Predicting Weekend AirBnB Asking Prices in London

Group 1 Project Report: Jackson Small, Elizabeth Barnum, Sophia Kohn, Danielle Bodziak, Thomas Tibbetts, Osvaldo Pelaez, AJ Romaniello, Patrick de Beer, Jadan Colon

Background

Launched in 2008, Airbnb is an online platform where users can rent and list properties for both short and long-term stays (About us, 2016). Since its inception, Airbnb has become a global leader in vacation rentals, featuring over 8 million listings in 220+ countries (About us, 2016).
London, one of Airbnb's most prolific markets since the company's conception, currently features 96,182 active Airbnb listings, making it an ideal location for our research (London, n.d.).

Hotel studies consistently highlight location as a key factor: specifically proximity to city centers, tourist attractions, and public transportation hubs (Kim et al., 2018; Yang et al., 2018; Zhang et al., 2011).
Studies of the Airbnb market have determined that size, distance from the city center, and host ratings all play a large role in determining guests' willingness to pay higher prices (Gibbs et al., 2018; Teubner & Dann, 2017; Wang & Nicolau, 2017).
However, many of these studies are limited by the fact that they assess Airbnbs over multiple cities, not accounting for the spatial dependence of the real estate market (Tang et a., 2019).

Objective

We aim to use regression analysis to predict Airbnb weekend prices in London based on property characteristics and guest preferences.
By focusing on weekend prices in a single city, we aim to eliminate confounding factors and better understand price determinants, thus providing actionable insights for hosts, travelers, and market analysts.
This predictive model may assist new AirBnB hosts in pricing their properties competitively and in-tune with market trends, while travelers can make estimates for stay prices based on prior customer experiences and desired property characteristics.

About the Dataset

The dataset was made public on the Hugging Face database, deriving from supplementary material for the article "Determinants of Airbnb prices in European cities: A spatial econometrics approach" (Gyódi & Nawaro, 2021). It consists of almost 5,500 observations, each row representing a listing for an Airbnb property.
The data was collected according to the following methodology:

- The listings collected were restricted to Inner London.
- The listings were also limited to just weekends (Friday-Sunday)
- The prices were found 4-6 weeks before travel dates using automated search queries.
  - Offers were collected in 2019 between 6/28 and 7/12 for a trip from 8/9 - 8/11
  - Each observation records the price of a two-night stay for two guests (including fees)
  - Listings for 6+ people were excluded

This data collection methodology assists in our research because restricting the data collection to just Inner London helps to mitigate the effects of confounding factors like listing location. Additionally, the actual prices were collected within a very short range of time, giving us more certainty that seasonality will not play a strong role in our model.

In [103…
```
import pandas as pd
import numpy as np
from sklearn.linear_model import (LinearRegression, PoissonRegressor, LassoCV,
                                  RidgeCV, ElasticNetCV, Ridge, Lasso, ElasticNet)
from sklearn.metrics import root_mean_squared_error, mean_squared_error
from sklearn.metrics import mean_absolute_percentage_error
from sklearn.model_selection import cross_validate, train_test_split
from matplotlib import pyplot as plt
import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor
import pylab
import seaborn as sb
from sklearn.preprocessing import StandardScaler, SplineTransformer
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.inspection import PartialDependenceDisplay, partial_dependence
```

Importing the dataset:

In [104…
```
df = pd.read_csv("airbnb.csv")
print(df.columns)
df = df.drop(["Unnamed: 0", "_id"], axis=1)
```

```
Index(['Unnamed: 0', '_id', 'realSum', 'room_type', 'room_shared',
       'room_private', 'person_capacity', 'host_is_superhost', 'multi', 'biz',
       'cleanliness_rating', 'guest_satisfaction_overall', 'bedrooms', 'dist',
       'metro_dist', 'attr_index', 'attr_index_norm', 'rest_index',
       'rest_index_norm'],
      dtype='object')
```

Continuous Variables

- realSum: The full price of accommodation for two people for two nights (in Euros)
- Person_capacity: Max capacity of the Airbnb
- Cleanliness_rating: Score from 0 to 10 representing Airbnb cleanliness
- Guest_satisfaction_overall: Score from 1 to 100 indicating guest satisfaction
- Bedrooms: Number of bedrooms in the Airbnb
- Dist: Distance from the city center
- Metro_dist: Distance from the nearest metro station
- Attr_index_norm: Score between 1 to 100 judging the attraction of the location
- Rest_index_norm: Score between 0 to 100 indicating the score of the restaurants nearby

Categorical Variables

- Multi Listing (True/False): Indicates whether or not the host has 2-4 Airbnb listings.
- Business Listing (True/False): Indicates whether or not the host has more than 4 Airbnb listings.
- Superhost (True/False): Indicates whether or not the host is labeled as a "Superhost" on the Airbnb platform.
- Room Type: Indicates whether the listing is for an "Entire home/apartment," a "Private room," or a "Shared room."

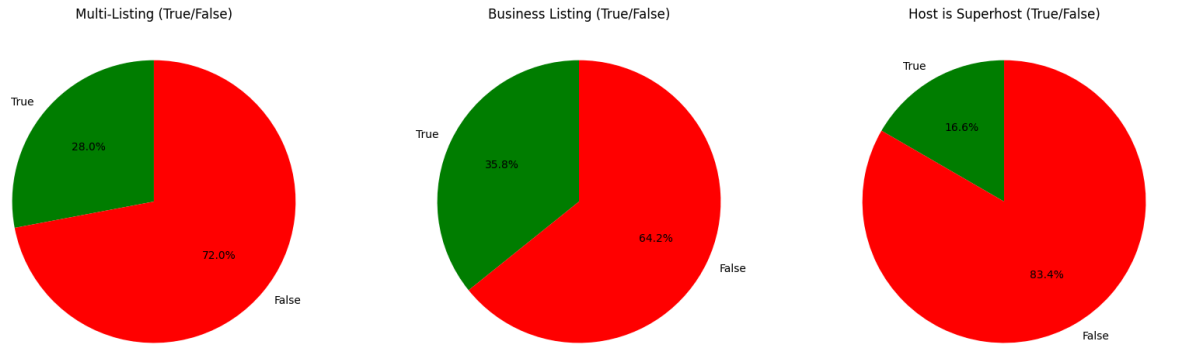
Data exploration

First we will plot and comment on the distributions of each of the categorical variables.

In [105…

```
fig, axes = plt.subplots(1, 3, figsize=(20,10))

tf_labels = [True, False]

multi_sizes = [0,0]
for i in [0,1]:
    multi_sizes[i] = (df.multi == tf_labels[i]).sum()

axes[0].pie(multi_sizes,
            labels=tf_labels,
            autopct='%1.1f%%',
            colors = ["green", "red"],
            startangle = 90)
axes[0].set_title("Multi-Listing (True/False)")

biz_sizes = [0,0]
for i in [0,1]:
    biz_sizes[i] = (df.biz == tf_labels[i]).sum()

axes[1].pie(biz_sizes,
            labels=tf_labels,
```

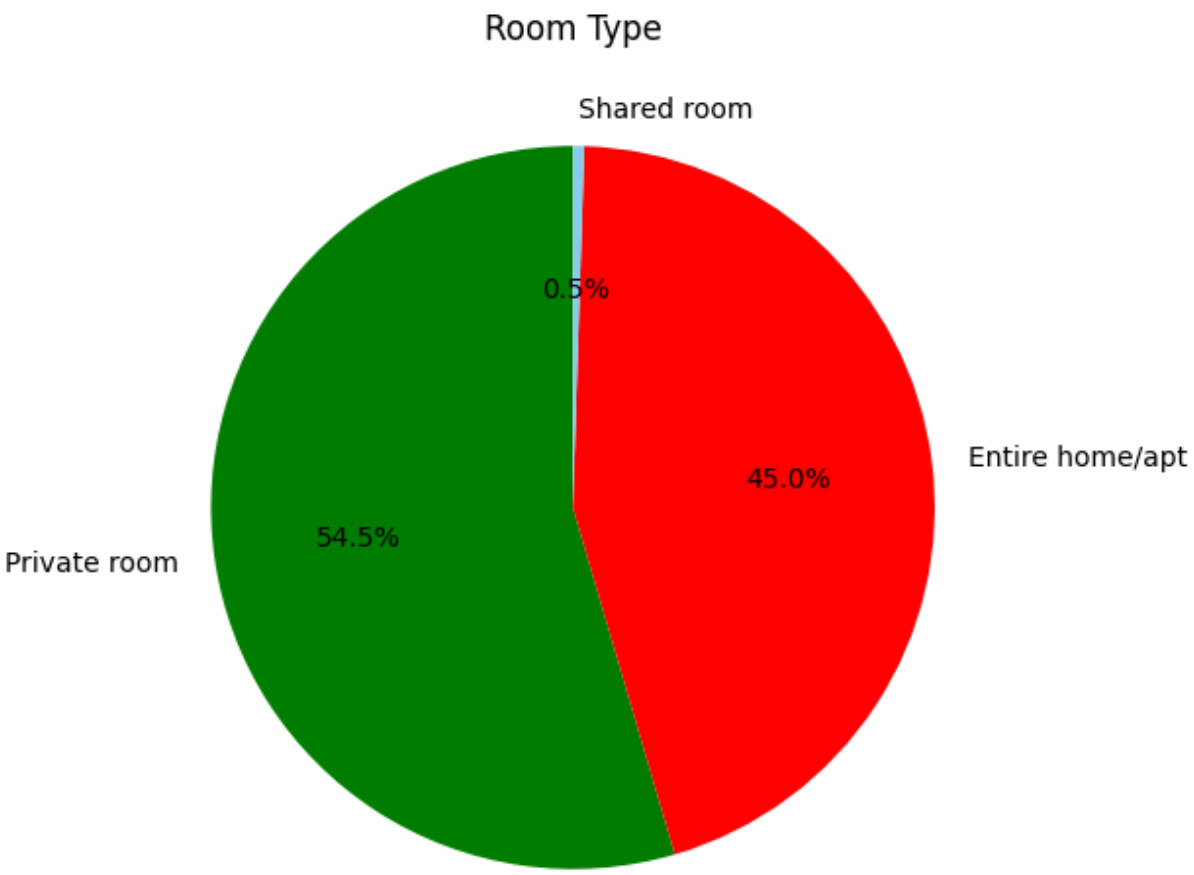
```
            autopct='%1.1f%%',
            colors = ["green", "red"],
            startangle = 90)
axes[1].set_title("Business Listing (True/False)")

superhost_sizes = [0,0]
for i in [0,1]:
    superhost_sizes[i] = (df.host_is_superhost == tf_labels[i]).sum()

axes[2].pie(superhost_sizes,
            labels=tf_labels,
            autopct='%1.1f%%',
            colors = ["green", "red"],
            startangle = 90)
axes[2].set_title("Host is Superhost (True/False)")

plt.show()
```



- Of the listings sampled, a significant proportion of them are listed by hosts who have more than 4 listings.
- Only about 16.6% of the sampled listings were owned by hosts with "Superhost" status.

In [106…

```
fig, ax = plt.subplots(1,1, figsize = (6,6))

room_labels = df.room_type.unique()
rt_sizes = [0,0,0]

for i in [0,1,2]:
    rt_sizes[i] = (df.room_type == room_labels[i]).sum()

ax.pie(rt_sizes,
       labels=room_labels,
       autopct='%1.1f%%',
       colors = ["green", "red", "skyblue"],
       startangle = 90)
ax.set_title("Room Type")
plt.show()
```

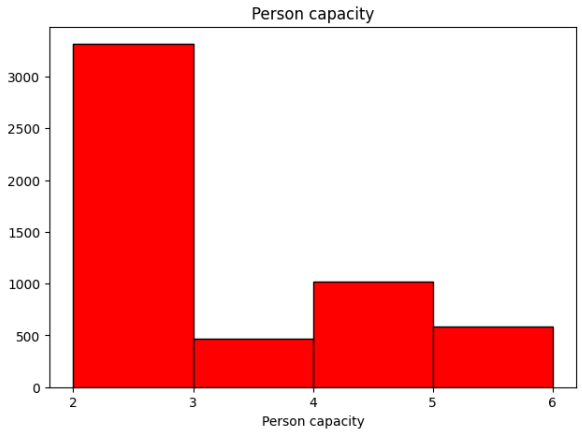


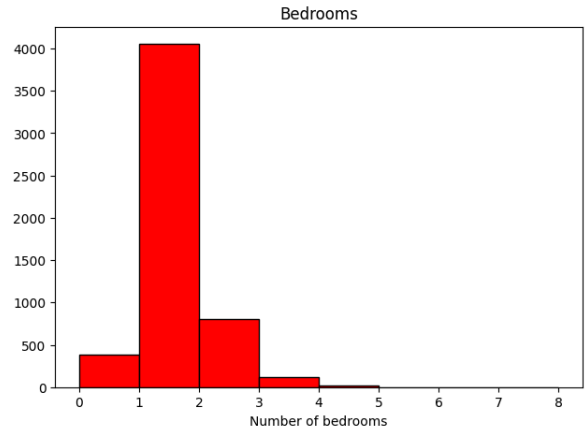
- Most of the sampled listings are private rooms, closely followed by listings for an entire home/apartment.
- Listings for shared rooms make up only 0.5% of those sampled.

Now we take a look at the continuous distributions:

In [107…

```
fig, axes = plt.subplots(1,2, figsize=(16,5))

axes[0].hist(df.person_capacity,
             bins=np.linspace(df.person_capacity.min(), df.person_capacity.max(), n
             color = "red",
             edgecolor="black")
axes[0].set_title("Person capacity")
axes[0].set_xlabel("Person capacity")
axes[0].set_xticks([2,3,4,5,6])

axes[1].hist(df.bedrooms,
             bins=np.linspace(df.bedrooms.min(), df.bedrooms.max(), num=9),
             color = "red",
             edgecolor="black")
axes[1].set_title("Bedrooms")
axes[1].set_xlabel("Number of bedrooms")
plt.show()
```

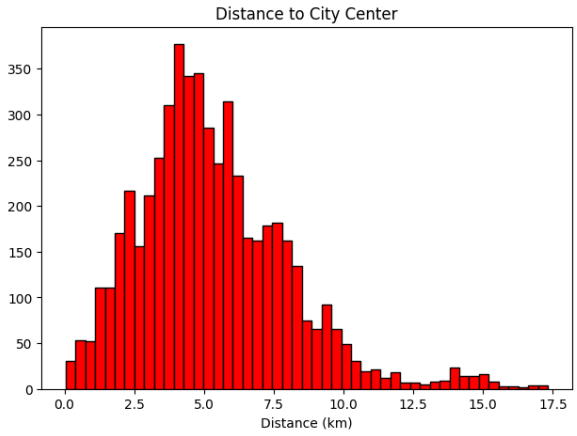


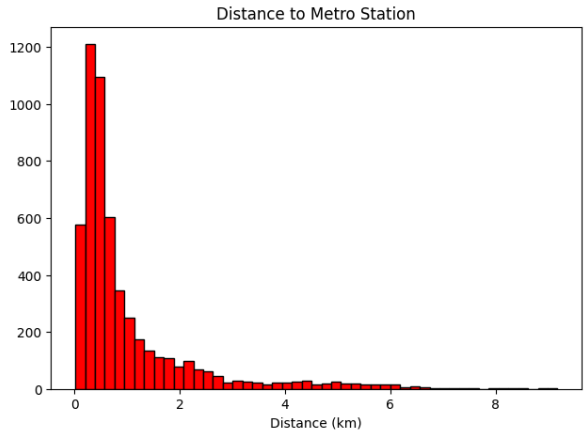


The distributions of person capacity and bedrooms are both very uneven and clustered to the lower side. We make note of the fact that there are extremely few listings with more than 3 bedrooms.

In [108…

```
fig, axes = plt.subplots(1,2, figsize=(16,5))

axes[0].hist(df.dist,
             bins=np.linspace(df.dist.min(), df.dist.max(), num=50),
             color = "red",
             edgecolor="black")
axes[0].set_title("Distance to City Center")
axes[0].set_xlabel("Distance (km)")

axes[1].hist(df.metro_dist,
             bins=np.linspace(df.metro_dist.min(), df.metro_dist.max(), num=50),
             color = "red",
             edgecolor="black")
axes[1].set_title("Distance to Metro Station")
axes[1].set_xlabel("Distance (km)")
plt.show()
```

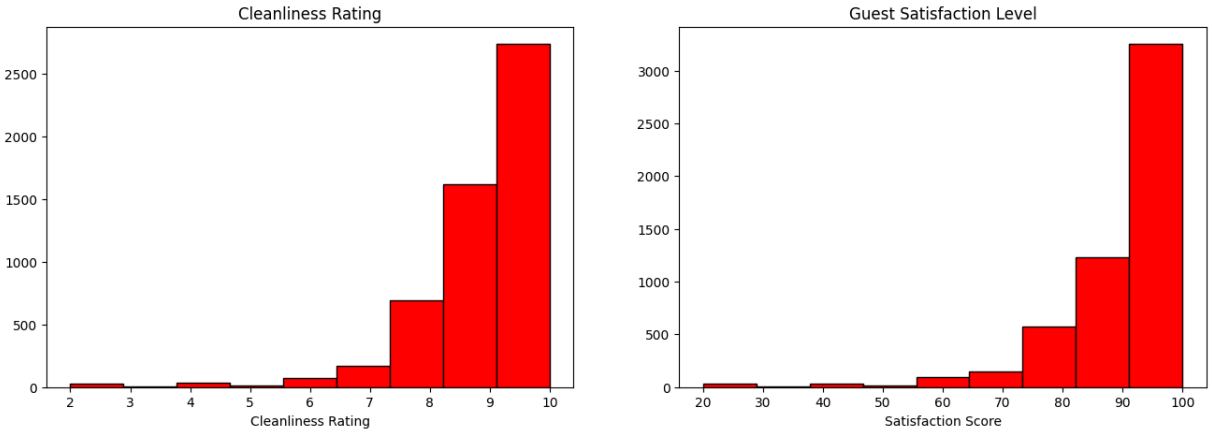




- A great majority of the listings are within 2 km of a metro station. However, the distribution of distance to the city center is spread wider from around 1-10 km, with a peak at 5 km.

In [109…

```
fig, axes = plt.subplots(1, 2, figsize=(16,5))

axes[0].hist(df.cleanliness_rating,
             bins=np.linspace(df.cleanliness_rating.min(),
                              df.cleanliness_rating.max(),
                              num=10),
             color = "red",
             edgecolor="black")
axes[0].set_title("Cleanliness Rating")
axes[0].set_xlabel("Cleanliness Rating")

axes[1].hist(df.guest_satisfaction_overall,
             bins=np.linspace(df.guest_satisfaction_overall.min(),
                              df.guest_satisfaction_overall.max(),
                              num=10),
             color = "red",
             edgecolor="black")
axes[1].set_title("Guest Satisfaction Level")
axes[1].set_xlabel("Satisfaction Score")
plt.show()
```

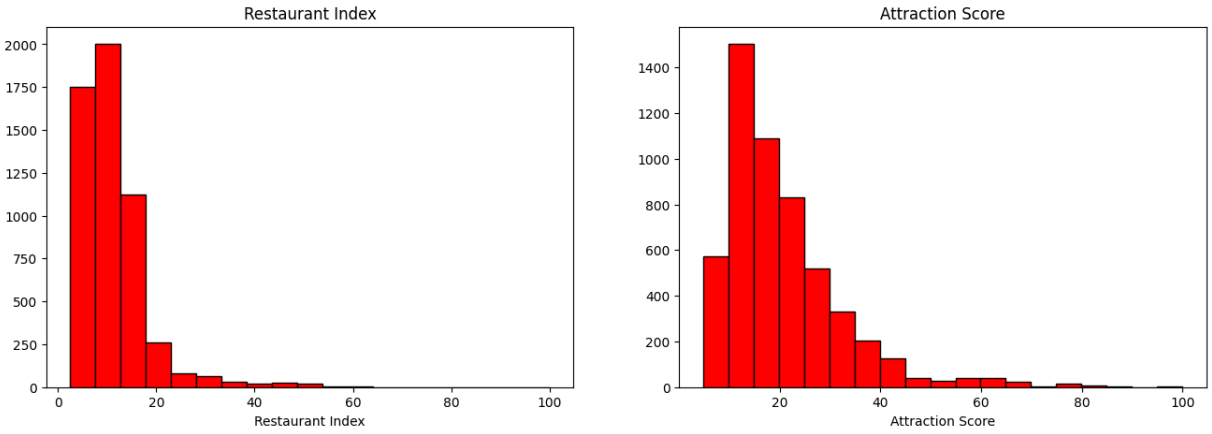


The cleanliness rating and guest satisfaction variables are very similarly distributed, strongly clustered around the highest possible score. Many of the sampled listings satisfied their guests' expectations completely.

In [110…

```
fig, axes = plt.subplots(1, 2, figsize=(16,5))

axes[0].hist(df.rest_index_norm,
             bins=np.linspace(df.rest_index_norm.min(),
                              df.rest_index_norm.max(),
                              num=20),
             color = "red",
             edgecolor="black")
axes[0].set_title("Restaurant Index")
axes[0].set_xlabel("Restaurant Index")

axes[1].hist(df.attr_index_norm,
             bins=np.linspace(df.attr_index_norm.min(),
                              df.attr_index_norm.max(),
                              num=20),
             color = "red",
```

```
            edgecolor="black")
axes[1].set_title("Attraction Score")
axes[1].set_xlabel("Attraction Score")
plt.show()
```



- The restaurant index is very clustered toward the lower values. Only a rare few properties achieve exceptional restaurant index scores.
- The attraction index has a somewhat more even spread.

Before exploring the univariate relationships between the predictors and response variable, we will perform some transformations on the dataset.

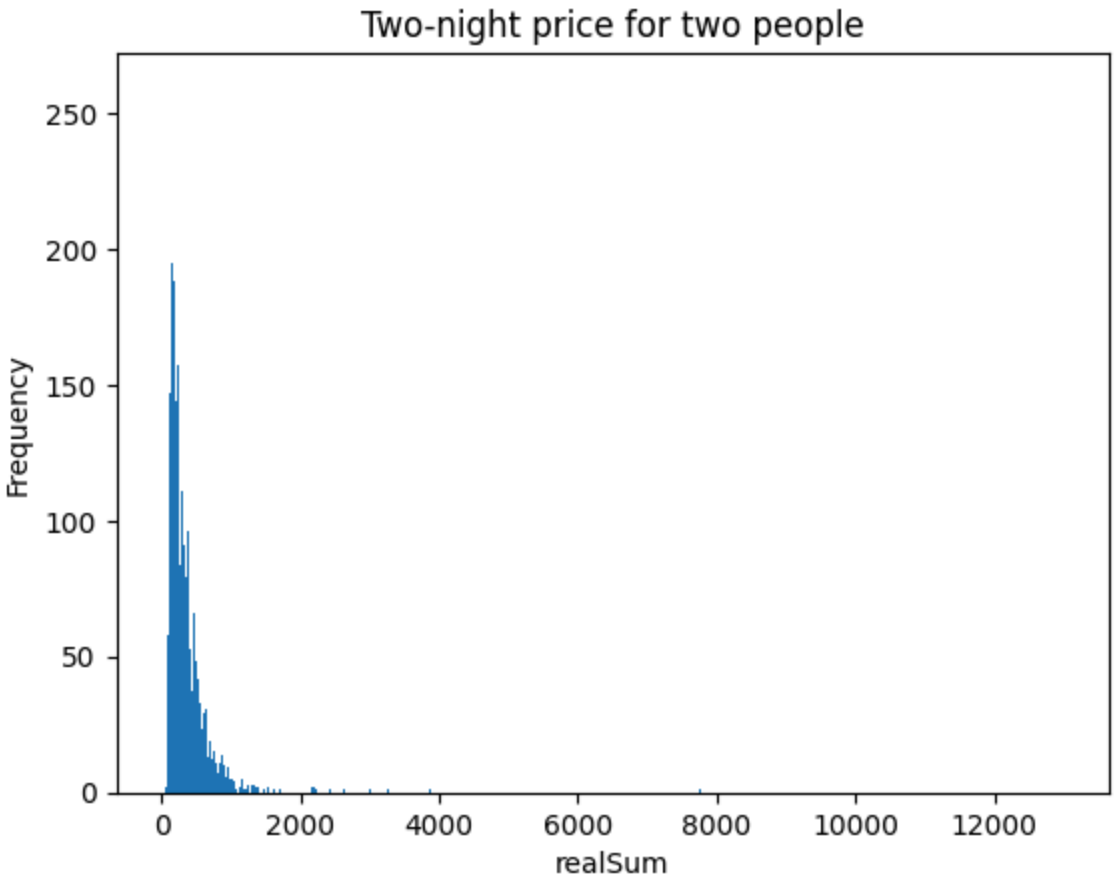
Data preprocessing

The realSum response variable represents the price of a two-night stay for two-people at each property. Its distribution is highly uneven and skewed right, as can be seen from summary statistics and a histogram.

In [111…
```
df["realSum"].describe()
```

Out[111…
```
count     5379.000000
mean       364.389747
std        437.742534
min         54.328653
25%        174.510219
50%        268.115431
75%        438.274654
max      12937.275101
Name: realSum, dtype: float64
```

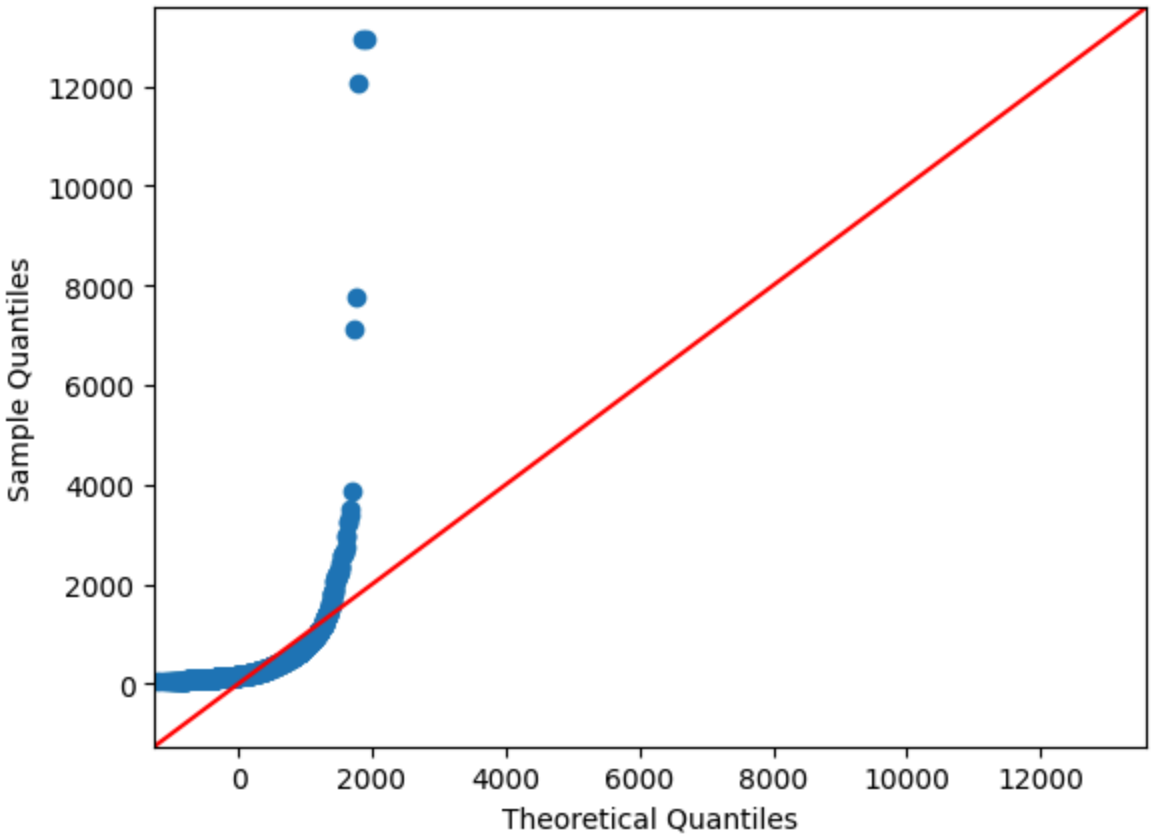
In [112…
```
plt.hist(df.realSum, bins = range(0,13000,10))
plt.xlabel('realSum')
plt.ylabel('Frequency')
plt.title('Two-night price for two people')
plt.show()
```



A Q-Q plot shows that the distribution of realSum is highly non-normal, which is concerning.

In [113…

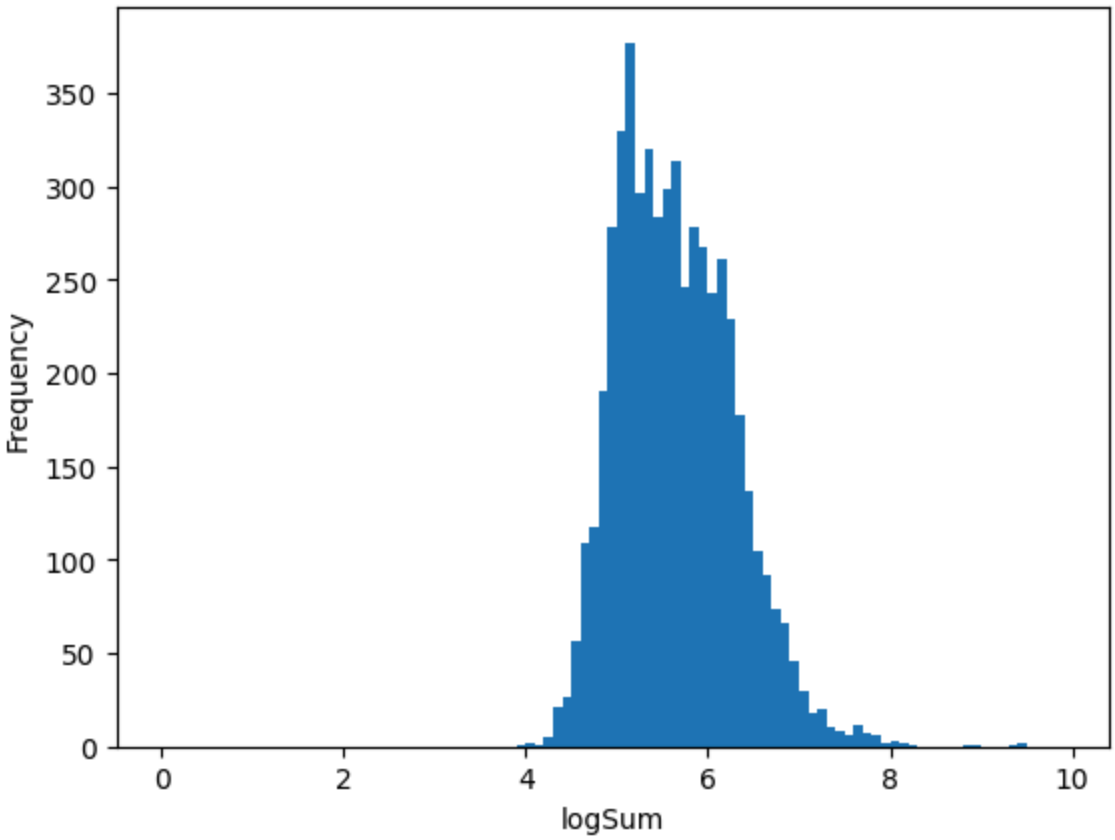
```
realSum_qqplot = sm.qqplot(df.realSum,
                           loc = df.realSum.mean(),
                           scale = df.realSum.std(),
                           line='45')
```



In order to adjust the distribution of the response to aid our modeling process, we will perform a log transformation on the entire column. We define the transformed response variable as "logSum."

In [114…

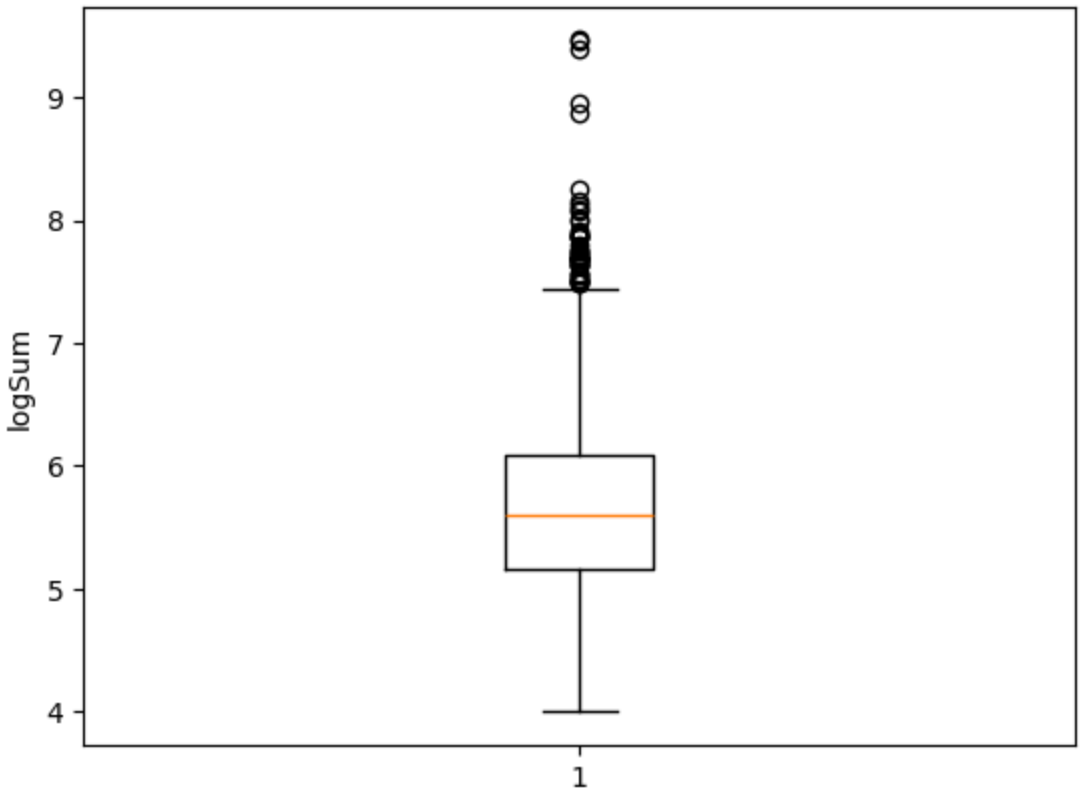
```
newdf = df.copy()
newdf.realSum = np.log(newdf.realSum)
plt.hist(newdf["realSum"],
         bins = [x/10.0 for x in range(0,100,1)])
plt.xlabel('logSum')
plt.ylabel('Frequency')
plt.show()
```



logSum appears to be much more evenly distributed than realSum. However there are still a significant number of outliers, as can be seen from a boxplot.

In [115…

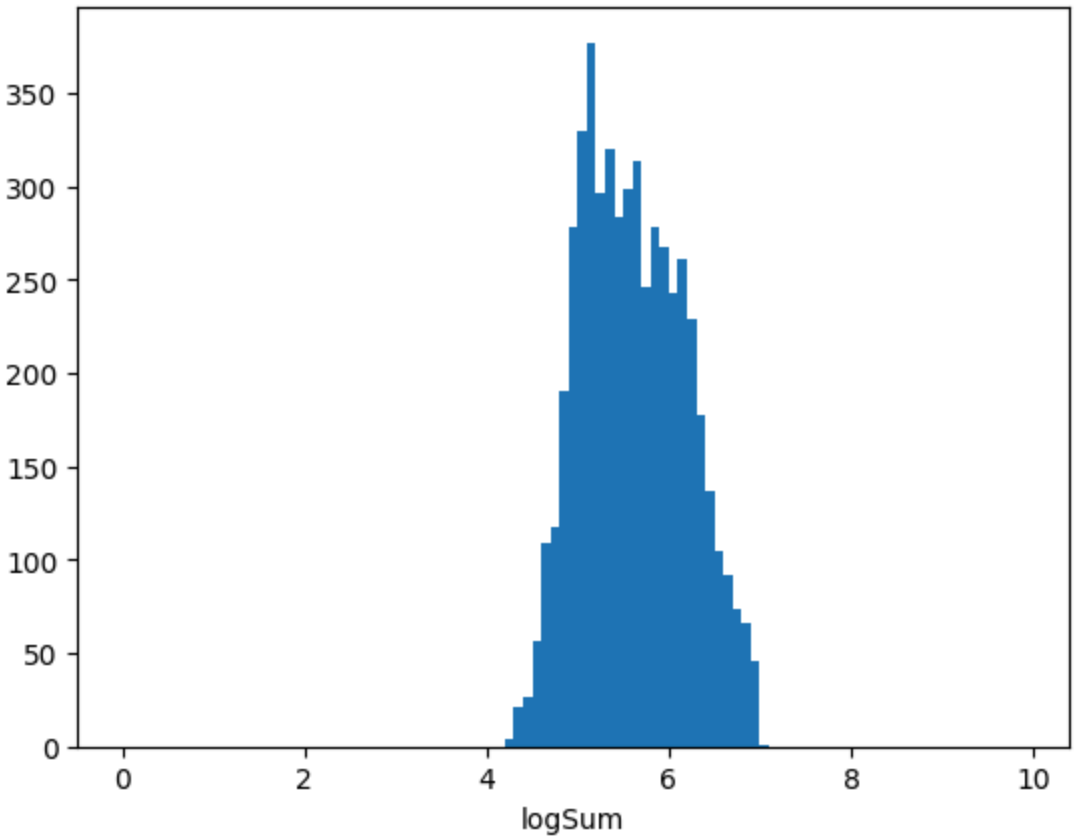
```
plt.boxplot(newdf["realSum"])
plt.ylabel('logSum')
plt.show()
```



We will handle outliers in logSum using the rule of thumb that any observations outside the interval [Q1 - 1.5 * IQR, Q3 + 1.5 * IQR] are considered outliers.

In [116…

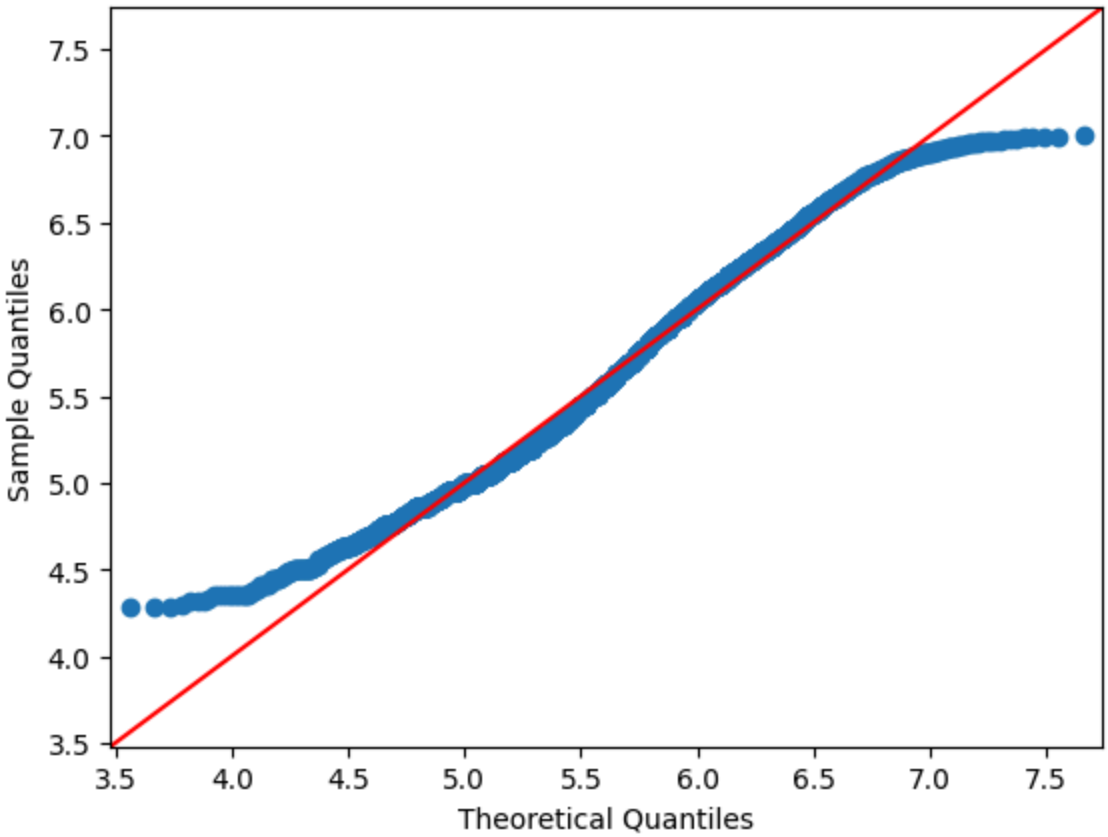
```
filtereddf = newdf[(newdf.realSum < 7.003709) & (newdf.realSum > 4.24112)]
plt.hist(filtereddf.realSum,
         bins = [x / 10.0 for x in range(0,100,1)])
plt.xlabel("logSum")
plt.show()
```



After transforming the response and removing outliers, we are left with 5,245 observations. As seen from the new QQ-plot, the distribution of logSum is much more approximately normal.

In [117…

```
logSum_qqplot = sm.qqplot(filtereddf.realSum,
                          loc = filtereddf.realSum.mean(),
                          scale = filtereddf.realSum.std(),
                          line = '45')
```



Defining X and y

In [118…

```
X_columns = ["room_shared",
             "room_private",
             "person_capacity",
             "host_is_superhost",
             "multi",
             "biz",
             "cleanliness_rating",
             "guest_satisfaction_overall",
             "bedrooms",
             "dist",
             "metro_dist",
             "attr_index_norm",
             "rest_index_norm"]
```

We create our design matrix X and convert the True/False columns to integer columns of 0s and 1s. The response column y is defined to be the logSum variable.

In [ ]:

```
X = filtereddf[X_columns]
type_convert = {'room_shared': int,
                'room_private': int,
                'host_is_superhost': int}
X = X.astype(type_convert)
y = filtereddf["realSum"]
y = y.rename("logSum")
```

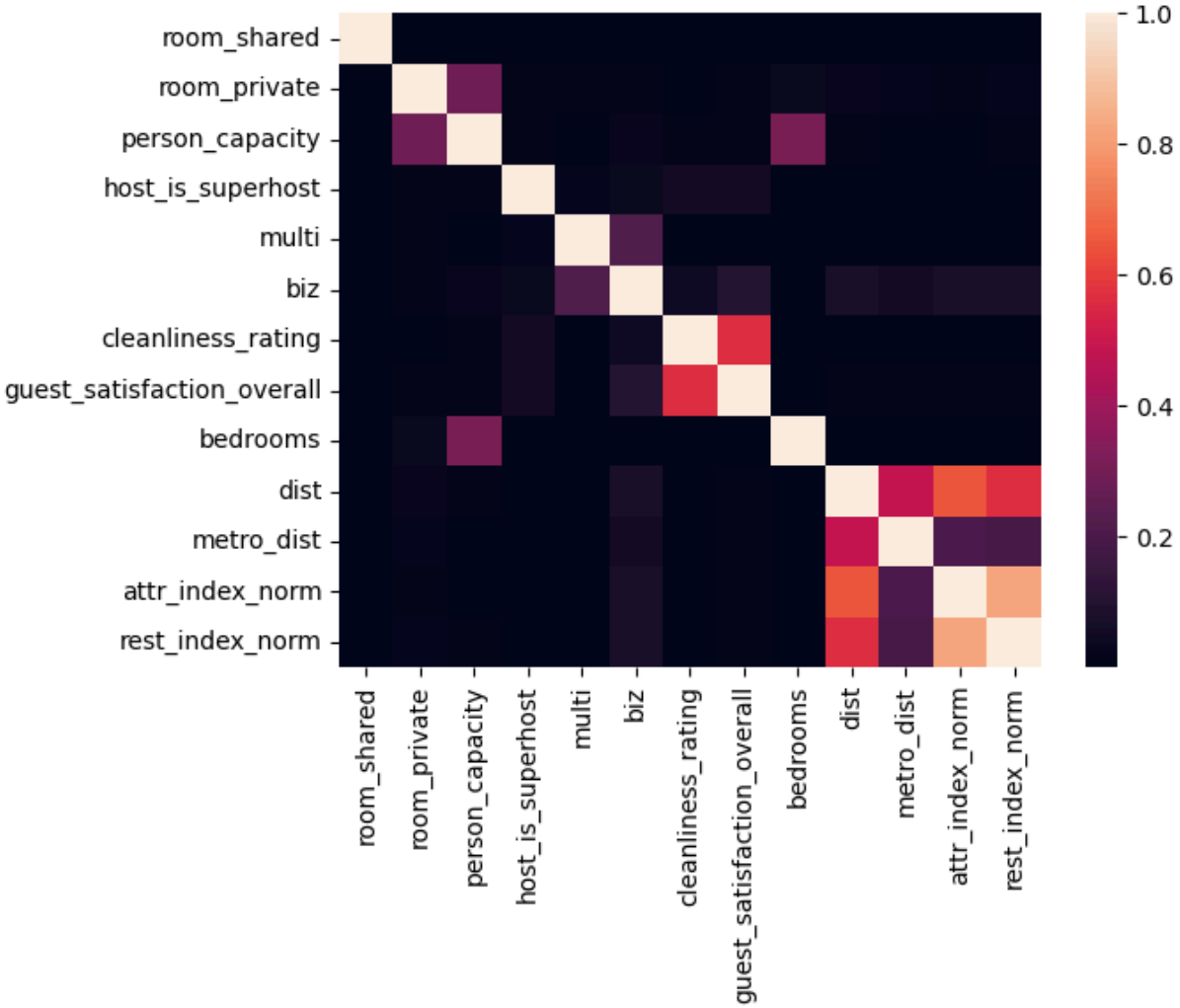
Examining multicollinearity and variable selection

Collinearity among different predictors could cause issues later with unstable parameter estimates and convergence of certain models, such as Poisson regression.

For this reason, we want to address any collinearity during the preprocessing. We first create a matrix whose ijth entry represents the R^2 between the ith and jth predictors. Then we create a heatmap to help us visualize which predictors are correlated with each other.

In [120…

```
corrplot = sb.heatmap(np.square(X.corr()))
```



With this correlation matrix we can identify some potentially problematic predictors, such as "cleanliness_rating", "guest_satisfaction_overall", "dist"/"metro_dist", and "attr_index"/"rest_index". To further explore multicollinearity, we print the VIFs (Variable Inflation Factors) for each variable. The VIF takes a value from 1 to infinity, with values above 10 considered indicative of potential multicollinearity.

In [121…
```
vifs = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
VIFSdf = pd.DataFrame({'Variable': X.columns, 'VIF': vifs})
print(VIFSdf)
```

	Variable	VIF
0	room_shared	1.016795
1	room_private	3.291195
2	person_capacity	12.867625
3	host_is_superhost	1.299199
4	multi	1.817243
5	biz	2.381678
6	cleanliness_rating	141.689208
7	guest_satisfaction_overall	146.584348
8	bedrooms	7.817964
9	dist	18.080680
10	metro_dist	3.350629
11	attr_index_norm	28.817345
12	rest_index_norm	21.614279

It appears that several variables have concerningly high VIFs, especially "cleanliness_rating" and "guest_satisfaction." We drop the "guest_satisfaction_overall" column to see if this improves the VIFs.

In [122…
```
X_drop1 = X.drop("guest_satisfaction_overall", axis=1)
vifs = [variance_inflation_factor(X_drop1.values, i) for i in range(X_drop1.shape[1
VIFSdf = pd.DataFrame({'Variable': X_drop1.columns, 'VIF': vifs})
print(VIFSdf)
```

	Variable	VIF
0	room_shared	1.016631
1	room_private	3.215701
2	person_capacity	12.737691
3	host_is_superhost	1.298075
4	multi	1.816084
5	biz	2.328597
6	cleanliness_rating	31.977015
7	bedrooms	7.810171
8	dist	17.539582
9	metro_dist	3.342570
10	attr_index_norm	28.591724
11	rest_index_norm	21.599504

This has improved the VIFs, but certain predictors like "cleanliness_rating", "dist", "attr_index_norm" and "rest_index_norm" still show concerningly high VIFs. We remove "attr_index" then check the VIFs again:

In [123…
```
X_drop2 = X_drop1.drop("attr_index_norm", axis=1)
vifs = [variance_inflation_factor(X_drop2.values, i) for i in range(X_drop2.shape[1
VIFSdf = pd.DataFrame({'Variable': X_drop2.columns, 'VIF': vifs})
print(VIFSdf)
```

```
              Variable        VIF
0          room_shared   1.014817
1         room_private   3.190164
2      person_capacity  12.652881
3    host_is_superhost   1.294854
4                multi   1.811972
5                  biz   2.307826
6   cleanliness_rating  27.282695
7             bedrooms   7.808463
8                 dist  14.970750
9           metro_dist   3.280400
10     rest_index_norm   7.250910
```

Again the VIFS have improved, but some still remain very high. Seeing from the correlation heatmap that "dist" has the most obvious linear relationships with other variables, we try removing that predictor next:

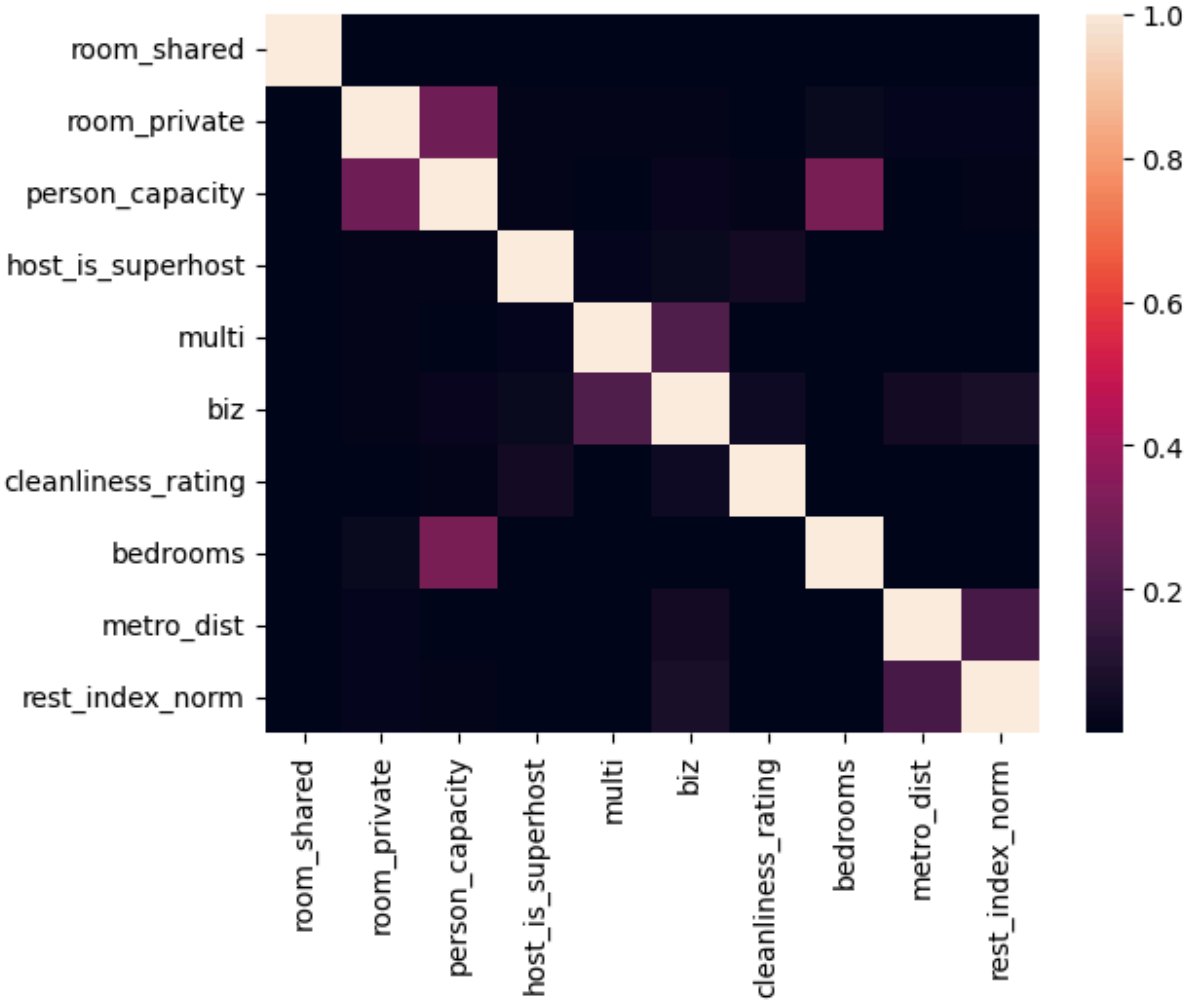
In [124…
```
X_drop3 = X_drop2.drop("dist", axis=1)
vifs = [variance_inflation_factor(X_drop3.values, i) for i in range(X_drop3.shape[1
VIFSdf = pd.DataFrame({'Variable': X_drop3.columns, 'VIF': vifs})
print(VIFSdf)
```

```
              Variable        VIF
0          room_shared   1.014553
1         room_private   3.131436
2      person_capacity  12.456245
3    host_is_superhost   1.290984
4                multi   1.807828
5                  biz   2.289574
6   cleanliness_rating  15.608946
7             bedrooms   7.806309
8           metro_dist   2.055033
9      rest_index_norm   4.611821
```

Now only two variables have VIF values greater than 10: "person_capacity" and "cleanliness_rating". We are reluctant to remove either from consideration, as they seem practically like reasonable predictors of the asking price on AirBnB. Since neither of their VIF values are overly problematic, we choose to keep all of the remaining variables.

The VIFs for each variable in this table look reasonably acceptable. We check the correlation heatmap once more to rule out any obviously correlated predictors.

In [125…
```
corrplot = sb.heatmap(np.square(X_drop3.corr()))
```



Confident that we have mitigated the most severe sources of multicollinearity, we select the remaining columns as the variables we'd like to use to predict logSum.

In [126…
```
X = X_drop3.copy()
```

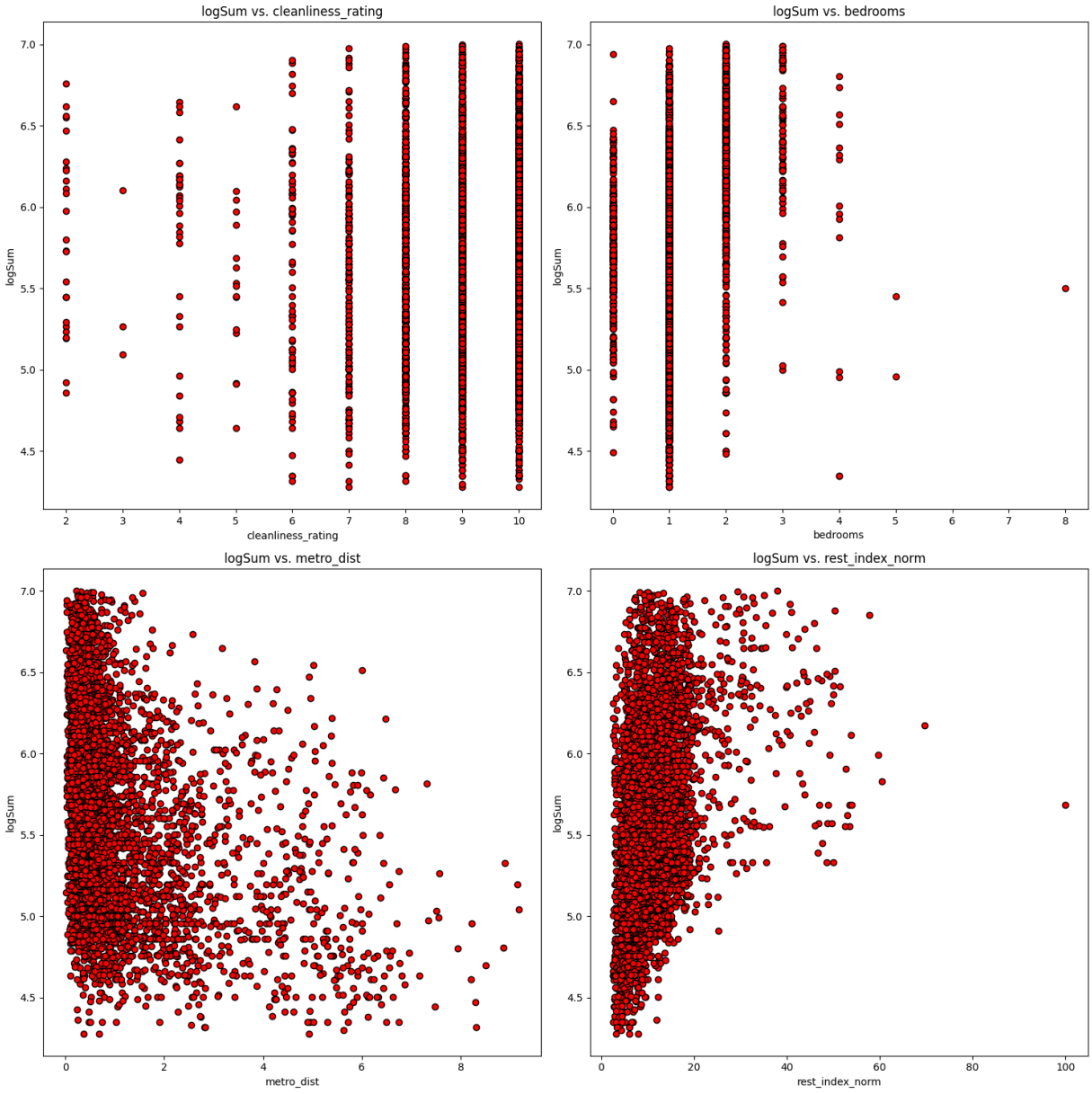
Univariate scatter plots

Now that the dataset is preprocessed, we examine univariate scatter plots of the predictors vs. the response to get a preliminary idea of the variable relationships.

In [127…
```
print(y.shape)

cont_variables = ["cleanliness_rating",
                  "bedrooms",
                  "metro_dist",
                  "rest_index_norm"]

fig = plt.figure(1, figsize=(15,15))

for (i, label) in enumerate(cont_variables):
    fig.add_subplot(2, 2, i+1)
    plt.scatter(X[label],
```

```
                y,
                c = "red",
                edgecolors = "black")
    plt.title("logSum vs. " + label)
    plt.ylabel("logSum")
    plt.xlabel(label)

plt.tight_layout()
plt.show()
```

```
(5245,)
```



- There appears to be a positive correlation between bedrooms and logSum.
- There may be a slight negative correlation between metro_dist and logSum.
- Additionally, there may be a slight positive correlation between rest_index_norm and logSum.

Split into training and test sets

Finally, we split the data into training and test sets. 70% of the observations are kept as training data, while the 30% is reserved to be used for testing the models.

In [128…
```
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    random_state=0,
                                                    test_size=0.3)
```

Now we are ready to begin testing various models for predicting logSum from the independent variables.

Modeling

Ordinary Linear Regression

We first try modeling logSum using ordinary linear regression. This model will attempt to express logSum as a linear combination of the predictor variables, selected to minimize the sum of squared errors.

In [129…
```
X_train_OLS = sm.add_constant(X_train)
results = sm.OLS(np.asarray(y_train), np.asarray(X_train_OLS)).fit()
print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.688
Model:                            OLS   Adj. R-squared:                  0.687
Method:                 Least Squares   F-statistic:                     807.3
Date:                Tue, 03 Dec 2024   Prob (F-statistic):               0.00
Time:                        19:21:50   Log-Likelihood:                -1040.9
No. Observations:                3671   AIC:                             2104.
Df Residuals:                    3660   BIC:                             2172.
Df Model:                          10
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          5.0911      0.051    100.608      0.000       4.992       5.190
x1            -0.7683      0.075    -10.302      0.000      -0.915      -0.622
x2            -0.6192      0.013    -46.999      0.000      -0.645      -0.593
x3             0.0950      0.006     14.985      0.000       0.083       0.107
x4             0.0421      0.015      2.784      0.005       0.012       0.072
x5             0.0249      0.014      1.830      0.067      -0.002       0.052
x6             0.0427      0.014      3.103      0.002       0.016       0.070
x7             0.0303      0.005      6.375      0.000       0.021       0.040
x8             0.1538      0.012     13.086      0.000       0.131       0.177
x9            -0.0716      0.005    -14.863      0.000      -0.081      -0.062
x10            0.0180      0.001     19.827      0.000       0.016       0.020
==============================================================================
Omnibus:                      354.737   Durbin-Watson:                   2.011
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              615.054
Skew:                           0.676   Prob(JB):                    2.77e-134
Kurtosis:                       4.481   Cond. No.                         220.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly spe
cified.
```

In [130…

```
OLS_train_MSE = mean_squared_error(y_train, results.predict(X_train_OLS))
print(OLS_train_MSE)
```

```
0.10323227049203328
```

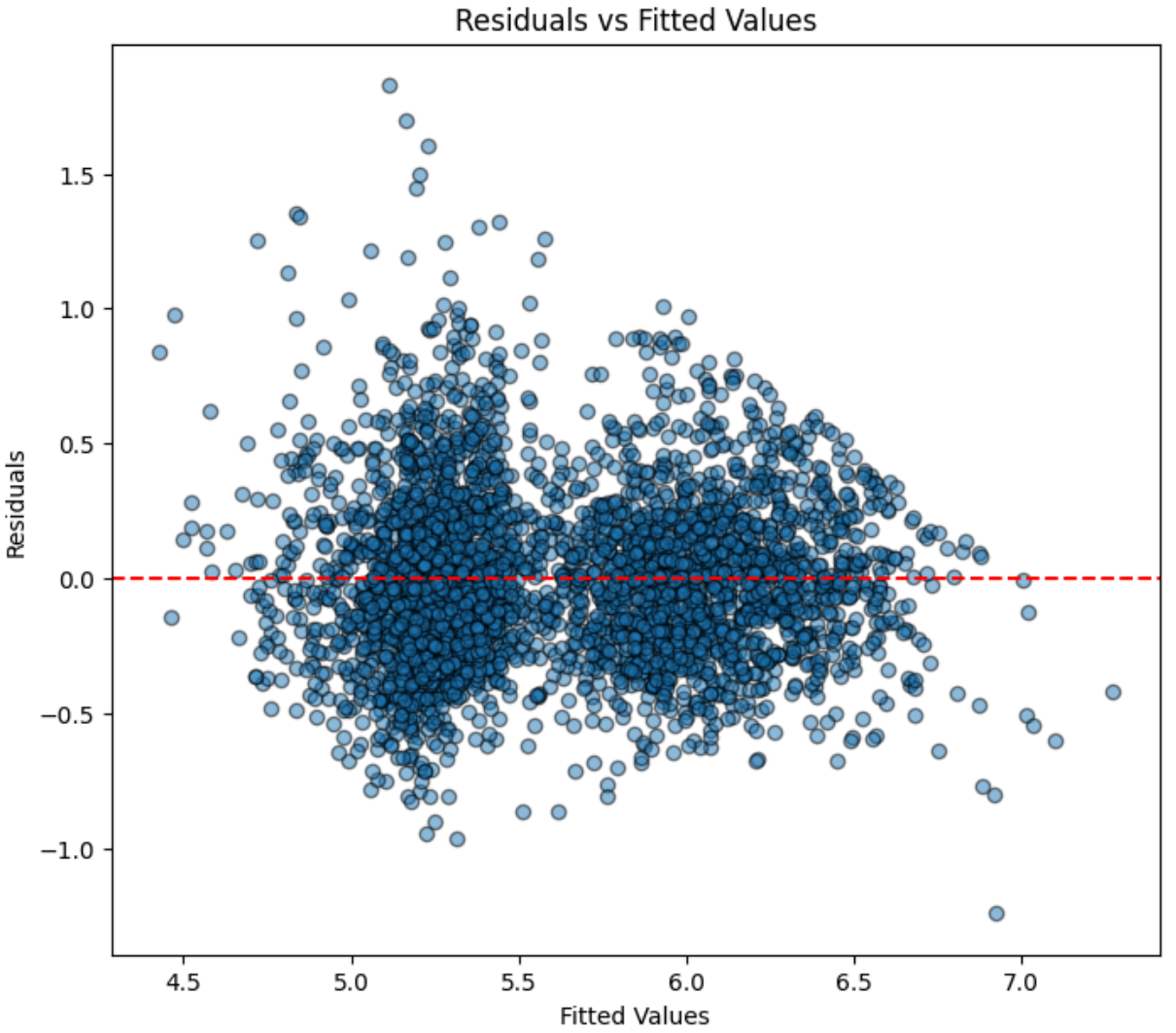
The training MSE for the Ordinary Linear Regression model is 0.10323

To visualize how well the Ordinary Linear Regression model is predicting logSum, we can view a residuals vs. fitted values plot. This plot will show us how far the predictions deviate from the true values within the training set.

In [131…

```
# residual plot
residuals = results.resid

plt.figure(figsize=(8, 7))
plt.scatter(results.fittedvalues,
            residuals,
            edgecolors="k",
            alpha=0.5)
```

```
plt.axhline(0,
            color='red',
            linestyle='--')
plt.xlabel('Fitted Values')
plt.ylabel('Residuals')
plt.title('Residuals vs Fitted Values')
plt.show()
```



The residuals appear to be mostly centered around zero, although they are slightly skewed positive, indicating that the OLS model might make underpredictions with high deviance from the true value.

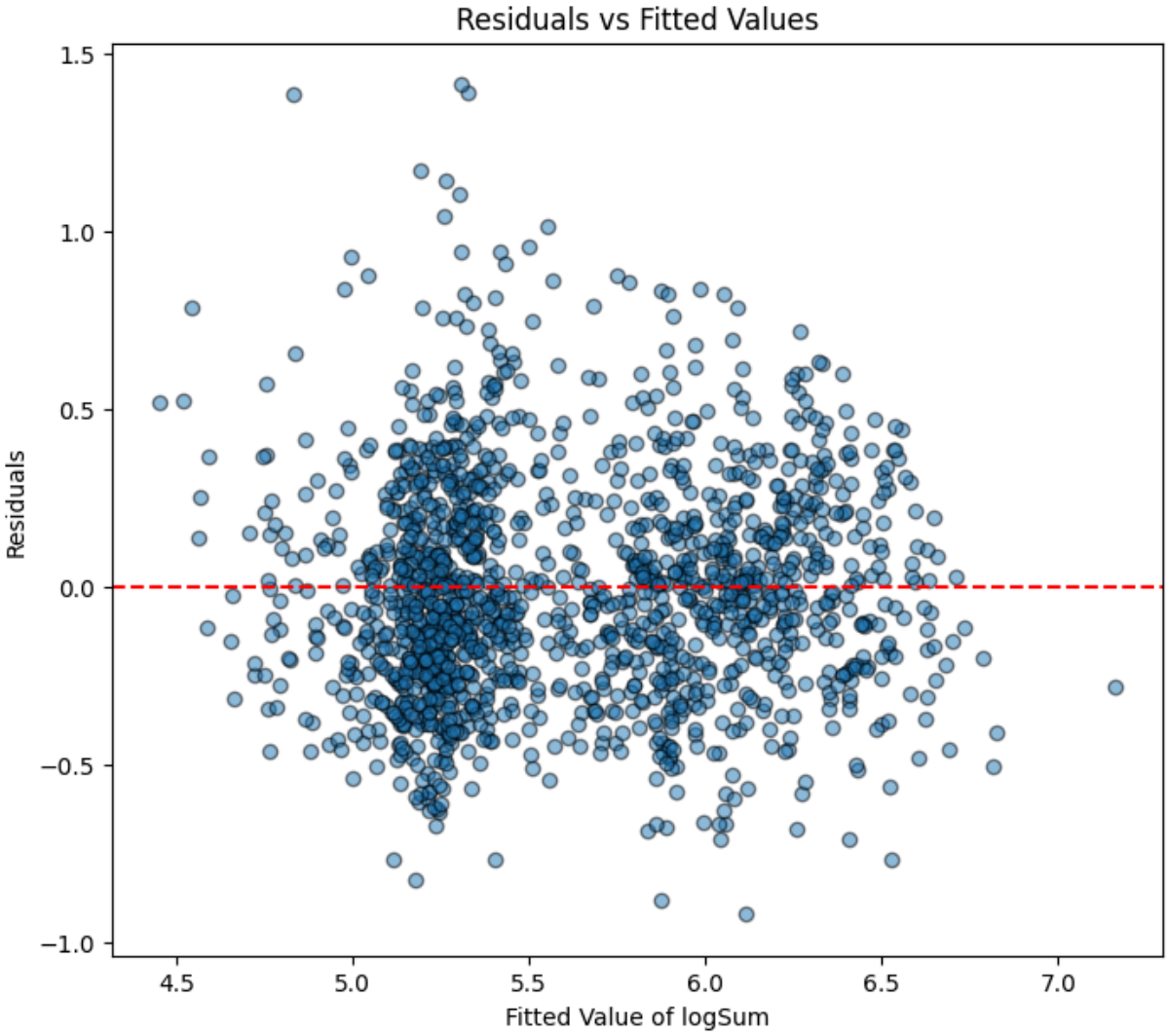
Now we calculate the MSE of the OLS model on the test set. This is a very important metric for measuring the performance of the predictive model, and it is what we will use to compare the models and select a final one. We also view a residuals vs. fitted plot for the test set, which looks very similar to the one for the training set.

In [132…

```
X_test_OLS = sm.add_constant(X_test)
OLS_test_MSE = mean_squared_error(y_test,
                                  results.predict(X_test_OLS))
print("Test MSE for Ordinary Least Squares Regression",
```

```
      OLS_test_MSE)
OLS_test_res = y_test - results.predict(X_test_OLS)

plt.figure(figsize=(8, 7))
plt.scatter(results.predict(X_test_OLS),
            OLS_test_res,
            edgecolors="k",
            alpha=0.5)
plt.axhline(0, color='red', linestyle='--')
plt.xlabel('Fitted Value of logSum')
plt.ylabel('Residuals')
plt.title('Residuals vs Fitted Values')
plt.show()
```

```
Test MSE for Ordinary Least Squares Regression 0.09827826280197151
```



Penalized Linear Regression

We are interested in testing various penalized regression models on the data. A penalized regression model undertakes essentially the same process as ordinary least squares, except that it also tries to minimize a specified penalty term.

This penalty term may improve the performance and generalizability of the model because it

acts to prevent the linear regression model from overfitting and reduces instability in parameter estimates. We will try three different types of penalties:

- Lasso (L1 Penalty)
- Ridge (L2 Penalty)
- Elastic Net (Combination of L1 and L2 penalty)

The strength of each penalty term is controlled by a hyperparameter alpha. This value will be optimized using cross-validation.

Lasso Regression

In order to use Lasso regression, we must first scale the predictor columns.

In [133…
```
# Standardize the features for Lasso
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)#this is standardized data for lasso
```

Now we use 5-fold cross-validation to determine the best value of alpha for the Lasso regression model. We also calculate the training MSE of the model using the optimal value of alpha.

In [134…
```
#Lasso Regression
lasso_cv = LassoCV(max_iter=5000,
                   cv=5,
                   alphas=np.logspace(-4, 2, 100),
                   random_state=42)
lasso_cv.fit(X_train, y_train)
lasso_best_alpha = lasso_cv.alpha_
lasso_training_mse = mean_squared_error(y_train,
                                        lasso_cv.predict(X_train))

print(f"Lasso: Best alpha = {lasso_best_alpha}, Training MSE = {lasso_training_mse:
```

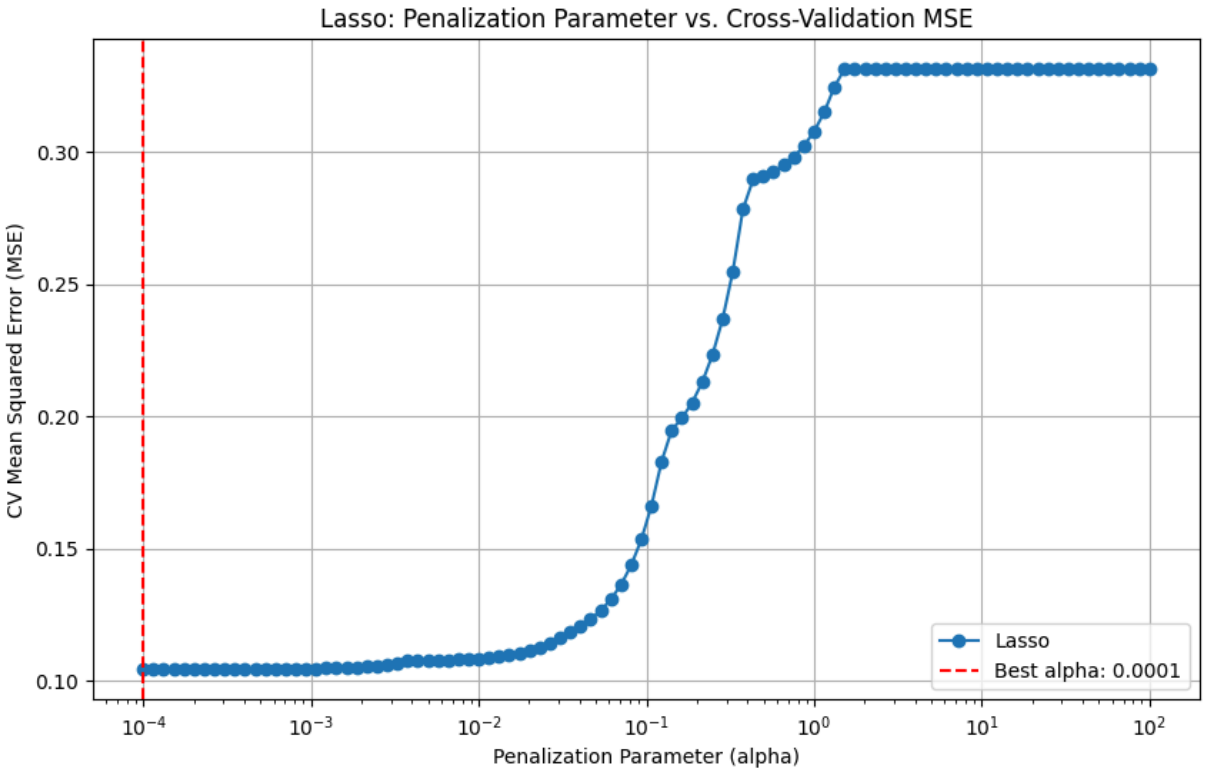
```
Lasso: Best alpha = 0.0001, Training MSE = 0.103235
```

Our results show that the best value for alpha is 0.0001 (which is the smallest one that we tried). This indicates that the model responds best to a minimal Lasso penalty, making it essentially equivalent to an ordinary least squares model. We can further deduce that the Lasso penalty is not useful for expanding the predictive power of the linear regression model.

We can visualize the performance of the model based on the value of the parameter alpha by plotting the parameter alpha against the corresponding model's cross-validation MSE.

In [135…
```
# Lasso: MSE for each alpha

fig, axes = plt.subplots(1, 1, figsize=(10, 6))
```

```
lasso_mse_mean = np.mean(lasso_cv.mse_path_, axis=1)
axes.plot(
    lasso_cv.alphas_,
    lasso_mse_mean,
    label="Lasso",
    marker='o'
)

axes.set_xscale('log')
axes.set_xlabel("Penalization Parameter (alpha)")
axes.set_ylabel("CV Mean Squared Error (MSE)")
axes.axvline(lasso_best_alpha,
             label = "Best alpha: "+ str(lasso_best_alpha),
             color = "red",
             linestyle = "--")
axes.set_title("Lasso: Penalization Parameter vs. Cross-Validation MSE")
axes.grid(True)
axes.legend()

plt.show()
```



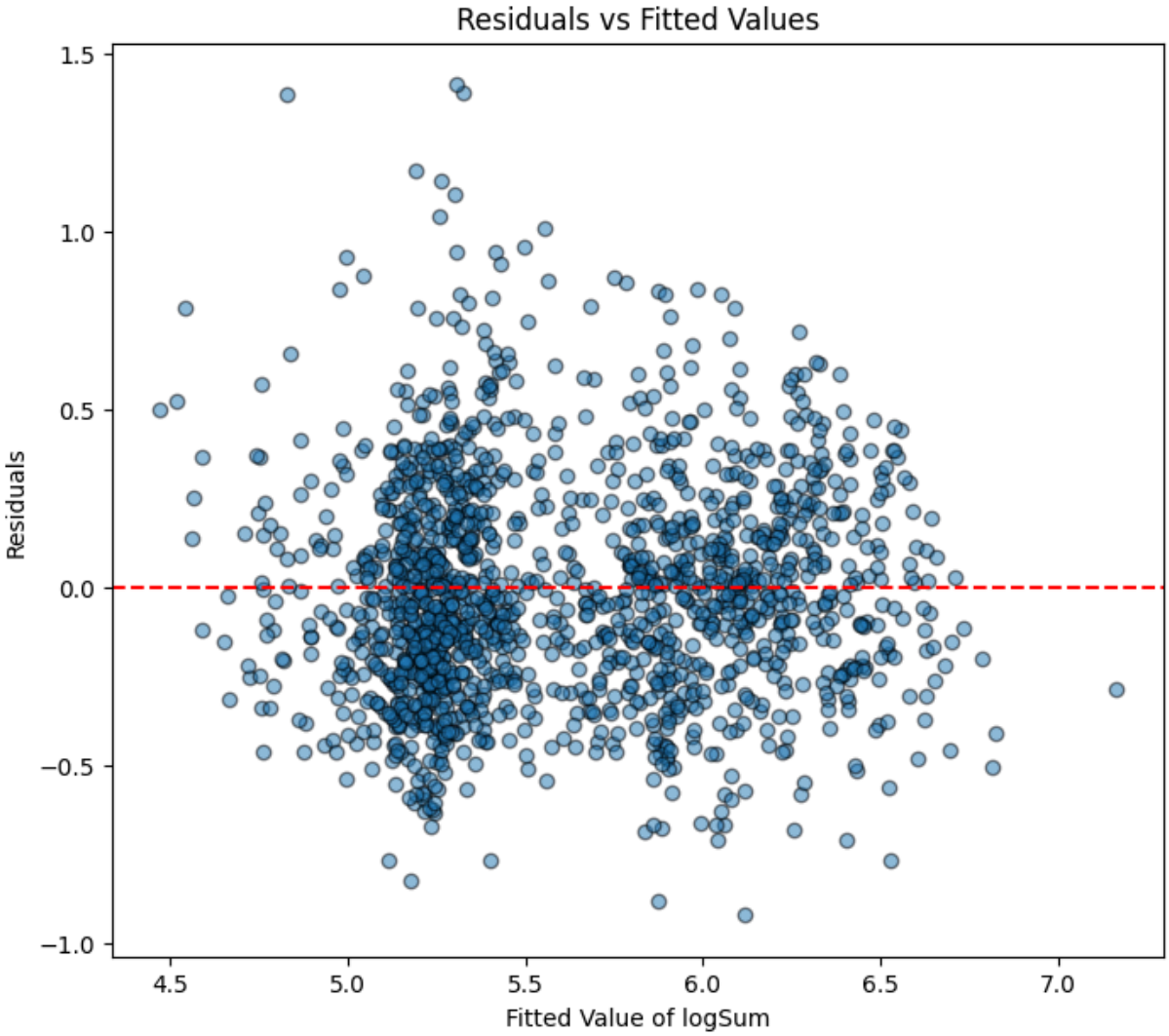
Test MSE for Lasso

In [136…

```
test_lasso = Lasso(alpha=0.0001)
results = test_lasso.fit(X_train, y_train)
lasso_test_mse = mean_squared_error(y_test,
                                    results.predict(X_test))
print("Test set MSE for Lasso regression model:", lasso_test_mse)

lasso_test_res = y_test - results.predict(X_test)

plt.figure(figsize=(8, 7))
```

```
plt.scatter(results.predict(X_test),
            lasso_test_res,
            edgecolors="k",
            alpha=0.5)
plt.axhline(0,
            color='red',
            linestyle='--')
plt.xlabel('Fitted Value of logSum')
plt.ylabel('Residuals')
plt.title('Residuals vs Fitted Values')
plt.show()
```

```
Test set MSE for Lasso regression model: 0.09827119765837194
```



The test MSE and residuals vs. fitted plot for Lasso regression are extremely similar to those of the ordinary least squares model. This is expected, as with such a small value of the hyperparameter alpha, the Lasso penalty term has almost no effect on the determination of the linear regression model.

Ridge Regression

Using ridge regression requires the predictor columns to be centered about 0. The following code cell performs this transformation.

In [137…
```
#Center the features for Ridge regression

X_train_centered = X_train - X_train.mean() #this is centered data for ridge
```

Using 5-fold cross-validation, we calculate that the best-performing alpha for Ridge Regression is about 0.3275.

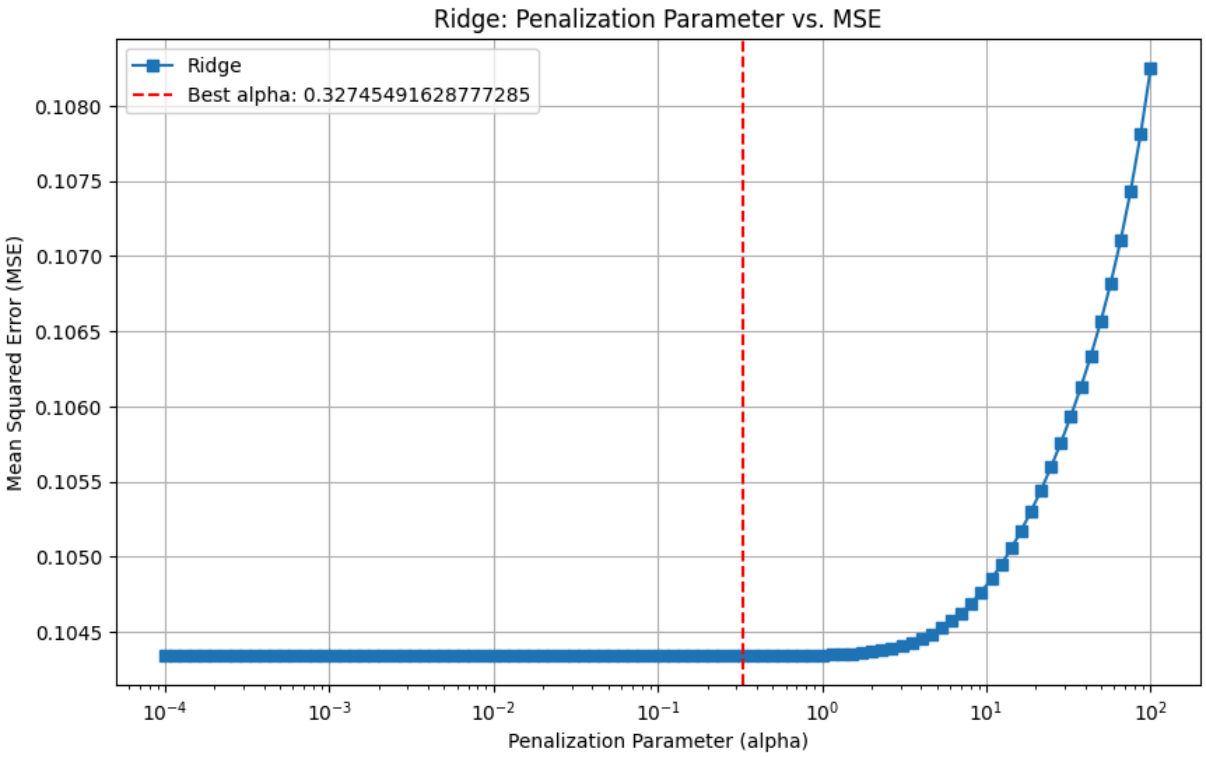
In [138…
```
#Ridge Regression
ridge_cv = RidgeCV(alphas=np.logspace(-4, 2, 100),
                   cv=5)
ridge_cv.fit(X_train_centered, y_train)

ridge_best_alpha = ridge_cv.alpha_
ridge_training_mse = mean_squared_error(y_train,
                                        ridge_cv.predict(X_train_centered))

print(f"Ridge: Best alpha = {ridge_best_alpha}, Training MSE = {ridge_training_mse:
```

```
Ridge: Best alpha = 0.32745491628777285, Training MSE = 0.103233
```

Interestingly, it seems that unlike the case of Lasso regression, for Ridge regression our cross-validation did not immediately select the lowest alpha value possible. This could imply that the Ridge penalty is effective for reducing the cross-validation MSE of the linear regression model. We want to visualize the performance of the Ridge regression model as a function of the hyperparameter alpha to see if this penalty has a significant effect on the model.

In [139…
```
# Ridge: MSE for each alpha

fig, axes = plt.subplots(1, 1, figsize=(10, 6))

ridge_mse = []
for alpha in np.logspace(-4, 2, 100):
    cv_scores = cross_val_score(Ridge(alpha),
                                X_train_centered,
                                y_train,
                                cv=5,
                                scoring="neg_mean_squared_error")
    ridge_mse.append(-np.mean(cv_scores))
axes.plot(
    np.logspace(-4, 2, 100),
    ridge_mse,
    label="Ridge",
    marker='s'
)

axes.set_xscale('log')
axes.set_xlabel("Penalization Parameter (alpha)")
axes.set_ylabel("Mean Squared Error (MSE)")
```

```
axes.axvline(ridge_best_alpha,
             label = "Best alpha: "+ str(ridge_best_alpha),
             color = "red",
             linestyle = "--")
axes.set_title("Ridge: Penalization Parameter vs. MSE")
axes.grid(True)
axes.legend()

plt.show()
```



From this plot depicting the cross-validation process, it appears that the best alpha for Ridge regression does not improve significantly on the performance of the model using the smallest value of alpha. As such, we do not expect the test MSE of the Ridge regression model to improve much on the ordinary least squares model.

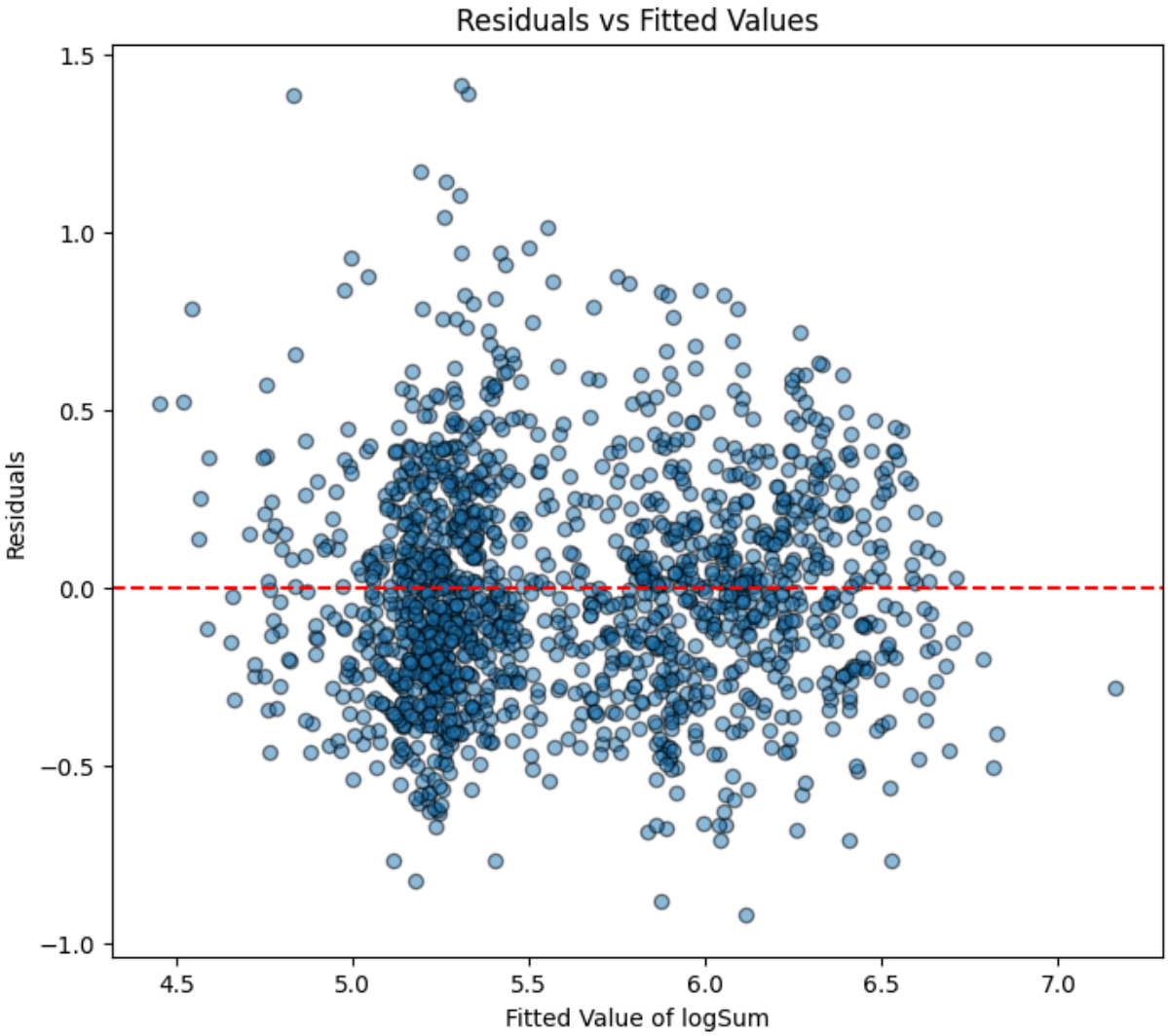
Ridge Test MSE

In [140…

```
test_ridge = Ridge(alpha=0.0001)
results = test_ridge.fit(X_train, y_train)
ridge_test_mse = mean_squared_error(y_test,
                                    results.predict(X_test))
print("Test set MSE for Ridge regression model:", ridge_test_mse)

ridge_test_res = y_test - results.predict(X_test)

plt.figure(figsize=(8, 7))
plt.scatter(results.predict(X_test),
            ridge_test_res,
            edgecolors="k",
            alpha=0.5)
plt.axhline(0,
```

```
            color='red',
            linestyle='--')
plt.xlabel('Fitted Value of logSum')
plt.ylabel('Residuals')
plt.title('Residuals vs Fitted Values')
plt.show()
```

```
Test set MSE for Ridge regression model: 0.0982782602495721
```



As expected, the test MSE and Residuals vs. Fitted plot for the Ridge Regression model look nearly identical to those of the ordinary least squares model.

Elastic Net Regression

For our final penalized linear regression model, we try using the elastic net penalty, which combines the Lasso and Ridge penalties. From our earlier results, we may not expect the Elastic Net penalty to be very effective at improving the test MSE of the model. Here we use cross-validation to determine the best value of the hyperparameter alpha, as well as the best value of the ratio of Lasso-to-Ridge penalty.

In [141…

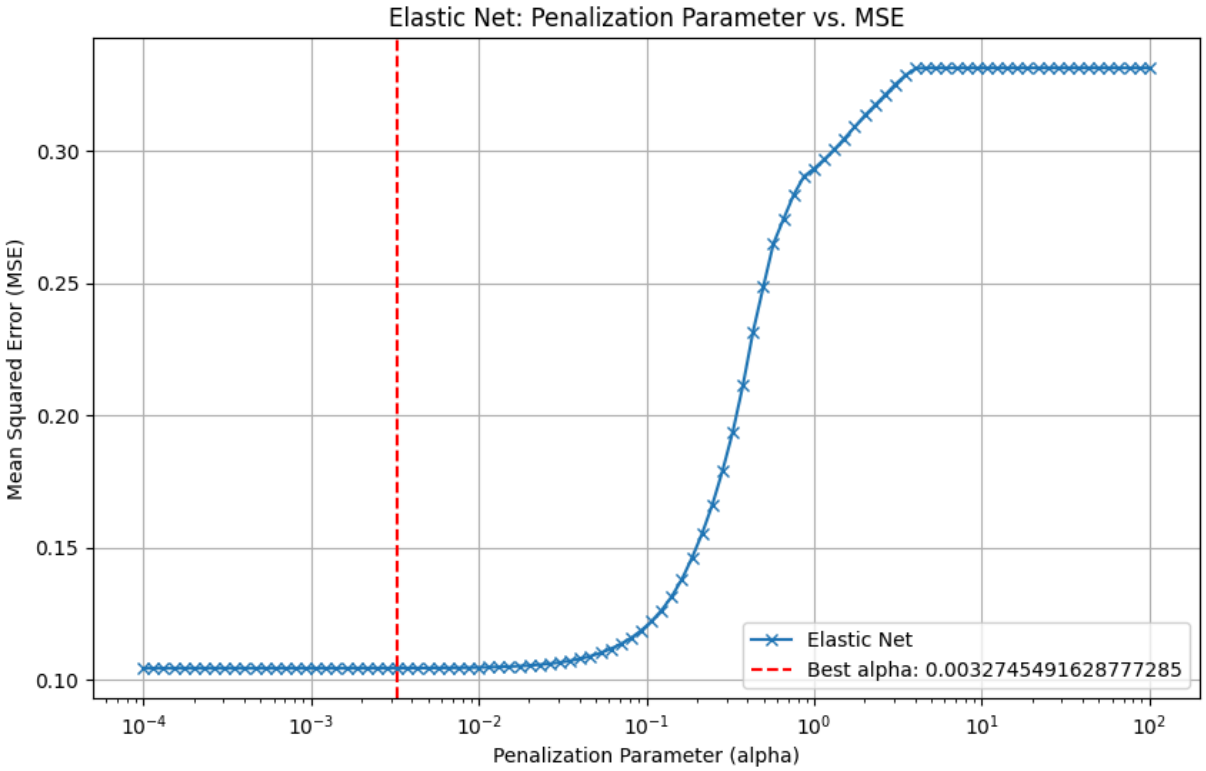
```
#Elastic Net Regression
elastic_net_cv = ElasticNetCV(
    l1_ratio=[0.1, 0.5, 0.7, 0.9],
    alphas=np.logspace(-4, 2, 100),
    cv=5,
    random_state=42
)
elastic_net_cv.fit(X_train_scaled, y_train)

elastic_net_best_alpha = elastic_net_cv.alpha_
elastic_net_best_l1_ratio = elastic_net_cv.l1_ratio_
elastic_net_training_mse = mean_squared_error(y_train,
                                              elastic_net_cv.predict(X_train_scaled

print("Elastic Net: Best alpha =", elastic_net_best_alpha)
print("Best l1_ratio =", elastic_net_best_l1_ratio)
print("Training MSE =", elastic_net_training_mse)
```

```
Elastic Net: Best alpha = 0.0032745491628777285
Best l1_ratio = 0.1
Training MSE = 0.10323569178032123
```

Once again it appears that a minimal penalty results in the best cross-validation MSE. We can also visualize this cross-validation process.

In [142…

```
# Elastic Net: MSE for each alpha

fig, axes = plt.subplots(1, 1, figsize=(10, 6))

elastic_net_mse_mean = np.mean(elastic_net_cv.mse_path_, axis=(0, 2))
axes.plot(
    elastic_net_cv.alphas_,
    elastic_net_mse_mean,
    label="Elastic Net",
    marker='x'
)

axes.set_xscale('log')
axes.set_xlabel("Penalization Parameter (alpha)")
axes.set_ylabel("Mean Squared Error (MSE)")
axes.axvline(elastic_net_best_alpha,
             label = "Best alpha: "+str(elastic_net_best_alpha),
             color = "red",
             linestyle = "--")
axes.set_title("Elastic Net: Penalization Parameter vs. MSE")
axes.grid(True)
axes.legend()

plt.show()
```

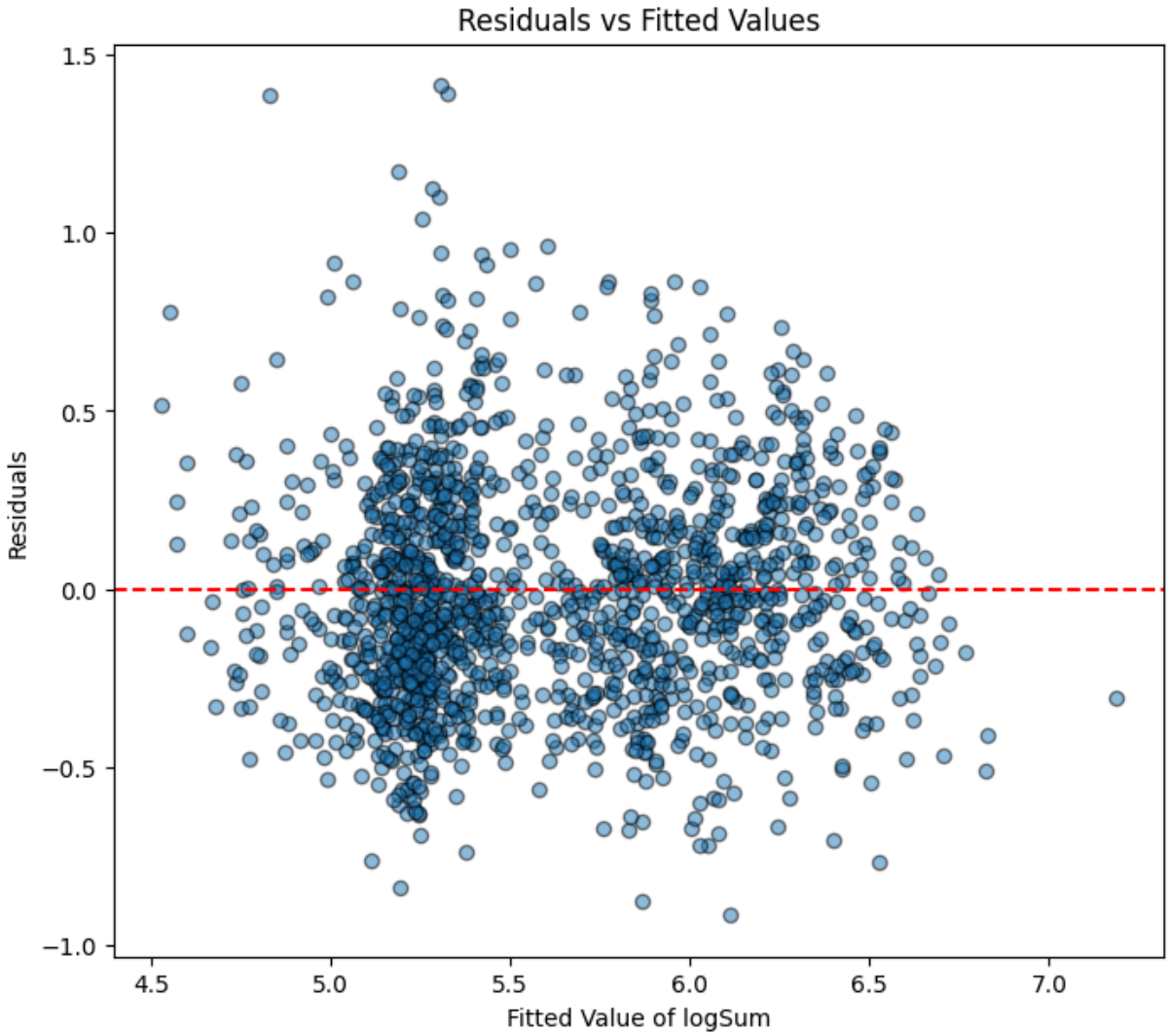


Elastic net test MSE

In [143…

```
test_elastic_net = ElasticNet(alpha= elastic_net_best_alpha)
results = test_elastic_net.fit(X_train, y_train)
elastic_net_test_mse = mean_squared_error(y_test,
                                          results.predict(X_test))
print("Test set MSE for Elastic Net regression model:",elastic_net_test_mse)

elastic_net_test_res = y_test - results.predict(X_test)

plt.figure(figsize=(8, 7))
plt.scatter(results.predict(X_test),
            elastic_net_test_res,
            edgecolors="k",
            alpha=0.5)
plt.axhline(0,
            color='red',
            linestyle='--')
plt.xlabel('Fitted Value of logSum')
plt.ylabel('Residuals')
plt.title('Residuals vs Fitted Values')
plt.show()
```

```
Test set MSE for Elastic Net regression model: 0.09908823248079651
```



Once again, it appears that the elastic net penalty did not help the linear regression model to outperform the ordinary least squares model.

Poisson Regression

We now attempt to predict logSum by using Poisson regression on the predictors.
This method models the response as a Poisson-distributed random variable with the mean being an exponentially-transformed linear function of the predictors.
We're interested in trying Poisson regression because it models the response as a strictly positive random variable, in contrast to linear regression which models the response as taking values from negative infinity to positive infinity.

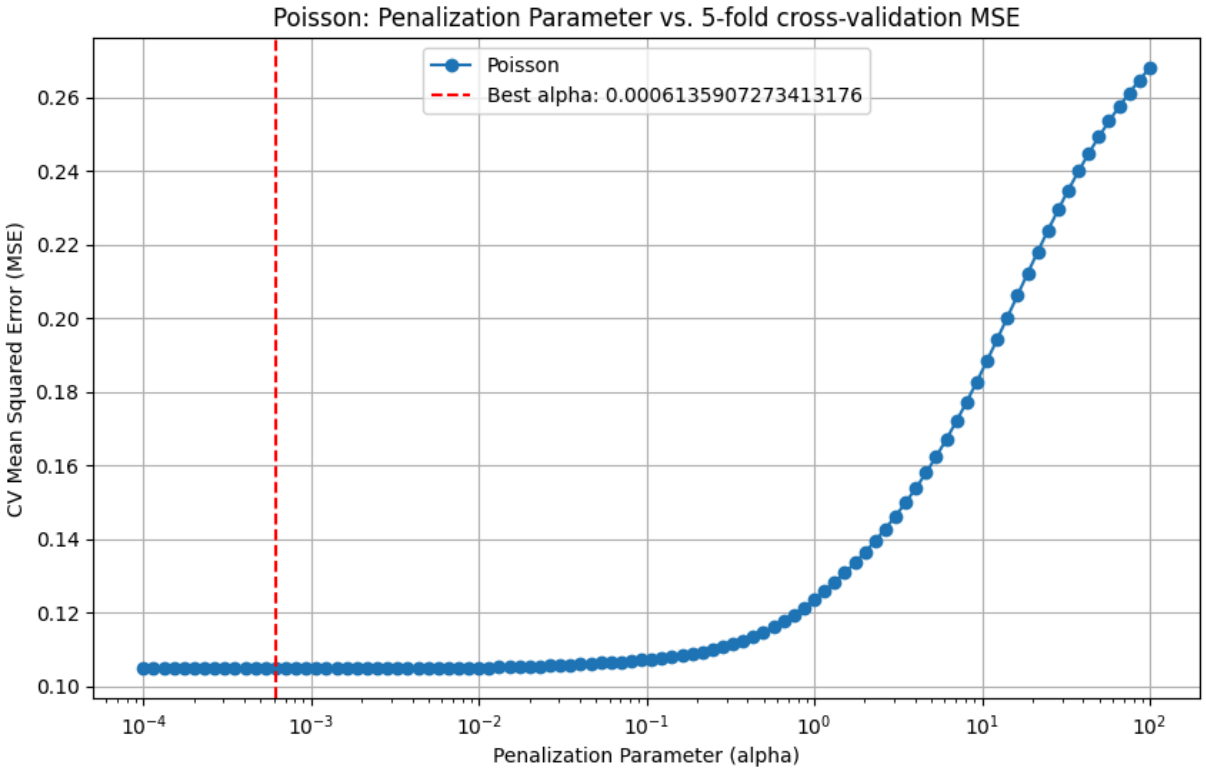
The sklearn function for Poisson regression automatically includes a Ridge penalty term parameterized by alpha. We used 5-fold cross-validation to optimize alpha.

In [144…

```
Poisson_cv_mse = []
alphas = np.logspace(-4,2,100)
for a in alphas:
```

```
    scores = cross_val_score(PoissonRegressor(alpha= a,
                                              solver="newton-cholesky"),
                                              X_train,
                                              y_train,
                                              scoring="neg_mean_squared_error")
    Poisson_cv_mse.append(-np.mean(scores))

poisson_best_alpha = alphas[np.argmin(Poisson_cv_mse)]

results = PoissonRegressor(alpha = poisson_best_alpha,
                           solver = "newton-cholesky").fit(X_train, y_train)

poisson_train_mse = mean_squared_error(y_train, results.predict(X_train))

print("Poisson: Best alpha =", poisson_best_alpha)
print("Training MSE =", poisson_train_mse)

fig, axes = plt.subplots(1, 1, figsize=(10, 6))

axes.plot(alphas,
          Poisson_cv_mse,
          label = "Poisson",
          marker = "o")
axes.set_xscale('log')
axes.set_xlabel("Penalization Parameter (alpha)")
axes.set_ylabel("CV Mean Squared Error (MSE)")
axes.axvline(poisson_best_alpha,
             label = "Best alpha: "+str(poisson_best_alpha),
             color = "red",
             linestyle = "--")
axes.set_title("Poisson: Penalization Parameter vs. 5-fold cross-validation MSE")
axes.grid(True)
axes.legend()
plt.show()
```

```
Poisson: Best alpha = 0.0006135907273413176
Training MSE = 0.10374460477229243
```



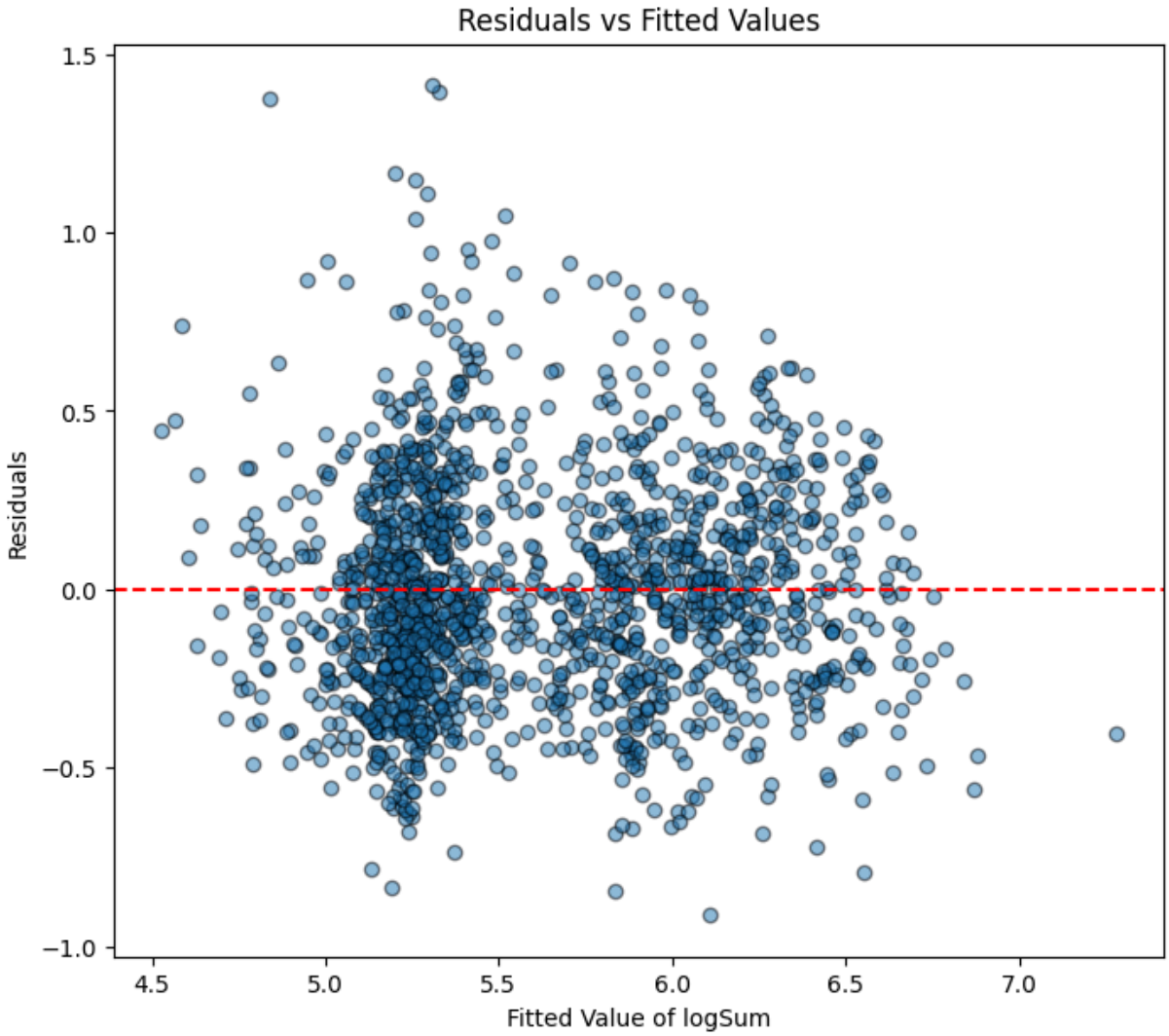
Similarly to the ordinary linear regression, the Poisson model seems to perform best with minimal penalization.

Poisson regression Test MSE:

In [145…

```
poisson_test_mse = mean_squared_error(y_test, results.predict(X_test))
print("Test set MSE of Poisson regression model:", poisson_test_mse)

poisson_test_res = y_test - results.predict(X_test)

plt.figure(figsize=(8, 7))
plt.scatter(results.predict(X_test),
            poisson_test_res,
            edgecolors="k", alpha=0.5)
plt.axhline(0, color='red',
            linestyle='--')
plt.xlabel('Fitted Value of logSum')
plt.ylabel('Residuals')
plt.title('Residuals vs Fitted Values')
plt.show()
```

```
Test set MSE of Poisson regression model: 0.09881328823752727
```



It appears that the Poisson model also did not improve significantly on the ordinary least squares model.

Cubic splines

We will now try predicting logSum using cubic spline regression. When using cubic splines, the range of each predictor is partitioned by k knots, allowing the effect of the predictor to be modeled by k+1 different cubic polynomials.

The polynomials are restricted such that they have a continuous second derivative.

After the basis function features have been calculated, the cubic spline is fit using Ridge Regression on the spline basis features. The Ridge penalty is added in order to regularize the parameter estimates, since representing as a spline basis results in a design matrix with many predictors.

We are interested in cubic spline regression because, unlike all of the previous methods, it is not restricted to being a linear model.

First we use 20-fold cross-validation to optimize the hyperparameter k, the number of knots for each predictor.

In [146…

```
# 2-25 knots
knotsamt = range(2,26)
splines_cv_errors = []

for numknots in knotsamt:
    spline = SplineTransformer(n_knots = numknots,
                               degree = 3,
                               knots='quantile')
    cubic_pipeline = make_pipeline(spline, Ridge(alpha=0.001))
    splines_cv_mse = -np.mean(cross_val_score(cubic_pipeline,
                                              X_train,
                                              y_train,
                                              cv=20,
                                              scoring="neg_mean_squared_error"))

    splines_cv_errors.append(splines_cv_mse)

optimalknots = knotsamt[np.argmin(splines_cv_errors)]

spline = SplineTransformer(n_knots = optimalknots,
                           degree = 3,
                           knots='quantile')

cubic_pipeline = make_pipeline(spline, Ridge(alpha=0.001))
cubic_pipeline.fit(X_train, y_train)
splines_train_mse = mean_squared_error(y_train,
                                       cubic_pipeline.predict(X_train))


print(f"Best number of knots = {optimalknots},",
      f"Training MSE = {splines_train_mse:.6f}")
```

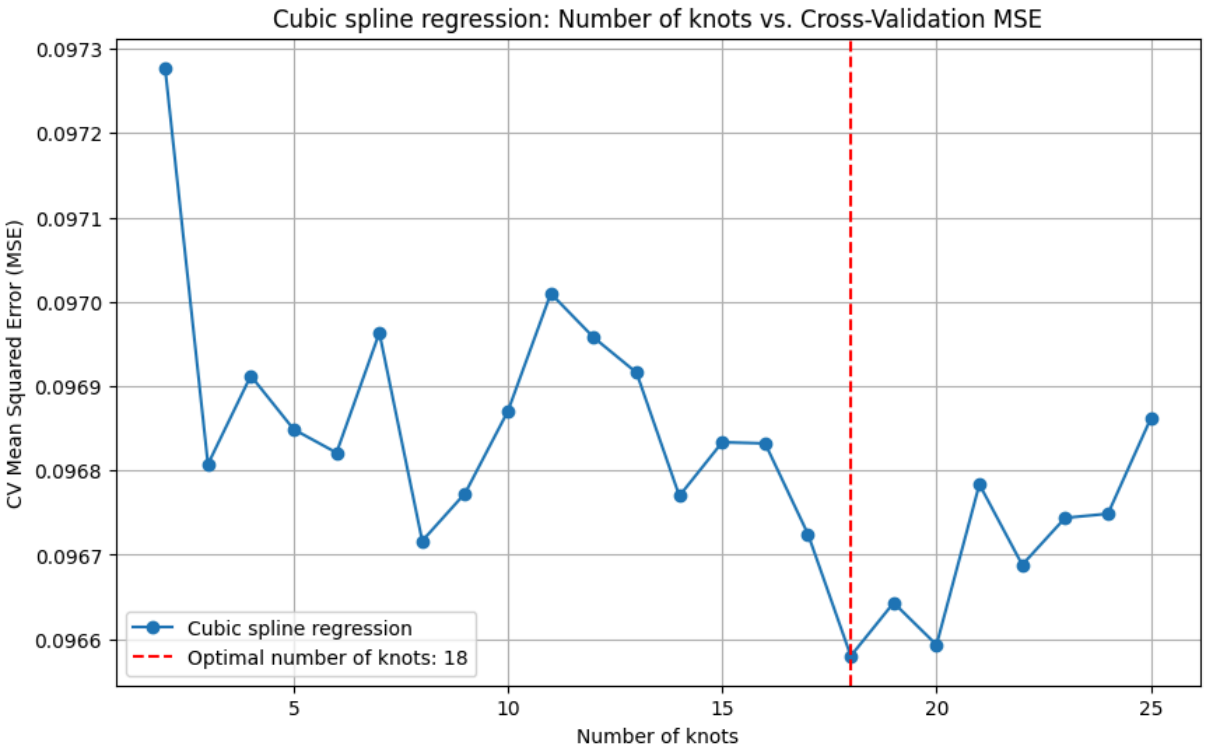
```
Best number of knots = 18, Training MSE = 0.093348
```

The number of knots resulting in the lowest cross-validation MSE was 18, as can be seen from the following plot.

In [147…

```
fig, axes = plt.subplots(1, 1, figsize=(10, 6))

axes.plot(knotsamt,
          splines_cv_errors,
          label = "Cubic spline regression",
          marker = "o")
axes.set_xlabel("Number of knots")
axes.set_ylabel("CV Mean Squared Error (MSE)")
axes.axvline(optimalknots,
             label = "Optimal number of knots: "+ str(optimalknots),
             color = "red",
             linestyle = "--")
axes.set_title("Cubic spline regression: Number of knots vs. Cross-Validation MSE")
axes.grid(True)
```

```
axes.legend()
plt.show()
```



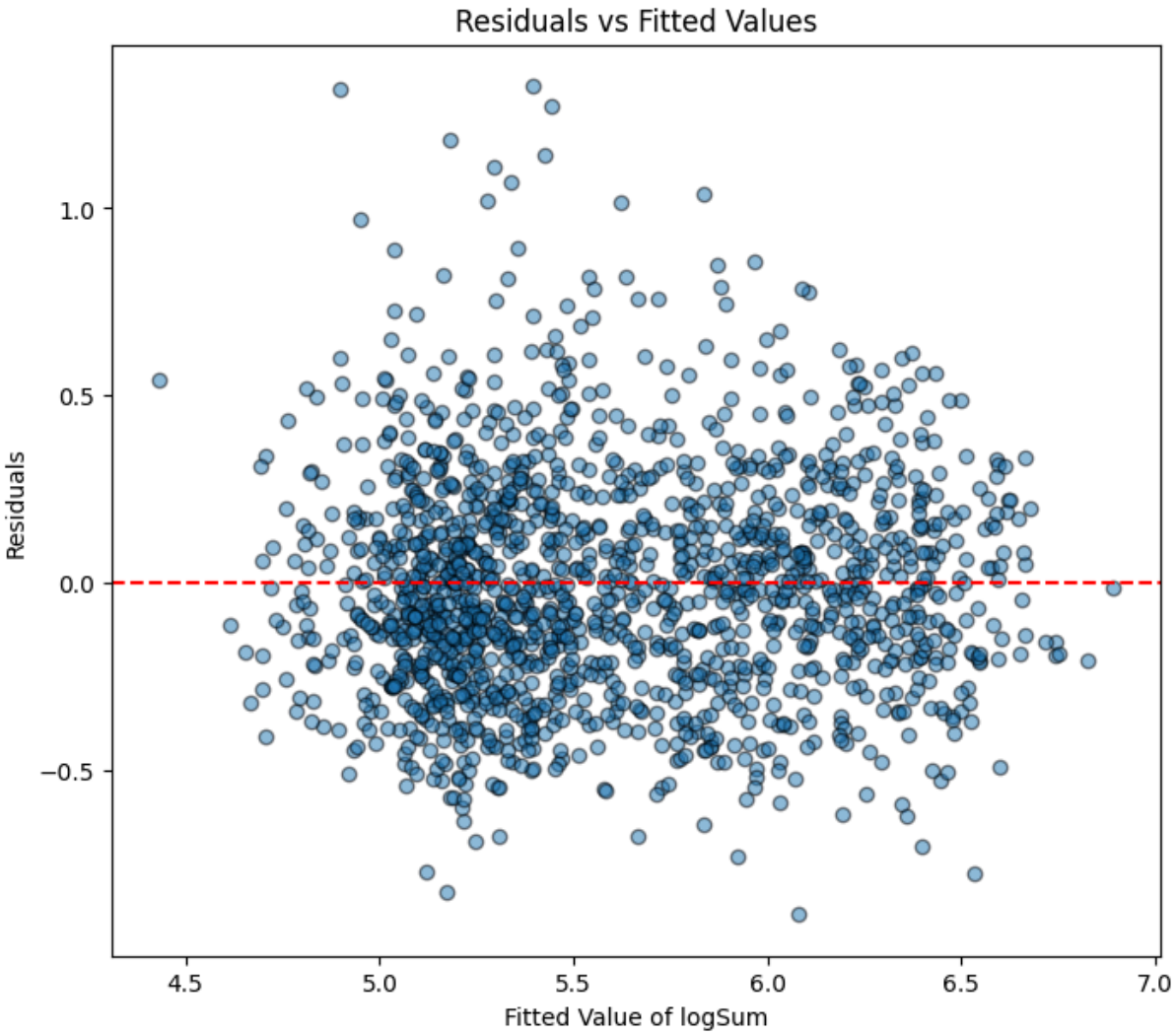
Cubic spline regression Test MSE:

In [148…
```
splines_test_mse = mean_squared_error(y_test,
                                      cubic_pipeline.predict(X_test))
print("Test set MSE for cubic spline regression:", splines_test_mse)
```

```
Test set MSE for cubic spline regression: 0.09016714199744544
```

This is the first method which has shown a significant improvement in test MSE over the ordinary least squares model. This may indicate that there is some nonlinear character to the relationship between the predictors and response which the previous linear models were simply unable to capture.

In [149…
```
splines_test_res = y_test - cubic_pipeline.predict(X_test)

plt.figure(figsize=(8, 7))
plt.scatter(cubic_pipeline.predict(X_test),
            splines_test_res,
            edgecolors="k",
            alpha=0.5)
plt.axhline(0,
            color='red',
            linestyle='--')
plt.xlabel('Fitted Value of logSum')
plt.ylabel('Residuals')
plt.title('Residuals vs Fitted Values')
plt.show()
```



Here we can view the Residuals vs. Fitted scatter plot for cubic spline regression on the test set. It appears more evenly spread and more closely balanced about zero than the scatter plots for the previous model.

Cumulative Model Evaluation

We will select our final model based on which one does the best at minimizing the test MSE. We list the test MSEs for each included model below:

In [150…

```
print("OLS Test MSE:", OLS_test_MSE)
print("Lasso Regression Test MSE:", lasso_test_mse)
print("Ridge Regression Test MSE:", ridge_test_mse)
print("Elastic Net Regression Test MSE:", elastic_net_test_mse)
print("Poisson Regression Test MSE:", poisson_test_mse)
print("Cubic Spline Regression Test MSE:", splines_test_mse)
```

```
OLS Test MSE: 0.09827826280197151
Lasso Regression Test MSE: 0.09827119765837194
Ridge Regression Test MSE: 0.0982782602495721
Elastic Net Regression Test MSE: 0.09908823248079651
Poisson Regression Test MSE: 0.09881328823752727
Cubic Spline Regression Test MSE: 0.09016714199744544
```

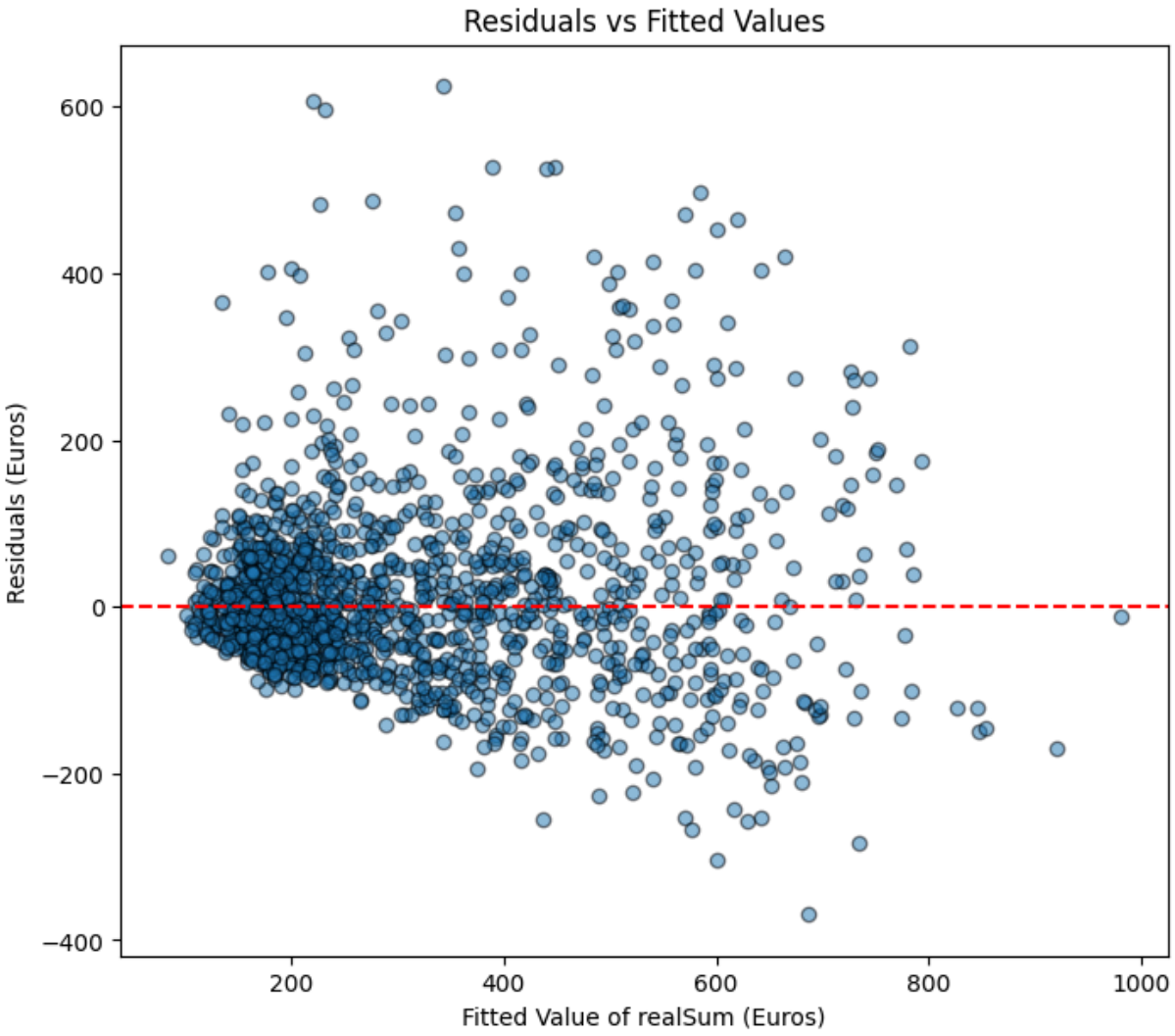
As seen from these results, none of the penalization techniques nor the Poisson regression were able to significantly improve on the test MSE of the ordinary least squares model. However, the cubic spline regression model does seem to significantly outperform all the other models in the context of test MSE. For this reason, we choose cubic spline regression as our best, final model. We will now perform further evaluation of the model.

Throughout this investigation we have been predicting and examining metrics in terms of logSum, which has obscure units. To get a more clear idea of how well this model can predict Airbnb prices, we revert the test set and predictions back to the units of realSum by exponentiating both of them. Now we can view the residuals and root-mean-squared-error for predictions of realSum, which have units of euros.

In [151…

```
splines_test_res = np.exp(y_test) - np.exp(cubic_pipeline.predict(X_test))
print("Test set RMSE (in euros):",
      root_mean_squared_error(np.exp(y_test),
                              np.exp(cubic_pipeline.predict(X_test))))

plt.figure(figsize=(8, 7))
plt.scatter(np.exp(cubic_pipeline.predict(X_test)),
            splines_test_res,
            edgecolors="k",
            alpha=0.5)
plt.axhline(0,
            color='red',
            linestyle='--')
plt.xlabel('Fitted Value of realSum (Euros)')
plt.ylabel('Residuals (Euros)')
plt.title('Residuals vs Fitted Values')
plt.show()
```

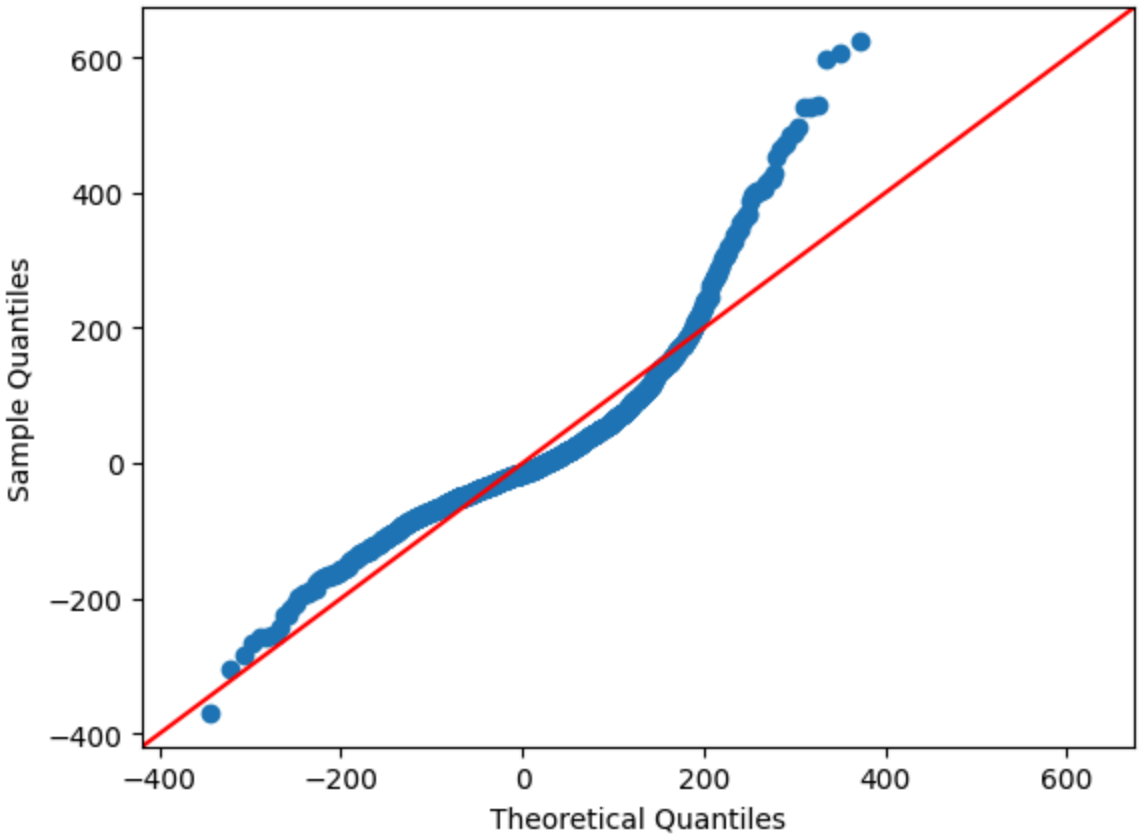
```
Test set RMSE (in euros): 111.99672657616031
```



From the RMSE and Residuals vs. Fitted plot, we can reasonably claim that the model predicts the asking price of most Airbnb properties within 100-200 dollars of their true value. We can also notice from the plot that the great majority of high-deviance observations tend to have a positive residual, which means that they were underpredicted by the model. We can try to confirm this observation by viewing a QQ-plot of the realSum residuals.

In [152…

```
realSum_res_qqplot = sm.qqplot(splines_test_res,
                               loc = np.mean(splines_test_res),
                               scale = np.std(splines_test_res),
                               line= '45')
```

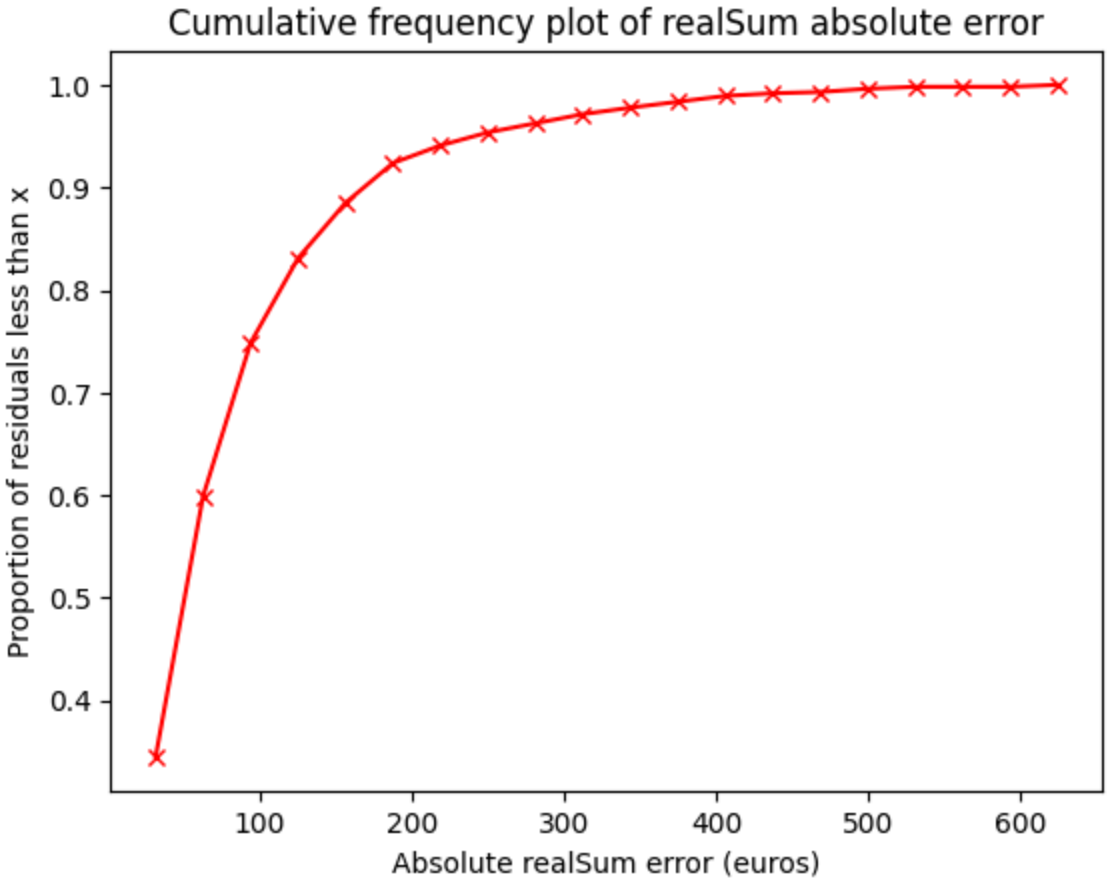


The deviance from the line at the right side of his QQ-plot confirms that there is a significant amount of high-deviance underpredictions. If we wanted to interpret this quality of the model, we might suggest that in the London Airbnb market, there are many properties that are greatly overpriced, and are listed with asking prices that far exceed those of listings with similar attributes. This exemplifies the model's usage for ensuring that a certain Airbnb listing is priced competitively according to the asking prices of similar properties.

We can also visualize the test set residuals of the model in another way. We create a cumulative frequency plot of the absolute error of each prediction.

In [153…

```
abs_residuals = np.abs(splines_test_res)
count, bins_count = np.histogram(abs_residuals,
                                 bins=20)
pdf = count/np.sum(count)
cdf = np.cumsum(pdf)

plt.plot(bins_count[1:],
         cdf,
         label="CDF",
         marker = "x",
         color="red")

plt.xlabel("Absolute realSum error (euros)")
plt.ylabel("Proportion of residuals less than x")
plt.title("Cumulative frequency plot of realSum absolute error")
plt.show()
```



For each x-value, the corresponding y-value indicates what proportion of the test predictions fell within x units of the real value. For example, we can see that about 75% of the listing prices were predicted within $100 of their true value. Also, about 90 200 of their true value.

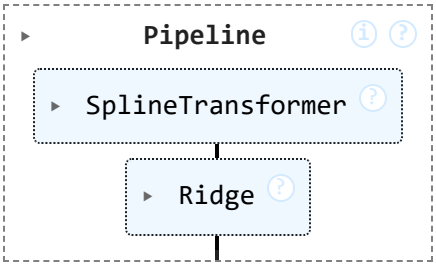
Model Analysis/Interpretation

The cubic spline regression model is additive in the effects of each variable, which means that we can use partial-dependence plots to visualize the effects of each predictor on the asking price of the Airbnb property. The partial dependence plot for a specific predictor visualizes the impact of adjusting that predictor while all other predictors remain fixed at their means.

In [154…

```
cubic_pipeline = make_pipeline(SplineTransformer(n_knots=optimalknots, degree=3, kn
cubic_pipeline.fit(X, y)
```

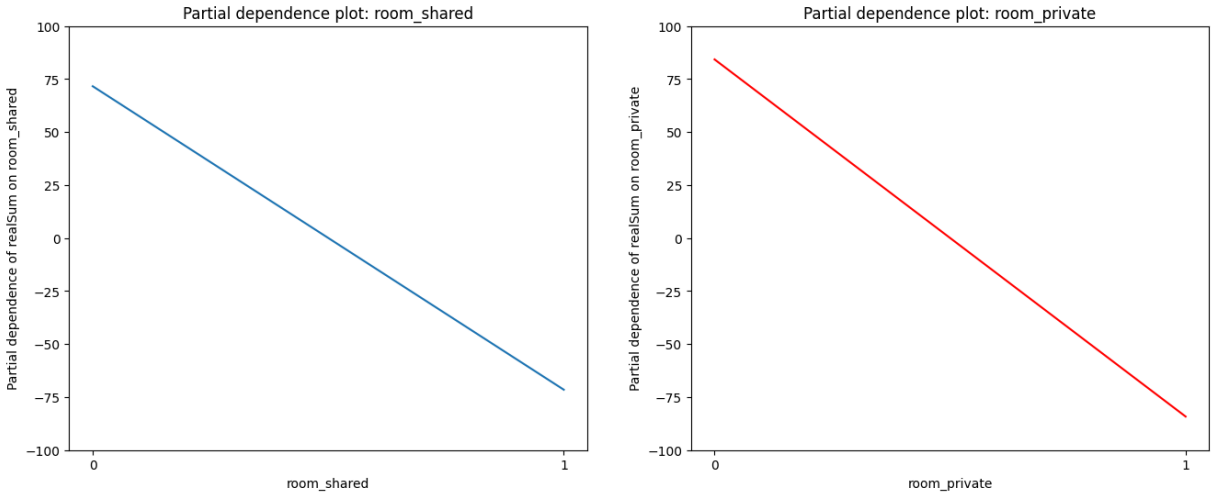
Out[154…



We first examine partial dependence plots for the "room_shared" and "room_private" dummy variables.

In [155…

```
fig, axes = plt.subplots(1, 2, figsize = (16,6))

pd_results = partial_dependence(cubic_pipeline, X, [0])
dep = np.exp(pd_results["average"][0])
dep -= np.mean(dep)
grid = pd_results["grid_values"][0]
axes[0].plot(grid, dep)
axes[0].set_title("Partial dependence plot: room_shared")
axes[0].set_xlabel("room_shared")
axes[0].set_ylabel("Partial dependence of realSum on room_shared")
axes[0].set_ybound(-100,100)
axes[0].set_xticks([0,1])

pd_results = partial_dependence(cubic_pipeline, X, [1])
dep = np.exp(pd_results["average"][0])
dep -= np.mean(dep)
grid = pd_results["grid_values"][0]
axes[1].plot(grid, dep, c="red")
axes[1].set_title("Partial dependence plot: room_private")
axes[1].set_xlabel("room_private")
axes[1].set_ylabel("Partial dependence of realSum on room_private")
axes[1].set_ybound(-100,100)
axes[1].set_xticks([0,1])

plt.show()
```

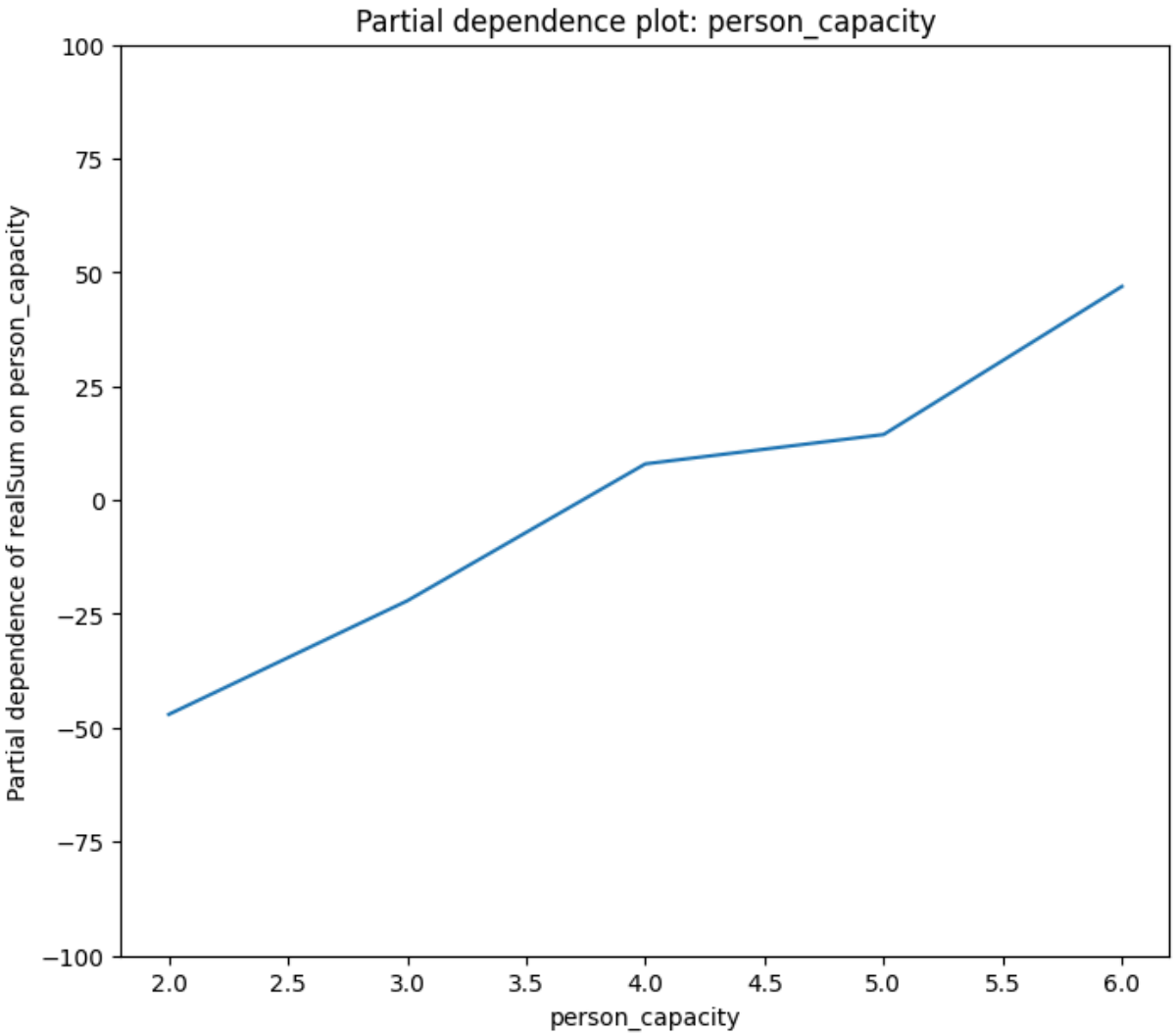


These plots indicate that if the listing is for either a shared or private room, the predicted asking price may drop by as much as 150 euros. This makes sense, as the baseline room type is "entire home/apartment," and it is obvious that a stay in a private or shared room would tend to sell for much less than a stay in an entire residence.

In [156…

```
fig, axes = plt.subplots(1, 1, figsize = (8,7))

pd_results = partial_dependence(cubic_pipeline, X, [2])
dep = np.exp(pd_results["average"][0])
```

```
dep -= np.mean(dep)
grid = pd_results["grid_values"][0]
axes.plot(grid, dep)
axes.set_title("Partial dependence plot: person_capacity")
axes.set_xlabel("person_capacity")
axes.set_ylabel("Partial dependence of realSum on person_capacity")
axes.set_ybound(-100,100)
plt.show()
```

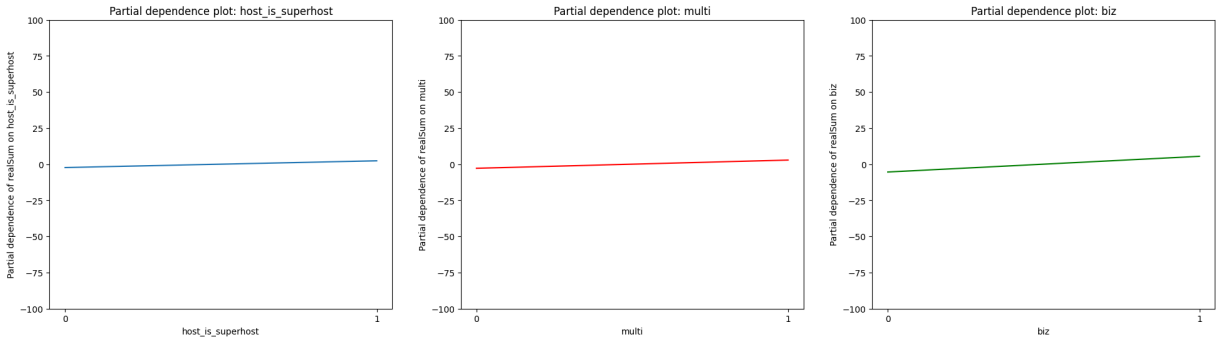


The partial dependence plot for person_capacity also displays an expected result: the asking price tends to increase along with the capacity of the listing. This aligns with the real-world situation, as properties which can accommodate a greater number of people are probably bigger and less available. The plot indicates that a property with a capacity of 6 people may cost roughly 100 euros more than a property that only fits two people.

In [157…

```
fig, axes = plt.subplots(1, 3, figsize = (24,6))

pd_results = partial_dependence(cubic_pipeline, X, [3])
dep = np.exp(pd_results["average"][0])
dep -= np.mean(dep)
grid = pd_results["grid_values"][0]
axes[0].plot(grid, dep)
```

```
axes[0].set_title("Partial dependence plot: host_is_superhost")
axes[0].set_xlabel("host_is_superhost")
axes[0].set_ylabel("Partial dependence of realSum on host_is_superhost")
axes[0].set_ybound(-100,100)
axes[0].set_xticks([0,1])

pd_results = partial_dependence(cubic_pipeline, X, [4])
dep = np.exp(pd_results["average"][0])
dep -= np.mean(dep)
grid = pd_results["grid_values"][0]
axes[1].plot(grid, dep, c="red")
axes[1].set_title("Partial dependence plot: multi")
axes[1].set_xlabel("multi")
axes[1].set_ylabel("Partial dependence of realSum on multi")
axes[1].set_ybound(-100,100)
axes[1].set_xticks([0,1])

pd_results = partial_dependence(cubic_pipeline, X, [5])
dep = np.exp(pd_results["average"][0])
dep -= np.mean(dep)
grid = pd_results["grid_values"][0]
axes[2].plot(grid, dep, c="green")
axes[2].set_title("Partial dependence plot: biz")
axes[2].set_xlabel("biz")
axes[2].set_ylabel("Partial dependence of realSum on biz")
axes[2].set_ybound(-100,100)
axes[2].set_xticks([0,1])
plt.show()
```

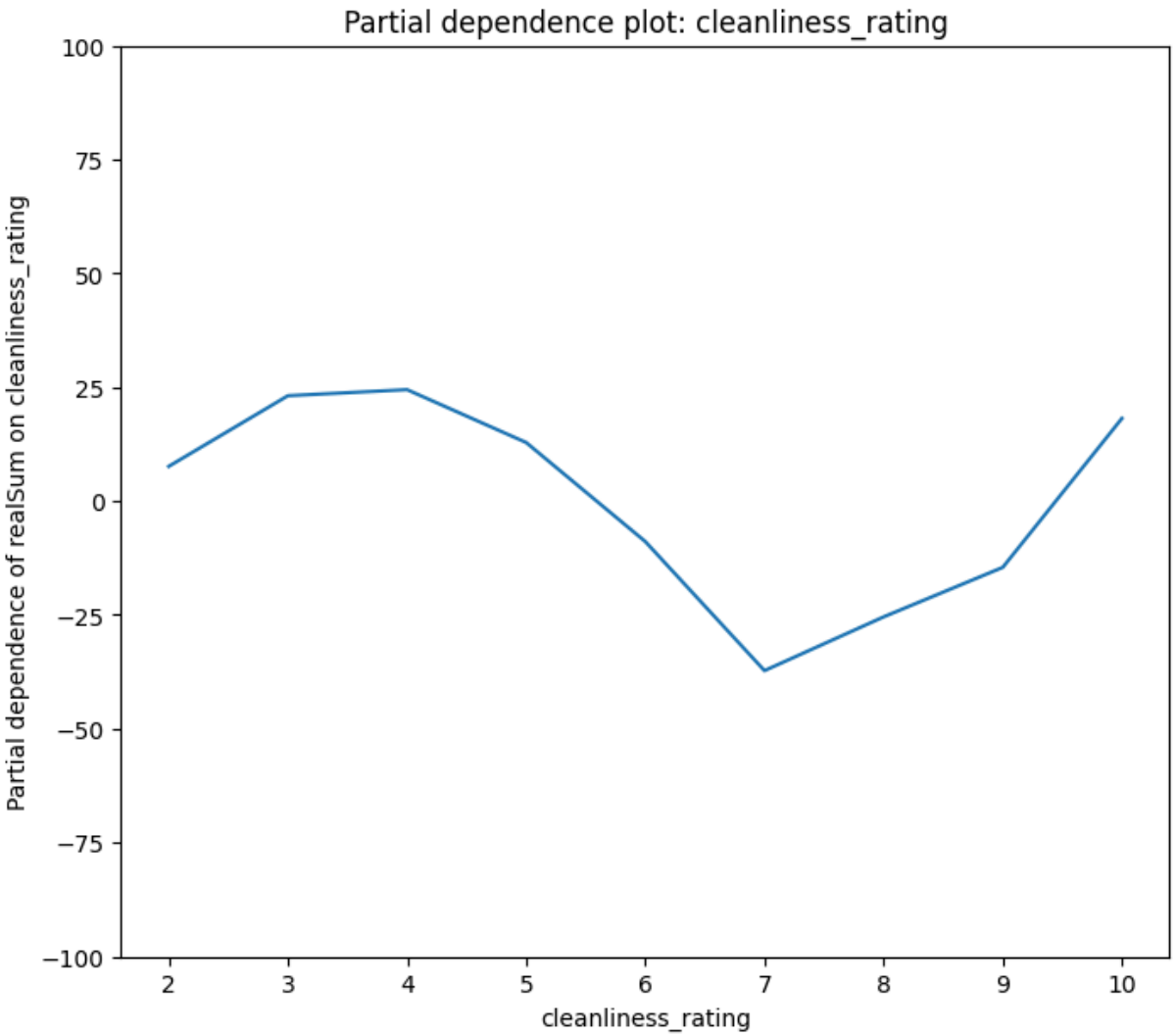


The partial dependence plots for host_is_superhost, multi, and biz are all very near 0. This indicates that none of these host-centered categorical variables are very significant predictors of realSum in the model.

In [158…

```
fig, axes = plt.subplots(1, 1, figsize = (8,7))

pd_results = partial_dependence(cubic_pipeline, X, [6])
dep = np.exp(pd_results["average"][0])
dep -= np.mean(dep)
grid = pd_results["grid_values"][0]
axes.plot(grid, dep)
axes.set_title("Partial dependence plot: cleanliness_rating")
axes.set_xlabel("cleanliness_rating")
axes.set_ylabel("Partial dependence of realSum on cleanliness_rating")
```

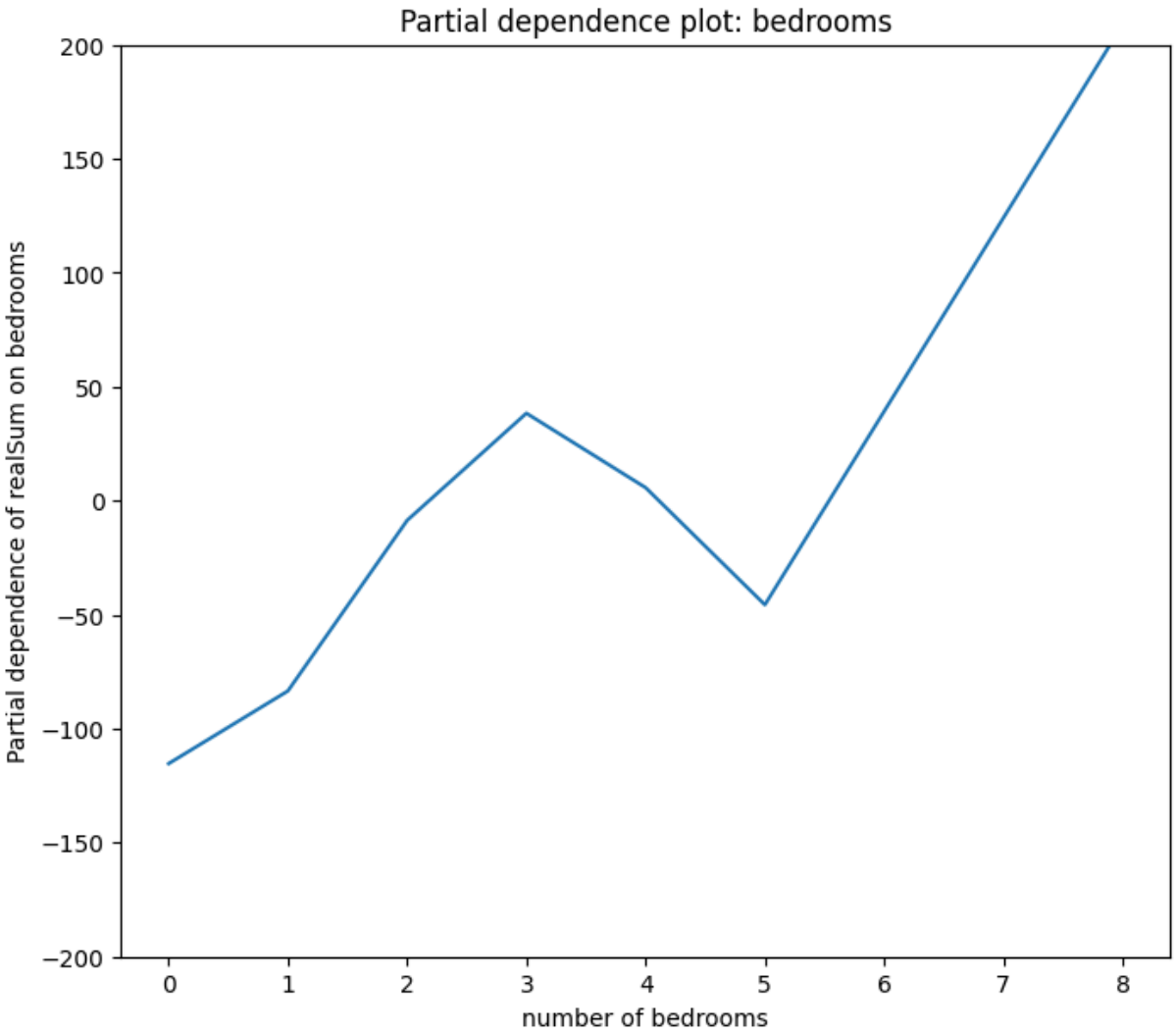
```
axes.set_ybound(-100,100)
plt.show()
```



The partial dependence plot for cleanliness_rating is difficult to interpret. It appears to peak at around 3-4, but it is counter-intuitive that unclean properties would sell for higher prices than the cleanest ones. One possible explanation is that there is some undetected confounding variable. Perhaps the properties with lower cleanliness ratings tend to be in busier parts of the city, and therefore have higher asking prices.

In [159…

```
fig, axes = plt.subplots(1, 1, figsize = (8,7))

pd_results = partial_dependence(cubic_pipeline, X, [7])
dep = np.exp(pd_results["average"][0])
dep -= np.mean(dep)
grid = pd_results["grid_values"][0]
axes.plot(grid, dep)
axes.set_title("Partial dependence plot: bedrooms")
axes.set_xlabel("number of bedrooms")
axes.set_ylabel("Partial dependence of realSum on bedrooms")
axes.set_ybound(-200,200)
plt.show()
```

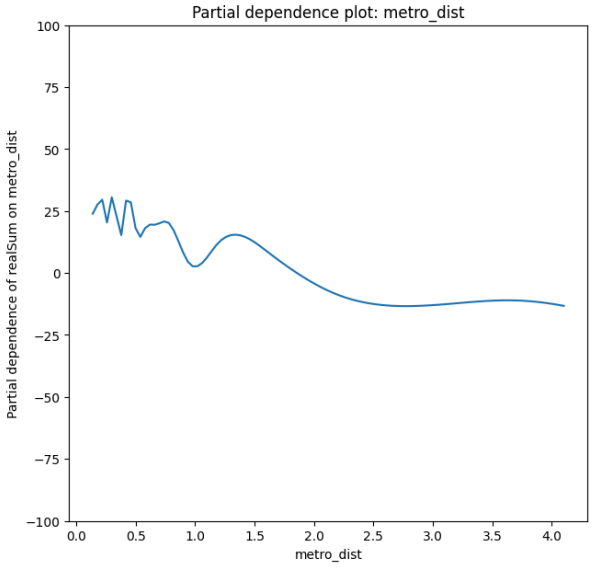


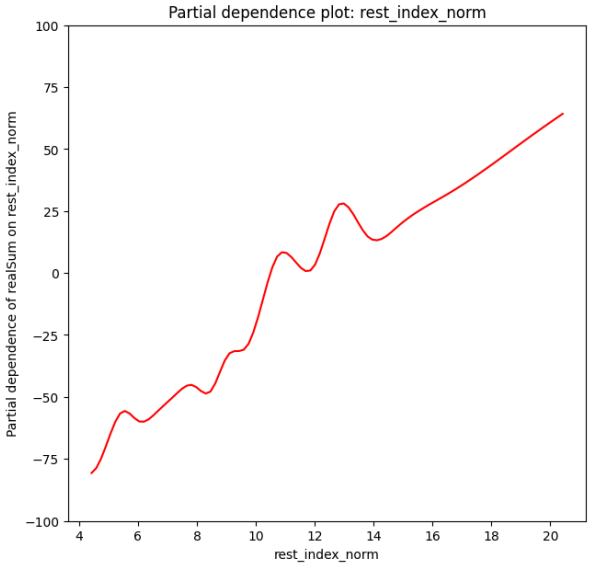
This partial dependence plot roughly confirms the natural assumption that properties with more bedrooms would be listed with higher asking prices. The downturn in the middle of the graph is not as easily interpreted. Recall from the histogram of bedrooms that there were very few observations with more than 3 bedrooms. The strange behavior of this partial dependency plot may be indirectly due to the few number of samples with 4-8 bedrooms. Still, we are able to see that properties with 3 bedrooms could sell for roughly 100 euros more than properties with only 1 bedroom.

In [160…

```
fig, axes = plt.subplots(1, 2, figsize = (16,7))

pd_results = partial_dependence(cubic_pipeline, X, [8])
dep = np.exp(pd_results["average"][0])
dep -= np.mean(dep)
grid = pd_results["grid_values"][0]
axes[0].plot(grid, dep)
axes[0].set_title("Partial dependence plot: metro_dist")
axes[0].set_xlabel("metro_dist")
axes[0].set_ylabel("Partial dependence of realSum on metro_dist")
axes[0].set_ybound(-100,100)

pd_results = partial_dependence(cubic_pipeline, X, [9])
```

```
dep = np.exp(pd_results["average"][0])
dep -= np.mean(dep)
grid = pd_results["grid_values"][0]
axes[1].plot(grid, dep, c="red")
axes[1].set_title("Partial dependence plot: rest_index_norm")
axes[1].set_xlabel("rest_index_norm")
axes[1].set_ylabel("Partial dependence of realSum on rest_index_norm")
axes[1].set_ybound(-100,100)
plt.show()
```





The partial dependence plot for metro_dist confirms the notion that properties within close proximities of metro stations will have higher prices than others. From the plot, it appears that the properties within 2 km of a metro station may sell for 15-40 euros more than properties farther away.

The other plot shows that rest_index_norm is a strong predictor of price, and properties with a good restaurant index tend to be much more expensive.

Conclusions

This investigation found that characteristics of the property, such as person capacity and number of bedrooms, and the surroundings of the property, such as the distance to metro stations and restaurants, are very strong predictors of asking price for weekend Airbnbs in London.

The best model explored in this investigation used regression with cubic splines, which outperformed strictly linear methods like ordinary least squares and Poisson regression. The partial dependence plots revealed that prices for entire homes/apartments tended to be much higher than those for private rooms or shared rooms. It was found that the price was positively associated with:

- person capacity

- number of bedrooms
- restaurant index

Price was found to be negatively associated with:

- distance from metro stations

Host-centered predictors such as the number of listings they owned and whether or not they were listed as a superhost were found to be insignificant as predictors of price.

It is possible that the cubic spline regressor performed best because it captured nonlinear character of the predictor-response relationship. For this reason, the research could be expanded by considering the use of other nonlinear or nonparametric methods such as random forests or deep learning. The dataset could also be expanded to listings in other European cities, in order to examine the confounding effect of location on Airbnb prices.

References

1. About us. (2016, October 25). Airbnb Newsroom. https://news.airbnb.com/about-us/

2. Gibbs, C., Guttentag, D., Gretzel, U., Morton, J., & Goodwill, A. (2018). Pricing in the sharing economy: A hedonic pricing model applied to airbnb listings. Journal of Travel & Tourism Marketing, 35, 46–56. https://doi.org/10.1080/10548408.2017.1308292

3. Kim, J., Jang, S., Kang, S., & Kim, S. H. J. (2018). Why are hotel room prices different? Exploring spatially varying relationships between room price and hotel attributes. Journal of Business Research, 1–12. https://doi.org/10.1016/j.jbusres.2018.09.006

4. London. (n.d.). Inside Airbnb. Retrieved November 19, 2024, from https://insideairbnb.com/london/

5. Tang, L. R., Kim, J., & Wang, X. (2019). Estimating spatial effects on peer-to-peer accommodation prices: Towards an innovative hedonic model approach. International Journal of Hospitality Management, 81, 43–53. https://doi.org/10.1016/j.ijhm.2019.03.012

6. Teubner, T., & Dann, D. (2017). Price determinants on airbnb: How reputation pays off in the sharing economy. Journal of Self-Governance and Management Economics, 5, 53. https://doi.org/10.22381/jsme5420173

7. Wang, D., & Nicolau, J. L. (2017). Price determinants of sharing economy based accommodation rental: A study of listings from 33 cities on Airbnb.com. International Journal of Hospitality Management, 62, 120–131. https://doi.org/10.1016/j.Ijhm.2016.12.007

8. Yang, Y., Mao, Z., & Tang, J. (2018). Understanding guest satisfaction with urban hotel location. Journal of Travel Research, 57, 243–259. https://doi.org/10.1177/0047287517691153

9. Zhang, H., Zhang, J., Lu, S., Cheng, S., & Zhang, J. (2011). Modeling hotel room price with geographically weighted regression. International Journal of Hospitality Management. https://doi.org/10.1016/j.ijhm.2011.03.010