Template for In-Class Kaggle Competition Writeup

CompSci 671

Due: Nov 26th 2024

Kaggle Competition Link

Your Kaggle ID: aaaronhsu (Aaron Hsu)

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.

1.1 Predictive Power

I expected that certain features would have significantly more predictive power than others. For example, I expected the feature *accommodates* would be more useful to include in the model than *last_review*.

As a starting point, I decided to train an XGBoost model with default parameters to explore the inital set of default features to provide insight into which features I should focus on engineering. I converted all features into numerical/boolean/categorical data.

From this, I noted that the features that had high predictive power and should be processed further were

1. *room_type*
2. *property_type*

Something else I found interesting was that *longitude* was included as a high predictive

Feature	Avg Importance
room_type	1.000000
accommodates	0.480873
property_type	0.480536
host_listings_count	0.344816
beds	0.344649
calculated_host_listings_count_entire_homes	0.333735
calculated_host_listings_count_private_rooms	0.316959
longitude	0.296735
host_total_listings_count	0.296483
minimum_nights	0.238767
instant_bookable	0.213358
host_acceptance_rate	0.189449
bedrooms	0.178655
calculated_host_listings_count_shared_rooms	0.147691
neighbourhood_cleansed	0.146651

Table 1: Top 15 features by average importance

power feature, but *latitude* was not. This led me to believe that it would be worth it to explore alternate ways to represent location data.

1.2 Difficulty of Processing

Some features require significantly more work than others to process than others. The features that I noted would be non-trivial to process were:

1. *description*
2. $property_type$
3. $neighbourhood_cleansed$
4. $room_type$
5. *amenities*
6. *reviews*

Because *description* and *reviews* would be challenging difficult to parse, I decided to drop these columns. I did consider the idea of using sentiment analysis using libraries like

Vader and NLTK, but decided against this because I predicted that most of the information that the feature would provide could be revealed by other default features. For example, most listings that have extensive description fields contain information about the listing that can be found in other fields like *accomodates* and $neighbourhood_cleansed$.

1.3 Feature Engineering

1.3.1 Property Type

I knew that the predictive power of this feature was high (from the initial predictive power test that I ran initially), so I knew that I'd need to process this feature into a more understandable format. Instead of one hot encoding the property type directly, I decided to group properties by their type to create a hierarchical classification system to reduce dimensionality before I one-hot encoded it. The idea behind this is that it better represents the market segments, so the model does not need to attempt to learn the distinction between features like *Private room in home* and *Private room in loft*, when there practically is none. The original dataset contained very specific property types (e.g., 'Private room in townhouse', 'Entire villa', 'Room in boutique hotel') which I consolidated into five main categories:

1. Premium Entire Properties (villas, townhouses, condos, etc.)
2. Standard Entire Properities (rental units, standard homes, serviced apartments)
3. Hotel-Style Accommodations (hotel rooms, boutique hotels, aparthotels)
4. Private Rooms (any form of private room)
5. Shared/Budget Accommodations (shared rooms, hostels, etc.)

I was able to validate this hierarchy by rerunning the feature importance, where the one-hot encoded property type indicators consistently ranked among the top predictive features (visualization below).

1.3.2 Neighbourhood Features

Though neighborhoods did not seem to hold very high predictive power (15th on the inital run), I assumed this was because there were many different types of neighborhoods and thus the model wasn't able to establish a clear correlation between neighborhood names and their relative price.

To more directly establish this relationship, I created an index for relative prices of listings for each neighborhood. Iterating over each listing, I tracked the average price of the listings in each neighborhood. I then divided this value by the average price of all listings to generate an index where:

- Neighborhoods with index value < 1 are "cheaper"
- Neighborhoods with index value > 1 are "expensive".
- Neighborhoods with < 5 listings have index values set to 1.

I also one hot encoded the *neighbourhood_group_cleansed* field, which contained borough information. Because there are only 5 boroughs I didn't feel the need to reduce dimensionality here.

1.3.3 Distance to Landmarks

I noticed in the initial feature importance evaluation that latitude was considered important, but not longitude. This led me to consider the ways that we could create a derived feature that would incorporate both.

I initially experimented with a simple feature that measured the log of the haversine distance of the listing from the center of the city using the (I used Times Square in this instance). I decided to take the log because I expected a non-linear (potentially expontential) relationship between the price of a listing and its location.

This performed quite well, so I decided to create features that measured the distance from other landmarks like the Brooklyn Bridge, World Trade Center, etc.. After confirming that these features were also valuable, I decided to omit them in place of two combined metrics: *log_min_landmark_distance* and *log_mean_landmark_distance*. I figured that having redundant features would unnecessarily overfit on distance metrics.

1.3.4 Amenities

The amenities feature was different from the rest of the features I processed because it was provided as a JSON-formatted string containing an array of amenities for each listing. Rather than one-hot encoding all possible amenities (which would have created excessive dimensionality), I focused on identifying and tracking "premium" amenities that would likely correlate with higher listing prices.

I settled on the amenities

- *Pool*
- *Hot tub*
- *Gym*
- *Doorman*
- *Elevator*
- *Free parking*
- *Washer*
- *Dryer*

I then created a simple count-based feature that tracked the number of premium amenities that each listing had.

```
df['premium_amenities_count'] = df['amenities_list'].apply(
    lambda x: sum(1 for amenity in x
    if any(premium in amenity for premium in premium_amenities))
)
```

1.3.5 Host Characteristics

The dataset provided several features related to the host, including *host_listings_count*, *host_response_time*, and various other metrics. Initial feature importance analysis showed that host-related features were significant predictors, particularly *calculated_host_listings_count_entire* and *calculated_host_listings_count_private_rooms*.

Thus, I decided to create a few derived features that measure a host's responsiveness and portfolio of listings.

- *is_professional_host*

```
df['is_professional_host'] = (df['host_listings_count'] > 3)
```

- *host_experience_years*

```
df['host_experience_years'] = (
    current_date - df['host_since']
).dt.total_seconds() / (365.25*24*60*60)
```

- *entire_home_ratio*

```
df['entire_home_ratio'] = (
  df['calculated_host_listings_count_entire_homes'] /
  df['host_listings_count'].replace(0, 1)
)
```

- *private_room_ratio*

```
df['private_room_ratio'] = (
  df['calculated_host_listings_count_private_rooms'] /
  df['host_listings_count'].replace(0, 1)
)
```

None of these features ended up in the top 15 of the final most important features. I suspect this is because the raw features *calculated_host_listings_count_private_rooms* and *calculated_host_listings_count_entire_homes* were sufficient for classification (and did show up in the top 15 predictive features).

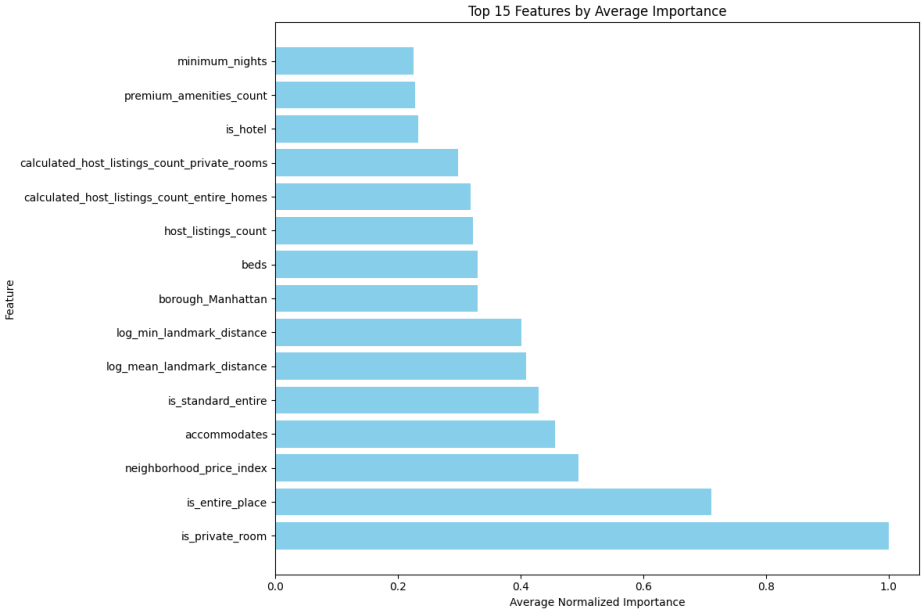

Figure 1: Feature importance plot

2 Models

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!

2.1 XGBoost

I initially chose to use XGBoost mainly because of its strong out-of-box performance for ordinal classification on low compute systems (I was running this locally). However, I later learned that XGBoost excels at classifying sparse datasets, which was important given that there were fields in our dataset that were missing (e.g. *review_scores* fields).

XGBoost also includes L1 (lasso) and L2 (ridge) regularization out of box.

2.2 LightGBM

I chose LightGBM as my second model because it efficiently handles of categorical features. This was particularly important here because our dataset had a significant number of categorical variables. LightGBM implements a leaf-wise tree growth strategy, as opposed to XGBoost's level-wise approach, which typically results in faster training times and better performance on larger datasets.

The model's Gradient-based One-Side Sampling (GOSS) technique was valuable for maintaining accuracy while reducing memory usage, which was crucial for local computation. Before I handled categorical variables (property types, neighborhoods, host response times, etc.) using one-hot encoding, I used LightGBM's special treatment of categorical features to input these features directly without preprocessing them, which initially simplified my pipeline.

2.3 Random Forest

I selected Random Forest as my third model because it uses a fundamentally different approach (bagging) compared to the boosting methods of XGBoost and LightGBM, providing a fundamentally different approach to the classificaiton problem. Random Forest's ability to train trees in parallel made it computationally efficient, and its averaging approach makes it naturally resistant to overfitting.

The model's robustness to outliers was particularly valuable given the varied nature of the pricing data, where some listings could be significant outliers. Also, Random Forest's feature importance calculations provided a useful comparison point to validate the importance scores from our boosting models.

2.4 Model Ensemble

As listed above, because the models have different approaches to the classification problem, I decided to build an ensemble of these three models to leverage their complementary strengths. The two boosting algorithms (XGBoost and LightGBM) use level v.s. leaf-wise tree growth and Random Forest uses bagging, which helps us mitigate the individual weaknesses of each model. For example, XGBoost might be sensitive to noisy data, LightGBM

might miss subtle numerical relationships, and Random Forest might miss sequential patterns. In all tests, the model ensemble performed significantly better (RMSE improvement of 0.03) than any individual model.

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.

3.1 XGBoost

XGBoost uses gradient boosting, which sequentially builds trees to minimize a loss function. In this case, I chose the multiclass softmax objective over ordinal regression because the price categories in our dataset were created using quantile binning, which means that the "distance" between categories isn't super meaningful for us (cat 2 is not necessarily twice the price of cat 1). The algorithm builds each new tree to predict the residuals (errors) of the previous trees, gradually improving the model's predictions. Additionally, XGBoost's implementation of second-order gradients in its training process helps it find better split points and converge faster than traditional gradient boosting methods.

It took roughly 10 seconds to train each of the 5 folds (M2 Macbook Air).

3.2 LightGBM

LightGBM employs a leaf-wise tree growth strategy that chooses the leaf with maximum delta loss to grow, resulting in more efficient training compared to traditional level-wise growth. The algorithm implements Gradient-based One-Side Sampling (GOSS) to focus on data instances with larger gradients while randomly sampling instances with smaller gradients, maintaining accuracy while reducing computation. LightGBM's exclusive feature bundling (EFB) technique efficiently handles categorical features by bundling mutually exclusive features, reducing memory usage. The model also uses histogram-based learning,

which buckets continuous features into discrete bins, speeding up training while maintaining good accuracy. The combination of these techniques made LightGBM particularly efficient for our large dataset with mixed feature types.

This also took roughly 10 seconds to train each of the 5 folds (M2 Macbook Air).

3.3 Random Forest

Random Forest training constructs multiple decision trees in parallel, with each tree using a bootstrap sample of the data and considering a random subset of features at each split. The algorithm implements bagging (Bootstrap Aggregating), which helps reduce overfitting by introducing randomness both in the data samples used for each tree and in the features considered at each split. The final prediction is made by averaging (for regression) or voting (for classification) across all trees, providing natural protection against outliers and noise in the data. Random Forest's training process is naturally parallel, allowing efficient use of multiple CPU cores, and its random feature selection at each split helps ensure that all trees are sufficiently different from each other. The algorithm also provides built-in out-of-bag error estimation, which gives us a reliable estimate of the generalization error without requiring a separate validation set.

This took around 1.5 seconds to train each of the 5 folds (M2 Macbook Air).

3.4 Ensemble Model

The ensemble model combines predictions from XGBoost, LightGBM, and Random Forest through a simple rounded average of each of the models predictions. Each individual model is trained independently on the same training data, but this ensures that if two of the models agree on the same price, the final prediction is highly likely to be that price (assuming the third model is within a quantile, which is reasonable given that they were trained on the same dataset/problem).

Each fold of the ensemble took around 18 seconds to train (M2 Macbook Air).

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.

4.1 GridSearchCV

Initially, I used GridSearchCV with a parameter grid

```
param_grid = {
  'max_depth': [3, 4, 5, 6, 7, 8],
  'learning_rate': [0.01, 0.05, 0.1, 0.2],
  'n_estimators': [100, 200, 300, 400],
  'min_child_weight': [1, 3, 5, 7],
  'subsample': [0.6, 0.7, 0.8, 0.9]
}
```

to find an optimal set of parameters. The benefit of doing this is that it's an exhaustive search and thus will find the optimal set of parameters out of the values provided. However, each time I tweaked my features/model, I'd have to rerun the grid search, which was computationally expensive (this took a little more than an hour to run each time).

4.2 BayesSearchCV

Instead, I decided to use BayesSearchCV to find an optimal set of parameters. Because this model uses probabilistic modeling to guide the search process instead of exhaustively searching a space, it ran significantly faster. This meant that I was able to provide it a significantly larger search space without compromising runtime. Each search ended up taking around 30 minutes.

```
search_spaces = {
  'learning_rate': Real(0.01, 0.3, prior='log-uniform'),
  'max_depth': Integer(3, 10),
  'min_child_weight': Integer(1, 7),
```

```
  'subsample': Real(0.6, 1.0),
  'colsample_bytree': Real(0.6, 1.0),
  'n_estimators': Integer(100, 1000),
  'reg_alpha': Real(0.001, 10, prior='log-uniform'),
  'reg_lambda': Real(0.001, 10, prior='log-uniform'),
}
```

One of the added benefits of this is that we are no longer bound to discrete values. This meant that the parameters that were selected were tuned more precisely (e.g. we ended up with an XGBoost learning rate of 0.0156757227368663 instead of 0.01).

At the end, I ended up with the following hyperparameters for each of the following models

1. XGBoost

```
params = {
    'objective': 'multi:softmax',
    'num_class': 6,
    'min_child_weight': 1,
    'colsample_bytree': 0.6,
    'reg_lambda': 0.001,
    'eval_metric': ['mlogloss', 'merror'],
    'tree_method': 'hist',
    'random_state': 42,

    'colsample_bytree': 0.6,
    'learning_rate': 0.0156757227368663,
    'max_depth': 9,
    'min_child_weight': 1,
    'n_estimators': 882,
    'reg_alpha': 0.053626850522901205,
    'reg_lambda': 0.001,
    'subsample': 0.8744536442470929
}
```

2. LightGBM

```
params = {
    'objective': 'multiclass',
    'num_class': 6,
    'feature_fraction': 0.8,
    'bagging_fraction': 0.8,
    'bagging_freq': 5,
    'metric': 'multi_logloss',
    'random_state': 42,
    'reg_alpha': 0.001,
    'reg_lambda': 0.001,

    'colsample_bytree': 0.7692725338291122,
    'learning_rate': 0.07485528354670547,
    'max_depth': 10,
    'min_child_samples': 2,
    'n_estimators': 100,
    'num_leaves': 87,
    'reg_alpha': 0.001,
    'reg_lambda': 0.001,
    'subsample': 0.8553076706085649
}
```

3. Random Forest

```
params = {
    'n_estimators': 200,
    'max_depth': 10,
    'min_samples_split': 5,
    'min_samples_leaf': 2,
    'random_state': 42
}
```

4.3 Tuning Visualization

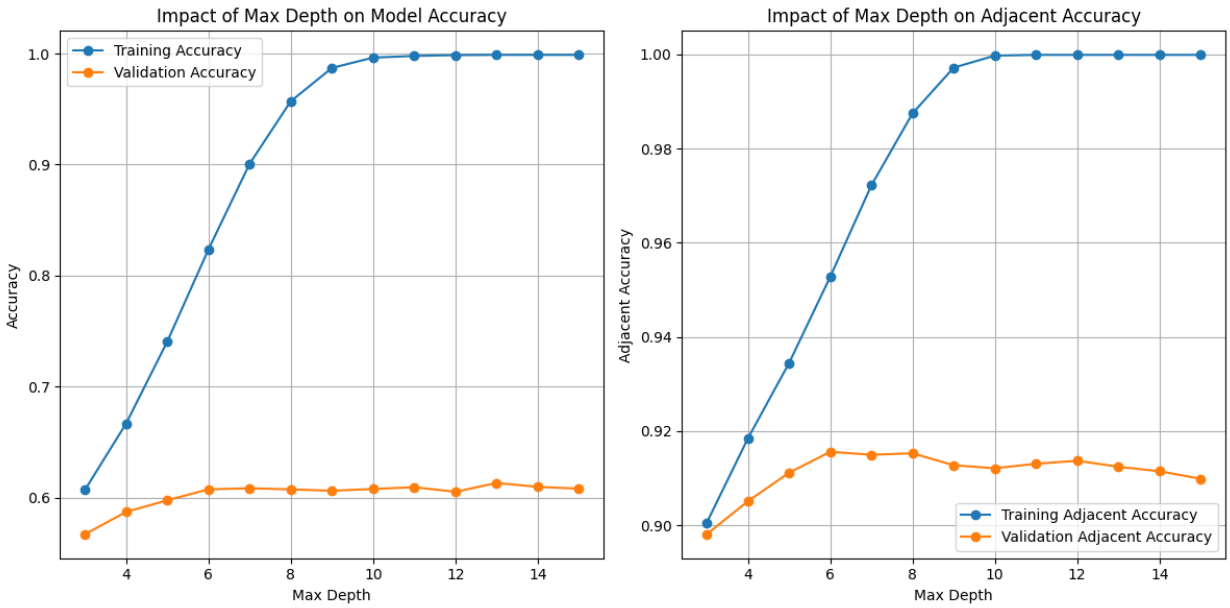

Figure 2: XGBoost Max Depth Tuning

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.

5.1 Stratified KFold

For this classification problem, I decided to use Stratified K-Fold cross-validation with k=5 to ensure that the fold maintained the same distribution of price categories as the full dataset. Although the dataset was already fairly balanced, I wanted to ensure that the folds that were being generated were also balanced.

This approach also gave a few other benefits, namely that every listing appears in the validation set once and that with 5 folds, each fold had roughly 3,000 data points which I found was a good balance between computational cost and performance. If I was running this on a more powerful computer, I would have increased the fold number to 10 for higher training stability.

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?

6.1 Regressive v.s. Ordinal

When I first started this project, I chose to treat the problem as a regression task, predicting continuous price values and rounding them to the nearest category (0-5). This approach seemed intuitive because price is naturally a continuous values but this ended up leading to subpar models.

The main issue was that lose categorical information when using regression. The regressive approach assumes equal distances between categories. For example a listing that was predicted to be 3 should not necessarily be twice as expensive as a listing that was rated a 1.5. Because the categories were built using quantile binning, the "distance" between each range is not very valuable.

After changing the model to predict ordinal values, my accuracy jumped by almost 0.1.

6.2 Feature Engineering Redundancy

Unsurprisingly, a few of the features that I built out did not end up being very useful (*avg_review_score*, *listing_age_days*, *host_since*). Some of these features I continued to tweak to provide more predictive power and others I removed to decrease model complexity.

Initially, my amenities processing was significantly more complicated. Instead of choosing a small set of amenities to be "premium" and storing the count as a single feature, I binned the amenities into different qualities. I created categories *premium_amenities*, *essential_amenities*, *mediocre_amenities*, etc. and stored the count for each one. These features were not evaluated to have high predictive power, so I attempted to build derived features like *ratio_premium_amenities*. However, these did not seem to be useful either, so I ended up dropping these new features and kept the count for premium amenities.

Another feature that I initially overengineered was the distance measurement to landmarks. As I mentioned in the feature engineering section, I started with calculating the log distance of each listing to Times Square, which had significant predictive power. Because of this, I decided to incorporate a larger list of landmarks and had a feature for each one. These values also had high predictive power, but I noticed that many of the features had

high correlation values, indicating that some of these features when considered in aggregate were not contributing significantly to the predictive accuracy of the model.

```
corr_matrix = {
  'times_square_empire_state': 0.94,
  'wall_street_world_trade': 0.92,
  'rockefeller_times_square': 0.96
}
```

Instead, I decided to aggregate each distance feature into two features that captured the values sufficiently

```
df['log_min_landmark_distance'] = df[landmark_distance_columns].min(axis=1)
df['log_mean_landmark_distance'] = df[landmark_distance_columns].mean(axis=1)
```

7 Predictive Performance

Upload the submissions from your best model to Kaggle and put your Kaggle username in this section so we can verify that you uploaded something. Also, compare the effectiveness of the models that you used via the Root Mean Squared Error (RMSE) score that we are using to evaluate you on the Kaggle site. Half (10) of the points from this section will be awarded based on your performance relative to your peers. We will use the following formula to grade performance in the Kaggle competition:

$$\text{Points } = \min\left(10, \left\lfloor \frac{\text{Percentile Rank}}{10} \right\rfloor + 1\right) \tag{1}$$

For example, being in the 90th percentile will give you $\frac{90}{10} + 1 = 10$ points. Being in the 34th percentile will give you $\left\lfloor \frac{34}{10} \right\rfloor + 1 = 4$ points.

Bonus Criteria: Up to 5 points are awarded for using interpretable modeling approaches or feature engineering techniques that enhance interpretability. Possible ways to earn these points include:

- **Fully Interpretable Model (e.g., Decision Tree, Generalized Additive Models):** Use of inherently interpretable models, like a decision tree or linear/logistic regression, that allow for clear reasoning about predictions.

- **Interpretable Feature Engineering Pipeline:** Creation of features that are easily interpretable, or clear explanations provided for any engineered features that contribute to interpretability.
- **Feature Importance Analysis:** Provides and discusses a feature importance analysis to identify and explain key predictive factors.

Note that it is possible to do very poorly in the competition but still get an A on this assignment if the other sections are filled out satisfactorily. This encourages you to take risks! Use plots or other diagrams to visually represent the accuracy of your model and the predictions it makes.

7.1 Kaggle Results

My username is aaaronhsu and is listed under "Aaron Hsu" on the Kaggle leaderboards (#97 on the public leaderboard with score 0.84147).

However, I did not get to reselect the submission that was used for final grading, which is multiple days old.

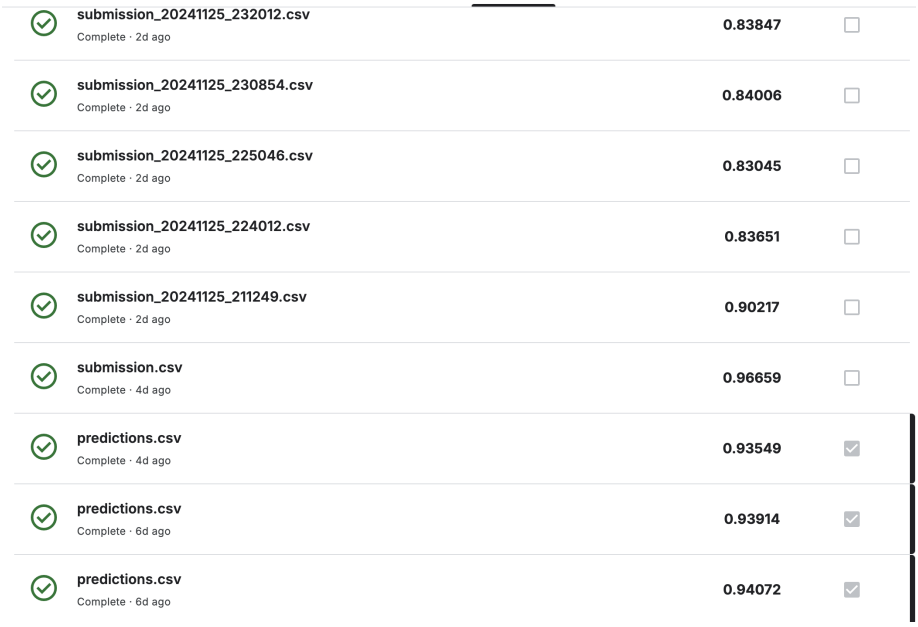

Figure 3: Kaggle Submission History

If possible, I'd like to use the latest 3 submissions prior to the deadline for my final standing calculation.

7.2 Interpretability

In this competition, I made an effort to only create features that were intuitive and to use tree-based models for increased interpretability. My feature engineering pipeline is straightforward and structured such that an intermediate csv is created *processed_train/test.csv* that only contains features that I used for my model.

In the end, I used 25 features for the training of my final model, with 13 of them being results of one-hot-encoding.

- neighborhood_price_index
- accommodates
- beds
- bedrooms
- borough_Brooklyn
- borough_Manhattan
- borough_Queens
- borough_Bronx
- borough_Staten_Island
- calculated_host_listings_count_entire_homes
- calculated_host_listings_count_private_rooms
- minimum_nights
- premium_amenities_count
- instant_bookable
- host_listings_count
- availability_rate_30
- avg_review_score
- listing_age_days

- log_mean_landmark_distance
- log_min_landmark_distance
- is_entire_place
- is_private_room
- is_hotel
- is_shared
- is_premium_entire
- is_standard_entire

8 Code

Copy and paste your code into your write-up document. Also, attach all the code needed for your competition. The code should be commented so that the grader can understand what is going on. Points will be taken off if the code or the comments do not explain what is taking place. Your code should be executable with the installed libraries and only minor modifications.

Code is attached in the associated ZIP file. I did not use Jupyter notebooks for this assignment, and instead opted to split each part of the pipeline into a separate file. Thus the order in which files are run is important.

My final model is in the folder *ordinal* and the files should be run in the following order:

1. ordinal/preprocess.py
2. ordinal/train.py
3. ordinal/predict.py

Relevant packages can be installed via the requirements.txt in the root directory.

I also have multiple files for certain tasks

1. Model definitions (ordinal/models.py)
2. Tuning hyperparameters (ordinal/hyperopt.py)

3. Visualizations (ordinal/feature_analysis.py, hyperparmeter_graph.py, visualize_xgboost.py)
4. Baseline feature evaluation (ordinal/simple_preprocess.py)

The files in the regressive folder follow a similar format.

9 Logistics

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