link to kaggle competition: https://www.kaggle.com/competitions/cs-671-fall-2024-final-project#

1 Exploratory Analysis:

How did you make sense of the provided dataset and get an idea of what might work? Did you use histograms, scatter plots or some sort of clustering algorithm? Did you do any feature engineering? Describe your thought process in detail for how you approached the problem and if you did any feature engineering in order to get the most out of the data?

EDA: The first step is definitely to see the distribution of the target variable, this will guide us to pick the algorithm and data processing steps
a) Distribution of target variable:

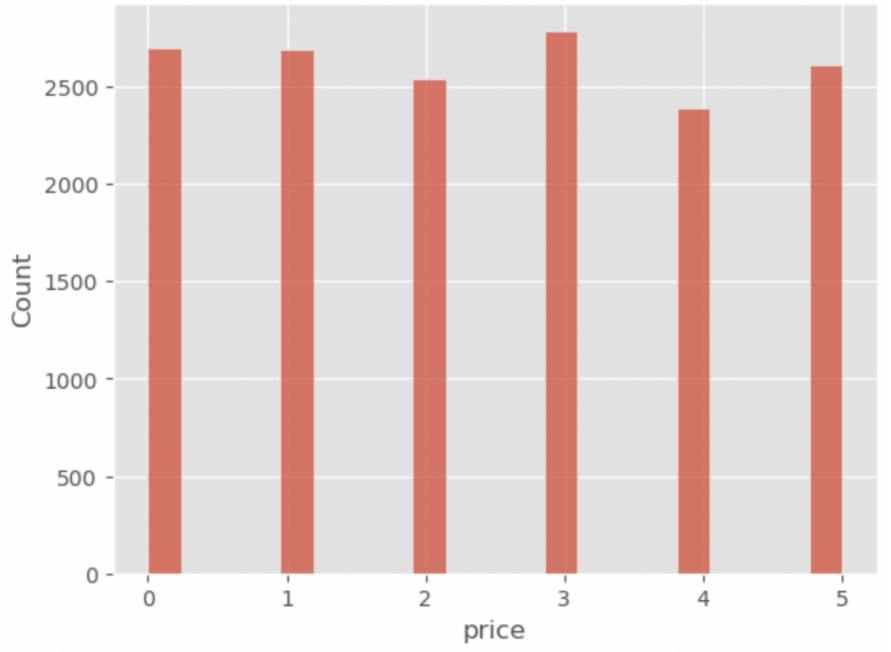

Fig 1: Distribution of price in the training set.
Very fortunately, the class seems to be relatively balanced. This means that the class imbalance might not be a serious issue we should pay much attention to.

b) Distribution of key predictor variables:
I then go to kaggle to see the distribution of some predictor variables that i think would be relevant.

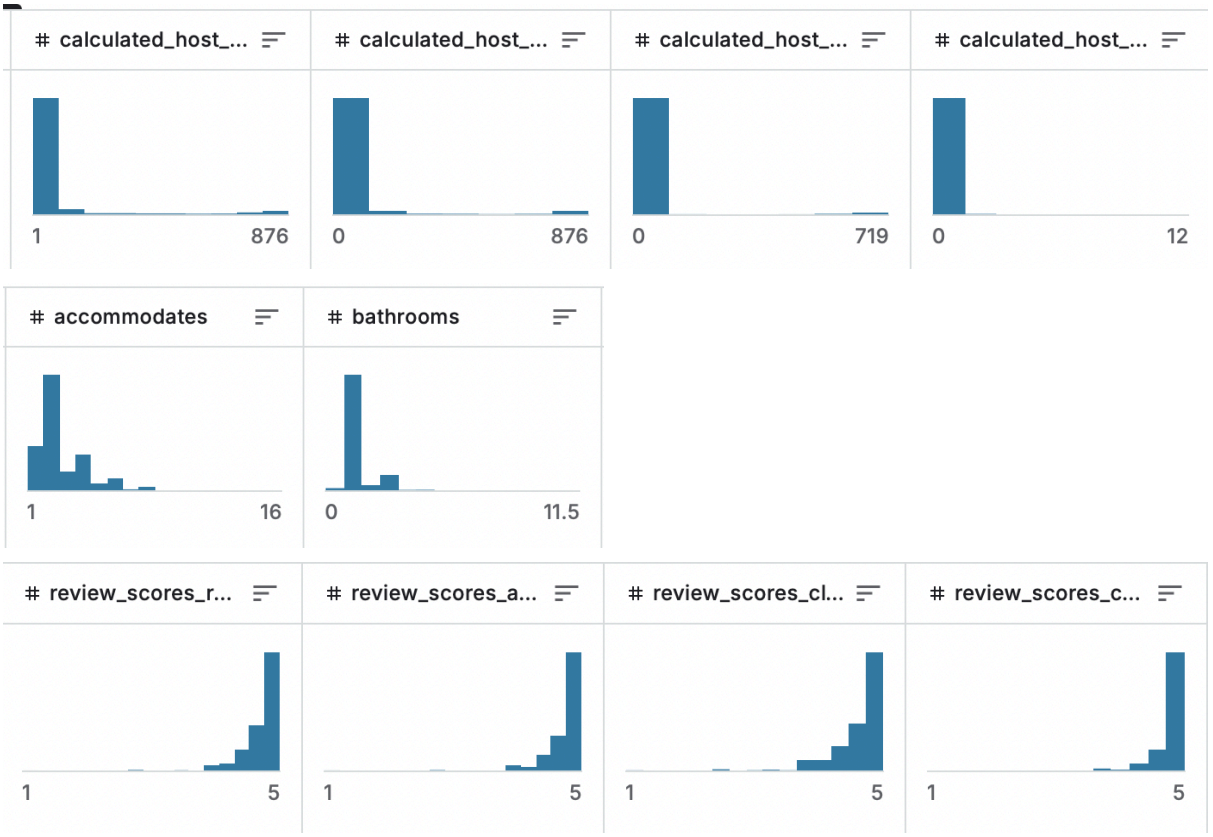

Fig 2 Kaggle provided distribution of numerical predictors

From the kaggle provided visualization of the key predictor, we can see that most predictors follow a non-normal distribution. This suggests that we should consider log normalization, or choose a model that is less sensitive to data skewness during the model training phase.

c) Explore predictor data: what are some features that provide strong information gain for each price level? Engineer corresponding features, the following is a bit of my ideas:

Feature engineering:
1): Dummy for grouped districts: see the price distribution of grouped districts.
Through grouping, it seems like houses in Manhattan are a bit more expensive, on average, than other districts, thus I engineered a dummy variable indicator with grouped_neighbourhood the housing/property is located at.

2) Housing categories vs price: so engineer dummies for housing categories
I engineered a plot showing the housing category's count distribution against different pricing categories. It will be very difficult to explain the difference in trends through human eyes, but the distribution is not unanimous across all pricing levels. Therefore, a series of dummy variables indicating housing/ property category is used as predictors for additional information.

3) amenities: distribution power law, what are some amenities that could indicate price levels?
For this, I intentionally picked the amenities that could indicate price levels.

	count
Smoke alarm	21161
Wifi	20567
Carbon monoxide alarm	19426
Kitchen	19190
Hot water	17645
Essentials	16925
Hangers	16425
Hair dryer	15092
Iron	15005
Refrigerator	14705
Dishes and silverware	14574
Air conditioning	14511
Bed linens	14217

For example, a property with a swimming pool, gym, and elevator could potentially indicate higher price.

Dummy variables are created as followed:

```
df['amenities_list'] = df['amenities'].str.replace(r'[\[\]"\\]', '', regex=True).str.split(', ')

# Define important amenities
important_amenities = [
    "Dishwasher", "Elevator", "TV", "Wifi", "Hair dryer", "Washer", "Dryer", "Extra pillows and blankets", "Long term stays allowed", "Private patio or balcony
    "Exterior security cameras on property", "Carbon monoxide alarm", "Patio or balcony", "Private entrance", "Smart lock", "Pets allowed",
    "Exterior security cameras", "Game console", "Cleaning available during stay", "Gym", "Free parking on premises", "Lock on bedroom door",
    "Backyard", "City skyline view", "Indoor fireplace", "Courtyard view", "Smoking allowed", "Breakfast", "Shared gym in building", "Central heating", "Host g
]

# Create binary flags for important amenities
for amenity in important_amenities:
    df[f'has_{amenity.replace(" ", "_").lower()}'] = df['amenities_list'].apply(lambda x: 1 if amenity in x else 0)
```

Fig3 Amenities

4 Luxury or not? What are some keywords that correspond to high/ low prices?

```
# A larger word bank for luxury detection
luxury_keywords = [
    "luxury", "premium", "exclusive", "five-star", "high-end",
    "spa", "pool", "sauna",  "villa", "mansion", "ocean view", "infinity pool",
    "breathtaking", "opulent", "pristine", "majestic", "penthouse", "rooftop terrace",
    "designer", "custom-built", "panoramic view", "beachfront",
    "resort", "boutique", "chalet", "royal", "imperial", "modern",
    "skyline", "blueground", "elevator", "suite", "hotel", "service", "downtown", "lounge"
]

import re

# Function to check if any luxury keyword is in the description
def is_luxury(description):
    if isinstance(description, str):
        # Combine keywords into a single regex pattern
        pattern = r'\b(' + '|'.join(luxury_keywords) + r')\b'
        if re.search(pattern, description.lower()):  # Case insensitive match
            return 1  # Luxury
    return 0  # Not luxury

df['is_luxury'] = df['description'].apply(is_luxury)
df['is_luxury'].value_counts()
```

```
keywords_price_0 = [
    "shared bathroom", "shared kitchen", "affordable", "budget-friendly",
    "cozy", "simple", "small", "basic furnishings", "short-term stay",
    "students", "travel nurses", "walk to train", "near hospital",
    "no frills", "peaceful", "safe", "clean", "budget",
    "furnished room", "quiet neighborhood", "mini fridge", "laundry nearby", "shared room",
    "street parking", "no smoking", "limited amenities", "basement", "shared bathroom"
]

import re

# Function to check if any luxury keyword is in the description
def is_not_luxury(description):
    if isinstance(description, str):
        # Combine keywords into a single regex pattern
        pattern = r'\b(' + '|'.join(keywords_price_0) + r')\b'
        if re.search(pattern, description.lower()):  # Case insensitive match
            return 1  # Luxury
    return 0  # Not luxury

df['is_not_luxury'] = df['description'].apply(is_not_luxury)
df['is_not_luxury'].value_counts()
```

is_not_luxury	count
0	14580
1	7843

price	count
0	1416
1	1352
2	1047
3	945
4	689
5	504

Fig4: Luxuries Dealing

I selected a bank of keywords using tf-idf, which is later explained in section 6, for possible indicators of high price accommodation and low price accommodations. The first subplot of this section indicates that using a sub portion of keyword that indicates lower price levels do result in a better identification of lower priced airbnbs, and using keyword that indicates higher pricing achieved similar effects. Thus, the binary indicator of whether is luxury is used as a predictor due to the information gain it brought.

5) Review sentiment Score (Reviews distribution and review score distribution. Why take sentiment analysis for reviews?)

This sentiment score measures the average sentiment of the top 5 reviews for a product, computed using the Hugging Face pipeline with the distilbert-base-uncased-finetuned-sst-2-english model for sentiment analysis. The reviews are processed in batches with tqdm providing a progress bar, and the score reflects a positive or negative sentiment based on the model's predictions.

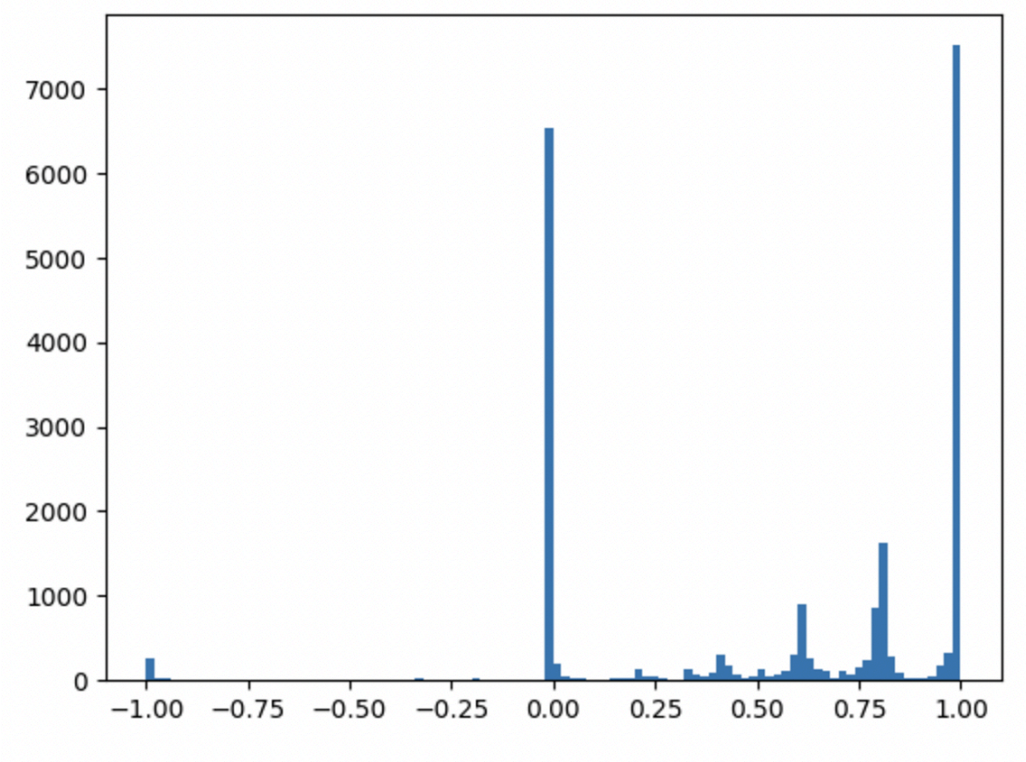

Fig 5: Review sentiment score of top 10 (if exists) reviews of each airbnb.
We clearly see that the distribution of the review sentiments are multi-model, which contributes to information gain. Therefore, I engineered a sentimental score as a predictive feature.

6) Price expensiveness of neighbourhoods:
It is apparent from the analysis of data that certain neighborhoods are more expensive than others, for example, neighbourhoods in east village compared to neighborhoods in queens. Thus, average price expensiveness of each neighbourhood is computed(without train test leakage) as an additional predictor variable.

7) Ratios: bedroom per bathroom, beds per accommodation.
Exploratory analysis shows that the relationship between these variables and price is not uniform. For example, it would not be unreasonable to expect that a higher bedroom per bathroom might have a lower price. Some information could be gained from this feature and is thus engineered.

d) Missing Value Analysis:
We need to impute missing values. Data analysis on the training data suggests that the missing values, particularly for reviews and reviews score, is not heavily skewed toward one specific price level. And since the distribution is not normal, I used a median to impute missing values.

2: Models (reference see section 7, citation)

You are required to use at least two different algorithms for generating predictions (although you can choose the best one as your Kaggle submission). These do not need to be algorithms we used in class. It would not be acceptable to use the same algorithm but with two different parameters or kernels. In this section, you will explain your reasoning behind the choice of algorithms. Specific motivations for choosing a certain algorithm may include computational efficiency, simple parameterization, ease of use, ease of training, or the availability of high-quality libraries online, among many other possible factors. If external libraries were used, describe them and identify the source or authors of the code (make sure to cite all references and figures that you use if someone else designed them). Try to be adventurous!

a) First I choosed a single XGBoost Model, a tree boosting model:

a(i) Rationale for choosing XGBoost, or especially tree based models:
Non-Normal Feature Distribution: Many numerical feature values (for example, review scores, bedrooms, accommodates, and engineered features I have: bathroom to bedroom ratios, review sentiment score…) do not follow a normal distribution. Traditional linear models assume normality and linear relationships, which may not capture the underlying patterns effectively. Tree-based models, like those used in XGBoost, are adept at handling non-normal and skewed distributions. So based on the distribution of predictor variables, a tree based model is suitable.

Non-Linear Relationships: Correlation analysis revealed that the relationship between the predictors and the target variable (price) is likely non-linear. XGBoost excels at modeling complex, non-linear interactions between features without the need for explicit transformation or lots of feature engineering.

Automatic handling or multicollinearity: Engineered features naturally shared high collinearity with the original columns when lightly processed. Thus, the loss function of XGboost helps to handle the potential multicollinearity problem.

b) I finally choosed to combine multiple models using ensembling, introducing other diverse algorithms CatBoost, Random forests, and LightGBM.

b(i) Rationale for ensemble models, with majority voting:
Variance Reduction: By averaging the predictions of different models, ensemble methods reduce the variance associated with individual models. This leads to more stable and reliable predictions.
Bias Mitigation: Combining models that make different assumptions can offset the biases inherent in individual models, leading to better overall performance.
Improved Generalization: Ensembles are less likely to overfit the training data, which enhances their ability to generalize to unseen data.

b(ii) Why are Catboost, LightGBM, and Random forest as submodels in addition to XGboost in the ensemble model?

CatBoost is particularly effective with datasets that contain categorical features (which is later explained in section 3). For random forest, as an ensemble of decision trees, Random Forests are robust to overfitting and can handle a large number of features. They are excellent at capturing non-linear relationships and interactions between variables, which is valuable given the complexity of the housing price prediction task. For lightGBM, it is a gradient boosting framework known for its efficiency and speed (which means more, faster hyperparameter tuning would be feasible).

3: Training

Here, for each of the algorithms used, briefly describe (5-6 sentences) the training algorithm used to optimize the parameter settings for that model. For example, if you used a support vector regression approach, you would probably need to reference the quadratic solver that works under-the-hood to fit the model. You may need to read the documentation for the code libraries you use to determine how the model is fit. This is part of the applied machine learning process! **Also, provide estimates of runtime (either wall time or CPU time) required to train your model.**

XGBoost:

XGBoost is an optimized implementation of gradient boosting that emphasizes speed and performance. The training algorithm builds an ensemble of decision trees sequentially, where each new tree aims to correct the errors of the previous trees. It minimizes a regularized objective function, which combines a convex loss function (log loss for classification) and a regularization term to penalize model complexity and prevent overfitting. XGBoost employs a second-order Taylor expansion to approximate the loss function, utilizing both first and second derivatives (gradients and Hessians) for more accurate and efficient optimization. The algorithm uses an approximate greedy method to find the best split at each node by efficiently computing the gain from all possible splits using the accumulated statistics of gradients and Hessians. Advanced features like sparsity-aware learning and parallel processing are incorporated to handle large-scale data efficiently.

Catboost:

CatBoost is a gradient boosting algorithm that excels at handling categorical features without extensive preprocessing. Its training algorithm builds an ensemble of symmetric (oblivious) decision trees using gradient boosting. CatBoost introduces a unique technique called ordered boosting, which reduces target leakage and overfitting by training models on permutations of the dataset and ensuring that each model is trained without using future information. To optimize the model parameters, CatBoost minimizes a loss function (such as cross-entropy for classification) using gradient descent, where gradients are computed in an unbiased manner through permutation-driven schemes. The algorithm also employs efficient encoding methods for categorical variables, like combining categorical features with target statistics while applying Bayesian averaging to prevent overfitting. This approach allows

CatBoost to handle high-cardinality categorical features effectively within the gradient boosting framework.

Random Forest:

Random Forest is an ensemble learning method that constructs multiple decision trees during training and aggregates their results to improve predictive performance. Each tree is trained on a bootstrap sample (random sampling with replacement) of the original dataset, introducing diversity among the trees. At each node in a tree, a random subset of features is selected for consideration when determining the best split, which further reduces correlation between trees. The splitting criterion is based on impurity measures like Gini impurity or entropy, aiming to maximize information gain at each split. The algorithm does not involve parameter optimization through iterative methods; instead, it relies on the ensemble of many unpruned, fully grown trees to reduce variance and prevent overfitting. The final prediction is made by aggregating the outputs of all individual trees, using majority voting for classification or averaging for regression tasks.

LightGBM:

LightGBM is a gradient boosting framework designed for efficiency and scalability, particularly with large datasets. The training algorithm optimizes the objective function using gradient-based methods but introduces several innovations to enhance speed and reduce memory usage. LightGBM grows trees leaf-wise (best-first) rather than level-wise, splitting the leaf with the maximum loss reduction, which can lead to deeper trees and better accuracy. To optimize parameter settings, it employs Gradient-based One-Side Sampling (GOSS), which retains instances with large gradients to focus on the most informative samples, and Exclusive Feature Bundling (EFB), which reduces the number of features by bundling mutually exclusive ones. The algorithm uses histogram-based decision tree learning, discretizing continuous features into bins to accelerate the computation of the best splits and reduce memory consumption. These optimizations allow LightGBM to handle large-scale data efficiently while maintaining high predictive performance.

Runtime Estimate for each model:

```
XGBoost best model training CPU time: 178.04 seconds

CATBoost best model training CPU time: 11.52 seconds

Random Forest best model training CPU time: 16.05 seconds

LightGBM best model training CPU time: 24.53 seconds
```

4: Hyperparameter Selection:

You also need to explain how the model hyperparameters were tuned to achieve some degree of optimality. Examples of what we consider hyperparameters are the number of

trees used in a random forest model, the regularization parameter for LASSO or the type of activation / number of neurons in a neural network model. These must be chosen according to some search or heuristic. It would not be acceptable to pick a single setting of your hyperparameters and not tune them further. **You also need to make at least one plot showing the functional relation between predictive accuracy on some subset of the training data and a varying hyperparameter**

To achieve optimal performance for each model, I performed hyperparameter tuning using *GridSearchCV* from scikit-learn. This method systematically works through multiple combinations of parameter tunes, cross-validating as it goes to determine which tune gives the best performance. The hyperparameters chosen for tuning were those most impactful on model performance, such as the number of estimators, learning rate, and tree depth. Detailed tuning and training code is as followed, taking XGboost as an example:

```python
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import classification_report
import xgboost as xgb

# Step 1: Prepare the data
X = data_train_processed.drop(columns=['price'])  # Features
y = data_train_processed['price']  # Target variable

# Step 2: Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 3: Define and fit an initial XGBoost model
model = xgb.XGBClassifier(eval_metric='mlogloss')

# Step 4: Parameter tuning using GridSearchCV
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [2, 3, 4],
    'learning_rate': [0.01, 0.1, 0.2],
    'subsample': [0.8, 0.9, 1],
    'colsample_bytree': [0.8, 0.9, 1]
}

grid_search = GridSearchCV(estimator=model, param_grid=param_grid, scoring='accuracy', cv=5, verbose=1)
grid_search.fit(X_train, y_train)

# Step 5: Get the best parameters
best_params = grid_search.best_params_
print("Best parameters found: ", best_params)

# Step 6: Refit the model with the best parameters
best_model_XGB = xgb.XGBClassifier(**best_params, use_label_encoder=False, eval_metric='mlogloss')
best_model_XGB.fit(X_train, y_train)

# Step 7: Evaluate the model
y_pred = best_model_XGB.predict(X_test)
print(classification_report(y_test, y_pred))
```

Fig6: Hyperparameter Tuning for XGboost using Gridsearch CV

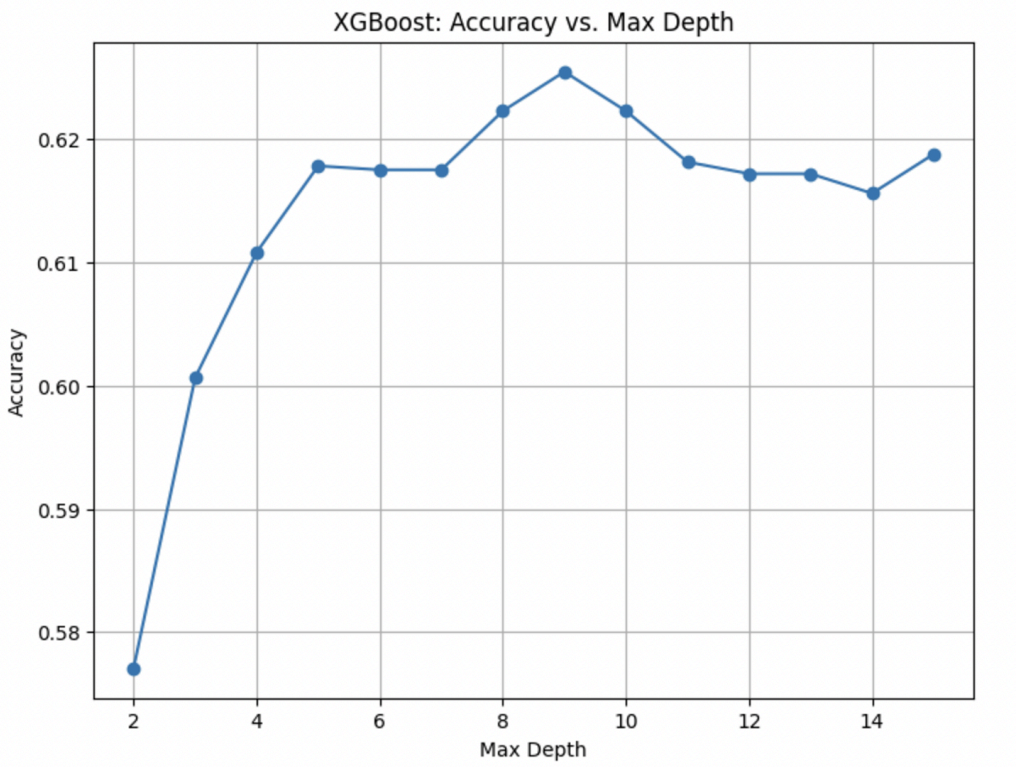

Fig 7: prediction accuracy over different depths, holding other variables fixed.
We see that the accuracy peaks around max depth of 9. Similar trends are observed across lightGBM, Catboost, and random forest. This suggests that an oversimple model with shallow depth could underfit the data, while an overly complicated model with greater depth might risk overfit the training data. The overall hyperparameter tuning of the final ensemble model is done separately in each contributing submodels, since the final model is an ensemble model with hard majority voting. My experiment in changing weighting of each submodel does not achieve practical or statistical significance against simply majority voting.

5: Data Splits

Finally, we need to know how you split up the training data provided for cross validation. Again, briefly describe your scheme for making sure that you did not overfit to the training data.

I am provided with two datasets, one training set, one testing set. The testing set is untouched. I processed the dataset in the following steps:

Step 1: missing values imputation: set random seed, conduct a 8:2 train-test split in the training set, and fill the missing values respectively using median/mean/other hybrid approach. For the provided test set, the exact same missing value handling steps is applied, except exclusively on the testing set alone. This ensures that there is no information leakage within the missing value imputation.

Step 2: feature engineering and dummy variables creation: In this step, training set and testing set(with identification encoded) are combined together for feature engineering and dummy variable creation. Since the dummy variable creation (training and testing has the

same categorical/discrete value for the dummy columns I selected, and there is no imputing reliant on the distribution of the combined dataset, and thus no information leakage across training set, development set, and testing set).

Step 3: Cross validation to avoid overfitting: I employed cross-validation on the training data after the preprocessing steps to further guard against overfitting. The validation folds were strictly drawn from data unseen during training folds.

Overall, there is no information leakage throughout data splitting and processing. Cross validation in the training step further lowers the likelihood of model overfitting.

6 Reflection on Progress

Making missteps is a natural part of the process. If there were any steps or bugs that really slowed your progress, put them here! What was the hardest part of this competition?

a) **ISSUE:** imbalance prediction accuracy/F1 across different price levels
Prediction of in-between classes (price 1, 2, 3, 4) was very difficult, with the XGboost model having <0.65 F1 score for each of the four classes. On the contrary, predictions for price 0 and price 5 achieve a relatively higher F1 score. Similar behavior is observed across random forest models, CATboost models, and lightGBM models. Tuning of hyperparameters of these models, or changing seed to further observe results of different division of datasets, did not successfully solve the issue of imbalance.

	precision	recall	f1-score	support
0	0.83	0.84	0.84	540
1	0.58	0.64	0.61	524
2	0.50	0.47	0.49	513
3	0.48	0.52	0.50	556
4	0.55	0.47	0.51	463
5	0.82	0.79	0.80	544
accuracy			0.63	3140
macro avg	0.63	0.62	0.62	3140
weighted avg	0.63	0.63	0.63	3140

Fig 8: Result of a XGboost model I trained, which eventually became a part of the final ensemble model.

Causation: Upon plotting the price distribution of the training set, which is relatively even, we wouldn't expect to have I engineered features primarily for high/low prices and chose amenities for high prices (features like isLuxury, isNotLuxury, for example.) I should engineer more features that serve to distinguish between prices.

Solution/ future room of improvement:
a (i) engineer feature from description:
I utilized a TF-IDF to try to mine keywords distinct to each price level, and try to make an indicator feature that aims to reduce entropy in classification. For example, I can use tf-idf to mine distinct words that describe price level 1, like “budget friendly, shared bathroom, students, queens…”

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd

# Filter out price = -1 and group descriptions by price
filtered_df = df[df['price'] != -1]
price_groups = filtered_df.groupby('price')['description'].apply(lambda x: ' '.join(x.dropna()))

# Compute TF-IDF for the descriptions
tfidf = TfidfVectorizer(stop_words='english', max_features=500)  # Adjust max_features as needed
tfidf_matrix = tfidf.fit_transform(price_groups)

# Create a DataFrame to display TF-IDF scores
tfidf_df = pd.DataFrame(tfidf_matrix.toarray(), index=price_groups.index, columns=tfidf.get_feature_names_out())

# Extract top keywords for each price level
top_keywords = {}
for price in tfidf_df.index:
    top_keywords[price] = set(tfidf_df.loc[price].sort_values(ascending=False).head(200).index)  # Top 50 keywords

# Find non-overlapping keywords for each price level
distinct_keywords = {}
for price, keywords in top_keywords.items():
    other_keywords = set().union(*[kw for p, kw in top_keywords.items() if p != price])
    distinct_keywords[price] = keywords - other_keywords

# Print the distinct keywords for each price level
for price, keywords in distinct_keywords.items():
    print(f"Price = {price}: {', '.join(keywords)}")
```

```
Price = 0: curtains, medical, spaces, furniture, bedford, club, ceiling, managed, crown, cooking, use, community, outside, door, 125, stuy, quality, mattress,
Price = 1: bronx, make, areas, privacy, university, cable, additional, note, super, mall
Price = 2: separate, lower, sunny, lined
Price = 3: speed
Price = 4: duplex
Price = 5: ground, dedicated, sleeps, concierge, fitness, simply, balcony, restaurant, include, discover, madison, retreat, chelsea, beautifully, bar, world,
```

Fig 9: using Tf-idf to mine distinct keywords for each price level

However, the result was not very good for price level 1, 2, 3, 4, as the selected keywords do not seem to clearly distinguish the price levels (near uniform distribution). Further investigation might be needed.

a (ii) engineer feature from amenities:
I used the distribution of top amenities across properties of each price level. Select these dummies to provide additional information for classification. For example, a property with an elevator/ swimming pool would had a higher chance of being in a higher price level than an airbnb located in the basement.

b) **ISSUE**: Long runtime of hyperparameter tuning due to Grid Search:
Causation: By setting the depth of the tree to high, the model risks overfitting, and the excessive training time would lead to a
Solution: first do a grid search, identify the best combination of parameters. Holding other parameters fixed, experiment the change in the results metrics by altering depth of the trees to see how the performance evolves as the depth of the tree increases. Finally, pick a depth with reasonable runtime to fit but also with decent predictive performance.

c) ISSUE: Overfitting on the testing set, leading to poor kaggle score
Causation: the missing values are initially imputed together, causing information leakage.
Solution: Process the missing values separately for training set, development set, and testing set, resulting in slight increase in kaggle score and the training loss convergence.

7: Predictive Performance:

kaggle: Alan Huang
alanhuangatduke@gmail.com

8: Code:

https://colab.research.google.com/drive/1GlLxTG9YyWR1P9WDvYTAdI7pkR0pCTWU#scrollTo=F1tzNxxeFNTQ

9 Citation:

Chen, T., & Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–794.

Github link: https://github.com/dmlc/xgboost, provided by the DMLC community

Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A. V., & Gulin, A. (2018). CatBoost: Unbiased Boosting with Categorical Features. *Advances in Neural Information Processing Systems*, 6638–6648.

Github link: https://github.com/catboost/catboost, developed by Yandex

Breiman, L. (2001). Random Forests. *Machine Learning*, 45(1), 5–32.

Link: https://scikit-learn.org/stable/, provided by scikit-learn

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., ... & Liu, T.-Y. (2017). LightGBM: A Highly Efficient Gradient Boosting Decision Tree. *Advances in Neural Information Processing Systems*, 3146–3154.

Github link: https://github.com/microsoft/LightGBM, provided by microsoft