Airbnb Price Recommendations (Technical)

Imputing Null values

Bedrooms: Null values in the bedrooms column was handled by parsing the name column which often included the amount of bedrooms.

	name	bedrooms
42204	Home in Waikoloa Village · ★4.95 · 2 bedrooms ...	NaN
86009	Bed and breakfast in Waimea · ★4.63 · 1 bedroo...	NaN
107013	Home in Kailua-Kona · ★4.82 · 1 bedroom · 1 be...	NaN
1385780	Cottage in Holualoa · ★4.97 · Studio · 1 bed ·...	NaN
354771	Guest suite in Kailua-Kona · ★4.86 · Studio · ...	NaN

Bathrooms: Null values in the bathroom columns were handled by parsing the bathrooms_txt column

	bathrooms	bathrooms_text
0	NaN	1 bath
1	NaN	1 bath
2	NaN	1 bath
3	NaN	1 bath
4	NaN	1 bath
...	...	...

Below is the code used to impute null values

```
def impute_columns(row):
    if pd.isna(row['bedrooms']):
        # First, check for 'bedrooms' or 'Bedrooms'
        pattern_bedrooms = r'(\d+)\s*[Bb]edrooms?'
        match_bedrooms = re.search(pattern_bedrooms, row['name'])

        if match_bedrooms:
            bedrooms = int(match_bedrooms.group(1))
        else:
            # If 'bedrooms' not found, check for 'studio' or 'Studio'
            pattern_studio = r'\b[Ss]tudio\b'
            match_studio = re.search(pattern_studio, row['name'])

            if match_studio:
                bedrooms = 0  # or 1, depending on how you want to count studios
            else:
                bedrooms = np.nan
    else:
        bedrooms = row['bedrooms']

    # Handle bathrooms (unchanged from your original function)
    if pd.isna(row['bathrooms']):
        if pd.isna(row['bathrooms_text']):
            bathrooms = np.nan

        elif 'half-bath' in row['bathrooms_text'].lower():
            bathrooms = 0.5
        else:
            try:
                bathrooms = float(row['bathrooms_text'].split(" ")[0])
            except (ValueError, IndexError):
                bathrooms = np.nan
    else:
        bathrooms = row['bathrooms']

    if pd.isna(row['bathrooms_text']):
        shared = np.nan
    else:
        if 'shared' in row['bathrooms_text'].lower():
            shared = 1
        else:
            shared = 0
    return pd.Series([bedrooms, bathrooms,shared], index=['bedrooms', 'bathrooms','isShared_bathrooms'])
```

```
import swifter
# Load from parquet
RERUN=False
if RERUN:
    df[['bedrooms','bathrooms','isShared_bathrooms']]=df.swifter.apply(impute_columns, axis=1)
else:
    df = pd.read_parquet('after_impute_1.parquet')
df[['bedrooms','bathrooms','isShared_bathrooms']].isnull().sum()
print(df['bedrooms'].unique())
```

```
[ 1.  0.  3.  2.  4.  5.  8. nan  7.  6. 10.  9. 17.]
```

Remaining bedroom columns were imputed with linear regression.

```
def impute_with_regression(row,models,feature_subset):

    for model,subset in zip(models,feature_subset):
        try:
            X = pd.DataFrame([row[subset]])
            preds = model.predict(X)
            return round(2 * preds[0]) / 2
        except Exception as e:

            continue
    print('Warning: a value was unable to be imputed')
    return np.nan

def make_model(df,predictor_columns,cat_cols,predict_column,print_scores=False):
    models=[] # Store models after fitting them
    scores=[] # Store cross validated scores of each model
    feature_subsets=[]
    df_copy=df.copy()
    original_columns = df_copy.columns.tolist() # For finding the dummy column names
    df_copy=pd.get_dummies(df_copy,columns=cat_cols,drop_first=True) #Create dummy columns and add to dataframe
    dummy_columns = [col for col in df_copy.columns if col not in original_columns] #Get names of all dummy columns


    for i in range(len(predictor_columns)):
        lr=Lasso(alpha=7.0)
        subset=predictor_columns[i:len(predictor_columns)]+dummy_columns #Reduce the columns list (the columns list should be ordered in a
        non_null = df_copy.dropna(subset=subset) #only include rows where each column in the subset is not null
        X_train=non_null[subset]
        y_train=non_null[predict_column]
        lr.fit(X_train,y_train)
        scores.append(-1*cross_val_score(lr,X_train,y_train,cv=2,scoring='neg_mean_squared_error').mean())
        models.append(lr)
        feature_subsets.append(subset)
    # Zip models, scores, and subsets together
    models_scores_subsets = list(zip(models, scores, feature_subsets))

    # Sort by scores in descending order
    models_scores_subsets.sort(key=lambda x: x[1], reverse=True)

    # Unzip the sorted list
    sorted_models, sorted_scores, sorted_subsets = zip(*models_scores_subsets)
    if print_scores:
        for score,subset in zip(sorted_scores,sorted_subsets):
            print("{:<15}{:<15}".format(",".join(subset),score))

    return list(sorted_models), list(sorted_scores), list(sorted_subsets)


full_model_columns=['beds','bathrooms','accommodates','isShared_bathrooms'] #Columns that predict bedrooms
categorical_cols=['room_type'] #Categorical columns that predict bedrooms
linear_models, _, subsets=make_model(df_listings,full_model_columns,categorical_cols,'bedrooms',print_scores=True)


if not any(col.startswith('room_type_') for col in df.columns):
    df = pd.get_dummies(df, columns=['room_type'], drop_first=True)
else:
    print("Dummy columns for 'room_type' already exist.")
null_bedrooms=df[df['bedrooms'].isna()]
fill_with=null_bedrooms.swifter.apply(lambda x: impute_with_regression(x,linear_models,subsets),axis=1)
df['bedrooms']=df['bedrooms'].fillna(fill_with)
```

This resulted in very little remaining null values in the bedrooms and bathrooms columns. A few remained in the bathrooms column which meant that those rows had to be dropped.

Ratings: There were many missing ratings columns which needed to be imputed. These were imputed using the average for the listings particular number of bedrooms, number of bathrooms, and guests accommodated.

```
import pandas as pd

def impute_ratings(df, columns):
    df_copy=df.copy()
    df_copy['missing_rating']=[False]*len(df_copy)
    for column in columns:

        D = {column: 'mean'}
        non_null_column = df_copy[~df_copy[column].isna()]

        average_df = non_null_column.groupby(['bedrooms', 'bathrooms','accommodates']).agg(D).reset_index()
        average_df = average_df.rename(columns={column: f'{column}_avg'})

        mapping_df = average_df.set_index(['bedrooms', 'bathrooms','accommodates'])[f'{column}_avg']

        null_rows = df_copy[df_copy[column].isna()]

        for idx, row in null_rows.iterrows():
            key = (row['bedrooms'], row['bathrooms'],row['accommodates'])
            if key in mapping_df:
                df_copy.at[idx, column] = mapping_df[key]
                df_copy.at[idx,'missing_rating']=True
    return df_copy


rating_columns = [
    'review_scores_rating',
    'review_scores_accuracy',
    'review_scores_cleanliness',
    'review_scores_checkin',
    'review_scores_communication',
    'review_scores_location',
    'review_scores_value'
]

df2 = impute_ratings(df, rating_columns)
```

Doing the imputing this way meant that there had to be an average for the specific combination of bedrooms/bathrooms/accommodates for each null listing. This was not always the case, which meant that there was remaining null values after the imputing. I decided to impute the remaining null values by neighborhood.

```
import pandas as pd
import numpy as np

def impute_ratings_by_neighbourhood(df, rating_columns):
    # Create a DataFrame to hold averages for all rating columns
    D = {col: 'mean' for col in rating_columns}

    # Create a DataFrame with only non-null ratings
    non_null_ratings = df.dropna(subset=rating_columns)

    # Group by neighbourhood_cleansed to get averages
    avg_by_neighbourhood = non_null_ratings.groupby(['neighbourhood_cleansed']).agg(D).reset_index()
    avg_by_neighbourhood = avg_by_neighbourhood.rename(columns={col: f'{col}_avg_neighbourhood' for col in rating_columns})

    # Merge the averages back to the original DataFrame
    df = df.merge(avg_by_neighbourhood, on='neighbourhood_cleansed', how='left')

    # Fill missing values in the rating columns with the neighbourhood averages
    for col in rating_columns:
        mask = df[col].isna()  # Only update where the original value is null
        df.loc[mask, col] = df.loc[mask, f'{col}_avg_neighbourhood']

    # Drop the temporary average columns
    df.drop(columns=[f'{col}_avg_neighbourhood' for col in rating_columns], inplace=True)

    return df

# Example usage
rating_columns = [
    'review_scores_rating',
    'review_scores_accuracy',
    'review_scores_cleanliness',
    'review_scores_checkin',
    'review_scores_communication',
    'review_scores_location',
    'review_scores_value'
]

df3 = impute_ratings_by_neighbourhood(df2, rating_columns)
```

This got rid of all null values in important columns.

Feature Engineering

isHoliday

The time vs. price graph was the primary reason for deciding to create this feature. The graph shows spikes in prices during each major holiday period. Refer to slide 10 for the graph. The specific dates used to represent each holiday were chosen mostly by my understanding of what the holiday is. However, I did do some tests to try to narrow down specific periods that were significantly different in terms of price compared to neighboring time periods. I defined a period that represented the possible holiday period and then compared the difference in price of each possible period within the window and the adjacent periods (outside of the window).

```python
def analyze_holiday_period(df, start_date, end_date, min_days=1,p_threshold=0.05):
    start_date = pd.to_datetime(start_date)
    end_date = pd.to_datetime(end_date)

    # Calculate the length of the holiday period
    holiday_length = (end_date - start_date).days + 1

    # Define the outside window
    outside_window = df[
        ((df['date'] < start_date) & (df['date'] >= start_date - pd.Timedelta(days=holiday_length))) |
        ((df['date'] > end_date) & (df['date'] <= end_date + pd.Timedelta(days=holiday_length)))
    ]

    # Calculate mean and std for the outside window
    outside_mean = outside_window['log_price'].mean()
    outside_std = outside_window['log_price'].std()

    # Aggregate the data by date
    daily_avg = df.groupby('date')['log_price'].mean().reset_index()

    # Define the holiday period in the aggregated data
    holiday_period = daily_avg[(daily_avg['date'] >= start_date) & (daily_avg['date'] <= end_date)]

    best_period_tuple = None
    best_period_dict = None
    best_length = 0
    results = []

    # Check all possible sub-ranges within the holiday period
    for i in range(len(holiday_period)):
        for j in range(i + min_days - 1, len(holiday_period)):  # Start from i + min_days - 1 to include the min_days
            sub_range = holiday_period.iloc[i:j + 1]
            sub_prices = sub_range['log_price']
            sub_mean = sub_prices.mean()

            # Perform t-test on log transformed data
            t_stat, p_value = stats.ttest_1samp(sub_prices, outside_mean)
            is_significant = p_value < p_threshold and sub_mean > outside_mean

            sub_std = sub_prices.std()  # Calculate standard deviation for the sub-range
            log_prices_list = sub_prices.tolist()  # Convert the sub_prices Series to a list

            results.append({
                'start_date': sub_range['date'].min(),
                'end_date': sub_range['date'].max(),
                'mean': sub_mean,
                'std_dev': sub_std,  # Store standard deviation
                't_statistic': t_stat,
                'p_value': p_value,
                'is_significant': is_significant,
                'length': len(sub_range),
                'log_prices': log_prices_list  # Store the log prices for the range
            })
            # Update the best period if significant
            if is_significant and len(sub_range) > best_length:
                best_length = len(sub_range)
                best_period_tuple = (sub_range['date'].min(), sub_range['date'].max())
                best_period_dict = {
                    'start_date': sub_range['date'].min(),
                    'end_date': sub_range['date'].max(),
                    'mean': sub_mean,
                    'std_dev': sub_std,
                    't_statistic': t_stat,
                    'p_value': p_value,
                    'is_significant': is_significant,
                    'length': len(sub_range),
                    'log_prices': log_prices_list  # Store the log prices for the best period
                }

    return best_period_tuple, best_period_dict, results
```

This mostly showed insignificant differences, but did show that November 21 to November 26 was the most significantly different period from the adjacent windows (using days ahead and days before the window for comparison).

Significance of Thanksgiving

```
# Example usage
start_date = '2023-11-20'
end_date = '2023-11-30'
min_days = 3  # Example: exclude periods of 3 days or less
thanksgiving_dates, thanksgiving_dict,results_thanksgiving = analyze_holiday_period(df_no_outliers, start_date, end_date, min_days)

# Output results
print(f"Best Period: {thanksgiving_dates}")
print("\nDetailed Results:")
```

```
Best Period: (Timestamp('2023-11-21 00:00:00'), Timestamp('2023-11-26 00:00:00'))
```

I created the isHoliday feature by selecting the date periods that represented a holiday and assigning all the rows that had a date value within any of the periods to a value of True.

Create isHoliday feature

```
import datetime

def get_holiday_ranges(start_year):
    holiday_ranges = {
        "New Year's and Christmas": (datetime.date(start_year, 12, 19), datetime.date(start_year + 1, 1, 2)),
        "Memorial Day": (datetime.date(start_year + 1, 5, 25), datetime.date(start_year + 1, 5, 27)),  # Last Monday of May and surrounding da
        "Independence Day": (datetime.date(start_year + 1, 7, 2), datetime.date(start_year + 1, 7, 6)),
        "Columbus Day": (datetime.date(start_year + 1, 10, 8), datetime.date(start_year + 1, 10, 14)),  # Second Monday of October and surroun
        "Halloween": (datetime.date(start_year + 1, 10, 28), datetime.date(start_year + 1, 11, 1)),
        "Thanksgiving": (datetime.date(start_year, 11, 21), datetime.date(start_year, 11, 26)),  # As per your request
    }
    return holiday_ranges


# Example usage:
holiday_ranges_2023 = get_holiday_ranges(2023)
holiday_date_list = []
for start, end in holiday_ranges_2023.values():
    holiday_date_list.extend(pd.date_range(start, end).date)
holiday_date_list=set(holiday_date_list)
df['isHoliday']=df['date'].isin(holiday_date_list)
# Print some results to verify
print(df[['date', 'isHoliday']].head(10))
print("\nTotal holiday dates:", df['isHoliday'].sum())
```

```
        date  isHoliday
```

Aggregating the Data

As mentioned in the slides, Ii wanted to aggregate the data by a certain time period and use that time period as a feature. I decided to aggregate by week and make a feature that represented the passing of a week (7 days of time). See slide 11 for the reasons why a 7 day period was chosen as the aggregation period.

Creation of 'time' feature and aggregating the data by week.

Enough hosts are changing there price on a weekly basis, so I decided to aggregate by week

```
import pandas as pd
import numpy as np



# Step 2: Combine week and holiday information
df['week_holiday'] = df['week'].astype(str) + '_' + df['isHoliday'].astype(str)

# Step 3: Convert to continuous integer
df['time'] = pd.factorize(df['week_holiday'])[0]

# Sort the DataFrame by date to ensure the 'time' column is in chronological order
df = df.sort_values('date')

# Display the result
print(df[['date', 'isHoliday', 'week', 'week_holiday', 'time']].head(20))
```

```
              date  isHoliday  week week_holiday  time
0       2023-09-10      False    36     36_False     0
420115  2023-09-10      False    36     36_False     0
1458166 2023-09-10      False    36     36_False     0
1081121 2023-09-10      False    36     36_False     0
160600  2023-09-10      False    36     36_False     0
1080756 2023-09-10      False    36     36_False     0
1868255 2023-09-10      False    36     36_False     0
420480  2023-09-10      False    36     36_False     0
1458531 2023-09-10      False    36     36_False     0
1081486 2023-09-10      False    36     36_False     0
1080391 2023-09-10      False    36     36_False     0
160235  2023-09-10      False    36     36_False     0
1080026 2023-09-10      False    36     36_False     0
1079661 2023-09-10      False    36     36_False     0
1868620 2023-09-10      False    36     36_False     0
1758755 2023-09-10      False    36     36_False     0
421210  2023-09-10      False    36     36_False     0
1458896 2023-09-10      False    36     36_False     0
2059150 2023-09-10      False    36     36_False     0
420845  2023-09-10      False    36     36_False     0
```

I needed to create a continuous time variable that would represent the passing of time in the forward direction. I was aggregating by a 7 day period, but each holiday period didn't necessarily intersect with the beginning or end of such a period. To create the time variable and preserve the isHoliday feature I concatenated the value of the already created isHoliday feature with the week number, ordered the dataframe by date and then factorized the result. This gave me a continuous variable 'time' which took on values 1-58. 1 represented the 7 day period where the data began, and 58 represented the last 7 day period in the data. Somewhere in between were the holiday periods, which didn't necessarily represent 7 day periods.

isNew

The isNew feature was created to specify whether a listing was classified as new or not. I created it by parsing the ‘name’ column which contained the information.

Create isNew Feature

```
74]: import re
df_no_rating=df[df['missing_rating']]
def get_is_new(text):
    # Regular expression pattern to find the value immediately after *, allowing for optional space
    pattern = r"*\s*(\S+)"
    match = re.search(pattern, text)
    if match:
        value_after_star = match.group(1)
        if value_after_star.capitalize() == 'New':
            return True
    return False

df['isNew']=df['name'].apply(get_is_new)
```

Here is a look at how the data was aggregated. From this code you can see the main variables that are being considered as features.

```
agg_dict = {
    'price': 'mean',
    'neighbourhood_cleansed': 'first',
    'latitude': 'first',
    'longitude': 'first',
    'property_type': 'first',
    'accommodates': 'first',
    'bathrooms': 'first',
    'bedrooms': 'first',
    'review_scores_location': 'first',
    'review_scores_rating': 'first',
    'review_scores_cleanliness': 'first',
    'amenities': 'first',
    'room_type_Private room': 'first',
    'room_type_Hotel room': 'first',
    'room_type_Shared room': 'first',
    'month': 'first',
    'week': 'first',
    'isShared_bathrooms': 'first',
    'isHoliday': 'first',
    'isNew': 'first',
    'listing_url':'first',
    'missing_rating':'first',
    'number_of_reviews':'first'
}

df_aggregated = df.groupby(['listing_id', 'time']).agg(agg_dict).reset_index()


df_aggregated = df_aggregated[main_features]


print(df_aggregated.head())

print(f"Shape of aggregated DataFrame: {df_aggregated.shape}")
```

hasPool and hasHotTub

The amenities column contained a list of amenities that were associated with the listing. I parsed all of the unique words in the joined list of amenities. I did word counts to evaluate which amenities were being mentioned frequently. I found that there were many different ways that a host could mention that the listing had a pool or hot tub. Many of them were not associated with higher priced listings. For example, there could be a shared pool in an apartment complex or a small hotel, which wouldn't point to higher priced units. Ultimately, I decided to parse the amenities column to search for the phrase 'Private Pool' or 'Private Hot Tub'. Having these listed in the amenities column pointed to higher priced listings. See slide 18

for visual proof of this. Below is how the amenities column was parsed.

```
import swifter
def hasPrivatePoolorTub(amenities):
    pattern1 = r"private pool"
    pattern2 = r"private hot tub"
    pool=False
    hot_tub=False
    for text in amenities:
        if isinstance(text, str):
            if re.search(pattern1, text, re.IGNORECASE):
                pool=True
            elif re.search(pattern2, text, re.IGNORECASE):
                hot_tub=True
            if pool and hot_tub:
                break
    return pd.Series({'hasPool': pool, 'hasHotTub': hot_tub})
if APPLY:
    df_aggregated[['hasPool','hasHotTub']]=df_aggregated['amenities'].apply(hasPrivatePoolorTub)
```

Property Groups

As mentioned on slide 16 , there were many property types and I decided to place them into smaller groups to reduce the size of my feature space and prevent overfitting, for when it came time to start fitting models. I decided upon these groups by looking at the means of each property type and seeing which ones fell into similar price categories. I decided upon these groups.

```
Group 1: ['Room in boutique hotel']
Group 2: ['Entire villa']
Group 3: ['Entire home/apt']
Group 4: ['Entire home', 'Private room in serviced apartment']
Group 5: ['Entire serviced apartment', 'Room in hotel', 'Entire vacation home', 'Camper/RV', 'Shared room in bed and breakfast', 'Casa partic
ular', 'Private room in casa particular']
Group 6: ['Private room in resort', 'Entire townhouse', 'Private room in loft']
Group 7: ['Entire condo', 'Entire bungalow', 'Campsite', 'Private room in hut', 'Private room in bungalow', 'Private room in vacation home',
'Entire place', 'Private room in villa', 'Room in aparthotel', 'Shared room in casa particular', 'Shared room in home', 'Private room in eart
hen home', 'Private room in townhouse', 'Treehouse', 'Private room in tiny home']
Group 8: ['Entire rental unit', 'Farm stay', 'Private room', 'Private room in tent', 'Yurt', 'Private room in rental unit', 'Entire loft']
Group 9: ['Private room in home', 'Private room in guest suite', 'Private room in farm stay', 'Tiny home', 'Private room in guesthouse', 'Dom
e', 'Entire cabin', 'Private room in condo', 'Tent']
Group 10: ['Entire cottage', 'Private room in bed and breakfast', 'Entire guesthouse']
Group 11: ['Entire guest suite']
```

I then performed ANOVA and pairwise T tests on each combination to determine if they were significantly different from each other. All t tests were done using the log transformed price, which makes the data much closer to normal. The results showed significant differences between all groups. You can see the visualization of them on slide 17.

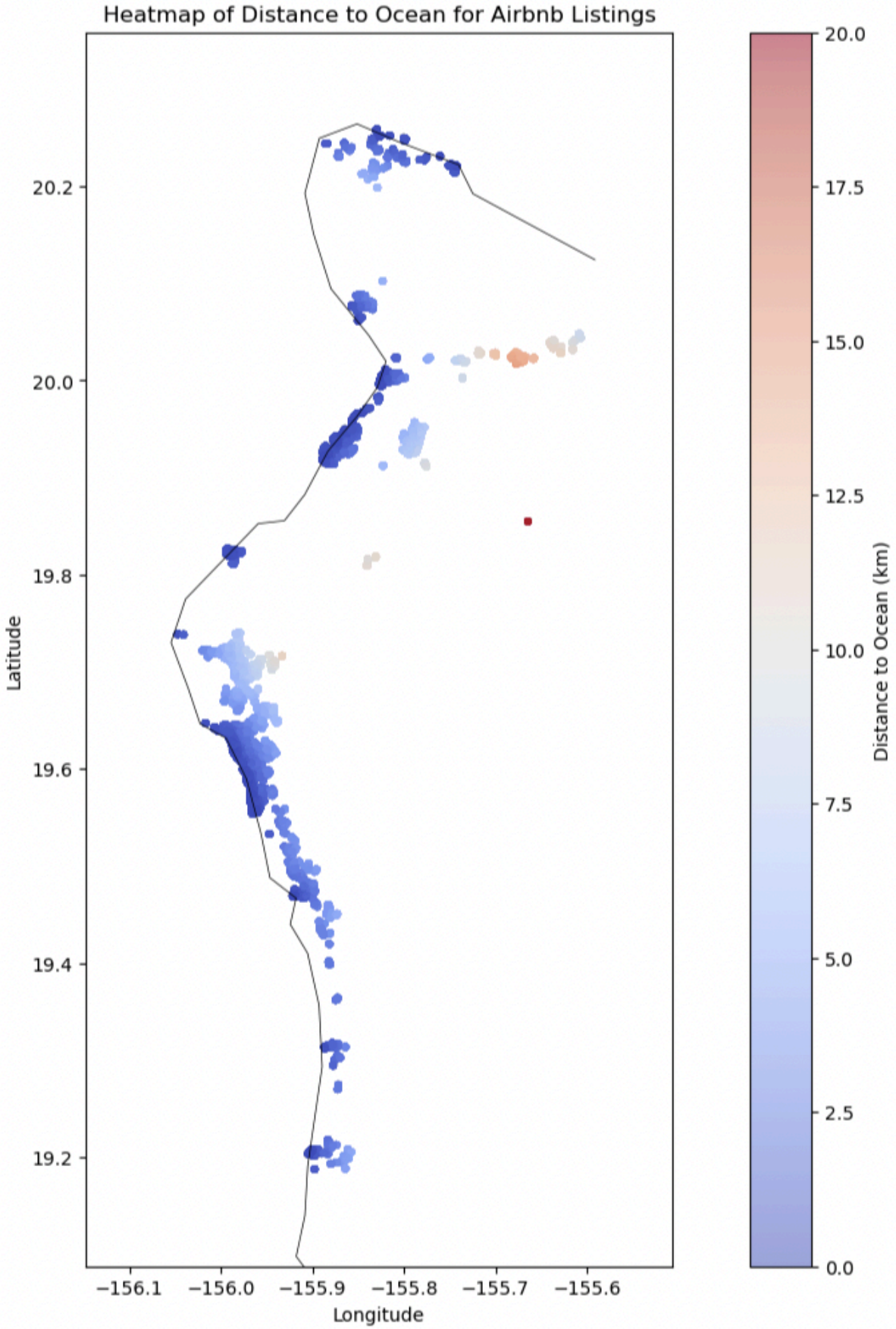
Geographic related features

distanceToOcean

The distanceToOcean feature was created by first creating a geojson file that contained the linestring of the Hawaiian coastline. This was done by tracing the coastline on the application provided by https://geojson.io/#map=2/0/20. The geojson file was then downloaded and I used it, along with shapely functionality, to calculate the distance (km) from each listings lat/lon

coordinates to the nearest point on the linestring. To see how the distanceToOcean variable is correlated with price refer to slide 15.

```
import pandas as pd
import geopandas as gpd
import matplotlib.pyplot as plt
from shapely.ops import nearest_points


gdf = gpd.GeoDataFrame(df, geometry=gpd.points_from_xy(df.longitude, df.latitude), crs="EPSG:4326")


gdf = gdf.to_crs("EPSG:32604")
west_hawaii_coast = west_hawaii_coast.to_crs("EPSG:32604")


def calculate_distance_to_ocean(point, coastline):
    nearest_point = nearest_points(point, coastline.unary_union)[1]
    return point.distance(nearest_point)

gdf['distanceToOcean'] = gdf.geometry.apply(calculate_distance_to_ocean, coastline=west_hawaii_coast)

# Convert distance to kilometers
gdf['distanceToOcean'] = gdf['distanceToOcean'] / 1000

# Convert back to original CRS for plotting
gdf = gdf.to_crs("EPSG:4326")
west_hawaii_coast = west_hawaii_coast.to_crs("EPSG:4326")


plt.figure(figsize=(15, 12))
scatter = plt.scatter(gdf.geometry.x, gdf.geometry.y, c=gdf['distanceToOcean'],
                      cmap='coolwarm', s=10, alpha=0.5, vmin=0, vmax=20)
plt.colorbar(scatter, label='Distance to Ocean (km)')


west_hawaii_coast.plot(ax=plt.gca(), color='black', linewidth=0.5)

plt.title('Heatmap of Distance to Ocean for Airbnb Listings')
plt.xlabel('Longitude')
plt.ylabel('Latitude')

# Adjust the plot limits to focus on the area of interest
plt.xlim(gdf.geometry.x.min() - 0.1, gdf.geometry.x.max() + 0.1)
plt.ylim(gdf.geometry.y.min() - 0.1, gdf.geometry.y.max() + 0.1)

plt.show()
```


Heatmap of Distance to Ocean for Airbnb Listings
Latitude
20.2
20.0
19.8
19.6
19.4
19.2
−156.1
−156.0
−155.9
−155.8
−155.7
−155.6
Longitude
Distance to Ocean (km)
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0

Resort Features

I created a feature that specifies which resort a listing is in or if they are not in a resort. To do this I used a similar process to creating distanceToOcean. I used **https://geojson.io/#map=2/0/20**. To map out polygons and then used shapely functionality to determine if the lat/lon coordinates were contained in that polygon.

Polygon created at website

Code to determine if listing is in resort

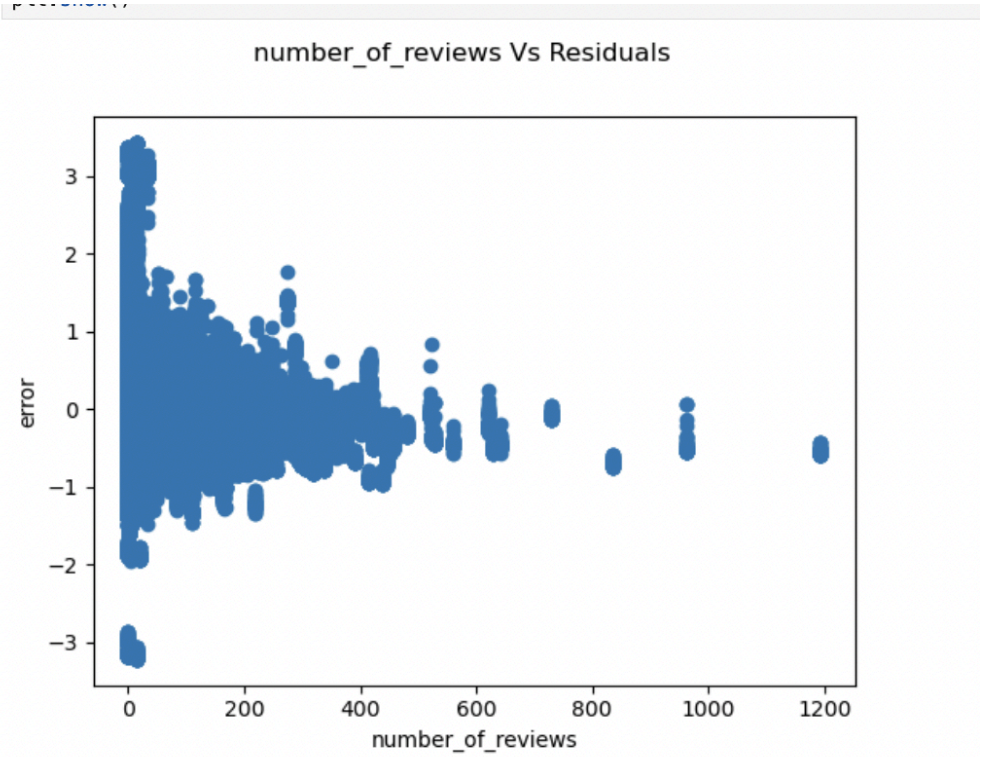
```
import pandas as pd
import geopandas as gpd
import matplotlib.pyplot as plt

# Load your resort polygons
mauna_kea = gpd.read_file('MaunaKea.geojson')
waikoloa_mauna_lani = gpd.read_file('Waikoloa:MaunaLani.geojson')
four_seasons = gpd.read_file('Fourseasons.geojson')

# Add a unique identifier for each resort area
mauna_kea['resort'] = 'Mauna Kea'
waikoloa_mauna_lani['resort'] = 'Mauna Lani/Waikaloa'
four_seasons['resort'] = 'Four Seasons'

# Combine all resort polygons into a single GeoDataFrame
all_resorts = gpd.GeoDataFrame(pd.concat([mauna_kea, waikoloa_mauna_lani, four_seasons], ignore_index=True))

# Ensure all GeoDataFrames have the same CRS
crs = "EPSG:4326"
all_resorts = all_resorts.to_crs(crs)
gdf = gdf.to_crs(crs)  # Ensure gdf is in the same CRS

print("gdf info:")
print(gdf.info())
print("\nall_resorts info:")
print(all_resorts.info())

print("gdf CRS:", gdf.crs)
print("all_resorts CRS:", all_resorts.crs)

try:
    joined = gpd.sjoin(gdf, all_resorts, how="left", predicate="within")
    print("Spatial join successful")
    print(joined.columns)

    # Create 'inResort' column
    gdf['inResort'] = joined['index_right'].notnull()

    # Create 'resortArea' column
    gdf['resortArea'] = joined['resort'].fillna('Not in Resort')

except Exception as e:
    print("Error in spatial join:", str(e))

    # Alternative method if spatial join fails
    def is_within_resort(point, resorts):
        for idx, row in resorts.iterrows():
            if point.within(row.geometry):
                return row['resort']
        return 'Not in Resort'

    gdf['resortArea'] = gdf.geometry.apply(lambda point: is_within_resort(point, all_resorts))
    gdf['inResort'] = gdf['resortArea'] != 'Not in Resort'

print("\nResults:")
print(gdf['inResort'].value_counts())
print(gdf['resortArea'].value_counts())
```

Refer to slide 14 to see how these resort features are correlated with price and to see where all of them are on the island.

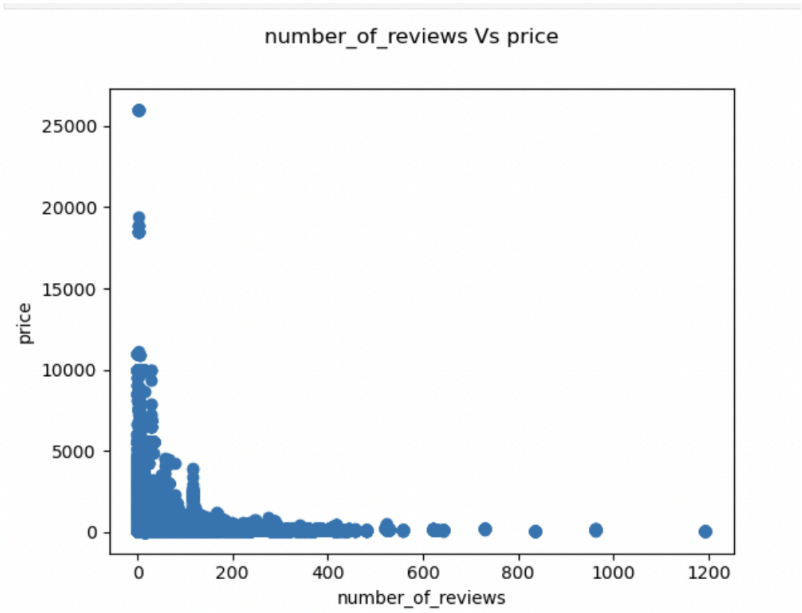
Anomaly Detection

I found many instances of prices that were mislabeled and did not actually represent the price of the listing. For details on this you can see slide 20. This was discovered by fitting models and looking at predicted prices versus actual prices, and then going to the website of the listings with high error. I then began to analyze price outliers and see if I could isolate and remove these data points from the data. I tried to trace these listings back to some common feature in one of the columns. I looked for relationships with error and some of the other unused columns such as number of reviews, minimum nights, availability, etc. I did find an interesting relationship with 'number_of_reviews' and the residuals. The lower the number of reviews the more the amount of error.

(note this model was fit with log_price)



This could be for several reasons. Listings may have a low number of reviews because their price is too high and there is less demand. There could also be listings that, for tax reasons, do not actually want their listing rented out and set it at a high price so no one rents it. Here is a graph that shows that these are associated with high priced listings.



Either way I wanted to eliminate these listings as much as possible. I wanted to make sure that I was giving my model examples that accurately represented the patterns and relationships that I was trying to use to predict price. Ultimately, I fit a linear model using the log_price and removed listings where the log_price was 50% higher than the predicted price. I also used some conditional logic to remove high priced listings with a small number of reviews. Finally, I removed listings with a price over $2,500 Below is the code that was used to remove outliers,

along with details of the regression model.

[19]:
```
import pandas as pd
import statsmodels.api as sm
X = df[features]
y = df['log_price']
df.dropna(subset=features + ['log_price'], inplace=True)
X = df[features]
y = df['log_price']
X = sm.add_constant(X)
model = sm.OLS(y, X).fit()
df['predicted_price'] = model.predict(X)
df['residuals'] = df['log_price'] - df['predicted_price']
# Identify outliers based on residuals
threshold = 1.5 * df['residuals'].std()  # You can adjust this threshold
outliers = df[(df['residuals'] > threshold) | (df['residuals'] < -threshold)]
# Display identified outliers
print("Identified Outliers:")
print(outliers[['log_price', 'predicted_price', 'residuals']])
print(model.summary())
```

```
Identified Outliers:
        log_price  predicted_price  residuals
870      6.210600         7.277264  -1.066664
871      6.210600         7.277264  -1.066664
872      6.210600         7.277264  -1.066664
873      6.210600         7.277264  -1.066664
874      6.210600         7.329605  -1.119005
...           ...              ...        ...
346402   6.586172         5.464087   1.122085
346403   6.586172         5.459784   1.126388
346404   6.586172         5.459784   1.126388
346405   6.586172         5.459784   1.126388
346406   6.586172         5.459784   1.126388

[33823 rows x 3 columns]
                            OLS Regression Results
==============================================================================
Dep. Variable:              log_price   R-squared:                       0.453
Model:                            OLS   Adj. R-squared:                  0.453
Method:                 Least Squares   F-statistic:                 1.595e+04
Date:                Fri, 23 Aug 2024   Prob (F-statistic):               0.00
Time:                        01:46:21   Log-Likelihood:            -2.8992e+05
No. Observations:              346581   AIC:                         5.799e+05
Df Residuals:                  346562   BIC:                         5.801e+05
Df Model:                          18
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const      -2.411e+09   5.48e+09     -0.440      0.660   -1.31e+10    8.32e+09
bedrooms       0.0768      0.002     39.443      0.000       0.073       0.081
bathrooms      0.2419      0.002    135.865      0.000       0.238       0.245
accommodates   0.0777      0.001    108.632      0.000       0.076       0.079
distanceToOcean   -0.0488      0.000   -116.968      0.000      -0.050      -0.048
hasPool        0.2330      0.008     29.945      0.000       0.218       0.248
hasHotTub      0.1660      0.006     26.913      0.000       0.154       0.178
month_1    2.411e+09   5.48e+09      0.440      0.660   -8.32e+09    1.31e+10
month_2    2.411e+09   5.48e+09      0.440      0.660   -8.32e+09    1.31e+10
month_3    2.411e+09   5.48e+09      0.440      0.660   -8.32e+09    1.31e+10
month_4    2.411e+09   5.48e+09      0.440      0.660   -8.32e+09    1.31e+10
month_5    2.411e+09   5.48e+09      0.440      0.660   -8.32e+09    1.31e+10
month_6    2.411e+09   5.48e+09      0.440      0.660   -8.32e+09    1.31e+10
month_7    2.411e+09   5.48e+09      0.440      0.660   -8.32e+09    1.31e+10
month_8    2.411e+09   5.48e+09      0.440      0.660   -8.32e+09    1.31e+10
month_9    2.411e+09   5.48e+09      0.440      0.660   -8.32e+09    1.31e+10
month_10   2.411e+09   5.48e+09      0.440      0.660   -8.32e+09    1.31e+10
month_11   2.411e+09   5.48e+09      0.440      0.660   -8.32e+09    1.31e+10
month_12   2.411e+09   5.48e+09      0.440      0.660   -8.32e+09    1.31e+10
==============================================================================
Omnibus:                    87687.492   Durbin-Watson:                   0.057
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           428326.422
Skew:                           1.147   Prob(JB):                         0.00
Kurtosis:                       7.939   Cond. No.                     1.36e+14
==============================================================================
```

```
price_outliers=get_outliers(df_cleaned,'log_price',1.5)
price_outliers.info()
price_outliers.sort_values(by='price', ascending=False)[['price','bedrooms','listing_url','accommodates','number_of_reviews']].to_csv('outlier
drops=[]
for ind, row in price_outliers.iterrows():
    if row['number_of_reviews']<4:
        drops.append(ind)
        print(ind)
    if row['price']<60:
        drops.append(ind)
    if row['price']>2500:
        drops.append(ind)
df_cleaned=df_cleaned[~df_cleaned.index.isin(drops)]
```

Model Fitting

I fit three types of models which included: RandomForrests, XgBoost, and extraTrees. For details on the specific parameters used see slide 23. To create training and testing sets I split the data by listing. This was so the test set would have listings that hadn't been seen during the fitting process.

```
import random
def train_test_split_airbnb(df,train_size):
    listings=df['listing_id'].unique()
    indices = list(range(len(listings)))
    train_indices=random.sample(indices, int(train_size*len(indices)))
    test_indices=[ ind for ind in indices if ind not in train_indices]
    train_listings=[listings[ind] for ind in train_indices]
    test_listings=[listings[ind] for ind in test_indices]
    train=df[df['listing_id'].isin(train_listings)]
    test=df[df['listing_id'].isin(test_listings)]
    return train,test


train, test=train_test_split_airbnb(df,0.8)

X_train=train[features]
y_train=train['price']
X_test=test[features]
y_test=test['price']
```

Because of the nature of the train/test split, I couldn't use gridSearchCV, so I created my own custom cross validation function for parameter grid searching.

```python
def custom_cross_validation(df, n_splits=5):
    listings = df['listing_id'].unique()
    n_listings = len(listings)

    random.shuffle(listings)

    fold_size = n_listings // n_splits

    folds = [list(listings[i * fold_size:(i + 1) * fold_size]) for i in range(n_splits)]
    leftover = listings[n_splits * fold_size:]
    for i, listing in enumerate(leftover):
        folds[i].append(listing)

    for i in range(n_splits):
        val_listings = folds[i]
        train_listings = [listing for j in range(n_splits) if j != i for listing in folds[j]]
        train = df[df['listing_id'].isin(train_listings)]
        val = df[df['listing_id'].isin(val_listings)]

        yield train, val
```

Application of custom_cross_validation

```python
param_grid = {
    'n_estimators': [100,250, 350],
    'max_depth': [None, 16,17],
    'min_samples_split': [2, 3,5],
    'min_samples_leaf': [4, 5],
    'max_features': ['sqrt','log2',0.9]
}
param_combinations = list(itertools.product(
    param_grid['n_estimators'],
    param_grid['max_depth'],
    param_grid['min_samples_split'],
    param_grid['min_samples_leaf'],
    param_grid['max_features']
))
print(param_combinations)
with open('extraTreesSearchFINAL.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerow(['n_estimators', 'max_depth', 'min_samples_split', 'min_samples_leaf', 'max_features', 'RMSE', 'R2'])
    for grid in param_combinations:
        RMSE = []
        R2 = []
        extra_trees = ExtraTreesRegressor(
            n_estimators=grid[0],
            max_depth=grid[1],
            min_samples_split=grid[2],
            min_samples_leaf=grid[3],
            max_features=grid[4],
            random_state=42,
            n_jobs=-1

        )
        for train, val in custom_cross_validation(df, n_splits=5):
            X_train = train[features]
            y_train = train['price']
            X_val = val[features]
            y_val = val['price']
            print('im in the first loop')

            extra_trees.fit(X_train, y_train)
            y_pred = extra_trees.predict(X_val)

            rmse = np.sqrt(mean_squared_error(y_val, y_pred))
            r2 = r2_score(y_val, y_pred)
            RMSE.append(rmse)
            R2.append(r2)
        print('RMSE:', np.mean(RMSE))
        print('R2:', np.mean(R2))
        writer.writerow([grid[0], grid[1], grid[2], grid[3], grid[4], np.mean(RMSE), np.mean(R2)])
```

```
[(100, None, 2, 4, 'sqrt'), (100, None, 2, 4, 'log2'), (100, None, 2, 4, 0.9), (100, None, 2, 5, 'sqrt'), (100, None, 2, 5, 'log2'), (100, None, 2, 5, 0.9), (100, None, 3, 4, 'sqrt'), (100, None, 3, 4, 'log2'), (100, None, 3, 4, 0.9), (100, None, 3, 5, 'sqrt'), (100, None, 3, 5, 'log2'), (100, None, 3, 5, 0.9), (100, None, 5, 4, 'sqrt'), (100, None, 5, 4, 'log2'), (100, None, 5, 4, 0.9), (100, None, 5, 5, 'sqrt'), (100, None, 5, 5, 'log2'), (100, None, 5, 5, 0.9), (100, 16, 2, 4, 'sqrt'), (100, 16, 2, 4, 'log2'), (100, 16, 2, 4, 0.9), (100, 16, 2, 5, 'sqrt'), (100, 16, 2, 5, 'log2'), (100, 16, 2, 5, 0.9), (100, 16, 3, 4, 'sqrt'), (100, 16, 3, 4, 'log2'), (100, 16, 3, 4, 0.9), (100, 16, 3, 5, 'sqrt'), (100, 16, 3, 5, 'log2'), (100, 16, 3, 5, 0.9), (100, 16, 5, 4, 'sqrt'), (100, 16, 5, 4, 'log2'), (100, 16, 5, 4, 0.9), (100, 16, 5, 5, 'sqrt'), (100, 16, 5, 5, 'log2'), (100, 16, 5, 5, 0.9), (100, 17, 2, 4, 'sqrt'), (100, 17, 2, 4, 'log2'), (100, 17, 2, 4, 0.9), (100, 17, 2, 5, 'sqrt'), (100, 17, 2, 5, 'log2'), (100, 17, 2, 5, 0.9), (100, 17, 3, 4, 'sqrt'), (100, 17, 3, 4, 'log2'), (100, 17, 3, 4, 0.9), (100, 17, 3, 5, 'sqrt'), (100, 17, 3, 5, 'log2'), (100, 17, 3, 5, 0.9), (100, 17, 5, 4, 'sqrt'), (100, 17, 5, 4, 'log2'), (100, 17, 5, 4, 0.9), (100, 17, 5, 5, 'sqrt'), (100, 17, 5, 5, 'log2'), (100, 17, 5, 5, 0.9), (250, None, 2, 4, 'sqrt'), (250, None, 2, 4, 'log2'), (250, None, 2, 4, 0.9), (250, None, 2, 5, 'sqrt'), (250, None, 2, 5, 'log2'), (250, None, 2, 5, 0.9), (250, None, 3, 4, 'sqrt'), (250, None, 3, 4, 'log2'), (250, None, 3, 4, 0.9), (250, None, 3, 5, 'sqrt'), (250, None, 3, 5, 'log2'), (250, None, 3, 5, 0.9), (250, None, 5, 4, 'sqrt'), (250, None, 5, 4, 'log2'), (250, None, 5, 4, 0.9), (250, None, 5, 5, 'sqrt'), (250, None, 5, 5, 'log2'), (250, None, 5, 5, 0.9), (250, 16, 2, 4, 'sqrt'), (250, 16, 2, 4, 'log2'), (250, 16, 2, 4, 0.9), (250, 16, 2, 5, 'sqrt'), (250, 16, 2, 5, 'log2'), (250, 16, 2, 5, 0.9), (250, 16, 3, 4, 'sqrt'), (250, 16, 3, 4, 'log2'), (250, 16, 3, 4, 0.9), (250, 16, 3, 5, 'sqrt'), (250, 16, 3, 5, 'log2'), (250, 16, 3, 5, 0.9), (250, 16, 5, 4, 'sqrt'), (250, 16, 5, 4, 'log2'), (250, 16, 5, 4, 0.9), (250, 16, 5, 5, 'sqrt'), (250, 16, 5, 5, 'log2'), (250, 16, 5, 5, 0.9), (250, 17, 2, 4, 'sqrt'), (250, 17, 2, 4, 'log2'), (250, 17, 2, 4, 0.9), (250, 17, 2, 5, 'sqrt'), (250, 17, 2, 5, 'log2'), (250, 17, 2, 5, 0.9), (250, 17, 3, 4, 'sqrt'), (250, 17, 3, 4, 'log2'), (250, 17, 3, 4, 0.9), (250, 17, 3, 5, 'sqrt'), (250, 17, 3, 5, 'log2'), (250, 17, 3, 5, 0.9), (250, 17
```

Scores were written to csv files and then I compared them. In the end, XgBoost was found to be the best model with 75% R2 and 119 RMSE. The xgBoost and other tree based methods were a good choice for this regression problem because they are able to capture non-linear relationships and more importantly interactions between features. The SHAP plot illustrates this nicely. For example, if you look at the plot for distanceToOcean you will notice that for most low feature values (blue), the prediction increases but only slightly, while for other low feature values the prediction is increased by a higher amount. This is likely because distanceToOcean is more important for listing containing certain attributes. For example, maybe it is more important on the south side compared to the north side of the island. Maybe it increases the prediction more when not in a resort area. We don't know the exact reason why it is, but the model is capturing interactions. This is exactly why random forests and other tree-based methods are so prone to overfitting, because they will find interactions everywhere, possible ones that are not actually meaningful. I think this model does a good job of capturing the underlying relationship between the interactions of features and the target variable. The model could certainly be improved by finding additional features and perhaps by finding more reliable data. Nevertheless, this model is a viable option for an application that provides hosts with suggested prices on the island of Hawaii.