In [ ]:
```
import pandas as pd
import pickle
import xgboost as xgb
import lightgbm as lgb
from catboost import CatBoostRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.feature_selection import RFE, SelectKBest, mutual_info_regression
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
```

In [ ]:
```
df = pd.read_csv("/kaggle/input/model-training/model_training.csv")

senti_dummy = pd.read_csv('/kaggle/input/model-training/model_training_senti.csv
```

In [ ]:
```
def preprocess_text(text):
    """
    Preprocess the text by removing punctuation and stopwords, and converting to

    Parameters:
        text (str): Input text.

    Returns:
        str: Preprocessed text.
    """
    # Remove punctuation
    text = text.translate(str.maketrans('', '', string.punctuation))
    # Convert to lowercase
    text = text.lower()
    # Tokenize the text
    tokens = word_tokenize(text)
    # Remove stopwords
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [token for token in tokens if token not in stop_words]
    # Join tokens back into a string
    preprocessed_text = ' '.join(filtered_tokens)
    return preprocessed_text


def clean_zipcode(zipcode):
    if pd.isnull(zipcode):
        return None  # Return None for NaN values
    zipcode_str = str(zipcode)  # Convert to string
    if len(zipcode_str) == 5:
        return int(zipcode_str)  # Convert to integer if length is 5
    else:
        return None  # Return None for other cases


def label_encode_column(df, column_name):
    """
    Perform label encoding on a column in a DataFrame.

    Parameters:
        df (DataFrame): The input DataFrame.
```

```
        column_name (str): The name of the column to be label encoded.

    Returns:
        DataFrame: The input DataFrame with the specified column label encoded.
    """
    # Instantiate LabelEncoder
    label_encoder = LabelEncoder()

    # Fit label encoder and transform the column
    df[column_name + '_encoded'] = label_encoder.fit_transform(df[column_name])

    return df , label_encoder
```

```
[nltk_data] Downloading package punkt to /usr/share/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to /usr/share/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [ ]:

```
# Assuming df is your DataFrame
df['zipcode'] = df['zipcode'].apply(clean_zipcode)

# Specify the columns with missing values
columns_with_missing = ['zipcode']

# Create a copy of the DataFrame with only the columns containing missing values
df_missing = df[columns_with_missing].copy()

# Instantiate the KNNImputer with the desired number of neighbors (n_neighbors)
imputer = KNNImputer(n_neighbors=5)

# Fit the imputer on the data with missing values and transform the data
imputed_data = imputer.fit_transform(df_missing)

# Convert the imputed data back to a DataFrame
imputed_df = pd.DataFrame(imputed_data, columns=columns_with_missing)

# Round and convert to integer
imputed_df['zipcode'] = imputed_df['zipcode'].round().astype(int)

# Replace the missing values in the original DataFrame with the imputed values
df[columns_with_missing] = imputed_df
```

In [ ]:

```
df,label_encoder = label_encode_column(df, 'neighbourhood')


null_indices = df[df['neighbourhood'].isnull()].index.tolist()


df.loc[null_indices, 'neighbourhood_encoded'] = float('nan')
```

In [ ]:

```
# Specify the columns with missing values
columns_with_missing = ['neighbourhood_encoded']

# Create a copy of the DataFrame with only the columns containing missing values
df_missing = df[columns_with_missing].copy()

# Instantiate the KNNImputer with the desired number of neighbors (n_neighbors)
imputer = KNNImputer(n_neighbors=5)

# Fit the imputer on the data with missing values and transform the data
```

```
imputed_data = imputer.fit_transform(df_missing)

# Convert the imputed data back to a DataFrame
imputed_df = pd.DataFrame(imputed_data, columns=columns_with_missing)

# Round and convert to integer
imputed_df['neighbourhood_encoded'] = imputed_df['neighbourhood_encoded'].round(

# Replace the missing values in the original DataFrame with the imputed values
df[columns_with_missing] = imputed_df

df['neighbourhood'] = label_encoder.inverse_transform(df['neighbourhood_encoded'
```

In [ ]:
```
# Getting dummy variablaggregatees for categorical columns
dummies_df = pd.get_dummies(df[['property_type', 'room_type', 'bed_type',
                                'cancellation_policy', 'city',
                                'host_has_profile_pic', 'host_identity_verified
                                'instant_bookable']], dtype=int)

# Getting dummy variables for 'cleaning_fee' column
cleaning_fee_df = pd.get_dummies(df['cleaning_fee'], dtype=int, prefix='cleaning
```

In [ ]:
```
# Extracting month from date columns and adding them as new columns
df['first_review_m'] = pd.to_datetime(df['first_review']).dt.month
df['host_since_m'] = pd.to_datetime(df['host_since']).dt.month
df['last_review_m'] = pd.to_datetime(df['last_review']).dt.month
```

In [ ]:
```
# Selecting columns of interest
int_df = df[['id', 'log_price', 'accommodates', 'bathrooms', 'zipcode','neighbou
             'host_response_rate', 'number_of_reviews', 'review_scores_rating',
             'bedrooms', 'latitude', 'longitude','beds', 'review_duration', 'tim
             'host_tenure', 'average_review_score', 'amenities_count',
             'has_wireless_internet', 'has_kitchen', 'has_heating', 'has_essenti
             'has_smoke_detector','first_review_m', 'host_since_m', 'last_review
```

In [ ]:
```
ModelTraining_df = pd.concat([int_df, dummies_df, cleaning_fee_df,senti_dummy],
```

Feature Selection

Report: Feature Selection Before Model Fitting for High-Dimensionality Data

Abstract:

When dealing with datasets containing a large number of features (high dimensionality), selecting the most relevant features before training a machine learning model is generally the recommended approach. This report explores the benefits of feature selection and the drawbacks of fitting all features, along with specific scenarios where fitting all features might be considered.

Benefits of Feature Selection First:

- **Reduced Training Time:** By eliminating irrelevant or redundant features, the model trains on a smaller dataset, leading to faster training and potentially lower computational cost.
- **Improved Model Performance:** Irrelevant features can introduce noise and hinder the model's ability to learn the true relationships between features and the target variable. Feature selection focuses the model on the most informative features, potentially leading to better accuracy and generalizability on unseen data.
- **Reduced Overfitting:** Overfitting occurs when the model learns patterns specific to the training data that don't generalize well. Irrelevant features can contribute to overfitting. Feature selection helps create a model that performs well on both training and testing data.
- **Improved Interpretability:** With fewer features, it's easier to understand how the model makes predictions. This is crucial for debugging issues, explaining the model's behavior, and gaining insights into the data.

Drawbacks of Fitting All Features First:

- **Increased Training Time:** Training on all features takes longer, especially for large datasets.
- **Potential for Overfitting:** Including irrelevant features can lead to overfitting, impacting the model's performance on unseen data.
- **Reduced Interpretability:** With a large number of features, it's difficult to understand which features contribute the most to the model's predictions.

When Might Fitting All Features Be Considered?

- **Limited Data:** If you have a very small dataset, feature selection might remove too much information, potentially harming model performance.
- **Domain Knowledge Lacking:** If you have limited knowledge about the features and their relationship to the target variable, feature selection methods might not be as effective.

Conclusion:

For high-dimensional data (many features), selecting the best features before fitting the model is generally preferred. It leads to faster training, potentially better performance, and often provides a more interpretable model. However, there might be situations where fitting all features is considered due to limited data or lack of domain knowledge.

- Start with filter methods: Due to their efficiency and interpretability, filter methods are a good first approach to identify potentially relevant features. Popular choices include correlation analysis, information gain, and L1 regularization.

- Refine with wrapper or embedded methods: If needed, refine the feature selection using wrapper methods (forward selection, RFE) or embedded methods (LASSO) for potentially better performance. These methods can be computationally expensive, so use them after filter methods have narrowed down the candidate features.

- Consider PCA: Especially for very high-dimensional data, PCA can be a valuable tool to reduce dimensionality while preserving the most informative features.

In [ ]:
```
X = ModelTraining_df.drop('log_price', axis=1)
y = ModelTraining_df['log_price']
```

In [ ]:
```
# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_

# Perform feature selection
k_best = SelectKBest(mutual_info_regression, k=20)  # Select top 20 features bas
X_train_selected = k_best.fit_transform(X_train, y_train)
X_test_selected = k_best.transform(X_test)

# Get selected feature indices
selected_feature_indices = k_best.get_support(indices=True)

# Get selected feature names
selected_features = X.columns[selected_feature_indices]

print("Selected features:", selected_features)
```

```
Selected features: Index(['accommodates', 'bathrooms', 'zipcode', 'neighbourhood_
encoded',
       'number_of_reviews', 'review_scores_rating', 'bedrooms', 'latitude',
       'longitude', 'beds', 'time_since_last_review', 'host_tenure',
       'average_review_score', 'amenities_count', 'property_type_Apartment',
       'room_type_Entire home/apt', 'room_type_Private room',
       'room_type_Shared room', 'cleaning_fee__False', 'cleaning_fee__True'],
      dtype='object')
```

In [ ]:
```
# # Standardize features
# scaler = StandardScaler()
# X_scaled = scaler.fit_transform(X)

# # RFE
# estimator = LinearRegression()
# rfe_selector = RFE(estimator, n_features_to_select=15, step=1)  # Select top 5
# X_rfe_selected = rfe_selector.fit_transform(X_scaled, y)
# selected_rfe_features = X.columns[rfe_selector.get_support(indices=True)]

# print("Selected features using RFE:", selected_rfe_features)
```

Wrapper or Embedded Methods (e.g., RFR):

1. **Computationally Expensive:** Wrapper methods involve training the model multiple times with different subsets of features, which can be computationally expensive, especially for large datasets or complex models.

2. **Model Dependent:** Wrapper methods rely on the performance of a specific machine learning model. If the chosen model is not well-suited for the data or the task, the feature selection process may not yield optimal results.

3. **Overfitting Risk:** There's a risk of overfitting, especially with wrapper methods like forward selection, where features are added one by one. This can lead to selecting

features that are only relevant to the training set and do not generalize well to unseen data.

In [ ]:
```
# # Standardize features
# scaler = StandardScaler()
# X_scaled = scaler.fit_transform(X)

# # PCA
# pca = PCA(n_components=15)  # Reduce to 5 principal components
# X_pca = pca.fit_transform(X_scaled)

# print("PCA components:", pca.components_)
```

PCA (Principal Component Analysis):

1. **Loss of Interpretability:** PCA transforms the original features into a new set of orthogonal features (principal components), which may not be directly interpretable in terms of the original features. This loss of interpretability can make it challenging to understand the meaning of each principal component.

2. **Information Loss:** PCA aims to maximize variance in the data, but in doing so, it may discard some information that is not captured by the principal components with the highest variance. This can lead to a loss of information, especially if the lower-variance components contain important features.

3. **Nonlinear Relationships:** PCA assumes linear relationships between variables. If the underlying relationships in the data are nonlinear, PCA may not effectively capture these relationships, leading to suboptimal dimensionality reduction.

Model Training

In [ ]:
```
# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_

# Initialize regression models
models = {
    'linear_regression': LinearRegression(),
    'ridge_regression': Ridge(),
    'lasso_regression': Lasso(),
    'elastic_net_regression': ElasticNet(),
    'decision_tree_regression': DecisionTreeRegressor(),
    'random_forest_regression': RandomForestRegressor(),
    'gradient_boosting_regression': GradientBoostingRegressor(),
    'xgboost_regression': xgb.XGBRegressor(),
    'lightgbm_regression': lgb.LGBMRegressor(),
    'catboost_regression': CatBoostRegressor(silent=True)
}

# Train each model
for name, model in models.items():
```

```
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    # Calculate Mean Squared Error (MSE)
    mse = mean_squared_error(y_test, y_pred)
    mae = mean_absolute_error(y_test, y_pred)
    # Print the evaluation metric
    print(f"{name} MSE: {mse}")
    print(f"{name} MSE: {mae}")

    print("-"*30)


# Save the models dictionary to a pickle file
with open('trained_models.pkl', 'wb') as f:
    pickle.dump(models, f)
```

```
linear_regression MSE: 0.24105005154305392
linear_regression MSE: 0.36903727653140234
------------------------------
ridge_regression MSE: 0.24105137756307524
ridge_regression MSE: 0.369037250018861
------------------------------
lasso_regression MSE: 0.5072913482976185
lasso_regression MSE: 0.5559326285430588
------------------------------
elastic_net_regression MSE: 0.4017577169551864
elastic_net_regression MSE: 0.4900197170786799
------------------------------
decision_tree_regression MSE: 0.31182273901256996
decision_tree_regression MSE: 0.4024522341569183
------------------------------
random_forest_regression MSE: 0.15521153081387837
random_forest_regression MSE: 0.28350343755029006
------------------------------
gradient_boosting_regression MSE: 0.1727719373971861
gradient_boosting_regression MSE: 0.3036576074580185
------------------------------
xgboost_regression MSE: 0.15205490511774514
xgboost_regression MSE: 0.2824325192706379
------------------------------
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing
was 0.008831 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 2146
[LightGBM] [Info] Number of data points in the train set: 59288, number of used f
eatures: 20
[LightGBM] [Info] Start training from score 4.780538
lightgbm_regression MSE: 0.1529626331375161
lightgbm_regression MSE: 0.28420173675727073
------------------------------
catboost_regression MSE: 0.1477884671345181
catboost_regression MSE: 0.2784356276589088
------------------------------
```

In [ ]:

```
# Load the trained models from the pickle file
with open('trained_models.pkl', 'rb') as f:
    models = pickle.load(f)
```

```
# Initialize dictionaries to store evaluation metrics for each model
evaluation_metrics = {
    'Model': [],
    'MSE': [],
    'MAE': [],
    'R2': []
}

# Evaluate the performance of each model
for name, model in models.items():
    # Predict on the test data
    y_pred = model.predict(X_test)

    # Calculate evaluation metrics
    mse = mean_squared_error(y_test, y_pred)
    mae = mean_absolute_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    # Store evaluation metrics in the dictionary
    evaluation_metrics['Model'].append(name)
    evaluation_metrics['MSE'].append(mse)
    evaluation_metrics['MAE'].append(mae)
    evaluation_metrics['R2'].append(r2)
```

SHAP

In [ ]:

```
import shap

# Initialize the SHAP explainer for each model
xgb_explainer = shap.Explainer(xgb_reg)
lgb_explainer = shap.Explainer(lgb_reg)
catboost_explainer = shap.Explainer(catboost_reg)

# Compute SHAP values for each model
xgb_shap_values = xgb_explainer.shap_values(X_test)
lgb_shap_values = lgb_explainer.shap_values(X_test)
catboost_shap_values = catboost_explainer.shap_values(X_test)

# Store SHAP values in the trained_models dictionary
trained_models['XGBoost']['shap_values'] = xgb_shap_values
trained_models['LightGBM']['shap_values'] = lgb_shap_values
trained_models['CatBoost']['shap_values'] = catboost_shap_values

# Save the trained models dictionary to a pickle file
with open('trained_models_with_metrics_and_shap.pkl', 'wb') as f:
    pickle.dump(trained_models, f)
```

In [ ]:

```
# Initialize XGBoost Regression with hyperparameters
xgb_reg = xgb.XGBRegressor(
    n_estimators=100,
    max_depth=3,
    learning_rate=0.1,
    subsample=0.8,
    colsample_bytree=0.8,
    objective='reg:squarederror',
    random_state=42
```

```
)

# Initialize LightGBM Regression with hyperparameters
lgb_reg = lgb.LGBMRegressor(
    n_estimators=100,
    max_depth=3,
    learning_rate=0.1,
    subsample=0.8,
    colsample_bytree=0.8,
    objective='regression',
    random_state=42
)

# Initialize CatBoost Regression with hyperparameters
catboost_reg = CatBoostRegressor(
    n_estimators=100,
    max_depth=6,
    learning_rate=0.1,
    subsample=0.8,
    colsample_bylevel=0.8,
    objective='MAE',
    random_seed=42
)

# Train XGBoost Regression
xgb_reg.fit(X_train, y_train)

# Train LightGBM Regression
lgb_reg.fit(X_train, y_train)

# Train CatBoost Regression
catboost_reg.fit(X_train, y_train)

# Initialize a dictionary to store models, evaluation metrics, and feature impor
trained_models = {
    'XGBoost': {'model': xgb_reg, 'evaluation_metrics': {}, 'feature_importance'
    'LightGBM': {'model': lgb_reg, 'evaluation_metrics': {}, 'feature_importance
    'CatBoost': {'model': catboost_reg, 'evaluation_metrics': {}, 'feature_impor
}

# Evaluate each model and store evaluation metrics
for model_name, model_data in trained_models.items():
    model = model_data['model']
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    mae = mean_absolute_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    model_data['evaluation_metrics'] = {'MSE': mse, 'MAE': mae, 'R2': r2}

# Store feature importance for XGBoost and LightGBM models
trained_models['XGBoost']['feature_importance'] = xgb_reg.feature_importances_
trained_models['LightGBM']['feature_importance'] = lgb_reg.feature_importances_

# CatBoost provides feature importance as part of its model object
trained_models['CatBoost']['feature_importance'] = catboost_reg.get_feature_impo

# Save the trained models dictionary to a pickle file
with open('trained_models_with_metrics.pkl', 'wb') as f:
    pickle.dump(trained_models, f)
```

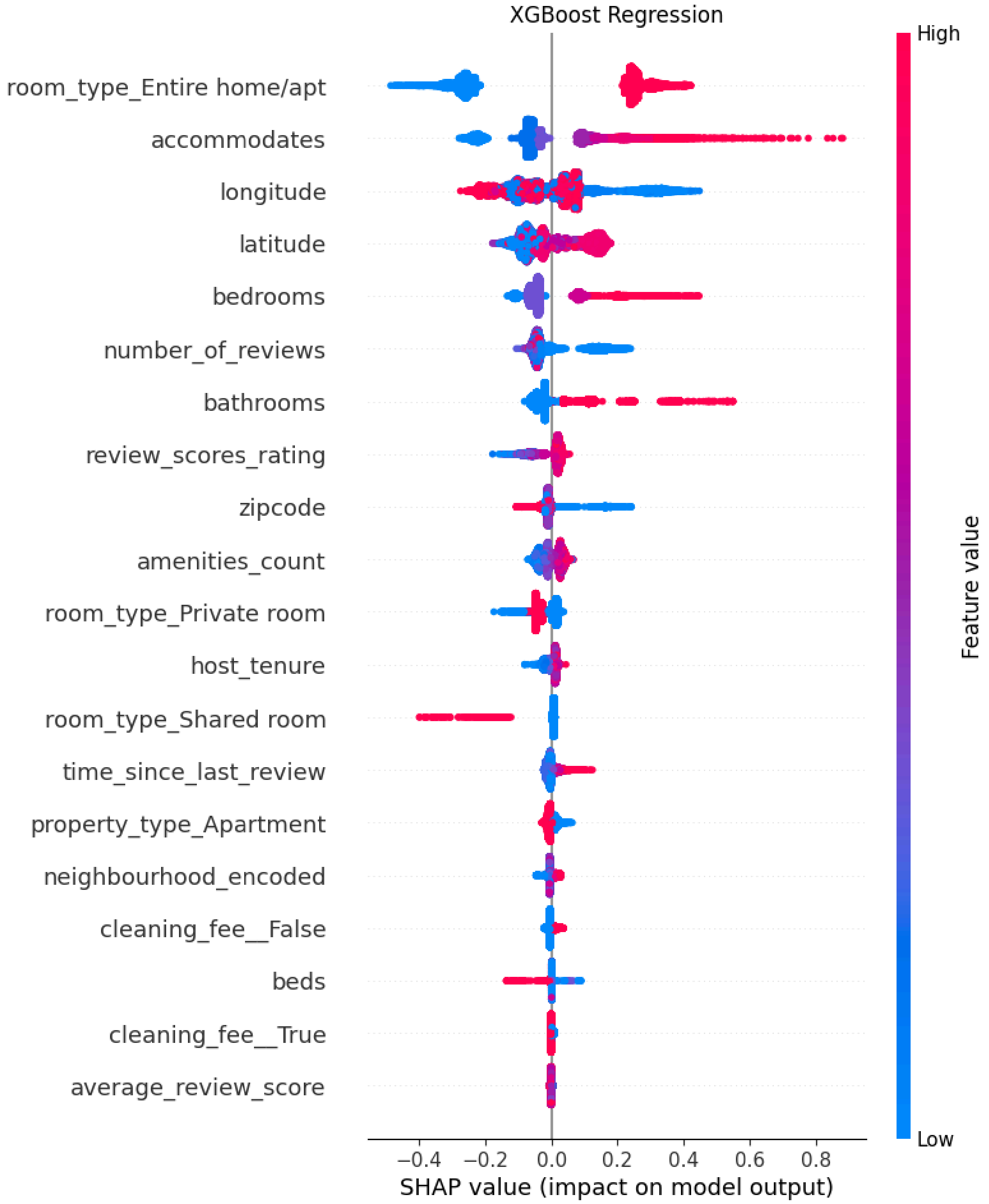
```
[LightGBM] [Warning] Found whitespace in feature_names, replace with underlines
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing
was 0.015452 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 2146
[LightGBM] [Info] Number of data points in the train set: 59288, number of used f
eatures: 20
[LightGBM] [Info] Start training from score 4.780538
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
0:	learn: 0.5292717	total: 13ms	remaining: 1.28s
1:	learn: 0.5020531	total: 25.1ms	remaining: 1.23s
2:	learn: 0.4780849	total: 37.3ms	remaining: 1.21s
3:	learn: 0.4580973	total: 49.2ms	remaining: 1.18s
4:	learn: 0.4407128	total: 61.4ms	remaining: 1.17s
5:	learn: 0.4262902	total: 73.5ms	remaining: 1.15s
6:	learn: 0.4133027	total: 84.7ms	remaining: 1.13s
7:	learn: 0.4029372	total: 96.6ms	remaining: 1.11s
8:	learn: 0.3933309	total: 108ms	remaining: 1.09s
9:	learn: 0.3844541	total: 121ms	remaining: 1.09s
10:	learn: 0.3775177	total: 133ms	remaining: 1.07s
11:	learn: 0.3705298	total: 145ms	remaining: 1.06s
12:	learn: 0.3654351	total: 157ms	remaining: 1.05s
13:	learn: 0.3605758	total: 169ms	remaining: 1.04s
14:	learn: 0.3565841	total: 181ms	remaining: 1.02s
15:	learn: 0.3528353	total: 193ms	remaining: 1.01s
16:	learn: 0.3492485	total: 203ms	remaining: 993ms
17:	learn: 0.3461501	total: 215ms	remaining: 978ms
18:	learn: 0.3434251	total: 226ms	remaining: 964ms
19:	learn: 0.3408800	total: 238ms	remaining: 951ms
20:	learn: 0.3388004	total: 249ms	remaining: 937ms
21:	learn: 0.3368661	total: 260ms	remaining: 923ms
22:	learn: 0.3348756	total: 272ms	remaining: 912ms
23:	learn: 0.3333416	total: 283ms	remaining: 897ms
24:	learn: 0.3311657	total: 294ms	remaining: 882ms
25:	learn: 0.3298528	total: 306ms	remaining: 872ms
26:	learn: 0.3286015	total: 319ms	remaining: 863ms
27:	learn: 0.3273916	total: 332ms	remaining: 855ms
28:	learn: 0.3262418	total: 343ms	remaining: 840ms
29:	learn: 0.3246378	total: 355ms	remaining: 828ms
30:	learn: 0.3235921	total: 366ms	remaining: 815ms
31:	learn: 0.3222863	total: 378ms	remaining: 803ms
32:	learn: 0.3211385	total: 389ms	remaining: 789ms
33:	learn: 0.3199293	total: 400ms	remaining: 777ms
34:	learn: 0.3190597	total: 411ms	remaining: 763ms
35:	learn: 0.3183955	total: 423ms	remaining: 751ms
36:	learn: 0.3176559	total: 434ms	remaining: 738ms
37:	learn: 0.3169993	total: 445ms	remaining: 725ms
38:	learn: 0.3164459	total: 456ms	remaining: 714ms
39:	learn: 0.3155340	total: 469ms	remaining: 704ms
40:	learn: 0.3145677	total: 481ms	remaining: 692ms
41:	learn: 0.3141420	total: 492ms	remaining: 680ms
42:	learn: 0.3134567	total: 503ms	remaining: 667ms
43:	learn: 0.3128501	total: 515ms	remaining: 655ms
44:	learn: 0.3121332	total: 528ms	remaining: 645ms
45:	learn: 0.3115547	total: 539ms	remaining: 633ms
46:	learn: 0.3106945	total: 551ms	remaining: 621ms
47:	learn: 0.3101771	total: 562ms	remaining: 609ms
48:	learn: 0.3097459	total: 574ms	remaining: 598ms
49:	learn: 0.3090876	total: 586ms	remaining: 586ms
```

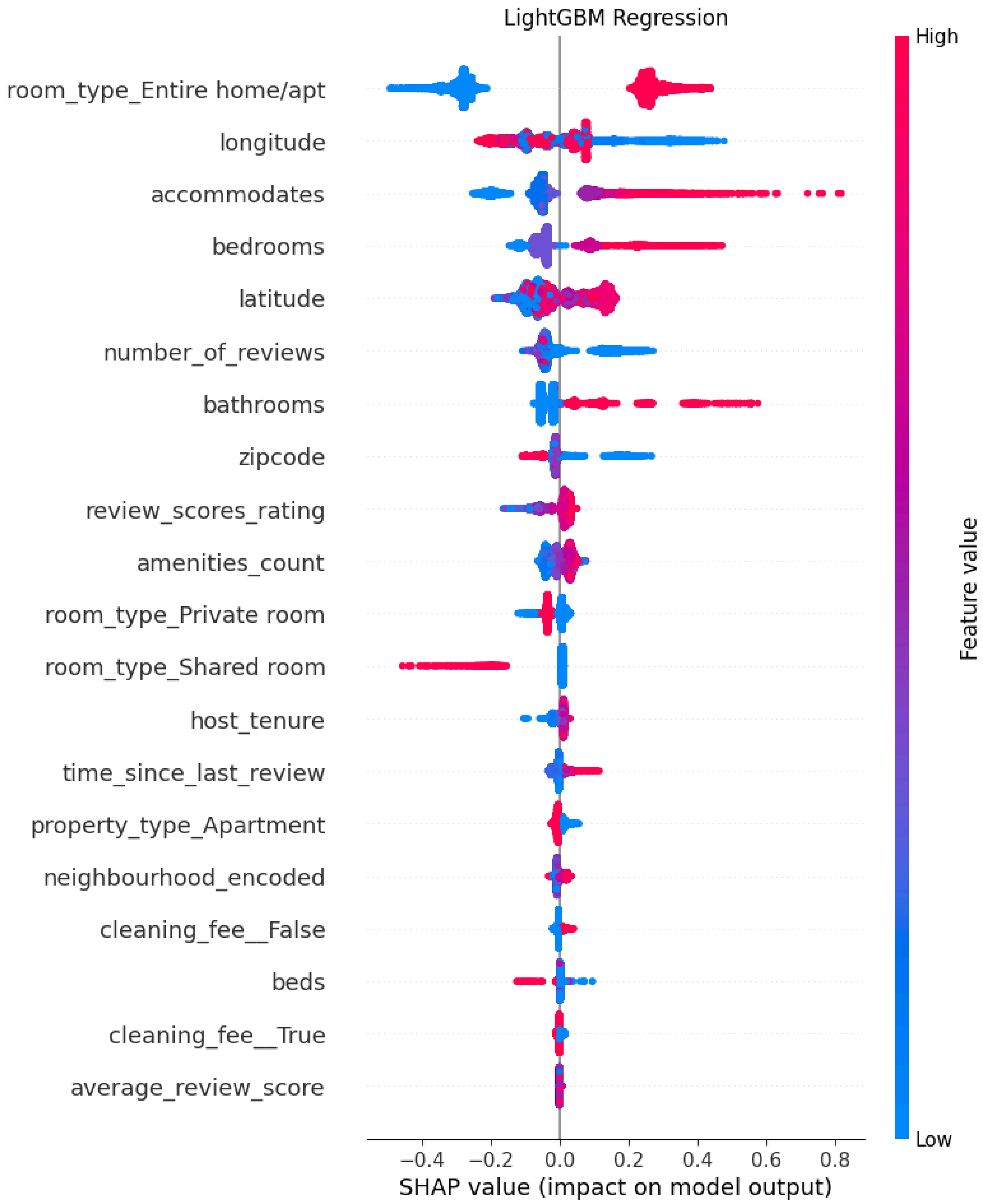
```
50:	learn: 0.3088007	total: 597ms	remaining: 574ms
51:	learn: 0.3081422	total: 609ms	remaining: 562ms
52:	learn: 0.3076767	total: 620ms	remaining: 550ms
53:	learn: 0.3073437	total: 631ms	remaining: 538ms
54:	learn: 0.3069962	total: 642ms	remaining: 526ms
55:	learn: 0.3065169	total: 655ms	remaining: 514ms
56:	learn: 0.3062033	total: 667ms	remaining: 503ms
57:	learn: 0.3058495	total: 679ms	remaining: 492ms
58:	learn: 0.3054431	total: 691ms	remaining: 480ms
59:	learn: 0.3051142	total: 701ms	remaining: 467ms
60:	learn: 0.3046800	total: 713ms	remaining: 456ms
61:	learn: 0.3042588	total: 724ms	remaining: 444ms
62:	learn: 0.3040620	total: 737ms	remaining: 433ms
63:	learn: 0.3038499	total: 748ms	remaining: 421ms
64:	learn: 0.3035325	total: 760ms	remaining: 409ms
65:	learn: 0.3030499	total: 772ms	remaining: 398ms
66:	learn: 0.3028277	total: 784ms	remaining: 386ms
67:	learn: 0.3024201	total: 795ms	remaining: 374ms
68:	learn: 0.3020773	total: 806ms	remaining: 362ms
69:	learn: 0.3018288	total: 817ms	remaining: 350ms
70:	learn: 0.3016000	total: 829ms	remaining: 338ms
71:	learn: 0.3011825	total: 840ms	remaining: 327ms
72:	learn: 0.3008193	total: 854ms	remaining: 316ms
73:	learn: 0.3005548	total: 870ms	remaining: 306ms
74:	learn: 0.3002296	total: 882ms	remaining: 294ms
75:	learn: 0.3001088	total: 893ms	remaining: 282ms
76:	learn: 0.2997899	total: 904ms	remaining: 270ms
77:	learn: 0.2994804	total: 916ms	remaining: 258ms
78:	learn: 0.2992194	total: 927ms	remaining: 247ms
79:	learn: 0.2990569	total: 938ms	remaining: 235ms
80:	learn: 0.2988599	total: 950ms	remaining: 223ms
81:	learn: 0.2985505	total: 962ms	remaining: 211ms
82:	learn: 0.2983718	total: 978ms	remaining: 200ms
83:	learn: 0.2980636	total: 998ms	remaining: 190ms
84:	learn: 0.2978309	total: 1.01s	remaining: 179ms
85:	learn: 0.2974944	total: 1.03s	remaining: 167ms
86:	learn: 0.2973512	total: 1.04s	remaining: 156ms
87:	learn: 0.2971847	total: 1.06s	remaining: 144ms
88:	learn: 0.2970160	total: 1.07s	remaining: 132ms
89:	learn: 0.2968159	total: 1.08s	remaining: 120ms
90:	learn: 0.2965025	total: 1.09s	remaining: 108ms
91:	learn: 0.2963253	total: 1.1s	remaining: 95.7ms
92:	learn: 0.2958876	total: 1.11s	remaining: 83.8ms
93:	learn: 0.2957152	total: 1.12s	remaining: 71.8ms
94:	learn: 0.2954276	total: 1.14s	remaining: 59.9ms
95:	learn: 0.2952674	total: 1.15s	remaining: 47.9ms
96:	learn: 0.2949206	total: 1.16s	remaining: 35.9ms
97:	learn: 0.2947312	total: 1.17s	remaining: 23.9ms
98:	learn: 0.2946000	total: 1.18s	remaining: 12ms
99:	learn: 0.2943565	total: 1.19s	remaining: 0us
```

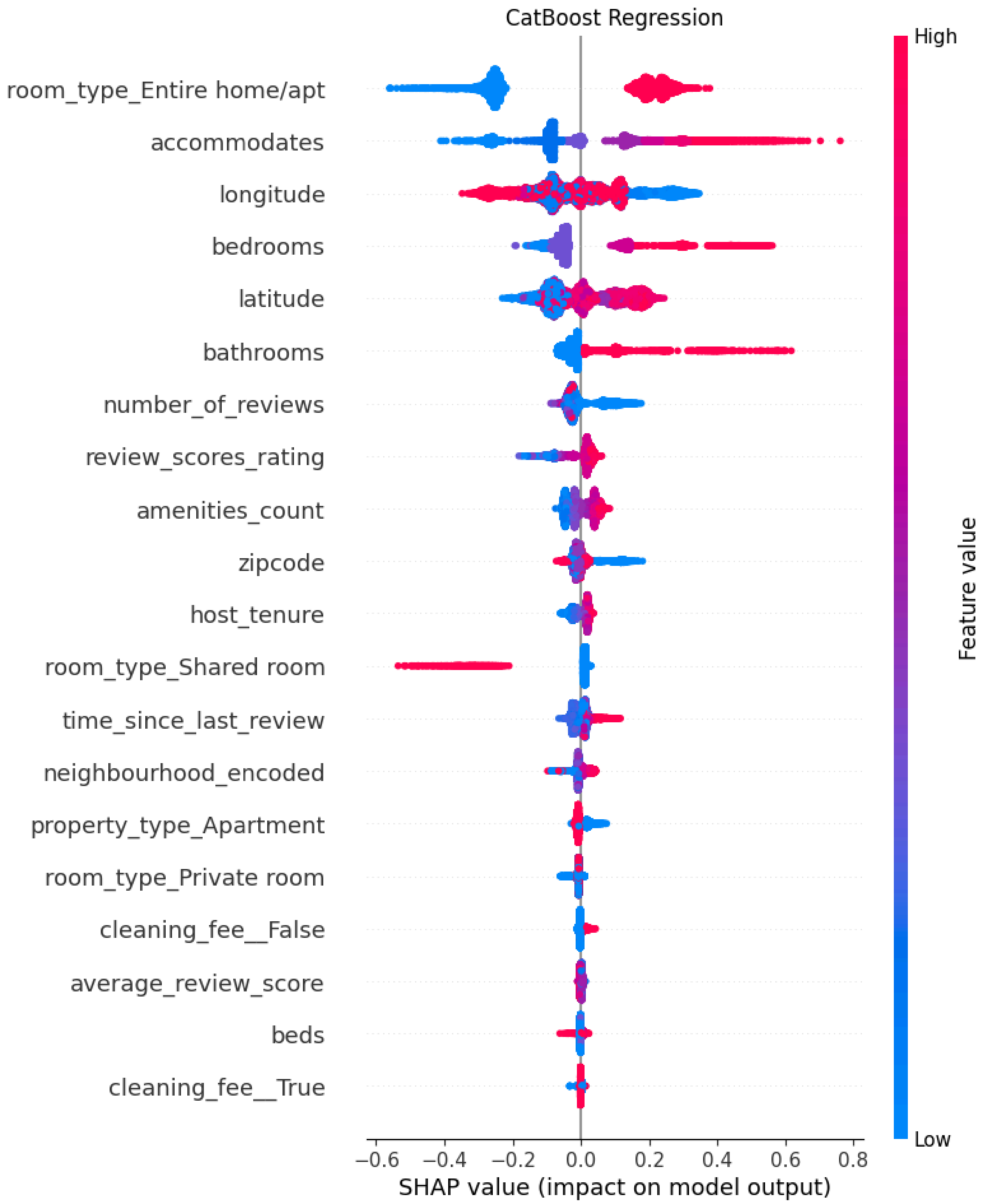
In [ ]:

```
# Plot SHAP summary plots for each model
shap.summary_plot(xgb_shap_values, X_test, feature_names=X_test.columns, show=Fa
plt.title('XGBoost Regression')
plt.savefig('xgboost_shap_summary_plot.png')
plt.show()

shap.summary_plot(lgb_shap_values, X_test, feature_names=X_test.columns, show=Fa
plt.title('LightGBM Regression')
plt.savefig('lightgbm_shap_summary_plot.png')
```

```
plt.show()

shap.summary_plot(catboost_shap_values, X_test, feature_names=X_test.columns, sh
plt.title('CatBoost Regression')
plt.savefig('catboost_shap_summary_plot.png')
plt.show()
```




LightGBM Regression
room_type_Entire home/apt
longitude
accommodates
bedrooms
latitude
number_of_reviews
bathrooms
zipcode
review_scores_rating
amenities_count
room_type_Private room
room_type_Shared room
host_tenure
time_since_last_review
property_type_Apartment
neighbourhood_encoded
cleaning_fee__False
beds
cleaning_fee__True
average_review_score
−0.4 −0.2 0.0 0.2 0.4 0.6 0.8
SHAP value (impact on model output)
High
Low
Feature value



In [ ]:
```
!jupyter nbconvert --to html HomeStay_OHE_MT.ipynb
```

In [ ]: