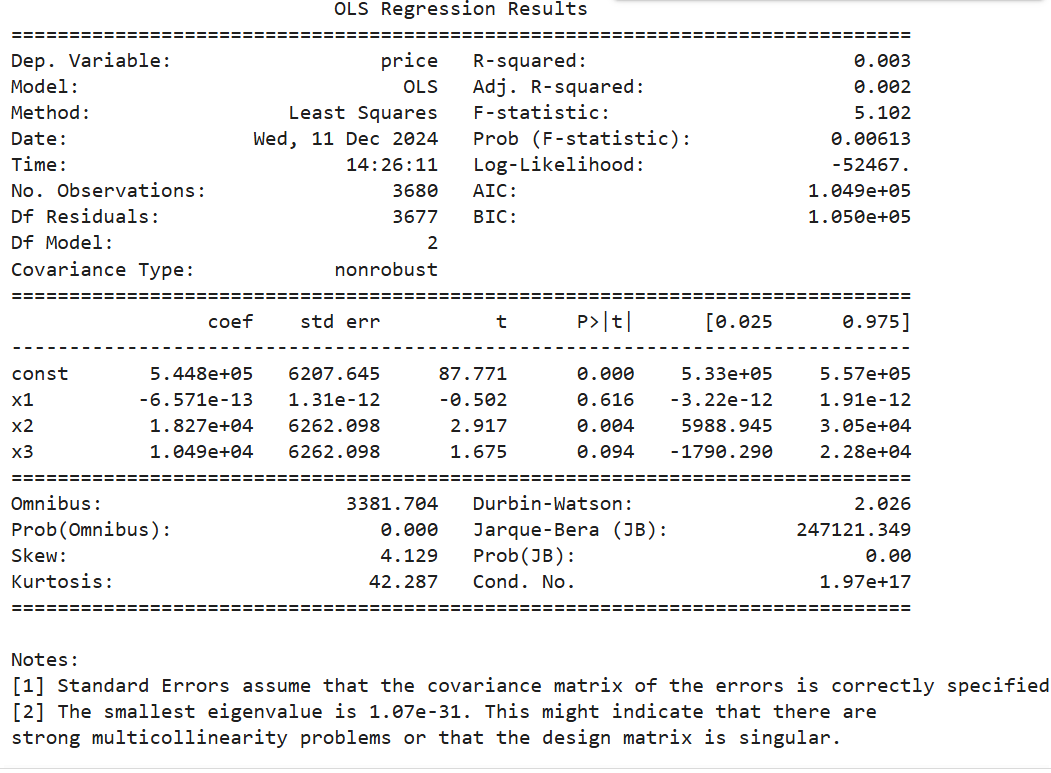
results = model.fit()

# Print model summary

print(results.summary())

**Description**: The code trains a linear regression model using Ordinary Least Squares (OLS) to predict the price based on the given features. The statsmodels.OLS class is used to create the model, and the fit() method is used to train the model on the training data. The summary() method provides a detailed statistical summary of the model, including coefficients, p-values, R-squared, and other relevant statistics.

**Output:**



**Code:** # Create new data for future prediction (only date-based features)

future\_date = pd.to\_datetime('2025-03-15')  # Example date

future\_data = pd.DataFrame({

    'const': [1],  # Constant term

    'year': [future\_date.year],

    'month': [future\_date.month],

    'dayofweek': [future\_date.dayofweek]

},index=[0])

# Make predictions

predicted\_prices = results.predict(future\_data)

print(predicted\_prices)

**Description:** The code creates a new data point with the specified features (year, month, day of week) and uses the trained linear regression model to predict the price for that future date.

**Output:**

0 652083.874014

dtype: float64

**Code**: # ... (previous code for data preparation, model training) ...

# Create new data for future prediction (only date-based features)

future\_date = pd.to\_datetime('2025-03-15')  # Example date

future\_data = pd.DataFrame({

    'const': [1],  # Constant term

    'year': [future\_date.year],

    'month': [future\_date.month],

    'dayofweek': [future\_date.dayofweek]

},index=[0])

# Make predictions

predicted\_price = results.predict(future\_data)[0]  # Get the predicted price value

# Print the prediction with a message

print(f"Predicted house price for {future\_date.strftime('%Y-%m-%d')}: {predicted\_price:,.2f}")

**Description:** The code predicts a future house price using a trained linear regression model. It creates a new data point with the specified date, extracts relevant features, and uses the model to predict the price**.**

**Output:**

Predicted house price for 2025-03-15: 652,083.87

**Colab link:**

[**https://colab.research.google.com/drive/1hsZo695NkO3ztfswhDz3SZzadSyD0Mdn?usp=sharing**](https://colab.research.google.com/drive/1hsZo695NkO3ztfswhDz3SZzadSyD0Mdn?usp=sharing)