**House Price Prediction**

By

**Khan Mohammad Moin Abdul Moiz (23CO37)**

**Antuley Aman Siraj (23CO25)**



**Department of Computer Engineering**

**Anjuman-I-Islam’s Kalsekar Technical Campus**

**School of Engineering and Technology**

Plot No. 2 & 3, Sector-16, Near Thana Naka, Khanda Gaon,

New Panvel, Navi Mumbai. 410206

**January 2025**

**Aim :-** To develop a Random Forest-based machine learning model for predicting House prices.

**Description :-**

* **Case Study :-** 
  + This case study explores the use of Random Forest Regression to predict house prices in Mumbai, leveraging machine learning to address the city's dynamic real estate market.
  + Using a dataset containing details such as property area, number of BHK, region, and locality, the model was trained to estimate prices accurately.
  + Key preprocessing steps included encoding categorical variables, log-transforming prices for variance stability, and outlier removal.
  + The dataset was standardized and split into training and testing sets, with the Random Forest model demonstrating excellent performance based on metrics like MAE, RMSE, and R² score.
  + The model enables users to input property details and receive predicted prices, offering a practical tool for real estate decision-making while highlighting the effectiveness of machine learning in handling complex, non-linear data.
* **Features :-** 
  + **Area:** The size of the property in square feet.
  + **BHK:** The number of bedrooms, halls, and kitchens in the property.
  + **Region:** The broader geographical area within Mumbai where the property is located (e.g., Thane, Navi Mumbai).
  + **Locality:** The specific neighborhood or locality within the region where the property is located.
  + **Price (Target Feature):** The cost of the property, used as the dependent variable for prediction.
* **Output :-** 
  + The output is the predicted property price based on the given input features.
  + The user provides the following input values:
    - Property Area (in Square Feet)
    - Number of BHK
    - Region
    - Locality
  + Using a trained Random Forest model, these inputs are processed to generate a predicted property price.
  + The predicted price serves as a reliable guide for potential buyers or sellers in the Mumbai real estate market.

**Implementation :-**

# **House Price Prediction Model**

This Jupyter notebook demonstrates a house price prediction model built using the Random Forest Regressor. The goal of this project is to predict house prices based on various features such as area, number of BHK, region, and locality.

The model performs the following steps:

The notebook includes data preprocessing, feature scaling, and model evaluation with metrics like Mean Absolute Error, Root Mean Squared Error, and R² Score.

We will use a dataset containing house prices, and the model will be trained to predict the price of houses based on the given input features.

**Google Drive Integration**

We will begin by mounting Google Drive to access the dataset and save the trained model.

from google.colab import drive

drive.mount(‘/content/drive’)

# **Importing Necessary Libraries**

In this step, we will import the necessary libraries required for the project. These libraries will help in:

* **Data Manipulation**: `pandas` will be used for handling the dataset and performing operations like reading, cleaning, and manipulating data.
* **Numerical Operations**: `numpy` will be used for performing mathematical calculations, such as handling arrays and matrices.
* **Machine Learning**: We will use `RandomForestRegressor` from `sklearn` to create the Random Forest model, and `train\_test\_split` for splitting the dataset into training and testing sets.
* **Data Scaling and Label Encoding**: `StandardScaler` will be used to scale the features, and `LabelEncoder` will encode categorical variables like region and locality.
* **Model Evaluation**: `mean\_absolute\_error`, `mean\_squared\_error`, and `r2\_score` will be used to evaluate the performance of our model.
* **Statistical Operations**: `scipy.stats` will be used for removing outliers from the data.
* **Data Visualization**: `matplotlib.pyplot` will be used for plotting graphs and visualizing the model's performance.
* Let's start by importing these libraries

# Cell 1: Importing necessary libraries

# Importing pandas for data manipulation

import pandas as pd

# Importing numpy for numerical operations

import numpy as np

# Importing train\_test\_split for splitting the dataset into training and testing sets

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

# Importing RandomForestRegressor for the Random Forest model

from sklearn.ensemble import RandomForestRegressor

# Importing StandardScaler for scaling features

from sklearn.preprocessing import StandardScaler, LabelEncoder

# Importing mean\_absolute\_error, mean\_squared\_error, and r2\_score for model evaluation

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

# Importing stats from scipy for statistical operations, used to remove outliers

from scipy import stats

# Importing matplotlib.pyplot for creating visualizations

import matplotlib.pyplot as plt

# **Loading the Data**

In this step, we will load the dataset to examine its contents and structure. The dataset we're working with is a CSV file titled "Mumbai House Prices.csv". We will use the `pandas` library to read the CSV file into a DataFrame.

After loading the data, we will display the first few rows to inspect the dataset and understand its structure, including the available features such as the price, area, region, and locality.

# Cell 2: Load the data

# Reading the 'Mumbai House Prices.csv' file into a DataFrame

data = pd.read\_csv('Mumbai House Prices.csv')

# Display the first few rows of the dataset to inspect its structure

data.head()

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **bhk** | **type** | **locality** | **area** | **price** | **price\_unit** | **region** | **status** | **age** |
| **0** 3 | Apartment | Lak And Hanware The Residency Tower | 685 | 2.50 | Cr | Andheri West | Ready to move | New |
| **1** 2 | Apartment | Radheya Sai Enclave Building No 2 | 640 | 52.51 | L | Naigaon East | Under Construction | New |
| **2** 2 | Apartment | Romell Serene | 610 | 1.73 | Cr | Borivali West | Under Construction | New |
| **3** 2 | Apartment | Soundlines Codename Urban Rainforest | 876 | 59.98 | L | Panvel | Under Construction | New |
| **4** 2 | Apartment | Origin Oriana | 659 | 94.11 | L | Mira Road East | Under Construction | New |
|  |  |  |  |  |  |  |  |  |



# **Data Preprocessing**

In this step, we preprocess the data to make it suitable for modeling.

1. **Price Conversion**: The 'price' in the dataset is recorded in either "Crore (Cr)" or "Lakh (L)" units. We define a function, `convert\_price()`, to convert these price units into actual numerical values.

   - 1 Cr = 10^7

   - 1 L = 10^5

2. **Log Transformation**: Since house prices tend to be skewed, we apply a log transformation to the price values to reduce the skewness. This will help in making the model more stable and effective by ensuring that the target variable follows a more normal distribution.

The transformed prices are stored in a new column, 'log\_price', which will be used as the target variable for training the model.

# Cell 3: Data Preprocessing

# Function to convert price based on the unit (Cr or L)

def convert\_price(price, unit):

    if unit == 'Cr':

        return price \* 10\*\*7  # Convert price from 'Cr' to actual number (1 Cr = 10^7)

    elif unit == 'L':

        return price \* 10\*\*5  # Convert price from 'L' to actual number (1 L = 10^5)

    return price  # Return the price as is if unit is not Cr or L

# Applying the 'convert\_price' function to each row in the DataFrame and creating a new 'price' column

data['price'] = data.apply(lambda row: convert\_price(row['price'], row['price\_unit']), axis=1)

# Applying Log transformation to the price to reduce skewness

data['log\_price'] = np.log1p(data['price'])  # log1p is used for log(1 + x) transformation

# Defining the target variable y as the transformed 'log\_price' column

y = data['log\_price']

# **Handling 'Unknown' Category for Categorical Encoding**

In this step, we handle the categorical variables, specifically for the 'region' and 'locality' columns.

We need to account for the possibility that some categories in these columns might not be present in the training data. To handle this, we do the following:

1. **Adding 'unknown' Category**: We first add an 'unknown' category to both the 'region' and 'locality' columns. This ensures that if an unseen category is encountered during encoding or prediction, it can be safely mapped to this 'unknown' category.

2. **Label Encoding**: Label encoding is applied to the categorical variables to convert them into numerical values. Label encoders are trained on both the existing categories and the newly added 'unknown' category.

By handling the 'unknown' category, we ensure that the model can handle new or missing categories during prediction.

# Cell 4: Handle 'unknown' category for categorical encoding

# Function to ensure 'unknown' category is included in LabelEncoder

def ensure\_unknown\_in\_label\_encoder(label\_encoder, data\_column):

    # Check if 'unknown' is not in the existing classes, if not, add it

    if 'unknown' not in label\_encoder.classes\_:

        label\_encoder.classes\_ = np.append(label\_encoder.classes\_, 'unknown')

    return label\_encoder  # Return the updated label encoder

# Function to add 'unknown' category to the categorical variable before encoding

def add\_unknown\_category(df, column):

    # Convert the column to 'category' type

    df[column] = df[column].astype('category')

    # Add 'unknown' as a new category in the column

    df[column] = df[column].cat.add\_categories('unknown')

    return df  # Return the updated DataFrame

# Updated workflow for encoding categorical variables

# Add 'unknown' category to 'region' column

data = add\_unknown\_category(data, 'region')

# Add 'unknown' category to 'locality' column

data = add\_unknown\_category(data, 'locality')

# Initialize LabelEncoder for 'region'

label\_encoder\_region = LabelEncoder()

# Fit the encoder on the 'region' column (including 'unknown')

label\_encoder\_region.fit(data['region'])

# Ensure 'unknown' category is handled correctly in the encoder

label\_encoder\_region = ensure\_unknown\_in\_label\_encoder(label\_encoder\_region, data['region'])

# Initialize LabelEncoder for 'locality'

label\_encoder\_locality = LabelEncoder()

# Fit the encoder on the 'locality' column (including 'unknown')

label\_encoder\_locality.fit(data['locality'])

# Ensure 'unknown' category is handled correctly in the encoder

label\_encoder\_locality = ensure\_unknown\_in\_label\_encoder(label\_encoder\_locality, data['locality'])

# Transform the 'region' and 'locality' columns using the fitted label encoders

data['region'] = label\_encoder\_region.transform(data['region'])

data['locality'] = label\_encoder\_locality.transform(data['locality'])

# **Feature Engineering: Removing Outliers**

In this step, we perform feature engineering by focusing on relevant features and removing outliers. Here's what happens:

1. **Feature Selection**: We select the features ('area', 'region', 'locality', 'bhk') that are important for predicting the house price. These will be used as input for the model.

2. **Outlier Detection**: We use the **`Z-score`** method to detect outliers in the selected features. A Z-score represents how far a data point is from the mean in terms of standard deviations. A Z-score greater than 3 or less than -3 typically indicates an outlier.

3. **Removing Outliers**: We create a mask to identify rows without outliers (Z-score < 3) and apply this mask to filter out rows with outliers from both the features and the target variable.

This helps ensure that the model is not affected by extreme values that could skew predictions and performance.

# Cell 5: Feature Engineering: Drop irrelevant or less important features

# Select the features ('area', 'region', 'locality', 'bhk') for the model input

X = data[['area', 'region', 'locality', 'bhk']]

# Check for outliers using Z-score method

z\_scores = np.abs(stats.zscore(X))  # Calculate the Z-scores of the features

# Mask to identify rows that do not contain outliers (Z-score < 3 for all columns)

mask = (z\_scores < 3).all(axis=1)  # True for rows without outliers

# Filter out the rows with outliers based on the mask

X\_no\_outliers = X[mask]

# Apply the same mask to the target variable 'y'

y\_no\_outliers = y[mask]

# **Data Splitting: Training and Test Sets**

In this step, we split the dataset into training and testing sets to evaluate the model's performance:

1. **Training Set**: 80% of the data is used for training the model.

2. **Testing Set**: 20% of the data is reserved for testing the model.

This split ensures that the model can be trained on one portion of the data and evaluated on an unseen portion to check its generalization ability. The `train\_test\_split` function from scikit-learn is used, with a random state set to 42 for reproducibility.

# Cell 6: Split the data into training and test sets

# Split the dataset into training (80%) and testing (20%) sets

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

# X\_train and X\_test contain the features (inputs) for the training and testing sets, respectively

# y\_train and y\_test contain the target values (log-transformed price) for the training and testing sets, respectively

# **Feature Scaling: Standardization**

In this step, we scale the features using the **StandardScaler** to standardize the data:

1. **Scaling**: Each feature is scaled to have a mean of 0 and a standard deviation of 1.

2. **Training Set**: The `StandardScaler` is first fitted on the training data, and then the data is transformed to ensure that all features have the same scale.

3. **Test Set**: The same scaler (fitted on the training data) is used to transform the test data, ensuring consistency between the training and test sets.

Standardizing the features helps improve the performance and convergence of many machine learning algorithms.

# Cell 7: Scale the features

# Initialize the StandardScaler to standardize the features

scaler = StandardScaler()

# Fit the scaler to the training data and transform it, scaling each feature to have mean 0 and standard deviation 1

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

# Use the same scaler (fitted on training data) to transform the test data

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

# **Training the Random Forest Model**

In this step, we initialize and train a **RandomForestRegressor** model:

1. **Model Initialization**: We set the number of trees (`n\_estimators`) in the forest to 50 and fix the `random\_state` to ensure reproducibility.

2. **Model Training**: The model is trained on the scaled training data (`X\_train\_scaled`) and the target variable (`y\_train`).

Random forests are an ensemble method that combines multiple decision trees to improve the accuracy and robustness of predictions.

# Cell 8: Train the Random Forest model

# Initialize the RandomForestRegressor with 50 trees and a fixed random seed for reproducibility

rf\_model = RandomForestRegressor(n\_estimators=50, random\_state=42)

# Train the Random Forest model on the scaled training data (X\_train\_scaled) and the target variable (y\_train)

rf\_model.fit(X\_train\_scaled, y\_train)





▾

RandomForestRegressor

i [?](https://scikit-learn.org/1.5/modules/generated/sklearn.ensemble.RandomForestRegressor.html)

RandomForestRegressor(n\_estimators=50, random\_state=42)

# **Model Evaluation**

Once the model has been trained, we evaluate its performance using several metrics to understand how well it generalizes on unseen data:

1. **Predictions**: We use the trained model to predict the target variable (`log\_price`) on the scaled test data (`X\_test\_scaled`).

2. **Evaluation Metrics**:

- **Mean Absolute Error (MAE)**: Measures the average magnitude of the errors in predictions, without considering their direction.

- **Mean Squared Error (MSE)**: Measures the average squared difference between the predicted and actual values, penalizing larger errors.

- **Root Mean Squared Error (RMSE)**: The square root of MSE, providing a measure of error in the same units as the target variable.

- **R² Score**: A statistical measure that indicates how well the model explains the variance in the target variable.

These metrics help assess the accuracy and effectiveness of the model in predicting house prices.

# Cell 9: Evaluate the model

# Predict the target variable (log\_price) on the scaled test data (X\_test\_scaled) using the trained model

y\_pred\_rf = rf\_model.predict(X\_test\_scaled)

# Calculate the Mean Absolute Error (MAE) between actual and predicted values

mae = mean\_absolute\_error(y\_test, y\_pred\_rf)

# Calculate the Mean Squared Error (MSE) between actual and predicted values

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

# Calculate the Root Mean Squared Error (RMSE) by taking the square root of MSE

rmse = np.sqrt(mse)

# Calculate the R² (R-squared) score, a measure of how well the model explains the variance in the data

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

# Print the evaluation metrics to assess model performance

print(f"Random Forest Model Evaluation:")

print(f"Mean Absolute Error: {mae}")

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

print(f"Root Mean Squared Error: {rmse}")

print(f"R² Score: {r2}")

 Random Forest Model Evaluation:

Mean Absolute Error: 0.09942332139194304

Mean Squared Error: 0.03177482977034582

Root Mean Squared Error: 0.17825495721114132

R² Score: 0.950901237171796

# **Cross-validation for Model Evaluation**

Cross-validation is a robust method to evaluate the performance of the model on different subsets of the training data. In this step, we perform 5-fold cross-validation to assess how well the Random Forest model generalizes. The evaluation metric used is the Mean Absolute Error (MAE), which measures the average magnitude of the errors in the predictions.

We will use the `cross\_val\_score` function from `sklearn.model\_selection` to perform the cross-validation and print the results, including the individual MAE scores for each fold, the mean MAE across all folds, and the standard deviation of the MAE.

# Cell 10: Cross-validation for model evaluation

from sklearn.model\_selection import cross\_val\_score

# Perform 5-fold cross-validation on the Random Forest model with the scaled training data

cv\_scores = cross\_val\_score(rf\_model, X\_train\_scaled, y\_train, cv=5, scoring='neg\_mean\_absolute\_error')

# The cross-validation scores are negative because the default scoring for regression is negative, so we need to convert it to positive

cv\_scores = -cv\_scores  # Convert to positive values

# Print the cross-validation results

print(f"Cross-validation MAE scores: {cv\_scores}")

print(f"Mean Cross-validation MAE: {cv\_scores.mean():.2f}")

print(f"Standard Deviation of Cross-validation MAE: {cv\_scores.std():.2f}")

# Cross-validation MAE scores: [0.10732814 0.10566239 0.10515438 0.10508376 0.10667155]

# Mean Cross-validation MAE: 0.11

# Standard Deviation of Cross-validation MAE: 0.00

# **Saving the Model and Components**

After training and evaluating the Random Forest model, it's essential to save the trained model and other necessary components for later use or deployment. This can be done using the `joblib` library, which efficiently handles saving and loading models.

We will save:

1. **Trained Model**: The Random Forest model (`rf\_model`) is saved to a file called `house\_price\_model.pkl`. This allows us to reuse the model without needing to retrain it each time.

2. **Scaler**: The `StandardScaler` used for feature scaling is saved to a file called `scaler.pkl`. This ensures that the same scaling method is applied when making predictions with new data.

3. **Label Encoders**: The label encoders for the `region` and `locality` columns are saved to files (`label\_encoder\_region.pkl` and `label\_encoder\_locality.pkl`). These encoders are needed to transform categorical values when making predictions with new data.

By saving these components, we make the model ready for use in future applications or deployments.

# Cell 11: Save the model

import joblib

# Save the trained Random Forest model to a file called 'house\_price\_model.pkl'

joblib.dump(rf\_model, 'house\_price\_model.pkl')

# Save the scaler (StandardScaler) used for feature scaling to a file called 'scaler.pkl'

joblib.dump(scaler, 'scaler.pkl')

# Save the label encoder for the 'region' column to a file called 'label\_encoder\_region.pkl'

joblib.dump(label\_encoder\_region, 'label\_encoder\_region.pkl')

# Save the label encoder for the 'locality' column to a file called 'label\_encoder\_locality.pkl'

joblib.dump(label\_encoder\_locality, 'label\_encoder\_locality.pkl')

# **Visualizing Actual vs Predicted Prices**

To assess the performance of our Random Forest model, we will compare the actual house prices in the test set with the predicted prices. This is done by plotting a scatter plot of the actual prices against the predicted prices.

The process involves:

1. **Reverse Log Transformation**: Since the target variable (`price`) was log-transformed during model training, we need to reverse this transformation using `np.expm1()` to obtain the actual prices.

2. **Scatter Plot**: We will plot the actual prices on the x-axis and the predicted prices on the y-axis. The plot will include a red dashed line representing perfect predictions where the actual price equals the predicted price.

3. **Interpretation**: A good model will have points clustered closely around the red dashed line, indicating that the predicted prices are close to the actual prices.

This visualization will give us an intuitive sense of how well the model is performing.

# Predict on the test set

y\_pred\_rf = rf\_model.predict(X\_test\_scaled)  # Use the trained Random Forest model to make predictions on the scaled test data

# Reverse the log transformation on the predictions and true values

y\_pred\_rf\_exp = np.expm1(y\_pred\_rf)  # Apply the inverse log transformation (expm1) to the predicted values to get the actual prices

y\_test\_exp = np.expm1(y\_test)  # Apply the inverse log transformation (expm1) to the actual values in the test set to get the actual prices

# Scatter plot of actual vs predicted prices

plt.figure(figsize=(10, 6))  # Create a new figure for plotting with a specified size

plt.scatter(y\_test\_exp, y\_pred\_rf\_exp, color='blue', alpha=0.5)  # Scatter plot of actual prices vs predicted prices, with blue points and 50% transparency

plt.plot([0, max(y\_test\_exp)], [0, max(y\_test\_exp)], color='red', linestyle='--')  # Plot a red dashed line for perfect predictions (where actual = predicted)

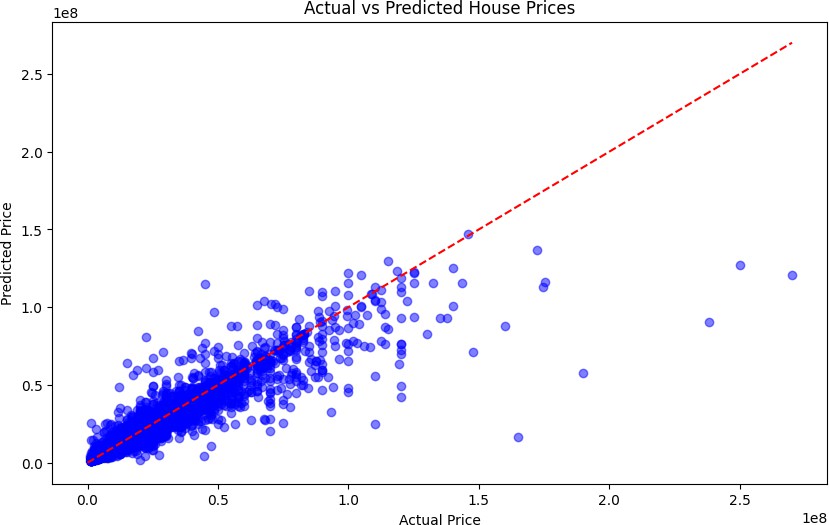
plt.xlabel('Actual Price')  # Label the x-axis as 'Actual Price'

plt.ylabel('Predicted Price')  # Label the y-axis as 'Predicted Price'

plt.title('Actual vs Predicted House Prices')  # Set the title of the plot as 'Actual vs Predicted House Prices'

plt.show()  # Display the plot





# **Comparing Actual vs Predicted Price Distribution**

To further evaluate the performance of our Random Forest model, we will compare the distribution of actual prices with the predicted prices. This is done using histograms, which will allow us to see how closely the predicted prices align with the actual prices.

**Steps:**

1. **Actual Price Distribution**: A histogram of the actual house prices from the test set will be plotted in blue. This shows how the actual prices are distributed.

2. **Predicted Price Distribution**: A histogram of the predicted house prices will be plotted in red. This shows how the predicted prices are distributed.

3. **Comparison**: By comparing the two histograms, we can visually assess how well the model is capturing the actual price distribution. If the predicted price distribution closely matches the actual price distribution, it indicates good model performance.

The histogram will help us understand whether the model's predictions are aligned with the general distribution of prices in the test data.

# Create a new figure with a specified size (10x6 inches)

plt.figure(figsize=(10, 6))

# Plot actual price distribution

plt.hist(y\_test\_exp, bins=50, alpha=0.5, label='Actual Price', color='blue')  # Create a histogram for the actual prices with 50 bins, blue color, and 50% transparency

# Plot predicted price distribution

plt.hist(y\_pred\_rf\_exp, bins=50, alpha=0.5, label='Predicted Price', color='red')  # Create a histogram for the predicted prices with 50 bins, red color, and 50% transparency

# Display the legend to label the two histograms

plt.legend()

# Label the x-axis as 'Price'

plt.xlabel('Price')

# Label the y-axis as 'Frequency'

plt.ylabel('Frequency')

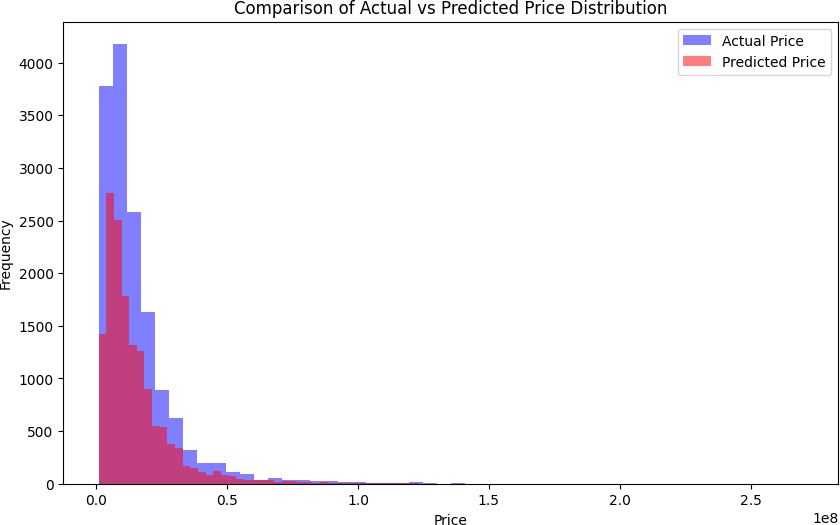
# Set the title of the plot

plt.title('Comparison of Actual vs Predicted Price Distribution')

# Display the plot

plt.show()





# **Learning Curves**

Learning curves are a great way to visualize how well a model performs as the amount of training data increases. By plotting the model's training and cross-validation scores over different training sizes, we can assess the following:

1. **Training Score**: The model's performance on the training set as the size of the training data increases. If the training score is high and stable, it indicates that the model is able to learn well from the data.

2. **Cross-validation Score**: The model's performance on the cross-validation set (held-out data) as the size of the training data increases. This score provides an estimate of how well the model will generalize to unseen data.

**Key Observations:**

- **High training score, low cross-validation score**: This could indicate overfitting, where the model learns the training data too well but performs poorly on unseen data.

- **Similar training and cross-validation scores**: This indicates a well-generalized model that is not overfitting or underfitting.

By analyzing the learning curves, we can determine whether the model is underfitting, overfitting, or performing well on both the training and cross-validation sets.

# Import the learning\_curve function to generate learning curves

from sklearn.model\_selection import learning\_curve

# Generate learning curves

train\_sizes, train\_scores, test\_scores = learning\_curve(rf\_model, X\_train\_scaled, y\_train, cv=5)

# This function computes the learning curve for the model using 5-fold cross-validation

# train\_sizes: Different sizes of the training dataset

# train\_scores: Model's performance on the training set at each training size

# test\_scores: Model's performance on the cross-validation set at each training size

# Create a new figure with a specified size (10x6 inches)

plt.figure(figsize=(10, 6))

# Plot the average training score for each training size

plt.plot(train\_sizes, np.mean(train\_scores, axis=1), label='Training score', color='blue')

# The `np.mean(train\_scores, axis=1)` computes the average score across all cross-validation folds

# Plot the average test score for each training size

plt.plot(train\_sizes, np.mean(test\_scores, axis=1), label='Cross-validation score', color='red')

# The `np.mean(test\_scores, axis=1)` computes the average score across all cross-validation folds

# Label the x-axis as 'Training Size'

plt.xlabel('Training Size')

# Label the y-axis as 'Score'

plt.ylabel('Score')

# Set the title of the plot as 'Learning Curves'

plt.title('Learning Curves')

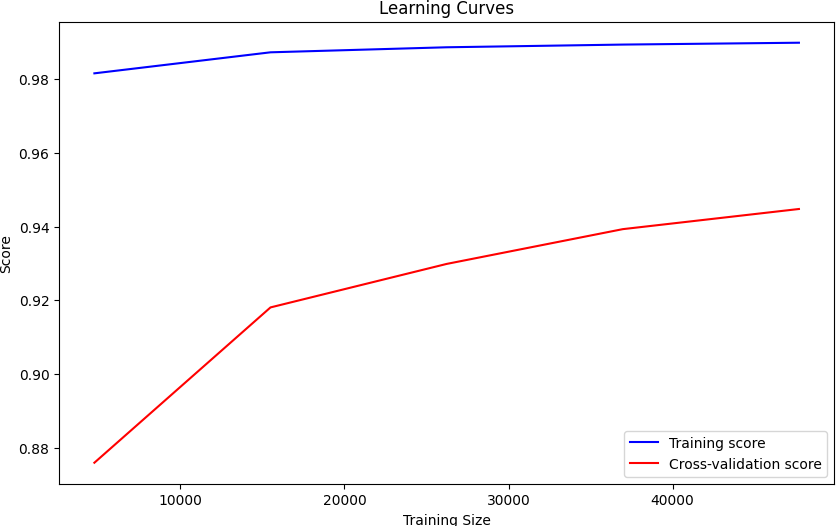
# Display the legend to differentiate between training and cross-validation scores

plt.legend()

# Display the plot

plt.show()





# **Testing the Model with User Input**

In this section, we define a function `test\_model\_with\_user\_input()` that allows the user to input data for a house, such as the area, number of BHK, region, and locality. The model then predicts the price of the house based on this input.

The steps followed are:

1. **Input Collection**: The user is prompted to input the area (in square feet), number of BHK, region, and locality of the house.

2. **Encoding**: The region and locality inputs are encoded using pre-trained label encoders. If the user enters a region or locality not present in the original training data, it is encoded as 'unknown'.

3. **Feature Combination**: The inputs are combined into a DataFrame that matches the feature structure used during model training.

4. **Scaling**: The feature values are scaled using the pre-fitted scaler (`StandardScaler`) that was applied during training, ensuring the input data is processed in the same way as the training data.

5. **Prediction**: The scaled input is passed through the trained Random Forest model to predict the price (in log scale), and the log transformation is reversed to obtain the predicted house price.

6. **Result**: The predicted price is then displayed to the user with proper formatting.

This function helps demonstrate how the trained model can be used to predict house prices based on user inputs.

# Cell 10: Function to test the model with user input

def test\_model\_with\_user\_input():

    # Get user input for area (in sqft), BHK, region, and locality

    area = float(input("Enter the area (in sqft): "))  # Convert input to float for area

    bhk = int(input("Enter the number of BHK: "))  # Convert input to integer for number of BHK

    region = input("Enter the region: ").strip()  # Clean extra spaces from the region input

    locality = input("Enter the locality: ").strip()  # Clean extra spaces from the locality input

    # Preprocess the input region and locality using the label encoders

    # Check if region is in the known classes, if not assign 'unknown'

    if region not in label\_encoder\_region.classes\_:

        region\_encoded = label\_encoder\_region.transform(['unknown'])[0]  # Encode 'unknown'

    else:

        region\_encoded = label\_encoder\_region.transform([region])[0]  # Encode the input region

    # Check if locality is in the known classes, if not assign 'unknown'

    if locality not in label\_encoder\_locality.classes\_:

        locality\_encoded = label\_encoder\_locality.transform(['unknown'])[0]  # Encode 'unknown'

    else:

        locality\_encoded = label\_encoder\_locality.transform([locality])[0]  # Encode the input locality

    # Combine the user input into a DataFrame with the corresponding feature names

    X\_input = pd.DataFrame([[area, region\_encoded, locality\_encoded, bhk]], columns=['area', 'region', 'locality', 'bhk'])

    # Scale the input data using the pre-fitted scaler (scaler from training)

    X\_input\_scaled = scaler.transform(X\_input)  # Apply the same scaling transformation used during training

    # Use the trained Random Forest model to predict the price for the input data

    y\_pred = rf\_model.predict(X\_input\_scaled)

    # Convert the predicted log price back to the original price using the inverse of the log transformation

    predicted\_price = np.expm1(y\_pred[0])  # Reverse the log transformation (exponentiate to get the original value)

    # Print the predicted price to the user

    print(f"Area: {area}")  # Print the input area

    print(f"BHK: {bhk}")  # Print the input number of BHK

    print(f"Region: {region}")  # Print the input region

    print(f"Locality: {locality}")  # Print the input locality

    print(f"Predicted Price: {predicted\_price:,.2f}")  # Format and print the predicted price with commas and 2 decimal places

# Cell 11: Test with user input

test\_model\_with\_user\_input()  # Call the function to test the model with user input

 Area: 400.0

BHK: 1

Region: Thane West

Locality: Majiwada

Predicted Price: 7,240,443.78

**Conclusion :-**

In this project, we developed a machine learning model using a Random Forest Regressor to predict house prices based on key features such as area, BHK (bedrooms), region, and locality. The dataset used contained both numerical and categorical variables, which were processed and transformed appropriately to ensure high model accuracy. A log transformation was applied to the target variable to reduce skewness, and features were scaled for improved model performance.

We implemented an effective data preprocessing pipeline that included handling missing values, converting categorical data using label encoding, and removing outliers to improve the robustness of the model. The model was trained and evaluated using several metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R² score. The Random Forest model showed strong performance, with cross-validation results indicating minimal variance in the model’s error.

Additionally, we tested the model’s performance on user-provided input data, offering real-time predictions. Cross-validation and learning curves were used to further evaluate the model's generalization ability, ensuring that it is capable of making accurate predictions even with unseen data.

In conclusion, the Random Forest model developed in this project is a reliable tool for predicting house prices based on input features. The process from data preprocessing to model evaluation demonstrated the importance of handling data carefully and choosing the right model for regression tasks. Future improvements could include experimenting with other models, such as Gradient Boosting or XGBoost, and incorporating more advanced feature engineering techniques to further enhance prediction accuracy.