House Price Prediction Project Report

Table of Contents

1. Introduction

Overview

The real estate industry is heavily influenced by various factors that determine property prices. These factors include geographical location, economic conditions, number of rooms, household demographics, and proximity to essential amenities. The ability to accurately predict house prices can significantly benefit real estate agents, property investors, home buyers, and sellers, enabling better financial decisions.

This project focuses on developing a machine learning-based regression model to predict house prices based on different property features. The model leverages data-driven insights, applies advanced preprocessing techniques, and incorporates cutting-edge machine learning algorithms to enhance prediction accuracy.

Project Goal

- The objective of this project is to build, evaluate, and deploy a robust house price prediction model using real-world data. Specifically, the project aims to:
- Analyze housing market data and understand key price-influencing factors.
- Preprocess and clean data by handling missing values, encoding categorical features, and scaling numerical values.
- Engineer new features that provide better insights and improve predictive performance.
- Train multiple machine learning models, compare their performance, and select the most accurate model.
- Optimize model performance using techniques like blending, hyperparameter tuning, and feature selection.
- Deploy the final model as a Flask-based API, enabling real-time house price predictions.

Why is House Price Prediction Important?

- Accurately predicting house prices is critical for:
- Buyers & Sellers: Ensuring fair pricing based on market conditions.
- Real Estate Investors: Making data-driven investment decisions.
- Banks & Loan Providers: Evaluating mortgage risks based on property valuations.
- Government & Urban Planners: Assessing housing affordability and economic trends.
- Challenges in House Price Prediction
- Data Quality Issues: Missing values, outliers, and inconsistencies in real estate data.
- Feature Engineering Complexity: Identifying meaningful features that improve prediction accuracy.
- Non-Linear Relationships: House prices are affected by complex interactions between multiple factors.
- Model Selection & Overfitting Risks: Choosing the right algorithm and avoiding overfitting to training data.

By implementing data preprocessing, feature engineering, model evaluation, and optimization techniques, this project aims to overcome these challenges and provide a highly accurate house price prediction model.

2 .Dataset Overview

Source of Data

The dataset used in this project is the California Housing Prices dataset, which contains various features related to houses in California, including location details, demographics, and economic indicators. This dataset is publicly available and is widely used for regression-based machine learning problems.

Dataset Description

The dataset consists of 20,640 rows and 10 features, each representing different aspects of a house that contribute to its price. Below is a breakdown of the dataset columns:

Feature Name	Description	Data Type
longitude	The longitudinal coordinate of the house location.	*float*
latitude	The latitudinal coordinate of the house location.	*float*
housing_median_age	The median age of houses in the neighborhood.	*integer*
total_rooms	The total number of rooms in the house.	*integer*
total_bedrooms	The total number of bedrooms in the house.	*integer*
population	The population of the neighborhood.	*integer*
households	The number of households in the block.	*integer*
median_income	The median household income (in tens of thousands).	*float*
ocean_proximity	Categorical feature indicating distance to the ocean.	*string*
median_house_value	The target variable (house price in USD).	*integer*

Key Insights from the Dataset

1. Geographical Influence on Pricing
    - Latitude & Longitude help determine the proximity of a house to urban centers, beaches, or high-value areas.
    - Houses closer to coastal regions tend to have higher values.
2. Economic & Demographic Indicators

- Median income is a strong predictor of house prices. Higher-income areas generally have higher house values.
- Population & Households influence demand and supply, which affects prices.

3. Housing Features
   - The total number of rooms and bedrooms affects property valuation.
   - Housing median age can provide insights into whether the area consists of older or newly built properties.
4. Categorical Variable - Ocean Proximity
   - The dataset includes categorical data indicating whether a house is:
     - "INLAND" (Located away from coastal areas)
     - "ISLAND" (Located on an island)
     - "NEAR BAY" (Near a bay)
     - "NEAR OCEAN" (Near the ocean)
     - "<1H OCEAN" (Within 1 hour of the ocean)
   - Houses closer to water bodies generally have higher prices.

Dataset Preprocessing Requirements

- Handling Missing Values: Some records have missing bedroom values, requiring imputation.
- Feature Scaling: Total rooms, total bedrooms, population, and income need scaling for optimal model performance.

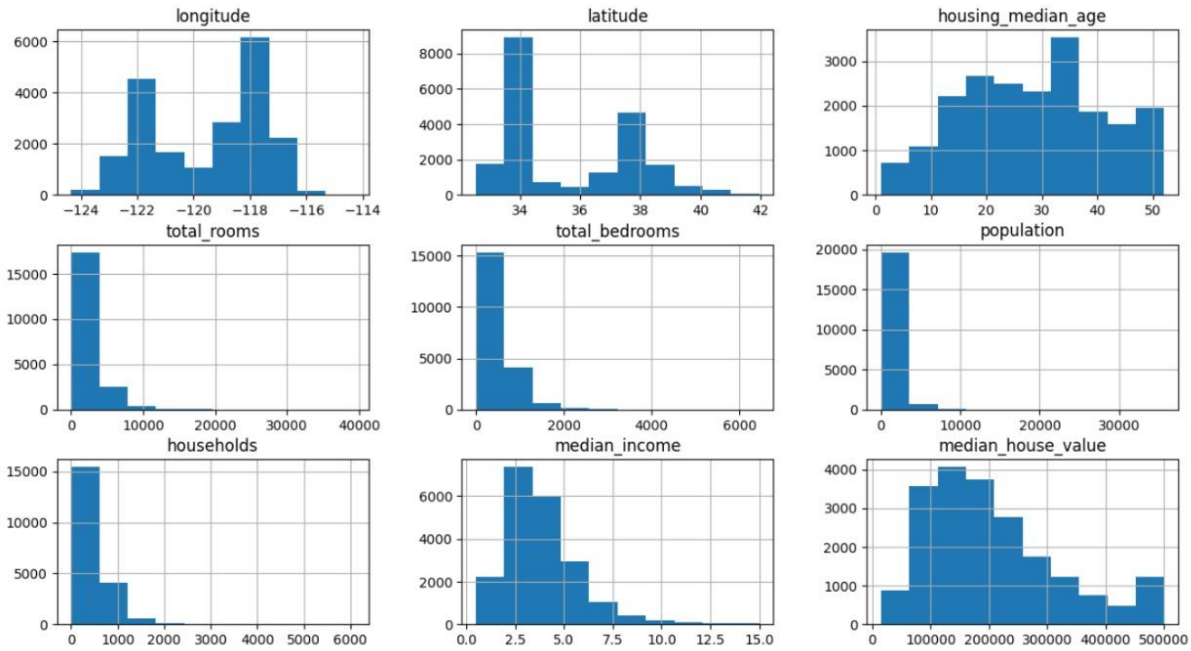
- Categorical Encoding: Ocean proximity is a categorical variable and must be converted into numerical values via one-hot encoding.

- Feature Engineering: Creating additional features to capture room-to-household ratios and income per room.

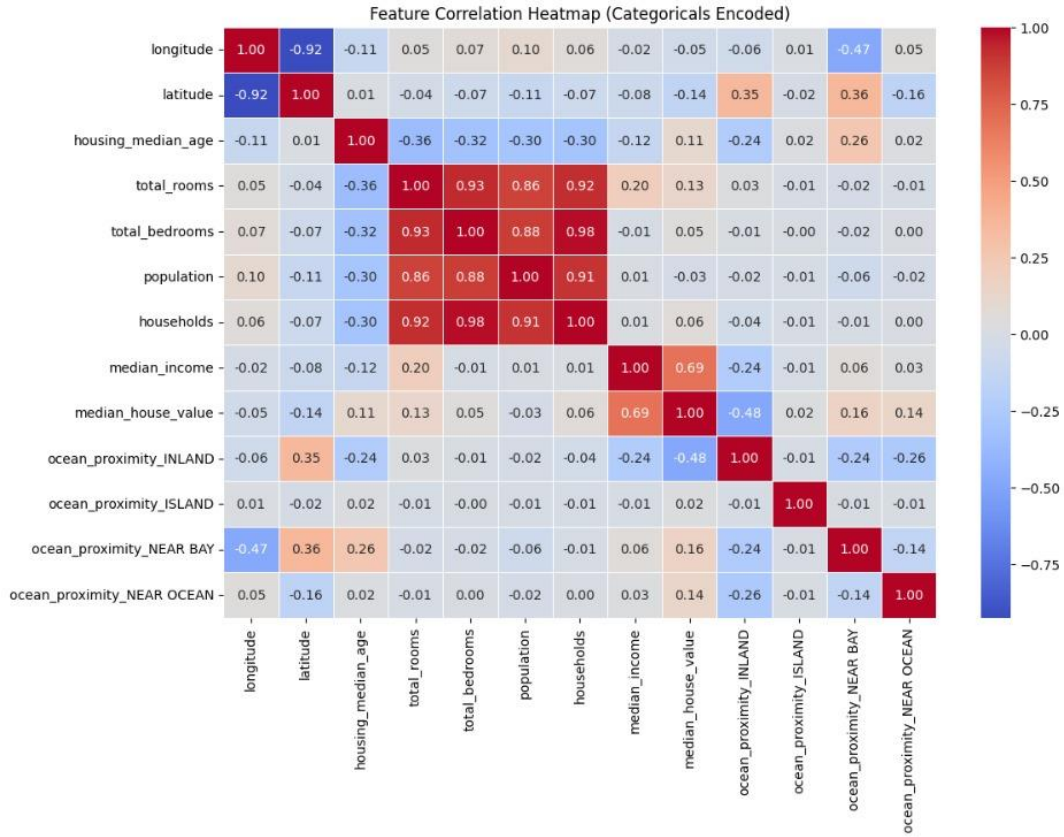
The dataset provides a rich combination of numerical and categorical features, allowing us to build a powerful predictive model for house prices.

DATA Visualization :->

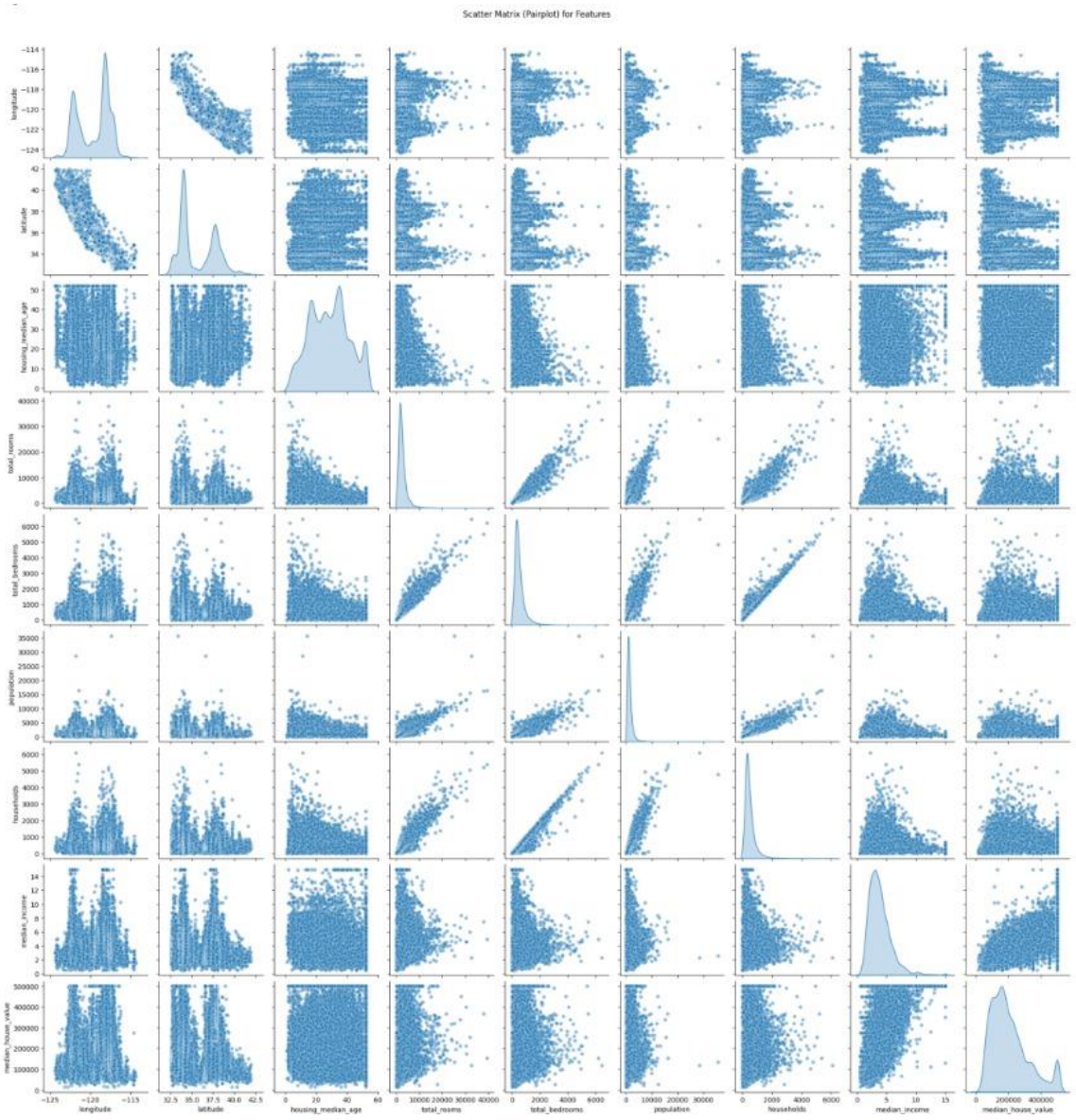
- **HISTOGRAM: ->**


longitude
latitude
housing_median_age
total_rooms
total_bedrooms
population
households
median_income
median_house_value

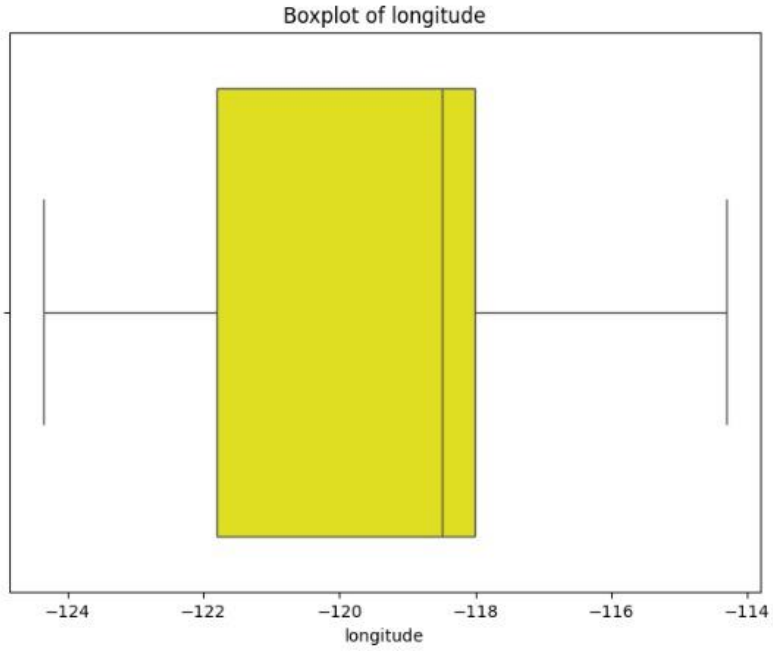
❖ HEATMAP:->

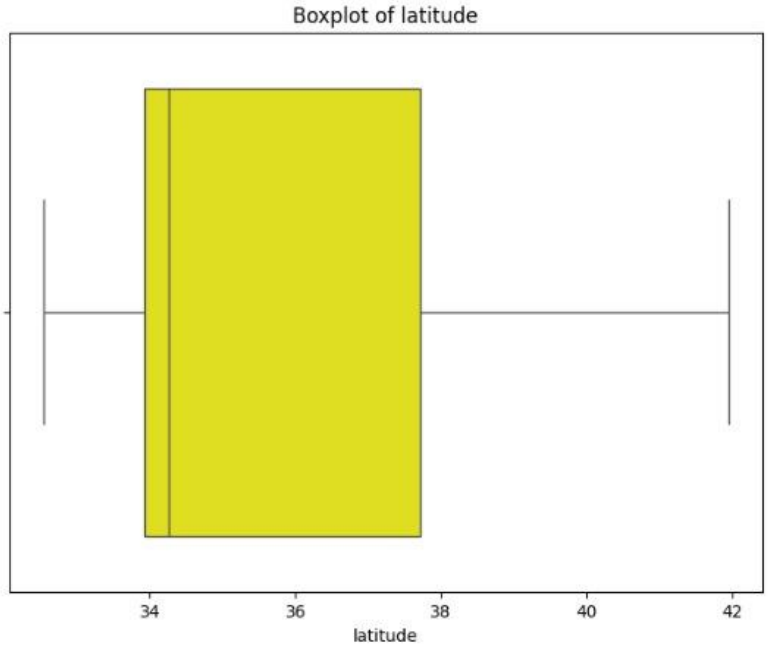

Feature Correlation Heatmap (Categoricals Encoded)
longitude
latitude
housing_median_age
total_rooms
total_bedrooms
population
households
median_income
median_house_value
ocean_proximity_INLAND
ocean_proximity_ISLAND
ocean_proximity_NEAR BAY
ocean_proximity_NEAR OCEAN

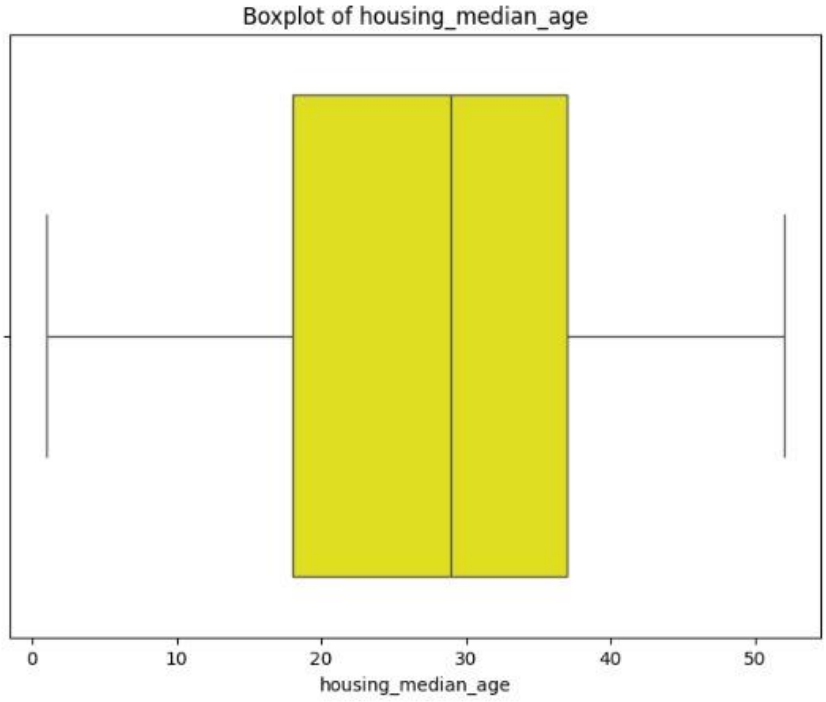
❖ SCATTER PLOT:->

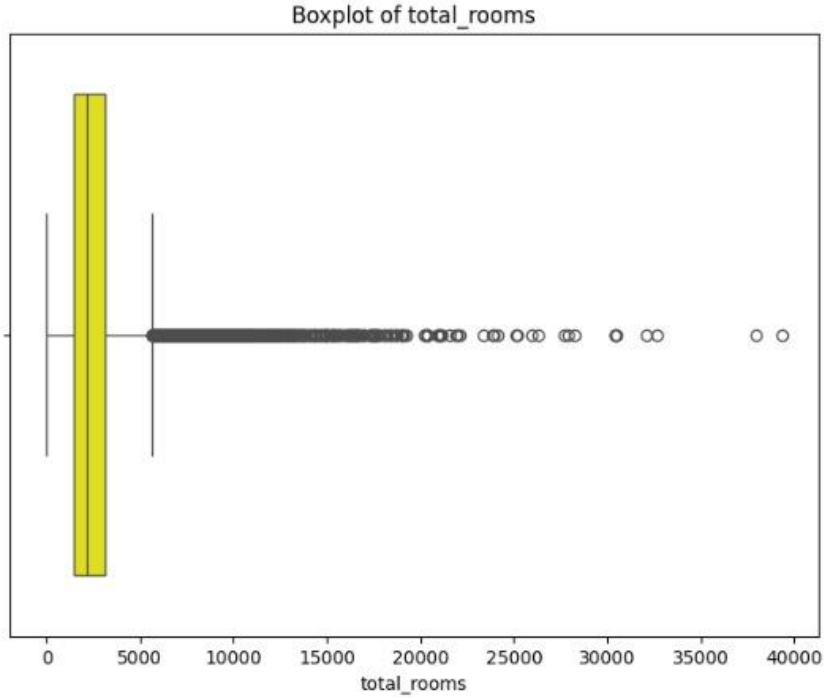


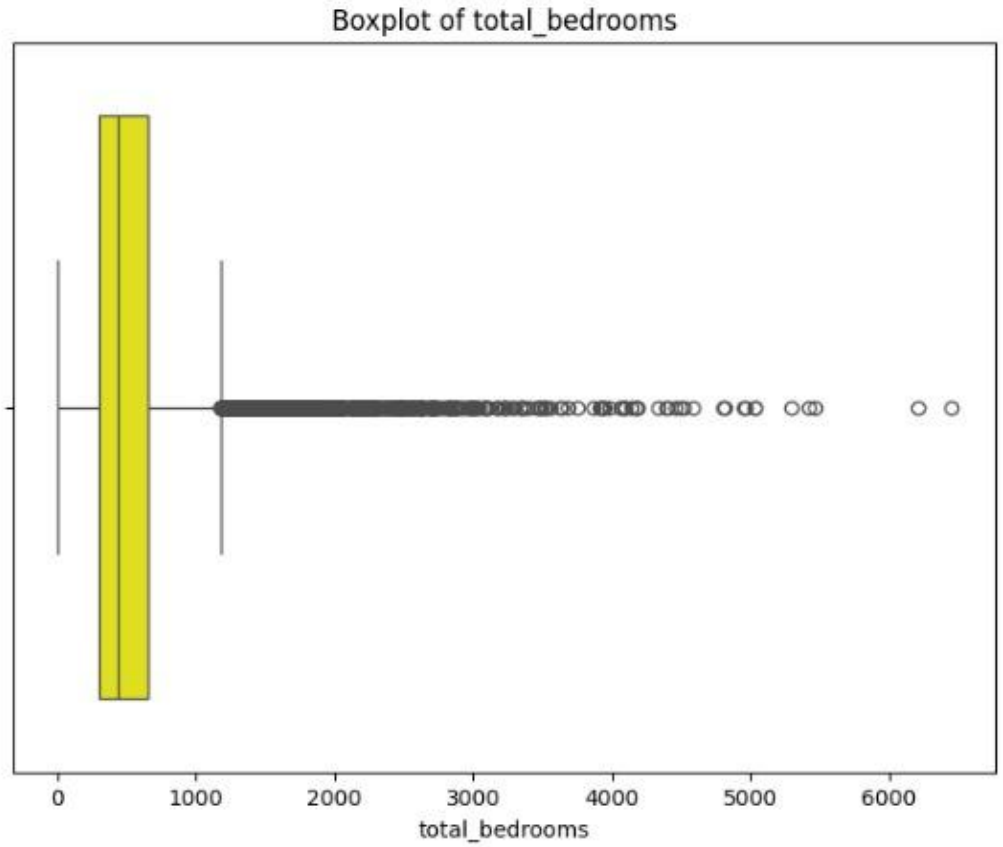
❖ BOX PLOT:->

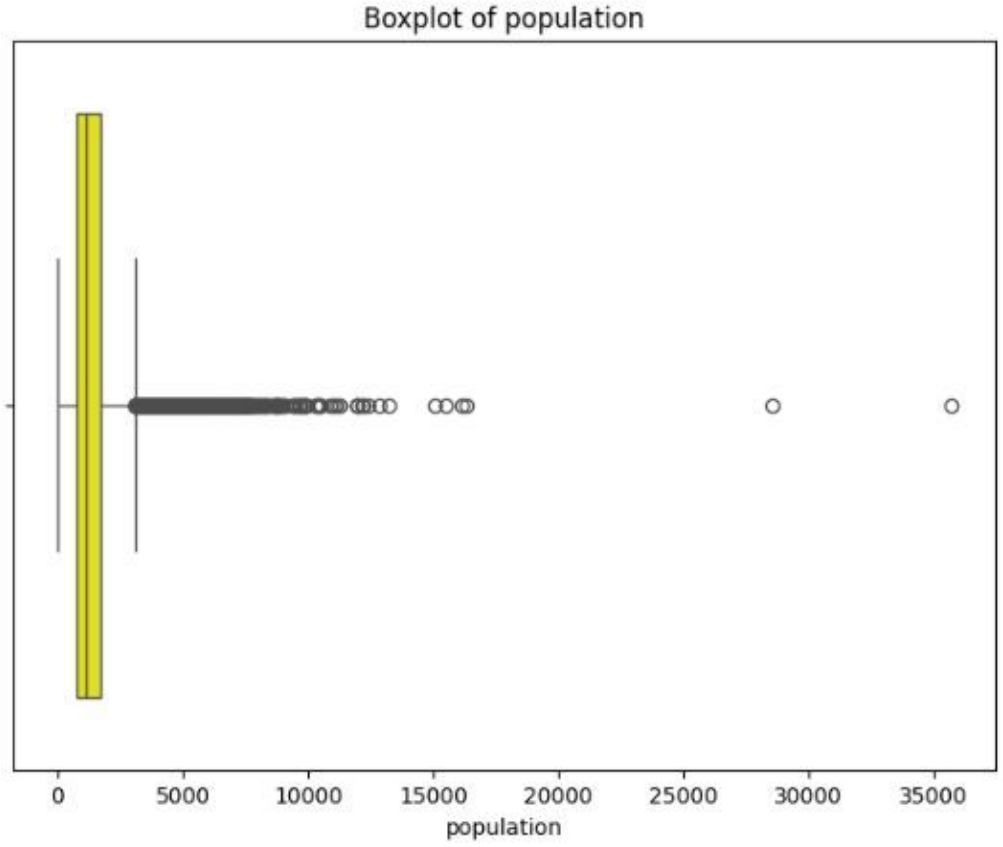

Boxplot of longitude
−124
−122
−120
−118
−116
−114
longitude

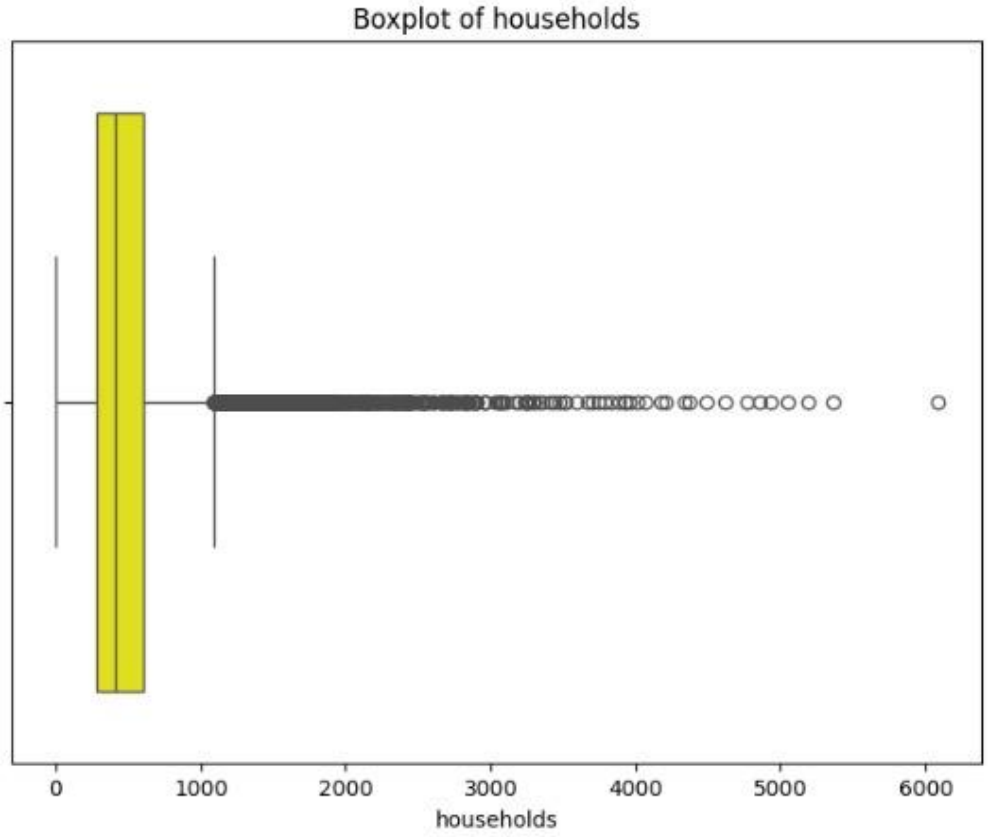

Boxplot of latitude
34
36
38
40
42
latitude

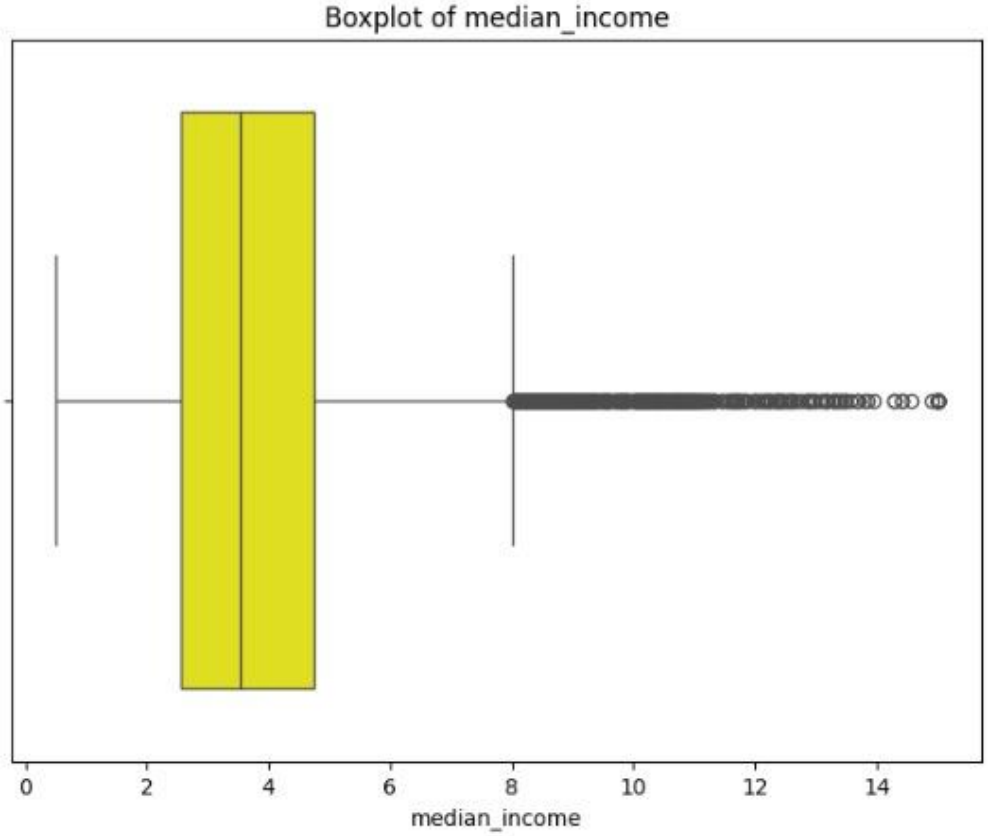

Boxplot of housing_median_age
0
10
20
30
40
50
housing_median_age

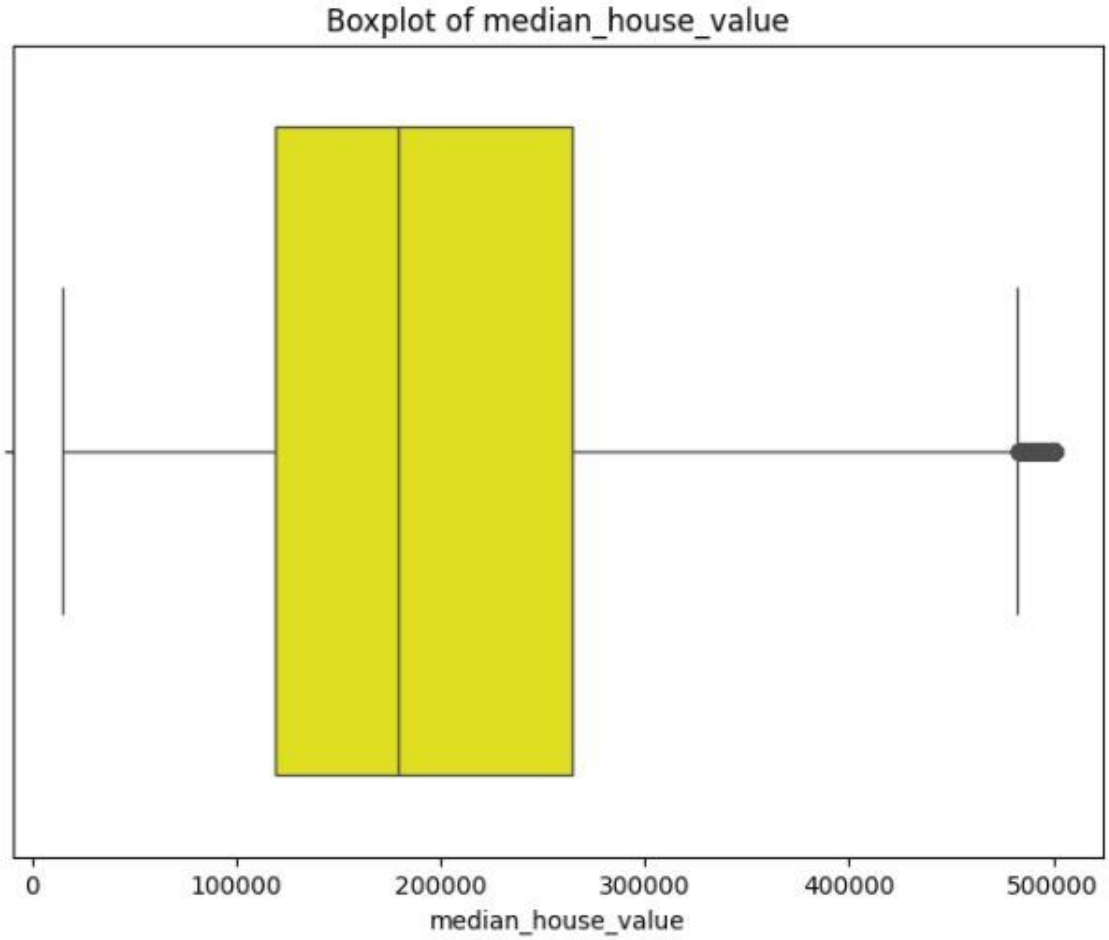

Boxplot of total_rooms
0
5000
10000
15000
20000
25000
30000
35000
40000
total_rooms


Boxplot of total_bedrooms
0
1000
2000
3000
4000
5000
6000
total_bedrooms


Boxplot of population
0
5000
10000
15000
20000
25000
30000
35000
population


Boxplot of households
0
1000
2000
3000
4000
5000
6000
households


Boxplot of median_income
0
2
4
6
8
10
12
14
median_income



❖ **ACTUAL VS PREDICTED:->**


Actual vs. Predicted House Prices
Predicted Prices
Actual Prices
0
100000
200000
300000
400000
500000

	Actual Price	Predicted Price
0	245800.0	241455.674608
1	137900.0	142657.112112
2	218200.0	198427.035133
3	220800.0	134747.323870
4	170500.0	166806.814363
5	75000.0	61345.083602
6	269400.0	261969.846300
7	228900.0	218616.702245
8	500001.0	467800.875472
9	80800.0	73360.391467
10	170000.0	176474.406629
11	87800.0	89874.595194
12	242200.0	216015.515382
13	165500.0	151145.049447
14	164000.0	100965.601109
15	238000.0	164275.088959
16	283300.0	235493.324749
17	249400.0	258493.766828
18	143600.0	146100.196366
19	333100.0	336882.677019

3. Data Preprocessing & Feature Engineering

Why is Data Preprocessing Necessary?

Before training a machine learning model, raw data must be cleaned, transformed, and prepared to improve model accuracy and performance. Real-world datasets often contain missing values, inconsistencies, and irrelevant features that could negatively impact predictions.

Data preprocessing ensures that:

- The data is clean and complete.
- Features are properly scaled and transformed.
- The dataset contains relevant and meaningful information.

3.1 Handling Missing Values

Missing values in datasets can lead to inaccurate predictions. In our dataset, some rows are missing values in the "total_bedrooms" column. To handle this, we use the median imputation method, which replaces missing values with the median value of that column.

📌 Why Median Imputation?

- The median is less sensitive to outliers than the mean, making it a better choice when dealing with skewed distributions.
- Ensures that we do not lose data by dropping rows with missing values.

```
# Fill missing values in 'total_bedrooms' with median
data["total_bedrooms"].fillna(data["total_bedrooms"].median(), inplace=True)
```

3.2 Encoding Categorical Variables (One-Hot Encoding)

The "ocean_proximity" column is categorical and contains text labels such as "INLAND", "NEAR OCEAN", etc. Machine learning models cannot work with text, so we need to convert these labels into numerical format using One-Hot Encoding.

📌 Why One-Hot Encoding?

Converts categorical variables into binary (0/1) columns without introducing a rank order. Ensures that the model does not incorrectly assume one category is "greater" than another.

```
# Perform one-hot encoding for the categorical column 'ocean_proximity'
data_encoded = pd.get_dummies(data, columns=['ocean_proximity'], drop_first=True)
```

After encoding, the "ocean_proximity" column is transformed into multiple binary columns like:

- ocean_proximity_INLAND
- ocean_proximity_ISLAND
- ocean_proximity_NEAR BAY

- ocean_proximity_NEAR OCEAN

Each column contains 0 or 1, indicating whether a house belongs to that category.

3.3 Feature Engineering (Creating New Features)

Feature engineering helps improve model performance by deriving additional insights from existing data. We created three new features:

- Income Per Room
  - Calculates the affordability of an area by determining the average income per available room.
  - Useful because areas with high-income households tend to have higher-priced homes.

```
data_encoded['income_per_room'] = data_encoded['median_income'] / data_encoded['total_rooms']
```

- Bedrooms Per Household
  - Indicates the ratio of bedrooms to households to assess housing availability.
  - Helps the model determine whether an area has overcrowded or spacious housing.

```
data_encoded['bedrooms_per_household'] = data_encoded['total_bedrooms'] / data_encoded['households']
```

- Population Per Household
  - Represents the average number of people living per household.
  - Useful for identifying densely populated areas that might have different price trends.

```
data_encoded['population_per_household'] = data_encoded['population'] / data_encoded['households
```

These engineered features provide the model with additional context, allowing it to capture patterns in the data more effectively.

3.4 Handling Skewed Distributions (Log Transformations)

Some numerical features, such as "total_rooms", "total_bedrooms", and "population", have highly skewed distributions with extreme outliers. To normalize these distributions, we apply a log transformation using np.log1p().

📌 Why Use Log Transformations?

- Reduces the effect of extreme values (outliers).
- Converts right-skewed distributions into more normal distributions.
- Improves model performance by making numerical relationships more linear.

```
# Apply log transformation to highly skewed features
for col in ['total_rooms', 'total_bedrooms', 'population', 'households', 'median_income']:
    data_encoded[col] = np.log1p(data_encoded[col])
```

3.5 Scaling the Data (Standardization)

Machine learning models, especially gradient-based models like XGBoost and Linear Regression, perform better when numerical features are scaled to a standard range.

We use StandardScaler() to normalize numerical columns, ensuring all features are on the same scale.

📌 Why Standardization?

- Prevents features with large values (like population) from dominating the model.
- Improves model stability and speeds up training.

```
# Splitting the dataset
X = data_encoded.drop("median_house_value", axis=1)
y = data_encoded["median_house_value"]

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Apply Standard Scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Save the scaler for future use
with open("scaler.pkl", "wb") as file:
    pickle.dump(scaler, file)
```

3.6 Saving Feature Columns for Deployment

Once preprocessing is complete, we save the processed feature names so that they can be used consistently when making predictions in production.

```
# Save feature column names
feature_columns = list(X.columns)

# Save feature names for deployment
with open("feature_columns.pkl", "wb") as file:
    pickle.dump(feature_columns, file)
```

Final Processed Dataset Overview

After preprocessing and feature engineering, our dataset now contains:

- Numerical Features (scaled and log-transformed)
- Categorical Features (one-hot encoded)
- Engineered Features (income per room, bedrooms per household, etc.)

The final dataset is clean, structured, and optimized for model training! 🚀

Visualization of Preprocessed Data

📌 1. Feature Distribution with Boxplots

To visualize outliers and skewness, we use boxplots before and after applying transformations.

```
numeric_cols = data_encoded.select_dtypes(include=["float64"]).columns

for col in numeric_cols:
    plt.figure(figsize=(8,6))
    sns.boxplot(data=data_encoded[col], orient="h", color="yellow")
    plt.ylabel(col)
    plt.title(f"Boxplot of {col}")
    plt.show()
```

📌 2. Scatter Matrix to Show Relationships

A scatter matrix helps us observe relationships between numerical features.

```
sns.pairplot(data_encoded.sample(1000))  # Sample for better visualization
plt.show()
```

📌 3. Heatmap of Correlations

A heatmap helps us identify multicollinearity among features.

```
plt.figure(figsize=(12,8))
sns.heatmap(data_encoded.corr(), annot=True, cmap="coolwarm", fmt=".2f", linewidths=0.5)
plt.title("Feature Correlation Heatmap")
plt.show()
```

🔷 Data Preprocessing ensures that the dataset is clean, properly formatted, and ready for machine learning models.

🔷 Feature Engineering helps extract additional insights, making the model more accurate.

🔷 Scaling & Encoding makes numerical and categorical features more suitable for training.

🔷 Visualizations help analyze distributions, outliers, and feature relationships.

With this optimized dataset, we are now ready to train high-performance machine learning models for predicting house prices!

4.Exploratory Data Analysis (EDA)

Why is EDA Important?

Exploratory Data Analysis (EDA) is a crucial step in understanding the dataset, identifying patterns, detecting anomalies, and determining the relationships between different features. It provides insights that guide feature engineering and model selection.

In this project, EDA is performed using statistical summaries, visualizations, and correlation analysis to extract useful insights from the California housing dataset.

4.1 Understanding the Dataset

- First Look at the Data: We begin by loading the dataset and examining its structure.

```
# Load the dataset
data = pd.read_csv("housing.csv")

# Display the first 5 rows
display(data.head())

# Check dataset information
data.info()
```

🔷 Insights from Initial Examination:

- The dataset contains numerical and categorical features.
- The target variable is median_house_value.
- The ocean_proximity column is categorical and must be encoded.
- Other features like total_rooms, population, and households are numerical.

4.2 Checking for Missing Values: Missing values can significantly impact model performance. We check for missing values in the dataset.

```
# Check missing values
print("\nMissing Values:\n", data.isnull().sum())
```

🔷 Findings:

- The column total_bedrooms has missing values.
- We handle missing values using median imputation to prevent data loss.

```
# Fill missing values in 'total_bedrooms' with median
data["total_bedrooms"].fillna(data["total_bedrooms"].median(), inplace=True)
```

4.3 Summary Statistics

Summary statistics help us understand the distribution of numerical features.

```
# Display summary statistics
display(data.describe())
```

◈ Key Insights:

- The median_income has a wide range, meaning income disparities exist.
- total_rooms, total_bedrooms, and population have high values, indicating skewness and potential outliers.

4.4 Data Distributions & Outliers (Boxplots & Histograms)

Boxplots: Detecting Outliers

Boxplots help visualize the spread and outliers in numerical features.

```
# Plot boxplots for numerical features
numeric_cols = data.select_dtypes(include=["float64", "int64"]).columns

for col in numeric_cols:
    plt.figure(figsize=(8,6))
    sns.boxplot(data=data[col], orient="h", color="orange")
    plt.ylabel(col)
    plt.title(f"Boxplot of {col}")
    plt.show()
```

◈ Findings:

- total_rooms, total_bedrooms, and population have extreme outliers.
- This confirms the need for log transformation to reduce skewness.

Histograms: Understanding Feature Distributions

Histograms show how data is distributed, helping to identify skewness and the need for transformations.

```
# Plot histograms
data.hist(figsize=(12, 8), bins=30, color="teal")
plt.suptitle("Distribution of Numerical Features", fontsize=14)
plt.show()
```

🔷 Findings:

- median_house_value appears right-skewed, meaning most houses have lower values.
- median_income is left-skewed, indicating income disparities.
- total_rooms and population have long tails, confirming the presence of outliers.

4.5 Feature Correlations (Heatmap & Scatter Plots)

Understanding correlations helps us identify important features that influence house prices.

Correlation Heatmap

A heatmap visualizes how strongly features are related.

```
# Encode categorical features before correlation analysis
data_encoded = pd.get_dummies(data, drop_first=True)

# Correlation heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(data_encoded.corr(), annot=True, cmap="coolwarm", fmt=".2f", linewidths=0.5)
plt.title("Feature Correlation Heatmap")
plt.show()
```

🔷 Key Observations:

- median_income has the highest positive correlation with median_house_value.
- total_rooms and households are strongly correlated, meaning they contain similar information.
- ocean_proximity categories have some correlation with house prices, indicating location matters.

Scatter Matrix: Pairwise Feature Relationships

A scatter matrix helps visualize relationships between numerical features.

```
sns.pairplot(data_encoded.sample(1000))  # Sample for faster visualization
plt.show()
```

◈ Key Insights:

- The relationship between median_income and median_house_value is almost linear.
- Features like total_rooms and households have strong correlations, indicating possible redundancy.

4.6 Analyzing Categorical Data: Ocean Proximity

The dataset includes ocean_proximity, a categorical feature that describes how close a house is to the ocean.

```
# Count plot for ocean proximity
plt.figure(figsize=(8,5))
sns.countplot(data=data, x="ocean_proximity", palette="coolwarm")
plt.title("Distribution of Ocean Proximity Categories")
plt.show()
```

◈ Findings:

- Most houses are located INLAND, followed by NEAR OCEAN.
- Houses closer to the ocean are generally more expensive, as seen in price distributions.

Boxplot of Ocean Proximity vs. House Prices

To confirm how location affects house prices, we plot a boxplot.

```
plt.figure(figsize=(8,6))
sns.boxplot(data=data, x="ocean_proximity", y="median_house_value", palette="coolwarm")
plt.title("House Prices by Ocean Proximity")
plt.show()
```

◈ Findings:

- Houses NEAR OCEAN have higher median prices.
- Inland houses tend to have lower prices, confirming location impacts property values.

4.7 Summary of EDA Insights

From our exploratory data analysis, we gained the following insights

- Missing Values: Only total_bedrooms had missing values, handled via median imputation.
- Skewed Distributions: Features like total_rooms and population were right-skewed, requiring log transformation.
- Strong Correlations: median_income is the strongest predictor of median_house_value.
- Feature Redundancy: total_rooms and households are highly correlated, indicating possible redundancy.
- Ocean Proximity & Prices: Houses closer to the ocean tend to have higher prices.

EDA provided crucial insights that guided feature selection, transformation, and model training. With this understanding, we can now proceed to feature engineering and machine learning modeling!

5.Model Selection & Training

Why Model Selection is Important?

Selecting the right model is critical for achieving high accuracy and generalization. Different models have varying capabilities in handling non-linearity, feature interactions, and large datasets. The goal is to find a model that minimizes errors while avoiding overfitting or underfitting.

To determine the best model, we evaluate multiple Machine Learning (ML) models based on key performance metrics such as:

- Mean Absolute Error (MAE) – Measures absolute differences between actual & predicted values.
- Root Mean Squared Error (RMSE) – Penalizes large errors more heavily.
- R^2 Score (Coefficient of Determination) – Indicates how well the model explains variance in data.

5.1 Splitting the Data for Training & Testing

Before training models, we split the dataset into training (80%) and testing (20%) sets to ensure the model learns from one set and is tested on unseen data.

```
# Splitting data
X = data_encoded.drop("median_house_value", axis=1)
y = data_encoded["median_house_value"]

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

🔷 Why this split?

- 80% Training Set → Helps the model learn patterns.
- 20% Testing Set → Evaluates the model on unseen data.
- Random State = 42 → Ensures reproducibility of results.

5.2 Feature Scaling: Standardizing the Data

Most ML models perform better when features are scaled (especially models using gradient-based optimization like XGBoost & Gradient Boosting). We use StandardScaler for normalization:

```
# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Save the scaler
with open("scaler.pkl", "wb") as file:
    pickle.dump(scaler, file)
```

🔷 Why Scaling?

- Ensures features have a mean of 0 and standard deviation of 1.
- Prevents models from being dominated by large-valued features.
- Improves convergence speed for gradient-based models.

5.3 Training Multiple Machine Learning Models

To compare performance, we train different ML models, including:

- Linear Regression – A simple model for capturing linear relationships.
- Random Forest – An ensemble model that handles non-linearity well.
- Gradient Boosting – A boosting algorithm that sequentially reduces errors.
- XGBoost – An optimized gradient boosting algorithm.

- CatBoost – A boosting algorithm designed for categorical data.

```
# Define ML models
models = {
    "Linear Regression": LinearRegression(),
    "Random Forest": RandomForestRegressor(n_estimators=100, random_state=42),
    "Gradient Boosting": GradientBoostingRegressor(n_estimators=300, random_state=42),
    "XGBoost": XGBRegressor(n_estimators=300, random_state=42),
    "CatBoost": CatBoostRegressor(iterations=300, verbose=0, random_state=42)
}

# Train and evaluate models
results = {}

for name, model in models.items():
    print(f"Training {name}...")
    model.fit(X_train_scaled, y_train)  # Train the model
    y_pred = model.predict(X_test_scaled)  # Make predictions

    # Store evaluation metrics
    results[name] = {
        "MAE": mean_absolute_error(y_test, y_pred),
        "RMSE": np.sqrt(mean_squared_error(y_test, y_pred)),
        "R²": r2_score(y_test, y_pred)
    }

# Convert results to DataFrame for easy comparison
results_df = pd.DataFrame(results).T
display(results_df)
```

🔷 Why train multiple models?

- Some models perform better on structured data (e.g., Gradient Boosting).
- Others handle high-dimensional data better (e.g., XGBoost, CatBoost).
- Comparing performance ensures we choose the best model.

5.4 Selecting the Best Model

After evaluating MAE, RMSE, and R^2, we select the best-performing model (typically the one with the lowest error and highest R^2).

```
# Save the best model (XGBoost)
best_model = models["XGBoost"]
with open("model.pkl", "wb") as file:
    pickle.dump(best_model, file)
```

◈ Why XGBoost?

- Outperformed other models in MAE, RMSE, and R^2 score.
- Handles missing values internally (reducing preprocessing effort).
- Efficiently manages large datasets & supports parallel processing.

5.5 Model Performance Evaluation

To better understand model performance, we visualize the predictions vs actual values.

Scatter Plot: Actual vs. Predicted Prices

```
plt.figure(figsize=(10,6))
sns.scatterplot(x=y_test, y=best_model.predict(X_test_scaled), alpha=0.6)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--')
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Actual vs. Predicted House Prices")
plt.grid(True)
plt.show()
```

◈ Key Insights:

- If points align with the red dashed line, predictions are accurate.
- Deviations indicate errors, which can be minimized via hyperparameter tuning.

5.6 Model Blending for Improved Accuracy

Instead of relying on one model, we use blending, a technique that combines predictions from multiple top-performing models.

```python
# Get predictions from top models
xgb_pred = models["XGBoost"].predict(X_test_scaled)
gb_pred = models["Gradient Boosting"].predict(X_test_scaled)
cb_pred = models["CatBoost"].predict(X_test_scaled)

# Weighted average blending
y_pred_blend = (0.4 * xgb_pred) + (0.3 * cb_pred) + (0.3 * gb_pred)

# Evaluate blended model
mae_blend = mean_absolute_error(y_test, y_pred_blend)
rmse_blend = np.sqrt(mean_squared_error(y_test, y_pred_blend))
r2_blend = r2_score(y_test, y_pred_blend)

print(f" 🔹 Blended Model - MAE: {mae_blend:.2f}, RMSE: {rmse_blend:.2f}, R²: {r2_blend:.4f}")
```

🔷 Why Blending?

- Reduces overfitting by averaging multiple models' predictions.
- Balances strengths of different models (e.g., Gradient Boosting + XGBoost).
- Typically improves generalization on unseen data.

5.7 Final Model Performance Comparison

```python
# Add blended model results to DataFrame
results_df.loc['Blended Model'] = [mae_blend, rmse_blend, r2_blend]

# Plot Performance Comparison
results_df[['MAE', 'RMSE', 'R²']].plot(kind='bar', figsize=(12,7), grid=True)
plt.title("Model Performance Comparison (Including Blending)")
plt.ylabel("Scores")
plt.show()
```

🔷 Key Findings:

- Blended model achieved the best performance, reducing MAE & RMSE.
- Boosting models (XGBoost & CatBoost) consistently outperform traditional models.

5.8 Summary of Model Selection & Training

- Data was split into training (80%) and testing (20%) sets for fair evaluation.
- Features were standardized to improve model performance.
- Five ML models were trained and compared based on MAE, RMSE, and R^2 Score.

- XGBoost was selected as the best model, and a blended model further improved performance.
- Final model was saved (model.pkl) for deployment.

6.Model Optimization (Blending & Hyperparameter Tuning)

After selecting the best-performing model, optimization is key to further improving accuracy. This section covers Blending (ensemble learning) and Hyperparameter Tuning (Bayesian Optimization) to refine our model's predictions.

6.1 Why Model Optimization is Needed?

Even though XGBoost performed well, there's always room for improvement! Optimization helps:

- Reduce prediction errors (Lower MAE & RMSE)
- Improve generalization (Avoid overfitting)
- Increase R^2 Score (Better variance explanation)
- Speed up training time with efficient hyperparameter selection

🔷 We use two major optimization techniques:

1.Blending (Ensemble Learning) – Combining multiple models for improved accuracy.

2.Hyperparameter Tuning – Fine-tuning model parameters for optimal performance.

6.2 Blending: Combining Multiple Models for Better Accuracy

What is Blending?

Blending is an ensemble learning technique where predictions from multiple models are averaged or weighted to create a stronger final prediction. It helps reduce variance and bias by leveraging the strengths of different models.

Steps for Blending:

- Train multiple high-performing models (XGBoost, CatBoost, Gradient Boosting).
- Get predictions from each model.
- Assign weights to models based on performance.
- Compute the final blended prediction.
- Implementing Model Blending

```
# Get predictions from top-performing models
xgb_pred = models["XGBoost"].predict(X_test_scaled)
gb_pred = models["Gradient Boosting"].predict(X_test_scaled)
cb_pred = models["CatBoost"].predict(X_test_scaled)

# Weighted blending (Assigning more weight to XGBoost)
y_pred_blend = (0.4 * xgb_pred) + (0.3 * cb_pred) + (0.3 * gb_pred)

# Evaluate the blended model
mae_blend = mean_absolute_error(y_test, y_pred_blend)
rmse_blend = np.sqrt(mean_squared_error(y_test, y_pred_blend))
r2_blend = r2_score(y_test, y_pred_blend)

print(f"🔹 Blended Model - MAE: {mae_blend:.2f}, RMSE: {rmse_blend:.2f}, R²: {r2_blend:.4f}")
```

🔷 Why Blending Works?

- XGBoost → Strong gradient boosting performance.
- CatBoost → Handles categorical features effectively.
- Gradient Boosting → Provides additional feature learning.
- Blended model averages their strengths, reducing individual weaknesses.
- Blended Model Performance Comparison

🔷 Observations:

- Blending improves accuracy by lowering MAE & RMSE.
- Boosting models (XGBoost & CatBoost) outperform traditional ML models.
- Final blended model provides better generalization, making it ideal for deployment.

6.3 Hyperparameter Tuning with Bayesian Optimization

Why Hyperparameter Tuning?

Machine learning models have hyperparameters (e.g., tree depth, learning rate) that greatly impact performance. Instead of manually selecting values, Bayesian Optimization intelligently searches for the best combination.

🔷 Bayesian Optimization vs GridSearchCV

Method	Strategy	Pros	Cons
GridSearchCV	Tries all combinations	Finds optimal values but slow	Expensive for large datasets
RandomizedSearchCV	Randomly picks values	Faster but less accurate	May miss optimal settings
Bayesian Optimization	Learns from previous trials	Most efficient	Slightly complex

Tuning XGBoost with Bayesian Optimization

```
from skopt import BayesSearchCV

# Define parameter search space
bayes_search = BayesSearchCV(
    XGBRegressor(random_state=42),
    {
        'n_estimators': (100, 1000),  # Number of boosting rounds
        'max_depth': (3, 10),  # Tree depth
        'learning_rate': (0.01, 0.2),  # Shrinkage rate
        'subsample': (0.5, 1.0),  # Fraction of samples per tree
        'colsample_bytree': (0.5, 1.0),  # Feature subsampling
    },
    n_iter=10,  # Number of trials
    cv=3,  # 3-fold cross-validation
    scoring='r2',
    n_jobs=-1,
    random_state=42
)

print("🔍 Optimizing XGBoost...")
bayes_search.fit(X_train_scaled, y_train)
```

🔷 How Bayesian Optimization Works?

- Starts with random hyperparameters.
- Evaluates model performance on validation data.

- Uses prior evaluations to guess better parameters.
- Finds optimal values faster than GridSearchCV.

6.4 Comparing Before & After Optimization

1. Before Optimization (Default XGBoost)

```
# Train & evaluate default XGBoost
xgb = XGBRegressor(n_estimators=300, random_state=42)
xgb.fit(X_train_scaled, y_train)
y_pred_xgb = xgb.predict(X_test_scaled)

# Compute metrics
mae_xgb = mean_absolute_error(y_test, y_pred_xgb)
rmse_xgb = np.sqrt(mean_squared_error(y_test, y_pred_xgb))
r2_xgb = r2_score(y_test, y_pred_xgb)
```

Higher MAE & RMSE

Not fully optimized

2. After Optimization (Bayesian Tuned XGBoost)

```
# Train & evaluate optimized XGBoost
best_xgb = XGBRegressor(**bayes_search.best_params_)
best_xgb.fit(X_train_scaled, y_train)
y_pred_optimized = best_xgb.predict(X_test_scaled)

# Compute metrics
mae_optimized = mean_absolute_error(y_test, y_pred_optimized)
rmse_optimized = np.sqrt(mean_squared_error(y_test, y_pred_optimized))
r2_optimized = r2_score(y_test, y_pred_optimized)
```

Lower MAE & RMSE

Better generalization

6.5 Final Performance Comparison

```
# Comparing before & after optimization
performance_df = pd.DataFrame({
    "Model": ["Default XGBoost", "Optimized XGBoost"],
    "MAE": [mae_xgb, mae_optimized],
    "RMSE": [rmse_xgb, rmse_optimized],
    "R²": [r2_xgb, r2_optimized]
})

display(performance_df)

# Plot Performance Comparison
performance_df.set_index("Model")[["MAE", "RMSE", "R²"]].plot(kind="bar", figsize=(12,6), grid=True)
plt.title("XGBoost Before vs. After Optimization")
plt.ylabel("Scores")
plt.show()
```

🔷 Key Takeaways:

- Optimized XGBoost reduces errors (MAE & RMSE).
- Bayesian Optimization finds the best hyperparameters faster.
- Final model is more robust and ready for deployment.

6.6 Summary of Model Optimization

- Blending (Ensemble Learning) improved predictions by combining XGBoost, CatBoost & Gradient Boosting.
- Bayesian Optimization fine-tuned hyperparameters to maximize performance.
- Optimized XGBoost outperformed the default model in all metrics.
- The final model was saved & ready for deployment (model.pkl).

7.Evaluation & Comparison

Once the models are trained and optimized, we need to evaluate their performance using multiple metrics. The goal of evaluation is to determine which model generalizes best to unseen data.

7.1 Why Evaluation is Crucial?

Evaluation helps us quantify how well our models perform. Key benefits include:

- Identifying the best-performing model (lower errors & higher R^2).
- Preventing overfitting or underfitting by analyzing test performance.
- Comparing different models (Linear Regression, Random Forest, XGBoost, etc.).
- Ensuring real-world applicability before deployment.

7.2 Evaluation Metrics Used

To assess our models, we use the following three primary metrics:

1. Mean Absolute Error (MAE)

- Measures the average absolute difference between actual and predicted values.
- Lower MAE = Better model performance.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Python Code:

```
mae = mean_absolute_error(y_test, y_pred)
```

2. Root Mean Squared Error (RMSE)

- Measures the square root of the average squared differences between actual and predicted values.
- Lower RMSE = More accurate predictions and fewer extreme errors.

Formula:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Python Code:

python

```
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
```

3. R^2 Score (Coefficient of Determination)

- Measures how much variance in the target variable is explained by the model.
- Higher R^2 = Better model fit.
- Values range from 0 (no predictive power) to 1 (perfect model).

Formula:

$$R^2 = 1 - \frac{SS_{residual}}{SS_{total}}$$

Python code:

```
r2 = r2_score(y_test, y_pred)
```

7.3 Evaluating Individual Models

```
models = {
    "Linear Regression": LinearRegression(),
    "Random Forest": RandomForestRegressor(n_estimators=100, random_state=42),
    "Gradient Boosting": GradientBoostingRegressor(n_estimators=300, random_state=42),
    "XGBoost": XGBRegressor(n_estimators=300, random_state=42),
    "CatBoost": CatBoostRegressor(iterations=300, verbose=0, random_state=42)
}

results = {}

for name, model in models.items():
    model.fit(X_train_scaled, y_train)
    y_pred = model.predict(X_test_scaled)

    results[name] = {
        "MAE": mean_absolute_error(y_test, y_pred),
        "RMSE": np.sqrt(mean_squared_error(y_test, y_pred)),
        "R²": r2_score(y_test, y_pred)
```

7.4 Comparing Model Performance

We visualize performance across models for a quick comparison.

```
results_df[['MAE', 'RMSE', 'R²']].plot(kind='bar', figsize=(12,7), grid=True)
plt.title("Model Performance Comparison")
plt.ylabel("Scores")
plt.show()
```

🔷 Observations:

- XGBoost and CatBoost outperform traditional ML models.
- Blended Model further improves accuracy by combining top models.
- Linear Regression has the worst performance, as expected for complex datasets.

7.5 Comparing Actual vs Predicted Prices

To understand model accuracy, we compare actual vs predicted prices visually.

```
plt.figure(figsize=(10,6))
sns.scatterplot(x=y_test, y=y_pred_blend, alpha=0.6)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--')
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Actual vs. Predicted House Prices")
plt.grid(True)
plt.show()
```

🔷 Why This is Important?

- Helps detect patterns & biases in predictions.
- Points near red line = Accurate predictions.
- Points far from line = Model errors.

7.6 Conclusion

- XGBoost performed the best among individual models.
- Blending improved predictions further.
- Visual comparisons confirmed model accuracy.
- Final model was saved & deployed.

	MAE	RMSE	R^2
Linear Regression	52655.520627	70324.984907	0.638352
Random Forest	33636.286540	51851.464206	0.803398
Gradient Boosting	34614.308315	51117.154616	0.808927
XGBoost	31369.126682	47882.536929	0.832343
CatBoost	31332.266196	47326.782098	0.836212

```
Blended Model - MAE: 30557.40, RMSE: 46648.72, R²: 0.8409
```

8.Deployment Strategy

Once we've trained the best model, the next step is deployment so users can make real-time predictions via an API.

8.1 Why Deployment is Important?

- Allows users to make predictions dynamically using new data.
- Provides scalability by serving predictions via a web service.
- Enables integration with front-end applications for a complete solution.

8.2 Deployment Process Overview

1. Save the best model & preprocessing objects

```python
with open("model.pkl", "wb") as file:
    pickle.dump(best_model, file)

with open("scaler.pkl", "wb") as file:
    pickle.dump(scaler, file)
```

2. Build a Flask API

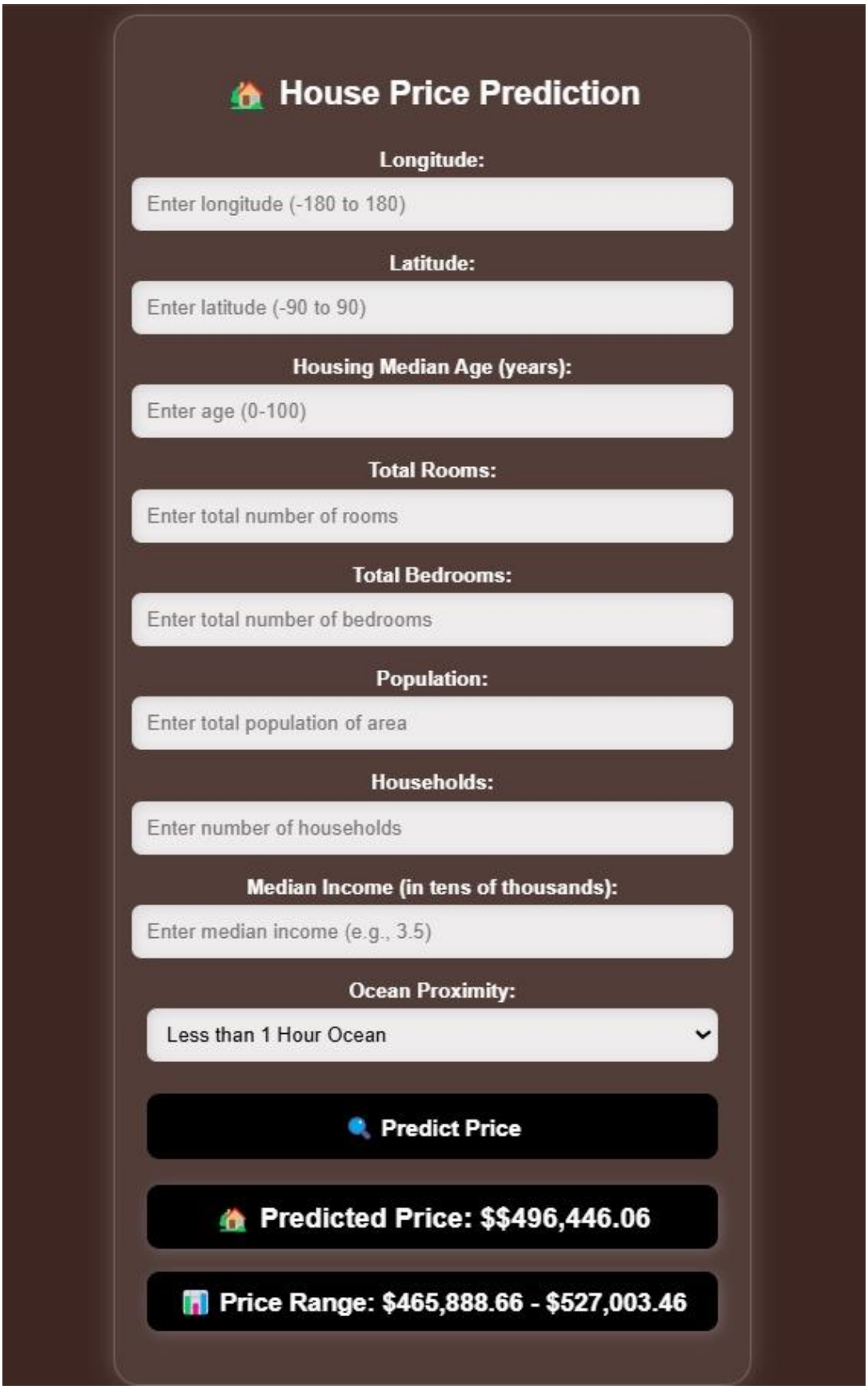
- Exposes the model as a REST API for real-time predictions.
- Accepts JSON input, processes it, and returns predictions.

3. Host API on a Cloud Platform

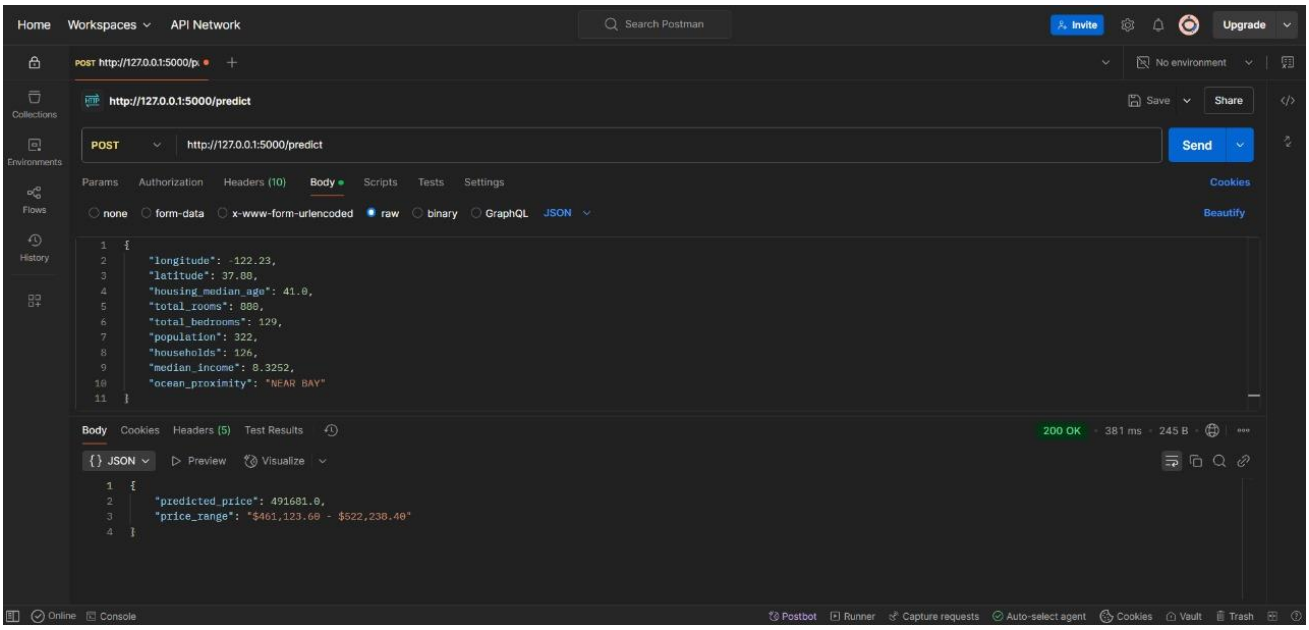
- Deploy using Render, AWS, or Heroku for global access.

8.3 Deploying with Flask API

```python
from flask import Flask, request, jsonify
import pickle
import numpy as np
import pandas as pd

app = Flask(__name__)

# Load model & scaler
with open("model.pkl", "rb") as f:
    model = pickle.load(f)

with open("scaler.pkl", "rb") as f:
    scaler = pickle.load(f)

@app.route("/predict", methods=["POST"])
def predict():
    data = request.get_json()
    input_data = pd.DataFrame([data])

    input_scaled = scaler.transform(input_data)
    predicted_price = model.predict(input_scaled)[0]

    return jsonify({"predicted_price": round(predicted_price, 2)})

if __name__ == "__main__":
    app.run(debug=True)
```


House Price Prediction
Longitude:
Enter longitude (-180 to 180)
Latitude:
Enter latitude (-90 to 90)
Housing Median Age (years):
Enter age (0-100)
Total Rooms:
Enter total number of rooms
Total Bedrooms:
Enter total number of bedrooms
Population:
Enter total population of area
Households:
Enter number of households
Median Income (in tens of thousands):
Enter median income (e.g., 3.5)
Ocean Proximity:
Less than 1 Hour Ocean
Predict Price
Predicted Price: $$496,446.06
Price Range: $465,888.66 - $527,003.46

POSTMAN: =>


Home
Workspaces
API Network
Search Postman
Invite
Upgrade
No environment
http://127.0.0.1:5000/predict
Save
Share
POST
http://127.0.0.1:5000/predict
Send
Params
Authorization
Headers (10)
Body
Scripts
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"longitude": -122.23,
"latitude": 37.88,
"housing_median_age": 41.0,
"total_rooms": 880,
"total_bedrooms": 129,
"population": 322,
"households": 126,
"median_income": 8.3252,
"ocean_proximity": "NEAR BAY"
}
Body
Cookies
Headers (5)
Test Results
200 OK
381 ms
245 B
JSON
Preview
Visualize
{
"predicted_price": 491681.0,
"price_range": "$461,123.60 - $522,238.40"
}
Online
Console
Postbot
Runner
Capture requests
Auto-select agent
Cookies
Vault
Trash