Imports

In [2]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.dummy import DummyRegressor
from sklearn.compose import make_column_transformer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.preprocessing import (
    MinMaxScaler,
    OneHotEncoder,
    OrdinalEncoder,
    StandardScaler,
    FunctionTransformer,
    KBinsDiscretizer
)
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.model_selection import (
    GridSearchCV,
    RandomizedSearchCV,
    cross_val_score,
    cross_validate,
    train_test_split,
)
from sklearn.linear_model import Ridge
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.feature_selection import RFECV, SequentialFeatureSelector
import shap
```

The dataset includes various information on Airbnb listings such as their location (borough, neighbourhood, and coordinates), price, room types, reviews, price and availability. These factors influence the popularity of a listing and its proxy, `reviews_per_month`, our feature of interest. As such, studying these features will be helpful in understanding trends that influence listing popularity.

In [3]:
```
listings = pd.read_csv('AB_NYC_2019.csv')
listings = pd.DataFrame(listings)
listings.head()
```

Out[3]:

	id	name	host_id	host_name	neighbourhood_group	neighbourhood	latit
0	2539	Clean & quiet apt home by the park	2787	John	Brooklyn	Kensington	40.64
1	2595	Skylit Midtown Castle	2845	Jennifer	Manhattan	Midtown	40.75
2	3647	THE VILLAGE OF HARLEM....NEW YORK !	4632	Elisabeth	Manhattan	Harlem	40.80
3	3831	Cozy Entire Floor of Brownstone	4869	LisaRoxanne	Brooklyn	Clinton Hill	40.68
4	5022	Entire Apt: Spacious Studio/Loft by central park	7192	Laura	Manhattan	East Harlem	40.79

Data splitting

In [4]:
```
train_df, test_df = train_test_split(listings, random_state=123, test_size=0.3)
```

EDA

- While all observations seem to have some association with our response variable, many class imbalances are present in different features of the dataset. This could lead to bias in our model and poor generalizable performance especially for the minority class. There is also missing data in `reviews_per_month`, `last_review`, `host_name`, and `name`. There are also inconsistent scales between most numeric features. In addition, most variables are numerical. We will hence have to drop the rows containing NAs in the target since we cannot impute the target.
- From our quantile calculation of the response variable value we can see that most values are under 1, that is, close to 0. As such, it is inappropriate to use MAPE as it might give misleading figures when the actual values are small. In addition, as our response variable has large outliers (train_df['reviews_per_month']max = 58.5) and RMSE is sensitive to such outliers, it would be more appropriate to use R_2 error since it gives a more reliable indication of the model ability to account for variance.

In [5]:
```
train_df['price'].mean()
```

Out[5]:
```
np.float64(151.528399462397)
```

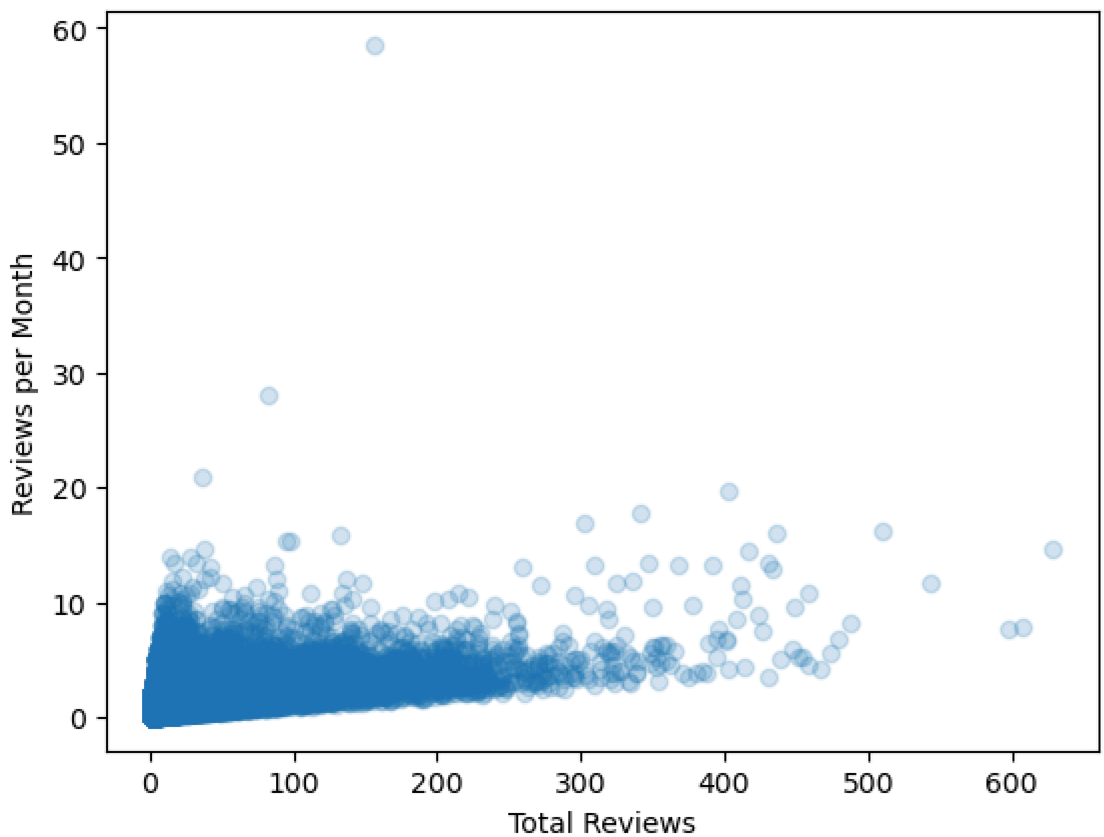
This indicates that the average price of a listing is $150.

In [6]:
```
train_df['number_of_reviews'].mean()
```

Out[6]:
```
np.float64(23.244813884181617)
```

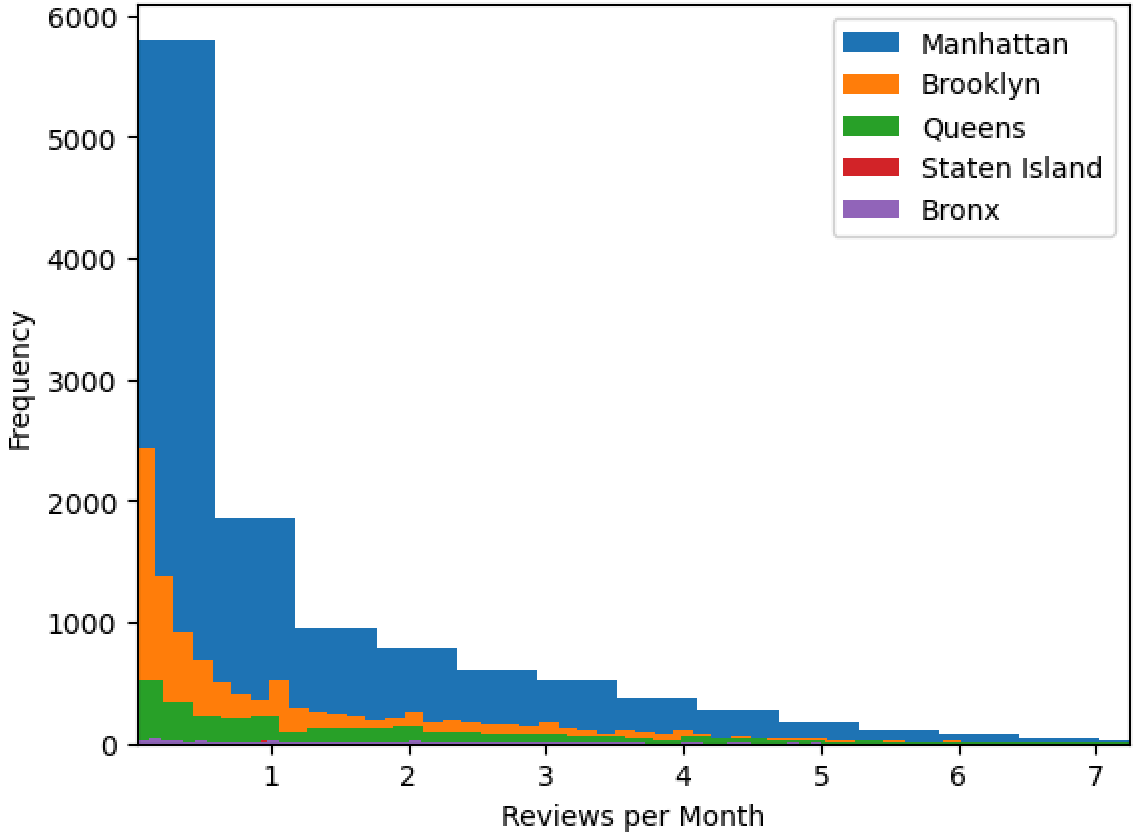
This indicates that on average a listing has 23.2 reviews.

In [7]:
```
plt.scatter(x=train_df['number_of_reviews'], y=train_df['reviews_per_month'], alpha
plt.xlabel('Total Reviews')
plt.ylabel('Reviews per Month')
plt.show()
```



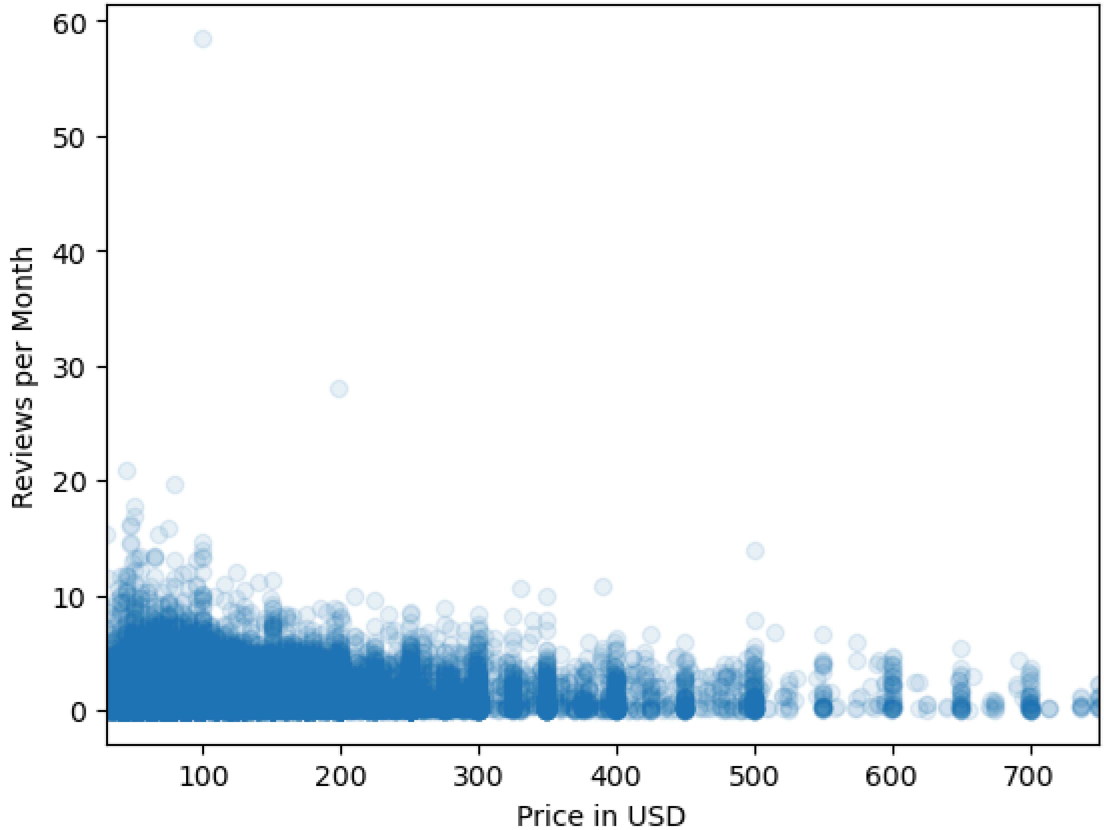
The scatterplot shows that the the number of monthly reviews appears to be positively associated with the total number of reviews a listing has.

In [8]:
```
groups = train_df['neighbourhood_group'].unique()
for i in groups:
    train_df[train_df['neighbourhood_group']==i]['reviews_per_month'].plot.hist(lab
    plt.xlabel('Reviews per Month')
    plt.legend()
    plt.xlim(train_df['reviews_per_month'].quantile(0.01),train_df['reviews_per_mon
plt.show()
```



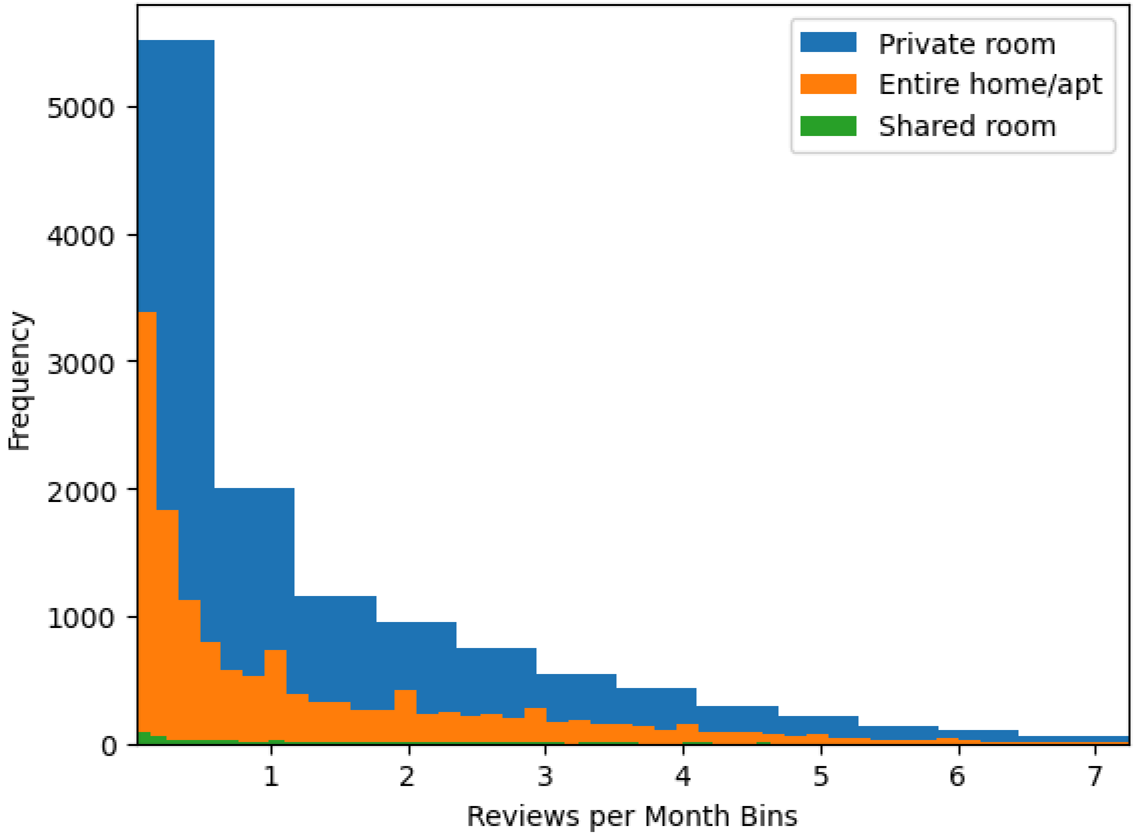
The histogram shows that while all categories generally follow a right-skewed dsitribution, different neighbourhood groups have slightly different distributions. In addition, we can see that there is a class imbalance present in the number of examples per group.

In [9]:
```
plt.scatter(x=train_df['price'], y=train_df['reviews_per_month'], alpha=0.1)
plt.xlabel('Price in USD')
plt.ylabel('Reviews per Month')
plt.xlim(train_df['price'].quantile(0.01), train_df['price'].quantile(0.99))
plt.show()
```



This scatterplot shows that the number of reviews per month appears to be negatively correlated with the price of a listing. This makes sense, as fewer guests would be willing to spend larger amounts of money to stay at an Airbnb.

In [10]:
```
groups = train_df['room_type'].unique()
groups[0],groups[1] = groups[1],groups[0]
for i in groups:
    train_df[train_df['room_type']==i]['reviews_per_month'].plot.hist(label=i, bins
    plt.xlabel('Reviews per Month Bins')
    plt.legend()
    plt.xlim(train_df['reviews_per_month'].quantile(0.01),train_df['reviews_per_mon
plt.show()
```



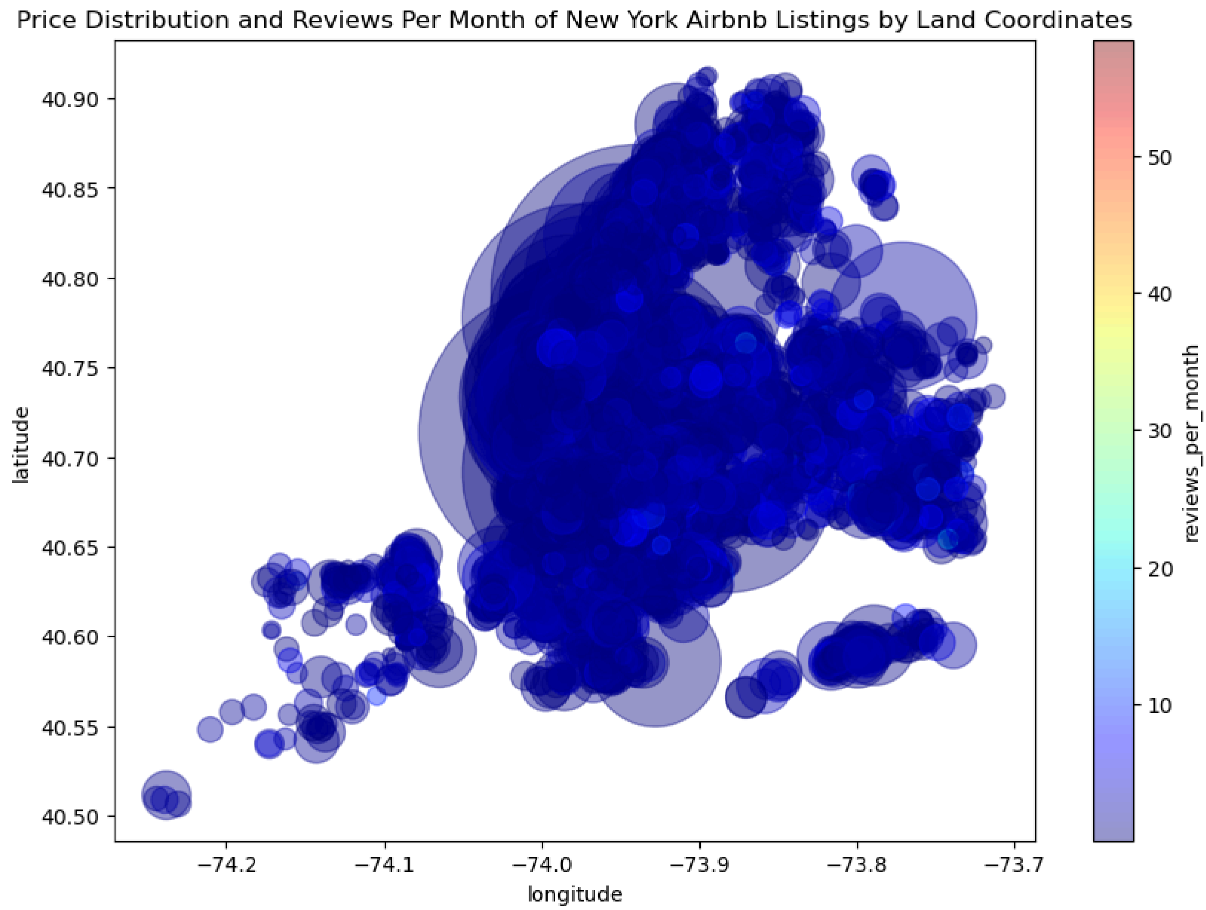
The histogram shows that all room types have right-skewed distributions but they are different in respect to shape and tail widths. In addition, we see a large class imbalance with respect to the types of rooms, with barely any examples having the shared room type.

In [11]:
```
train_df['reviews_per_month'].quantile(0.59)
```

Out[11]:
```
np.float64(1.0)
```

This indicates that approximately 60% of all examples observations have a response variable value less than or equal to 1. This is useful particularly if there exist outliers that inflate the mean.

In [12]:
```
plcm = train_df.plot(
    kind="scatter",
    x="longitude",
    y="latitude",
    alpha=0.4,
    s=train_df["price"]*2,
    figsize=(10, 7),
    c="reviews_per_month",
    cmap=plt.get_cmap("jet"),
    colorbar=True,
    sharex=False,
);

plcm.set_title("Price Distribution and Reviews Per Month of New York Airbnb Listing
plt.show()
```



The visualization shows that the highest-priced Airbnb listings are in central and northwestern New York, with prices decreasing as you move north and west. Lower-priced listings have a lighter color corresponding to more reviews per month.

Feature Engineering

We can perform discretization of latitude and longitude to create bins instead.

By dividing `number_of_reviews` by `calculated_host_listings_count` we can extract a new feature called `reviews_per_listing`. We will do this in the column transformer.

In [13]:
```
train_df = train_df.assign(
    reviews_per_listing=train_df["number_of_reviews"] / train_df["calculated_host_l
)
test_df = test_df.assign(
    reviews_per_listing=test_df["number_of_reviews"] / test_df["calculated_host_lis
)
train_df.head()
```

Out[13]:

	id	name	host_id	host_name	neighbourhood_group	neighbourho
36150	28736148	Cozy 1 Bedroom Apt in Hamilton Heights	43431867	Tommy	Manhattan	Washingt Heig
45223	34613254	Amazing One Bedroom at the Time Square Area/72B	48146336	Irina	Manhattan	Hell's Kitch
14316	11144496	New Spacious Master, Williamsburg	48819868	Nick	Brooklyn	Williamsbu
1691	766814	Adorable Midtown West Studio!	4022922	Caitlin	Manhattan	Hell's Kitch
20195	16162621	NEW! Exceptional 2BR/1BA Williamsburg Oasis	104781467	Russell	Brooklyn	Williamsbu

We can also add a feature by creating bins for `minimum_nights` to create meaningful intervals that can be used for prediction. We will do this later in the column transformer.

Preprocessing and Transformations

In [14]:

```
drop_cols = ['id', 'host_id', 'host_name', 'last_review','name']
numeric_cols = ['price', 'number_of_reviews', 'calculated_host_listings_count', 'av
discretized_cols = ['latitude','longitude']
discretized_cols2 = ['minimum_nights']
vector_cols=['name']
onehot_cols = ['neighbourhood_group', 'neighbourhood', 'room_type']

# adapted from: https://www.kaggle.com/code/mobasshir/machine-learning-issues-and-f
one_dim = FunctionTransformer(np.reshape, kw_args={'newshape':-1})

pipe = make_pipeline(SimpleImputer(strategy='constant'), one_dim, CountVectorizer(s
# Use this pipeline in your column transformer
transformer = make_column_transformer(
    (pipe, vector_cols),
    (KBinsDiscretizer(n_bins=20, encode="onehot"),discretized_cols),
    (KBinsDiscretizer(n_bins=3, encode="onehot"),discretized_cols2),
    (StandardScaler(), numeric_cols),
```

```
    (OneHotEncoder(sparse_output=False, handle_unknown='ignore'), onehot_cols),
    ("drop", drop_cols))
```

We are dropping `id`, `host_id`, and `host_name` since the id and name of a previous host are not useful in predicting the popularity of other listings. In addition, we are dropping `last_review` since we do not yet know how to process dates.

In [15]:
```
# Cannot impute the target
train_df = train_df.dropna(subset = ['reviews_per_month'])
test_df = test_df.dropna(subset = ['reviews_per_month'])
```

In [16]:
```
# Reducing training set size since later steps take too long for each model
train_df = train_df.sample(n=2000, replace=False)
```

Baseline Model

In [17]:
```
dummy = DummyRegressor(strategy="mean")
```

In [18]:
```
X_train = train_df.drop(columns=["reviews_per_month"])
y_train = train_df["reviews_per_month"]
X_test = test_df.drop(columns=["reviews_per_month"])
y_test = test_df["reviews_per_month"]
```

In [19]:
```
# Preprocessor not needed since dummy does not use input variables
dummy.fit(X_train, y_train)
dummy.score(X_train, y_train)
```

Out[19]: 0.0

Linear Models

In [20]:
```
first_linear_pipeline = make_pipeline(transformer, Ridge())
```

In [21]:
```
first_linear_res = pd.DataFrame(cross_validate(first_linear_pipeline, X_train, y_tr
print(first_linear_res.mean())
first_linear_res
```

```
fit_time      0.062782
score_time    0.026178
test_score    0.373732
dtype: float64
```

Out[21]:

	fit_time	score_time	test_score
0	0.064891	0.036087	0.253954
1	0.061486	0.015714	0.378873
2	0.062518	0.031720	0.380510
3	0.062501	0.016047	0.421955
4	0.062514	0.031323	0.433369

In [22]:
```
alpha_vals_large = np.logspace(-4, 4, 100)
param_grid = {"ridge__alpha":alpha_vals_large}
random_search_linear = RandomizedSearchCV(first_linear_pipeline,
                                          param_grid,
                                          n_jobs=-1,
                                          return_train_score=True)
random_search_linear.fit(X_train,y_train)
print(random_search_linear.best_score_)
print(random_search_linear.best_params_['ridge__alpha'])
```

```
0.4066020871014405
79.24828983539186
```

In [23]:
```
random_alpha = random_search_linear.best_params_['ridge__alpha'];
# If check below prevents negative values from being input as the alpha
if (random_alpha < 5):
    random_alpha = 5
param_grid_narrow = {"ridge__alpha":np.arange(random_alpha-5, random_alpha+5,2.5)}
grid_search_linear = GridSearchCV(first_linear_pipeline,
                                  param_grid_narrow,
                                  n_jobs=-1,
                                  return_train_score=True)
grid_search_linear.fit(X_train,y_train)
best_alpha = grid_search_linear.best_params_["ridge__alpha"]
print(grid_search_linear.best_score_,best_alpha)
```

```
0.4067691989884027 74.24828983539186
```

In [24]:
```
linear_tuned = make_pipeline(transformer, Ridge(alpha=best_alpha))
cv_results_linear = pd.DataFrame(cross_validate(linear_tuned, X_train, y_train, ret
ridge_sd = cv_results_linear['test_score'].std()
print(ridge_sd)
print(cv_results_linear.mean())
cv_results_linear
```

```
0.05298844013919663
fit_time       0.061995
score_time     0.016848
test_score     0.406769
train_score    0.443880
dtype: float64
```

Out[24]:

	fit_time	score_time	test_score	train_score
0	0.067458	0.015642	0.318263	0.468140
1	0.056339	0.006446	0.417398	0.443371
2	0.063401	0.015138	0.416812	0.440269
3	0.046874	0.031295	0.461699	0.432975
4	0.075905	0.015722	0.419674	0.434642

The linear model initially gives an mean cross-validation R2 score of 0.37. After tuning the alpha hyperparameter we get a model with alpha = ~74.25 and a mean cross-validation score of ~0.41. As such, there is an observable improvement in the mean score. Further, the small standard deviation value of 0.05 shows that the scores are generally consistent across folds.

Different Models

In [25]:

```
knn_pipe_first = make_pipeline(transformer, KNeighborsRegressor())
x1 = pd.DataFrame(cross_validate(knn_pipe_first, X_train, y_train, return_train_sco
print(x1)
print(x1.mean())
```

```
   fit_time  score_time  test_score  train_score
0  0.054062    0.284871    0.257699     0.546812
1  0.050102    0.032011    0.296209     0.539493
2  0.046884    0.033120    0.351382     0.547366
3  0.037175    0.047331    0.375484     0.548826
4  0.044782    0.046965    0.322375     0.572176
fit_time       0.046601
score_time     0.088860
test_score     0.320630
train_score    0.550935
dtype: float64
```

In [26]:

```
decision_tree_regular = make_pipeline(transformer, DecisionTreeRegressor())
x2 = pd.DataFrame(cross_validate(decision_tree_regular, X_train, y_train, return_tr
print(x2)
print(x2.mean())
```

```
   fit_time  score_time  test_score  train_score
0  0.097031    0.015626    0.147393          1.0
1  0.088405    0.015634    0.225886          1.0
2  0.093756    0.015615   -0.076939          1.0
3  0.093752    0.031253   -0.020082          1.0
4  0.078012    0.031238    0.137486          1.0
fit_time       0.090191
score_time     0.021873
test_score     0.082749
train_score    1.000000
dtype: float64
```

In [27]:
```
forest_pipeline = make_pipeline(transformer, RandomForestRegressor(n_jobs=-1))
x3 = pd.DataFrame(cross_validate(forest_pipeline, X_train, y_train, return_train_sc
print(x3)
print(x3.mean())
```

```
   fit_time  score_time  test_score  train_score
0  0.596687    0.074393    0.432886     0.938327
1  0.596918    0.082149    0.524065     0.930988
2  0.603303    0.075182    0.551523     0.933204
3  0.562782    0.059975    0.541804     0.932097
4  0.541677    0.065445    0.499822     0.927289
fit_time       0.580273
score_time     0.071429
test_score     0.510020
train_score    0.932381
dtype: float64
```

Model	Mean Training Score	Mean Cross-Val Score	Mean Score Time
KNeighborsRegressor	0.55	0.32	0.08
DecisionTreeRegressor	1.00	0.08	0.02
RandomForestRegressor	0.93	0.51	0.07

From the table above, we can see that DecisionTreeRegressor has the highest overfitting with a training R2 score of 1 and a validation R2 score of only 0.08. In contrast, RandomForestregressor has a lower mean training score of 0.93 and a much higher mean validation score of 0.51. This suggests that this model might generalize better but the large gap still suggest that overfitting is occurring. On the other hand, KNeighborsRegressor seems to show underfitting with both low mean training and cross-validation scores.

Only RandomForestRegressor beats the linear model, but it does so by a significant margin.

Feature Selection

In [28]:
```
#Setting cv to 2 in rfecv in the interest of time (takes 10+ mins even with 2 folds
rfe_knn_pipeline = make_pipeline(
    transformer,
    RFECV(RandomForestRegressor(n_jobs=-1), n_jobs=-1, cv=2),
    KNeighborsRegressor())
rfe_knn_res = pd.DataFrame(cross_validate(rfe_knn_pipeline, X_train, y_train, retur
print(rfe_knn_res)
print(rfe_knn_res.mean())
```

```
     fit_time  score_time  test_score  train_score
0  210.778316    0.053535    0.261354     0.564926
1  166.119870    0.050897    0.286956     0.546773
2  212.963956    0.040936    0.377878     0.601256
3  238.932022    0.045024    0.415175     0.569811
4  205.229174    0.031368    0.393888     0.597831
fit_time       206.804668
score_time       0.044352
test_score       0.347050
train_score      0.576119
dtype: float64
```

In [29]:

```
rfe_dtree_pipeline = make_pipeline(
    transformer,
    RFECV(DecisionTreeRegressor(), n_jobs=-1),
    DecisionTreeRegressor())
rfe_dtree_res = pd.DataFrame(cross_validate(rfe_dtree_pipeline, X_train, y_train, r
print(rfe_dtree_res)
print(rfe_dtree_res.mean())
```

```
    fit_time  score_time  test_score  train_score
0  12.202657    0.015632    0.101090     1.000000
1  15.406825    0.016828   -0.021628     0.523746
2  15.340677    0.015618    0.226981     0.524372
3  12.755322    0.015626    0.151651     0.505363
4  12.131724    0.015626    0.249928     0.482148
fit_time       13.567441
score_time      0.015866
test_score      0.141604
train_score     0.607126
dtype: float64
```

In [30]:

```
# Setting cv = 2 in the interest of time (takes 15+ minutes with cv=2)
rfe_forest_pipeline = make_pipeline(
    transformer,
    RFECV(RandomForestRegressor(n_jobs=-1),n_jobs=-1, cv=2),
    RandomForestRegressor(n_jobs=-1))
rfe_forest_res = pd.DataFrame(cross_validate(rfe_forest_pipeline, X_train, y_train,
print(rfe_forest_res)
print(rfe_forest_res.mean())
```

```
     fit_time  score_time  test_score  train_score
0  129.323299    0.075520    0.424532     0.937561
1  136.549193    0.070285    0.525885     0.931747
2  204.637739    0.076194    0.535672     0.930825
3  195.792348    0.070681    0.528762     0.932034
4  200.835507    0.073599    0.477448     0.934552
fit_time       173.427617
score_time       0.073256
test_score       0.498460
train_score      0.933344
dtype: float64
```

Model	RFECV Estimator	Mean Training Score	Mean Cross-Val Score	Mean Score Time
KNeighborsRegressor	RandomForestRegressor	0.58	0.35	0.07
DecisionTreeRegressor	DecisionTreeRegressor	0.61	0.14	0.02
RandomForestRegressor	RandomForestRegressor	0.93	0.50	0.07

As we can see the only model which shows significant improvement with RFECV is DecisionTreeRegressor. KNeighborsRegressor with RFECV only shows a marginal improvement of ~0.03 in the mean cross-validation score hence there is no need to use this model particuarly since each fold also now takes more than 3 minutes to fit. In the case of RandomForestRegressor, the model without RFECV actually has a higher mean score too.

Hyperparameter Optimization

In [31]:
```
n_neighbors_grid = 5 * np.arange(1,20,1)
param_grid_k = {"kneighborsregressor__n_neighbors":n_neighbors_grid}
random_search_knn = RandomizedSearchCV(knn_pipe_first,
                                       param_grid_k,
                                       n_jobs=-1,
                                       return_train_score=True)
random_search_knn.fit(X_train,y_train)
best_k_random = random_search_knn.best_params_['kneighborsregressor__n_neighbors']
print(best_k_random)
#Set k value to 11 if less than to prevent grid creation below from accepting vals
if (best_k_random < 11):
    best_k_random = 11
grid_search_knn = GridSearchCV(knn_pipe_first,
                               {"kneighborsregressor__n_neighbors":np.arange(best_k
                               n_jobs=-1,
                               return_train_score=True)
grid_search_knn.fit(X_train, y_train)
best_k_grid = grid_search_knn.best_params_['kneighborsregressor__n_neighbors']
print(grid_search_knn.best_score_, best_k_grid)
```

```
30
0.37346033780951154 31
```

In [32]:
```
mdepth_tune_grid = np.arange(1,10,1)
param_grid_mdepth = {"decisiontreeregressor__max_depth":mdepth_tune_grid}
random_search_mtree = GridSearchCV(rfe_dtree_pipeline,
                                   param_grid_mdepth,
                                   n_jobs=-1,
                                   return_train_score=True)
random_search_mtree.fit(X_train, y_train)
best_mdepth = random_search_mtree.best_params_['decisiontreeregressor__max_depth']
print(random_search_mtree.best_score_, best_mdepth)
```

```
0.2957115529723549 2
```

In [33]:
```
rforest_param_grid = {"randomforestregressor__n_estimators": 5 * np.arange(10, 30,
                      "randomforestregressor__max_depth": 5 * np.arange(1,10)}
grid_search_broad = GridSearchCV(forest_pipeline,
                                 rforest_param_grid,
                                 n_jobs=-1,
                                 return_train_score=True)
grid_search_broad.fit(X_train, y_train)
best_nestim_broad = grid_search_broad.best_params_['randomforestregressor__n_estima
best_md_broad = grid_search_broad.best_params_['randomforestregressor__max_depth']
if (best_nestim_broad < 5):
    best_nestim_broad = 5
if (best_md_broad < 5):
    best_md_broad = 5
grid_search_narrow = GridSearchCV(forest_pipeline,
                                  {"randomforestregressor__n_estimators": np.arange
                                   "randomforestregressor__max_depth": np.arange(be
                                  n_jobs=-1,
                                  return_train_score=True)
grid_search_narrow.fit(X_train, y_train)
best_nestim = grid_search_narrow.best_params_['randomforestregressor__n_estimators'
best_md = grid_search_narrow.best_params_['randomforestregressor__max_depth']
print(grid_search_narrow.best_score_, best_nestim, best_md)
```

```
0.5130673513326516 105 33
```

Model	Tuned Hyperparameter Value	R2 Accuracy Score
KNeighborsRegressor	n_neighbors: 31	~0.37
DecisionTreeRegressor with RFECV	max_depth: 2	~0.30
RandomForestRegressor	n_estimators: 105, max_depth: 33	~0.51

Of the models tuned, DecisionTreeRegressor achieves the highest relative increase in R2 score, achieving a mean cross-val score of approx. 0.30 with a tree of max depth 2. However, its performance still remains low compared to the other models. On the other hand, the score of RandomForestRegressor is still the highest but is about the same before and after tuning to 105 estimators and a max depth of 33. KNeighborsRegressor achieves a marginal gain of ~0.05 with the number of neighbors tuned to 31 in mean cross-val scores.

Interpretation and Feature Importances

In [34]:
```
rforest_tuned_pipeline = make_pipeline(transformer,
                                       RandomForestRegressor(n_estimators=best_nest
tuned_rf_results = pd.DataFrame(cross_validate(rforest_tuned_pipeline, X_train, y_t
print(tuned_rf_results.mean())
tuned_rf_results
```

```
fit_time       0.604574
score_time     0.082743
test_score     0.508949
train_score    0.933032
dtype: float64
```

Out[34]:

	fit_time	score_time	test_score	train_score
0	0.566389	0.070656	0.435701	0.932138
1	0.606201	0.084106	0.524278	0.933110
2	0.611740	0.089264	0.547460	0.933143
3	0.642110	0.079889	0.552193	0.934090
4	0.596430	0.089799	0.485111	0.932677

In [37]:

```
rforest_tuned_pipeline.fit(X_train, y_train)
vectorized_cnames = list(rforest_tuned_pipeline.named_steps["columntransformer"].na
kbins1 = list(rforest_tuned_pipeline.named_steps["columntransformer"].named_transfo

kbins2 = list(rforest_tuned_pipeline.named_steps["columntransformer"].named_transfo
ohe_names = list(rforest_tuned_pipeline.named_steps["columntransformer"].named_tran
feature_names_full = (
    vectorized_cnames + kbins1 + kbins2 + numeric_cols + ohe_names
)
feature_names_full
data = {
    "Importance": rforest_tuned_pipeline.named_steps["randomforestregressor"].featu
}
imp_df = pd.DataFrame(
    data=data,
    index=feature_names_full,
).sort_values(by="Importance", ascending=False)
imp_df.head(12)
```

Out[37]:

	Importance
number_of_reviews	0.383747
availability_365	0.118121
minimum_nights_2.0	0.050573
reviews_per_listing	0.040858
price	0.037626
neighbourhood_Springfield Gardens	0.034934
minimum_nights_0.0	0.032254
neighbourhood_East Elmhurst	0.031371
bedroom	0.012919
latitude_15.0	0.012410
apartment	0.011058
calculated_host_listings_count	0.009524

As we can see from the list of features organized by importance with respect to the best model out of the ones selected, the feature `number_of_reviews` dominates all others by a large margin. The second most important feature, `availability_365` has an importance score ~1/4th that of `number_of_reviews` .

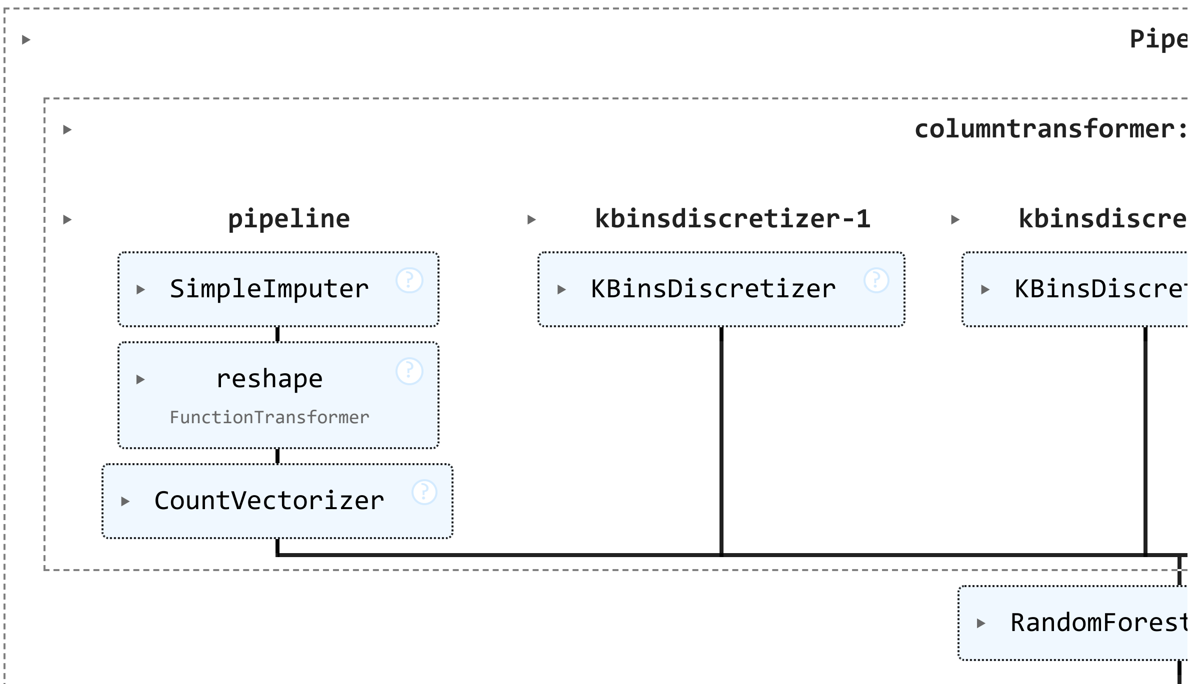
The importance of `number_of_reviews` also makes intuitive sense, since the more reviews a listing has, the more likely it is that more people are staying there and hence affecting the reviews per month.

Most features only have a minimal impact on the response variable, with the importance of the twelfth most important feature already less than 0.01.

Results on the Test Set

In [38]:
```
rforest_tuned_pipeline.fit(X_train, y_train)
```

Out[38]:



In [39]:
```
rforest_tuned_pipeline.score(X_test, y_test)
```

Out[39]:
```
0.46247327009049166
```

We have obtained a R2 score ~0.46, which is close to the mean cross-validation score of 0.51 using the tuned model (seen above). Although the test score is lower than the cross-validation score, this gap is minor which suggests that the model is performing consistently on both the validation and test datasets. This indicates that our model generalizes well to unseen data.

It also seems that there is not much optimization bias since the validation score of the grid search (0.51) is not much lower than the test score, which indicates that we do not have significant overfitting.

In [40]:
```
idx1 = 3468
idx2 = 8767
```

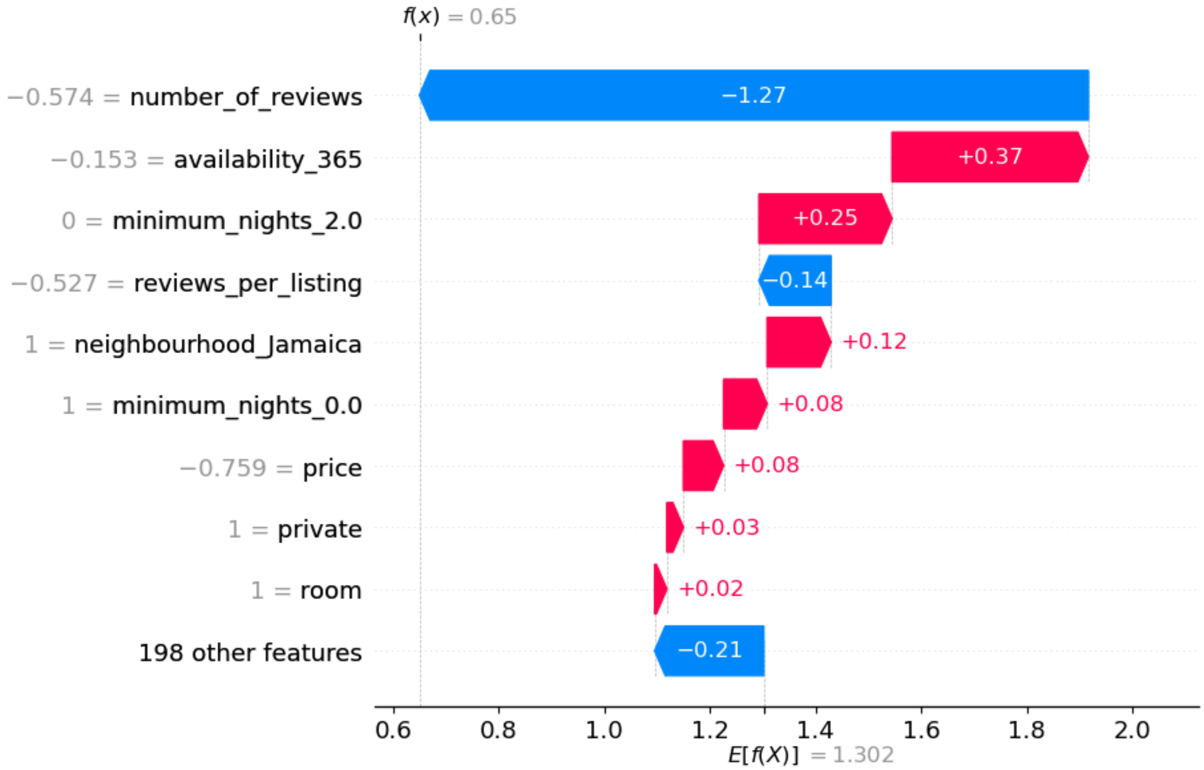
In [41]:
```
X_test_enc = pd.DataFrame(
    data = transformer.transform(X_test),
    columns=feature_names_full,
    index=X_test.index,
)
```

In [42]:
```
explainer = shap.TreeExplainer(rforest_tuned_pipeline.named_steps['randomforestregr
test_shap_values = explainer(X_test_enc)
test_shap_values
```

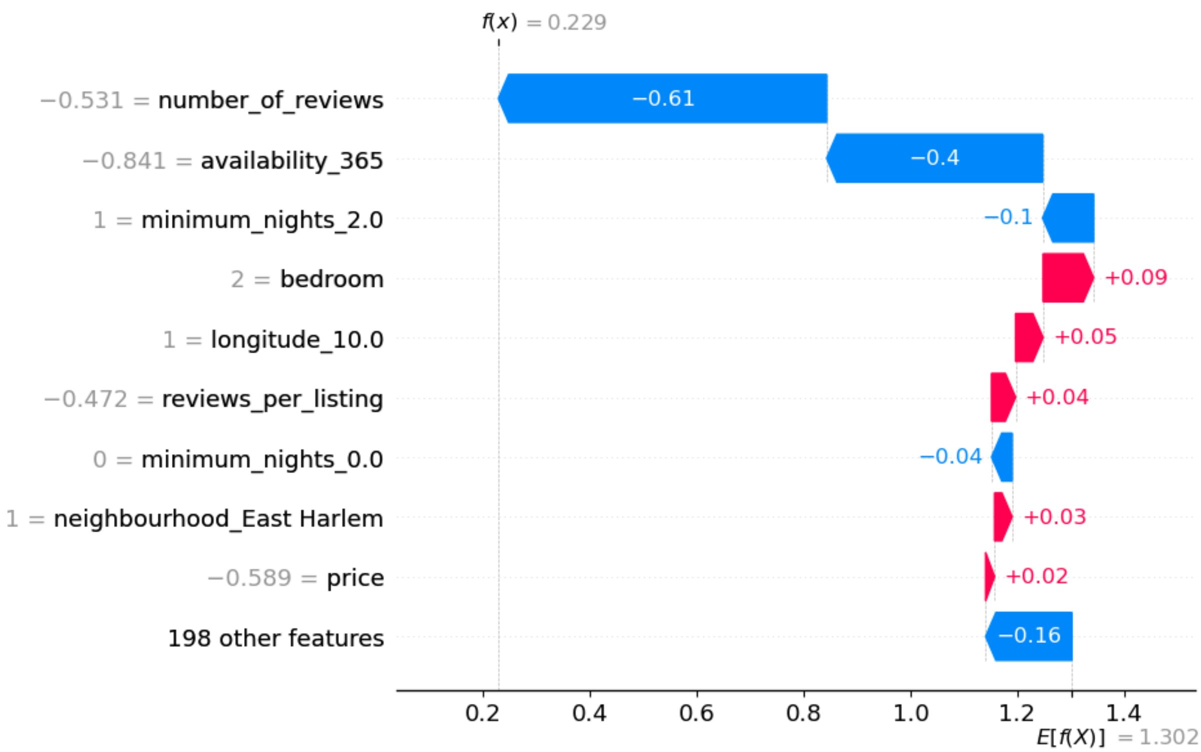
Out[42]:
```
.values =
array([[-2.03756330e-02,  9.70957776e-03, -1.37066801e-02, ...,
        -2.03346846e-02,  1.33787703e-04,  2.19710773e-04],
       [-6.61716672e-03, -1.36030906e-03,  3.81182539e-02, ...,
        -3.23622197e-03, -1.18176099e-03,  1.60923699e-05],
       [-5.70024282e-02,  8.44141776e-03, -2.14596113e-02, ...,
        -1.23574167e-02,  9.91109744e-03,  2.73368065e-04],
       ...,
       [-4.64012528e-03, -9.22152347e-04, -6.94715052e-03, ...,
        -5.14825113e-03, -9.55339115e-04, -2.95260758e-05],
       [-1.83124818e-02, -8.65186996e-04, -2.08455576e-02, ...,
         5.47297527e-03,  2.23059392e-03,  5.85629767e-05],
       [-3.33068639e-02, -8.86753116e-04, -1.74339625e-02, ...,
         7.77204182e-03,  5.36202592e-03, -2.83396979e-04]])

.base_values =
array([1.30211733, 1.30211733, 1.30211733, ..., 1.30211733, 1.30211733,
       1.30211733])

.data =
array([[0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 1., ..., 0., 1., 0.],
       [0., 0., 0., ..., 0., 1., 0.],
       ...,
       [0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 1., 0., 0.],
       [0., 0., 0., ..., 1., 0., 0.]])
```

In [43]:
```
shap.initjs()
```

In [44]:
```
shap.plots.waterfall(test_shap_values[idx1])
```



In [45]:
```
shap.plots.waterfall(test_shap_values[idx2])
```



Code for the two cells above adapted from: https://github.com/UBC-CS/cpsc330-2024W2/blob/main/lectures/202-203-Giulia-lectures/13_feat-importances.ipynb

Similar to what was seen above in feature importances, `number_of_reviews`, `availability_365`, and `minimum_nights_2.0` are the three most impactful features.

Interestingly, however, the shap values for `availability_365` and `minimum_nights_2.0` are positive for the first example and negative for the second. This means that in the first example these features are responsible for an increase in the predicted value whereas they are responsible for lowering the predicted value in the second example. This difference in the sign of the shap value might be due to the difference in the values of these two features in the two examples as well as the interaction of these values with other features.

In addition, after these three features, we can observe that the ordering of features by their absolute shap value is different. This might be because of the way the different values in these features between the two examples interact with other features.

Summary of Results

Task	Details
Models Used	Ridge, KNeighborsRegressor, DecisionTreeRegressor, RandomForestRegressor
Mean Cross-Validation Scores	Ridge: 0.37, KNN: 0.32, DecisionTreeRegressor: 0.08, RandomForestRegressor: 0.51
Mean Feature Selection Cross-Val Scores	KNN: 0.35, DecisionTreeRegressor: 0.14, RandomForestRegressor: 0.50
Hyperparameter Tuning	Ridge: [alpha = 74.25], KNN: [n_neighbours = 31], DecisionTreeRegressor (with RFECV): [max_depth = 2], RandomForestRegressor: [n_estimators: 105, max_depth: 33]
Tuned Mean Cross-Val Scores	Ridge: 0.41, KNN: 0.37, DecisionTreeRegressor (with RFECV): 0.30, RandomForestRegressor: 0.51
Feature Importance	number_of_reviews: 0.38, availability_365: 0.12, Rest are < 0.1
SHAP Insights	Top 3 Important Features are the same, but some increase the regression estimate in one example but decrease it in another. Features also have different value rankings between examples.

Out of all the models evaluated, the best one is the random forest regressor, which achieved a mean cross-validation R2 score of ~0.51 and a test R2 score of ~0.46. Most features (outside the top 10 by importance) had minimal impact on the predicted value, with the most significant one being the total number of reviews.

In the interest of time, we performed feature selection with only two folds for cross-validation. Given a lot more time or with a better device, performing feature selection with a larger number of folds might have resulted in a model with better accuracy. Similarly, tuning more hyper-parameters will take a long time, but would likely yield a better model. In particular, RandomForestRegressor, the model with the highest accuracy showed no imporvements after tuning and as such I would have liked to tune other hyperparameters too. Finally, the training set had to be reduced to 2000 examples as the notebook took 2+ hours to run even with this smaller subset. Training with the entire training set would likely produce a better model.