Predicting Prices from Bristol Airbnb Listings

Enric Fernández Sala

2025-03-26

1. Data Pre-processing and Feature Engineering

1.1. Loading and Understanding the Dataset

The dataset we are studying includes various attributes that describe Airbnb listings from Bristol, such as price, location, number of rooms, and user reviews.

We will begin by loading the dataset, filtering out metadata, and retaining only variables that have demonstrated relevance in similar previous studies (Wang & Nicolau, 2017).

In addition, we will eliminate possible duplicated observations and read percentages as proportions to ease the analysis.

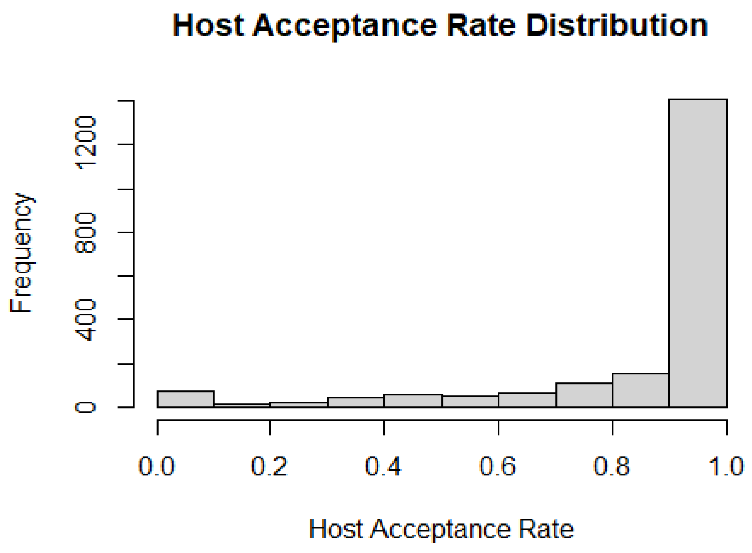
1.2. Missing Values

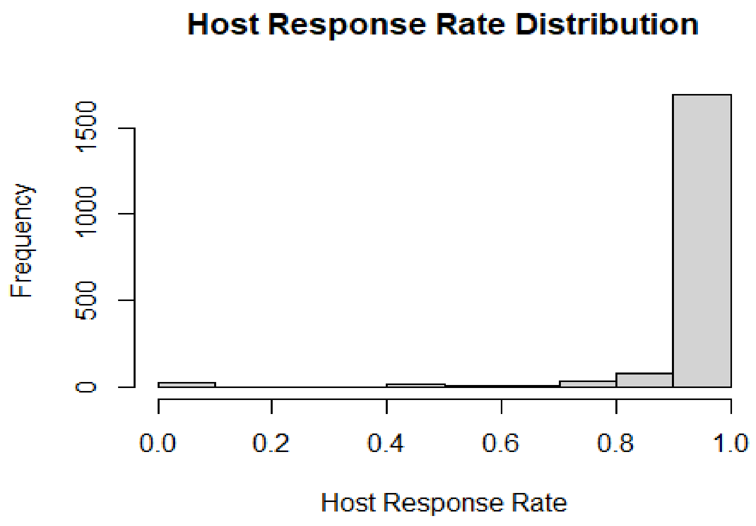
After pre-selecting the raw data we will be using, we need to clean it by addressing missing values. Additionally, some column's NAs were not formatted correctly for R to detect them, so we must correct it.

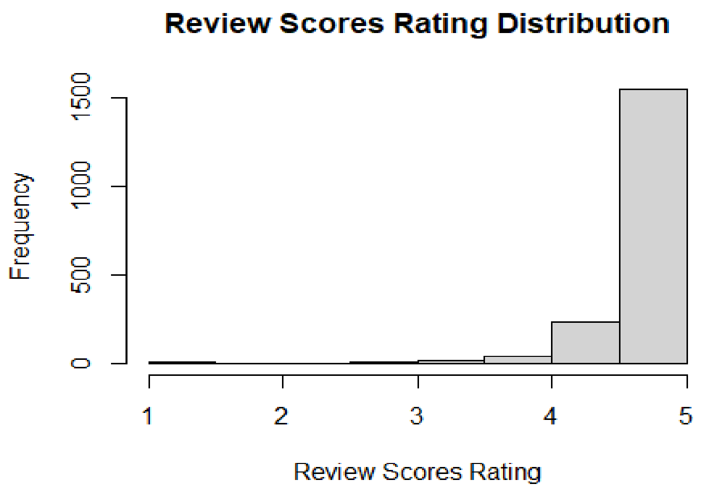
We may also remove observations where the price is missing, since this will not allow us to either train or test our models.

For categorical, binary or numeric variables with few discrete values, NAs can be imputed using the mode. Imputing NAs can introduce bias, as it assumes the missing values. However, this allows us to use all the other variables for the same observations.

The distributions of some numeric variables with a high proportion of NAs are skewed, thus using the median to impute NAs is more appropriate as it is not affected by extreme values, which allows to maintain the integrity of the dataset and facilitate the manipulation of many observations.



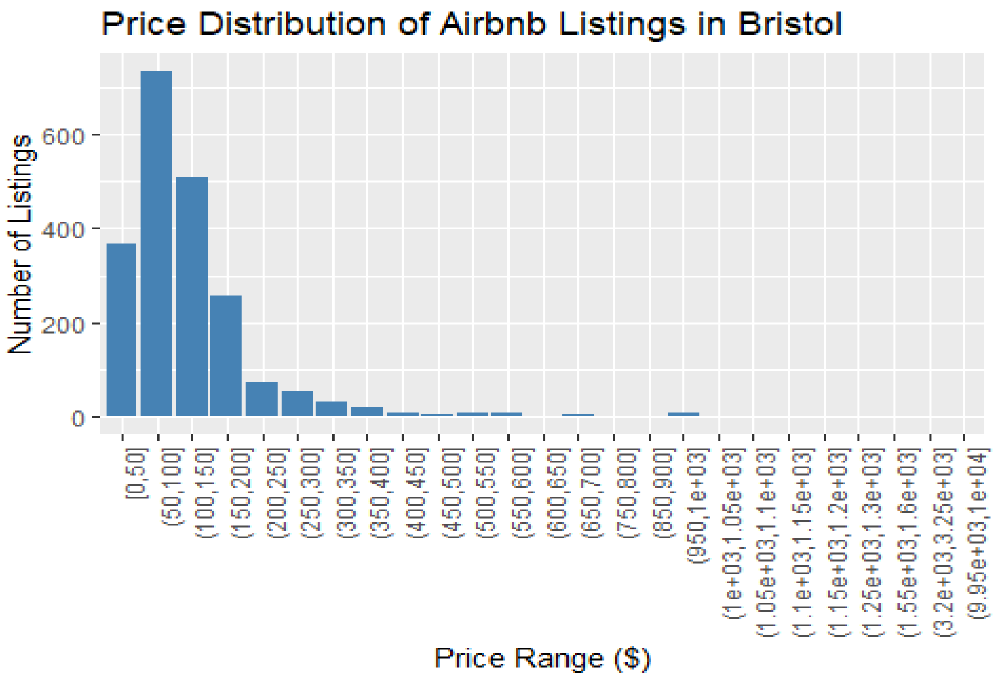




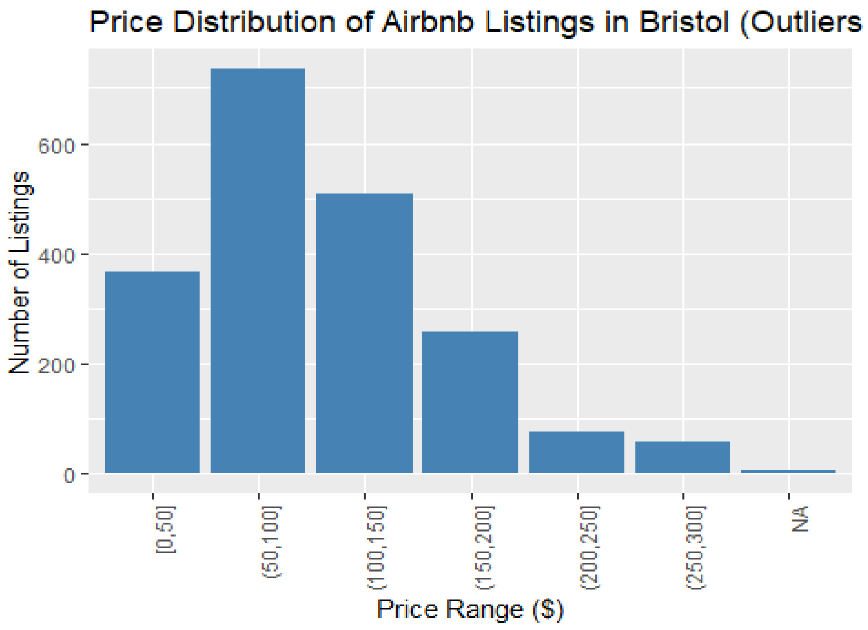
Finally, for host response time, since NAs are a large proportion, we can create the "Unknown" category.

1.3. Outlier Treatment and Feature Scaling

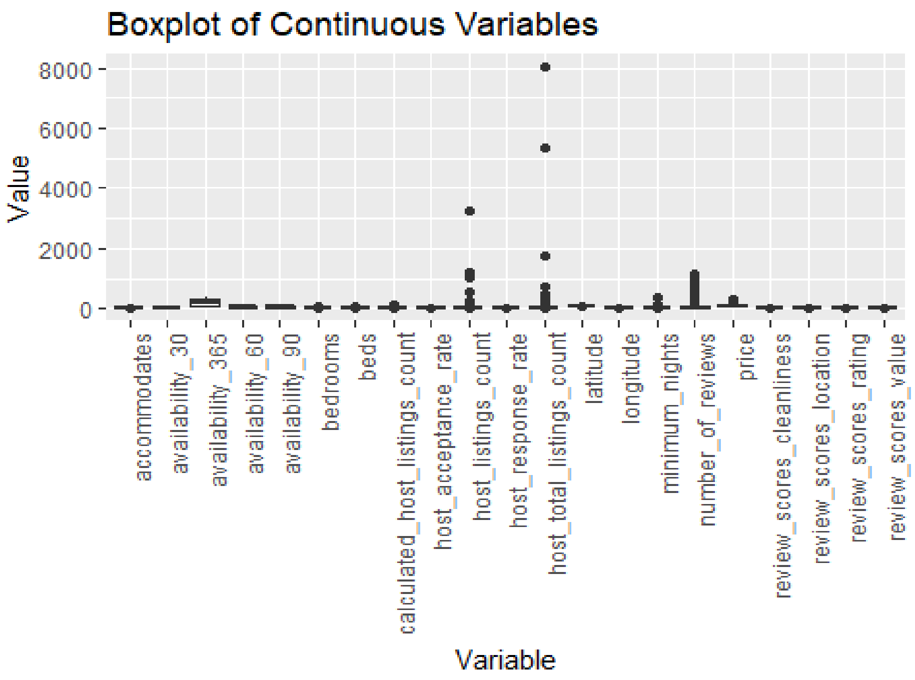
Firstly, we can check the distribution of our objective variable below.



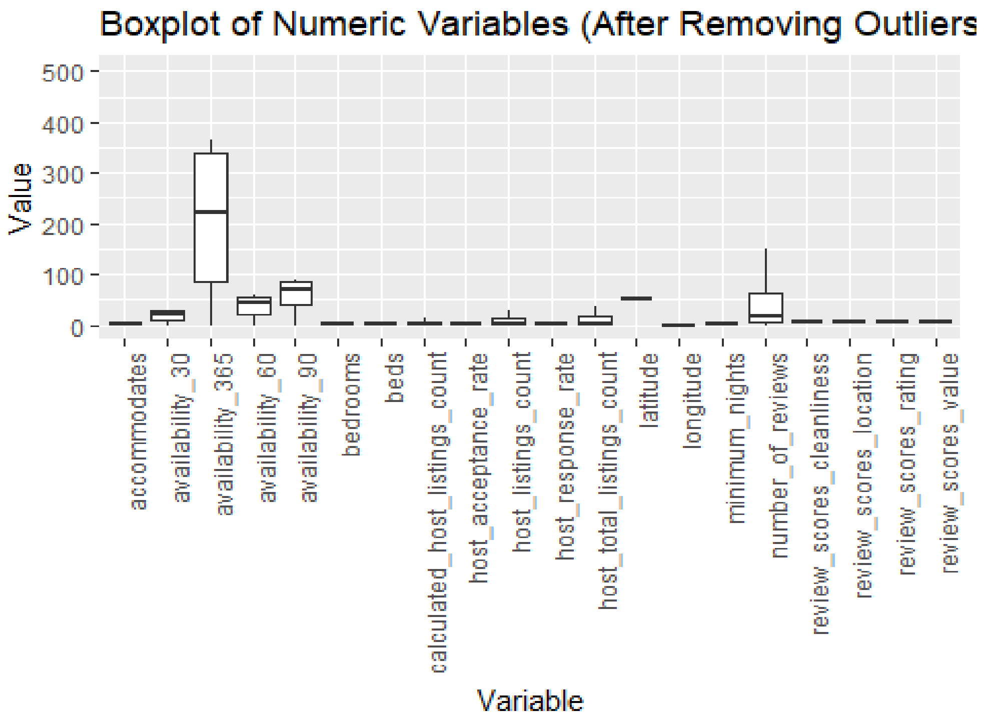
It is really skewed to the right, indicating that we have high outliers and, although they don't represent a significant proportion of prices, they can compromise our predictions due to their extreme values. We can remove the 5% of higher prices to avoid this.



We can check for outliers in other features by looking at boxplots. Outliers in the regressor variables are problematic because they can disproportionately influence the model, leading to biased estimates and reducing the overall accuracy of predictions.



We can see that there are many outliers as well. These can be capped at the 1st and 99th percentile values, significantly improving the distributions.

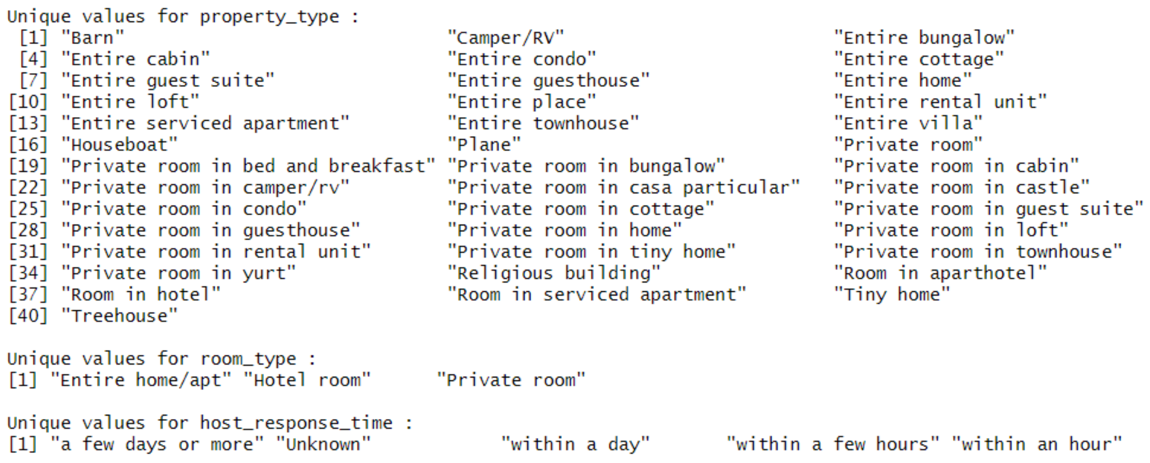


2. Feature Engineering

Feature engineering allows us to derive more useful variables from the ones that we already have by giving them a different format or transforming them. For example, transforming binary variables into numerical format, getting host experience from the date column or dividing bathroom text variable into a numerical and binary (for shared/private).

2.1. Categorical Encoding

Categorical variables like property type and room type have no inherent order, while response time is ordinal. We apply ordinal encoding for response time and one-hot encoding for the others.



```
Unique values for property_type :
 [1] "Barn"                         "Camper/RV"                        "Entire bungalow"
 [4] "Entire cabin"                 "Entire condo"                     "Entire cottage"
 [7] "Entire guest suite"           "Entire guesthouse"                "Entire home"
[10] "Entire loft"                  "Entire place"                     "Entire rental unit"
[13] "Entire serviced apartment"    "Entire townhouse"                 "Entire villa"
[16] "Houseboat"                    "Plane"                            "Private room"
[19] "Private room in bed and breakfast" "Private room in bungalow"    "Private room in cabin"
[22] "Private room in camper/rv"    "Private room in casa particular"  "Private room in castle"
[25] "Private room in condo"        "Private room in cottage"          "Private room in guest suite"
[28] "Private room in guesthouse"   "Private room in home"             "Private room in loft"
[31] "Private room in rental unit"  "Private room in tiny home"        "Private room in townhouse"
[34] "Private room in yurt"         "Religious building"               "Room in aparthotel"
[37] "Room in hotel"                "Room in serviced apartment"       "Tiny home"
[40] "Treehouse"

Unique values for room_type :
[1] "Entire home/apt" "Hotel room"      "Private room"

Unique values for host_response_time :
[1] "a few days or more" "Unknown"            "within a day"       "within a few hours" "within an hour"
```

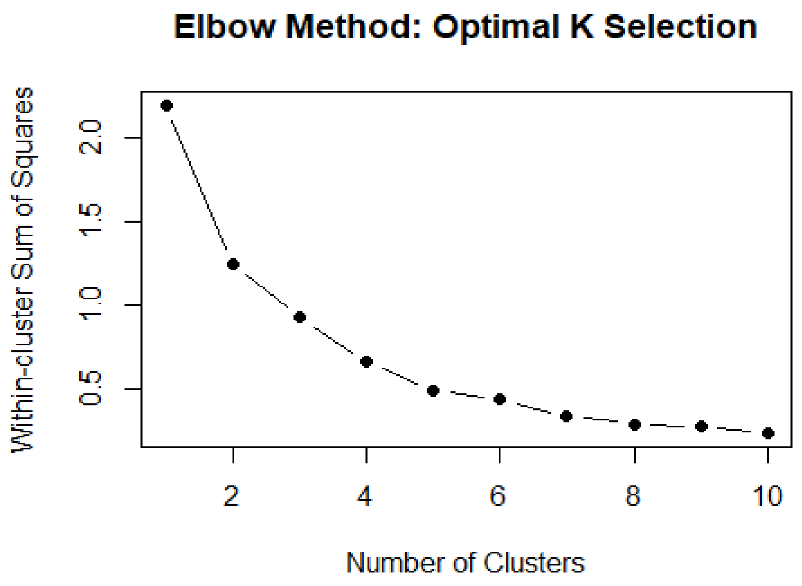
To reduce the high cardinality of property type, we apply a cutoff that removes categories representing only the 2.5% of observations. This simplifies the model by leaving only 6 categories, helping to focus on the most important and common property types.

2.2. Amenities

The amenities column is rich with many strings per observation. We encode each amenity as a binary variable to capture their impact, avoiding those that are present in fewer than 10% of listings with a cutoff point to avoid overfitting and improve model generalization.

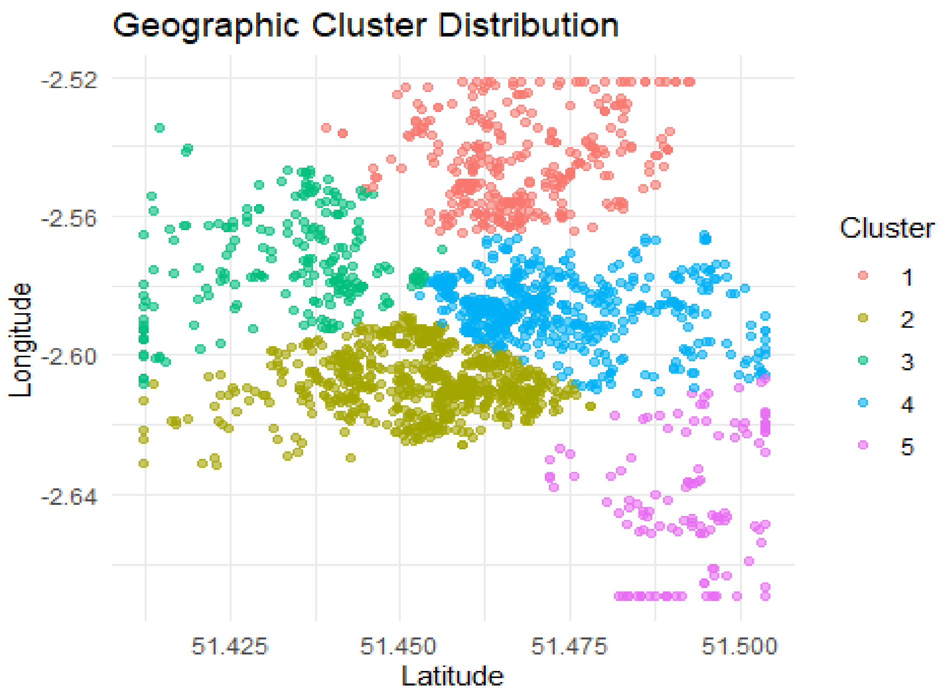
2.3. Clustering by Location

Longitude and latitude features can pose challenges for certain models due to their continuous nature and lack of explicit relationships (neighborhoods or streets with higher prices). To address this, we apply a clustering approach to group listings into five distinct geographical areas. This reduces complexity, allowing models to capture location-specific trends more effectively without relying on raw coordinates.



This plot shows us the relation of WSS (how compact or tight the clusters are) caused by the number of clusters with “k” means. An optimal number of clusters may be 5 due to a lower marginal loss of WSS further on.

The selection of clusters geographically is seen below.



Finally, we can erase the columns we used to create better variables and check the final structure of the dataframe and check the structure.

```
column_types
integer numeric ordered
      3     105       1
```

2.4. Scaling

Standardizing the features ensures that each variable contributes equally to the model by making them have a similar scale, avoiding dominance of high-magnitude variables. This helps improve the performance of many statistical learning methods such as multiple regression.

3. LASSO Regression

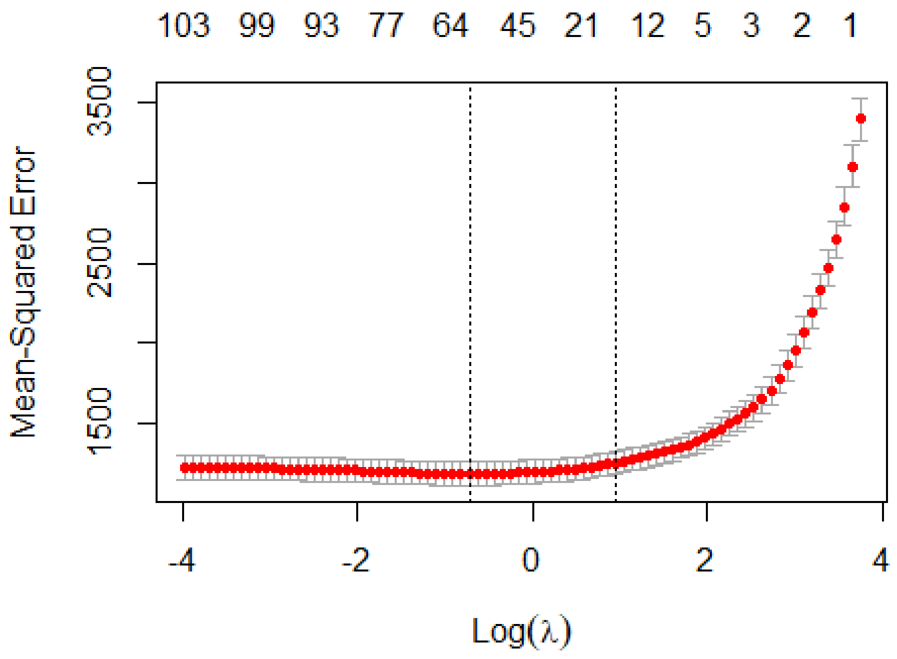
Lasso regression is a good model for predicting using this dataframe because it performs feature selection (+100 variables), manages multicollinearity (many features are correlated) and prevents overfitting with regularization, generalizing better to unseen data.

The first steps we can take is remove some categories (avoiding the dummy trap) and coordinates (already represented with clusters) to avoid unnecessary collinearity.

We may also impose an interaction term for review score variables to only use reviews when there has been one.

Next, we split the dataframe into training and testing sets and use 10-fold cross validation to obtain the optimal penalization parameter value (lambda), which's relation with MSE is seen below.

```
Optimal lambda value: 0.48553
```



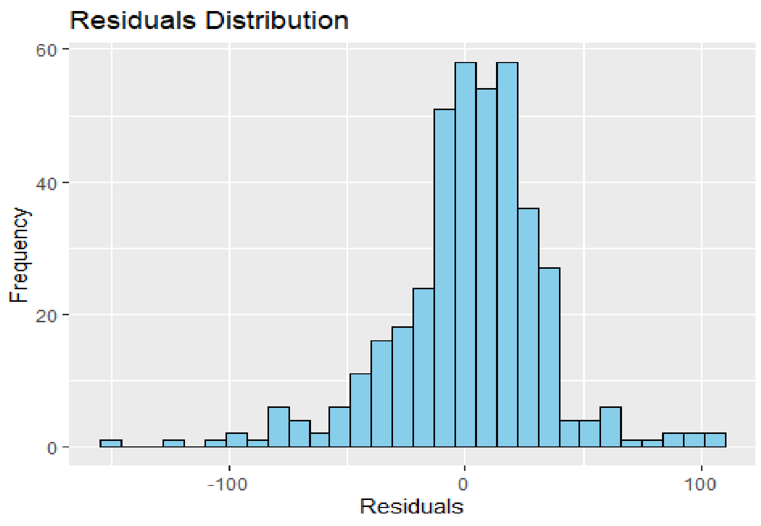
Now we can train the model with the optimal lambda to make predictions and evaluate their performance on the training and testing sets. Here are the results:

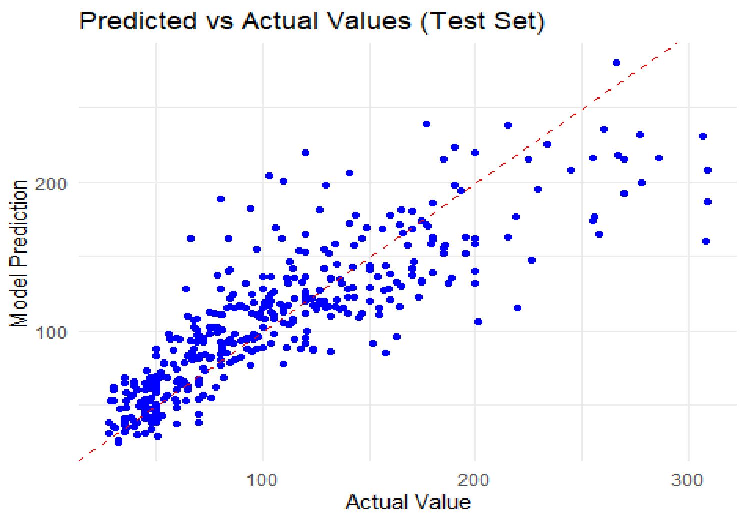
Data <chr>	MAE <dbl>	MSE <dbl>	RMSE <dbl>	R_squared <dbl>
TRAINING	23.31121	1063.710	32.61456	0.6874650
TEST	22.89301	1028.309	32.06725	0.7058193

We can see that for all criteria LASSO has almost the same performance in test and training sets. The RMSEs are quite high considering the price dimension.

```
summary(df_LASSO$price) #For comparison
 Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
 20.0    61.0    95.0   106.3   139.0   309.0 
```

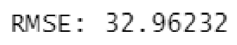
To understand better model performance, we can plot the distribution of residuals and the predicted vs actual prices.





We can see how great residuals increase alongside price, which compromises the performance of the model.

Having tried an logarithmic transformation on prices to see if it improves, the performance in the test set is approximately the same as without logarithm.



```
RMSE: 32.96232
```

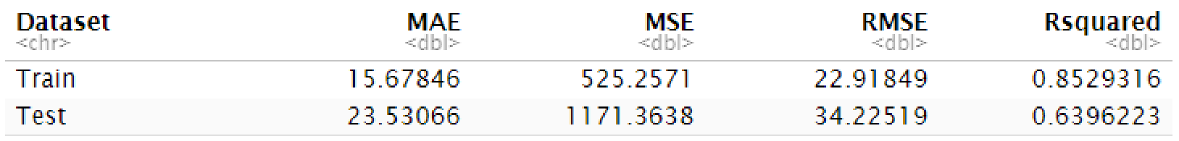
4. Gradient Boosting

If Lasso fails due to non-linearity in the residuals, tree-based methods like Gradient Boosting can improve predictions. These methods capture complex relationships by building decision trees that handle non-linear patterns without needing explicit transformations. This flexibility allows them to model interactions between features more effectively.

Let us impose interactions in reviews. In addition, GB does not need to remove variables with multicollinearity.

Cross-validation allows model training and hyperparameter optimization by splitting the data into subsets. In gradient boosting, it helps find the best hyperparameters by evaluating performance on different folds.

After computing the predictions, we can evaluate model performance.

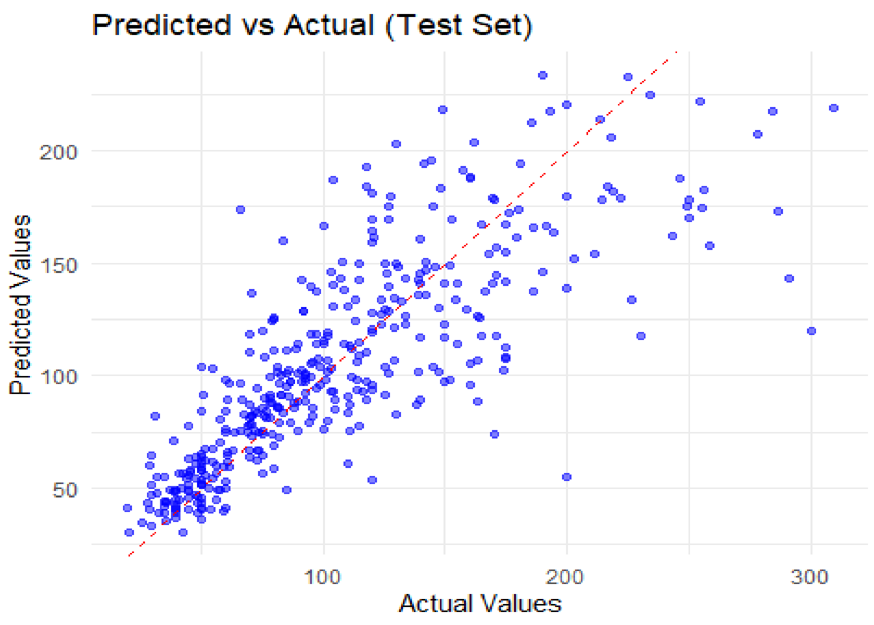


Dataset	MAE	MSE	RMSE	Rsquared
<chr>	<dbl>	<dbl>	<dbl>	<dbl>
Train	15.67846	525.2571	22.91849	0.8529316
Test	23.53066	1171.3638	34.22519	0.6396223

The model overfits the data because the RMSE is substantially lower in the training set than in the testing set.

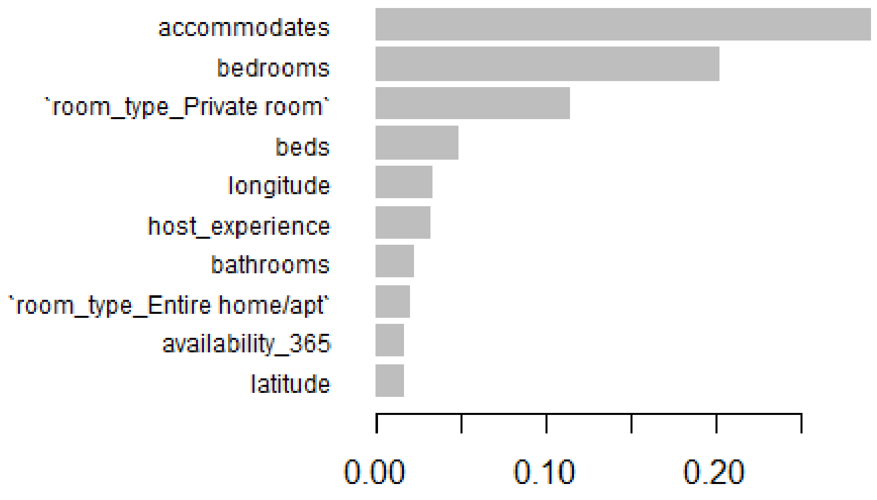
Regarding the testing data, performance is very similar to LASSO regression, suggesting that there has been no improvement in out-of-sample predictions.

We can visualize predicted vs actual values below.



The pattern is the same as with LASSO, errors grow as prices get higher.

Finally, we can see in the feature importance's graph a ranking of which features helped more the algorithm with prediction by repeatedly dividing the data in homogeneous groups.



5. Conclusions

Both Lasso Regression and Gradient Boosting performed similarly, struggling to predict high-priced listings due to outliers and non-linearity.

Lasso helped with feature selection but had high RMSE values, especially for higher prices.

Gradient Boosting overfitted the training data, offering no significant improvement.

Cuttoff points were modified, outliers were included, and other adjustments were made but there has been no significant improvement in model performances.

Feature engineering, including outlier treatment and categorical encoding, improved the dataset's structure, but more advanced models or transformations might be needed to better predict high-priced listings.

6. References

Wang, Dan & Nicolau, Juan. (2017). Price determinants of sharing economy-based accommodation rental: A study of listings from 33 cities on Airbnb.com. International Journal of Hospitality Management. 62. 120-131. 10.1016/j.ijhm.2016.12.007.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning* (1st ed.) [PDF]. Springer.

Rezazadeh Kalehbasti, P., Nikolenko, L., & Rezaei, H. (2021, August). Airbnb price prediction using machine learning and sentiment analysis. In *international cross-domain conference for machine learning and knowledge extraction* (pp. 173-184). Cham: Springer International Publishing.

Yang, S. (2021, March). Learning-based airbnb price prediction model. In *2021 2nd International Conference on E-Commerce and Internet Technology (ECIT)* (pp. 283-288). IEEE.

Tang, E., & Sangani, K. (2015). Neighborhood and price prediction for San Francisco Airbnb listings. *Departments of Computer science, Psychology, economics–Stanford University*, 021-01.

Mahyoub, M., Al Ataby, A., Upadhyay, Y., & Mustafina, J. (2023, January). AIRBNB price prediction using machine learning. In *2023 15th International Conference on Developments in eSystems Engineering (DeSE)* (pp. 166-171). IEEE.

Natekin, A., & Knoll, A. (2013). Gradient boosting machines, a tutorial. *Frontiers in neurorobotics*, *7*, 21.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, *58*(1), 267-288.

Likas, A., Vlassis, N., & Verbeek, J. J. (2003). The global k-means clustering algorithm. *Pattern recognition*, *36*(2), 451-461.

Cox, M. (2024). *Inside Airbnb: Adding data to the debate – Bristol listings (December 25, 2024)*. Inside Airbnb. Retrieved from https://data.insideairbnb.com/united-kingdom/england/bristol/2024-12-25/data/listings.csv.gz

OpenAI. (2023). *ChatGPT* (Mar 27 version) [Large language model]. https://openai.com

7. Appendix (Code)

```
library(readr)
library(tidyverse)

## ── Attaching core tidyverse packages ────────────────────────
tidyverse 2.0.0 ──
## ✓ dplyr     1.1.4     ✓ purrr     1.0.2
## ✓ forcats   1.0.0     ✓ stringr   1.5.1
## ✓ ggplot2   3.5.1     ✓ tibble    3.2.1
## ✓ lubridate 1.9.4     ✓ tidyr     1.3.1
## ── Conflicts ──────────────────────────────────────────
tidyverse_conflicts() ──
## ✕ dplyr::filter() masks stats::filter()
## ✕ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force
all conflicts to become errors

library(dplyr)
library(GGally)

## Registered S3 method overwritten by 'GGally':
##   method from
##   +.gg   ggplot2

library(ggplot2)
library(DescTools)

## Warning: package 'DescTools' was built under R version 4.4.3

library(jsonlite)

##
## Adjuntando el paquete: 'jsonlite'
##
## The following object is masked from 'package:purrr':
```

```
##
##     flatten
```

```r
library(fastDummies)
```

```
## Warning: package 'fastDummies' was built under R version 4.4.3
```

```r
library(glmnet)
```

```
## Cargando paquete requerido: Matrix
##
## Adjuntando el paquete: 'Matrix'
##
## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
##
## Loaded glmnet 4.1-8
```

```r
library(caret)
```

```
## Warning: package 'caret' was built under R version 4.4.3
```

```
## Cargando paquete requerido: lattice
##
## Adjuntando el paquete: 'caret'
##
## The following objects are masked from 'package:DescTools':
##
##     MAE, RMSE
##
## The following object is masked from 'package:purrr':
##
##     lift
```

```r
library(tidyverse)

# Load dataset
listings_bristol <- read_csv("C:/Users/Usuario/Downloads/listings
(4).csv/listings.bristol.csv")
```

```
## Rows: 2709 Columns: 75
## ── Column specification
────────────────────────────────────────────────────────
## Delimiter: ","
## chr  (24): listing_url, source, name, description,
neighborhood_overview, pi...
## dbl  (38): id, scrape_id, host_id, host_listings_count,
host_total_listings_...
## lgl   (8): host_is_superhost, host_has_profile_pic,
host_identity_verified, ...
## date  (5): last_scraped, host_since, calendar_last_scraped,
```

```r
first_review, la...
##
## ℹ Use `spec()` to retrieve the full column specification for this
data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet
this message.

# Remove duplicate rows
raw_df <- unique(listings_bristol)

# Select relevant variables
df_shortened <- raw_df %>%
  select(
    # Property characteristics
    property_type, room_type, accommodates, bathrooms_text, bedrooms,
beds, amenities, latitude, longitude,

    # Host information
    host_is_superhost, host_listings_count, host_total_listings_count,
host_response_rate, host_acceptance_rate, host_since, host_response_time,

    # Availability & minimum stay
    minimum_nights, availability_30, availability_60, availability_90,
availability_365,

    # Reviews & ratings
    number_of_reviews, review_scores_rating, review_scores_cleanliness,
review_scores_location, review_scores_value,

    # Booking & pricing factors
    instant_bookable, calculated_host_listings_count,

    # Target variable
    price
  )

# Convert categorical variables to factors
df_shortened <- df_shortened %>%
  mutate(across(c(property_type, room_type, host_is_superhost,
instant_bookable), as.factor))

# Convert percentage values to numeric
df_shortened<- df_shortened %>%
  mutate(
    host_response_rate = as.numeric(gsub("%", "", host_response_rate)) /
100,
    host_acceptance_rate = as.numeric(gsub("%", "",
host_acceptance_rate)) / 100
  )
```

```
## Warning: There were 2 warnings in `mutate()`.
## The first warning was:
## ℹ In argument: `host_response_rate = as.numeric(gsub("%", "",
##   host_response_rate))/100`.
## Caused by warning:
## ! NAs introducidos por coerción
## ℹ Run `dplyr::last_dplyr_warnings()` to see the 1 remaining warning.
```

```
# Define numeric variables
numeric_cols <- c("accommodates", "bedrooms", "beds", "latitude",
"longitude",
                  "host_listings_count", "host_total_listings_count",
"host_response_rate",
                  "host_acceptance_rate", "minimum_nights",
"availability_30", "availability_60",
                  "availability_90", "availability_365",
"number_of_reviews", "review_scores_rating",
                  "review_scores_cleanliness", "review_scores_location",
"review_scores_value",
                  "calculated_host_listings_count")

# Convert selected columns to numeric
df_shortened <- df_shortened %>%
  mutate(across(all_of(numeric_cols), ~ as.numeric(.)))


# Convert price by removing "$" and commas
df_shortened <- df_shortened %>%
  mutate(price = as.numeric(gsub("[$,]", "", price)))

# Convert categorical variables to factors
df_shortened <- df_shortened %>%
  mutate(across(c(property_type, room_type, host_is_superhost,
instant_bookable), as.factor))

# Replace "N/A" with NA in host response rate and acceptance rate
na_cols <- c("host_response_rate", "host_acceptance_rate",
"host_response_time")
df_shortened[na_cols] <- lapply(df_shortened[na_cols], function(x)
ifelse(x == "N/A", NA, x))

# Count missing values for each column
na_counts <- colSums(is.na(df_shortened))
na_counts <- na_counts[na_counts > 0]  # Filter only columns with missing
values
print(na_counts)  # Display columns with missing values
```

```
##             bathrooms_text                      bedrooms
beds
##                          1                           183
```

```
605
##         host_is_superhost        host_response_rate 
host_acceptance_rate 
##                        22                       550 
297 
##        host_response_time      review_scores_rating 
review_scores_cleanliness 
##                       550                       330 
330 
##    review_scores_location       review_scores_value 
price 
##                       330                       330 
602
```

```
# Separate data into those with and without a price
df_price <- df_shortened %>% filter(!is.na(price))
df_noprice <- df_shortened %>% filter(is.na(price))

library(DescTools)

# Fill missing values with mode for categorical variables
df_price$bathrooms_text[is.na(df_price$bathrooms_text)] <- "1 bath"
df_price$host_is_superhost[is.na(df_price$host_is_superhost)] <- "FALSE"
# Fill missing values with mode for some numerical variables
df_price$bedrooms[is.na(df_price$bedrooms)] <- 1
df_price$beds[is.na(df_price$beds)] <- 1



Mode(df_price$bathrooms_text)
```

```
## [1] "1 bath"
## attr(,"freq")
## [1] 1074
```

```
Mode(df_price$host_is_superhost)
```

```
## [1] FALSE
## attr(,"freq")
## [1] 1355
## Levels: FALSE TRUE
```

```
Mode(df_price$bedrooms)
```

```
## [1] 1
## attr(,"freq")
## [1] 1162
```

```
Mode(df_price$beds)
```

```
## [1] 1
## attr(,"freq")
## [1] 1004
```

```r
# Plot histograms to assess distributions before imputing missing values
hist(df_price$host_acceptance_rate, main = "Host Acceptance Rate
Distribution", xlab = "Host Acceptance Rate")

hist(df_price$host_response_rate, main = "Host Response Rate
Distribution", xlab = "Host Response Rate")

hist(df_price$review_scores_rating, main = "Review Scores Rating
Distribution", xlab = "Review Scores Rating") #Quite epresentative of the
others due to correlation

# Compute correlation between host response rate and acceptance rate
cor(df_price$host_response_rate, df_price$host_acceptance_rate, use =
"complete.obs")
```

```
## [1] 0.482623
```

```r
# Function to fill missing values with the median
fill_with_median <- function(column) {
  column[is.na(column)] <- median(column, na.rm = TRUE)
  return(column)
}

# Apply median imputation to numeric columns with missing values
df_price <- df_price %>%
  mutate(
    host_response_rate = fill_with_median(host_response_rate),
    host_acceptance_rate = fill_with_median(host_acceptance_rate),
    review_scores_rating = fill_with_median(review_scores_rating),
    review_scores_cleanliness =
fill_with_median(review_scores_cleanliness),
    review_scores_location = fill_with_median(review_scores_location),
    review_scores_value = fill_with_median(review_scores_value)
  )

#Create Unknown column for host response time (significant number of
missing values)
df_price$host_response_time[is.na(df_price$host_response_time)] <-
"Unknown"
df_price$host_response_time <- as.factor(df_price$host_response_time)

#Check for possible NAs left
na_countsp <- colSums(is.na(df_price))
na_countsp <- na_countsp[na_countsp > 0]  # Filter only columns with
missing values
print(na_countsp)
```

```
## named numeric(0)
```

```r
# Create price bins (intervals)
df_price <- df_price %>%
  mutate(price_category = cut(price, breaks = seq(0, max(price, na.rm =
TRUE), by = 50), include.lowest = TRUE))
# Count the number of listings per price category
price_counts <- df_price %>%
  group_by(price_category = cut(price, breaks = seq(0, max(price, na.rm =
TRUE) + 50, by = 50), include.lowest = TRUE)) %>%
  summarise(count = n())

# Plot price distribution using ggplot2
ggplot(price_counts, aes(x = price_category, y = count)) +
  geom_bar(stat = "identity", fill = "steelblue") +
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) +  # Rotate
labels for better readability
  labs(title = "Price Distribution of Airbnb Listings in Bristol",
       x = "Price Range ($)",
       y = "Number of Listings")

# PRICE OUTLIERS
# Calculate the 95th percentile price threshold
price_threshold <- quantile(df_price$price, 0.95, na.rm = TRUE)

# Filter out listings with prices above the threshold
df_price <- df_price %>% filter(price <= price_threshold)

# Recalculate the price counts based on the filtered data
price_counts <- df_price %>%
  group_by(price_category = cut(price, breaks = seq(0, max(price, na.rm =
TRUE), by = 50), include.lowest = TRUE)) %>%
  summarise(count = n())

# Plot price distribution using ggplot2
ggplot(price_counts, aes(x = price_category, y = count)) +
  geom_bar(stat = "identity", fill = "steelblue") +
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) +  # Rotate
labels for better readability
  labs(title = "Price Distribution of Airbnb Listings in Bristol
(Outliers Removed)",
       x = "Price Range ($)",
       y = "Number of Listings")

# Look for numeric columns
df_price %>%
  select(where(is.numeric)) %>%
  select(where(~n_distinct(.) > 2)) %>%  # Exclude dummies
  gather(key = "Variable", value = "Value") %>%
  ggplot(aes(x = Variable, y = Value)) +
  geom_boxplot() +
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
  labs(title = "Boxplot of Continuous Variables")
```

```r
# Function to adjust the highest and lowest values to defined percentiles
adjust_outliers_percentiles <- function(df_price, columns,
upper_percentile = 0.99, lower_percentile = 0.01) {
  for (col in columns) {
    # Calculate the upper (99%) and lower (1%) percentiles
    upper_limit <- quantile(df_price[[col]], upper_percentile, na.rm =
TRUE)
    lower_limit <- quantile(df_price[[col]], lower_percentile, na.rm =
TRUE)

    # Replace values above the upper percentile with the 99th percentile
value
    df_price[[col]] <- ifelse(df_price[[col]] > upper_limit, upper_limit,
                              ifelse(df_price[[col]] < lower_limit,
lower_limit, df_price[[col]]))
  }
  return(df_price)
}

# Select numeric columns, excluding 'price' and binary variables (only 2
unique values)
numeric_columns <- df_price %>%
  select(where(is.numeric)) %>%
  select(-price) %>%  # Exclude 'price'
  select(where(~n_distinct(.) > 2)) %>%  # Exclude columns with only 2
unique values (binary)
  colnames()

# Apply the function to adjust outliers to the percentiles
df_price <- adjust_outliers_percentiles(df_price, numeric_columns)

# Visualize the results with boxplots
df_price %>%
  select(where(is.numeric) & !starts_with("price")) %>%
  select(where(~n_distinct(.) > 2)) %>%  # Exclude binary numeric
variables
  gather(key = "Variable", value = "Value") %>%
  ggplot(aes(x = Variable, y = Value)) +
  geom_boxplot(outlier.shape = NA) +  # Hide the outliers
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
  labs(title = "Boxplot of Numeric Variables (After Removing Outliers)")

# CONVERTING BINARY VARIABLES TO NUMERICAL FORM
df_price$host_is_superhost <- ifelse(df_price$host_is_superhost ==
"TRUE", 1, 0)
df_price$instant_bookable <- ifelse(df_price$instant_bookable == "TRUE",
1, 0)

reference_date <- as.Date("2025-03-02")
# Compute difference between dates to get host experience
```

```r
df_price$host_experience <- as.numeric(difftime(reference_date,
df_price$host_since, units = "days")) / 365.25
# See new variable
summary(df_price$host_experience)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.2108  4.5866  8.3244  7.2403  9.6612 14.4997

# Create a binary variable indicating if a listing has reviews
df_price <- df_price %>%
  mutate(has_reviews = ifelse(number_of_reviews > 0, 1, 0))

# Create two variables from the bathroom column
df_price <- df_price %>%
  mutate(
    bathrooms = case_when(
      str_detect(str_to_lower(bathrooms_text), "half") ~ 0.5,  # Assign
0.5 for half-baths
      TRUE ~ as.numeric(str_extract(bathrooms_text, "\\d*\\.?\\d+"))  #
Extract numeric values
    ),
    shared_bathroom = ifelse(str_detect(str_to_lower(bathrooms_text),
"shared"), 1, 0)  # Detect shared
  )

#ORDINAL ENCODERS?
catg_vars <- c("property_type", "room_type", "host_response_time")
# Print the unique categories/levels for each variable
for (var in catg_vars) {
  cat("\nUnique values for", var, ":\n")
  if (is.factor(df_price[[var]])) {
    print(levels(df_price[[var]]))  # For factor variables, show the
levels
  } else if (is.character(df_price[[var]])) {
    print(unique(df_price[[var]]))  # For character variables, show
unique values
  } else {
    print(sort(unique(df_price[[var]])))  # For numeric variables, show
sorted unique values
  }
}

##
## Unique values for property_type :
##  [1] "Barn"                               "Camper/RV"
##  [3] "Entire bungalow"                    "Entire cabin"
##  [5] "Entire condo"                       "Entire cottage"
##  [7] "Entire guest suite"                 "Entire guesthouse"
##  [9] "Entire home"                        "Entire loft"
## [11] "Entire place"                       "Entire rental unit"
## [13] "Entire serviced apartment"          "Entire townhouse"
## [15] "Entire villa"                      "Houseboat"
## [17] "Plane"                             "Private room"
## [19] "Private room in bed and breakfast" "Private room in bungalow"
## [21] "Private room in cabin"             "Private room in camper/rv"
## [23] "Private room in casa particular"   "Private room in castle"
## [25] "Private room in condo"             "Private room in cottage"
## [27] "Private room in guest suite"       "Private room in guesthouse"
## [29] "Private room in home"              "Private room in loft"
## [31] "Private room in rental unit"       "Private room in tiny home"
## [33] "Private room in townhouse"         "Private room in yurt"
## [35] "Religious building"                "Room in aparthotel"
## [37] "Room in hotel"                     "Room in serviced apartment"
## [39] "Tiny home"                         "Treehouse"
##
## Unique values for room_type :
## [1] "Entire home/apt" "Hotel room"      "Private room"
##
## Unique values for host_response_time :
## [1] "a few days or more" "Unknown"            "within a day"
## [4] "within a few hours" "within an hour"
```

```r
#Ordinality for host response time
df_price$host_response_time <- factor(df_price$host_response_time,
                                      levels = c("a few days or more",
                                                 "within a day",
                                                 "within a few hours",
                                                 "within an hour",
                                                 "Unknown"),
                                      ordered = TRUE)



# ONE HOT ENCODING
categorical_columns <- c("property_type", "room_type")
#Calculate the frequency of each property type
property_type_freq <- table(df_price$property_type)
# Filter property types that are > 2.5% of total observations
property_types_over_2.5_percent <-
names(property_type_freq[property_type_freq / nrow(df_price) > 0.025])
# Create binary columns only for property types over 2.5% of observations
for (property_type in property_types_over_2.5_percent) {
  df_price[[make.names(property_type)]] <- ifelse(df_price$property_type
== property_type,1,0)
}

# Binary cols for room type
df_price <- dummy_cols(df_price, select_columns = "room_type",
remove_first_dummy = FALSE)

# AMENITIES
# Paso 1: Convertir amenities JSON a listas
```

```
df_price$amenities <- lapply(df_price$amenities, fromJSON)
# Paso 2: Crear una tabla de frecuencia de todas las amenities
all_amenities <- unlist(df_price$amenities)
amenity_freq <- table(all_amenities)
# Paso 3: Seleccionar amenities que aparecen en más del 10% de los
anuncios
cutoff <- 0.10
popular_amenities <- names(amenity_freq[amenity_freq / nrow(df_price) >
cutoff])
# Paso 4: Crear variables binarias para las amenities seleccionadas
for (amenity in popular_amenities) {
  df_price[[make.names(amenity)]] <- sapply(df_price$amenities,
function(x) ifelse(amenity %in% x, 1, 0))
}
# Paso 5: Mantener solo las columnas de las amenities seleccionadas (sin
calcular correlaciones)
df_price <- df_price %>% select(all_of(make.names(popular_amenities)),
everything())
# Ver las amenities finales seleccionadas
print(popular_amenities)
```

```
##  [1] "Backyard"
##  [2] "Baking sheet"
##  [3] "Bathtub"
##  [4] "Bed linens"
##  [5] "Blender"
##  [6] "Board games"
##  [7] "Body soap"
##  [8] "Books and reading material"
##  [9] "Carbon monoxide alarm"
## [10] "Central heating"
## [11] "Cleaning available during stay"
## [12] "Cleaning products"
## [13] "Clothing storage"
## [14] "Coffee"
## [15] "Coffee maker"
## [16] "Conditioner"
## [17] "Cooking basics"
## [18] "Dedicated workspace"
## [19] "Dining table"
## [20] "Dishes and silverware"
## [21] "Dishwasher"
## [22] "Dryer"
## [23] "Drying rack for clothing"
## [24] "Essentials"
## [25] "Exterior security cameras on property"
## [26] "Extra pillows and blankets"
## [27] "Fire extinguisher"
## [28] "First aid kit"
## [29] "Free parking on premises"
## [30] "Free street parking"
## [31] "Free washer – In unit"
## [32] "Freezer"
## [33] "Hair dryer"
## [34] "Hangers"
## [35] "Heating"
## [36] "Host greets you"
## [37] "Hot water"
## [38] "Hot water kettle"
## [39] "Iron"
## [40] "Kitchen"
## [41] "Laundromat nearby"
## [42] "Lock on bedroom door"
## [43] "Lockbox"
## [44] "Long term stays allowed"
## [45] "Luggage dropoff allowed"
## [46] "Microwave"
## [47] "Outdoor dining area"
## [48] "Outdoor furniture"
## [49] "Oven"
## [50] "Pets allowed"
## [51] "Portable fans"
## [52] "Private backyard – Fully fenced"
## [53] "Private entrance"
## [54] "Private patio or balcony"
## [55] "Refrigerator"
## [56] "Room-darkening shades"
## [57] "Self check-in"
## [58] "Shampoo"
## [59] "Shower gel"
## [60] "Smoke alarm"
## [61] "Stove"
## [62] "Toaster"
## [63] "TV"
## [64] "Washer"
## [65] "Wifi"
## [66] "Wine glasses"
```

```r
# CLUSTERING: ELBOW METHOD
set.seed(123)  # To ensure reproducibility of the results
wss <- numeric(10)  # Create an empty vector to store the WSS values
# Perform K-means for different numbers of clusters (from 1 to 10)
for (k in 1:10) {
  kmeans_result <- kmeans(df_price[, c("latitude", "longitude")], centers
= k)
  wss[k] <- kmeans_result$tot.withinss  # Store the WSS for each k
}
# Plot the Elbow Method
plot(1:10, wss, type = "b", pch = 19, xlab = "Number of Clusters", ylab =
"Within-cluster Sum of Squares",
     main = "Elbow Method: Optimal K Selection")

# Perform K-means clustering on latitude and longitude
set.seed(123)  # For reproducibility
kmeans_result <- kmeans(df_price[, c("latitude", "longitude")], centers =
5)  # Use 5 clusters, adjust as necessary
df_price$geo_cluster <- kmeans_result$cluster

# Plot the clusters on a scatter map of latitude and longitude
ggplot(df_price, aes(x = latitude, y = longitude, color =
as.factor(geo_cluster))) +
  geom_point(alpha = 0.6) +
  labs(title = "Geographic Cluster Distribution",
       x = "Latitude",
       y = "Longitude",
       color = "Cluster") +
  theme_minimal() +
  scale_color_discrete(name = "Cluster")

# Create binary columns for each cluster
df_price$cluster_1 <- ifelse(df_price$geo_cluster == 1, 1, 0)
df_price$cluster_2 <- ifelse(df_price$geo_cluster == 2, 1, 0)
df_price$cluster_3 <- ifelse(df_price$geo_cluster == 3, 1, 0)
df_price$cluster_4 <- ifelse(df_price$geo_cluster == 4, 1, 0)
df_price$cluster_5 <- ifelse(df_price$geo_cluster == 5, 1, 0)

# Use dplyr to erase columns
df_price_final <- df_price %>%
  select(-c(property_type, room_type, bathrooms_text, amenities,
host_since, price_category, geo_cluster))

# Obtain type of column
column_types <- sapply(df_price_final, function(x) class(x)[1])
# Count how many columns of each type
table(column_types)

## column_types
## integer numeric ordered
##       3     105       1

# Use dplyr to erase columns
df_price_final <- df_price %>%
  select(-c(property_type, room_type, bathrooms_text, amenities,
host_since, price_category, geo_cluster))

# Obtain type of column
column_types <- sapply(df_price_final, function(x) class(x)[1])
# Count how many columns of each type
table(column_types)
```

```
## column_types
## integer numeric ordered
##       3     105       1
```

```r
#NORMALISATION/STANDARIZTION
# Identify numeric columns (nonbinary)
non_binary_cols <- sapply(df_price_final, function(x) is.numeric(x) &&
max(x, na.rm = TRUE) != 1)
non_binary_cols["price"] <- FALSE
# See selected columns
names(df_price_final)[non_binary_cols]
```

```
##  [1] "accommodates"                   "bedrooms"
##  [3] "beds"                           "latitude"
##  [5] "longitude"                      "host_listings_count"
##  [7] "host_total_listings_count"      "minimum_nights"
##  [9] "availability_30"                "availability_60"
## [11] "availability_90"                "availability_365"
## [13] "number_of_reviews"              "review_scores_rating"
## [15] "review_scores_cleanliness"      "review_scores_location"
## [17] "review_scores_value"            "calculated_host_listings_count"
## [19] "host_experience"                "bathrooms"
```

```r
#STANDARISATION (ZSCORE FOR LINEAR REGRESSION AND TREES

# Apply standarisation
df_price_stdt <- df_price_final %>%
  mutate(across(
    .cols = which(non_binary_cols),   # Select non binary columns
    .fns = ~ scale(.)
  )) %>%
  mutate(across(
    .cols = which(non_binary_cols),
    .fns = as.numeric                 # Transform from matrix to
numerical
  ))

# Remove One Category to Avoid Dummy Trap
df_LASSO <- df_price_stdt %>%
  select(-c(`room_type_Private room`, cluster_1 , longitude, latitude))
#Coordinates already captures in clusters


#INTERACTIONS

df_LASSO$Rating_Interaction <-
df_LASSO$has_reviews*df_LASSO$review_scores_rating
df_LASSO$Cleanliness_Interaction <-
df_LASSO$has_reviews*df_LASSO$review_scores_cleanliness
df_LASSO$Location_Interaction <-
```

```
df_LASSO$has_reviews*df_LASSO$review_scores_location
df_LASSO$Value_Interaction <-
df_LASSO$has_reviews*df_LASSO$review_scores_value

# Only mantain Interaction Columns
df_LASSO <- df_LASSO %>%
  select(-has_reviews,
         -review_scores_rating,
         -review_scores_cleanliness,
         -review_scores_location,
         -review_scores_value)

df_LASSO$host_response_time <-
as.numeric(as.factor(df_LASSO$host_response_time))

# 1. Split the data into training and testing sets
set.seed(321)  # For reproducibility
trainIndex <- createDataPartition(df_LASSO$price, p = 0.8, list = FALSE)
# Use 80% for training
train_data <- df_LASSO[trainIndex, ]
test_data <- df_LASSO[-trainIndex, ]

# 2. Fit the Lasso model with cross-validation (cv.glmnet optimizes
lambda)
# In glmnet, "alpha = 1" specifies that we are using Lasso (L1
regularization)
lasso_model <- cv.glmnet(
  x = as.matrix(train_data %>% select(-price)),  # Exclude the dependent
variable 'price'
  y = train_data$price,
  alpha = 1,  # Lasso regularization (L1)
  standardize = TRUE,  # Standardizes the variables (important for Lasso)
  nfolds = 10,  # Number of folds in cross-validation
  lambda.min.ratio = 0.0001  # Minimum lambda ratio
)

# 3. Display the optimal lambda value
# Plot the cross-validation results for lambda
plot(lasso_model)

cat("Optimal lambda value:", lasso_model$lambda.min, "\n")

## Optimal lambda value: 0.48553

# 4. Train the model with the best lambda
best_lambda <- lasso_model$lambda.min
lasso_final_model <- glmnet(
  x = as.matrix(train_data %>% select(-price)),
  y = train_data$price,
  alpha = 1,
```

```r
  lambda = best_lambda,
  standardize = TRUE
)

# 5. PREDICTIONS

# Make predictions on the test set
test_predictions <- predict(lasso_final_model, s = best_lambda, newx =
as.matrix(test_data %>% select(-price)))
test_predictions <- as.vector(test_predictions)

# Make predictions on the training set
train_predictions <- predict(lasso_final_model, s = best_lambda, newx =
as.matrix(train_data %>% select(-price)))
train_predictions <- as.vector(train_predictions)

# 6. EVALUATE MODEL PERFORMANCE

# Function to calculate performance metrics
calculate_metrics <- function(actual, predicted) {
  mae <- mean(abs(predicted - actual))  # Mean Absolute Error
  mse <- mean((predicted - actual)^2)   # Mean Squared Error
  rmse <- sqrt(mse)                     # Root Mean Squared Error

  # R-squared Calculation
  ss_total <- sum((actual - mean(actual))^2)  # Total sum of squares
  ss_residual <- sum((predicted - actual)^2)  # Residual sum of squares
  r_squared <- 1 - (ss_residual / ss_total)

  # Return a named list
  return(list(MAE = mae, MSE = mse, RMSE = rmse, R_squared = r_squared))
}

# Calculate metrics for the test set
test_metrics <- calculate_metrics(test_data$price, test_predictions)

# Calculate metrics for the training set
train_metrics <- calculate_metrics(train_data$price, train_predictions)

# Create a comparison table for Train and Test metrics
metrics_table <- data.frame(
  Data = c("TRAINING", "TEST"),
  MAE = c(train_metrics$MAE, test_metrics$MAE),
  MSE = c(train_metrics$MSE, test_metrics$MSE),
  RMSE = c(train_metrics$RMSE, test_metrics$RMSE),
  R_squared = c(train_metrics$R_squared, test_metrics$R_squared)
)
```

```r
# Print the metrics table
print(metrics_table)

##       Data      MAE      MSE     RMSE R_squared
## 1 TRAINING 23.31121 1063.710 32.61456 0.6874650
## 2     TEST 22.89301 1028.309 32.06725 0.7058193

summary(df_LASSO$price) #For comparison

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    20.0    61.0    95.0   106.3   139.0   309.0

# Create a coefficients dataframe
coeff_df <- as.data.frame(as.matrix(coef(lasso_final_model, s =
best_lambda)))

# Add variable names
coeff_df$Variable <- rownames(coeff_df)

# Rename coefficient column

colnames(coeff_df)[1] <- "Coefficient"

# Filtrar solo coeficientes distintos de 0 y ordenar por importancia
coeff_df <- coeff_df %>%
  filter(Coefficient != 0) %>%
  arrange(desc(abs(Coefficient)))

# Print as a table
print(coeff_df, row.names = FALSE)

##  Coefficient                              Variable
##  60.60689815                           (Intercept)
##  25.90151384             room_type_Entire home/apt
##  19.68535539                          accommodates
##  16.12286604                             cluster_2
##  12.97601852                              bedrooms
##  12.38407522             Entire.serviced.apartment
##  -7.21203323                       shared_bathroom
##   5.67508194                            Dishwasher
##   5.56151562                            Essentials
##  -5.43753092               Long.term.stays.allowed
##  -5.05010819                    Free.street.parking
##   5.00698861                                 Dryer
##   4.66032687              Private.patio.or.balcony
##  -4.57341023                  Private.room.in.home
##   4.52287794                           Smoke.alarm
##  -4.37761149                          Refrigerator
##  -4.32866748                     number_of_reviews
##   4.28743004                             bathrooms
##   4.28432202                    host_response_time
##  -3.47576474                        minimum_nights
##   3.11883551                                  beds
##   3.09300551                Dishes.and.silverware
##   3.09011598                          Coffee.maker
##   2.92674839                 Location_Interaction
##  -2.83440038                           Board.games
##   2.83385839                Room.darkening.shades
##   2.51977080                 Free.washer...In.unit
##  -2.42883857                             cluster_5
##   2.33771395                          Entire.condo
##  -2.31692849                      Hot.water.kettle
##   2.23044313                                    TV
##  -2.21009574                           Entire.home
##  -2.17260876                               Bathtub
##   2.16062605              Cleanliness_Interaction
##  -2.13661599                     host_is_superhost
##  -2.05657755                    Value_Interaction
##   2.00170294                  Outdoor.dining.area
##   1.95064772                  Dedicated.workspace
##  -1.84469602                 Lock.on.bedroom.door
##  -1.82560209                          Pets.allowed
##  -1.65117898           Private.room.in.townhouse
##   1.56225923                             cluster_4
##  -1.35144413                               Shampoo
##  -1.15109503                                Coffee
##   1.10402954                       availability_30
##  -1.08472799                                  Oven
##  -0.99192516       calculated_host_listings_count
##   0.98439155             Private.room.in.rental.unit
##  -0.85923631                             Body.soap
##  -0.84663204            Extra.pillows.and.blankets
##   0.78699517                     Outdoor.furniture
##   0.72611521                        Cooking.basics
##   0.70829300            Books.and.reading.material
##  -0.69981075        Cleaning.available.during.stay
##   0.56940775       Private.backyard...Fully.fenced
##   0.43690960                               Toaster
##   0.30959439 Exterior.security.cameras.on.property
##   0.22731544                     Fire.extinguisher
##   0.22569242                       host_experience
##   0.08255518                          Wine.glasses

# Plot residualsDistribution
residuals <- test_predictions - test_data$price
ggplot(data.frame(residuals), aes(x = residuals)) +
  geom_histogram(bins = 30, fill = 'skyblue', color = 'black') +
  labs(title = "Residuals Distribution", x = "Residuals", y =
"Frequency")
```

```r
#Plot Predicted vs Actual Values
# Convert predictions to a vector
test_predictions <- as.vector(test_predictions)
# Create a dataframe with actual vs predicted values
LASSO_testresults_df <- data.frame(
  Actual = test_data$price,
  Predicted = test_predictions
)
# Plot Predicted vs Actual values
ggplot(LASSO_testresults_df, aes(x = Actual, y = Predicted)) +
  geom_point(color = 'blue') +   # Blue points
  geom_abline(slope = 1, intercept = 0, color = 'red', linetype =
'dashed') +  # Red reference line (ideal)
  labs(title = "Predicted vs Actual Values (Test Set)",
       x = "Actual Value",
       y = "Model Prediction") +
  theme_minimal()  # Minimalist style for the plot

y = train_data$price
y = log(train_data$price)
df_LASSO$host_response_time <-
as.numeric(as.factor(df_LASSO$host_response_time))

# 1. Split the data into training and testing sets
set.seed(321)  # For reproducibility
trainIndex <- createDataPartition(df_LASSO$price, p = 0.8, list = FALSE)
# Use 80% for training
train_data <- df_LASSO[trainIndex, ]
test_data <- df_LASSO[-trainIndex, ]

# 2. Fit the Lasso model with cross-validation (cv.glmnet optimizes
lambda)
lasso_model <- cv.glmnet(
  x = as.matrix(train_data %>% select(-price)),  # Exclude 'price'
  y = log(train_data$price),  # Use log(price)
  alpha = 1,
  standardize = TRUE,
  nfolds = 10,
  lambda.min.ratio = 0.0001
)

# 3. Predictions (exponentiate to return to original scale)
predictions <- predict(lasso_model, newx = as.matrix(test_data %>%
select(-price)), s = "lambda.min")
predicted_prices <- exp(predictions)  # Convert back to original price
scale

# 4. Compute error metrics
actual_prices <- test_data$price
```

```
rmse <- sqrt(mean((actual_prices - predicted_prices)^2))
cat("RMSE:", rmse, "\n")

## RMSE: 32.96232

# Standarized data only for decision tree methods
df_DT <- df_price_stdt
# Load necessary libraries
library(xgboost)

## Warning: package 'xgboost' was built under R version 4.4.3

##
## Adjuntando el paquete: 'xgboost'

## The following object is masked from 'package:dplyr':
##
##     slice

# GB DATASET
df_DT$Rating_Interaction <- df_DT$has_reviews *
df_DT$review_scores_rating
df_DT$Cleanliness_Interaction <- df_DT$has_reviews *
df_DT$review_scores_cleanliness
df_DT$Location_Interaction <- df_DT$has_reviews *
df_DT$review_scores_location
df_DT$Value_Interaction <- df_DT$has_reviews * df_DT$review_scores_value

# Keep only the necessary columns (the interactions)
df_DT <- df_DT %>%
  select(-has_reviews,
         -review_scores_rating,
         -review_scores_cleanliness,
         -review_scores_location,
         -review_scores_value)

# Cross-Validation Setup (4 folds)
train_control <- trainControl(
  method = "cv",        # Cross-validation
  number = 5,           # 5 folds for faster execution
  verboseIter = FALSE,   # To see the progress
  search = "grid"       # Grid search to find the best hyperparameters
)

# Hyperparameter grid for Grid Search with fewer combinations
tune_grid <- expand.grid(
  nrounds = c(25, 50),             # Reduced number of rounds
(iterations)
  max_depth = c(3, 5),             # Tree depth (reduced range)
  eta = c(0.05, 0.1),              # Learning rate
  gamma = c(0, 0.1),               # Regularization (minimum loss value)
```

```r
  colsample_bytree = c(0.8),        # Feature subsample (fixed value)
  min_child_weight = c(1, 5),       # Minimum child weight
  subsample = c(0.8)                # Row subsample (fixed value)
)

# Split the data into training (80%) and testing (20%)
set.seed(123)
trainIndex <- createDataPartition(df_DT$price, p = 0.8, list = FALSE)
df_train <- df_DT[trainIndex, ]
df_test <- df_DT[-trainIndex, ]

# Adjust the model using a smaller sample with cross-validation
set.seed(123)  # For reproducibility
df_train_sample <- df_train[sample(nrow(df_train), size = 1000), ]  # Use
a smaller sample

# Fit the model using cross-validation
xgb_model <- train(
  price ~ .,                # Formula to predict 'price'
  data = df_train_sample,   # Use the reduced training sample
  method = "xgbTree",       # XGBoost model
  trControl = train_control, # Cross-validation
  tuneGrid = tune_grid,     # Hyperparameter grid
  verbose = FALSE            # Show progress
)

## [12:59:13] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:13] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:13] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:13] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:13] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:13] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:14] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:14] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:14] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:14] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:14] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:15] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
```

```
## [12:59:15] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:15] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:15] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:16] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:16] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:16] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:16] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:16] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:17] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:17] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:17] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:17] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:17] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:18] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:18] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:18] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:18] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:18] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:19] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:19] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:19] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:19] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:19] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:20] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:20] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
```

```
## [12:59:20] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:20] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:21] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:21] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:21] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:21] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:21] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:21] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:22] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:22] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:22] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:22] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:22] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:22] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:23] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:23] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:23] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:23] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:24] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:24] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:24] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:24] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:24] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:24] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:25] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
```

```
## [12:59:25] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:25] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:25] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:25] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:26] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:26] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:26] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:26] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:26] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:27] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:27] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:27] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:27] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:27] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:27] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:28] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:28] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.
## [12:59:28] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is
deprecated, use `iteration_range` instead.

# Show the results of the hyperparameter search
print(xgb_model)

## eXtreme Gradient Boosting
##
## 1000 samples
##  107 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 800, 800, 800, 800, 800
## Resampling results across tuning parameters:
##
```

```
##   eta   max_depth  gamma  min_child_weight  nrounds  RMSE      
Rsquared 
##   0.05  3          0.0    1                 25       49.97927  
0.6406356
##   0.05  3          0.0    1                 50       37.58735  
0.6542664
##   0.05  3          0.0    5                 25       50.20676  
0.6373337
##   0.05  3          0.0    5                 50       37.35444  
0.6583666
##   0.05  3          0.1    1                 25       49.88822  
0.6364219
##   0.05  3          0.1    1                 50       37.59613  
0.6515637
##   0.05  3          0.1    5                 25       49.78957  
0.6421443
##   0.05  3          0.1    5                 50       37.41315  
0.6546089
##   0.05  5          0.0    1                 25       50.24341  
0.6433636
##   0.05  5          0.0    1                 50       37.07679  
0.6628180
##   0.05  5          0.0    5                 25       49.52613  
0.6600784
##   0.05  5          0.0    5                 50       36.64927  
0.6713694
##   0.05  5          0.1    1                 25       50.41085  
0.6497556
##   0.05  5          0.1    1                 50       37.08507  
0.6662720
##   0.05  5          0.1    5                 25       49.70398  
0.6568625
##   0.05  5          0.1    5                 50       36.75327  
0.6686977
##   0.10  3          0.0    1                 25       37.18600  
0.6571432
##   0.10  3          0.0    1                 50       34.91241  
0.6747521
##   0.10  3          0.0    5                 25       37.57941  
0.6461611
##   0.10  3          0.0    5                 50       35.47912  
0.6645734
##   0.10  3          0.1    1                 25       37.35712  
0.6529006
##   0.10  3          0.1    1                 50       35.19728  
0.6703508
##   0.10  3          0.1    5                 25       37.32678  
0.6527517
##   0.10  3          0.1    5                 50       35.06685  
0.6722192
```

```
##   0.10  5          0.0    1                  25        37.29786
0.6531814
##   0.10  5          0.0    1                  50        34.88874
0.6734514
##   0.10  5          0.0    5                  25        36.92704
0.6628192
##   0.10  5          0.0    5                  50        34.58539
0.6796936
##   0.10  5          0.1    1                  25        37.56748
0.6507088
##   0.10  5          0.1    1                  50        35.29935
0.6659051
##   0.10  5          0.1    5                  25        37.15741
0.6547282
##   0.10  5          0.1    5                  50        34.91820
0.6740532
##   MAE
##   34.22496
##   25.28240
##   34.27802
##   25.11712
##   34.09732
##   25.36972
##   34.14847
##   25.28592
##   34.58683
##   24.44576
##   34.12442
##   24.22225
##   34.78155
##   24.38933
##   34.37109
##   24.41082
##   25.09600
##   24.11957
##   25.35104
##   24.54296
##   25.31160
##   24.28926
##   25.24680
##   24.29280
##   24.86689
##   23.68956
##   24.44090
##   23.62162
##   24.89344
##   23.80818
##   24.76638
##   23.97392
##
```

```
## Tuning parameter 'colsample_bytree' was held constant at a value of
0.8
##
## Tuning parameter 'subsample' was held constant at a value of 0.8
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were nrounds = 50, max_depth = 5,
eta
##  = 0.1, gamma = 0, colsample_bytree = 0.8, min_child_weight = 5 and
subsample
##  = 0.8.
```

```
print(xgb_model$bestTune)  # Optimal hyperparameters
```

```
##    nrounds max_depth eta gamma colsample_bytree min_child_weight
subsample
## 28      50         5 0.1     0              0.8                5
0.8
```

```
# Predict on test and train data with the final model
train_preds <- predict(xgb_model, newdata = df_train)
test_preds <- predict(xgb_model, newdata = df_test)

# Calculate metrics for Train and Test
calc_metrics <- function(actual, predicted) {
  mae <- mean(abs(predicted - actual))
  mse <- mean((predicted - actual)^2)
  rmse <- sqrt(mse)
  rsq <- cor(predicted, actual)^2  # R²
  return(c(MAE = mae, MSE = mse, RMSE = rmse, Rsquared = rsq))
}

train_metrics <- calc_metrics(df_train$price, train_preds)
test_metrics <- calc_metrics(df_test$price, test_preds)

# Create a table with metrics
metrics_table <- data.frame(
  Dataset = c("Train", "Test"),
  MAE = c(train_metrics["MAE"], test_metrics["MAE"]),
  MSE = c(train_metrics["MSE"], test_metrics["MSE"]),
  RMSE = c(train_metrics["RMSE"], test_metrics["RMSE"]),
  Rsquared = c(train_metrics["Rsquared"], test_metrics["Rsquared"])
)

print(metrics_table)  # Display metrics
```

```
##   Dataset      MAE       MSE     RMSE  Rsquared
## 1   Train 15.67846  525.2571 22.91849 0.8529316
## 2    Test 23.53066 1171.3638 34.22519 0.6396223
```

```
# Predicted vs Actual plot (Test set)
ggplot(data = data.frame(Actual = df_test$price, Predicted = test_preds),
aes(x = Actual, y = Predicted)) +
  geom_point(color = "blue", alpha = 0.5) +
  geom_abline(intercept = 0, slope = 1, color = "red", linetype =
"dashed") +  # Ideal line
  theme_minimal() +
  labs(title = "Predicted vs Actual (Test Set)", x = "Actual Values", y =
"Predicted Values")

# Plot variable importance (Top 10)
importance <- xgb.importance(model = xgb_model$finalModel)
top_10_importance <- importance[1:10, ]
xgb.plot.importance(importance_matrix = top_10_importance)
```