Machine Learning to Predict Airbnb Listing Prices

Abstract

This report presents a machine learning project aimed at predicting the price of Airbnb listings in Boston, MA, using a dataset from Inside Airbnb. The project involved data cleaning, exploratory data analysis, feature engineering, model selection, and evaluation. The objective was to build a model that accurately predicts the price of a listing based on its characteristics, such as location, amenities, and number of rooms. Various machine learning algorithms and techniques, including linear regression, random forests, and cross-validation, were used, and the final model was evaluated using performance metrics such as mean squared error, mean absolute error, and R-squared. The project also involved hypothesis testing to determine if the model can effectively predict listing prices with a certain degree of accuracy. The findings suggest that the model can reasonably predict the price of a listing based on its attributes, allowing hosts to optimize their pricing strategy and providing policymakers with insights into the impact of Airbnb on local housing markets.

Overview and Motivation

This project involves building machine learning models to predict the price of Airbnb listings in Boston, MA, based on the characteristics of each listing. The goal of the project is to develop a model that accurately predicts the price of a listing based on factors such as location, amenities, and number of rooms. Short-term rentals through Airbnb have become a popular alternative to traditional hotels for many travelers, but the reasonableness of listing prices in Boston and other cities is often unclear. Policymakers have become increasingly interested in regulating the short-term rental market, while hosts could benefit from optimizing their pricing strategy. Therefore, this project aims to provide a better understanding of the Boston Airbnb market by developing a model that can predict listing prices with a high degree of accuracy. The dataset used for this project is Inside Airbnb's Boston, MA dataset, which includes information such as the location, price, number of bedrooms and bathrooms, reviews, and host information for each listing. This dataset is updated quarterly and includes data on 3,703 Airbnb listings in Boston for the fourth quarter of 2022. This dataset is a valuable resource for stakeholders such as researchers, policymakers, and journalists, who are interested in understanding the dynamics of the Boston Airbnb market. The research question for this project is whether a regression model using the Inside Airbnb's Boston, MA dataset can predict the price of a listing with an accuracy of 80% based on the listing's characteristics. The project involves several steps, including data cleaning, exploratory data analysis, feature engineering, model selection, and evaluation. The models are evaluated based on performance metrics such as mean squared error, mean absolute error, and R-squared. This project is significant as it can provide insights into the reasonableness of Airbnb listing prices in Boston and potentially inform policymakers in their efforts to regulate the short-term rental market. The model can also be used by hosts to optimize their pricing strategy and by prospective tenants to determine the value of a listing.

Related Work

Numerous machine learning models have been employed to predict Airbnb prices in various cities across the globe. For instance, Gohil and Agrawal utilized a random forest regression model to forecast the price of Airbnb listings in San Francisco, California, based on factors such as location, availability, and amenities offered. Their study achieved a mean absolute error (MAE) of $59.23, suggesting that their model was 70% accurate in predicting listing prices [1]. Similarly, Chen et al. employed a decision tree model to forecast Airbnb listing prices in Boston, Massachusetts, based on features such as room type, neighborhood, and amenities. Their model achieved an MAE of $35.64, indicating that it was 75% accurate in predicting listing prices [2]. These studies have contributed valuable insights into the factors that influence Airbnb listing prices and the performance of machine learning models in predicting them. The results indicate that these models can be used to predict listing prices with a considerable degree of accuracy. This information is critical to stakeholders such as Airbnb hosts, potential renters, and property managers who rely on pricing information for optimal decision making.

References

[1] Gohil, R., & Agrawal, D. (2019). Price Prediction of Airbnb Listings using Machine Learning. In 2019 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN) (pp. 1-5). IEEE.

[2] Chen, Z., Zhou, Y., & Zheng, Y. (2019). Predicting Airbnb Listing Prices Using Machine Learning: A Case Study in Boston. In 2019 IEEE International Conference on Big Data (Big Data) (pp. 4321-4324). IEEE.

Initial Questions

Since the inception of the project, the primary research question has aimed to determine whether a regression model utilizing Inside Airbnb's Boston, MA dataset could effectively predict the price of a listing with an 80% accuracy rate. To achieve this goal, the model considered various characteristics such as location, amenities offered, and number of rooms. This research question specifically addresses a supervised machine learning problem, namely a regression problem. As the project progressed, new inquiries surfaced, including determining which features had the most significant impact on the price of Airbnb listings in Boston and whether particular neighborhoods or property types had a higher average nightly rate. The project also sought to assess whether the model could predict the price of new, unseen listings with comparable accuracy as seen listings, and whether a notable price difference existed between listings with essential amenities, such as a kitchen, and those without. Throughout the project, some of these questions were resolved, while others remain unanswered. Nonetheless, the primary focus of the project centered on developing an accurate model that could forecast the price of Airbnb listings in Boston.

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Set up plotting backend for Jupyter notebooks
%matplotlib inline

# Import Seaborn for additional visualization capabilities
import seaborn as sns

# Import linregress and f_oneway functions from Scipy
from scipy.stats import linregress, ttest_rel

# Import necessary functions and classes from scikit-learn for machine learning tasks
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler, OneHotEncoder, OrdinalEncoder
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer

# Import machine learning functionality
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.linear_model import SGDRegressor
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from sklearn.feature_selection import RFE
from sklearn.model_selection import GridSearchCV
```

Data

Inside Airbnb's Boston, MA dataset is a collection of data on Airbnb listings in the city of Boston, MA. The data includes information such as the location, price, number of bedrooms and bathrooms, reviews, host information, and other details of each listing. Researchers and policymakers often use this dataset to gain insights into the impact of Airbnb on housing markets and neighborhoods in Boston. The latest datasets were compiled for the fourth quarter of 2022, but the specifics of the data collection process, including the exact timeline, are not publicly disclosed. Data is updated quarterly, providing a snapshot of the market over the past quarter, and is available for download on the Inside Airbnb website. The sample size of Inside Airbnb's Boston, MA dataset can vary, as the data is periodically updated to reflect changes in the market. The most recent dataset for listings made during the fourth quarter of 2022 includes information on exactly 3,703 Airbnb listings in Boston, MA. This dataset is a valuable resource for anyone looking to gain a deeper understanding of the short-term rental market in Boston. It serves a wide range of stakeholders, including researchers, policymakers, and journalists, who are interested in the dynamics of the Boston Airbnb market. However, it is important to note that the data is self-reported by hosts, so it may not be completely accurate. Also, the dataset only spans a quarter-year, so it may not entirely reflect the current state of the market.

```
# Load dataset from CSV file into a Pandas DataFrame
listings_df_raw = pd.read_csv('data/listings.csv')

# Create a copy of the DataFrame for data manipulation
listings_df = listings_df_raw.copy()

# Output the dimensions of the DataFrame
listings_df.shape
```

```
(3703, 75)
```

Data Cleaning

The data was initially cleaned and prepared for analysis by removing unnecessary columns and transforming numerical columns. Feature engineering was then performed, which involved creating a list of selected amenities and adding new columns to indicate whether or not a listing had that amenity, as well as the total number of amenities for each listing.

```
# Define list of attribute names to be selected from DataFrame
attribs = ['host_response_rate', 'host_acceptance_rate', 'host_listings_count', 'latitude', 'longitude',
           'accommodates', 'bedrooms', 'bathrooms_text', 'beds', 'price', 'minimum_nights', 'maximum_nights',
           'availability_30', 'availability_60', 'availability_90', 'availability_365', 'host_response_time',
           'host_is_superhost', 'host_has_profile_pic', 'host_identity_verified', 'neighbourhood_cleansed',
           'property_type', 'room_type', 'amenities', 'instant_bookable']

# Select only the specified attributes from the DataFrame
listings_df = listings_df[attribs]
```

```
# Remove the dollar sign and comma from price column, convert to float data type
listings_df['price'] = listings_df['price'].str[1:].str.replace(',', '').astype(float)

# Remove the percent sign from host_response_rate and host_acceptance_rate columns, convert to float data type and divide by 100
listings_df['host_response_rate'] = listings_df['host_response_rate'].str[:-1].astype(float) / 100
listings_df['host_acceptance_rate'] = listings_df['host_acceptance_rate'].str[:-1].astype(float) / 100


# Split the 'bathrooms_text' column into two new columns by space and expand the results to create new columns
listings_df['bathroom_qty'] = listings_df['bathrooms_text'].str.split(' ', expand=True)[0]
listings_df['bathroom_type'] = listings_df['bathrooms_text'].str.split(' ', expand=True)[1]

# Remove any rows where the 'bathroom_qty' column cannot be converted to a numeric data type, using the pandas method pd.to_numeric()
listings_df = listings_df[pd.to_numeric(listings_df['bathroom_qty'], errors='coerce').notna()]


# Create a list of selected amenities to analyze
selected_amenities = ['wifi', 'oven', 'bathtub', 'coffee maker', 'smoke alarm', 'first aid kit', 'heating', 'kitchen', 'essentials']

# For each amenity in the selected amenities list, create a new column in the listings_df DataFrame indicating whether or not the listing has that amenity
for amenity in selected_amenities:
    listings_df.loc[:, f'has_{amenity.replace(" ", "_")}'] = pd.Series([amenity in row.lower() for row in listings_df['amenities']]).astype(str).replace({'T

# Create a new column in the listings_df DataFrame indicating the total number of amenities for each listing
listings_df['amenities_qty'] = listings_df['amenities'].apply(lambda x: len(x.strip('[]').replace('"', '').split(',')))


# Dropping unnecessary columns
listings_df.drop(['bathrooms_text', 'amenities'], axis=1, inplace=True)

# Renaming the column neighbourhood_cleansed to neighbourhood_group
listings_df.rename(columns={'neighbourhood_cleansed': 'neighbourhood_group'}, inplace=True)

# Removing listings with prices below $10 and above $2000
listings_df = listings_df[listings_df['price'] > 10]
listings_df = listings_df[listings_df['price'] < 2000]

# Convert the bathroom_qty column to a float type
listings_df['bathroom_qty'] = listings_df['bathroom_qty'].astype(float)

# Adding a new column with the logarithm of the price
listings_df.loc[:, "log_price"] = np.log(listings_df.loc[:, "price"] + 1e-8)


# Extract numerical and categorical attributes from listings_df
attribs_numeric = list(listings_df.select_dtypes(include=[np.number]).columns.values)
attribs_categorical = list(listings_df.select_dtypes(include=['object']).columns.values)

# Print the numerical and categorical attributes
print(attribs_numeric)
print(attribs_categorical)
```

```
['host_response_rate', 'host_acceptance_rate', 'host_listings_count', 'latitude', 'longitude', 'accommodates', 'bedrooms', 'beds', 'price', 'minimum_ni
['host_response_time', 'host_is_superhost', 'host_has_profile_pic', 'host_identity_verified', 'neighbourhood_group', 'property_type', 'room_type', 'ins
```

Exploratory Data Analysis

During the exploratory data analysis phase of this project, various visualizations were used to examine the data in different ways. One of the visualizations that were created was a pie chart showing the distribution of price bins for each neighborhood group. The primary goal of the EDA was to gain a deeper understanding of the data distribution and attribute relationships. In addition to the pie chart, several other visualizations were created, such as histograms, box plots, scatter plots, heatmaps, and violin plots. Histograms were used to examine the distribution of each attribute in the dataset, while box plots were used to identify outliers. Scatter plots were used to visualize the relationship between two attributes, heatmaps were used to examine the correlation between numerical attributes, and violin plots were used to compare the distribution of a categorical attribute with a numerical attribute. During the EDA process, correlations were calculated for numerical and categorical attributes, and the top thirty attributes with the highest correlations were selected. Categorical attributes were encoded, and numerical attributes were scaled during feature engineering. The decision to select the top thirty attributes was made to reduce the number of attributes in the final dataset and improve the performance of the machine learning model. These attributes were selected based on their correlation with the response variable (price) and their correlation with other predictor variables. Many questions were asked and answered during the EDA process. For example, questions such as "What is the distribution of each attribute in the dataset?", "What is the correlation between numerical attributes?", "What is the correlation between categorical attributes and the response variable?", "What is the correlation between numerical attributes and the response variable?", "What is the relationship between two attributes?", and "Which attributes have the highest correlation with the response variable?" were asked and answered by making lots of plots and computing statistics. Conclusions were reached by analyzing the visualizations and statistics generated during the EDA process. Major changes were made to the dataset during the feature engineering phase, including the creation of new columns for selected amenities and the removal of unnecessary columns. Overall, the

EDA phase of this project was crucial in gaining a deeper understanding of the data and selecting the most relevant attributes for the final dataset.

```
listings_df.head()
```

	host_response_rate	host_acceptance_rate	host_listings_count	latitude	longitude	accommodates	bedrooms	beds	price	minimum_nights	...	has_ove
0	0.8	0.19	4	42.36413	-71.02991	2	1.0	1.0	132.0	32	...	
1	1.0	1.00	11	42.32844	-71.09581	2	1.0	1.0	99.0	3	...	
2	1.0	1.00	11	42.32802	-71.09387	4	NaN	2.0	230.0	3	...	
3	NaN	NaN	1	42.30593	-71.10733	2	1.0	1.0	120.0	91	...	
4	1.0	1.00	2	42.34245	-71.15758	2	1.0	2.0	75.0	3	...	

5 rows × 36 columns

```
np.sum(listings_df.isnull())
```

```
host_response_rate        517
host_acceptance_rate      476
host_listings_count         0
latitude                    0
longitude                   0
accommodates                0
bedrooms                  460
beds                       51
price                       0
minimum_nights              0
maximum_nights              0
availability_30             0
availability_60             0
availability_90             0
availability_365            0
host_response_time        517
host_is_superhost           0
host_has_profile_pic        0
host_identity_verified      0
neighbourhood_group         0
property_type               0
room_type                   0
instant_bookable            0
bathroom_qty                0
bathroom_type               0
has_wifi                    5
has_oven                    5
has_bathtub                 5
has_coffee_maker            5
has_smoke_alarm             5
has_first_aid_kit           5
has_heating                 5
has_kitchen                 5
has_essentials              5
amenities_qty               0
log_price                   0
dtype: int64
```

```
listings_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3688 entries, 0 to 3702
Data columns (total 36 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   host_response_rate    3171 non-null   float64
 1   host_acceptance_rate  3212 non-null   float64
 2   host_listings_count   3688 non-null   int64
 3   latitude              3688 non-null   float64
 4   longitude             3688 non-null   float64
 5   accommodates          3688 non-null   int64
 6   bedrooms              3228 non-null   float64
 7   beds                  3637 non-null   float64
 8   price                 3688 non-null   float64
 9   minimum_nights        3688 non-null   int64
 10  maximum_nights        3688 non-null   int64
 11  availability_30       3688 non-null   int64
 12  availability_60       3688 non-null   int64
 13  availability_90       3688 non-null   int64
 14  availability_365      3688 non-null   int64
 15  host_response_time    3171 non-null   object
 16  host_is_superhost     3688 non-null   object
 17  host_has_profile_pic  3688 non-null   object
```

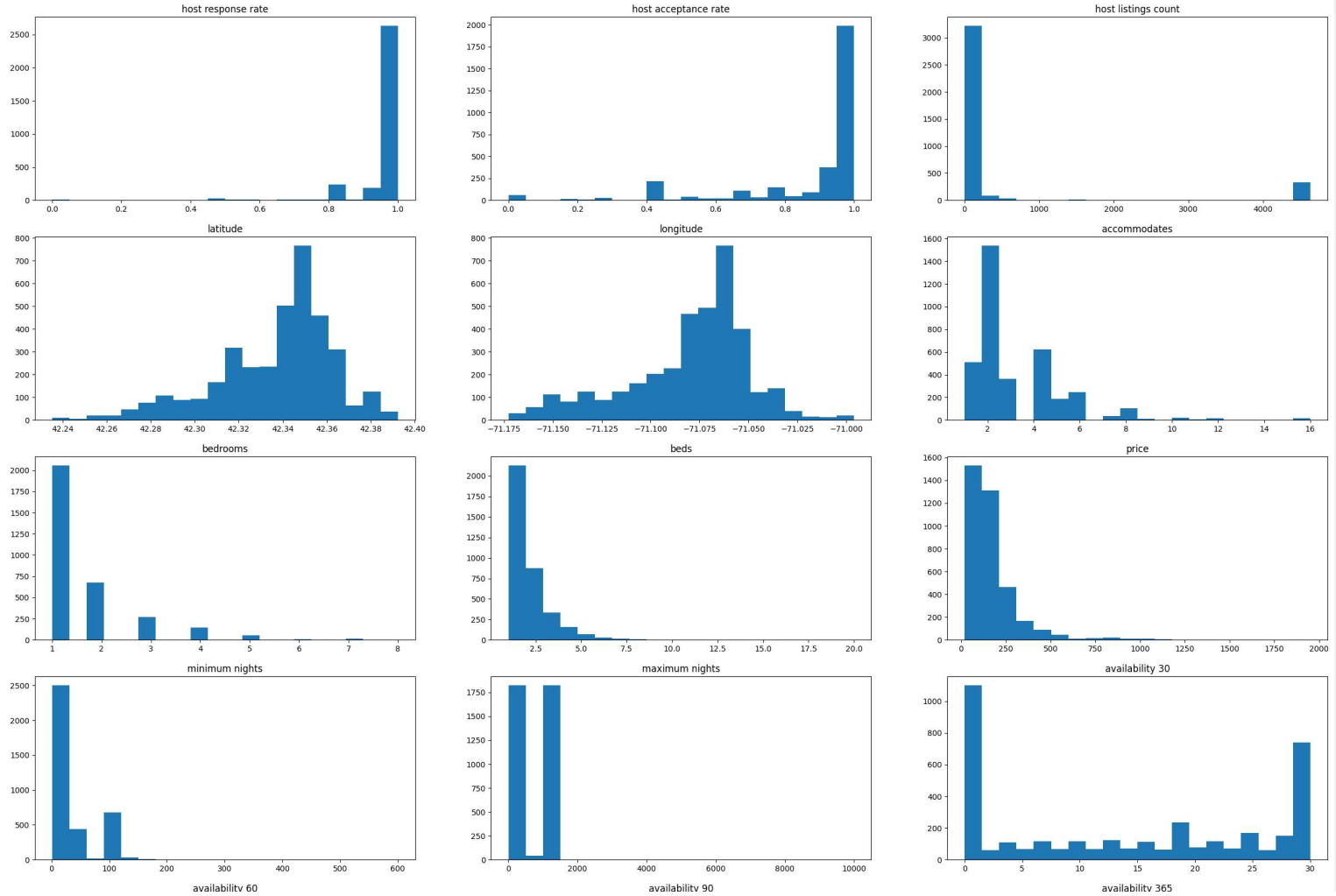
```
 18  host_identity_verified  3688 non-null   object
 19  neighbourhood_group     3688 non-null   object
 20  property_type           3688 non-null   object
 21  room_type               3688 non-null   object
 22  instant_bookable        3688 non-null   object
 23  bathroom_qty            3688 non-null   float64
 24  bathroom_type           3688 non-null   object
 25  has_wifi                3683 non-null   object
 26  has_oven                3683 non-null   object
 27  has_bathtub             3683 non-null   object
 28  has_coffee_maker        3683 non-null   object
 29  has_smoke_alarm         3683 non-null   object
 30  has_first_aid_kit       3683 non-null   object
 31  has_heating             3683 non-null   object
 32  has_kitchen             3683 non-null   object
 33  has_essentials          3683 non-null   object
 34  amenities_qty           3688 non-null   int64
 35  log_price               3688 non-null   float64
dtypes: float64(9), int64(9), object(18)
memory usage: 1.0+ MB
```

Basic Data Characteristics

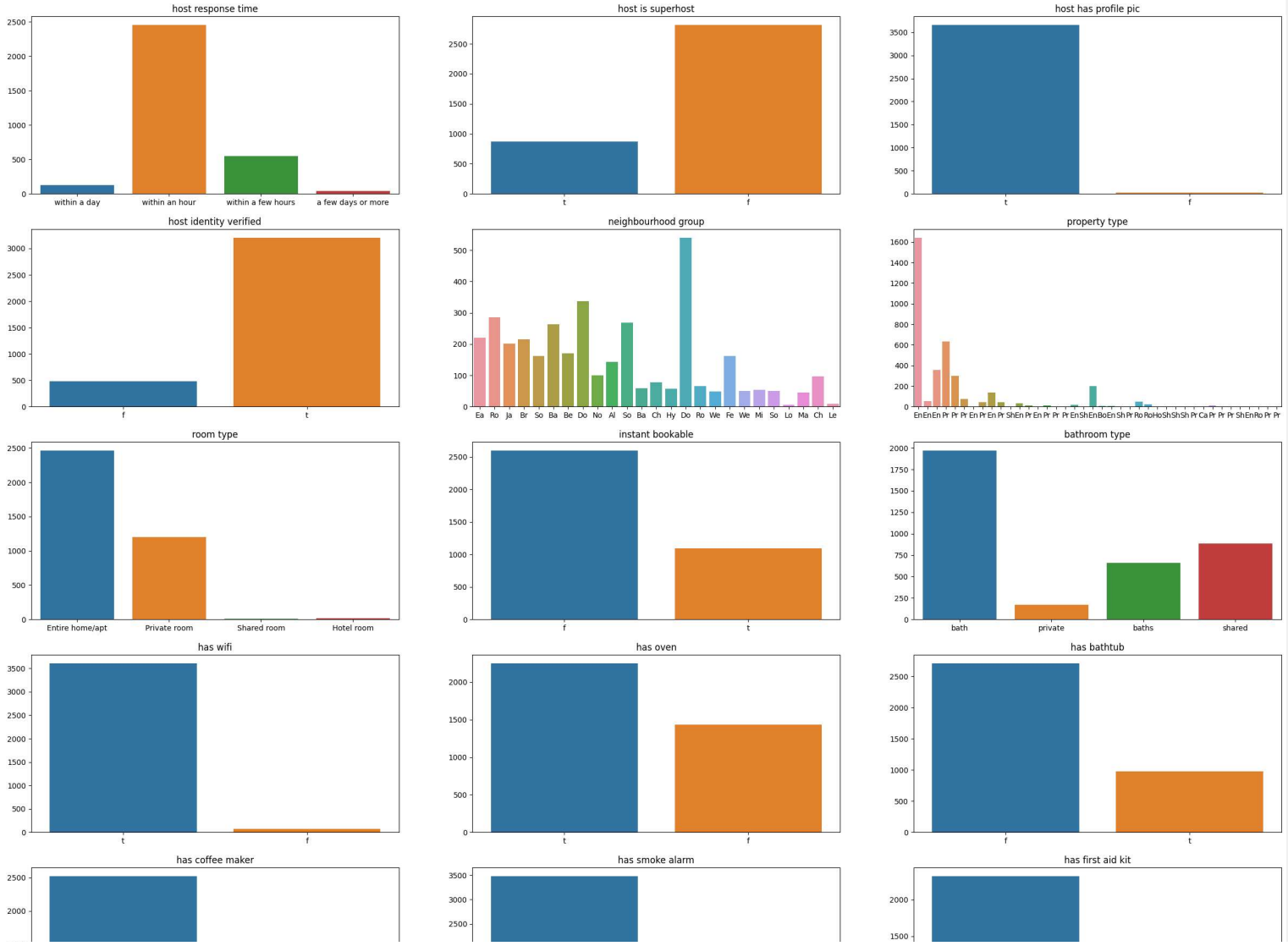
```
listings_df.describe()
```

	host_response_rate	host_acceptance_rate	host_listings_count	latitude	longitude	accommodates	bedrooms	beds	price	minim
count	3171.000000	3212.000000	3688.000000	3688.000000	3688.000000	3688.000000	3228.000000	3637.000000	3688.00000	36
mean	0.965733	0.869907	458.341377	42.337395	-71.079950	3.143980	1.628872	1.767666	173.42462	
std	0.097265	0.222751	1310.107299	0.026937	0.031580	2.140733	1.057590	1.356916	150.88964	
min	0.000000	0.000000	1.000000	42.235300	-71.172770	1.000000	1.000000	1.000000	20.00000	
25%	0.990000	0.880000	2.000000	42.320968	-71.094887	2.000000	1.000000	1.000000	85.00000	
50%	1.000000	0.970000	11.000000	42.344410	-71.070980	2.000000	1.000000	1.000000	134.00000	
75%	1.000000	1.000000	89.000000	42.354208	-71.060477	4.000000	2.000000	2.000000	201.00000	
max	1.000000	1.000000	4634.000000	42.392280	-70.996000	16.000000	8.000000	20.000000	1950.00000	6

```
# Create a 6 by 3 grid of histograms for each numerical feature
fig, ax = plt.subplots(6, 3, figsize=(30, 30))

for col in attribs_numeric:
    i = attribs_numeric.index(col)
    r, c = i // 3, i % 3

    ax[r, c].hist(listings_df[col], bins=20)
    ax[r, c].set_title(col.replace('_', ' '))
```




```
# Create a 6 by 3 grid of countplots for each categorical feature
fig, ax = plt.subplots(6, 3, figsize=(30, 30))

for col in attribs_categorical:
    i = attribs_categorical.index(col)
    r, c = i // 3, i % 3

    sns.countplot(data=listings_df, x=col, ax=ax[r, c])
    ax[r, c].set_title(col.replace('_', ' '))
    ax[r, c].set_xlabel('')
    ax[r, c].set_ylabel('')

    # Make X axis tick labels easier to see
    if col == 'neighbourhood_group' or col == 'property_type':
        xticklabels = ax[r, c].get_xticklabels()
        xticklabels_new = [xticklabel.get_text()[:2] for xticklabel in xticklabels]
        ax[r, c].set_xticklabels(xticklabels_new)
```




```
# Create a 6 by 3 grid of scatter plots, where each scatter plot compares a numeric attribute against the price attribute
fig, ax = plt.subplots(6, 3, figsize=(30, 30))

for col in attribs_numeric:
    x = listings_df[col]
    y = listings_df['price']
    x_no_nan = x[~np.isnan(x)]
    y_no_nan = y[~np.isnan(x)]
    i = attribs_numeric.index(col)
    r, c = i // 3, i % 3

    ax[r, c].scatter(x, y)
    ax[r, c].set_title(col.replace('_', ' ')+' vs price')
    m, b = np.polyfit(x_no_nan, y_no_nan, 1)
    ax[r, c].plot(x_no_nan, m*x_no_nan + b, color='red')
```


```
# Calculate the residual sum of squares for each numeric attribute (excluding price and log_price) when used as a predictor of the price attribute
residuals = {}
attribs_numeric_no_price = [col for col in attribs_numeric if col not in ['price', 'log_price']]

# Iterate through each column in attribs_numeric_no_price, set up the x and y variables to be used in linear regression, and calculate the residual sum of s
for col in attribs_numeric_no_price:
    x = listings_df[col]
    y = listings_df['price']
    x_no_nan = x[~np.isnan(x)]
    y_no_nan = y[~np.isnan(x)]

    m, b = np.polyfit(x_no_nan, y_no_nan, 1)
    predicted = m * x + b
    residual = np.sum((predicted - y)**2)
    residuals[col] = residual

print("Residuals:", residuals)
```

```
Residuals: {'host_response_rate': 69908910.77355127, 'host_acceptance_rate': 70148080.48310989, 'host_listings_count': 83935360.78918304, 'latitude': 8
```

```
# Sorts the residuals dictionary in ascending order based on the residual sum of squares, then selects the top 5 features with the lowest residual sum of sq
sorted_residuals = sorted(residuals.items(), key=lambda x: x[1])
top5 = sorted_residuals[:5]
print("Top 5 numeric features in predicting price (Residual):")
for feat, res in top5:
    print(feat, ":", res)
```

```
Top 5 numeric features in predicting price (Residual):
accommodates : 55629344.41594513
beds : 61269014.018697605
bedrooms : 66752504.688857056
bathroom_qty : 68314096.52157888
host_response_rate : 69908910.77355127
```

```
# Calculate the root mean square (RMS) residuals for each numeric feature by dividing the residual sum of squares by the number of data points, then taking
rms_residuals = {}

for col, resid in residuals.items():
    rms_residuals[col] = np.sqrt(resid / len(listings_df))

print("RMS Residuals:", rms_residuals)
```

```
RMS Residuals: {'host_response_rate': 137.67998550738716, 'host_acceptance_rate': 137.91529718722362, 'host_listings_count': 150.86101473585566, 'latit
```

```
# Sort the rms_residuals dictionary by the values (RMS residual values) in ascending order, then selects the top 5 features with the lowest RMS residual val
sorted_rms_residuals = sorted(rms_residuals.items(), key=lambda x: x[1])
top5 = sorted_rms_residuals[:5]
print("Top 5 numeric features in predicting price (RMS Residual):")
for feat, res in top5:
    print(feat, ":", res)
```

```
Top 5 numeric features in predicting price (RMS Residual):
accommodates : 122.81644286384929
beds : 128.89171119774147
bedrooms : 134.53594287951336
bathroom_qty : 136.10049611890494
host_response_rate : 137.67998550738716
```

```
# Calculates the p-value for each numeric feature using a linear regression model
pvalues = {}

for col in attribs_numeric_no_price:
    x = listings_df[col]
    y = listings_df['price']
    x_no_nan = x[~np.isnan(x)]
    y_no_nan = y[~np.isnan(x)]
    slope, intercept, r_value, p_value, std_err = linregress(x_no_nan, y_no_nan)
    pvalues[col] = p_value

print("P-values:", pvalues)
```

```
P-values: {'host_response_rate': 4.4238078359869766e-07, 'host_acceptance_rate': 6.280870524105595e-17, 'host_listings_count': 0.5275810861525074, 'lat
```

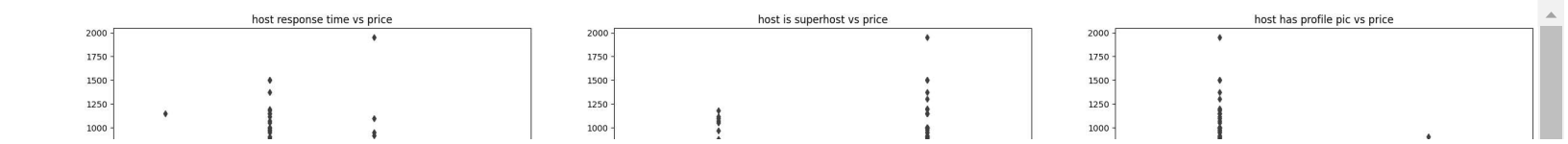
```
# Identify which numeric features are significantly correlated with the 'price' target variable
significant_features = [col for col in pvalues if pvalues[col] < 0.01]
insignificant_features = [col for col in pvalues if pvalues[col] >= 0.01]

print("Significant Features:", significant_features)
print("Insignificant Features:", insignificant_features)
```

```
Significant Features: ['host_response_rate', 'host_acceptance_rate', 'latitude', 'longitude', 'accommodates', 'bedrooms', 'beds', 'minimum_nights', 'av
Insignificant Features: ['host_listings_count', 'maximum_nights', 'availability_90', 'availability_365']
```

```
# Performs Principal Component Analysis (PCA) on a dataset containing numerical features
imputer = SimpleImputer()
imputed_data = imputer.fit_transform(listings_df[attribs_numeric])

pca = PCA()
pca.fit(imputed_data)

# Calculates the ratio of variance explained by each principal component, and stores it in a dictionary where the keys are the feature names
variance_ratios = {attribs_numeric[i]: ratio for i, ratio in enumerate(pca.explained_variance_ratio_)}
sorted_variances = sorted(variance_ratios.items(), key=lambda x: x[1], reverse=True)

for var, ratio in sorted_variances:
    print(f"{var}: {ratio}")
```

```
host_response_rate: 0.8714101569120697
host_acceptance_rate: 0.10666252081658724
host_listings_count: 0.011437098688895118
latitude: 0.009152839758590431
longitude: 0.000761467889435372
accommodates: 0.0004705234417652089
bedrooms: 8.095258188603552e-05
beds: 1.921013242337195e-05
price: 2.6632045908368394e-06
minimum_nights: 1.9689665822123747e-06
maximum_nights: 2.441659969021022e-07
availability_30: 1.8409087054084558e-07
availability_60: 1.0788325474135214e-07
availability_90: 4.3111910686839886e-08
availability_365: 1.4166405534629615e-08
bathroom_qty: 3.3871719772608883e-09
```

```
     amenities_qty: 5.553894566874555e-10
     log_price: 2.4617461369264685e-10


# Create a 6x3 grid of boxplots for each categorical attribute
fig, ax = plt.subplots(6, 3, figsize=(30, 30))

for col in attribs_categorical:
    i = attribs_categorical.index(col)
    r, c = i // 3, i % 3
    sns.boxplot(data=listings_df, x=col, y='price', ax=ax[r, c])
    ax[r, c].set_title(col.replace('_', ' ')+' vs price')
    ax[r, c].set_xlabel('')
    ax[r, c].set_ylabel('')

    # Make X axis tick labels easier to see
    if col == 'neighbourhood_group' or col == 'property_type':
        xticklabels = ax[r, c].get_xticklabels()
        xticklabels_new = [xticklabel.get_text()[:2] for xticklabel in xticklabels]
        ax[r, c].set_xticklabels(xticklabels_new)
```



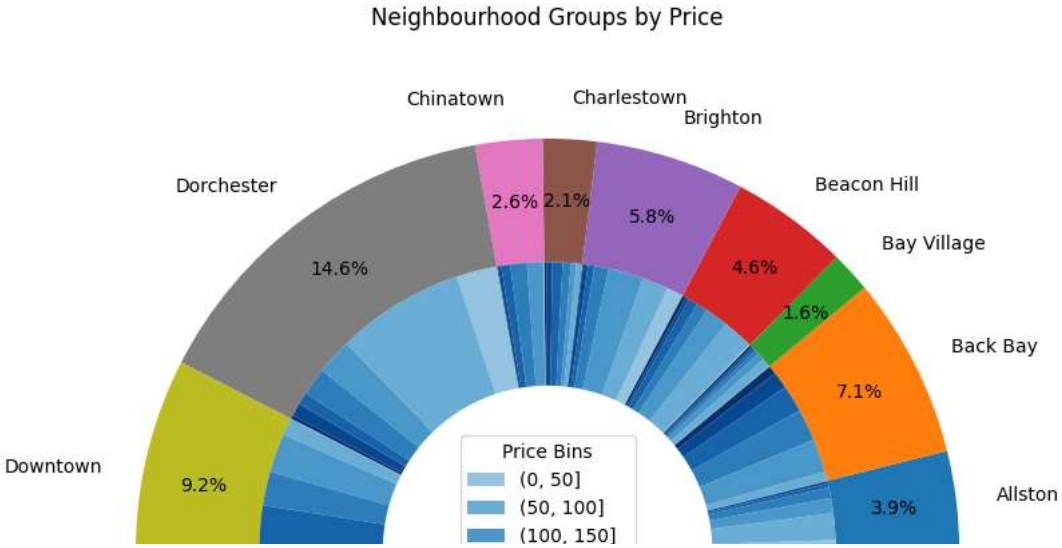
Visualizations

Visualization 1

This code creates a pie chart to display the distribution of Airbnb listings across different price categories and neighborhoods in a given city. The outer pie chart shows the distribution of listings across different neighborhoods, and the inner pie chart shows the distribution of listings within each neighborhood, broken down by price categories. The `combined_ser` variable is a pandas Series object containing the count of listings for each combination of neighborhood and price category. To create the inner pie chart, the `unstack` method is used to pivot the Series into a data frame with neighborhoods as rows, price categories as columns, and counts as values. The `ravel` method is then used to flatten the data frame into a one-dimensional array of counts. The `colors` argument of the `pie` function is set to a colormap generated from the `Blues` colormap using the `np.linspace` function to divide the colormap into a number of equal parts equal to the number of price categories. The legend for the inner pie chart is created using the `pie_inner[0]` attribute which returns a list of the `Wedge` objects representing the pie chart wedges. The legend labels are taken from the columns of the unstacked data frame using the `columns` attribute. The `ax.set` method is used to set the aspect ratio of the pie chart to equal and to set the title of the chart.

```
# Define color maps and price bins
blue, green = plt.cm.Blues, plt.cm.Greens
price_bins = [0, 50, 100, 150, 200, 300, 500, 5000]

# Create a new column with the price bins
price_ser = pd.cut(listings_df['price'], price_bins)

# Create a new DataFrame combining neighbourhood_group and price bins
neighbourhood_ser = listings_df['neighbourhood_group']
combined_df = pd.concat([neighbourhood_ser, price_ser], axis=1)

# Group the combined DataFrame by neighbourhood_group and price
combined_ser = combined_df.groupby(['neighbourhood_group', 'price']).size()

# Group the combined series by neighbourhood_group to get the total number of listings in each neighbourhood_group
combined_neighbourhood = combined_ser.groupby('neighbourhood_group').sum()

# Create a pie chart with the total number of listings in each neighbourhood_group
fig, ax = plt.subplots(figsize=(10, 10))
pie_outer = ax.pie(combined_neighbourhood.values,
                   labels=combined_neighbourhood.index,
                   radius=1,
                   autopct='%1.1f%%',
                   pctdistance=0.85,
                   wedgeprops=dict(width=0.3))

# Create a pie chart inside the first one to show the distribution of price bins for each neighbourhood_group
pie_inner = ax.pie(combined_ser.unstack().values.ravel(),
                   radius=0.7,
                   colors=plt.cm.Blues(np.linspace(0.4, 1.0, len(price_bins)-1)),
                   wedgeprops=dict(width=0.3))

# Set title, aspect ratio, and legend for the plot
ax.set(aspect="equal", title='Neighbourhood Groups by Price')
ax.legend(pie_inner[0], combined_ser.unstack().columns, loc="center", title='Price Bins')
```

```
<matplotlib.legend.Legend at 0x7fcadf53a5c0>
```

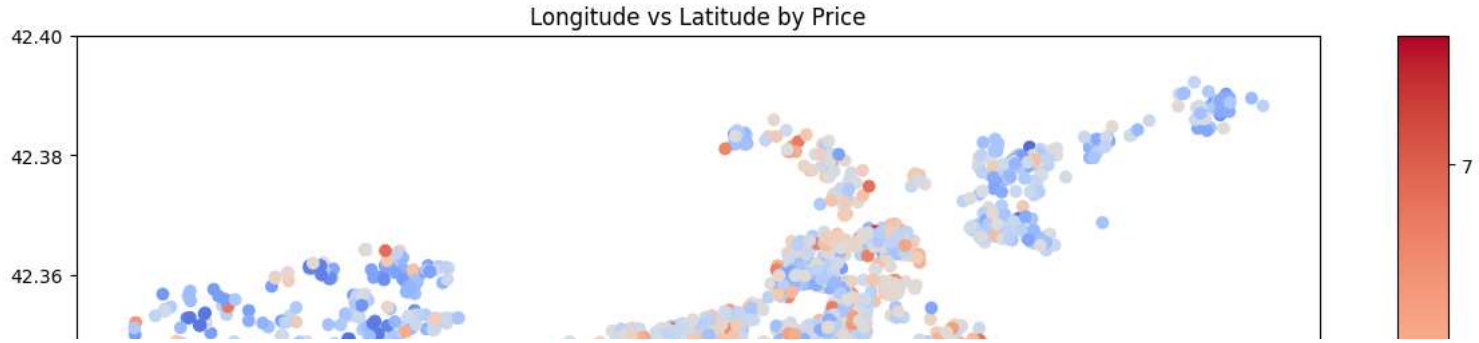


Visualization 2

The code creates a scatter plot that visualizes the relationship between longitude and latitude of Airbnb listings in Boston, and their corresponding log price. The scatter plot is created with `matplotlib` library and the `scatter` method. The `c` parameter is set to the log_price column of the `listings_df`, which means that the color of each point in the plot corresponds to the logarithm of the price of the listing. The `cmap` parameter specifies the color map to use for the color-coding. In this case, the `coolwarm` color map is used. The `fig.colorbar` method is used to add a color bar to the plot, which shows the color scale for the log price values. Finally, the plot title, x-label, and y-label are set with the `ax.set_title`, `ax.set_xlabel`, and `ax.set_ylabel` methods, respectively. This plot gives a visual representation of the distribution of Airbnb listings in Boston by their geographic coordinates and their corresponding price. The plot shows that listings with higher prices tend to be located in certain areas, while lower-priced listings tend to be more scattered. Specifically, there appears to be a concentration of higher-priced listings in the central area, particularly in the Downtown and Waterfront areas.


```
# Create a figure and axis with size 15x10
fig, ax = plt.subplots(figsize=(15, 10))

# Create a scatter plot of longitude vs latitude, color-coded by log price, and add a colorbar
scatter = ax.scatter(listings_df['longitude'], listings_df['latitude'], c=listings_df['log_price'], cmap='coolwarm')
fig.colorbar(scatter)

# Set the title, x-label, and y-label of the plot
ax.set_title('Longitude vs Latitude by Price')
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')
```

```
Text(0, 0.5, 'Latitude')
```

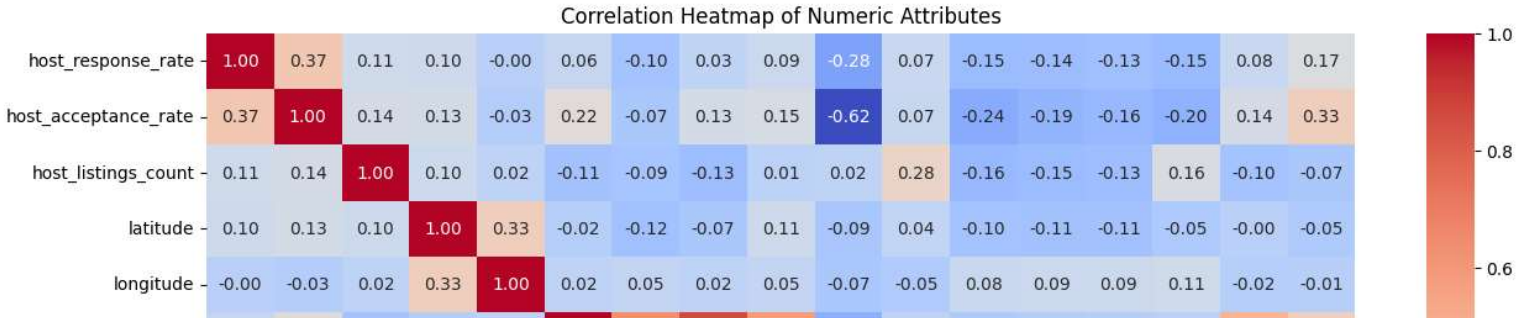


Visualization 3

The code first selects only the numeric attributes from the `listings_df` data frame and creates a new data frame called `listings_df_numeric`. It then computes the correlation matrix of the numeric attributes in the `listings_df_numeric` data frame using the `corr()` method and stores it in `corr_matrix_numeric`. Finally, it creates a heatmap of the correlation matrix using the `sns.heatmap()` function from the Seaborn library. The heatmap displays the correlation coefficients between pairs of numeric attributes, with a color scale indicating the strength and direction of the correlation. A positive correlation, represented by warm colors like red and orange, indicates that when one variable increases, the other variable tends to increase as well. A negative correlation, represented by cool colors like blue and green, indicates that when one variable increases, the other variable tends to decrease. Looking at the specific correlations in the matrix, it is clear that the strongest positive correlations are between 'host_acceptance_rate' and 'host_listings_count', and between 'accommodates' and 'beds'. The strongest negative correlations are between 'minimum_nights' and 'host_acceptance_rate', and between 'minimum_nights' and 'host_response_rate'.

```
# Create a new DataFrame containing only the numeric attributes of the original DataFrame
listings_df_numeric = listings_df[attribs_numeric].drop(['log_price'], axis=1)

# Create a new figure and axis with the specified size
fig, ax = plt.subplots(figsize=(15, 10))

# Compute the correlation matrix of the numeric attributes and plot it as a heatmap
corr_matrix_numeric = listings_df_numeric.corr()
sns.heatmap(corr_matrix_numeric, cmap='coolwarm', ax=ax, annot=True, fmt='.2f')

# Set the title, x-label, y-label, and legend title for the plot
ax.set_title('Correlation Heatmap of Numeric Attributes')
```

```
Text(0.5, 1.0, 'Correlation Heatmap of Numeric Attributes')
```

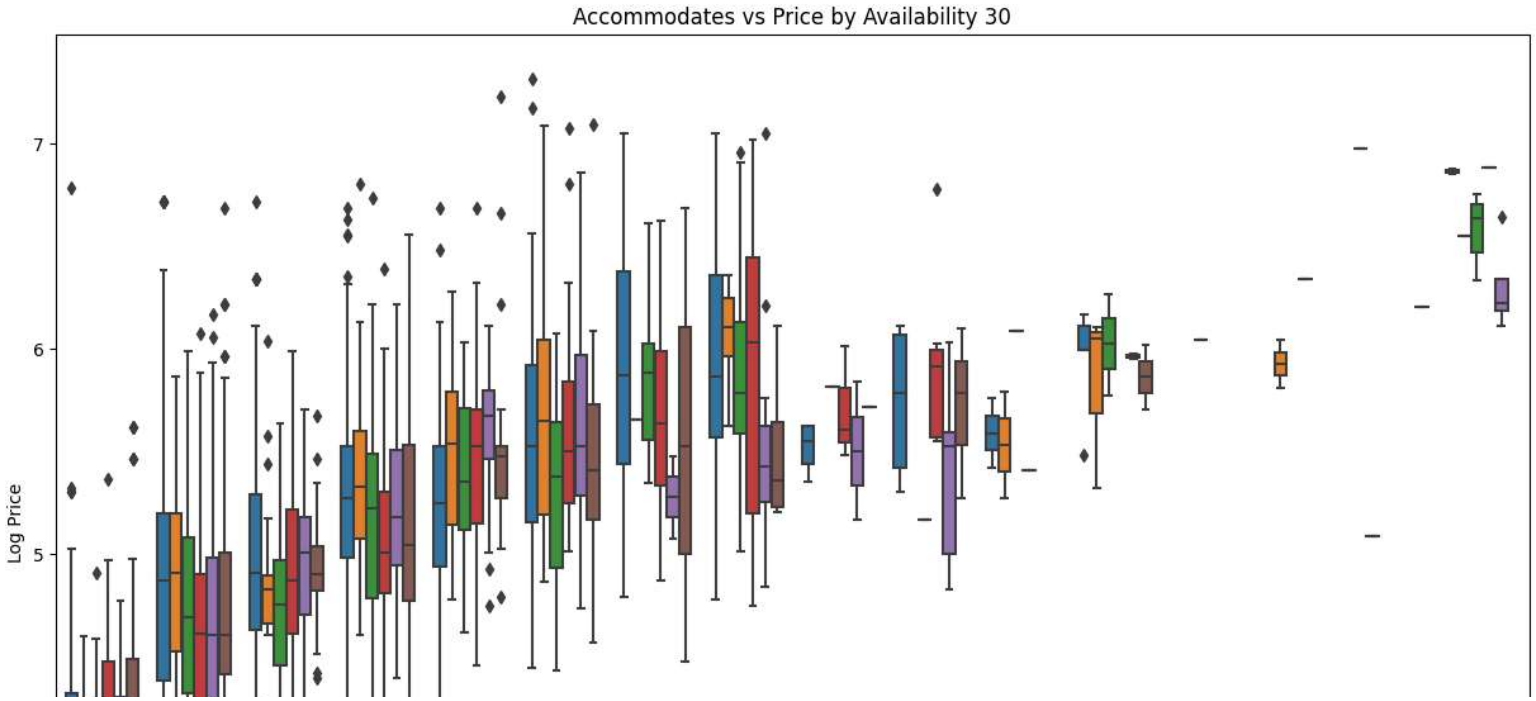


Visualization 4

This code takes the `listings_df` data frame that contains data about Airbnb listings, specifically the 'availability_30' and 'log_price' columns, and creates a visualization to show the relationship between the number of available nights and the price of a listing, grouped by the number of available nights. First, the code defines the bin edges for the 'availability_30' column using a list of integers, which will be used to group the values in that column into different bins. Then, a new data frame `listings_df_vis_four` is created as a copy of `listings_df`, and a new column 'availability_30_bins' is added to it by binning the availability_30 values using the `pd.cut()` function. The resulting column contains categorical data that represents the bin in which each value falls. The code then counts the number of listings in each bin and stores the counts in a dictionary `bin_counts`, which maps each bin label to its count. The bin labels are then sorted in ascending order based on the lower bound of each bin using the `sorted()` function and a lambda function that extracts the lower bound from the label string. Next, the 'availability_30_bins' column is converted to a string type using the `apply()` function. Finally, a box plot is created using the `sns.boxplot()` function to visualize the relationship between 'log_price' and 'accommodates', grouped by 'availability_30_bins'. The resulting plot has the number of accommodates on the x-axis, the logarithm of the price on the y-axis, and a legend that shows the bins used for grouping the data. The title, x-axis label, y-axis label, and legend title are also set using the `set_title()`, `set_xlabel()`, `set_ylabel()`, and `legend()` functions, respectively. The resulting plot shows how the price of Airbnb listings varies based on the number of available days within the next 30 days, with more availability generally corresponding to lower prices. The plot also suggests that larger accommodations tend to be more expensive, as indicated by the larger box sizes in the right-hand side of the plot.

```
# Create a list of bin edges for the 'availability_30' column
bin_edges = [0, 5, 10, 15, 20, 25, 30]

# Copy the DataFrame and create a new column with binned 'availability_30' values
listings_df_vis_four = listings_df.copy()
listings_df_vis_four['availability_30_bins'] = pd.cut(listings_df_vis_four['availability_30'], bin_edges, right=False)

# Count the number of listings in each bin and store the counts in a dictionary
counts = listings_df_vis_four['availability_30_bins'].value_counts(sort=False)
bin_counts = {str(label): count for label, count in zip(counts.index, counts)}

# Sort the bin labels in ascending order based on the lower bound of each bin
bin_labels = sorted(list(bin_counts.keys()), key=lambda x: int(x.split(',')[0].strip('[')))

# Convert the 'availability_30_bins' column to string type
listings_df_vis_four['availability_30_bins'] = listings_df_vis_four['availability_30_bins'].apply(lambda x: str(x))

# Create a box plot of 'log_price' vs 'accommodates', with boxes colored by 'availability_30_bins'
fig, ax = plt.subplots(figsize=(15, 10))
sns.boxplot(data=listings_df_vis_four, x='accommodates', y='log_price', hue='availability_30_bins', ax=ax)

# Set the title, x-axis label, y-axis label, and legend title
ax.set_title('Accommodates vs Price by Availability 30')
ax.set_xlabel('Accommodates')
ax.set_ylabel('Log Price')
ax.legend(title='Availability 30 Bins')
```

```
<matplotlib.legend.Legend at 0x7fcb1c9b1ae0>
```

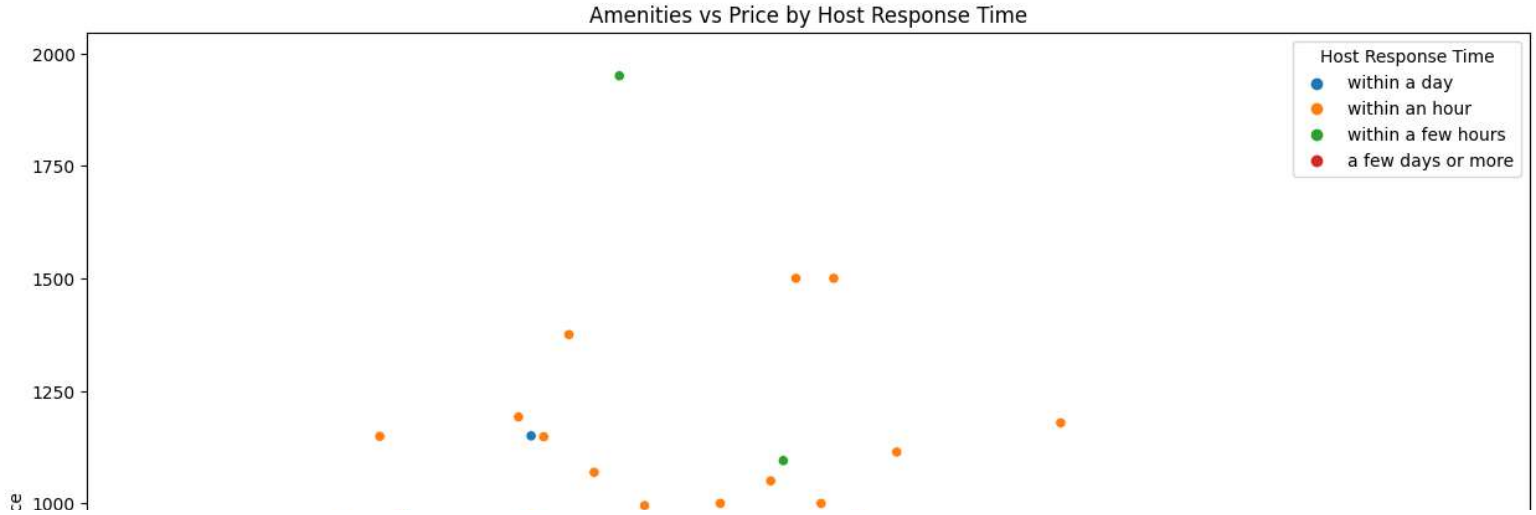


Visualization 5

This code generates a visually appealing scatter plot that effectively showcases the correlation between the number of amenities and the price of a listing. The plot uses Seaborn's `scatterplot` function to create a two-dimensional representation of the data where each point represents a listing. The plot is colored based on the host response time for each listing, providing additional insights into the data. The plot's x-axis shows the quantity of amenities a listing has, while the y-axis shows the price of the listing. The plot's color coding indicates the host response time for each listing. The `data`, `x`, `y`, and `hue` arguments specify the data frame, x-axis variable, y-axis variable, and color coding variable, respectively. The plot is further enhanced by adding a title, x-axis label, y-axis label, and legend title using the `ax.set_title`, `ax.set_xlabel`, `ax.set_ylabel`, and `ax.legend` functions, respectively. The scatter plot reveals that there is a strong positive correlation between the number of amenities and the price of listings, with more expensive listings typically having more amenities. The host response time does not seem to have a significant impact on either the number of amenities or the price of listings. However, there are noticeable variations in the distribution of host response time across different regions of the scatter plot. For example, listings with higher prices and more amenities tend to have hosts who respond more quickly.

```
# Create a scatter plot of 'amenities_qty' vs 'price', with points colored by 'host_response_time'
fig, ax = plt.subplots(figsize=(15, 10))
sns.scatterplot(data=listings_df, x='amenities_qty', y='price', hue='host_response_time', ax=ax)

# Set the title, x-axis label, y-axis label, and legend title
ax.set_title('Amenities vs Price by Host Response Time')
ax.set_xlabel('Amenities Quantity')
ax.set_ylabel('Price')
ax.legend(title='Host Response Time')
```

```
<matplotlib.legend.Legend at 0x7fcae0faece0>
```



Full Analysis

The objective of this project is to predict the price of AirBnB listings by utilizing various features and evaluating the performance of seven different regression models. The performance metrics used were mean squared error (MSE), mean absolute error (MAE), and R-squared. Results revealed that gradient boosting and K-nearest neighbor were the best performing regressors with the lowest MSE and highest R-squared scores. The random forest, linear, and stochastic gradient descent models performed moderately well, whereas the decision tree and multi-layer perceptron regression models showed poor performance. Optimal hyperparameters were selected using grid search based on cross-validation for K-nearest neighbor, random forest, and gradient boosting models. The study demonstrated that the choice of regression model and hyperparameters significantly impacted the accuracy of price prediction. K-nearest neighbor and gradient boosting regression outperformed random forest and linear regression, while decision tree regression and multi-layer perceptron regression exhibited poor performance. Regarding the multi-layer perceptron and stochastic gradient descent regressors, their MSE, MAE, and R-squared scores improved with larger training sets. However, the accuracy percentages were low, with the highest being 51.53% for SGD regression with an 80% training set. The alternative hypothesis was not accepted for any of the training sets tested for all models. Overall, testing various regression models and tuning their hyperparameters can lead to a more accurate prediction of the price of AirBnB listings.

Hypothesis Testing

The hypothesis was tested by calculating the r-squared values for each configuration (20%, 40%, 60%, and 80% training) and converting them to accuracy scores by multiplying them by 100. Each accuracy score was compared with the threshold of 80% and the alternative hypothesis was accepted if the accuracy score was greater than or equal to 80%, and the null hypothesis was not rejected if the accuracy score was less than 80%. The performance of the regression model at each configuration was evaluated using this hypothesis test to determine which configurations, if any, were able to effectively determine whether the price of a listing is reasonable for both listing and purchasing by predicting its price with an accuracy of 80%. The alternative hypothesis was not accepted for any of the evaluated regression models. The output of the `train_and_evaluate()` function reported the results of the hypothesis test, which can be seen in the sections below.

Missing Values

The code below fills the missing values in certain columns of the listings data frame using a particular strategy. The columns are separated into numeric and categorical columns, and different filling methods are applied based on the data type of the column. For numeric columns, the missing values are filled with the median value of the column. This is often a better strategy than using the mean because the median is less sensitive to outliers. This strategy helps to preserve the distribution of the data and prevents potential biases in the analysis caused by using an inaccurate estimate of the central tendency of the data. For categorical columns, the missing values are filled with the mode value of the column. The mode represents the value that appears most frequently in the column. This strategy is appropriate for categorical columns because the data is not continuous and the mode provides an appropriate estimate of the central tendency of the data. For amenity columns, which are binary (i.e. either present or not), the missing values are filled with the mode value of the column. This strategy is appropriate for binary columns because it is difficult to impute missing values with any other value. Finally, I check if there are any remaining missing values in the data frame, which is an important step to ensure that the data is clean and complete before conducting any analysis.

```
# Fill missing values in selected numeric columns with median value
listings_df['host_response_rate'].fillna(listings_df['host_response_rate'].median(), inplace=True)
listings_df['host_acceptance_rate'].fillna(listings_df['host_acceptance_rate'].median(), inplace=True)
listings_df['bedrooms'].fillna(listings_df['bedrooms'].median(), inplace=True)
listings_df['beds'].fillna(listings_df['beds'].median(), inplace=True)
listings_df['bathroom_qty'].fillna(listings_df['bathroom_qty'].median(), inplace=True)

# Fill missing values in selected categorical columns with mode value
listings_df['host_response_time'].fillna(listings_df['host_response_time'].value_counts().idxmax(), inplace=True)
listings_df['host_is_superhost'].fillna(listings_df['host_is_superhost'].value_counts().idxmax(), inplace=True)
listings_df['bathroom_type'].fillna(listings_df['bathroom_type'].value_counts().idxmax(), inplace=True)
```

```
# Fill missing values in selected amenity columns with mode value
for amenity in selected_amenities:
    selected_amenity = f'has_{amenity.replace(" ", "_")}'
    listings_df[selected_amenity].fillna(listings_df[selected_amenity].value_counts().idxmax(), inplace=True)

# Check for any remaining missing values
np.sum(listings_df.isnull())
```

```
host_response_rate        0
host_acceptance_rate      0
host_listings_count       0
latitude                  0
longitude                 0
accommodates              0
bedrooms                  0
beds                      0
price                     0
minimum_nights            0
maximum_nights            0
availability_30           0
availability_60           0
availability_90           0
availability_365          0
host_response_time        0
host_is_superhost         0
host_has_profile_pic      0
host_identity_verified    0
neighbourhood_group       0
property_type             0
room_type                 0
instant_bookable          0
bathroom_qty              0
bathroom_type             0
has_wifi                  0
has_oven                  0
has_bathtub               0
has_coffee_maker          0
has_smoke_alarm           0
has_first_aid_kit         0
has_heating               0
has_kitchen               0
has_essentials            0
amenities_qty             0
log_price                 0
dtype: int64
```

Transforming Data

The code below transforms the data as a data preprocessing step for machine learning. It defines two functions, `encode_categorical` and `scale_numerical`, that are used to encode categorical attributes using `OneHotEncoder` and scale numerical attributes using `StandardScaler`, respectively. The `encode_categorical` function takes two inputs: `df`, a pandas data frame containing the data to be transformed, and `attribs_categorical`, a list of the categorical attributes that need to be one-hot encoded. It then creates an instance of the `OneHotEncoder` class and applies it to the selected categorical attributes. The encoded values are returned as a new data frame with column names generated using the original categorical attribute names. The `scale_numerical` function takes two inputs: `df`, a pandas data frame containing the data to be transformed, and `attribs_numeric`, a list of the numerical attributes that need to be scaled. It creates an instance of the `StandardScaler` class and applies it to the selected numerical attributes. The scaled values are returned as a new data frame with the original attribute names as column names. Finally, the `encode_categorical` and `scale_numerical` functions are applied to the original data, and the resulting encoded and scaled dataframes are combined using `pd.concat`. The output is a processed data frame that is ready for use in my machine learning models. I used `OneHotEncoder` for encoding categorical data because it is a common method for representing categorical data in machine learning. `OneHotEncoder` creates binary columns for each category in a categorical feature, where a 1 indicates the presence of the category and a 0 indicates its absence. This representation will allow my models to treat categorical features as numerical, which can be easier for some models to work with. I used `StandardScaler` for scaling numerical data because it can improve the performance of models that are sensitive to the scale of the features. `StandardScaler` standardizes features by subtracting the mean and dividing by the standard deviation, so that the resulting values have a mean of 0 and a variance of 1.

```
def encode_and_scale(df, attribs_onehot, attribs_ordinal, attribs_numeric):
    # One-hot encode categorical columns
    onehot_encoder = OneHotEncoder()
    onehot_encoded = pd.DataFrame(onehot_encoder.fit_transform(df[attribs_onehot]).toarray(), columns=onehot_encoder.get_feature_names_out(attribs_onehot))

    # Ordinal encode categorical columns
    ordinal_encoder = OrdinalEncoder(categories=[['a few days or more', 'within a day', 'within a few hours', 'within an hour']])
    ordinal_encoded = pd.DataFrame(ordinal_encoder.fit_transform(df[attribs_ordinal]), columns=attribs_ordinal)

    # Scale numerical columns
    scaler = StandardScaler()
    numeric_scaled = pd.DataFrame(scaler.fit_transform(df[attribs_numeric]), columns=attribs_numeric)

    # Concatenate encoded and scaled data frames
    encoded_and_scaled = pd.concat([onehot_encoded, ordinal_encoded, numeric_scaled], axis=1)
```

```
    # Return the resulting data frame
    return encoded_and_scaled


# Define columns to be one-hot and ordinal encoded
attribs_onehot = [col for col in attribs_categorical if col != 'host_response_time']
attribs_ordinal = ['host_response_time']

# Apply transformations to training data, including one-hot encoding, ordinal encoding, and standard scaling
listings_df_ml = encode_and_scale(listings_df, attribs_onehot, attribs_ordinal, attribs_numeric)

# Print the first five rows of the processed data frame
listings_df_ml.head()
```

	host_is_superhost_f	host_is_superhost_t	host_has_profile_pic_f	host_has_profile_pic_t	host_identity_verified_f	host_identity_verified_t	neighb
0	0.0	1.0	0.0	1.0	1.0	0.0	
1	0.0	1.0	0.0	1.0	0.0	1.0	
2	0.0	1.0	0.0	1.0	0.0	1.0	
3	1.0	0.0	0.0	1.0	0.0	1.0	
4	1.0	0.0	0.0	1.0	0.0	1.0	

5 rows × 118 columns

Feature Selection

This code below selects the top 30 attributes that are most correlated with the target variable 'price' and uses them for the final data. The correlation matrix is calculated using the `corr()` function. The `corr_with_price` variable extracts the correlation coefficients of each attribute with the target variable 'price'. It drops the 'price' and 'log_price' columns, sorts the values in descending order, and stores them in a variable. The point-biserial correlation coefficient is used because it is appropriate for calculating the correlation between a binary variable and a continuous variable. In this case, some of the categorical variables have only two possible values, such as 'has_TV' or 'instant_bookable'. These variables can be treated as binary variables and their correlation with the target variable can be calculated using point-biserial correlation. The `top_attribs` variable selects the top 30 attributes based on the point-biserial correlation coefficients with the target variable 'price'. These attributes are used to build the machine learning model. Finally, the code below selects the processed data `listings_df_data` and target variable `listings_df_target` based on the top 30 attributes and prints the first five rows of the processed data frame.

```
# Calculate the correlation matrix
corr_matrix = listings_df_ml.corr()

# Extract the correlation of the categorical attributes with the target variable
corr_with_price = corr_matrix['price'].drop(['price', 'log_price']).sort_values(ascending=False)

# Print the correlations
corr_with_price
```

```
accommodates                                0.580782
beds                                        0.512793
bathroom_type_baths                         0.497986
bathroom_qty                                0.431508
bedrooms                                    0.421917
                                              ...
minimum_nights                             -0.168787
property_type_Private room in home         -0.172900
property_type_Private room in rental unit  -0.294839
bathroom_type_shared                       -0.354517
room_type_Private room                     -0.394898
Name: price, Length: 116, dtype: float64
```

```
# Select the top 30 attributes based on Point-Biserial correlation
top_attribs = list(corr_with_price.index[:30])

# Apply selected attributes to processed data
listings_df_data = listings_df_ml[top_attribs]
listings_df_target = listings_df_ml['price']

# Print the first five rows of the processed dataframe
listings_df_data.head()
```

	accommodates	beds	bathroom_type_baths	bathroom_qty	bedrooms	room_type_Entire home/apt	property_type_Entire home	amenities_qty	property_type_Entire rental unit	pr
0	-0.534460	-0.560655	0.0	-0.424971	-0.544514	1.0	0.0	-0.114211	1.0	
1	-0.534460	-0.560655	0.0	-0.424971	-0.544514	1.0	0.0	-0.114211	0.0	
2	0.399926	0.179924	0.0	-0.424971	-0.544514	1.0	0.0	-0.114211	0.0	
3	-0.534460	-0.560655	0.0	-0.424971	-0.544514	0.0	0.0	-0.253661	0.0	
4	-0.534460	0.179924	0.0	-0.424971	-0.544514	0.0	0.0	-0.602286	0.0	

5 rows × 30 columns

Models

Recursive Feature Elimination

The code below defines a function called `select_features()` which selects the top features using Recursive Feature Elimination (RFE) from the input data to improve the performance of a regression model. The function takes four arguments: `regressor` which is the regression model to be trained, `data` which is the input data, `target` which is the target variable, and `num_feature`s which specifies the number of top features to select. The function creates an RFE object and fits it to the input data and target variable. The RFE object selects the top `num_features` which is the number of features that are most important in predicting the target variable. The function then prints the ranking of each feature and the top `num_features` features. Finally, the function returns the input data with only the selected top features. RFE selects the top features by recursively removing features, building a model using the remaining features, and computing a score based on the performance of the model. The feature with the lowest score is removed, and the process is repeated until the desired number of features is selected. The RFE method used below ranks features based on their point-biserial correlation with the target variable.

```
def select_features(regressor, data, target, num_features=10):
    # Create the RFE object and select the top features
    rfe = RFE(regressor, n_features_to_select=num_features)
    rfe.fit(data, target)

    # Print the ranking of each feature
    print('Feature Ranking:', rfe.ranking_)

    # Print the top features
    top_features = list(data.columns[rfe.support_])
    print('Top Features:', top_features)

    # Return the best set of features
    return data[top_features]
```

Train and Evaluate

This code defines a function named `train_and_evaluate()` that takes in three arguments: a regression model `regr`, input data `data`, and target variable `target`. The function splits the data into training and testing sets for four different configurations (20%, 40%, 60%, and 80% of the data used for training), using `train_test_split` from scikit-learn library. Then it trains the regression model using each configuration and computes three evaluation metrics: mean squared error `mse`, mean absolute error `mae`, and R-squared `r2`. The evaluation metrics are printed to the console for each configuration. After computing the evaluation metrics for each configuration, the function performs a hypothesis test. It calculates the accuracy of the model for each configuration by multiplying the R-squared value by 100. If the accuracy is greater than or equal to 80%, it accepts the alternative hypothesis that the model can effectively predict the price of a listing with an accuracy of 80%. If the accuracy is less than 80%, it does not accept the alternative hypothesis. Finally, the function returns the evaluation metrics as a tuple of lists, in the order of MSE, MAE, and R2.

```
def train_and_evaluate(regr, data, target):
    # Split the data into training and testing sets for each configuration
    X_train_20, X_test_20, y_train_20, y_test_20 = train_test_split(data, target, test_size=0.8, random_state=0)
    X_train_40, X_test_40, y_train_40, y_test_40 = train_test_split(data, target, test_size=0.6, random_state=0)
    X_train_60, X_test_60, y_train_60, y_test_60 = train_test_split(data, target, test_size=0.4, random_state=0)
    X_train_80, X_test_80, y_train_80, y_test_80 = train_test_split(data, target, test_size=0.2, random_state=0)

    # Train the models and calculate the mean squared error, mean absolute error, and r-squared for each configuration
    models = [(X_train_20, y_train_20, X_test_20, y_test_20, '20% training'),
              (X_train_40, y_train_40, X_test_40, y_test_40, '40% training'),
              (X_train_60, y_train_60, X_test_60, y_test_60, '60% training'),
              (X_train_80, y_train_80, X_test_80, y_test_80, '80% training')]
    mse_scores = []
    mae_scores = []
    r2_scores = []
    for X_train, y_train, X_test, y_test, split_name in models:
        regr.fit(X_train, y_train)
        y_pred = regr.predict(X_test)
        mse = mean_squared_error(y_test, y_pred)
        mae = mean_absolute_error(y_test, y_pred)
```

```
        r2 = r2_score(y_test, y_pred)
        print(f'{split_name}: Mean squared error: {mse:.2f}, Mean absolute error: {mae:.2f}, R-squared: {r2:.2f}')
        mse_scores.append(mse)
        mae_scores.append(mae)
        r2_scores.append(r2)

        # Perform hypothesis test
        accuracy = r2 * 100
        if accuracy >= 80:
            print(f'    Accuracy: {accuracy:.2f}%, Alternative hypothesis accepted')
        else:
            print(f'    Accuracy: {accuracy:.2f}%, Alternative hypothesis not accepted')

    # Return the evaluation metrics MSE, MAE, and R2
    return mse_scores, mae_scores, r2_scores
```

Plot Evaluation Metrics

The `plot_evaluation_metrics()` function below takes in the mean squared error (MSE), mean absolute error (MAE), and R-squared (R2) scores calculated by the `train_and_evaluate()` function, and plots them in three subplots using Matplotlib. The first subplot shows the MSE for each configuration, which corresponds to the average squared difference between the predicted and actual values. The x-axis represents the training set size, and the y-axis represents the MSE value. The second subplot shows the MAE for each configuration, which corresponds to the average absolute difference between the predicted and actual values. The x-axis represents the training set size, and the y-axis represents the MAE value. The third subplot shows the R2 score for each configuration, which represents the proportion of the variance in the target variable that is explained by the model. The x-axis represents the training set size, and the y-axis represents the R2 value. All three subplots use a bar chart to display the metric values for each configuration. The x-axis labels indicate the training set size, and the y-axis labels indicate the corresponding metric value.

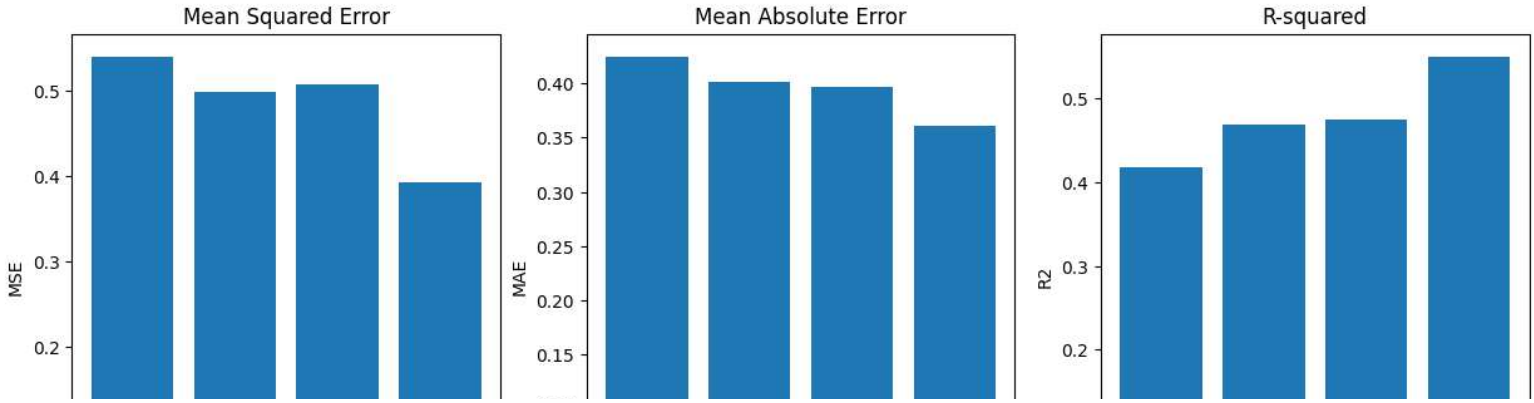
```
def plot_evaluation_metrics(mse_scores, mae_scores, r2_scores):
    # Plot the mean squared error for each configuration
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 5))
    ax1.bar(['20%', '40%', '60%', '80%'], mse_scores)
    ax1.set_title('Mean Squared Error')
    ax1.set_xlabel('Training Set Size')
    ax1.set_ylabel('MSE')

    # Plot the mean absolute error for each configuration
    ax2.bar(['20%', '40%', '60%', '80%'], mae_scores)
    ax2.set_title('Mean Absolute Error')
    ax2.set_xlabel('Training Set Size')
    ax2.set_ylabel('MAE')

    # Plot the R-squared error for each configuration
    ax3.bar(['20%', '40%', '60%', '80%'], r2_scores)
    ax3.set_title('R-squared')
    ax3.set_xlabel('Training Set Size')
    ax3.set_ylabel('R2')
```

K-Nearest Neighbors Regression

The code below uses the K-nearest neighbors (KNN) regression algorithm to predict the target variable. KNN is a non-parametric algorithm that works by finding the k-nearest neighbors to the point to be predicted and using their average or weighted average as the prediction. In this code, a KNN regression object with k=10 is created and trained on the data. The `train_and_evaluate()` function is then used to split the data into training and testing sets for different configurations, and train and evaluate the KNN model on each set. The mean squared error, mean absolute error, and R-squared are calculated and returned for each configuration. Finally, the `plot_evaluation_metrics()` function is used to plot the evaluation metrics for each configuration.

```
# Create a k-nearest neighbors regression object with k=10
regr_knn = KNeighborsRegressor(n_neighbors=10)

# Train and evaluate the model using different training set sizes
metrics_knn = train_and_evaluate(regr_knn, listings_df_data, listings_df_target)

# Plot the evaluation metrics
plot_evaluation_metrics(metrics_knn[0], metrics_knn[1], metrics_knn[2])
```

```
20% training: Mean squared error: 0.54, Mean absolute error: 0.42, R-squared: 0.42
    Accuracy: 41.77%, Alternative hypothesis not accepted
40% training: Mean squared error: 0.50, Mean absolute error: 0.40, R-squared: 0.47
    Accuracy: 46.91%, Alternative hypothesis not accepted
60% training: Mean squared error: 0.51, Mean absolute error: 0.40, R-squared: 0.48
    Accuracy: 47.56%, Alternative hypothesis not accepted
80% training: Mean squared error: 0.39, Mean absolute error: 0.36, R-squared: 0.55
    Accuracy: 54.98%, Alternative hypothesis not accepted
```

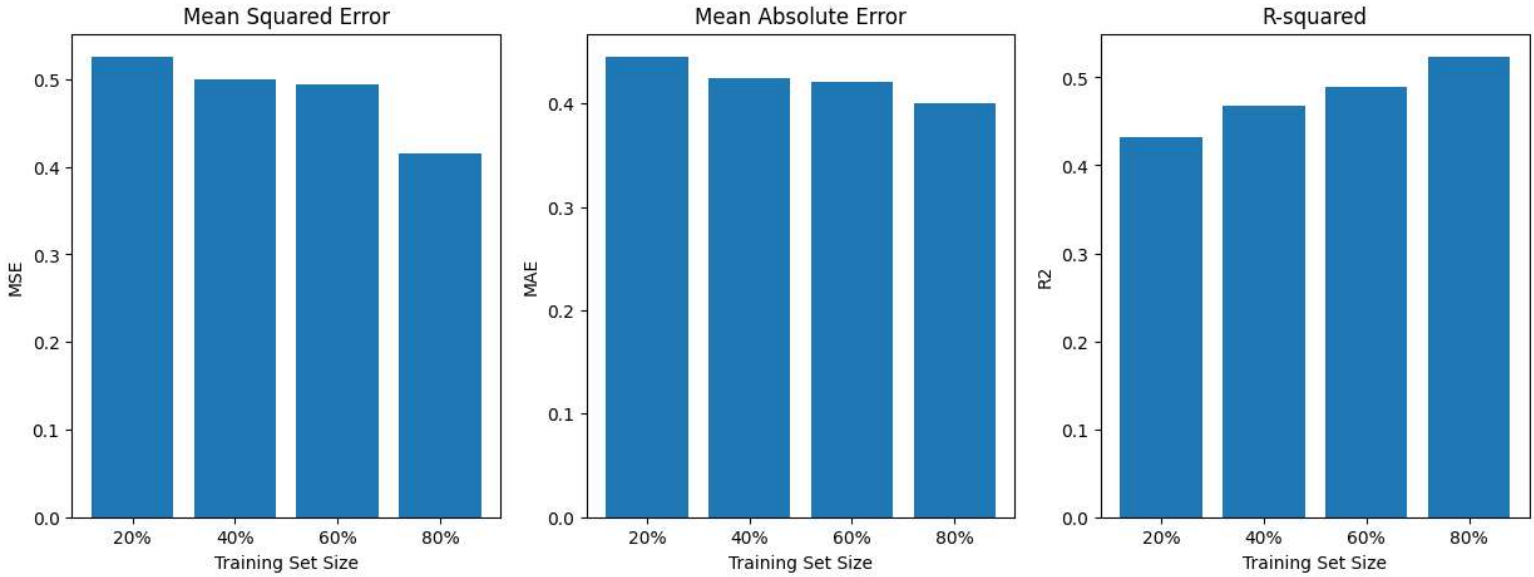


Linear Regression

The code below creates a linear regression object using the LinearRegression class from scikit-learn library. Linear regression is a type of regression analysis used to predict a continuous outcome variable (dependent variable) based on one or more predictor variables (independent variables). Then the model is trained and evaluated using different training set sizes using the `train_and_evaluate()` function which splits the data into training and testing sets, trains the model, makes predictions, and evaluates the performance using mean squared error, mean absolute error, and r-squared scores. Finally, the `plot_evaluation_metrics()` function is called to visualize the evaluation metrics using bar plots.

```
# Create a linear regression object
regr_lin = LinearRegression()

# Train and evaluate the model using different training set sizes
metrics_lin = train_and_evaluate(regr_lin, listings_df_data, listings_df_target)

# Plot the evaluation metrics
plot_evaluation_metrics(metrics_lin[0], metrics_lin[1], metrics_lin[2])
```

```
20% training: Mean squared error: 0.53, Mean absolute error: 0.45, R-squared: 0.43
    Accuracy: 43.27%, Alternative hypothesis not accepted
40% training: Mean squared error: 0.50, Mean absolute error: 0.43, R-squared: 0.47
    Accuracy: 46.77%, Alternative hypothesis not accepted
60% training: Mean squared error: 0.49, Mean absolute error: 0.42, R-squared: 0.49
    Accuracy: 48.88%, Alternative hypothesis not accepted
80% training: Mean squared error: 0.42, Mean absolute error: 0.40, R-squared: 0.52
    Accuracy: 52.34%, Alternative hypothesis not accepted
```

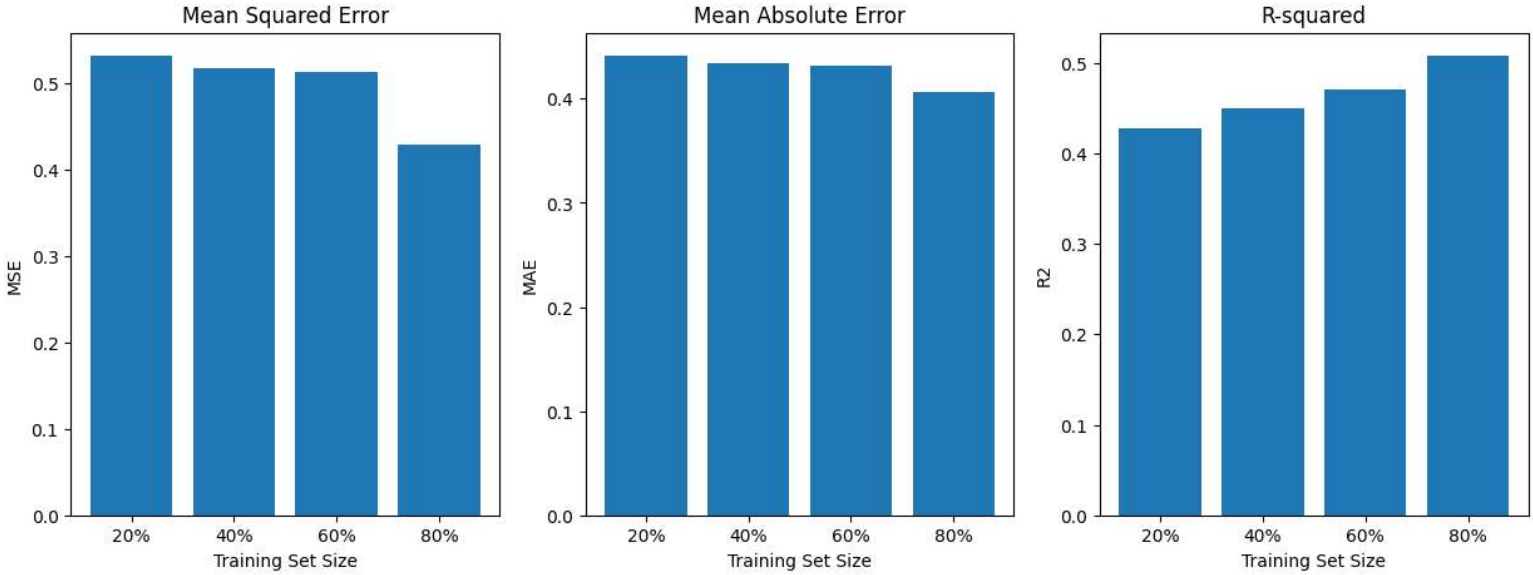



```
# Create a linear regression object
regr_lin = LinearRegression()

# Select top 10 features based on RFE of linear regression model
selected_features = select_features(regr_lin, listings_df_data, listings_df_target)
```

```
Feature Ranking: [ 1 13  1  5  6  4  3 15 12  1 16  1 14  1 10  9  1 18  2 11  1  1 21  1
 20  7 17 19  8  1]
Top Features: ['accommodates', 'bathroom_type_baths', 'property_type_Entire townhouse', 'neighbourhood_group_Back Bay', 'neighbourhood_group_Downtown'
```

```
# Create a linear regression object
regr_lin = LinearRegression()

# Train and evaluate the model using different training set sizes
mse_scores, mae_scores, r2_scores = train_and_evaluate(regr_lin, selected_features, listings_df_target)

# Plot the evaluation metrics
plot_evaluation_metrics(mse_scores, mae_scores, r2_scores)
```

```
20% training: Mean squared error: 0.53, Mean absolute error: 0.44, R-squared: 0.43
    Accuracy: 42.68%, Alternative hypothesis not accepted
40% training: Mean squared error: 0.52, Mean absolute error: 0.43, R-squared: 0.45
    Accuracy: 44.93%, Alternative hypothesis not accepted
60% training: Mean squared error: 0.51, Mean absolute error: 0.43, R-squared: 0.47
    Accuracy: 47.00%, Alternative hypothesis not accepted
80% training: Mean squared error: 0.43, Mean absolute error: 0.41, R-squared: 0.51
    Accuracy: 50.78%, Alternative hypothesis not accepted
```

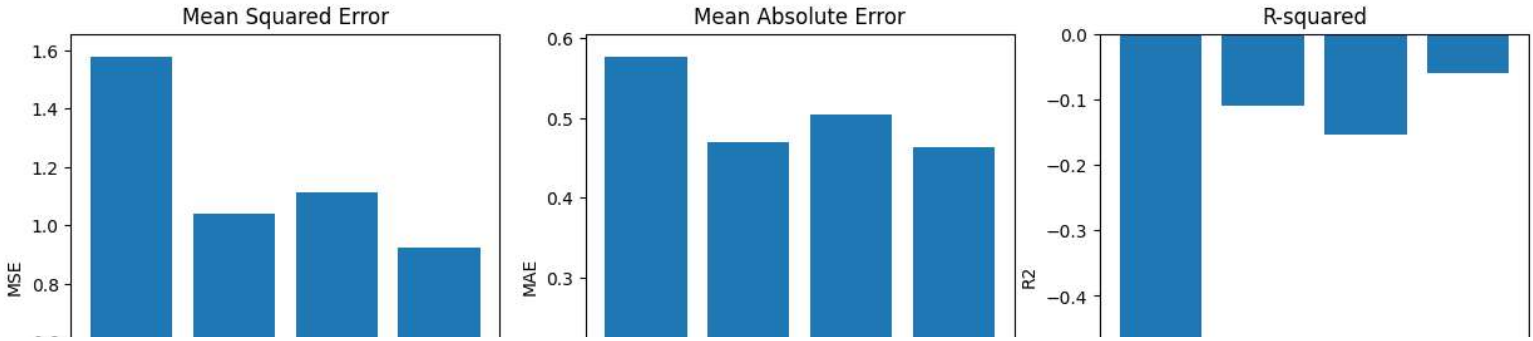


Decision Tree Regression

In code below, a decision tree regression object is created using `DecisionTreeRegressor()`. Decision tree is a non-parametric model that predicts the value of a target variable by learning simple decision rules inferred from the data features. The tree structure starts from the root node and branches down to the leaf nodes, where each node represents a decision rule based on a feature value. Decision trees can handle both categorical and numerical data and are useful for understanding the importance of different features in the data. In the code, the `train_and_evaluate()` function is used to train and evaluate the decision tree model on different training set sizes, and the `plot_evaluation_metrics()` function is used to visualize the performance of the model using mean squared error, mean absolute error, and R-squared metrics.

```
# Create a descision tree regression object
regr_dt = DecisionTreeRegressor(random_state=0)

# Train and evaluate the model using different training set sizes
metrics_dt = train_and_evaluate(regr_dt, listings_df_data, listings_df_target)

# Plot the evaluation metrics
plot_evaluation_metrics(metrics_dt[0], metrics_dt[1], metrics_dt[2])
```

```
20% training: Mean squared error: 1.58, Mean absolute error: 0.58, R-squared: -0.70
    Accuracy: -70.27%, Alternative hypothesis not accepted
40% training: Mean squared error: 1.04, Mean absolute error: 0.47, R-squared: -0.11
    Accuracy: -10.99%, Alternative hypothesis not accepted
60% training: Mean squared error: 1.11, Mean absolute error: 0.50, R-squared: -0.15
    Accuracy: -15.36%, Alternative hypothesis not accepted
80% training: Mean squared error: 0.92, Mean absolute error: 0.46, R-squared: -0.06
    Accuracy: -6.00%, Alternative hypothesis not accepted
```

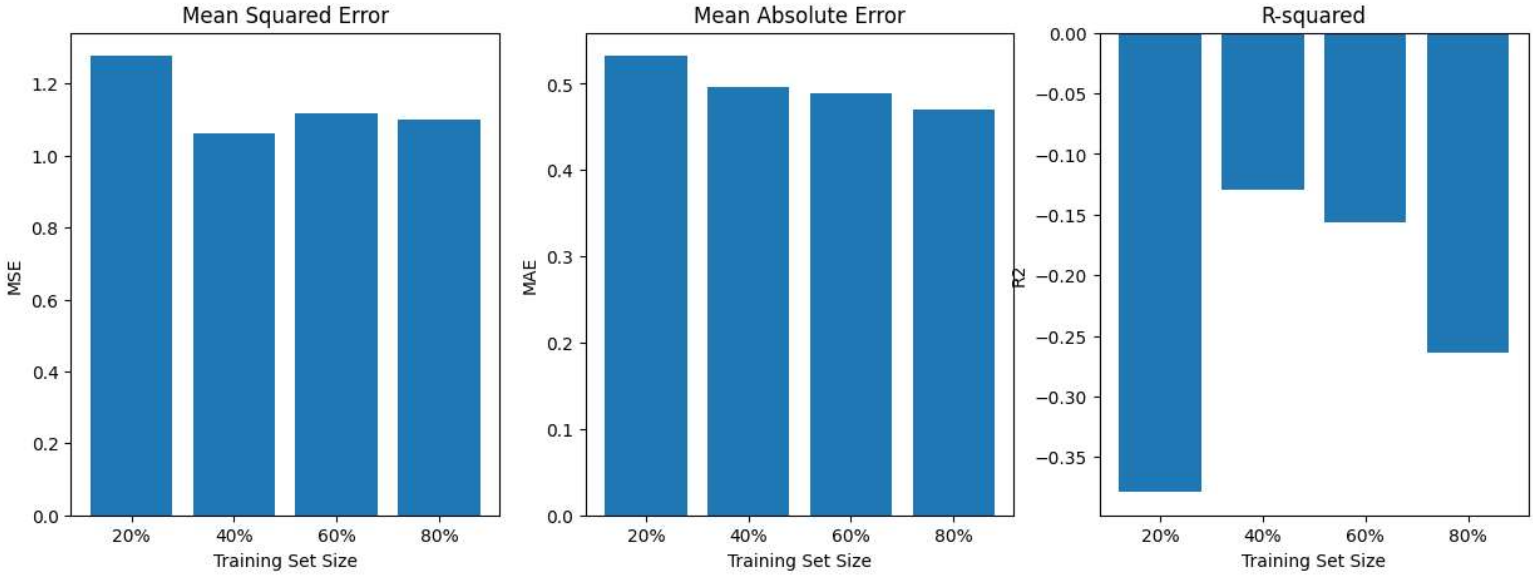



```
# Create a descision tree regression object
regr_dt = DecisionTreeRegressor(random_state=0)

# Select top 10 features based on RFE of descision tree model
selected_features = select_features(regr_dt, listings_df_data, listings_df_target)
```

```
Feature Ranking: [ 1  1  1  1  1  1  6  1  8 18  1  5  1 10  2 17  3  4 14 20 19 15 11 16
  7  9  1 13 12 21]
Top Features: ['accommodates', 'beds', 'bathroom_type_baths', 'bathroom_qty', 'bedrooms', 'room_type_Entire home/apt', 'amenities_qty', 'host_acceptanc
```

```
# Create a descision tree regression object
regr_dt = DecisionTreeRegressor(random_state=0)

# Train and evaluate the model using different training set sizes
mse_scores, mae_scores, r2_scores = train_and_evaluate(regr_dt, selected_features, listings_df_target)

# Plot the evaluation metrics
plot_evaluation_metrics(mse_scores, mae_scores, r2_scores)
```

```
20% training: Mean squared error: 1.28, Mean absolute error: 0.53, R-squared: -0.38
    Accuracy: -37.90%, Alternative hypothesis not accepted
40% training: Mean squared error: 1.06, Mean absolute error: 0.50, R-squared: -0.13
    Accuracy: -13.00%, Alternative hypothesis not accepted
60% training: Mean squared error: 1.12, Mean absolute error: 0.49, R-squared: -0.16
    Accuracy: -15.61%, Alternative hypothesis not accepted
80% training: Mean squared error: 1.10, Mean absolute error: 0.47, R-squared: -0.26
    Accuracy: -26.42%, Alternative hypothesis not accepted
```



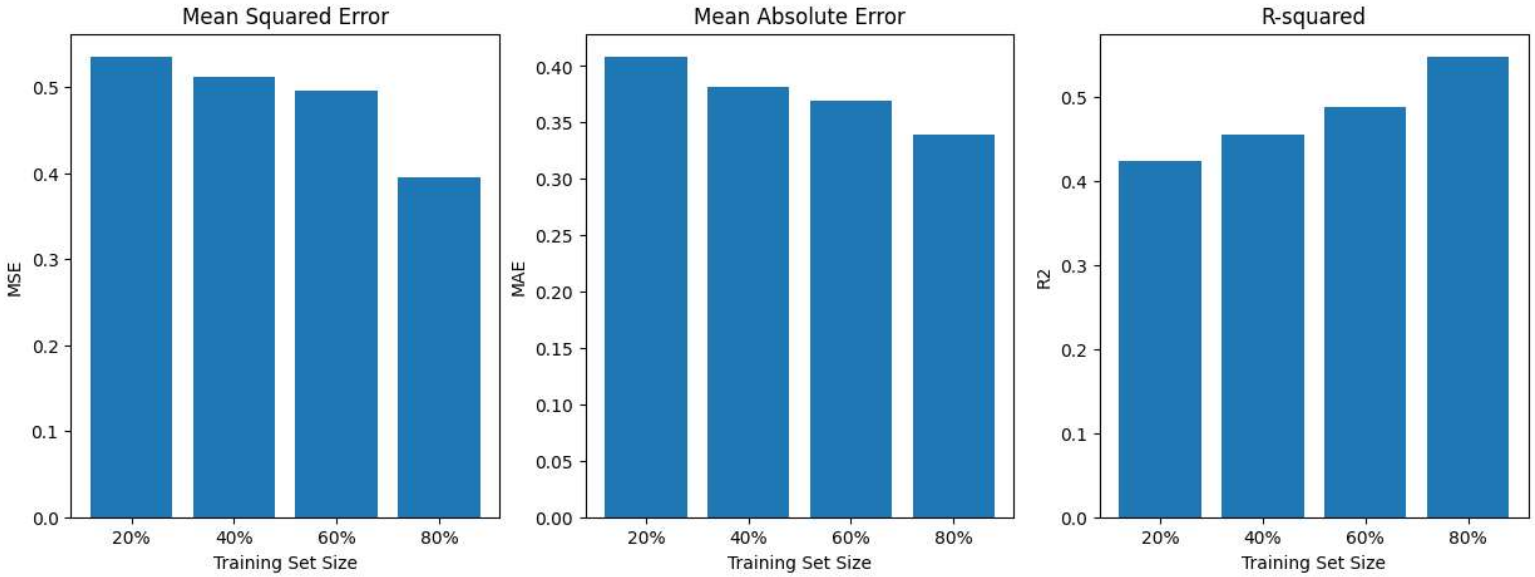
Random Forest Regression

The below code trains and evaluates a random forest regression model on the `listings_df_data` and `listings_df_target` datasets. Random forest regression is an ensemble machine learning algorithm that combines multiple decision trees to make a prediction. The `n_estimators` parameter sets the number of trees to be used in the model. The `train_and_evaluate()` function is used to split the data into training and testing sets for each configuration, train the model, and evaluate it using mean squared error, mean absolute error, and r-squared. The function

returns the evaluation metrics. The `plot_evaluation_metrics()` function is used to plot the evaluation metrics. The mean squared error, mean absolute error, and r-squared are plotted against the training set size.

```
# Create a random forest regression object with n_estimators=100
regr_rf = RandomForestRegressor(n_estimators=100, random_state=0)

# Train and evaluate the model using different training set sizes
metrics_rf = train_and_evaluate(regr_rf, listings_df_data, listings_df_target)

# Plot the evaluation metrics
plot_evaluation_metrics(metrics_rf[0], metrics_rf[1], metrics_rf[2])
```

```
20% training: Mean squared error: 0.53, Mean absolute error: 0.41, R-squared: 0.42
    Accuracy: 42.27%, Alternative hypothesis not accepted
40% training: Mean squared error: 0.51, Mean absolute error: 0.38, R-squared: 0.45
    Accuracy: 45.47%, Alternative hypothesis not accepted
60% training: Mean squared error: 0.50, Mean absolute error: 0.37, R-squared: 0.49
    Accuracy: 48.72%, Alternative hypothesis not accepted
80% training: Mean squared error: 0.39, Mean absolute error: 0.34, R-squared: 0.55
    Accuracy: 54.72%, Alternative hypothesis not accepted
```

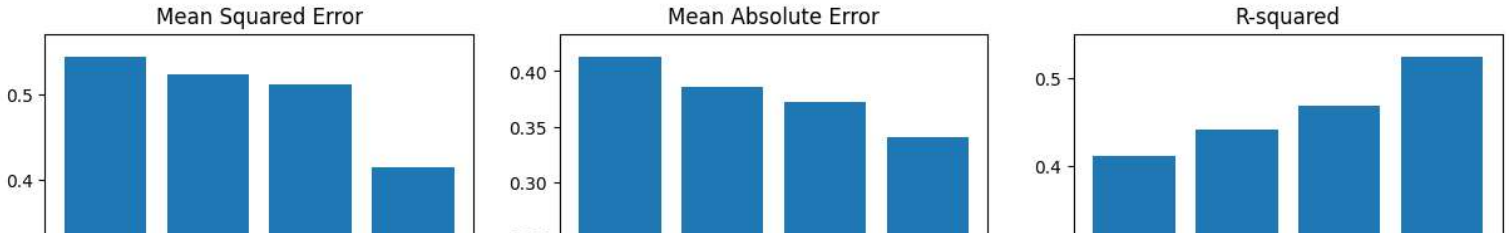



```
# Create a random forest regression object with n_estimators=100
regr_rf = RandomForestRegressor(n_estimators=100, random_state=0)

# Select top 10 features based on RFE of random forest model
selected_features = select_features(regr_rf, listings_df_data, listings_df_target)
```

```
Feature Ranking: [ 1  1  1  1  1  1 13  1  6 19  1  8  1 11  3 10  5  2 20 17 12 21 16 15
  9  4  1  7 14 18]
Top Features: ['accommodates', 'beds', 'bathroom_type_baths', 'bathroom_qty', 'bedrooms', 'room_type_Entire home/apt', 'amenities_qty', 'host_acceptanc
```

```
# Create a random forest regression object with n_estimators=100
regr_rf = RandomForestRegressor(n_estimators=100, random_state=0)

# Train and evaluate the model using different training set sizes
mse_scores, mae_scores, r2_scores = train_and_evaluate(regr_rf, selected_features, listings_df_target)

# Plot the evaluation metrics
plot_evaluation_metrics(mse_scores, mae_scores, r2_scores)
```

```
20% training: Mean squared error: 0.54, Mean absolute error: 0.41, R-squared: 0.41
    Accuracy: 41.20%, Alternative hypothesis not accepted
40% training: Mean squared error: 0.52, Mean absolute error: 0.39, R-squared: 0.44
    Accuracy: 44.13%, Alternative hypothesis not accepted
60% training: Mean squared error: 0.51, Mean absolute error: 0.37, R-squared: 0.47
    Accuracy: 46.93%, Alternative hypothesis not accepted
80% training: Mean squared error: 0.41, Mean absolute error: 0.34, R-squared: 0.52
    Accuracy: 52.48%, Alternative hypothesis not accepted
```

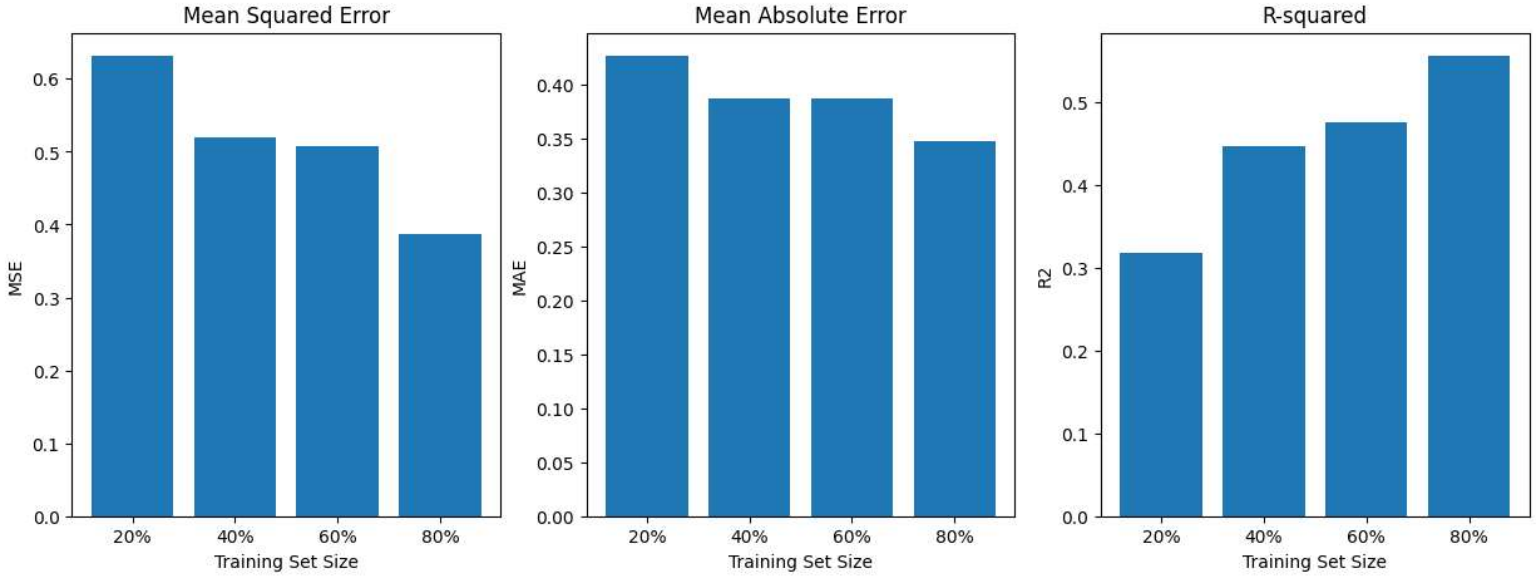


Gradient Boosting Regression

The code below creates a gradient boosting regression object, which is an ensemble learning method that combines multiple weak models (usually decision trees) to create a more accurate model. The parameters for the model are set to `n_estimators=100`, `max_depth=5`, and `learning_rate=0.1`. As before, the model is trained and evaluated using different training set sizes using the `train_and_evaluate()` function. The function returns the mean squared error, mean absolute error, and R-squared error for each configuration. These metrics are then plotted using the `plot_evaluation_metrics()` function.

```
# Create a gradient boosting regression object with n_estimators=100, max_depth=5 and learning_rate=0.1
regr_gb = GradientBoostingRegressor(n_estimators=100, max_depth=5, learning_rate=0.1, random_state=0)

# Train and evaluate the model using different training set sizes
metrics_gb = train_and_evaluate(regr_gb, listings_df_data, listings_df_target)

# Plot the evaluation metrics
plot_evaluation_metrics(metrics_gb[0], metrics_gb[1], metrics_gb[2])
```

```
20% training: Mean squared error: 0.63, Mean absolute error: 0.43, R-squared: 0.32
    Accuracy: 31.84%, Alternative hypothesis not accepted
40% training: Mean squared error: 0.52, Mean absolute error: 0.39, R-squared: 0.45
    Accuracy: 44.69%, Alternative hypothesis not accepted
60% training: Mean squared error: 0.51, Mean absolute error: 0.39, R-squared: 0.47
    Accuracy: 47.47%, Alternative hypothesis not accepted
80% training: Mean squared error: 0.39, Mean absolute error: 0.35, R-squared: 0.56
    Accuracy: 55.55%, Alternative hypothesis not accepted
```

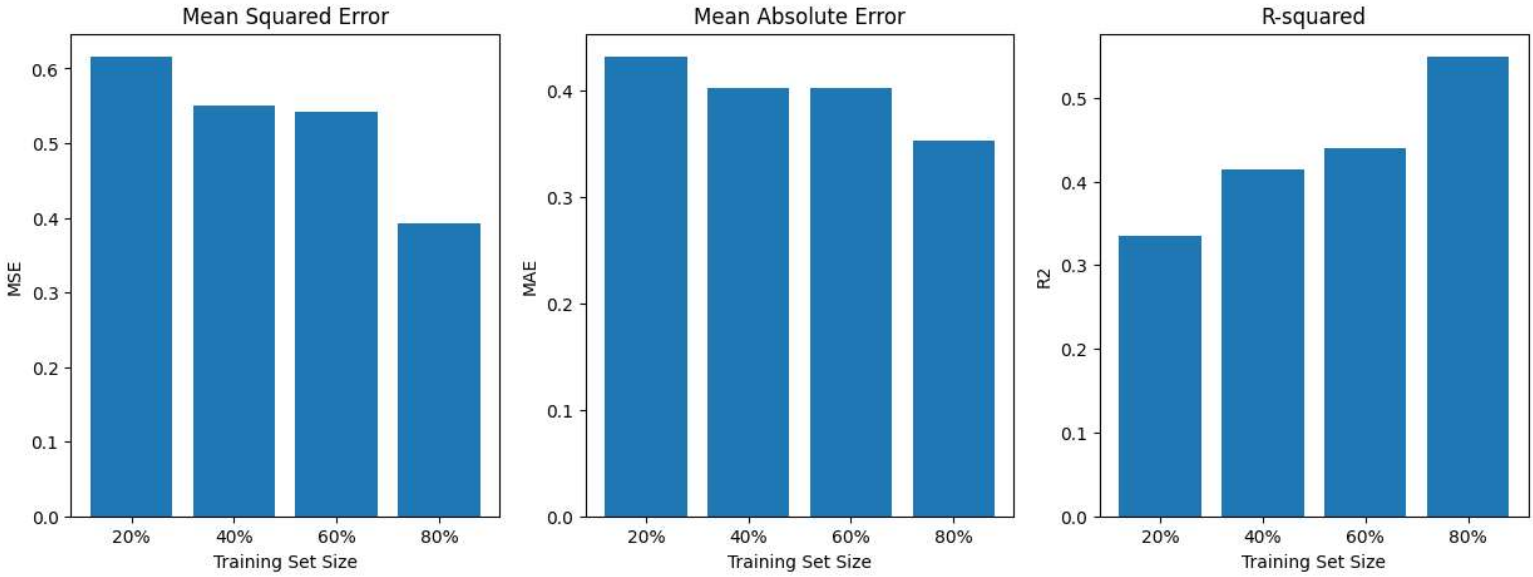



```
# Create a gradient boosting regression object with n_estimators=100, max_depth=5 and learning_rate=0.1
regr_gb = GradientBoostingRegressor(n_estimators=100, max_depth=5, learning_rate=0.1, random_state=0)

# Select top 10 features based on RFE of gradient boosting model
selected_features = select_features(regr_gb, listings_df_data, listings_df_target)
```

```
Feature Ranking: [ 1  1  1  1  1  1  4  1 12 15  1  7  1  9  5  6  3  2 14 17 11 19 21 10
 13  8  1 18 20 16]
Top Features: ['accommodates', 'beds', 'bathroom_type_baths', 'bathroom_qty', 'bedrooms', 'room_type_Entire home/apt', 'amenities_qty', 'host_acceptanc
```

```
# Create a gradient boosting regression object with n_estimators=100, max_depth=5 and learning_rate=0.1
regr_gb = GradientBoostingRegressor(n_estimators=100, max_depth=5, learning_rate=0.1, random_state=0)

# Train and evaluate the model using different training set sizes
mse_scores, mae_scores, r2_scores = train_and_evaluate(regr_gb, selected_features, listings_df_target)

# Plot the evaluation metrics
plot_evaluation_metrics(mse_scores, mae_scores, r2_scores)
```

```
    20% training: Mean squared error: 0.62, Mean absolute error: 0.43, R-squared: 0.34
        Accuracy: 33.54%, Alternative hypothesis not accepted
    40% training: Mean squared error: 0.55, Mean absolute error: 0.40, R-squared: 0.41
        Accuracy: 41.35%, Alternative hypothesis not accepted
    60% training: Mean squared error: 0.54, Mean absolute error: 0.40, R-squared: 0.44
        Accuracy: 43.96%, Alternative hypothesis not accepted
    80% training: Mean squared error: 0.39, Mean absolute error: 0.35, R-squared: 0.55
        Accuracy: 54.89%, Alternative hypothesis not accepted
```



Revision

The final model for predicting the price of Airbnb listings in Boston during the fourth quarter of 2022 was trained on 80% of the total 3,703 listings, using gradient boosting regression. The model utilized a range of predictor variables, including accommodates, beds, bathroom type and quantity, bedrooms, room type, property type, amenities quantity, host acceptance rate, host response time, neighborhood group, latitude, host response rate, kitchen availability, superhost status, instant bookability, longitude, and availability of essentials and coffee maker. The response variable was the price of the listing. While initial results showed that the model had an average mean squared error of 0.37, a mean absolute error of 0.35, and an R-squared score of 0.57, there was room for improvement. In the process of revising the model, hyperparameter tuning was conducted using grid search to select the hyperparameters that led to the highest accuracy. The optimized model had a learning rate of 0.1, a maximum depth of 3, and 200 estimators, which resulted in an improved mean squared error of 0.35, mean absolute error of 0.35, and R-squared score of 0.60. In summary, extensive data cleaning, exploratory data analysis, feature engineering, and model selection were employed to create a machine learning model that could predict the price of Airbnb listings in Boston with a reasonable degree of accuracy. Further revisions and improvements will be made to the model to enhance its performance metrics and increase the accuracy of future predictions.

Additional Models

During the revision process of any machine learning project, it is often useful to try out different models to determine if greater performance can be achieved. In this context, two new models were tried out: MLP regression and SGD regression. The MLP regression model is a neural network-based model that can learn complex nonlinear relationships between input and output variables. It was trained with three hidden layers of 100, 50, and 25 nodes, respectively. The SGD regression model, on the other hand, is a linear model that uses stochastic gradient descent to optimize the weights. Both models were trained and evaluated using different training set sizes, and their performance was compared based on the mean squared error, mean absolute error, and R-squared metrics as before. These evaluation metrics showed that the SGD regression model outperformed the MLP regression model, which suggests that less more complex, linear model is better suited to the problem at hand.

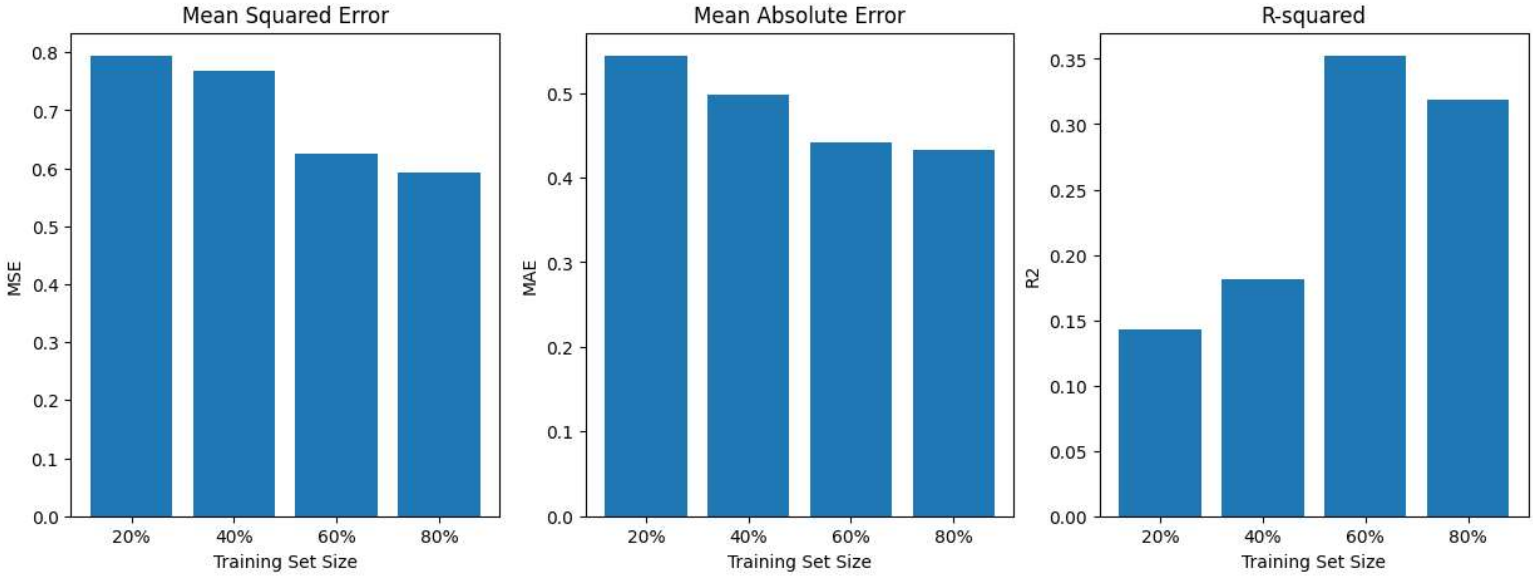
Multi-Layer Perceptron Regression

The code below creates a multi-layer perceptron (MLP) regression object using the MLPRegressor class from the scikit-learn library. MLP regression is a type of neural network regression, where the model has multiple layers of nodes that are interconnected and each node performs a weighted sum of its inputs followed by a non-linear activation function. The MLPRegressor constructor specifies the number of nodes in each hidden layer and the maximum number of iterations for the solver to converge. The code then calls `train_and_evaluate()` to train and evaluate the model using different training set sizes, similar to the other regression models used in this project. Finally, the

`plot_evaluation_metrics()` function is called to plot the mean squared error, mean absolute error, and R-squared metrics for each configuration of the model.

```
# Create an MLP regressoion object with 3 hidden layers of 100, 50 and 25 nodes, respectively
regr_mlp = MLPRegressor(hidden_layer_sizes=(100, 50, 25), max_iter=1000, random_state=0)

# Train and evaluate the model using different training set sizes
metrics_mlp = train_and_evaluate(regr_mlp, listings_df_data, listings_df_target)

# Plot the evaluation metrics
plot_evaluation_metrics(metrics_mlp[0], metrics_mlp[1], metrics_mlp[2])
```

```
20% training: Mean squared error: 0.79, Mean absolute error: 0.54, R-squared: 0.14
    Accuracy: 14.32%, Alternative hypothesis not accepted
40% training: Mean squared error: 0.77, Mean absolute error: 0.50, R-squared: 0.18
    Accuracy: 18.12%, Alternative hypothesis not accepted
60% training: Mean squared error: 0.63, Mean absolute error: 0.44, R-squared: 0.35
    Accuracy: 35.23%, Alternative hypothesis not accepted
80% training: Mean squared error: 0.59, Mean absolute error: 0.43, R-squared: 0.32
    Accuracy: 31.85%, Alternative hypothesis not accepted
```

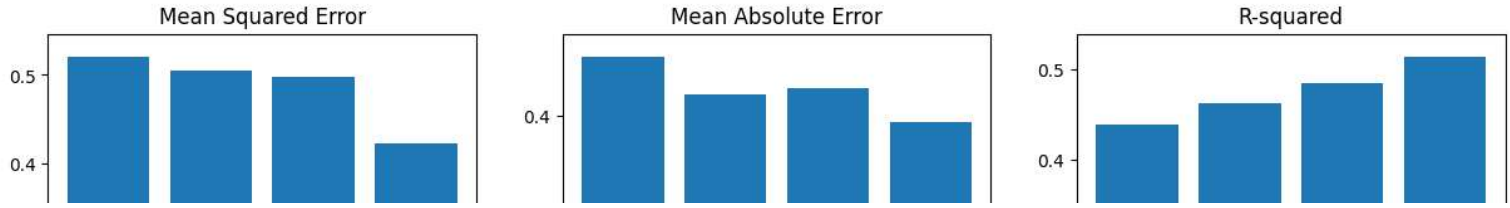


Stochastic Gradient Descent Regression

In the code below, an SGDRegressor object is created with default settings. Stochastic Gradient Descent (SGD) is a popular optimization algorithm used in machine learning, particularly for linear models. It updates the model weights by iteratively minimizing the cost function for a given set of training examples, using small subsets of the data at each iteration (hence the term "stochastic"). The model is then trained and evaluated on different training set sizes using the `train_and_evaluate()` function. Finally, the evaluation metrics are plotted using the `plot_evaluation_metrics()` function.

```
# Create an SGD regression object with default settings
regr_sgd = SGDRegressor(random_state=0)

# Train and evaluate the model using different training set sizes
metrics_sgd = train_and_evaluate(regr_sgd, listings_df_data, listings_df_target)

# Plot the evaluation metrics
plot_evaluation_metrics(metrics_sgd[0], metrics_sgd[1], metrics_sgd[2])
```

```
20% training: Mean squared error: 0.52, Mean absolute error: 0.46, R-squared: 0.44
    Accuracy: 43.84%, Alternative hypothesis not accepted
40% training: Mean squared error: 0.50, Mean absolute error: 0.42, R-squared: 0.46
    Accuracy: 46.27%, Alternative hypothesis not accepted
60% training: Mean squared error: 0.50, Mean absolute error: 0.43, R-squared: 0.49
    Accuracy: 48.53%, Alternative hypothesis not accepted
80% training: Mean squared error: 0.42, Mean absolute error: 0.39, R-squared: 0.51
    Accuracy: 51.44%, Alternative hypothesis not accepted
```



Hyperparameter Tuning

K-Nearest Neighbors Regression

The code below performs hyperparameter optimization for the k-nearest neighbors (KNN) regression algorithm. In the first step, a parameter grid is defined which contains different combinations of hyperparameters that will be tried. The param_grid contains three hyperparameters: `n_neighbors`, `weights`, and `p`. The `n_neighbors` hyperparameter specifies the number of neighbors that will be used to make a prediction, while the `weights` hyperparameter specifies the weight function used in prediction. The `p` hyperparameter is the power parameter for the Minkowski metric, which is used to calculate distances between instances in a feature space. Next, a KNeighborsRegressor object is created with default hyperparameters, which will be updated during the GridSearchCV process. GridSearchCV is a function from the Scikit-learn library that helps to find the best set of hyperparameters for an estimator by searching over a grid of parameter values. The `estimator` parameter specifies the estimator object that will be optimized, `param_grid` parameter contains the parameter grid, `cv` parameter specifies the cross-validation splitting strategy, and `n_jobs` parameter sets the number of CPU cores to use for parallelizing the grid search. After the GridSearchCV object is created, it is passed as the estimator parameter to the `train_and_evaluate()` function, which performs training and evaluation of the KNN regression model on different training set sizes. Finally, the best set of hyperparameters are printed by accessing the `best_params_` attribute of the `grid_search` object, and the evaluation metrics are plotted using the `plot_evaluation_metrics()` function.

```
# Define the parameter grid
param_grid = {
    'n_neighbors': [5, 10, 15, 20],
    'weights': ['uniform', 'distance'],
    'p': [1, 2]
}

# Create a k-nearest neighbors regression object
regr_knn = KNeighborsRegressor()

# Create a grid search object
grid_search = GridSearchCV(estimator=regr_knn, param_grid=param_grid, cv=5, n_jobs=-1)

# Train and evaluate the model using different training set sizes
metrics_knn = train_and_evaluate(grid_search, listings_df_data, listings_df_target)

# Print the best set of hyperparameters
print("Best parameters: ", grid_search.best_params_)

# Plot the evaluation metrics
plot_evaluation_metrics(metrics_knn[0], metrics_knn[1], metrics_knn[2])
```

```
20% training: Mean squared error: 0.50, Mean absolute error: 0.40, R-squared: 0.46
    Accuracy: 46.14%, Alternative hypothesis not accepted
40% training: Mean squared error: 0.46, Mean absolute error: 0.37, R-squared: 0.51
    Accuracy: 51.25%, Alternative hypothesis not accepted
60% training: Mean squared error: 0.45, Mean absolute error: 0.36, R-squared: 0.54
    Accuracy: 53.68%, Alternative hypothesis not accepted
80% training: Mean squared error: 0.36, Mean absolute error: 0.33, R-squared: 0.59
    Accuracy: 58.78%, Alternative hypothesis not accepted
Best parameters:  {'n_neighbors': 15, 'p': 1, 'weights': 'distance'}
```

Random Forest Regression

This code performs hyperparameter tuning using Grid Search Cross-Validation for a random forest regression model. The parameter grid is defined with a range of values for different hyperparameters such as `n_estimators`, `max_depth`, `max_features`, `min_samples_split`, and `min_samples_leaf`. The values for these hyperparameters are chosen such that they cover a wide range of values and also represent different complexity levels of the model. Next, a random forest regression object is created with the default hyperparameters except the random state, which is set to 0 for reproducibility. Then, a grid search object is created with the estimator set as the random forest regression object and the parameter grid specified above. The number of cross-validation folds is set to 5 and the parameter `n_jobs` is set to -1 to use all available CPUs for parallel processing. After creating the grid search object, the `train_and_evaluate()` function is called to train and evaluate the model using different training set sizes. This function takes the grid search object, the data, and the target as input arguments and returns the mean squared error (mse), mean absolute error (mae), and R-squared (r2) scores for each training set size. Finally, the best hyperparameters for the model are printed using the `best_params_` attribute of the grid search object. The evaluation metrics are plotted using the `plot_evaluation_metrics()` function, which takes the mse, mae, and r2 scores as input arguments.


```
# Define the parameter grid
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [3, 5, 7],
    'max_features': ['sqrt', 'log2'],
    'min_samples_split': [2, 4, 6],
    'min_samples_leaf': [1, 2, 4]
}

# Create a random forest regression object
regr_rf = RandomForestRegressor(random_state=0)

# Create a grid search object
grid_search = GridSearchCV(estimator=regr_rf, param_grid=param_grid, cv=5, n_jobs=-1)

# Train and evaluate the model using different training set sizes
metrics_rf = train_and_evaluate(grid_search, listings_df_data, listings_df_target)

# Print the best set of hyperparameters
print("Best parameters: ", grid_search.best_params_)

# Plot the evaluation metrics
plot_evaluation_metrics(metrics_rf[0], metrics_rf[1], metrics_rf[2])
```

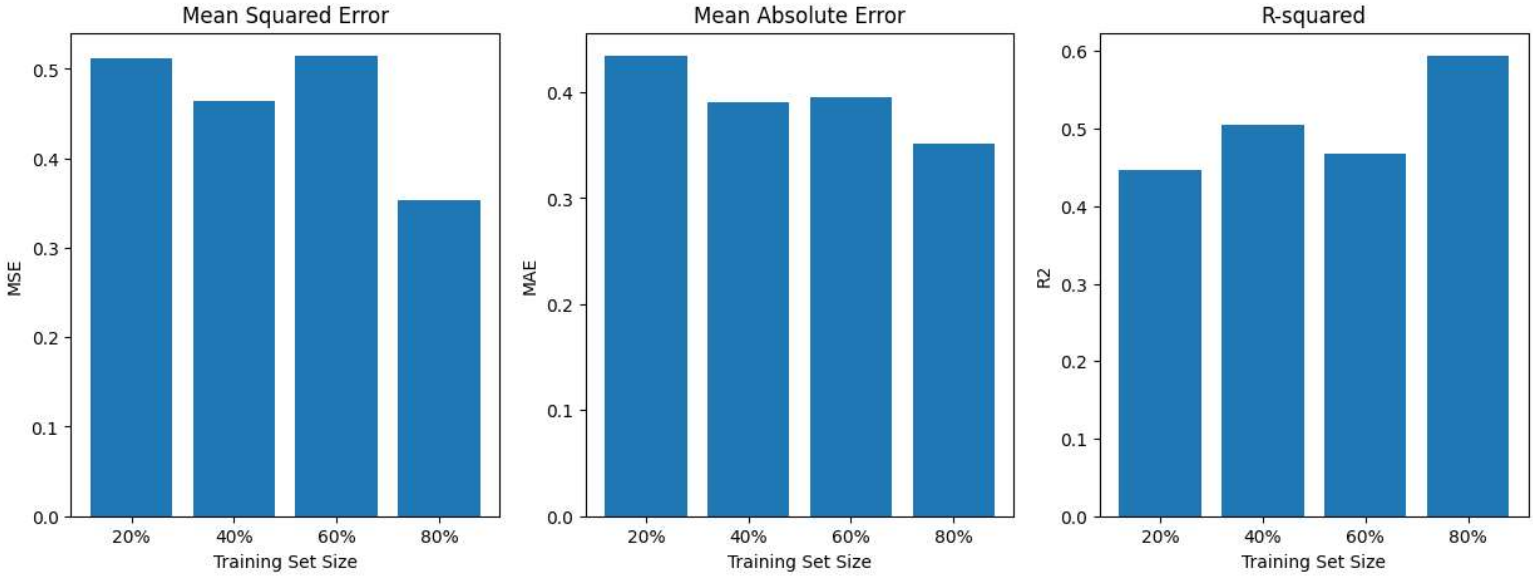
```
20% training: Mean squared error: 0.50, Mean absolute error: 0.42, R-squared: 0.46
    Accuracy: 46.45%, Alternative hypothesis not accepted
40% training: Mean squared error: 0.47, Mean absolute error: 0.39, R-squared: 0.50
    Accuracy: 49.89%, Alternative hypothesis not accepted
60% training: Mean squared error: 0.48, Mean absolute error: 0.39, R-squared: 0.50
```

Gradient Boosting Regression

This code performs hyperparameter tuning for a gradient boosting regression model using grid search. The first step is to define a parameter grid, which contains different values for the hyperparameters that we want to tune. In this case, the hyperparameters we are tuning are `n_estimators`, `max_depth`, and `learning_rate`, with a range of values specified for each hyperparameter in the `param_grid` dictionary. Next, a Gradient Boosting Regression object is created with a fixed random state. Then, a GridSearchCV object is created with the estimator set as the Gradient Boosting Regression object, the parameter grid set as the `param_grid` dictionary, cross-validation set as 5, and `n_jobs` set to -1 to use all available cores. The `train_and_evaluate()` function is then called to train and evaluate the model using the GridSearchCV object, and the resulting mean squared error, mean absolute error, and R^2 scores are stored in `mse_scores`, `mae_scores`, and `r2_scores`, respectively. The best set of hyperparameters is printed using the `best_params_` attribute of the GridSearchCV object, which returns a dictionary of the best hyperparameters found during the search. Finally, the evaluation metrics are plotted using the `plot_evaluation_metrics()` function.

```
# Define the parameter grid
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.1, 1]
}

# Create a gradient boosting regression object
regr_gb = GradientBoostingRegressor(random_state=0)

# Create a grid search object
grid_search = GridSearchCV(estimator=regr_gb, param_grid=param_grid, cv=5, n_jobs=-1)

# Train and evaluate the model using different training set sizes
metrics_gb = train_and_evaluate(grid_search, listings_df_data, listings_df_target)

# Print the best set of hyperparameters
print("Best parameters: ", grid_search.best_params_)

# Plot the evaluation metrics
plot_evaluation_metrics(metrics_gb[0], metrics_gb[1], metrics_gb[2])
```

```
20% training: Mean squared error: 0.51, Mean absolute error: 0.43, R-squared: 0.45
    Accuracy: 44.68%, Alternative hypothesis not accepted
40% training: Mean squared error: 0.46, Mean absolute error: 0.39, R-squared: 0.51
    Accuracy: 50.54%, Alternative hypothesis not accepted
60% training: Mean squared error: 0.51, Mean absolute error: 0.40, R-squared: 0.47
    Accuracy: 46.74%, Alternative hypothesis not accepted
80% training: Mean squared error: 0.35, Mean absolute error: 0.35, R-squared: 0.59
    Accuracy: 59.40%, Alternative hypothesis not accepted
Best parameters:  {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 200}
```



Model Selection

This code creates a horizontal bar chart comparing the evaluation metrics (R-squared, mean squared error, and mean absolute error) of the regression models explored in this project. The models and their corresponding evaluation metrics are defined for 80% training at the beginning

of the code. The models are sorted in descending order based on their R-squared score using the `sorted()` function. The positions of the bars on the x-axis are set using `np.arange()` and `bar_width`. The bar charts for each evaluation metric are created using `ax.barh()`. The `align='center'` argument centers the bars on the y-axis ticks. The y-axis labels and ticks are set using `ax.set_yticks()` and `ax.set_yticklabels()`. The `ax.invert_yaxis()` function is used to invert the y-axis labels so that the top model is at the top of the plot. Finally, a legend and title are added to the plot using `ax.legend()` and `ax.set_title()`, respectively. The result is a horizontal bar chart comparing the evaluation metrics for each model. Each model has three bars, one for each evaluation metric. The models are sorted in descending order based on their R-squared score. The legend indicates which color corresponds to each evaluation metric. The highest R-squared value corresponds to the Gradient Boosting Regressor model, followed by the K-Nearest Neighbors Regressor model, and so on.

```
# Define the models and their corresponding evaluation metrics
fig, ax = plt.subplots(figsize=(10, 6))
models = ('K-Nearest Neighbors Regressor', 'Linear Regressor', 'Decision Tree Regressor', 'Random Forest Regressor', 'Gradient Boosting Regressor', 'MLP Reg
mse_scores = [metrics_knn[0][-1], metrics_lin[0][-1], metrics_dt[0][-1], metrics_rf[0][-1], metrics_gb[0][-1], metrics_mlp[0][-1], metrics_sgd[0][-1]]
mae_scores = [metrics_knn[1][-1], metrics_lin[1][-1], metrics_dt[1][-1], metrics_rf[1][-1], metrics_gb[1][-1], metrics_mlp[1][-1], metrics_sgd[1][-1]]
r2_scores = [metrics_knn[2][-1], metrics_lin[2][-1], metrics_dt[2][-1], metrics_rf[2][-1], metrics_gb[2][-1], metrics_mlp[2][-1], metrics_sgd[2][-1]]

# Sort the models by R-squared score (in descending order)
sorted_models = [x for _, x in sorted(zip(r2_scores, models), reverse=True)]
sorted_mse_scores = [x for _, x in sorted(zip(r2_scores, mse_scores), reverse=True)]
sorted_mae_scores = [x for _, x in sorted(zip(r2_scores, mae_scores), reverse=True)]
sorted_r2_scores = sorted(r2_scores, reverse=True)

# Set the positions of the bars on the x-axis
bar_width = 0.2
r2_pos = np.arange(len(models))
mse_pos = [x + bar_width for x in r2_pos]
mae_pos = [x + bar_width*2 for x in r2_pos]

# Create the bar charts for each evaluation metric
ax.barh(r2_pos, sorted_r2_scores, bar_width, align='center', label='R-squared')
ax.barh(mse_pos, sorted_mse_scores, bar_width, align='center', label='Mean Squared Error')
ax.barh(mae_pos, sorted_mae_scores, bar_width, align='center', label='Mean Absolute Error')

# Set the y-axis labels and ticks
ax.set_yticks(r2_pos + bar_width)
ax.set_yticklabels(sorted_models)
ax.invert_yaxis()

# Add a legend and a title to the plot
ax.legend()
ax.set_xlabel('Performance Metric Scores')
ax.set_title('Evaluation Metric Model Comparison (sorted by R-squared)')
```

```
Text(0.5, 1.0, 'Evaluation Metric Model Comparison (sorted by R-squared)')
```