Template for In-Class Kaggle Competition Writeup

CompSci 671

Due: Nov 26th 2024

Kaggle Competition Link

Your Kaggle ID (on the leaderboard): Zakk Heile (Zakk Heile on Kaggle)

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.

I started off by making histograms of the distribution of each numerical feature. The one that was most interesting to me was the categorical response variable 'price'. The distribution was essentially even which makes sense given that these come from six divided quantiles, but there was some variation in it.

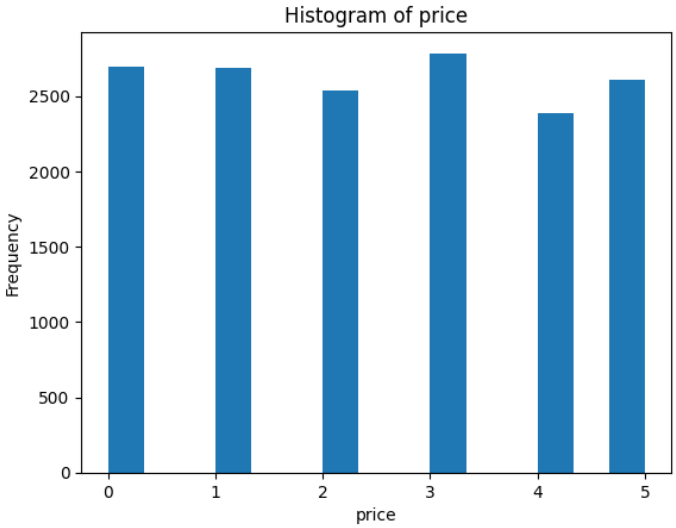
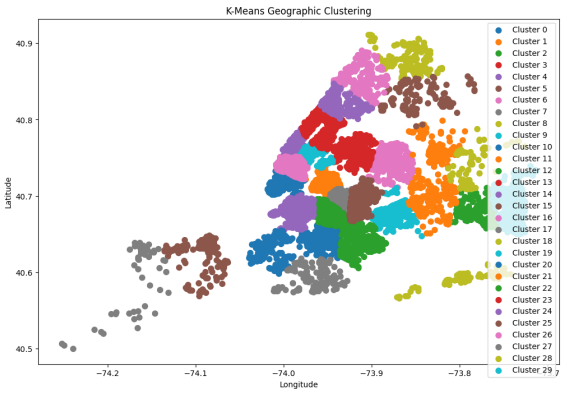
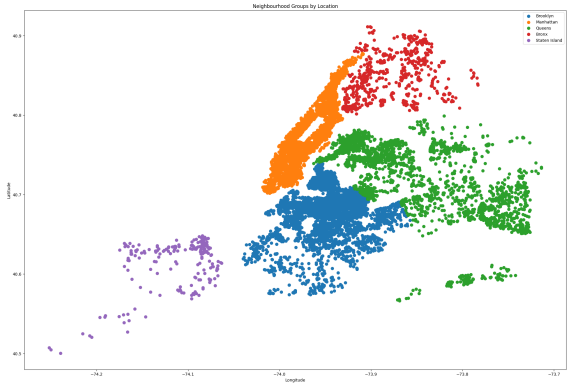


Figure 1: Distribution of Prices

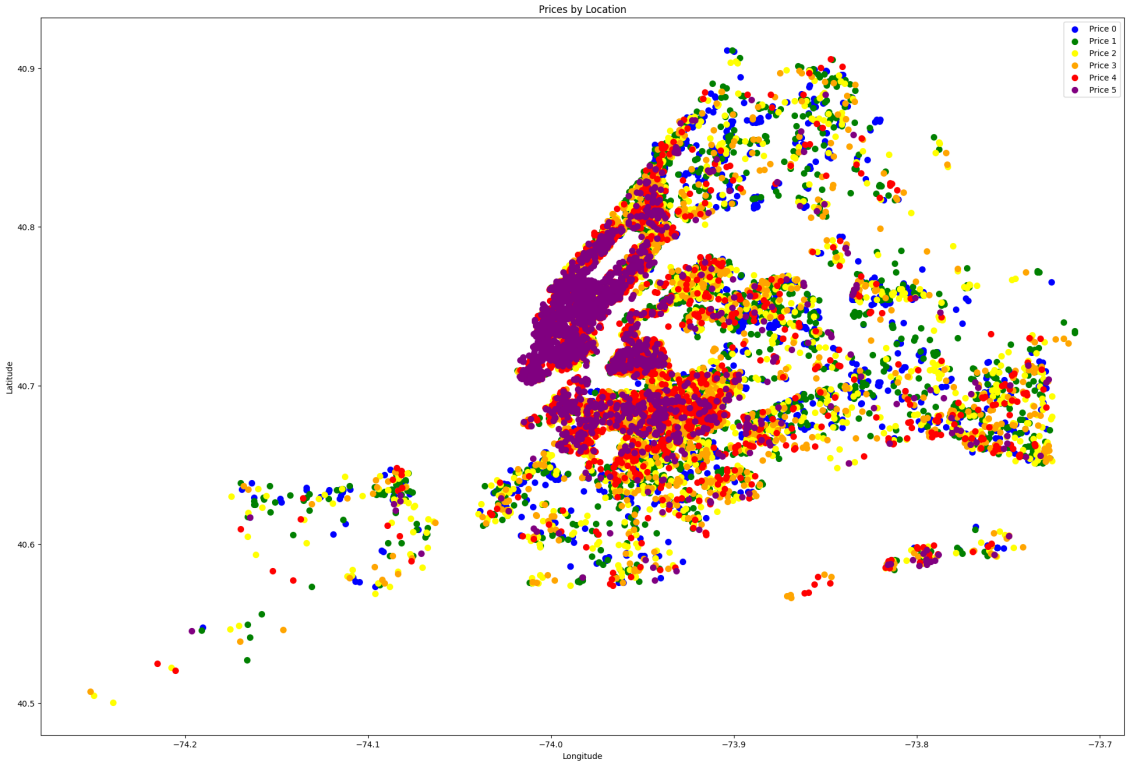
The distribution of latitude and longitude was also of interest to me. This was something that you may not expect to have a clear most common value, but it did in both cases. One can verify by looking at the coordinates that this is the New York City Manhattan area. I was particularly concerned with how the location information was encoded because say for a decision tree algorithm, you would be making a lot of splits (Lat $>$ X and Lat $<$ Y and Long $<$ Z and so on) to make predictions on a certain location, however, the neighborhood location alleviated some of my worries. There are both neighborhood and neighborhood group so the model has the ability to capture a certain small or large radius with only one split. However, there is no good medium size or flexibility between the sizes of groups. You must consider the interaction between latitude and longitude in general, but especially so because the data points in the standard geographic basis are not aligned with either of them. Because of that, I added additional features - rotated axes to try to align with more of the subdivisions/clusters. This accomplishes two things - it allows the model to make different polygons in deeper models to encode clusters or neighborhoods, but also allows it to get at the area it wants in just a single split. Before this, I tried K-Means clustering on the data but by inspection, I was never happy with how it grouped areas of New York. Using my own judgement, I felt that no matter the number of clusters it was still making odd decisions that defied what I knew about the area.



(a) K-Means Clustering



(b) Neighborhood Groups



(c) Price by Location

Figure 2: Visualizations of Clustering and Pricing by Location

Another variable I was interested in was 'hostacceptancerate' - why would a host reject someone? My conclusion is that they must live at the house for a large portion of the time, so this could be an interesting predictor. I also instinctively felt like 'minimumnights' and 'maximumnights' would not be useful, but it could potentially tell you what type of vacation it is meant for which could have further implications on price.

Another thing I did was make the price a binary variable (splitting on the lower 3 quantiles

and upper 3 quantiles) and plot ROC curves for all individual numerical features. In the interest of intepretability, I did not fit any complicated models that would mess up the monotonicity of the variables. In other words, I didn't want a model trying to maximize predictive power by not just fitting a line with a consistent direction - it was important for me to have the idea that if I moved a variable value up, it would always predict the same direction and it not depend on where I moved it from. For that reason, I didn't fit Random Forest models. I chose Logistic Regression which gives me cut-offs that must act that way.

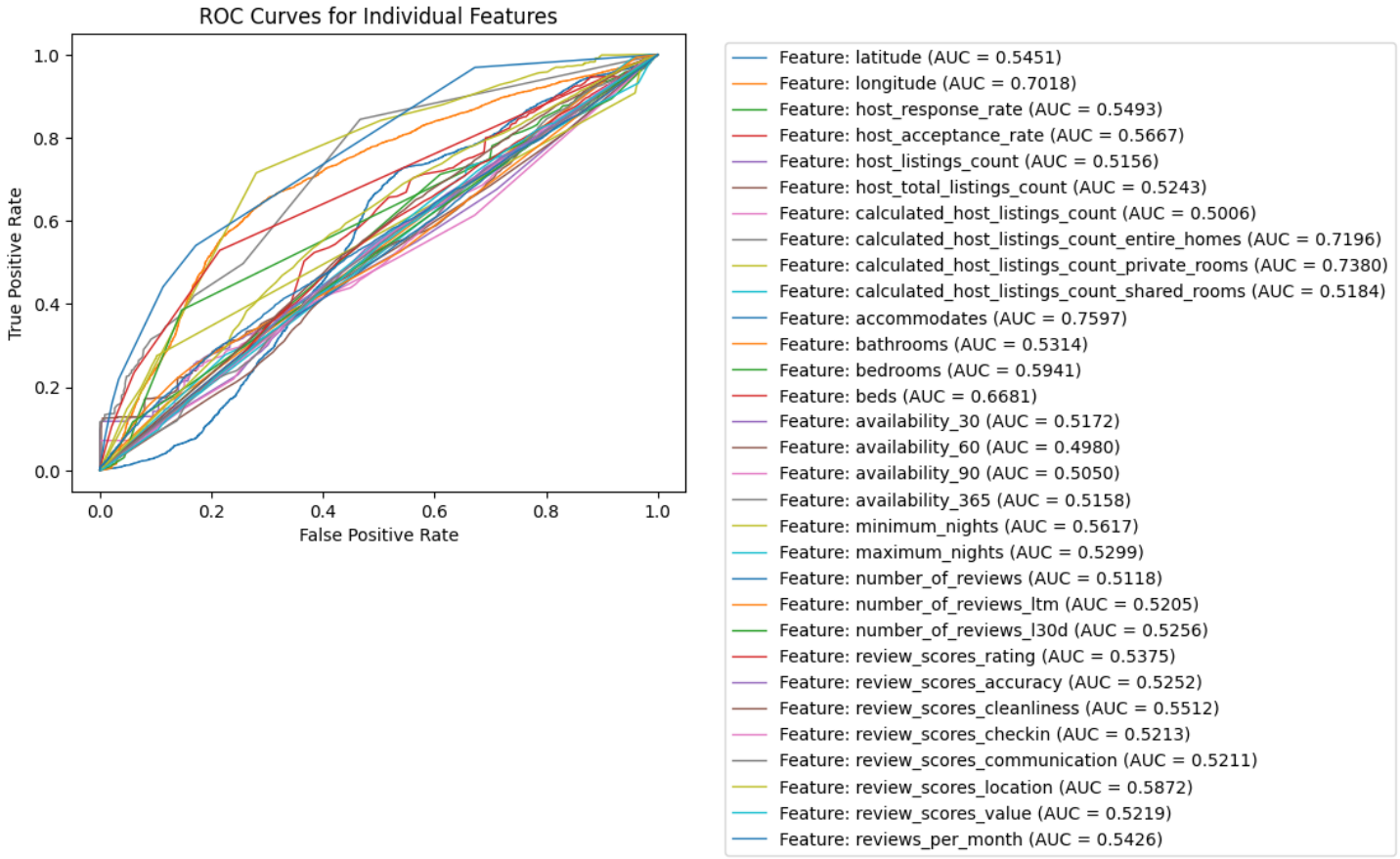


Figure 3: ROC Curves: Performance of Predictive Models

The feature with the highest AUC in this analysis was 'accommodates' which is not surprising. Also ranked highly was 'hostlistingcount', variables, 'longitude', 'beds', and 'bedrooms'. Beds and Bedrooms were not that surprising, but the number of host listings being (potentially) such a good predictor was surprising. In contrast, the number of reviews seems to not be that powerful of a predictor, at least in isolation.

As far as the features went, in general, I one-hot encoded most categorical things. I did this for 'propertytype', 'neighborhood' variables, and 'roomtype'. I dropped 'bathroomstext'

because there is already a numerical 'bathrooms' in the dataset. In the case of ordinal variables like 'hostresponsetime', I chose a logical numerical ordering to assign the set categories to. For any date, I eliminated hours/minutes/seconds and just recorded it as days since 1/1/2000. This was an arbitrary choice but it was a logical way to convert to integer values. While we have 'accommodates' as a predictor, I also wanted square footage, so using regular expressions, I searched the name, descriptions, and reviews field, taking the first number to come before square footage (or other variations). If it was not found, which is a lot of them, I recorded it as -1000 (though I ended up not using this metric because of how much was missing). With missing values in general, I often made them a new category instead of imputing them because in the majority of the cases, I thought that a variable was not missing at random and the amount of them was large enough to justify making the model learn another case. However, for bedrooms and bathrooms, which had relatively small numbers of missing values, I imputed them using the median but in the event of a non-integer value, rounded them in the direction of a mean - to try to account for different distributions. Then, I filled the number of beds with the most common number of beds given the number of bedrooms. I should note that there were only 3 observations missing how many bedrooms, so I am only using an estimate to compute the most common in 3 cases out of the tens of thousands. All the review scores which are 1-5 float ratings were missing if the host had no reviews, so I opted to assign these to -1 because it is fundamentally different than having any rating 1-5. However, for reviews per month, that is quite literally 0 so I assigned it as such. In cases where the host response and acceptance rate were missing, I assigned it another value, -10, because they are likely not missing at random. Most likely, they just don't have enough data to display/compute it, and there is a large enough sample to let the model figure out why.

There was also an anomaly that for a surprising amount of the data, the number of beds was less than the number of bedrooms (and sometimes as big of a difference as 1 to 9). Because of this, I replaced beds with the maximum of beds and bedrooms.

The following table shows a sample of rows where the number of `beds` is less than the number of `bedrooms`:

Accommodates	Beds	Bedrooms	Bathrooms
1	1.0	4.0	1.5
1	1.0	4.0	0.0
1	1.0	4.0	0.0
1	1.0	6.0	2.0
1	1.0	4.0	0.0
⋮	⋮	⋮	⋮
6	1.0	2.0	1.0
1	1.0	3.0	0.0
1	1.0	4.0	0.0
2	0.0	1.0	2.0
1	1.0	4.0	0.0

Table 1: Sample rows where `beds` are less than `bedrooms`.

The total number of rows where `beds` are less than `bedrooms` is 1007.

For rows where `beds` are not less than `bedrooms`, the average ratio of `bedrooms` to `beds` is:

$$\text{Average Ratio} = 0.80$$

Perhaps the most important covariate to consider was the amenities. I started off by just one-hot encoding them but obviously, the feature space exploded. After considering NLP techniques and other ways to represent the features, I feared that that would be more complicated than it was worth given that there are so many different key words and varying lengths and it may not be that successful to encode them in a single vector. I am not saying it wouldn't have worked but instead I opted to use inclusion-exclusion criteria on specific amenities I thought were useful after reading through the unique set of amenities. For instance, I am searching for swimming pool. To count as a swimming pool, you must have the word pool but without table (conflict with pool table). Additionally, I have two categories for parking, when it says free with it, or when it doesn't. The list goes on but that was my general approach to how to represent the amenities numerically - a list of carefully chosen binary variables with inclusion/exclusion criteria.

Lastly, I also created a few new features. Intuitively, it makes sense that shared bathrooms would be an important covariate, so I made a binary variable that indicates whether there are shared bathrooms. I also included a host proportion stat derived from two other statistics: the proportion of entire homes, private rooms, or shared rooms that they list.

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!

I tried a bunch of different models individually - GAMs, XGBoost, Random Forests, MLPs, Logistic Regression, and Linear Regression. With Random Forest, I got a 0.63 MSE on new data. With Boosting, I got a 0.81 MSE. With Logistic Regression, I got a 1.33 MSE, and so on. GAMs got an astonishingly low accuracy of 0.26. Some of these have been further improved after more data processing, but this gives an idea of their relative performance to each other. The point is clear, and it makes sense why this would be the case. Logistic Regression, Generative Additive Models, and Linear models all suffer because there are no interactions between covariates. How can latitude be a good predictor without longitude? Wouldn't the effect on price when you add a bathroom depend on where you are? Any model that is strictly considering the effects of each variable on their own is not going to do a good job. Furthermore, when you have as many features as I have - the dataset already had more than 50 but then I extracted important amenities, one-hot encoded categorical variables, and so on. The number of features now is really massive, and one model simply can't use them all. The reason why random forests are doing so well is because they can build different trees with different predictors. Beyond just the promise of decreasing variance, random forests are able to attack this dataset from so many different angles when other models can only attack it one way. That is why Boosting is also close in terms of MSE, it too has this capability. I wish I could compute a Rashomon set because the space of good models would be insanely large, but it would be too computational. I am trying to capitalize on the fact that there are so many good models that are also so different and use them together. I also ruled SVMs out because of the massive number of dimensions each observation has and SVMs are not adaptive like some of the other models I could use. By and large, this meant that random forests were the way to go.

I wanted to extend the idea of a random forest further, so I turned to stacking. I created

a bunch of level 1 models and fed their predictions to a final level 2 model which would have the last say. In my level 1 models, I really wanted to get a bunch of different perspectives. Because random forests were so successful, I used a bunch of them with differing parameters - depth, minimum samples to split, features available to split on at a given split, and the Greedy criterion for splitting. It might be intuitive that you would only want to split using MSE, but I also wanted to get models that systematically did something different and not just keep making the same kind of greedy decision. That is why I also split on the Poisson metric. In each of the forests of those different measures, I have deep trees that are allowed to overfit, trees that can be deep but can't overfit, and shallow trees. Additionally, I have boosting ensembles of a variety of depths. I also thought it was important to feed the metamodel some different more straightforward linear models because then it could potentially use them as a first gauge to then use the more complicated models to zone in on exactly what is happening. I have standard linear models, linear models with interaction terms or higher degree terms, and regularized regression methods like ElasticNet which combines Lasso and Ridge regularization. For an alternative to random forests that are single-models, I added 3 MLPs with different parameter sets. For my layer 2 model, the metamodel, I first thought that I could just fit something simple like Logistic Regression or a Linear model. These ideas did okay - I got an MSE of 0.57 and 0.59 respectively, but using a Random Forest as my metamodel got an MSE of 0.53. At the time, I was really afraid of overfitting by giving it the final say. It does make sense that given the large amount of models in the first layer that it would need to do more than a weighted average between them and learn the shortcomings of one in conjunction with another model. I tried 3 different depths of trees in the random forest through cross-validation and while the deepest tree did the best, it was just slightly better than the middle-depth tree. At this point, I was concerned about the complexity of my models and how many I stacked, so I computed feature importance for my metamodel and also looked at pairwise similarity between predictions in the layer 1 models to thin out how many models I was using. This was a systematic approach - I first removed anything that was less than 0.005 (proportion of different predictions), and then moved the cutoff up to a percent or two. After this process, I got it down to 6 models which involved kNN, Random Forest(s), Boosting, and a Linear model. Intuitively, it is clear that all these models are doing different things and form a very diverse ensemble. This did not outperform my original much more complex stacked ensemble but was extremely comparable and came within 0.0015 in my sandbox testing. This process served as initial hyperparameter tuning for the layer 1 models as well because I trained a variety of parameter sets per model in my original very dense layer and filtered them out on similarity and value to the metamodel.

We give a confusion matrix for the stacked ensemble on new data. From this, we can see

that the model is rarely more than 1 class off.

	Predicted Class					
Actual Class	1	2	3	4	5	6
1	247	75	6	0	0	0
2	26	155	106	14	1	0
3	2	35	177	79	8	0
4	0	9	71	201	56	0
5	0	2	20	78	159	25
6	0	0	6	17	79	230

Table 2: Confusion Matrix

External Libraries and References

- **Pandas** (1): Used for data manipulation, including loading datasets, and handling missing values.
- **NumPy** (2): Efficient numerical computations and matrix oeprations.
- **Matplotlib** (3): Utilized for visualizing data distributions, feature importance, and geographic clusters using scatter plots.
- **Scikit-learn** (4): Machine learning tasks such as model training, hyperparameter tuning, and evaluating metrics like RMSE and feature importance. Specific components include:
  - `LogisticRegression` for binary classification tasks.
  - `RandomForestRegressor` and `GradientBoostingRegressor` for regression modeling.
  - `StandardScaler` for feature standardization.
  - `GridSearchCV` for hyperparameter optimization.
- **XGBoost** (5): Used as a gradient-boosting model for its efficiency and ability to handle structured data in regression tasks.
- **Optuna** (6): Applied for automated hyperparameter optimization using a flexible and efficient trial-based approach.
- **Ast**: Used to safely parse and evaluate strings (e.g., parsing amenities in JSON-like formats).

- **Regex**: Regular expressions are used for extracting structured information such as square footage or specific keywords from textual data.

Algorithms and Figures

- **K-Means Clustering**: Implemented using Scikit-learn to cluster geographic coordinates.
- **ROC Curves and AUC**: Calculated and plotted using Scikit-learn's `roc_curve` and `auc` methods.
- **Random Forest Feature Importance**: Some of the feature importance visualizations rely on Scikit-learn's built-in `feature_importances_` attribute.

References

[1] Pandas Documentation. Retrieved from https://pandas.pydata.org/.

[2] NumPy Documentation. Retrieved from https://numpy.org/.

[3] Matplotlib Documentation. Retrieved from https://matplotlib.org/.

[4] Scikit-learn Documentation. Retrieved from https://scikit-learn.org/stable/.

[5] XGBoost Documentation. Retrieved from https://xgboost.readthedocs.io/en/stable/.

[6] Optuna Documentation. Retrieved from https://optuna.org/.

3 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.

The training data was divided into two sets: L1Train and L2Train. This is because our training process involved a two-layer model architecture.

The layer 1 data was used to train the layer 1 models and the layer 1 models then made predictions on new data - the layer 2 data.

The layer 2 data was further split into 5 folds for 5-fold cross validation. The meta model was trained on 80% of the layer 2 data, and tested on the other 20%, and this process was repeated on a rotated/new partition of the data 4 more times. This allowed us to test the model performance on new data and not overly cut into our training resources. I was concerned about training resources more than normal given that I was training two layers of models with the second layer having to be trained on predictions that the layer 1 models did on data that they were not trained on.

Optuna was used to optimize hyperparameters of the metamodel. We also tried different algorithms instead of just Random Forest which we ended up sticking with - each of them also had their hyperparameters optimized. Given the large possibilities for parameters and algorithms in the layer 1 models, we manually adjusted them and reran them, taking note ourselves of the likely best combination. We elaborate more in the next section.

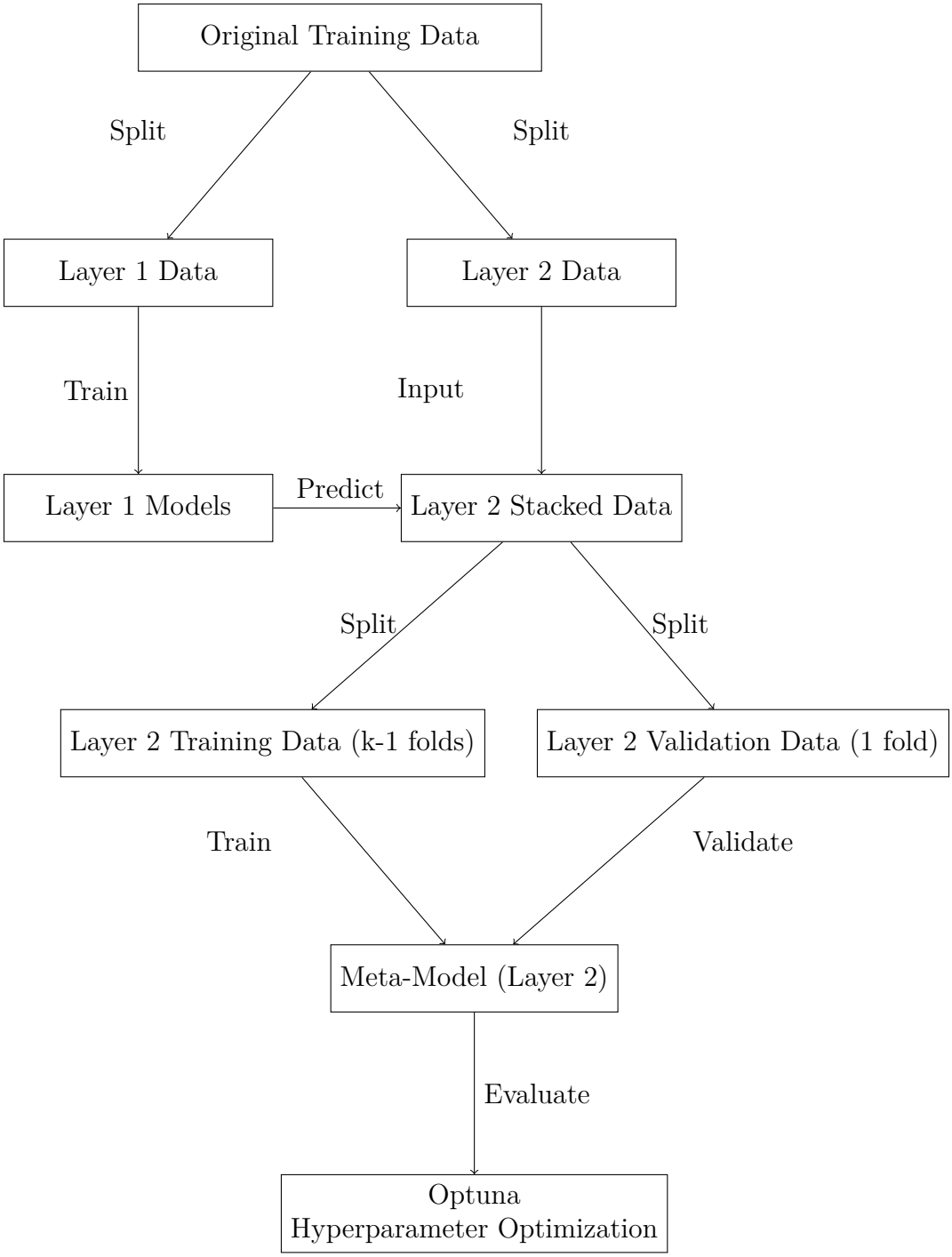


Figure 4: Data Partitioning and Cross-Validation for Two-Layer Model with Optuna Tuning

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.

These are average RSME results on new data (averaged across 5 instances of new data because of 5-fold cross-validation). I used Optuna to perform hyperparameter optimization which is designed to efficienctly search for the best hyperparameters given a user inputted range for each parameter. It does so intelligently rather than exhaustively.

Model Stack	**Best Parameters (RF Metamodel)**	**Score**
Deep RF, Boosting (1, 4, 8, 12 depths), Elastic	n_estimators: 3556, max_depth: 6, min_samples_split: 9, max_features: 0.3053	0.7519
Deep RF, Boosting (1, 4, 8 depths), Elastic	n_estimators: 2937, max_depth: 7, min_samples_split: 18, max_features: 0.1735	0.7518
Deep RF, Boosting (1, 4, 8 depths), Elastic, KNN Eucl	n_estimators: 2671, max_depth: 7, min_samples_split: 9, max_features: 0.2895	0.7514
Deep RF, Boosting (4, 8 depths), Elastic , KNN Eucl	n_estimators: 1830, max_depth: 6, min_samples_split: 2, max_features: 0.2905	0.7541

Table 3: Stacking Results Summary

A score in the table was calculated by training the layer 1 models, then using Optuna to tune the parameters of the layer 2 model which itself uses 5-fold cross-validation. This is a quite computationally intensive process given for each set of base models, you have to tune hyperparameters for the output model using cross-validation.

For the best model stack above (the third one), we show the Optuna hyperparameter tuning results. At this point, there is still a question of whether kNN should be added or different parameters should be used given the neglible difference in RSME with and without kNN.

I replaced the Euclidean Distance k=10 kNN model with the cosine metric instead and the best hyperparameters found in Optuna tuning received a score of 0.757 on new data which performed worse than all other layer 1 combinations.

Trial	Score	n_estimators	max_depth	min_samples_split	max_features
0	0.7563	1024	8	20	0.3594
1	0.7602	2117	9	6	0.6964
2	0.7549	1519	8	9	0.3677
3	0.7515	2293	7	4	0.1296
4	0.7551	3420	4	20	0.6047
5	0.7561	1022	8	6	0.6188
6	0.7578	3607	9	14	0.8058
7	0.7880	3820	2	9	0.9401
8	0.7624	1875	9	14	0.9324
9	0.7651	1566	3	20	0.9913
10	0.7533	4944	6	3	0.1309
11	0.7549	4945	6	2	0.1028
12	0.7551	2666	6	2	0.1176
13	0.7592	4563	5	5	0.2707
14	0.7523	2772	7	3	0.2518
15	**0.7514**	**2671**	**7**	**9**	**0.2895**
16	0.7572	2337	10	11	0.4729
17	0.7529	3128	7	9	0.2284
18	0.7536	4071	5	12	0.4871
19	0.7551	2500	7	17	0.3631
20	0.7594	3078	5	7	0.2112
21	0.7518	2809	7	4	0.2869
22	0.7533	2149	7	5	0.3166
23	0.7553	2877	8	4	0.4531
24	0.7521	3268	7	8	0.1872
25	0.7578	1806	10	7	0.4149
26	0.7644	2310	4	4	0.2835
27	0.7556	2731	6	12	0.1860
28	0.7569	4017	6	10	0.5230
29	0.7545	1410	8	14	0.3260
30	0.7536	2480	9	7	0.1546
31	0.7519	3290	7	8	0.1785
32	0.7515	3442	7	5	0.1809

Table 4: Layer 2 Parameter Tuning Results on Best Layer 1 Models

From these results, we create a plot of the average score for trials of depth d for $d \in [1, 7]$. From this, we see that with a relatively small number of models, the layer 2 metamodel is not incentivize to overfit, even if it can split on every model.

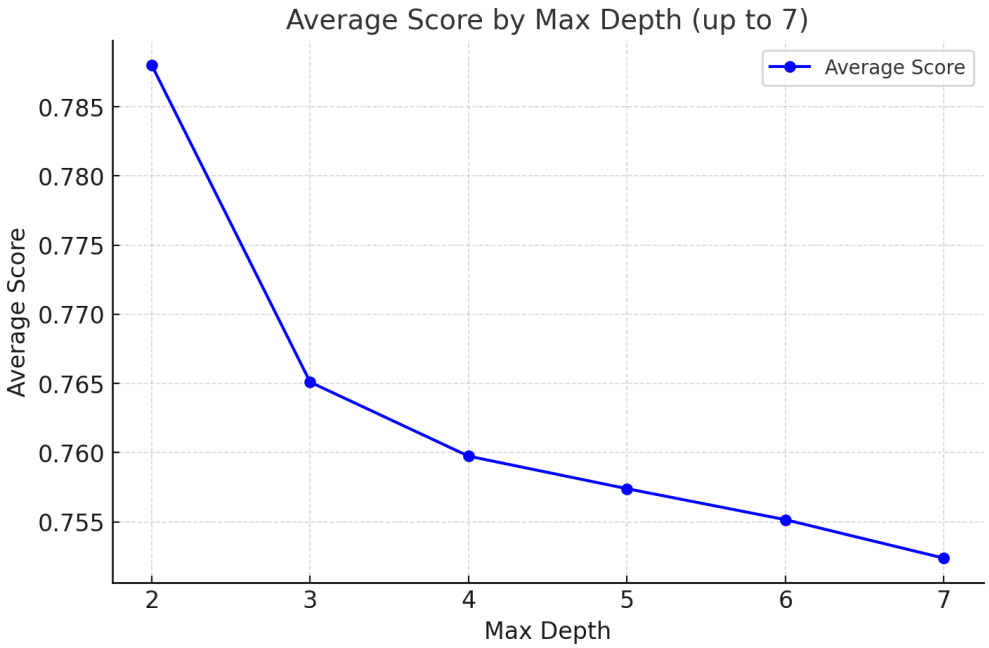


Figure 5: Average Score by Max Depth

Earlier, we had tried many different metamodels and Random Forest performed the best, but now that we have greatly simplified the number of layer 1 models, it is worth investigating how different metamodels would perform. We did this and performed cross-validation through Optuna tuning when applicable (e.g. not over linear regression).

Metamodel	**Cross-Validation Score**
Random Forest	0.7514
Linear Regression	0.7577
Logistic Regression	0.7934
Gradient Boosting	0.8290
Single Decision Tree	0.7905

Table 5: Performance of Different Metamodels

Now we see that there is a completely valid case for using Linear Regression as the metamodel. If I did this, it would be very interpretable given that the no more than half-dozen different columns fed in (model predictions) would all be on the same scale. Given the very small difference in RSME, I am opting to change my metamodel to be a linear model.

Now that we established the new metamodel, I wanted to further tune the hyperparameters of the layer 1 models. We decided to use two XGBoosts of depth 4 and 8, and a deeper Random Forest of depth 22 (which intuitively are nice and diverse), but what about other

hyperparameters? I wanted the number of estimators to be high to reduce the variance in predictions and make a variety of trees, but the one thing that was left untouched was the learning rate in boosting.

Learning Rate	**Value (Loss)**
0.3	0.7544
0.1	0.7425
0.06	0.7382
0.05	0.7336
0.04	0.7418
0.02	0.7477

Table 6: Performance of different learning rates in XGBoost trials.

The quoted value is the RSME from the output layer, not just the loss on the individual boosting model.

Thus, we choose a learning rate of 0.05 for both depths of XGBoost.

The equation of the linear metamodel with our current layer 1 models is:

$$Price = -0.0501 + 0.4806 \cdot \text{RForest} + 0.3721 \cdot \text{XGB\_depth4} + 0.1382 \cdot \text{XGB\_depth8} + 0.0397 \cdot \text{Elastic\_1}$$

As one would expect, the sum of coefficients roughly adds up to 1. For more interpretability, one could disallow an intercept term. From this, we can also see that Elastic regression is not that important in the model output and we could simply our layer 1 models if we desired.

I made one more change to bring up my score to the top of the leaderboard: added a few more diverse models at the expense of simplicity.

Sacrificing simplicity, I added kNN back in as well as a XGBoost with decision stumps. The goal with this the decision stumps in particular to capture some of the binary variables that were important on their own (like gym, pool, golf course, elevator, etc) that did not make it into the deeper models. This gave an RSME of 0.72167.

This was fine, but to try to boost my performance even more, I added a 4th XGBoost model, deeper than anything before. This boosted my final public leaderboard score to 0.71836 and also performed well in my local cross-validation.

The linear metamodel output for this final architecture is:

$$\begin{aligned}\text{Price} = &-0.0447 + 0.1609 \cdot \text{RF} + 0.1229 \cdot \text{XGB}_{\text{stump}} \\ &+ 0.3108 \cdot \text{XGB}_4 + 0.2982 \cdot \text{XGB}_8 + 0.1521 \cdot \text{XGB}_{12} \\ &- 0.0509 \cdot \text{ElasticNet} + 0.0352 \cdot \text{kNN}_{10}\end{aligned}$$

Although it is a bit more complex than the 4-model (or even 3-model) version, this weighted average of models is really interpretable and helps to suggest further optimization and changes to make to the stacking architecture. For one, removing ElasticNet and kNN would likely not have a large change in the model output. This would simplify it down to 5 layer-1 models.

However, for the purposes of the competition, I opted for the more complex version to capture potential marginal gains from every unique type of model, even if their contributions are small.

Lastly, I wanted to include a cross-validation result from a single decision tree that I used to get an intuition about how deep trees should be. I will display a depth 4 tree and its splits because anything bigger is too big to be displayed here. However, I also looked at the plots for bigger trees to better understand effective splits, even if this is not a globally optimal tree and deeper trees use the earlier depth as a prefix.

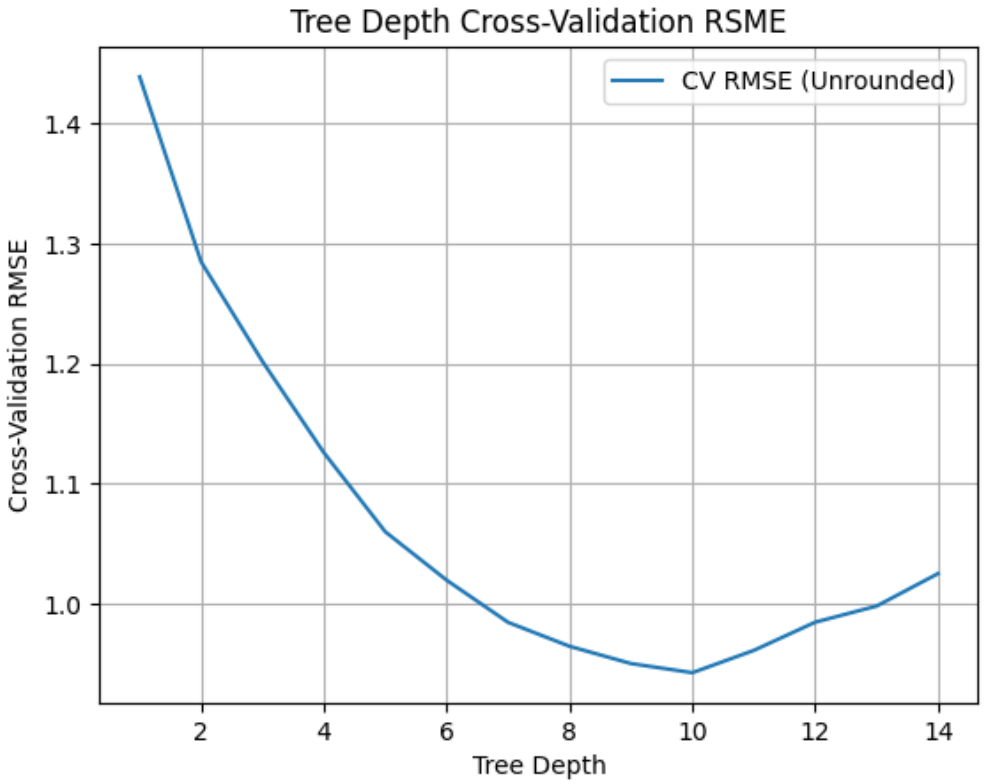
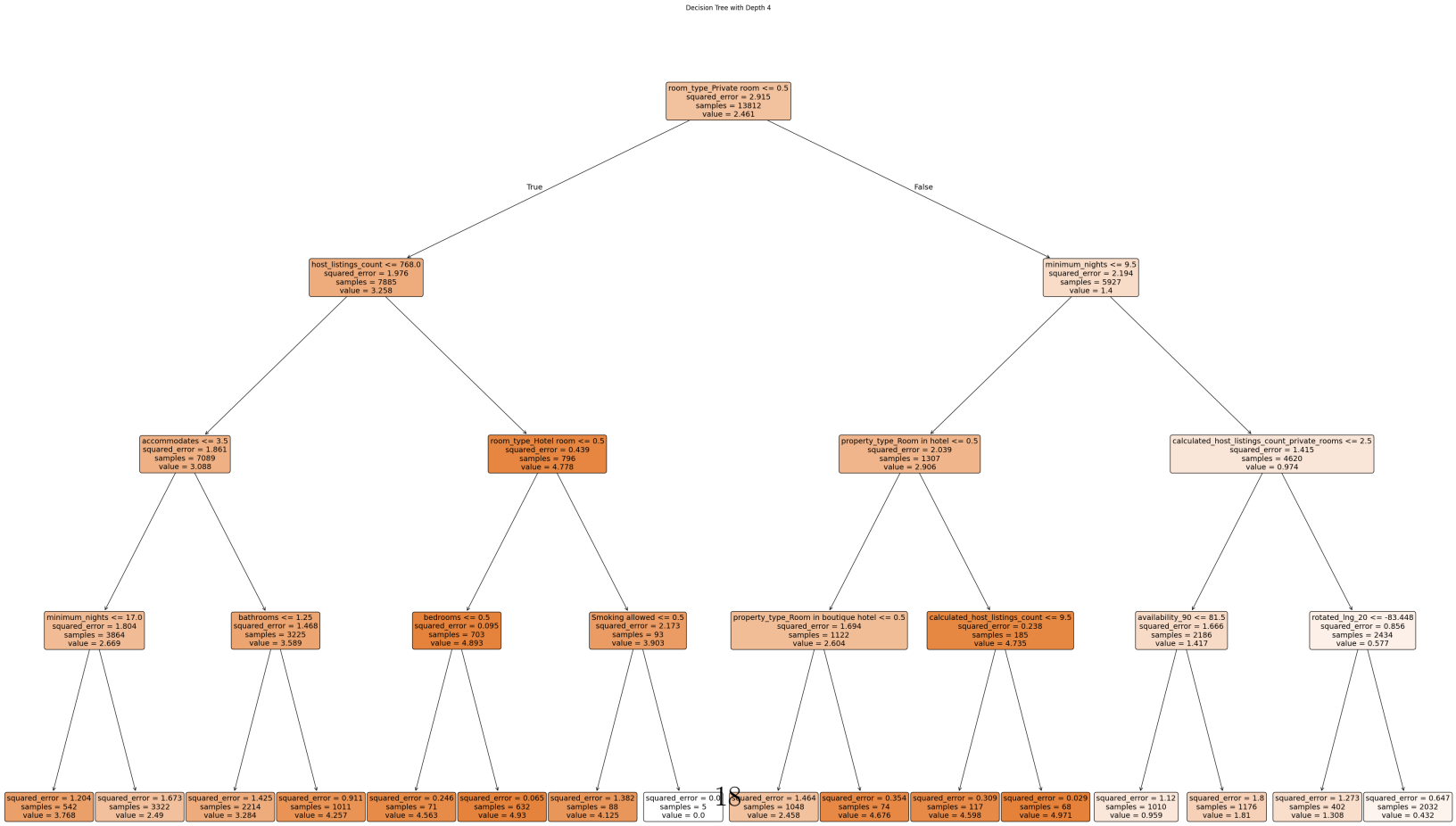


Figure 6: CV Result on Decision Tree Depth



5 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.

Random Forests randomly select features at each split to create decision trees on bootstrapped subsets of the data. The split criterion is a hyperparameter, but I am solving a regression problem and making greedy choices on splits to optimize for MSE. I am optimizing for the following (unrounded) expression:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})^2$$

Other alternatives are Gini impurity or entropy.

At each split, the algorithm will evaluate all the possible splits across the randomly selected features to choose from at that split and greedily choose the one that maximizes the reduction in MSE. In the Scikit-learn implementation, there is no post-pruning or pruning the tree after it is fully grown.

With only a few thousand trees (3200) trained greedily, it took no more than 90 seconds to train on standard hardware.

KNN stores the entire dataset in memory and makes predictions by identifying the k nearest points to a query data point using a distance metric. Here, we use Euclidean distance. The actual algorithm to carry out this task is selected based on data characteristics like size or number of features. Some of them include Ball Trees which organize points into nested hyperspheres, k-dimensional trees which are binary search trees, or brute-force pairwise distances.

This algorithm took a on the order of tens of seconds to train.

XGBoost or Gradient Boosting trains sequential decision trees where each tree corrects errors from previous ones by minimizing a loss function on what can be viewed as a reweighted dataset (higher priority to those that it missed in previous iterations). The default regression

loss function, which we are using, is

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

. It uses a second-order Taylor series expansion which uses the gradient and Hessian matrix to optimize the loss function (this is a generalization of trying to predict residuals or errors that the current model has made).

$$L(y, \hat{y}) \approx L(y, \hat{y}_t) + \frac{\partial L}{\partial \hat{y}}(\hat{y} - \hat{y}_t) + \frac{1}{2}\frac{\partial^2 L}{\partial \hat{y}^2}(\hat{y} - \hat{y}_t)^2$$

By default, XGBoost has regularization terms which I left in the model with the default settings.

$$\text{Objective} = \sum_{i=1}^{n} L(y_i, \hat{y}_i) + \sum_{t=1}^{k} \Omega(f_t)$$

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

Where T is the number of leaves and γ and λ are additional regularization terms.

This algorithm took 1 minute to train with 3200 trees of small/medium depth. In our final stacked ensemble, we used trained 4 different XGBoosts of depth 1, 4, 8, and 12. This took a total of 4 minutes.

Linear Regression has a closed-form solution with the biggest overhead being computing an inverse. Linear Regression minimizes the sum of squares on a dataset given the constraint that it must be a hyperplane. This algorithm ran instantly.

Elastic Regression on the other hand does not have a closed-form solution as it combines L1 and L2 regularization (L1 being problematic). L1 regularization encourages coefficients to be 0 and L2 regularization discourages large coefficients. To solve this optimization problem, Elastic Net uses gradient descent to iteratively update the coefficients and intercepts for a loss function dictated by given regularization hyperparameters.

$$L(\beta) = \|y - X\beta\|^2 + \lambda_2\|\beta\|_2^2 + \lambda_1\|\beta\|_1$$

This algorithm needs features to be scaled because the regularization is based on the coefficient size. For that, I used Standard scaling which takes each feature to a mean of 0

and standard deviation of 1.

Compared to linear regression, this algorithm took a large amount of time. It was took just over 1 minute.

All models were trained on a Windows 11 Laptop with an i7 Processor and 16GB RAM. All times are quoted as training one set of hyperparameters - without cross-validation.

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?

I found it difficult to know whether my feature engineering was helpful. I experimented with adding the average price for neighborhoods and neighborhood groups as well as the average price for the k nearest neighbors geographically - but these actually shockingly hurt my model performance. For the former, I deduced that it was because some neighborhoods were missing in the test data but nothing was missing in the training data so the algorithms didn't know there could be missing values, so when I encoded it as -1, that was handled for the first time in the test set. For the latter, it was tougher to say beyond a different sampling of data being in areas that were more spread out than the training data.

I also found it tough to know how many clusters to do for location, which led me to make what I discussed above. I want to restrict the model to think in only logical ways, so I give it only logical features, but the neighborhoods were too big and the neighborhood groups were too small (many of them weren't in the test data). If I used clustering, it tended to do an odd job and then I would have 70 clusters (or so) all one-hot encoded meaning models couldn't use the majority of clusters in one decision tree for example.

I also struggled with model complexity because initially my solution was to keep throwing models in my first layer until I did better, but then afterwards I realized that the majority of them weren't really helping and they were predicting very similarly.

I also found myself keep coming back to do more data pre-processing as I trained models. One primary reason for this was looking at feature importance on predictive models I trained and getting more ideas as to what could be clearly associated with higher or lower prices.

One of the biggest things I learned is how helpful interpretable machine learning models are from the perspective of someone fitting the models. It was really satisfying after I switched to a linear regression metamodel because I could see what base models were important, which

had minimal effects, and so on, so I could have a much better idea how to keep tuning the layer 1 model types and parameters in the stacked ensemble.

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:

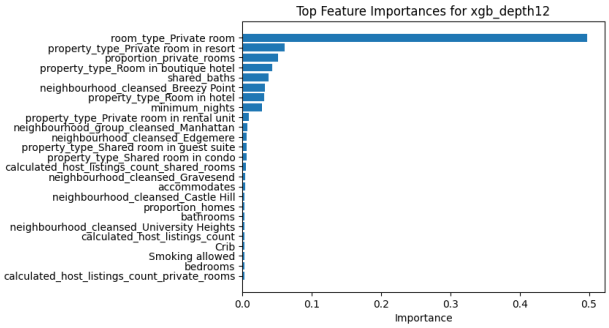
- **Fully Interpretable Model (e.g., Decision Tree, Generalized Additive Models):** Use of inherently interpretable models, like a decision tree or linear/logistic regression, that allow for clear reasoning about predictions.
- **Interpretable Feature Engineering Pipeline:** Creation of features that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

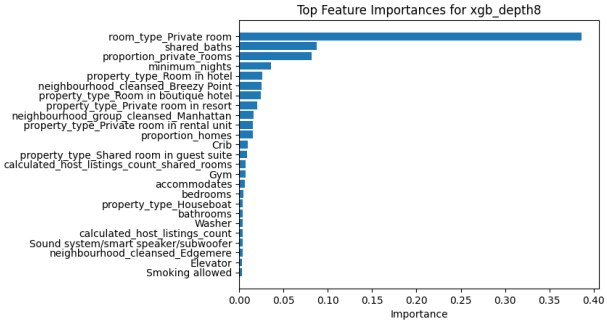
Username: Zakk Heile

Model	**Description**	**RMSE**
Random Forest (Depth = 22)	Deep ensemble trees	0.791
XGBoost (Stump)	Boosted stump	0.851
XGBoost (Depth = 4)	Boosted trees with shallow depth	0.750
XGBoost (Depth = 8)	Boosted trees with medium depth	0.742
Linear Regression	Simple linear model	1.010
Logistic Regression	Simple logistic regression model	1.151
Elastic Net	Regularized linear regression	0.997
MLP (Medium Hidden Layers)	Neural network with medium-sized layers	0.891
KNN (k = 10, Euclidean Distance)	k-Nearest Neighbors with Euclidean metric	0.998
Stacked Ensemble (Layer 1 + 2)	Linear metamodel	**0.71836** on Kaggle

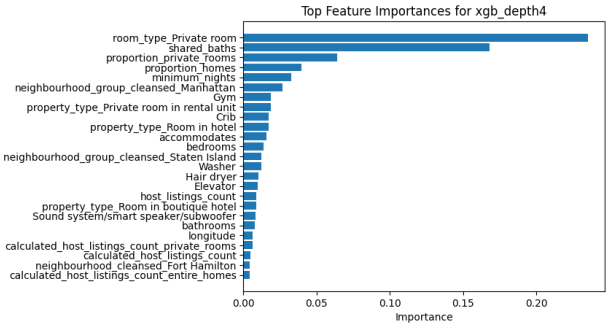
Table 7: Performance of Model Performance



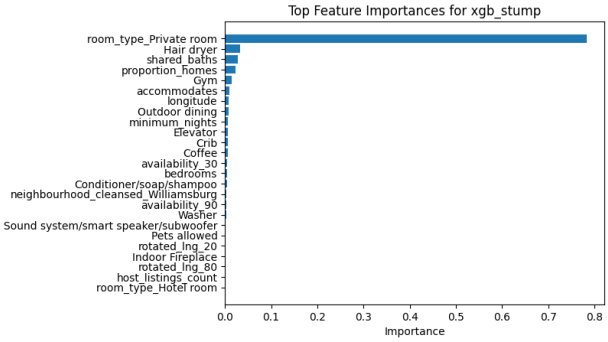
(a) XGB12 Model



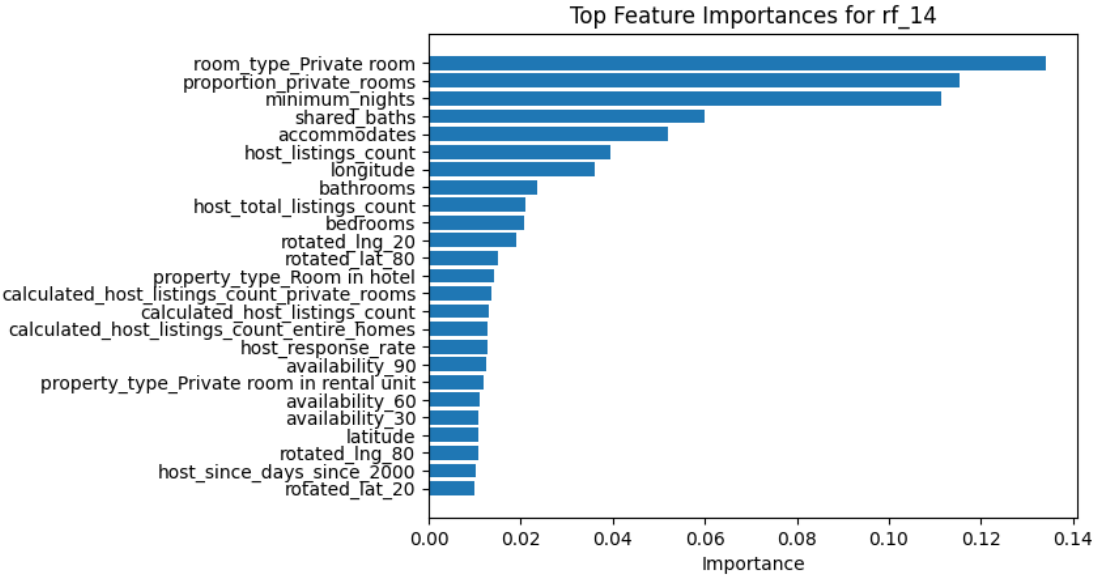
(b) XGB8 Model



(c) XGB4 Model



(d) XGB Stump Model



(e) Random Forest Model

Figure 8: Layer 1 Feature Importance (Tree-Based)

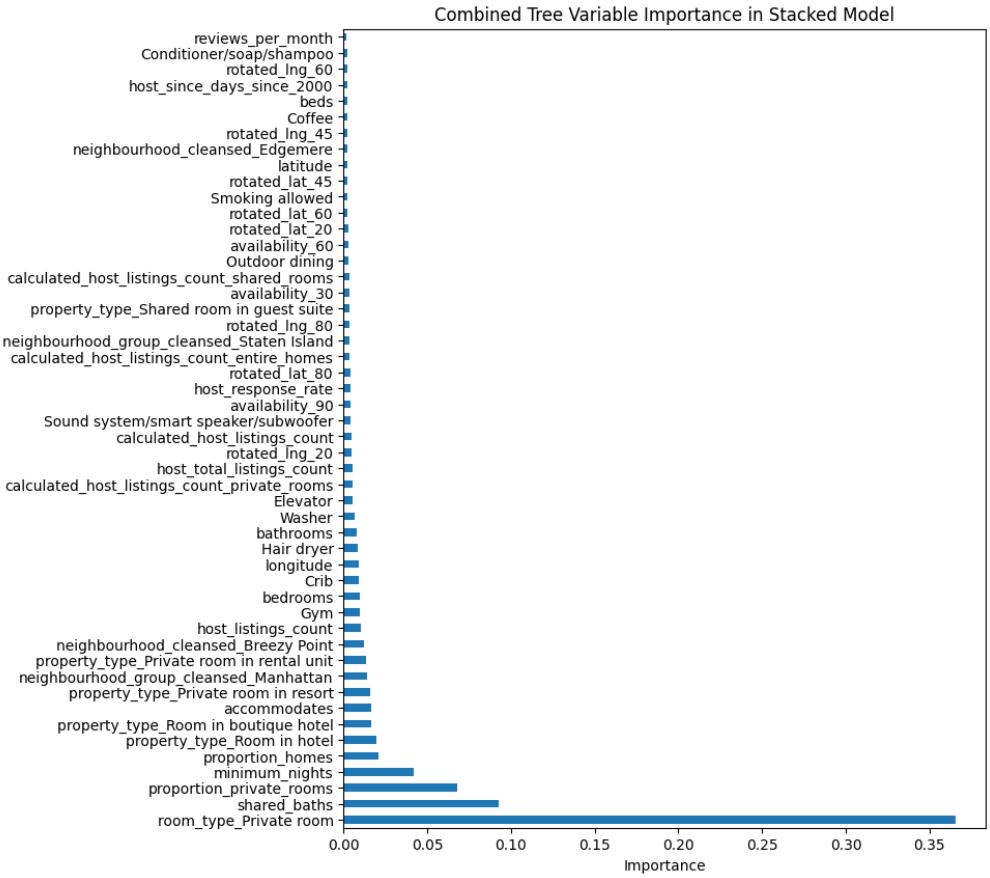


Figure 9: Weighted Tree Feature Importance by Metamodel Coefficient

This feature importance is weighted by the coefficients of the linear model, which is a very fair way to do it, however, this does not include the Elastic Regression model or kNN. For elastic regression, I could have used the coefficients, but then you get an interpretability issue because they are different feature importance and thus on different scales. kNN has no good feature importance.

Thus, while this explains the majority of the feature importance, there is still a small amount not explained, though it really is quite small given the small weights in the metamodel for kNN and elastic regression.

Next, we will assess variable importance with model reliance. We choose to scramble the layer 2 data, which contains the new predictions generated by the layer 1 models and is subsequently used to train the metamodel. We independently scramble each column of the layer 2 data, prompt the layer 1 models to generate new predictions, and feed these into the metamodel. The reported score in the table represents the difference in RMSE between the original and scrambled data, as predicted by the same metamodel.

Feature	Model Reliance Difference
minimum_nights	0.2577
accommodates	0.1429
room_type_Private room	0.1297
longitude	0.0878
shared_baths	0.0816
bedrooms	0.0606
host_response_rate	0.0501
host_listings_count	0.0477
host_total_listings_count	0.0436
rotated_lng_20	0.0381
availability_90	0.0293
calculated_host_listings_count_private_rooms	0.0244
calculated_host_listings_count_entire_homes	0.0234
rotated_lat_80	0.0230
rotated_lng_80	0.0216
host_since_days_since_2000	0.0202
Washer	0.0184
rotated_lat_60	0.0170
last_review_days_since_2000	0.0162
calculated_host_listings_count	0.0155
property_type_Private room in rental unit	0.0152
bathrooms	0.0145
review_scores_location	0.0130
availability_30	0.0127
rotated_lat_45	0.0112
reviews_per_month	0.0105
availability_365	0.0101
review_scores_communication	0.0087
review_scores_value	0.0083
latitude	0.0080
Crib	0.0080
Coffee	0.0076
review_scores_accuracy	0.0073
maximum_nights	0.0069
beds	0.0062
availability_60	0.0062
first_review_days_since_2000	0.0058
Hair dryer	0.0055
property_type_Private room in home	0.0047
rotated_lng_60	0.0040
Trash compactor	0.0040
review_scores_rating	0.0036
neighbourhood_cleansed_Forest Hills	0.0036
HDTV/TV/Television	0.0036
host_has_profile_pic	0.0033
number_of_reviews	0.0033
property_type_Private room in townhouse	0.0033
property_type_Room in boutique hotel	0.0029

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.

https://github.com/zakk-h/AirbnbPricesML

The GitHub repository for the project can be found at: https://github.com/zakk-h/AirbnbPricesML

9 Logistics

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