**CODE EXPLAINATION**

Let's break down the Python code step by step.

**1. Initial Setup and Data Loading**

Python

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

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

from sklearn.linear\_model import LinearRegression

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

# Load the dataset from CSV

df = pd.read\_csv('/content/Housing (1).csv')

print(df.head())

* **import pandas as pd**: Imports the Pandas library, essential for data manipulation and analysis, especially with DataFrames.
* **import matplotlib.pyplot as plt**: Imports Matplotlib's pyplot module, used for creating static, interactive, and animated visualizations in Python.
* **import seaborn as sns**: Imports Seaborn, a high-level data visualization library based on Matplotlib. It provides a more aesthetically pleasing way to create statistical graphics.
* **from sklearn.model\_selection import train\_test\_split**: Imports train\_test\_split from scikit-learn, a function to divide datasets into training and testing subsets.
* **from sklearn.linear\_model import LinearRegression**: Imports LinearRegression from scikit-learn, the specific model we'll use for prediction.
* **from sklearn.metrics import mean\_squared\_error, r2\_score**: Imports evaluation metrics: mean\_squared\_error (for calculating the average squared difference between actual and predicted values) and r2\_score (for calculating the R-squared value, indicating how well the model explains the variance in the target variable).
* **df = pd.read\_csv('/content/Housing (1).csv')**: This line reads the data from a CSV file named 'Housing (1).csv' into a Pandas DataFrame called df.
* **print(df.head())**: Displays the first few rows of the DataFrame, giving a quick look at the data structure and initial values.

**2. Checking for Missing Values (Output Shown)**

Python

print(df.isnull().sum())

* **print(df.isnull().sum())**: This code checks for missing values (NaN) in each column of the DataFrame. isnull() returns a boolean DataFrame indicating missing values, and sum() counts the number of True values (missing values) for each column. The output (not provided in your prompt for this specific line) would show a list of columns with the count of missing values in each. In a typical scenario, if all counts are 0, it means no missing values.

**3. Data Preprocessing: Handling Categorical Variables**

Python

# Identify categorical columns

categorical\_cols = ['mainroad', 'guestroom', 'basement', 'hotwaterheating',

'airconditioning', 'prefarea', 'furnishingstatus'] # Added 'furnishingstatus' to the list

# Apply one-hot encoding

df = pd.get\_dummies(df, columns=categorical\_cols)

* **categorical\_cols = [...]**: Defines a list of column names that contain categorical data (non-numeric values that represent categories, like 'yes'/'no' or 'furnished'/'unfurnished'). These need to be converted into a numerical format for the linear regression model.
* **df = pd.get\_dummies(df, columns=categorical\_cols)**: This is a crucial step called **one-hot encoding**. For each categorical column, pd.get\_dummies() creates new binary (0 or 1) columns.
  + For example, if 'mainroad' has values 'yes' and 'no', it will create mainroad\_yes and mainroad\_no columns. If 'mainroad' was 'yes', mainroad\_yes would be 1 and mainroad\_no would be 0, and vice-versa.
  + This transforms categorical features into a numerical format suitable for machine learning algorithms.

**4. Separating Features and Target Variable**

Python

# Separate features (X) and target variable (y)

X = df.drop('price', axis=1)

y = df['price']

* **X = df.drop('price', axis=1)**: Creates the feature matrix X. It takes the original DataFrame df and drops the 'price' column (axis=1 indicates dropping a column). All remaining columns in X will be used as input features to predict the price.
* **y = df['price']**: Creates the target variable y, which is simply the 'price' column from the DataFrame. This is what our model will try to predict.

**5. Splitting Data into Training and Testing Sets**

Python

# 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)

* **train\_test\_split(X, y, test\_size=0.2, random\_state=42)**: This function divides the data into four parts:
  + X\_train: Features for training the model.
  + X\_test: Features for evaluating the model (unseen data).
  + y\_train: Target values corresponding to X\_train.
  + y\_test: Target values corresponding to X\_test.
* **test\_size=0.2**: Specifies that 20% of the data will be used for testing, and the remaining 80% for training.
* **random\_state=42**: Ensures reproducibility. If you run the code again with the same random\_state, you'll get the exact same split.

**6. Model Initialization and Training**

Python

# Initialize the model

KKLmodel = LinearRegression() # Note: Your code later uses 'model.fit', implying 'model' was intended here.

# Convert all column names to strings

X\_train.columns = X\_train.columns.astype(str)

X\_test.columns = X\_test.columns.astype(str) # Also convert column names in X\_test

# Fit the model

model.fit(X\_train, y\_train) # Assumes 'model' was defined as KKLmodel or similar

* **KKLmodel = LinearRegression()**: Initializes an instance of the LinearRegression model.
* **X\_train.columns = X\_train.columns.astype(str)** and **X\_test.columns = X\_test.columns.astype(str)**: This is a practical step. Sometimes, after one-hot encoding or other operations, column names might not be simple strings (e.g., they could be integers or mixed types). Scikit-learn models generally prefer string column names, so this line ensures all column names are consistently strings.
* **model.fit(X\_train, y\_train)**: This is the core training step. The fit() method trains the LinearRegression model using the training features (X\_train) and their corresponding target values (y\_train). The model learns the coefficients (weights) for each feature that best predict the housing prices. *(There's a small inconsistency here: you initialized KKLmodel but then used model.fit. Assuming KKLmodel was meant to be model or that model was defined elsewhere.)*

**7. Making Predictions**

Python

# Make predictions

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

* **y\_pred = model.predict(X\_test)**: After training, the predict() method is used to make predictions on the unseen test data (X\_test). The predicted prices are stored in y\_pred.

**8. Evaluating the Model**

Python

# 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}")

* **mse = mean\_squared\_error(y\_test, y\_pred)**: Calculates the Mean Squared Error. This metric measures the average of the squares of the errors (the difference between the actual and predicted values). Lower MSE indicates a better fit.
* **r2 = r2\_score(y\_test, y\_pred)**: Calculates the R-squared value. This metric represents the proportion of variance in the dependent variable that can be predicted from the independent variables. R-squared ranges from 0 to 1, where 1 indicates that the model perfectly predicts the target variable's variance.
* **print(...)**: Displays the calculated MSE and R-squared values.

**Output Interpretation:**

* **Mean Squared Error: 1754318687330.6614**: This is a large number, indicating that the squared differences between actual and predicted prices are substantial. This can happen with large price ranges or when the model doesn't capture all the nuances.
* **R-squared: 0.6529242642153188**: An R-squared of approximately 0.65 means that about 65.29% of the variance in housing prices can be explained by the features included in your model. This is a moderately good fit, but there's still a significant portion of variance (about 34.71%) that the model doesn't account for, suggesting room for improvement.

**9. Visualization: Actual vs. Predicted Prices**

Python

plt.scatter(y\_test, y\_pred)

plt.xlabel("Actual Prices")

plt.ylabel("Predicted Prices")

plt.title("Actual Prices vs. Predicted Prices")

plt.show()

* **plt.scatter(y\_test, y\_pred)**: Creates a scatter plot where the x-axis represents the y\_test (actual prices) and the y-axis represents y\_pred (predicted prices).
* **plt.xlabel(...), plt.ylabel(...), plt.title(...)**: Set the labels for the axes and the title of the plot for clarity.
* **plt.show()**: Displays the generated plot.
* **Interpretation:** In an ideal scenario, the points in this plot would fall perfectly along a 45-degree line, indicating that predicted values exactly match actual values. Deviations from this line show prediction errors.

**10. Visualization: Residual Plot**

Python

residuals = y\_test - y\_pred

plt.scatter(y\_test, residuals)

plt.axhline(y=0, color='red', linestyle='--')

plt.xlabel("Actual Prices")

plt.ylabel("Residuals")

plt.title("Residual Plot")

plt.show()

* **residuals = y\_test - y\_pred**: Calculates the residuals, which are the differences between the actual prices and the predicted prices.
* **plt.scatter(y\_test, residuals)**: Creates a scatter plot of residuals against the actual prices.
* **plt.axhline(y=0, color='red', linestyle='--')**: Draws a horizontal dashed red line at y=0.
* **plt.xlabel(...), plt.ylabel(...), plt.title(...)**: Set labels and title.
* **plt.show()**: Displays the plot.
* **Interpretation:** A good residual plot shows a random scatter of points around the zero line, with no discernible patterns (like a cone shape or a curve). Patterns in residuals suggest that the model might not be capturing certain relationships in the data, indicating that a different model or additional features might be needed.

**11. Attempting Prediction on New Data (Incomplete Code)**

Python

new\_data = [[3, 2, 1500, 4000, 1, 0, 0, 3]]

* **new\_data = [[...]]**: This line defines a list of lists, intended to represent a single new data point for which you want to predict the price.
* **Missing model.predict(new\_data)**: To actually get a prediction for new\_data, you would need to preprocess new\_data in the same way X\_train was preprocessed (e.g., one-hot encode its categorical features, ensure column order matches X\_train), then pass it to model.predict(). The provided code snippet only defines the new\_data but doesn't use the trained model to make a prediction on it.

In summary, this code implements a basic linear regression pipeline: loading data, preprocessing categorical features using one-hot encoding, splitting data, training a linear regression model, evaluating its performance with MSE and R-squared, and visualizing the actual vs. predicted values and residuals. The final line hints at future prediction on new data, though the prediction call itself is missing.