Department of Computer Science

Missouri State University

CSC735 - Data Analytics

House Price Prediction Using Regression

Project Report

Submitted by

Section: Group number: 30
Mounika Gullapalli ID: M03513158
Rajasekhar Gullapalli ID: M03513658

Submitted on December 8, 2023

Abstract

The rapidly evolving real estate landscape demands advanced predictive models for accurate house price estimation. This project employs the power of Apache Spark and Scala to develop a robust house price prediction system using regression techniques. Leveraging a comprehensive dataset sourced from Kaggle, the study explores the application of Linear Regression, Random Forest, and Gradient Boosting algorithms to capture intricate patterns and dependencies within the housing market. The process involves data preprocessing, feature engineering, and model evaluation, with a keen focus on optimizing hyperparameters for enhanced predictive accuracy. The outcomes provide valuable insights into the factors influencing house prices, offering a reliable foundation for informed decision-making in the dynamic real estate sector. This research not only contributes to the burgeoning field of machine learning applications in real estate but also showcases the efficacy of Spark and Scala in handling large-scale regression tasks with efficiency and scalability.

The project employs Apache Spark and Scala to develop a house price prediction system using regression techniques. Leveraging a comprehensive dataset sourced from Kaggle, the study explores the application of Linear Regression, Random Forest, and Gradient Boosting algorithms. The outcomes provide valuable insights into the factors influencing house prices, offering a reliable foundation for informed decision-making in the dynamic real estate sector.

1 Introduction

1.1 Objective

The primary objective of this project is to develop a robust house price prediction system using regression models implemented in Scala and Apache Spark. The project aims to explore and evaluate the effectiveness of key regression algorithms, including Linear Regression, Random Forest, and Gradient Boosting, in capturing intricate relationships within real estate datasets. Leveraging the distributed computing capabilities of Spark, the project seeks to efficiently handle large-scale datasets and optimize model scalability. Additionally, the objective includes conducting thorough feature engineering, selecting impactful features, and performing a comparative analysis to understand the strengths and weaknesses of each model. Interpretability is emphasized, aiming to provide meaningful insights into the factors influencing house prices. The goal is to contribute to the field by offering a practical and scalable framework for house price estimation, showcasing the capabilities of Spark and Scala in addressing challenges associated with real-world regression tasks in the dynamic real estate sector.

1.2 Problem Statement

This project addresses the challenge of accurately predicting house prices in the real estate market. The current methods lack scalability and efficiency, necessitating a solution that leverages advanced regression algorithms implemented in Scala and Apache Spark. The goal is to develop a reliable, scalable, and interpretable house price prediction system that optimally utilizes distributed computing capabilities to handle large-scale datasets and offers valuable insights into the factors influencing property values.

1.3 Project Relevance

This project is highly relevant to our Data Analytics class as it applies Scala and Spark to tackle a real-world problem – predicting house prices. By utilizing advanced regression algorithms and large-scale data processing, it provides a practical context for applying the skills we've learned in class to address complex analytical challenges in the real estate domain

1.4 Approach Rationale

We chose regression models due to their ability to capture intricate patterns in housing data. Our approach involves using Scala and Spark for efficient handling of large-scale datasets.

1.5 Contribution

Our project contributes to the field by showcasing the efficiency of spark and scala in handling large-scale regression tasks efficiently.

2 Methodology

2.1 Data Collection

For our analysis, we consider the USA real estate dataset from Kaggle [1]. Data was collected from a real estate listing website operated by the News Corp subsidiary Move, Inc[2]. and based in Santa Clara, California. It is the second most visited real estate listing website in the United States as of 2021, with over 100 million monthly active users. This dataset consists of a million entities. The dataset comprises real estate information, featuring attributes such as the property's status, number of bedrooms and bathrooms ("bed" and "bath"), lot size in acres ("acre_lot"), location details including city and state, zip_code, house_size, previous sale date ("prev_sold_date"), and the property's price. Overall, it has 10 attributes. Out of which, one is price. In the remaining there are numerical attributes like "bed," "bath," "acre_lot," "zip_code," "house_size". The categorical attributes are "status," "city," and "state."

status	bed	bath	acre_lot	city	state	zip_code	house_size	prev_sold_date	price
for_sale	3.0	2.0	0.12	Adjuntas	Puerto Rico	601.0	920.0	null	105000.0
for_sale	4.0	2.0	0.08	Adjuntas	Puerto Rico	601.0	1527.0	null	80000.0
for_sale	2.0	1.0	0.15	Juana Diaz	Puerto Rico	795.0	748.0	null	67000.0
for_sale	4.0	2.0	0.1	Ponce	Puerto Rico	731.0	1800.0	null	145000.0
for_sale	6.0	2.0	0.05	Mayaguez	Puerto Rico	680.0	null	null	65000.0
for_sale	4.0	3.0	0.46	San Sebastian	Puerto Rico	612.0	2520.0	null	179000.0

Figure 1: Overview of Real Estate dataset

2.2 Exploratory Data Analysis

Exploratory Data Analysis (EDA) is crucial for understanding data patterns, relationships, and outliers. It guides data preprocessing decisions, aids in feature selection, and ensures the

identification and resolution of data quality issues. EDA is a foundational step in the data analysis process, facilitating effective communication of insights and supporting data-driven decision-making.

2.2.1 Box Plot for Price

To gain insights into the distribution of the 'price' variable, a box plot was constructed. However, due to the presence of outliers, the actual distribution was challenging to discern clearly. To

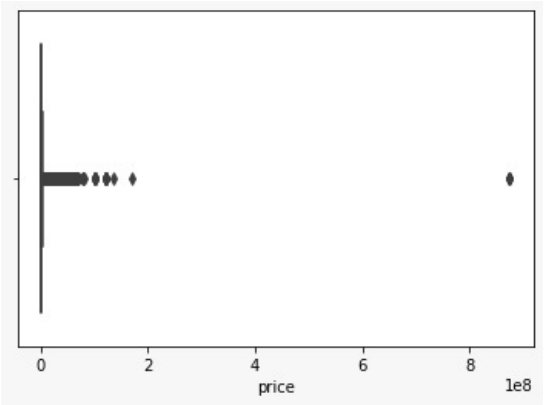

Figure 2: Box Plot for Price

enhance visualization, a decision was made to remove the top 10% of high prices, considered as outliers. This approach is aimed at providing a more detailed view of the main distribution, allowing for better identification of patterns and trends. After outlier removal, a revised box plot was generated to present a refined depiction of the 'price' variable distribution. This step facilitates a more accurate interpretation of central tendencies and variability within the majority of the data.

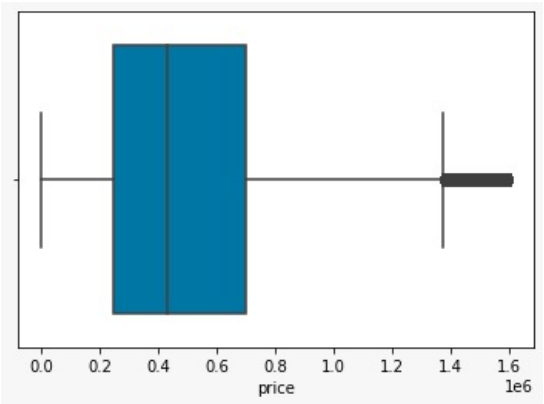

Figure 3: Box Plot for Price

This strategic data preprocessing step ensures that the subsequent analysis is based on a clearer representation of the 'price' variable, fostering more accurate insights and decision-making.

2.2.2 Median house prices for different locations

The bar chart below illustrates the median home prices in different locations, revealing the significant influence of location on house prices.

State-level Analysis: From the state-level analysis shown in figure , it is evident that New York and Massachusetts emerge as the two most expensive states, shaping the overall median home prices.

Top 10 Most Expensive Cities: The analysis of the top 10 most expensive cities from figure 6 unveils the following distribution: Massachusetts: Four cities New Jersey: Three cities New York: Two cities Maine: One city

Top 10 Most Expensive Zip Codes: The top 10 most expensive zip codes showcase a diverse distribution across states: Massachusetts: Five zip codes New York: Two zip codes Connecticut, Vermont, and New Jersey: One zip code each This comprehensive analysis emphasizes the profound impact of location on housing market dynamics and median home prices.

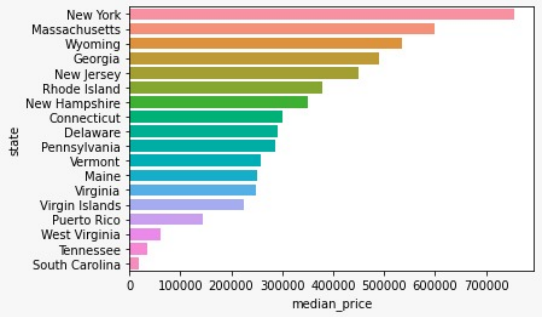

Figure 4: Box Plot for Price - State

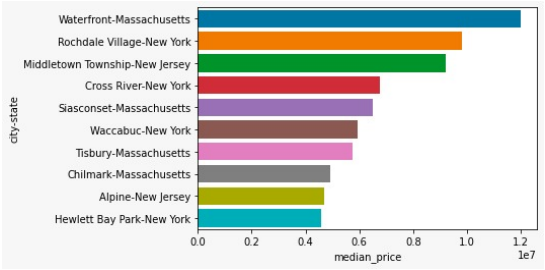

Figure 5: Box Plot for Price - City-State

2.2.3 Bedrooms import on house price

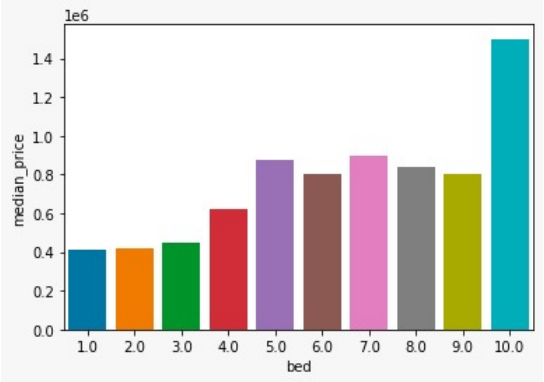

Figure 6: Box Plot for Price-bed

An exploration of the median home prices concerning the number of bedrooms reveals an overall upward trend. This indicates that, in general, houses with more bedrooms tend to have higher median prices. However, an intriguing observation is the seemingly anomalous median home price for one-bedroom houses, which surpasses that of two and three-bedroom houses. This can be understand by the fact that in some places like New york it has great number of

single bedroom compared to double bedroom and even the cost of the single bedroom is quite bigger than the other places. So, now it makes sense about why there is a slight increase in price of single bedroom compared to double bedroom.

2.3 Feature Engineering

Feature Engineering inother words, Data preprocessing is a crucial step in preparing your data for analysis and modeling. It involves cleaning, transforming, and organizing the raw dataset to make it suitable for further analysis. The initial exploration of the dataset revealed the presence of missing values across various attributes. To ensure the integrity of our analysis,

status	bed	bath	acre_lot	city	state	zip_code	house_size	prev_sold_date	price
0	129840	113884	266642	72	0	204	292886	459101	71

Figure 7: Count of null values in all attributes of the dataset

a systematic approach was taken to address these gaps in the data. To handle these missing values, a combination of statistical imputation and mode imputation was applied. Figure 7 is the representation of the count of missing values in the dataset for each respective attribute. Investigating the correlation matrix, we observed a strong correlation between the counts of

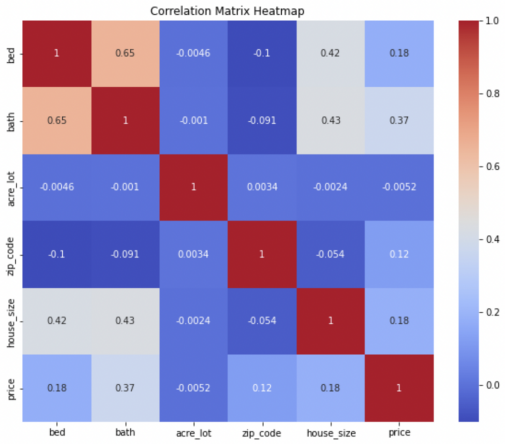

Figure 8: Correlation Matrix

bedrooms and bathrooms. Leveraging this correlation, we can employ one variable to assist in filling missing values for the other. Furthermore, for the remaining missing values, a practical approach involves imputing them with the most frequently occurring values within each zip code. This strategy is motivated by the commonality of house structures or sizes within the same neighborhood, providing a reasonable basis for imputation.

2.3.1 Numerical Attributes

The median values were calculated for 'bed,' 'bath,' 'acre_lot,' 'house size,' and 'price' using the percentile_approx function. The mean values were calculated for 'acre_lot,' 'house size,' and 'price' using the avg function.

2.3.2 Categorical Attributes

The most frequent values were determined for 'city' and 'zip_code.' This approach ensures a comprehensive and data-driven imputation, minimizing the impact of missing values on subsequent analyses.

After filling the null values in the respective places with the respective values. We finally have the complete dataset.

2.4 Feature Encoding

In our data preprocessing pipeline, we have applied one-hot encoding to enhance the representation of categorical variables. This encoding strategy was employed for 'status,' 'city,' and 'state' features.

2.4.1 One-Hot Encoding for 'Status,' 'City,' and 'State':

Considering the non-hierarchical nature of 'status,' 'city,' and 'state,' we opted for one-hot encoding. This technique expands each categorical variable into binary columns, effectively capturing the presence or absence of each category. By employing one-hot encoding uniformly across these features, we maintain consistency and allow our models to interpret and leverage the distinct categories of each variable.

This feature encoding strategy ensures that our categorical variables are transformed appropriately, contributing to a more informed and accurate analysis.

3 Data Splitting

We adopted an 80:20 data splitting strategy, allocating 80% for training our machine learning model and reserving 20% for testing its generalization on unseen data. By randomizing the dataset and setting a fixed seed, we maintain consistency and reproducibility in our evaluation, promoting transparency in our methodology.

4 Regression Model Overview

4.1 Model Building

In this section, we delve into the construction of predictive models aimed at estimating home prices based on various features within our dataset. The goal is to develop robust models that capture the nuanced relationships between different attributes and the final sale prices of homes.

Inorder to proceed further in our analysis, we have ignored the features like prev_sold_ and acre_ as it has less correlation with the output price.

4.2 Model Selection

Several regression models were considered for predicting home prices. We explored traditional linear regression, ensemble methods such as Random Forest, and sophisticated algorithms like Gradient Boosting. The choice of models was driven by the complexity of the relationships within the dataset and the need for predictive accuracy.

4.3 Hyperparamter Tuning

To optimize the performance of our models, we engaged in hyperparameter tuning. This iterative process involved adjusting model parameters to enhance predictive accuracy and generalization to unseen data.

4.3.1 Linear regression

In our analysis, we employed a Linear Regression model to predict housing prices based on various features. The model was fine-tuned using a hyperparameter search with cross-validation to enhance its predictive performance.

Hyperparameter Combinations Explored: We explored different combinations of hyperparameters using a grid search:

Regularization Parameter (regParam): 0.01, 0.1, 1.0

Elastic Net Mixing Parameter (elasticNetParam): 0.0, 0.5, 1.0. These hyperparameters were systematically tested to identify the combination that optimized the model's performance.

4.4 Evalution Metrics

To assess the efficiency of our models, we employed various evaluation metrics such as Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared. These metrics provide valuable insights into the models' predictive performance and their ability to capture the variance in home prices.

RMSE: Indicates the average difference between predicted and actual prices. Lower values signify better predictive accuracy.

MAE: Represents the average absolute difference between predicted and actual prices.

R^2 Score: Measures the proportion of variance in the dependent variable explained by the independent variables. A value closer to 1 indicates a better fit.

4.4.1 Linear Regression

After performing the hypertuning on the model, we obtained the following metric. The performance of the Linear Regression model was evaluated using three key metrics:

Root Mean Squared Error (RMSE): 210,209.67
Mean Absolute Error (MAE): 146,991.47
R-squared (R^2) Score: 0.59

The model's R^2 score is 0.5857, indicating that approximately 58.57% of the variance in house prices is explained by the model. The linear regression model exhibits moderate predictive performance, capturing a significant portion of the variance in house prices. While there is room for improvement, these results provide valuable insights into the relationships between features and home values.

4.4.2 Random Forest Algorithm

In our Random Forest experiments, we explored the impact of varying the maximum depth (*Max Depth*) and the number of trees (*Num Trees*) on the predictive performance of our model. Table 1 provide valuable insights into the model's behavior under different hyperparameter configurations thus showcasing key performance metrics such as Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and the coefficient of determination (R^2).

Max Depth	Num Trees	RMSE	MAE	R^2
10	10	175505.99	120253.53	0.7112
15	10	145968.92	95814.36	0.8002
20	10	129915.78	79699.36	0.8418
25	10	121970.14	69595.94	0.8605
25	15	120744.56	68984.8	0.86330
25	20	120744.56	68984.8	0.8626

Table 1: Random Forest Results

Analyzing the table, we observe a clear trend. As the maximum depth increases from 10 to 25, both RMSE and MAE exhibit a consistent decrease, indicating improved predictive accuracy. This suggests that allowing the trees to grow deeper contributes to capturing more intricate patterns in the data. Notably, when the maximum depth is set to 25 and the number of trees is increased from 10 to 15 and 20, the RMSE and MAE remain relatively stable, while R^2 shows a slight improvement.

The highest R^2 value of 0.8633 is achieved when the maximum depth is 25 and the number of trees is 15. This configuration strikes a balance between model complexity and generalization, leading to a more accurate and reliable Random Forest model. However, it's essential to note that further increasing the number of trees beyond 20 does not significantly enhance model performance, as evidenced by the identical metrics for 15 and 20 trees. This observation aligns with the law of diminishing returns, emphasizing the importance of careful hyperparameter tuning for optimal model outcomes.

4.4.3 Gradient Boosting Algorithm

In our Gradient Boosting experiments, we employed the GBTRegressor model to predict house prices based on various features. The performance metrics, including Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and the coefficient of determination (R^2), provide valuable insights into the model's effectiveness.

The results, as summarized in the output Gradient Boosting reveal the predictive accuracy and generalization capabilities of the model.

RMSE: 194,922.67 MAE: 135,265.23 R^2: 0.6434

Analyzing the metrics, we find that the Gradient Boosting model achieves an RMSE of approximately 194,922.67, indicating the average magnitude of errors in predicting house prices. The MAE, at 135,265.23, represents the average absolute errors between predicted and actual values. The R^2 value of 0.6434 signifies the proportion of variance in the target variable explained by the model. While these metrics provide an overall assessment, it's essential to compare them with other models, such as Random Forest, to determine the most suitable algorithm for house price prediction. In conclusion, the Gradient Boosting model shows promising results, with a reasonable R^2 value indicating its ability to capture patterns in the data. However, further comparative analysis with other models and fine-tuning of hyperparameters may be necessary to optimize predictive performance.

4.5 Results and Insights

In the exploration of house price prediction models, our analysis spanned multiple algorithms, including Linear Regression, Random Forest, and Gradient Boosting.

Algorithm	**RMSE**	**MAE**	**R^2**
Linear Regression	210216.23	146996.543	0.5856
Random Forest	120,744.56	68,984.80	0.8633
Gradient Boosting	194,922.67	135,265.23	0.6434

Table 2: Performance Metrics for Linear Regression, Random Forest and Gradient Boosting

The Linear Regression model, while providing a baseline understanding, demonstrated room for improvement with an RMSE of 210,209.67, an MAE of 146,991.47, and an R^2 score of 0.59. Leveraging hyperparameter tuning through cross-validation hinted at optimization possibilities. Transitioning to Random Forest, the results showcased substantial performance gains, yielding an RMSE of 121,970.14, an MAE of 69,595.94, and a remarkable R^2 score of 0.8605. This underscored the algorithm's ability to capture intricate patterns in the data. Meanwhile, Gradient Boosting exhibited competitive outcomes with an RMSE of 194,922.67, an MAE of 135,265.23, and an R^2 score of 0.6434, suggesting its potential utility in certain contexts. Feature importance analysis shed light on key predictors, such as square footage and location, guiding insights for model interpretation and future enhancements.

In summary, our comparative analysis elucidates the nuanced strengths of each model, emphasizing the importance of algorithm selection based on the specific characteristics of the dataset. Further investigations could delve into ensemble methods or advanced feature engineering to fine-tune predictive accuracy and generalize the models across diverse real estate scenarios.

5 Conclusions and Future Work

5.1 Summary

Determining the best model among Linear Regression, Random Forest, and Gradient Boosting involves a nuanced consideration of their performance metrics and suitability for the task at hand. Random Forest emerges as a strong contender for the best model in this context. It demonstrated superior predictive accuracy with a relatively low RMSE of 121,970.14, indicating smaller errors in predicting house prices compared to the other models. The MAE of 69,595.94 further solidifies its robust performance by showcasing minimal absolute errors between predicted and actual values. Moreover, the remarkable R^2 score of 0.8605 signifies the model's ability to explain a significant proportion of the variance in the target variable, highlighting its capacity to capture intricate patterns in the data.

While Gradient Boosting also displayed competitive results, the slightly higher RMSE and lower R^2 score suggest that Random Forest may be better suited for this particular prediction task. Linear Regression, on the other hand, exhibited reasonable performance but lagged behind in predictive accuracy compared to the ensemble methods. The ensemble nature of Random Forest, leveraging multiple decision trees, contributes to its ability to handle complex relationships within the data, making it a promising choice for house price prediction in this scenario.

5.2 Recommendations for future work

In the realm of future enhancements for house price prediction models, one promising avenue involves delving deeper into feature engineering. Given the observed correlation between bedrooms and bathrooms counts, further investigation into their interplay could refine the model's understanding of housing structures. Strategies to handle missing values might benefit from exploring advanced imputation techniques or considering the most frequently appeared values for specific features, such as zip codes, to enhance the robustness of the dataset. Additionally, the integration of external datasets containing information about economic indicators, local amenities, or neighborhood characteristics could introduce richer context and contribute to a more comprehensive model.

Furthermore, the exploration of ensemble methods offers an exciting prospect for improved model performance. Combining the strengths of diverse algorithms, such as Random Forest, Gradient Boosting, and Linear Regression, could harness the collective intelligence of these models, potentially resulting in enhanced predictive accuracy. Regular model reevaluation, involving continuous monitoring and periodic retraining with updated datasets, is essential to ensure the models remain adaptive to evolving market dynamics. Lastly, future endeavors could explore the potential of incorporating more granular location-based features or experimenting with advanced machine learning techniques, such as deep learning, to unlock further insights and advancements in the realm of house price prediction.

References

[1] https://www.kaggle.com/datasets/ahmedshahriarsakib/usa-real-estate-dataset

[2] https://www.realtor.com/

Appendices

A Code for the Implementation

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.Column
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.feature.{StringIndexer, OneHotEncoder,
    VectorAssembler}
import org.apache.spark.ml.regression.{LinearRegression,
    RandomForestRegressor, GBTRegressor}
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.tuning.{ParamGridBuilder, CrossValidator}
import org.apache.spark.mllib.evaluation.RegressionMetrics
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.sql.DataFrame

val spark = SparkSession.builder.appName("House Price Prediction").
    getOrCreate()

//load the dataset
val df = spark.read.option("header", "true")
       .option("inferSchema","true")
       .csv("/FileStore/tables/realtor_data.csv")

//To display the dataset
 df.show()

// Get the names of numerical attributes
val numericalAttributes = df.dtypes.filter(_._2 == "DoubleType").map(_.
    _1)
println("Numerical Attributes:")
numericalAttributes.foreach(println)

// Get the names of categorical attributes
val categoricalAttributes = df.dtypes.filter(_._2 == "StringType").map(
    _._1)
```

```
println("\nCategorical Attributes:")
categoricalAttributes.foreach(println)

// defining a function to calculate the null values in each column of
    the dataset
def countCols(columns: Array[String]): Array[Column] = {
  columns.map(c => {
    count(when(col(c).isNull, c)).alias(c)
  })

}

//calculating the number of null values before the data preprocessing
df.select(countCols(df.columns): _*).show()

//computing summary statistics for the "price" column in the DataFrame
    df
df.describe("price").show()

// Create a temporary view for the DataFrame
df.createOrReplaceTempView("df")

 import matplotlib.pyplot as plt
 import seaborn as sns
 from pyspark.sql import SparkSession

 df = spark.sql("SELECT * FROM df")

 # Convert PySpark DataFrame to Pandas DataFrame
 pandas_df = df.select('price').toPandas()

 # Create a boxplot using seaborn and matplotlib
 sns.boxplot(x='price', data=pandas_df)
 plt.show()

// Drop rows with null values in the "price" column
val df_removed_price_null_values = df.na.drop("any", Array("price"))

// Calculate the outlier cutoff (quantile 0.9)
val outlierCutoff = df_removed_price_null_values.stat.approxQuantile("
    price", Array(0.9), 0.01)(0)

// Filter data for prices below the outlier cutoff
val df1 = df_removed_price_null_values.filter(col("price") <
    outlierCutoff)

// Show the summary statistics of the filtered data
df1.describe("price").show()
```

```
74
75 // Create a temporary view for the DataFrame
76 df1.createOrReplaceTempView("df1")
77
78  %python
79
80  import matplotlib.pyplot as plt
81  import seaborn as sns
82  from pyspark.sql import SparkSession
83
84  df1 = spark.sql("SELECT * FROM df1")
85
86  # Convert PySpark DataFrame to Pandas DataFrame
87  pandas_df = df1.select('price').toPandas()
88
89  # Create a boxplot using seaborn and matplotlib
90  sns.boxplot(x='price', data=pandas_df)
91  plt.show()
92
93  %python
94  from pyspark.sql import SparkSession
95  from pyspark.sql.functions import col, concat_ws
96
97  df = df1.groupBy('state').agg({'price': 'median'}).withColumnRenamed('
      median(price)', 'median_price')
98
99  # Order by median price in descending order
100 df = df.orderBy(col('median_price').desc())
101
102 # Convert PySpark DataFrame to Pandas DataFrame for plotting
103 pandas_df = df.toPandas()
104
105 # Create a barplot using seaborn
106 sns.barplot(data=pandas_df, x='median_price', y='state')
107 plt.show()
108
109
110 %python
111 from pyspark.sql import SparkSession
112 from pyspark.sql.functions import col, concat_ws
113
114 # Group by 'zip_code' and 'state', calculate the median price, and
      sort in descending order
115 df = (
116     df1.groupBy('zip_code', 'state')
117     .agg({'price': 'median'})
118     .withColumnRenamed('median(price)', 'median_price')
119     .orderBy(col('median_price').desc())
```

```
    .limit(10)
    .withColumn('zip_code-state', concat_ws('-', col('zip_code').cast('int').cast('string'), col('state')))
    .select('zip_code-state', 'median_price')
)
# Convert PySpark DataFrame to Pandas DataFrame for plotting
pandas_df = df.toPandas()

# Create a barplot using seaborn
sns.barplot(data=pandas_df, x='median_price', y='zip_code-state')
plt.show()


%python
from pyspark.sql.functions import col,concat_ws

# Group by 'city' and 'state', calculate the median price, and sort in descending order
df = (df1.groupBy('city', 'state')
    .agg({'price': 'median'})
    .withColumnRenamed('median(price)', 'median_price')
    .orderBy(col('median_price').desc())
    .limit(10)
    .withColumn('city-state', concat_ws('-', col('city'), col('state')))
    .select('city-state', 'median_price')
)

# Convert PySpark DataFrame to Pandas DataFrame for plotting
pandas_df = df.toPandas()

# Create a barplot using seaborn
sns.barplot(data=pandas_df, x='median_price', y='city-state')
plt.show()


%python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when
import seaborn as sns
import matplotlib.pyplot as plt

# Create a new DataFrame with 'bed' values capped at 10
df = df1.withColumn('bed', col('bed').cast('int')).withColumn('bed', when(col('bed') >= 10, 10).otherwise(col('bed')))

# Group by 'bed' and calculate the median price
df_price_bed = df.groupBy('bed').agg({'price': 'median'}).
```

```
    withColumnRenamed('median(price)', 'median_price')

# Convert PySpark DataFrame to Pandas DataFrame for plotting
pandas_df_price_bed = df_price_bed.toPandas()

# Create a barplot using seaborn
sns.barplot(data=pandas_df_price_bed, x='bed', y='median_price')
plt.show()


%python
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
import seaborn as sns
import matplotlib.pyplot as plt

# Select relevant numerical columns for correlation calculation
numerical_columns = ['bed', 'bath', 'acre_lot', 'zip_code', '
    house_size']  # Replace with your actual column names
selected_columns =  numerical_columns + ['price']
df_selected = df1.select(selected_columns)

# Calculate the correlation matrix
corr_df = df_selected.toPandas().corr()

# Plot the heatmap using Seaborn
plt.figure(figsize=(10, 8))
sns.heatmap(corr_df, annot=True, cmap='coolwarm')
plt.title("Correlation Matrix Heatmap")
plt.show()


// Calculate mean and median to fill the null values
val bedMedian = df1.selectExpr("percentile_approx(bed, 0.5, 10000) as
    bed_median").first().getDouble(0)
val bathMedian = df1.selectExpr("percentile_approx(bath, 0.5, 10000) as
     bath_median").first().getDouble(0)
val acreLotMean = df1.selectExpr("round(avg(acre_lot)) as acre_lot_mean
    ").first().getDouble(0)
val houseSizeMean = df1.selectExpr("round(avg(house_size)) as
    house_size_mean").first().getDouble(0)
val priceMean = df1.selectExpr("round(avg(price)) as price_mean").first
    ().getDouble(0)

// Find the most frequent city
val mostFrequentCity = df1.groupBy("city").agg(count("*").alias("count"
    ))
  .orderBy(col("count").desc)
```

```
  .select("city")
  .first()
  .getString(0)

val mostFrequentZipCode = df1.groupBy("zip_code")
  .agg(count("*").alias("count"))
  .orderBy(col("count").desc)
  .select("zip_code")
  .first()
  .getDouble(0)

//Fill the null values
val dfFilled = df1
  .na.fill(bedMedian, Seq("bed"))
  .na.fill(bathMedian, Seq("bath"))
  .na.fill(acreLotMean, Seq("acre_lot"))
  .na.fill(houseSizeMean, Seq("house_size"))
  .na.fill(priceMean, Seq("price"))
  .na.fill(mostFrequentCity, Seq("city"))
  .na.fill(mostFrequentZipCode,Seq("zip_code"))
  .drop("prev_sold_date")

//Calculating the count of null values after filling
dfFilled.select(countCols(dfFilled.columns): _*).show()

//one-hot-encoding to convert categorical variables
// Step 1: Index the categorical column
val indexer = new StringIndexer()
  .setInputCols(Array("status","city", "state"))
  .setOutputCols(Array("status_index","city_index", "state_index"))
  .setHandleInvalid("keep")

val indexedDf = indexer.fit(dfFilled).transform(dfFilled)

// Step 2: One-hot encode the indexed columns
val oneHotEncoder = new OneHotEncoder()
  .setInputCols(Array("status_index","city_index", "state_index"))
  .setOutputCols(Array("status_vec","city_vec", "state_vec"))

val oneHotEncoderModel = oneHotEncoder.fit(indexedDf)
val encoded = oneHotEncoderModel.transform(indexedDf)

val finalDf = encoded.drop("status", "city", "state","status_index","
    city_index","state_index")

// choose the features
val featureCols = Array("bed", "bath","house_size","zip_code","
    status_vec", "city_vec", "state_vec")
```

```
250 val assembler = new VectorAssembler().setInputCols(featureCols).
    setOutputCol("features")
251 val assembledDf = assembler.transform(finalDf.withColumnRenamed("price"
    , "label"))
252 val assembleDf2 = assembledDf.select("features","label")
253
254 //Split the dataset into train and test data
255 val Array(trainData, testData) = assembledDf.randomSplit(Array(0.8,0.2)
    , seed = 123
256
257 //Inital linear regression
258 // Make predictions
259 val linearRegression = new LinearRegression()
260   .setLabelCol("label")
261   .setFeaturesCol("features")
262
263 // Train models
264 val linearRegressionModel = linearRegression.fit(trainData)
265
266 // Make predictions
267 val predictionsLR = linearRegressionModel.transform(testData)
268
269 // Evaluate models
270 val evaluatorLR = new RegressionEvaluator().setLabelCol("label").
    setPredictionCol("prediction").setMetricName("mse")
271
272 val mseLR2 = evaluatorLR.evaluate(predictionsLR)
273
274 // Calculate additional metrics using RegressionMetrics
275 val metricsLR = new RegressionMetrics(predictionsLR.select("prediction"
    , "label").rdd.map(x => (x.getDouble(0), x.getDouble(1))))
276 val rmseLR = metricsLR.rootMeanSquaredError
277 val maeLR = metricsLR.meanAbsoluteError
278 val r2LR = metricsLR.r2
279
280 println(s"Linear Regression RMSE: $rmseLR, MAE: $maeLR, R^2}: $r2LR")
281
282 // To Obtain the best model for the linear regression
283 // Define your Linear Regression model
284 val linearRegression = new LinearRegression()
285   .setLabelCol("label")
286   .setFeaturesCol("features")
287
288 // Define a grid of hyperparameters to search over
289 val paramGrid = new ParamGridBuilder()
290   .addGrid(linearRegression.regParam, Array(0.01, 0.1, 1.0))  //
    Regularization parameter
291   .addGrid(linearRegression.elasticNetParam, Array(0.0, 0.5, 1.0))  //
```

```
    Elastic Net mixing parameter
  .build()

// Define an evaluator
val evaluator = new RegressionEvaluator()
  .setLabelCol("label")
  .setPredictionCol("prediction")
  .setMetricName("mse")

// Define a CrossValidator
val crossValidator = new CrossValidator()
  .setEstimator(linearRegression)
  .setEstimatorParamMaps(paramGrid)
  .setEvaluator(evaluator)
  .setNumFolds(5)  // Number of folds in cross-validation

// Fit the model with the best hyperparameters
val bestModel = crossValidator.fit(trainData)

// Make predictions on the test set using the best model
val predictions = bestModel.transform(testData)

// Evaluate models
val evaluator2 = new RegressionEvaluator().setLabelCol("label").
    setPredictionCol("prediction").setMetricName("mse")
val mseLR2 = evaluator2.evaluate(predictions)

// Calculate additional metrics using RegressionMetrics
val metricsLR = new RegressionMetrics(predictions.select("prediction",
    "label").rdd.map(x => (x.getDouble(0), x.getDouble(1))))
val rmseLR = metricsLR.rootMeanSquaredError
val maeLR = metricsLR.meanAbsoluteError
val r2LR = metricsLR.r2

println(s"Best model Linear Regression RMSE: $rmseLR, MAE: $maeLR, R^2:
    $r2LR")

// Initialize random forest model with maxdepth 25 and NumTrees 15
val randomForestRegressor = new RandomForestRegressor().setLabelCol("
    label").setFeaturesCol("features").setMaxDepth(25).setNumTrees(15)

// Train models
val randomForestModel = randomForestRegressor.fit(trainData)

// Make predictions
val predictionsRF = randomForestModel.transform(testData)

// Evaluate models
```

```
335 val evaluatorRF = new RegressionEvaluator().setLabelCol("label").
    setPredictionCol("prediction").setMetricName("mse")
336 val mseRF = evaluatorRF.evaluate(predictionsRF)
337
338 // Similar calculations for Random Forest and Gradient Boosting
339 val metricsRF = new RegressionMetrics(predictionsRF.select("prediction"
    , "label").rdd.map(x => (x.getDouble(0), x.getDouble(1))))
340 val rmseRF = metricsRF.rootMeanSquaredError
341 val maeRF = metricsRF.meanAbsoluteError
342 val r2RF = metricsRF.r2
343
344 println(s"Random Forest train- RMSE: $rmseRF, MAE: $maeRF, R^2: $r2RF")
345
346
347 // Initialize gradient Boosting model
348 val gradientBoostingRegressor = new GBTRegressor().setLabelCol("label")
    .setFeaturesCol("features")
349
350 // Train models
351 val gradientBoostingModel = gradientBoostingRegressor.fit(trainData)
352
353 // Make predictions
354 val predictionsGB = gradientBoostingModel.transform(testData)
355
356 // Evaluate models
357 val evaluator = new RegressionEvaluator().setLabelCol("label").
    setPredictionCol("prediction").setMetricName("mse")
358
359 val mseGB = evaluator.evaluate(predictionsGB)
360
361 // Similar calculations for Random Forest and Gradient Boosting
362 val metricsGB = new RegressionMetrics(predictionsGB.select("prediction"
    , "label").rdd.map(x => (x.getDouble(0), x.getDouble(1))))
363 val rmseGB = metricsGB.rootMeanSquaredError
364 val maeGB = metricsGB.meanAbsoluteError
365 val r2GB = metricsGB.r2
366
367 println(s"Gradient Boosting RMSE: $rmseGB, MAE: $maeGB, R^2: $r2GB")
```

Listing 1: Project Code