Template for In-Class Kaggle Competition Writeup

CompSci 671

Due: Nov 26th 2024

Kaggle Competition Link

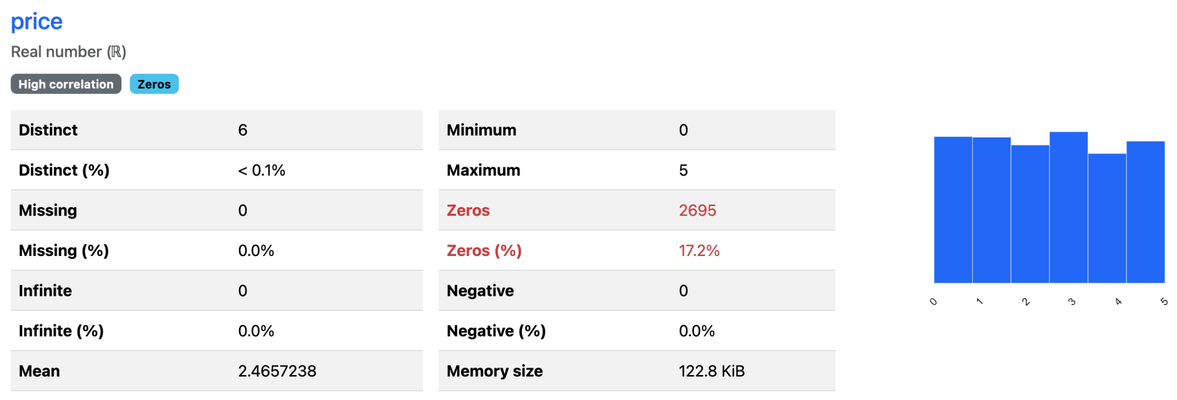
Your Kaggle ID (on the leaderboard): zoe1trick

1 Exploratory Data Analysis

1.1 Overview of the Airbnb Dataset

The dataset was Ib-scraped from Airbnb and contains house listing information in New York City. The task is to build a machine learning model to predict the price range of a particular house listing, ranging from \$0 to \$1000, divided equally into six bins. Figure 1 provides an overview of the distribution of the response variable, *price*, in the training data. The distribution appears to be fairly uniform, so I do not need to address class imbalance issues.

My exploratory analysis is divided into three parts based on the type of predictors: text variables, categorical variables, and numeric variables.



price

Real number (ℝ)

High correlation Zeros

Distinct	6	Minimum	0
Distinct (%)	< 0.1%	Maximum	5
Missing	0	Zeros	2695
Missing (%)	0.0%	Zeros (%)	17.2%
Infinite	0	Negative	0
Infinite (%)	0.0%	Negative (%)	0.0%
Mean	2.4657238	Memory size	122.8 KiB

Figure 1: Price distribution in the training data

1.2 Text Variables

The dataset contains few text variables: *name*, *description*, *amenities*, and *reviews*. While *name* and *description* are straightforward strings of sentences, *amenities* is a list represented as a string, and *reviews* contain text in multiple languages, posing additional challenges for analysis. As seen in the word clouds (not shown here), conjunction words such as "and," "on," and "or" are frequent in these strings. Therefore, to extract meaningful information, I performed data cleaning to remove stop words, special characters, and extra spaces.

For the *name* variable, I attempted to extract keywords such as "spacious," "cozy," and "luxurious." HoIver, these words are too specific and introduced bias. Ultimately, I decided not to include this variable in my final model.

Regarding the *description* variable, I hypothesized that a more detailed description might correlate with the price. Therefore, I calculated the word count of the host's description. Missing descriptions are assigned a word count of zero, which makes sense because the host did not provide any details. The length of the description proved to be an important feature, as shown in Figure 5 and in the feature importance plot presented later.

For the *amenities* variable, which has no missing values, I tried three methods to extract meaningful features:

1. **Latent Dirichlet Allocation (LDA):** I applied LDA to identify latent topics within the amenities listings. HoIver, the resulting topics are not Ill-separated, making it difficult to assign clear categories to each topic. For example, the word "alarm" appeared in several topics, and kitchen-related items like "cooking" and "oven" (which should belong to a "kitchen supplies" category) are classified in Topic 9.

```
Topic 0: coffee, cleaning, clothing, storage, products, freezer,
kettle, table, water, wine
Topic 1: alarm, kitchen, tv, wifi, carbon, monoxide, smoke, ac,
washer, workspace
Topic 2: aid, kit, first, fare, extinguisher, alarm, dedicated,
workspace, monoxide, carbon
Topic 3: building, paid, staff, gym, premises, alloId, off,
elevator, dryer, checkin
Topic 4: property, security, exterior, cameras, tub, keypad,
fare, extinguisher, hot, noise
Topic 5: outdoor, area, grill, bbq, furniture, dining, patio,
balcony, private, backyard
Topic 6: alloId, crib, pets, chair, high, entrance, dryer,
unit, private, long
Topic 7: bedroom, door, lock, alarm, kitchen, wifi, ac, smart,
workspace, dedicated
Topic 8: host, greets, parking, free, cable, standard, street,
essentials, dryer, hangers
Topic 9: lockbox, self, checkin, basics, cooking, oven, dishes,
silverware, hot, water
```

2. **Word2Vec:** I also experimented with Word2Vec embeddings, but the results are less promising than those from LDA.

3. **Pre-defined Categories:** I attempted to map amenities to pre-defined categories (e.g., kitchen supplies: kettle, oven, dishwasher). HoIver, including these categories in the model negatively impacted performance.

Ultimately, I opted for a simpler yet effective approach: counting the number of amenities listed for each property. As shown in Figure 6, properties with more amenities tend to have higher prices.

As for the *reviews* variable, it contains text in multiple languages, such as Chinese and Spanish. Due to time constraints, I did not utilize this column in my analysis. In future work, performing sentiment analysis on reviews could be beneficial, although I believe that much of the sentiment information is already captured in the rating scores.

1.3 Categorical Variables

For the variables *neighborhood_group_cleansed* and *neighborhood_cleansed*, there are no missing values. After examining the mean price statistics for each category, I determined that these variables are valuable for my model. To encode them numerically, I used ordinal encoding, which can handle unseen categories in the test data.

The *property_type* variable originally contains 59 distinct categories. To simplify the analysis and avoid duplication with the *room_type* variable, I consolidated the categories into 14 broader groups. This also helped to address duplicates in the original categories, such as "casa particular" and "room with breakfast."

I also created a new categorical variable derived from the *bathrooms_text* field. I observed that this field contains information about the number of bathrooms and whether they are shared or private. I extracted this information to create separate features for the number of bathrooms and the type of bathroom, and then discarded the original text.

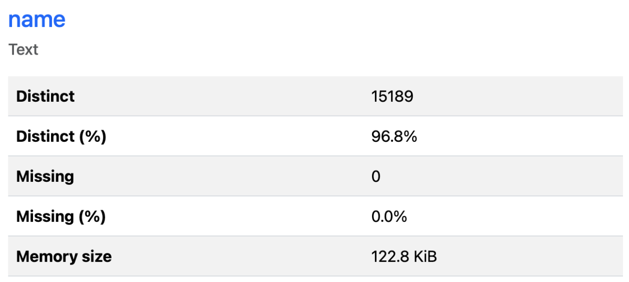
1.4 Numeric Variables

I engineered several new numeric features to enhance my model:

- **has_review**: Indicates whether the listing has any reviews, derived from *number_of_reviews*.
- **days_since_last_review**: Calculates the number of days from November 1, 2024, to the date of the last review. Missing values are filled with the median, as the presence of reviews is already indicated by *has_review*.
- **first_review_recalculated** and **last_review_recalculated**: Combine the year since Airbnb's founding in 2008 with the month (divided by 12) to create a continuous time variable.
- **availability_90_ratio**: Provides a straightforward representation of how frequently the listing is available in a 90-day window.
- **host_years**: Indicates how many years the host has been active on Airbnb.
- **availability_30_60**: Calculates the ratio of availability betIen 30-day and 60-day windows, aiming to capture short-term availability trends.
- **bed_per_accommodate** and **bathroom_per_accommodate**: Provide a clearer understanding of the space available per guest during the stay.

Given that location is a critical factor for Airbnb listings in New York City, I attempted to create a *geo_category* variable from the latitude and longitude coordinates, hoping to extract additional information for the model. HoIver, this feature did not significantly improve the model's performance and was subsequently removed, considering that other features already adequately represent the geographical location.

For missing values in the numeric variables, I filled them with zero where appropriate (e.g., *host_is_superhost*, *beds*, *bathrooms*, *number_of_reviews_ltm*). For other variables, I imputed missing values using the mean.



name

Text

Distinct	15189
Distinct (%)	96.8%
Missing	0
Missing (%)	0.0%
Memory size	122.8 KiB

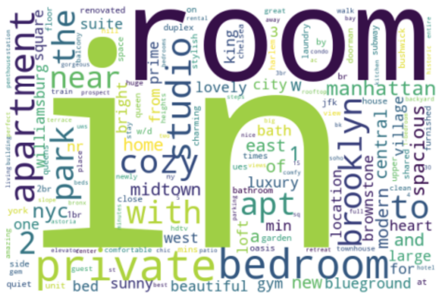
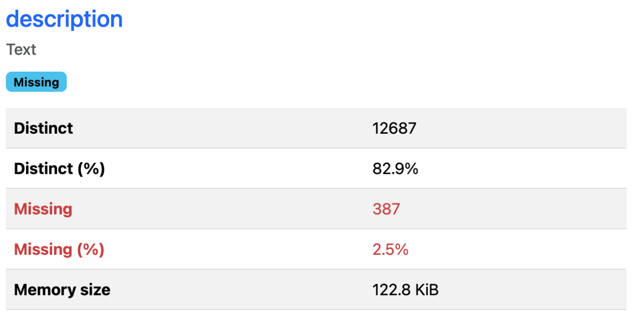


Figure 2: Word Cloud: Name



description

Text

Missing

Distinct	12687
Distinct (%)	82.9%
Missing	387
Missing (%)	2.5%
Memory size	122.8 KiB

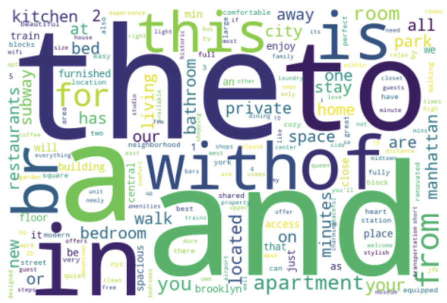
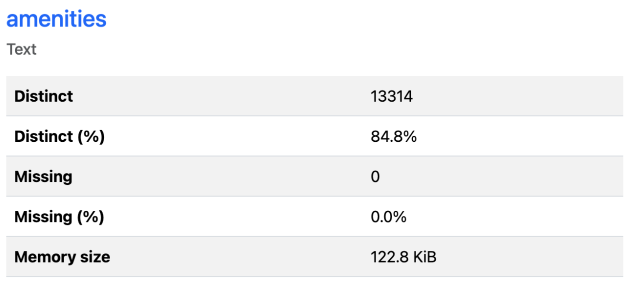


Figure 3: Word Cloud: Description



amenities

Text

Distinct	13314
Distinct (%)	84.8%
Missing	0
Missing (%)	0.0%
Memory size	122.8 KiB

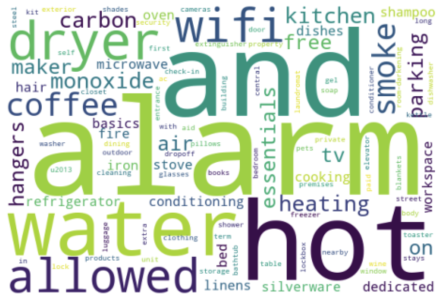


Figure 4: Word Cloud: Amenities

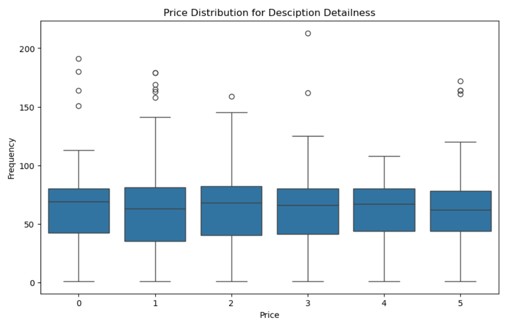


Figure 5: Detailness vs. price

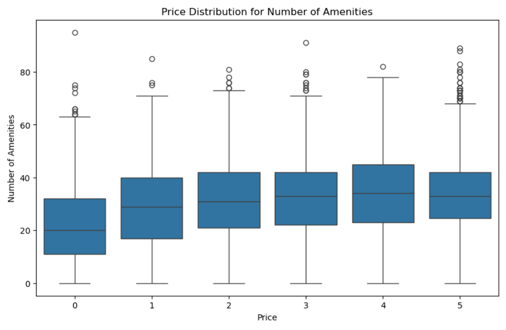


Figure 6: Number of Amentities vs. price

2 Models

2.1 Initial Model Selection

For the initial modeling phase, I selected two algorithms: **XGBoost** and **Random Forest**. The choice was motivated by their proven effectiveness in handling structured data, their ability to model complex nonlinear relationships, and the availability of online source as Ill.

- **Random Forest** is an ensemble learning method that constructs multiple decision trees during training and outputs the mean prediction of the individual trees. The trees are naturally more interpretable and explainable. This is the main reason I chose as the first model to begin. The feature importance plot gives me insights on the tangling relationship in the dataset.
- **XGBoost** is an optimized distributed gradient boosting algorithm, which is highly efficient, flexible, and portable. It generally outperforms ensemble algorithms like **Random Forest**. Indeed in my first trial, it has RMSE of 0.79 while **Random Forest** has 0.83 after tuning.

2.2 Enhancing Model Performance with Stacking

To further improve the predictive poIr of our model, I employed a **stacking** approach by combining multiple algorithms: **XGBoost**, **Random Forest**, **Extra Trees**, and **CatBoost**, using **Ridge Regression** as the meta-model.

Stacking, or stacked generalization, is an ensemble learning technique that combines multiple base models to improve overall performance. Each base model captures different aspects or patterns in the data, and the meta-model learns to optimally combine their predictions.

Extra Trees **Extra Trees** (Extremely Randomized Trees) is similar to Random Forest but introduces more randomness by selecting cut points randomly for each feature, which can reduce variance and help with overfitting. The reasons for including Extra Trees are:

- **Model Diversity:** Adding Extra Trees increases the diversity among base models, which is beneficial for ensemble methods.
- **Computational Efficiency:** Extra Trees are faster to train compared to other tree-based models because they use the entire dataset and random splits.

CatBoost **CatBoost** is a gradient boosting algorithm that handles categorical variables natively, reducing the need for extensive preprocessing. The motivations for selecting CatBoost include:

- **Handling of Categorical Variables:** CatBoost efficiently processes categorical features without explicit encoding, preserving information and reducing the risk of introducing biases.
- **Ease of Training:** It requires minimal data preprocessing and hyperparameter tuning.
- **High-Quality Library:** CatBoost provides a robust and Ill-documented library, facilitating seamless integration.

Ridge Regression as Meta-Model For the meta-model, I chose **Ridge Regression**, a linear model with L2 regularization. The reasons for this choice are:

- **Simplicity and Interpretability:** Ridge Regression is simple to implement and its coefficients can provide insights into how each base model contributes to the final prediction.
- **Regularization:** The L2 penalty helps prevent overfitting by shrinking the coefficients, which is especially useful when combining outputs from multiple models.
- **Computational Efficiency:** Training a Ridge Regression model is computationally inexpensive compared to more complex models.

3 Training

3.1 Extra Trees Regressor

Training Algorithm: The model is trained using the entire training dataset without bootstrapping. For each tree in the ensemble, the algorithm selects a random subset of features at each split point and chooses a random split value for each feature. The best among these random splits is chosen based on the reduction in variance (for regression tasks).

Parameter Optimization: I used *Randomized Search Cross-Validation* to tune hyperparameters such as the number of estimators, maximum depth, minimum samples split, and maximum features. This approach allows for efficient exploration of the hyperparameter space by testing a fixed number of random parameter combinations. The scikit-learn library's implementation of Extra Trees was utilized, which provides efficient methods for building and evaluating the ensemble.

Runtime Estimate: Training the Extra Trees Regressor took approximately **24.7 seconds**.

3.2 Random Forest Regressor

Training Algorithm: Each tree in the forest is trained on a bootstrap sample of the data. At each node, a random subset of features is considered for splitting, and the best split is chosen based on the reduction in variance. The trees are grown to their maximum depth unless constraints are specified, which helps capture complex patterns in the data.

Parameter Optimization: Hyperparameter tuning was performed using *Randomized Search Cross-Validation*. I optimized parameters such as the number of estimators, maximum depth, minimum samples split, minimum samples leaf, and maximum features. This process helps balance the bias-variance trade-off and improves the model's generalization ability. The scikit-learn library was used for implementation, leveraging its efficient handling of large datasets and parallel processing capabilities.

Runtime Estimate: Training the Random Forest Regressor took approximately **11 minutes and 42 seconds**.

3.3 XGBoost Regressor

Training Algorithm: XGBoost uses gradient boosting framework where new trees are added to correct the errors made by existing trees. It minimizes a regularized objective function that combines a convex loss function (e.g., mean squared error) with a regularization term that penalizes the complexity of the model to prevent overfitting. The training involves computing gradients and second-order gradients (Hessian) to guide the optimization process.

Parameter Optimization: I performed hyperparameter tuning using *Randomized Search Cross-Validation* to optimize parameters such as the number of estimators, learning rate, maximum depth, subsample ratio, column subsample ratio, regularization parameters (alpha and lambda), and minimum child Iight. The XGBoost library's scikit-learn API was used, which provides efficient implementations and supports parallel processing.

Runtime Estimate: Training the XGBoost Regressor took approximately **5 minutes and 17 seconds**.

3.4 CatBoost Regressor

Training Algorithm: CatBoost employs an ordered boosting process, which reduces overfitting caused by target leakage in traditional gradient boosting algorithms. It uses symmetric decision trees (oblivious trees), where the same splitting criterion is applied across all nodes at the same level, leading to faster inference and improved generalization. The algorithm also utilizes efficient encoding methods for categorical variables.

Parameter Optimization: Hyperparameters such as the number of iterations, learning rate, depth of the trees, L2 leaf regularization, bagging temperature, and border count (number of splits for numerical features) Ire optimized using *Randomized Search Cross-Validation.* The CatBoost library was used for implementation, benefiting from its optimized training procedures and automatic handling of categorical variables.

Runtime Estimate: Training the CatBoost Regressor took approximately **6.37 seconds**.

4 Hyperparameter Selection

For hyperparameter optimization, I employed **Randomized Search Cross-Validation** provided by the scikit-learn library. This method allows for efficient exploration of a wide range of hyperparameter values by sampling a fixed number of parameter settings from specified distributions. It is more computationally efficient than grid search, especially when dealing with a large number of hyperparameters and values.

I defined hyperparameter grids for each model based on domain knowledge and initial experimentation. The performance metric used for evaluation during cross-validation was the negative root mean squared error (RMSE), as our goal was to minimize the RMSE on the validation sets.

4.1 XGBoost Regressor

The **XGBoost Regressor** has several hyperparameters that significantly impact model performance. The key hyperparameters I tuned include:

- **Number of Estimators (`n_estimators`):** The number of trees in the ensemble.
- **Maximum Depth (`max_depth`):** The maximum depth of each tree.
- **Learning Rate (`learning_rate`):** The step size shrinkage used in updates to prevent overfitting.
- **Subsample Ratio (`subsample`):** The fraction of observations to be randomly sampled for each tree.
- **Column Subsample Ratio (`colsample_bytree`):** The fraction of columns to be randomly sampled for each tree.
- **Regularization Parameters (`reg_alpha`, `reg_lambda`):** L1 and L2 regularization terms on Iights.
- **Minimum Child Iight (`min_child_Iight`):** The minimum sum of instance Iight needed in a child.

Parameter Grid:

```
xgb_param_grid = {
    'n_estimators': [1500, 2000, 2500],
    'max_depth': [7, 9, 11],
    'learning_rate': [0.01, 0.005],
    'subsample': [0.8, 0.9, 1],
```

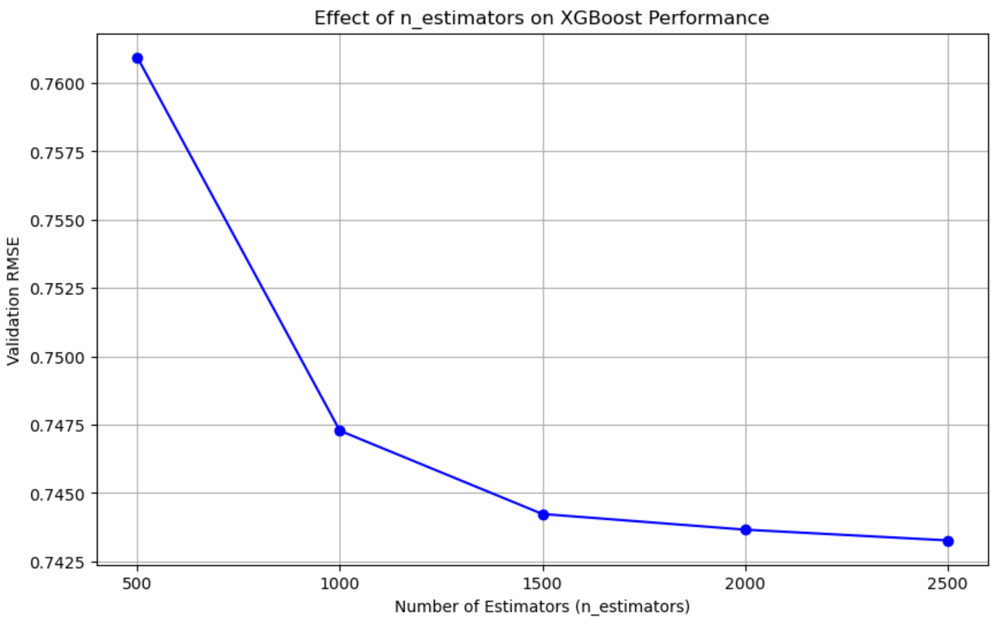


Figure 7: Effect of Number of Estimators on XGBoost Regressor Performance

```
    'colsample_bytree': [0.8, 0.9, 1],
    'gamma': [0.2],
    'reg_alpha': [0.1],
    'reg_lambda': [1.5, 2.0],
    'min_child_Iight': [7],
}
```

Effect of `n_estimators` on Model Performance:

To illustrate the impact of hyperparameter tuning, I analyzed how varying the number of estimators (`n_estimators`) affects the model's RMSE. Figure 7 shows the relationship betIen `n_estimators` and the cross-validated RMSE.

Analysis: As observed, increasing the number of estimators generally leads to a decrease in RMSE, indicating improved predictive performance.

4.2 Random Forest Regressor

For the **Random Forest Regressor**, I focused on tuning the following hyperparameters:

- **Number of Estimators (`n_estimators`):** Number of trees in the forest.
- **Maximum Depth (`max_depth`):** Maximum depth of the trees.

- **Maximum Features (`max_features`):** Number of features to consider when looking for the best split.

Parameter Grid:

```
rf_param_grid = {
    'n_estimators': [200, 500, 1000],
    'max_depth': [10, 20, 25],
    'max_features': ['sqrt', 'log2', 0.8],
    'min_samples_split': [2],
    'min_samples_leaf': [1],
}
```

Using `RandomizedSearchCV`, I sampled hyperparameter combinations and evaluated performance with 10-fold cross-validation. The optimal hyperparameters Ire those that minimized the validation RMSE.

4.3 CatBoost Regressor

For the **CatBoost Regressor**, hyperparameters tuned included:

- **Iterations (`iterations`):** Number of boosting iterations.
- **Learning Rate (`learning_rate`):** Step size shrinkage.
- **Depth (`depth`):** Depth of the trees.
- **L2 Leaf Regularization (`l2_leaf_reg`):** L2 regularization coefficient.
- **Bagging Temperature (`bagging_temperature`):** Controls the variance of the bagging.
- **Border Count (`border_count`):** Number of splits for numerical features.

Parameter Grid:

```
cat_param_grid = {
    'iterations': [500, 1000, 2000],
    'learning_rate': [0.01, 0.05, 0.1],
    'depth': [4, 6, 10],
    'l2_leaf_reg': [1, 3, 5],
    'bagging_temperature': [0, 1, 3],
    'border_count': [32, 64, 128],
}
```

I utilized CatBoost's efficient implementation of cross-validation to find the best hyperparameters, focusing on minimizing RMSE on the validation sets.

4.4 Extra Trees Regressor

For the **Extra Trees Regressor**, the primary hyperparameters tuned are:

- **Number of Estimators (`n_estimators`):** Number of trees in the forest.
- **Maximum Depth (`max_depth`):** Maximum depth of the trees.
- **Maximum Features (`max_features`):** Number of features to consider at each split.

Parameter Grid:

```
etr_param_grid = {
    'n_estimators': [100, 300],
    'max_depth': [None, 10, 20],
    'max_features': ['auto', 'sqrt'],
}
```

I applied `RandomizedSearchCV` to identify the optimal hyperparameters that minimize RMSE.

4.5 Results of Hyperparameter Tuning

The hyperparameter tuning process allowed us to find the optimal settings for each model, which significantly improved their performance on the validation sets. Table 1 summarizes the best hyperparameters found for each model.

Table 1: Best Hyperparameters for Each Model

Model	**Best Hyperparameters**
XGBoost Regressor	`{'n_estimators': 2500, 'max_depth': 9, 'learning_rate': 0.005, 'subsample': 0.9, 'colsample_bytree': 0.8, 'gamma': 0.2, 'reg_alpha': 0.1, 'reg_lambda': 1.5, 'min_child_Iight': 7}`
Random Forest Regressor	`{'n_estimators': 1000, 'max_depth': 25, 'max_features': 'sqrt', 'min_samples_split': 2, 'min_samples_leaf': 1}`
CatBoost Regressor	`{'iterations': 2000, 'learning_rate': 0.05, 'depth': 6, 'l2_leaf_reg': 3, 'bagging_temperature': 1, 'border_count': 64}`
Extra Trees Regressor	`{'n_estimators': 300, 'max_depth': None, 'max_features': 'sqrt'}`

5 Data Splits

For hyperparameter tuning and model evaluation, I used **K-Fold Cross-Validation** with shuffling to maintain randomness in the data distribution. Specifically, I utilized `KFold` with 10 splits:

```
from sklearn.model_selection import KFold

cv = KFold(n_splits=10, shuffle=True, random_state=42)
```

Explanation: The data was divided into 10 folds, and the model was trained and validated 10 times, each time using a different fold as the validation set and the remaining folds as the training set. The `shuffle` parameter was set to `True` to randomize the data before splitting, which helps in obtaining more reliable cross-validation results.

To mitigate the risk of overfitting, I implemented several strategies:

- **Cross-Validation:** Using K-Fold cross-validation provides a more robust estimate of model performance on unseen data and reduces the likelihood of overfitting to a particular train-test split.
- **Hyperparameter Tuning:** I systematically tuned hyperparameters using cross-validation, selecting the parameters that minimized validation error, which helps prevent overfitting to the training data.

- **Regularization:** Models like Ridge Regression include regularization terms that penalize overly complex models, thus reducing overfitting.
- **Feature Selection and Engineering:** I carefully selected and engineered features based on their relevance and avoided including highly correlated or redundant features that could lead to overfitting.
- **Randomization and Shuffling:** By shuffling the data before splitting, I ensured that each fold is representative of the overall dataset, which enhances the reliability of cross-validation results.

6 Reflection on Progress

The process of developing a predictive model for Airbnb listing prices presented several challenges and learning opportunities. One of the most significant hurdles was dealing with the complexity and messiness of the variables in the dataset. The sheer number of features and their varying types made it impractical to test every possible combination and their impact on the model. This complexity often led to a trial-and-error approach, which was time-consuming and sometimes unproductive.

Another major challenge was effectively preprocessing and extracting meaningful information from the text variables using natural language processing (NLP) techniques. Variables like *name*, *description*, *amenities*, and *reviews* contained unstructured text data that was often noisy and inconsistent. Identifying the best methods to clean, tokenize, and vectorize this data to capture relevant features without introducing bias or overfitting proved to be difficult. Despite experimenting with techniques like word counts, TF-IDF vectors, and topic modeling, integrating these features into the model did not yield significant improvements in predictive performance.

Furthermore, implementing the stacking regressor did not enhance the model's performance as much as anticipated. Combining multiple base models with a meta-model was expected to capture a wider range of patterns and interactions in the data. However, the stacked model's performance was only marginally better than that of the individual base models. This outcome suggests that there may be issues with the stacking implementation, such as data leakage, or that the base models were already capturing most of the predictive signal, leaving little room for improvement.

Time constraints also played a role in limiting the exploration of additional avenues that might have improved the model. For instance, deeper hyperparameter tuning, feature selection methods, or alternative algorithms like neural networks could have been investigated. Moreover, computational resources were a bottleneck when training complex models, particularly during cross-validation and hyperparameter optimization, which slowed down the iterative process of model development.

Overall, the hardest part of this competition was managing the balance between model complexity and practical constraints like time and computational power. It highlighted the importance of strategic feature engineering, efficient model selection, and thorough validation to build robust predictive models. Future work could focus on automating parts of the feature selection process, exploring advanced NLP techniques for text data, and optimizing computational efficiency to allow for more extensive experimentation.

7 Predictive Performance

Kaggle username: **zoe1trick**

I was getting 0.73 on cross-validation scores locally while getting 0.75 on the Kaggle leaderboard. This suggests that my model is still overfitting. Possible ways to solve this issue are to increase the regularization penalty or to implement early stopping in my training process.

- **Fully Interpretable Model (e.g., Decision Tree, Generalized Additive Models):** I have used random forest and extra trees in combination of gradient boosting algorithms. The decision trees are inherently more transparent and grant huge insights during feature selection. Details please see the following descriptions.
- **Interpretable Feature Engineering Pipeline:** see the section 1.2 that I created many interpretable features in feature engineering.
- **Feature Importance Analysis:** Based on the feature importance produced by Random Forest, I found ... [See the following section 8]

8 Feature Importance Analysis

8.1 Location Coordinates

- **Features:** longitude, latitude

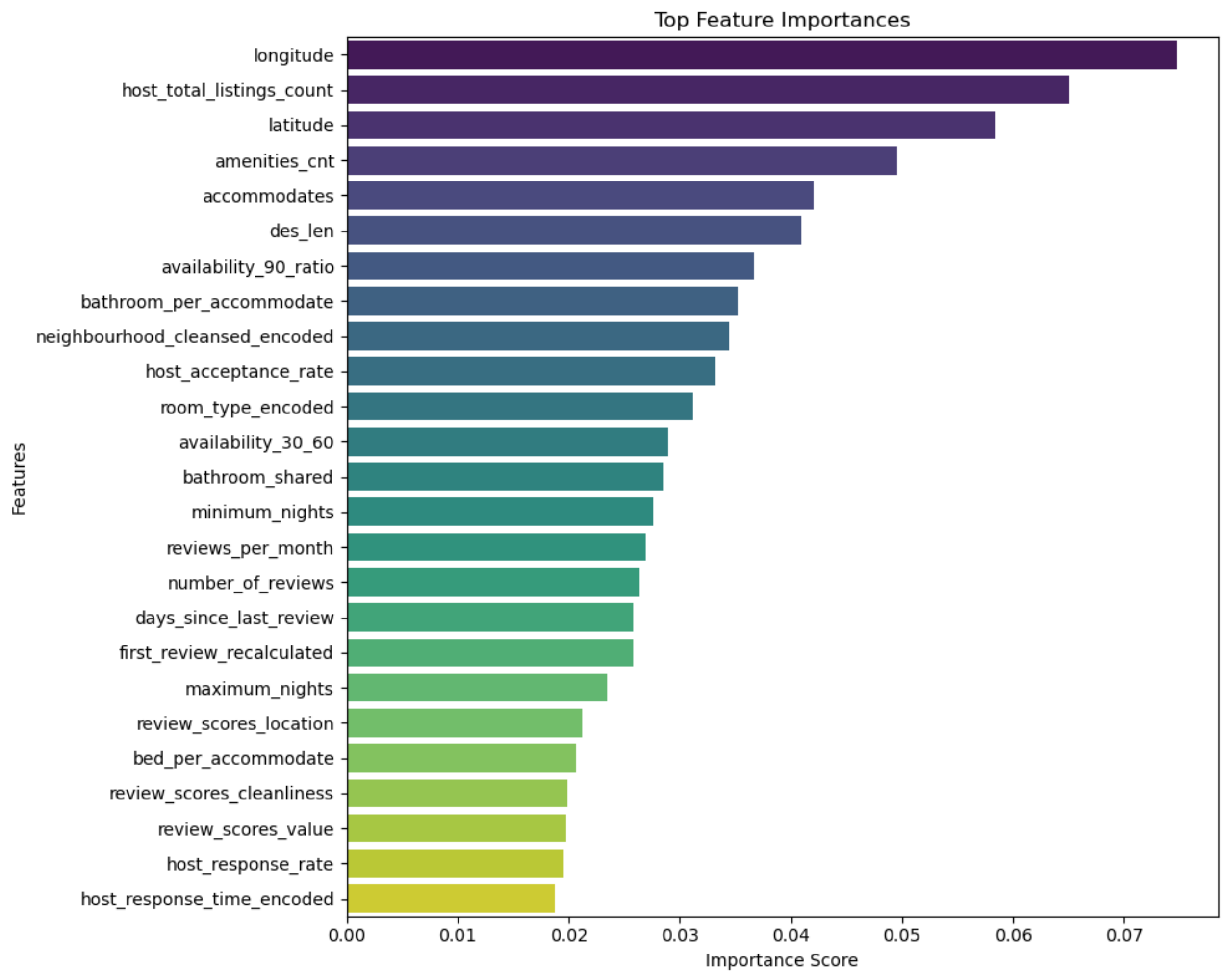


Figure 8: Feature Importance From Random Forest

- **Impact:** Captures proximity to landmarks, attractions, and overall neighborhood desirability
- **Price Effect:** Prime locations command higher prices

8.2 Host's Total Listings

- **Feature:** host_total_listings_count
- **Impact:** Reflects professionalism and potential economies of scale
- **Price Effect:** May lead to more competitive pricing

8.3 Number of Amenities

- **Feature:** amenities_cnt
- **Impact:** Indicates comfort and convenience level
- **Price Effect:** More amenities justify higher prices

8.4 Accommodation Capacity

- **Feature:** accommodates
- **Impact:** Reflects property size and guest capacity
- **Price Effect:** Larger properties generally have higher prices

8.5 Description Length

- **Feature:** des_len
- **Impact:** Indicates host effort in marketing the property
- **Price Effect:** Detailed descriptions may allow for higher pricing

8.6 Availability in 90 days

- **Feature:** availability_90_ratio
- **Impact:** Indicates the popularity of the listings in a 3-month period
- **Price Effect:** Popular housing might have economic pricing or other unique properties that led to hight price

9 Code

In [1]:
```python
import pandas as pd
import numpy as np
import math

import os
os.chdir("/Users/jiangeleanor/Duke/ECE 687D Algorithm ML/project
    ")

from ydata_profiling import ProfileReport

from sklearn.preprocessing import LabelEncoder
from sklearn.feature_extraction.text import TfidfVectorizer

import re

import matplotlib.pyplot as plt
import seaborn as sns

train = pd.read_csv("data/train.csv")
test = pd.read_csv("data/test.csv")

# a trick library to save time for EDA
profile = ProfileReport(train, title="profile_report")
profile.to_file("profile_report.pdf")

# amenities
## clean
import ast
freq_words = ['you', 'on', 'and', 'the', 'or', 'n', 'for', 'with
    ', 'in', 'at']
```

In [2]:

```
def clean_amenities(amenities_str):
    try:
        amenities_lst = ast.literal_eval(amenities_str)
    except:
        amenities_lst = []
    amenities_clean = []
    for amenity in amenities_lst:
        # remove special characters and extra spaces
        # e.g., Wi-Fi vs. wifi
        amenity_clean = re.sub(r'[^A-Za-z0-9\s]', '', amenity.
           lower().strip())
        # Remove freq_words using regex
        pattern = r'\b(?:' + '|'.join(re.escape(word) for word
           in freq_words) + r')\b'
        amenity_clean = re.sub(pattern, '', amenity_clean)
        # turn air conditioning to ac
        amenity_clean = amenity_clean.replace('air conditioning'
           , 'ac')
        # Remove extra spaces
        amenity_clean = re.sub(r'\s+', ' ', amenity_clean).strip
           ()
        amenities_clean.append(amenity_clean)
    return amenities_clean

train['amenities_clean'] = train['amenities'].apply(
    clean_amenities)
test['amenities_clean'] = test['amenities'].apply(
    clean_amenities)
print(train['amenities_clean'])

# amenities count
train['amenities_cnt'] = train['amenities_clean'].apply(lambda x
    : len(x))
test['amenities_cnt'] = test['amenities_clean'].apply(lambda x:
    len(x))

plt.figure(figsize=(10, 6))
sns.boxplot(x='price', y='amenities_cnt', data=train)
plt.title('Price Distribution for Number of Amenities')
plt.xlabel('Price')
plt.ylabel('Number of Amenities')
plt.show()
```

In [3]:
```
from sklearn.feature_extraction.text import CountVectorizer

# Join amenities into a single string per listing
train['amenities_str'] = train['amenities_clean'].apply(lambda x
    : ' '.join(x))
test['amenities_str'] = test['amenities_clean'].apply(lambda x:
    ' '.join(x))

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import NMF
from sklearn.decomposition import LatentDirichletAllocation
```

In [4]:
```
## Use TF-IDF features
tfidf_vectorizer = TfidfVectorizer()
tfidf_train = tfidf_vectorizer.fit_transform(train['
    amenities_str'])
tfidf_test = tfidf_vectorizer.transform(test['amenities_str'])

# Use count features
vectorizer = CountVectorizer()
dtm = vectorizer.fit_transform(train['amenities_str'])
# filter out the count <= 30
dtm_df = pd.DataFrame(dtm.toarray(), columns=vectorizer.
    get_feature_names_out())
mask = ~dtm_df.columns.str.match(r'^\d+$')
dtm_df = dtm_df.loc[:, mask]
amenities_sum = dtm_df.sum(axis=0)
dtm_df = dtm_df.loc[:, amenities_sum[amenities_sum>30].index.
    tolist()]
filtered_amenities = dtm_df.columns.tolist()

# convert to df
tfidf_train_df = pd.DataFrame(tfidf_train.toarray(), columns=
    tfidf_vectorizer.get_feature_names_out()).reindex(columns=
    filtered_amenities).fillna(0)

tfidf_test_df = pd.DataFrame(tfidf_test.toarray(), columns=
    tfidf_vectorizer.get_feature_names_out()).reindex(columns=
    filtered_amenities).fillna(0)

n_topics = 10
topic_model = NMF(n_components=n_topics, random_state=42)
# topic_model = LatentDirichletAllocation(n_components=n_topics,
     random_state=42)

amentities_topics_train = topic_model.fit_transform(
    tfidf_train_df)
amenities_topics_train_df = pd.DataFrame(amentities_topics_train
    , columns=[f'amenity_topic_{i}' for i in range(n_topics)])
amentities_topics_test = topic_model.transform(tfidf_test_df)
amenities_topics_test_df = pd.DataFrame(amentities_topics_test,
    columns=[f'amenity_topic_{i}' for i in range(n_topics)])
```

In [5]:
```
## Display the topics
def display_topics(model, feature_names, num_top_words):
    for topic_idx, topic in enumerate(model.components_):
        message = f"Topic {topic_idx}: "
        message += ", ".join([feature_names[i] for i in topic.
           argsort()[:-num_top_words - 1:-1]])
        print(message)
display_topics(topic_model, filtered_amenities, n_topics)

def clean_description(des):

    pattern = r'[-&]|\b(and|or)\b'

    if des is np.nan:
        des = ''
    else:
        des = re.sub(pattern, '', des, flags=re.IGNORECASE).
           lower().strip()

    return des

train['description_clean'] = train['description'].apply(
   clean_description)
test['description_clean'] = test['description'].apply(
   clean_description)

train['des_len'] = train['description_clean'].apply(lambda x:
   len(x.split(' ')))
test['des_len'] = test['description_clean'].apply(lambda x: len(
   x.split(' ')))

# has_review
train['has_review'] = train['reviews'].notna()
test['has_review'] = test['reviews'].notna()

from sklearn.preprocessing import OrdinalEncoder

# label encoder neighbourhood_cleansed (217 distinct values)
oe_neighbourhood = OrdinalEncoder(handle_unknown='
   use_encoded_value', unknown_value=-1)
oe_neighbourhood.fit(train[['neighbourhood_cleansed']])

# Transform both training and test data
```

In [6]:
```
train['neighbourhood_cleansed_encoded'] = oe_neighbourhood.
    transform(train[['neighbourhood_cleansed']])
test['neighbourhood_cleansed_encoded'] = oe_neighbourhood.
    transform(test[['neighbourhood_cleansed']])

# label encoder neighbourhood_group_cleansed (* distinct values)
le_group = LabelEncoder()
train['neighbourhood_group_cleansed_encoded'] = le_group.
    fit_transform(train['neighbourhood_group_cleansed'])
train.drop(['neighbourhood_cleansed', '
    neighbourhood_group_cleansed'], axis=1, inplace=True)

test['neighbourhood_group_cleansed_encoded'] = le_group.
    transform(test['neighbourhood_group_cleansed'])
test.drop(['neighbourhood_cleansed', '
    neighbourhood_group_cleansed'], axis=1, inplace=True)

# room_type
enc = LabelEncoder()
train['room_type_encoded'] = enc.fit_transform(train['room_type'
    ])
test['room_type_encoded'] = enc.transform(test['room_type'])

train.drop('room_type', axis=1, inplace=True)
test.drop('room_type', axis=1, inplace=True)
```

In [7]:
```
## property_type from 59 to 14
def organize_property_type(prop):

    prop = prop.lower()
    # rental
    if 'rental' in prop:
        return 'rental'
    # townhouse, bungalow
    if ('townhouse' in prop) or ('cottage' in prop):
        return 'townhouse'
    if ('bungalow' in prop) or ('villa' in prop) or ('vacation'
       in prop):
        return 'villa'
    if 'service' in prop:
        return 'serviced'
    # condo
    if 'condo' in prop:
        return 'condo'
    if 'loft' in prop:
        return 'loft'
    if 'hostel' in prop:
        return 'hostel'
    if ('homestay' in prop) or ('breakfast' in prop) or ('casa'
       in prop):
        return 'homestay'
    if 'tiny' in prop:
        return 'tiny'
    if 'guest' in prop:
        return 'guesthouse'
    if 'camper' in prop:
        return 'camper'
    if ('home' in prop) or ('entire place' in prop):
        return 'home'
    if ('hotel' in prop) or ('resort' in prop):
        return 'hotel'
    else:
        return 'others'
train['property_type_org'] = train['property_type'].apply(
   organize_property_type)
test['property_type_org'] = test['property_type'].apply(
   organize_property_type)

enc = OrdinalEncoder(handle_unknown='use_encoded_value',
   unknown_value=-1)
```

In [8]:
```
train['property_type_encoded'] = enc.fit_transform(train[['
    property_type_org']])
test['property_type_encoded'] = enc.transform(test[['
    property_type_org']])
train['host_response_time'] = train['host_response_time'].fillna
    ('unknown')
test['host_response_time'] = test['host_response_time'].fillna('
    unknown')

enc = LabelEncoder()
train['host_response_time_encoded'] = enc.fit_transform(train['
    host_response_time'])
test['host_response_time_encoded'] = enc.transform(test['
    host_response_time'])

# extract if the bathroom is shared
train['bathroom_shared'] = train['bathrooms_text'].apply(lambda
    x: 1 if (isinstance(x, str) and re.search('shared', x)) or (x
     == 0) else 0)
test['bathroom_shared'] = test['bathrooms_text'].apply(lambda x:
     1 if (isinstance(x, str) and re.search('shared', x)) or (x
    == 0) else 0)
train.drop('bathrooms_text', axis=1, inplace=True)
test.drop('bathrooms_text', axis=1, inplace=True)

# How many years the host has been hosting from 2024
train['host_years'] = train['host_since'].apply(lambda x: 2024-
    int(x.split('-')[0]))
test['host_years'] = test['host_since'].apply(lambda x: 2024-int
    (x.split('-')[0]))

# from 2008 since aribnb founded, 2008 = year 0, when is the
    first_review and last_review
# fill NA with -1
train['first_review_recalculated'] = train['first_review'].apply
    (lambda x: int(x.split('-')[0]) - 2008 + int(x.split('-')[1])
     / 12 if isinstance(x, str) else np.nan)
train['last_review_recalculated'] = train['last_review'].apply(
    lambda x: int(x.split('-')[0]) - 2008 + int(x.split('-')[1])
    / 12 if isinstance(x, str) else np.nan)
```

In [9]:
```
test['first_review_recalculated'] = test['first_review'].apply(
    lambda x: int(x.split('-')[0]) - 2008 + int(x.split('-')[1])
    / 12 if isinstance(x, str) else np.nan)
test['last_review_recalculated'] = test['last_review'].apply(
    lambda x: int(x.split('-')[0]) - 2008 + int(x.split('-')[1])
    / 12 if isinstance(x, str) else np.nan)

train['bathrooms'] = train['bathrooms'].fillna(0)
test['bathrooms'] = test['bathrooms'].fillna(0)

train['bathroom_per_accommodate'] = train['bathrooms'] / train['
    accommodates']
test['bathroom_per_accommodate'] = test['bathrooms'] / test['
    accommodates']

train['beds'] = train['beds'].fillna(0)
test['beds'] = test['beds'].fillna(0)

train['bed_per_accommodate'] = train['beds'] / train['
    accommodates']
test['bed_per_accommodate'] = test['beds'] / test['accommodates'
    ]

train['availability_90_ratio'] = train['availability_90']/90
test['availability_90_ratio'] = test['availability_90']/90

train['availability_30_60'] = train.apply(lambda row: row['
    availability_30'] / row['availability_60'] if row['
    availability_60']!=0 else 0, axis=1)
test['availability_30_60'] = test.apply(lambda row: row['
    availability_30'] / row['availability_60'] if row['
    availability_60']!=0 else 0, axis=1)

# ---
# figure out how long the listing might be empty
from datetime import datetime
cur_date = pd.to_datetime('2024-11-01')
train['days_since_last_review'] = (cur_date - pd.to_datetime(
    train['last_review'])).dt.days
test['days_since_last_review'] = (cur_date - pd.to_datetime(test
    ['last_review'])).dt.days
```

In [10]:

```
## ---
fill_zero_feat = ['host_is_superhost', 'beds', 'bathrooms', '
    number_of_reviews_ltm']
for feat in fill_zero_feat:
    train[feat] = train[feat].fillna(0)
    test[feat] = test[feat].fillna(0)

# after filling with int 0, the original boolean value would
    result value error
# convert all to numeric
train['host_is_superhost'] = train['host_is_superhost'].apply(
    int)
test['host_is_superhost'] = test['host_is_superhost'].apply(int)

# ---
fill_mean_feat = ['host_response_rate', 'host_acceptance_rate',
    'review_scores_rating', 'review_scores_accuracy',
                  'review_scores_cleanliness', '
                      review_scores_checkin', '
                      review_scores_communication', '
                      days_since_last_review',
                  'review_scores_location', 'review_scores_value
                      ', 'reviews_per_month', '
                      first_review_recalculated', '
                      last_review_recalculated']
for feat in fill_mean_feat:
    mean_val = train[feat].mean()
    train[feat] = train[feat].fillna(mean_val)
    test[feat] = test[feat].fillna(mean_val)

# ---
# dummy indicates if have reviews in the past year
train['has_reviews_ltm'] = train['number_of_reviews_ltm'].apply(
    lambda x: 1 if x!=0 else 0)
test['has_reviews_ltm'] = test['number_of_reviews_ltm'].apply(
    lambda x: 1 if x!=0 else 0)

# how many reviews per month in the recent year
train['reviews_per_month_ltm'] = train['number_of_reviews_ltm']
    / 12
test['reviews_per_month_ltm'] = test['number_of_reviews_ltm'] /
    12
```

In [11]:

```
features = ['latitude',
   'longitude', 'host_response_rate',
   'host_acceptance_rate', 'host_is_superhost',
   'host_total_listings_count',
   'availability_90_ratio', 'instant_bookable',
   'minimum_nights', 'maximum_nights', 'number_of_reviews',
   'number_of_reviews_ltm', 'review_scores_rating', '
      review_scores_accuracy',
   'review_scores_cleanliness', 'review_scores_checkin',
   'review_scores_communication', 'review_scores_location',
   'review_scores_value', 'reviews_per_month', 'amenities_cnt',
      'has_review',
   'neighbourhood_cleansed_encoded',
   'neighbourhood_group_cleansed_encoded', 'room_type_encoded',
   'bathroom_shared',
   # 'Kitchen Supplies',
   # 'House Infrastructure', 'Entertainment', 'Safety Features',
   # 'Outdoor Amenities', 'Luxury Amenities', 'Family-Friendly
      Amenities',
   'days_since_last_review', 'has_reviews_ltm', 'des_len',
   'bathroom_per_accommodate', 'bed_per_accommodate',
   'property_type_encoded', 'host_response_time_encoded', '
      accommodates',
   'reviews_per_month_ltm', 'first_review_recalculated', '
      last_review_recalculated',
   'availability_30_60']
from sklearn.feature_selection import VarianceThreshold

selector = VarianceThreshold(threshold=0.01)
selector.fit(train[features])
selected_features1 = [features[i] for i in range(len(features))
   if selector.get_support()[i]]
selected_features1
selected_features1 += ['latitude', 'longitude']
```

In [12]:

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier()
X, y = train[selected_features1], train['price']
rf.fit(X, y)

# Get feature importances
importances = pd.Series(rf.feature_importances_, index=
    selected_features1)
importances = importances.sort_values(ascending=False)

# Select top N features
N = 25
selected_features2 = importances.head(N).index.tolist()
print("Selected Features:", selected_features2)

import matplotlib.pyplot as plt
import seaborn as sns

# Set the size of the plot
plt.figure(figsize=(10, 8))

# Plot the top N features
sns.barplot(x=importances.head(N), y=importances.head(N).index,
    palette='viridis')

plt.title('Top Feature Importances')
plt.xlabel('Importance Score')
plt.ylabel('Features')
plt.tight_layout()
plt.show()

import pandas as pd
import numpy as np

# Models
import xgboost as xgb
from xgboost import XGBRegressor
import lightgbm as lgb
from lightgbm import LGBMRegressor
from sklearn.ensemble import RandomForestRegressor,
    StackingRegressor
```

In [13]:

```
from sklearn.model_selection import train_test_split,
   RandomizedSearchCV, cross_val_score, KFold
from sklearn.metrics import root_mean_squared_error, make_scorer
import tensorflow as tf

# Suppress warnings
import warnings
warnings.filterwarnings('ignore')

X_train_full = train[selected_features2]
y_train_full = train['price']

# Define RMSE scorer (negative because lower is better)
rmse_scorer = make_scorer(root_mean_squared_error,
   greater_is_better=False)

cv = KFold(n_splits=10, shuffle=True, random_state=42)

# Hyperparameter grid for XGBoost
xgb_param_grid = {
    'n_estimators': [1500, 2000, 2500],
    'max_depth': [7, 9, 11],
    'learning_rate': [0.01, 0.005],
    'subsample': [0.8, 0.9, 1],
    'colsample_bytree': [0.8, 0.9, 1],
    'gamma': [0.2],
    'reg_alpha': [0.1],
    'reg_lambda': [1.5, 2.0],
    'min_child_weight': [7],
}

xgb_reg = XGBRegressor(
    objective='reg:squarederror',
    random_state=42
)

# RandomizedSearchCV for XGBoost
xgb_random_search = RandomizedSearchCV(
    estimator=xgb_reg,
    param_distributions=xgb_param_grid,
    n_iter=20,
    scoring=rmse_scorer,
    cv=cv,
    verbose=1,
    random_state=42,
    n_jobs=-1
)
```

In [14]:

```
## Perform hyperparameter tuning
xgb_random_search.fit(X_train_full, y_train_full)

# Best hyperparameters
xgb_best_params = xgb_random_search.best_params_
print("Best XGBoost Hyperparameters:")
print(xgb_best_params)
best_xgb_reg = XGBRegressor(
    objective='reg:squarederror',
    random_state=42,
    **xgb_best_params
)

# Hyperparameter grid for Random Forest
rf_param_grid = {
    'n_estimators': [200, 500, 1000],
    'max_depth': [10, 20, 25],
    'max_features': ['sqrt', 'log2', 0.8],
    'min_samples_split': [2],
    'min_samples_leaf': [1],
}

rf_reg = RandomForestRegressor(random_state=42)

# RandomizedSearchCV for Random Forest
rf_random_search = RandomizedSearchCV(
    estimator=rf_reg,
    param_distributions=rf_param_grid,
    n_iter=20,
    scoring=rmse_scorer,
    cv=cv,
    verbose=1,
    random_state=42,
    n_jobs=-1
)

# Perform hyperparameter tuning
rf_random_search.fit(X_train_full, y_train_full)

# Best hyperparameters
rf_best_params = rf_random_search.best_params_
print("Best Random Forest Hyperparameters:")
print(rf_best_params)
```

In [15]:

```
best_rf_reg = RandomForestRegressor(
    random_state=42,
    **rf_best_params
)

from catboost import CatBoostRegressor

cat_param_grid = {
    'iterations': [500, 1000, 2000],
    'learning_rate': [0.01, 0.05, 0.1],
    'depth': [4, 6, 10],
    'l2_leaf_reg': [1, 3, 5],
    'bagging_temperature': [0, 1, 3],
    'border_count': [32, 64, 128],
}

cat_reg = CatBoostRegressor(random_state=42)

# RandomizedSearchCV for Random Forest
cat_random_search = RandomizedSearchCV(
    estimator=cat_reg,
    param_distributions=cat_param_grid,
    n_iter=20,
    scoring=rmse_scorer,
    cv=cv,
    verbose=1,
    random_state=42,
    n_jobs=-1
)

# Perform hyperparameter tuning
cat_random_search.fit(X_train_full, y_train_full)

# Best hyperparameters
cat_best_params = cat_random_search.best_params_
print("Best CatBoost Hyperparameters:")
print(cat_best_params)
best_cat_reg = CatBoostRegressor(
    random_state=42,
    **cat_best_params
)
```

In [16]:

```
from sklearn.ensemble import ExtraTreesRegressor
etr_param_grid = {
    'n_estimators': [100, 300],
    'max_depth': [None, 10, 20],
    'max_features': ['auto', 'sqrt'],
}

etr_reg = ExtraTreesRegressor(random_state=42)

# RandomizedSearchCV for Random Forest
etr_random_search = RandomizedSearchCV(
    estimator=etr_reg,
    param_distributions=etr_param_grid,
    n_iter=20,
    scoring=rmse_scorer,
    cv=cv,
    verbose=1,
    random_state=42,
    n_jobs=-1
)

# Perform hyperparameter tuning
etr_random_search.fit(X_train_full, y_train_full)

# Best hyperparameters
etr_best_params = etr_random_search.best_params_
print("Best CatBoost Hyperparameters:")
print(etr_best_params)
best_etr_reg = ExtraTreesRegressor(
    random_state=42,
    **etr_best_params
)
# Initialize the Extra Trees Regressor with the predefined
   parameters
best_etr_reg = ExtraTreesRegressor(**etr_best_params)
```

In [17]:
```
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

estimators = [
    ('xgb', best_xgb_reg),
    ('cat', best_cat_reg),
    # ('lgb', best_lgb_reg),
    ('rf', best_rf_reg),
    ('etr', best_etr_reg)
]

# arbitrarily choose alpha=1.0 for metal model Ridge
stacking_reg = StackingRegressor(
    estimators=estimators,
    final_estimator=Ridge(alpha=1.0),
    cv=cv,
    n_jobs=-1
)

# evaluate locally on cross-validation score
rd_cv_scores = cross_val_score(
    stacking_reg,
    X_train_full,
    y_train_full,
    cv=cv,
    scoring=rmse_scorer,
    n_jobs=-1
)

rd_cv_rmse_scores = -rd_cv_scores
print("Ridge Cross-validated RMSE scores:", rd_cv_rmse_scores)
print("Ridge Mean CV RMSE:", rd_cv_rmse_scores.mean())

# Fit the stacking regressor
stacking_reg.fit(X_train_full, y_train_full)
```

10 Logistics

The report must be submitted to Gradescope by November 26th. Good luck!