Kaggle Competition Writeup

Zachary Robers

Due: Nov 26th 2024

Kaggle Competition Link

Your Kaggle ID (on the leaderboard): zachrobers

1 Exploratory Analysis

Rather than biasing myself by looking at the accessible data, I began my investigation by answering the question: what would affect my willingness to pay for an Airbnb in New York City?

I arrived at four primary factors:

1. **Size:** How many people can stay in the Airbnb?
2. **Location:** Is the Airbnb located in a nice neighborhood with close proximity to businesses and tourist attractions?
3. **Timing:** Was I coming to the city for a particular event, holiday or season?
4. **Quality:** How nice is the property and what amenities does it offer?

Next, I turned to the dataset to see how the immediately accessible features incorporated information on these four primary factors.

1.1 Size

Of the four primary factors, I felt that the dataset most easily encompassed information on the size of the property. Once I reconciled the two different bathroom columns created by a switch from numerical to text representations and replaced any missing (NaN) cells with the average for the feature, size data could easily be accessed through the bedrooms and bathrooms features. Furthermore, a one hot encoding of the categorical feature "room type" offered additional insight on the layout of the space. I chose to one-hot encode "room type" over "property type" as there were only 4 different room type categories as opposed to 50+ property types. A reduced number of one-hot encoded categories reduces the risk of overfitting and reduced model performance from too many features.

1.2 Location

The dataset also provided sufficient information on location, although I felt as though location information was not best represented by the immediately accessible features. Latitude and Longitude coordinates carry very specific information, but in order for a model to hone in on on a specific neighborhood, it would need to be extremely complex: i.e. a decision tree would need to be very deep to split enough times on longitude and latitude to isolate small neighborhoods. Furthermore, the provided neighborhood groups seemed to be somewhat arbitrarily drawn without the most direct relevance to the price of an Airbnb.

These realizations motivated me to utilize clustering approaches to better incorporate locational information. In particular, I noticed the similarity of this problem to neighborhood analysis with single cell spatial transcriptomics data. The premise of neighborhood analysis in this context is that by looking at the distribution of the nearest neighbors to a given cell and then performing k-means clustering, one can group individual cells into biologically relevant "neighborhoods" - typically sections of tissue that function together in some fashion.

In essence, I extended this approach to finding actual neighborhoods with more relevance than the somewhat arbitrarily drawn neighborhoods provided in the given features. To do this, I followed the following feature/segmentation engineering pipeline:

1. **Normalization:** I normalized all of the features with min-max normalization across both the train and test datasets. I can use both the train and test datasets as clustering is an unsupervised technique in this context, meaning price labels are not necessary. Normalization is necessary for finding the nearest neighbors as otherwise certain features will have disproportionate influence.

2. **K-Nearest Neighbors (KNN):** I ran K-Nearest Neighbors on the Longitude and Latitude Data to find the K nearest properties geographically to each property.

3. **Neighbor Averaging:** Using these K neighbors, I created averaged neighborhood representations for each property by averaging each normalized feature over the k neighbors and creating a new data frame to store the averaged neighborhood representations.

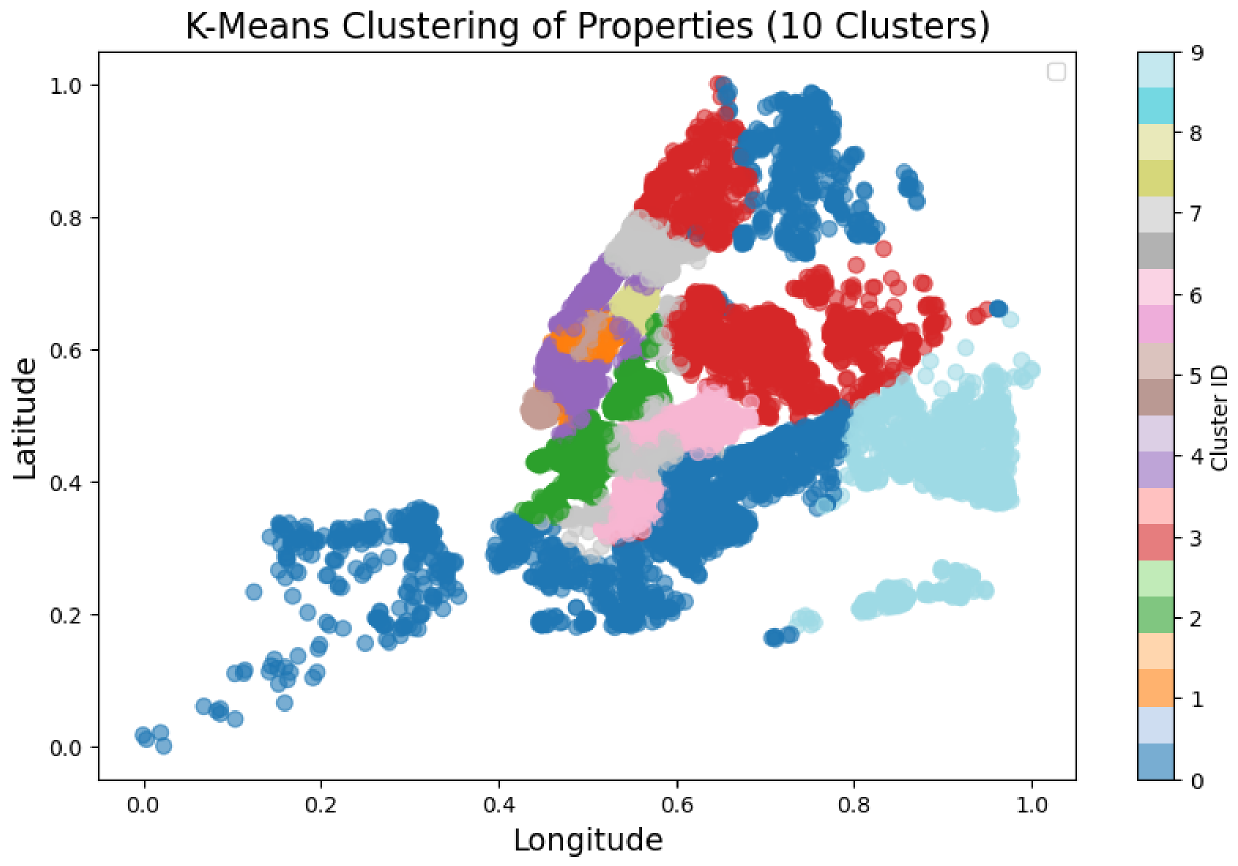

Figure 1: Result of k-means (k=10) clustering on averaged neighborhood representations found through KNN (k=500)

4. **K-Means Clustering** I performed K-Means Clustering on the averaged neighborhood representations to group the properties into K clusters.

It is important to note that directly applying K-Means Clustering would not be a valid approach to incorporating geographical data as the clusters would not naturally occur on geographic lines.

My first pipeline seeks to maximize interpretability by training individual trees within these clusters using the GOSDT library. The basis behind this idea is that by using unsupervised clustering first, one can create much less deep and thus more interpretable trees without greatly sacrificing performance as these trees no longer need to

My second pipeline uses one-hot encoding on the cluster data, treating each cluster as a better version of the original neighborhood feature. The nunber of neighbors in KNN and the number of clusters for K-means are treated as hyperparameters tuned during grid search cross-validation.

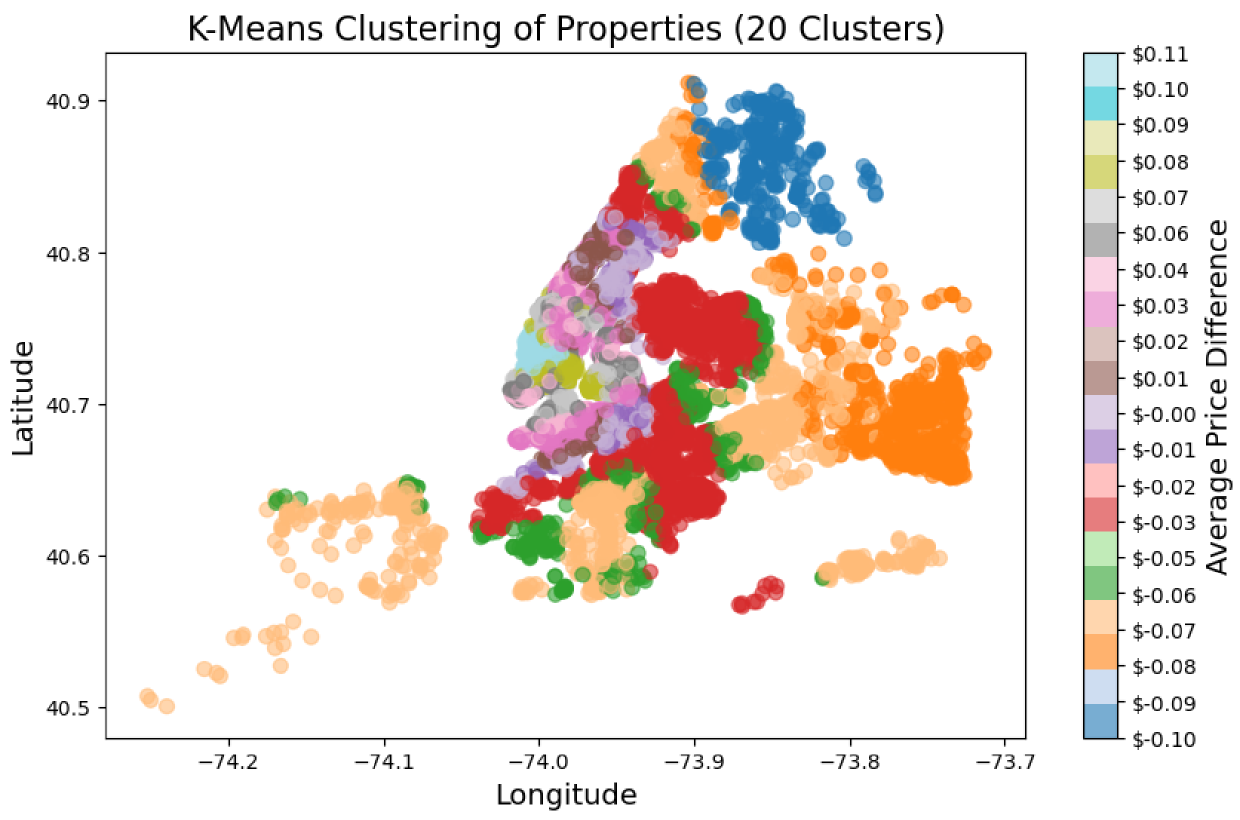

Figure 2: Result of k-means (k=20) clustering on averaged neighborhood representations of price difference (actual - predicted) found through KNN (k=500)

For my last approach (pipeline 3), I sought a way to incorporate all of the cluster information into a single feature as I feared that the one-hot encodings were leading to overfitting. This new approach is no longer unsupervized as the clustering is performed by looking at the locational disparity of a model trained without any locational information. Instead of averaged neighborhood representations with all of the features, I averaged the difference between the predicted price by an XGBoost Regressor without locational data and the actual price category across the neighborhoods and then performed clustering on these neigborhood-average price disparities. Instead of one-hot encoding these clusters, I created a new feature to represent the average price disparity (actual - predicted) for a given property's cluster.

Interestingly, this method of clustering displays a radial pattern with the highest price difference near Times Square in Manhattan and lower price differences further from the middle of Manhattan. This aligns with preconceived notions of where property prices are most expensive in New York City.

Since this clustering technique requires labels (no longer fully unsupervised), properties in the test dataset must be mapped to a cluster using a nearest neighbors majority vote.

1.3 Timing

Timing information proved to be difficult to incorporate based on the provided data. Ultimately, we seek to determine whether a property owner tends to only place the property on Airbnb at times in which the price may be higher (e.g. summer, holiday season, for sports games or concerts etc.). Since this data is not easily accessible, the hope is that it can be provided through some tangentially related features. I converted the dates for 'host since', 'first review', and 'last review' to datetime objects and then put them on a numerical scale so that they would be compatible with normalization necessitated by the processes described in the preceding section. Furthermore, I included significant information on the host and their practices (e.g. response time) as well as the number of reviews and the property's availability in hope of gauging whether the property is rented year round or only available for the most lucrative dates. Once more, I resolve NaN occurrences by utilizing averages where appropriate. In the case of no value for the datetime features, I replaced these missing values with the present date as a standard practice for indicating that the property has yet to be reviewed.

1.4 Quality

Quality information is largely incorporated in the one-hot encoding of the 'room type' and can be inferred from neighborhood clusters. However, seeking to identify additional factors affecting how nice the property may be, I also created one-hot encodings of 10 amenities that I determined from personal experience to have a significant impact on how nice a property is.

2 Models

As mentioned in 1.2, I utilized K-Nearest Neighbors and K-Means Clustering to generate neighborhood clusters. As justified previously, it is necessary to use K-Nearest Neighbors on the locational data to create averaged neighborhood representations as otherwise clusters would not naturally appear on geographic lines. The sklearn.neighbors library provides an efficient implementation to find the nearest neighbors for a property. I selected K-means cluster to generated the clustering rather than hierarchical clustering because the data naturally forms "ball-like" groups rather than more complex clusters which can more easily be extracted through an agglomerative technique. Once more, sklearn provides an easy to use and efficient implementation of K-Means Clustering.

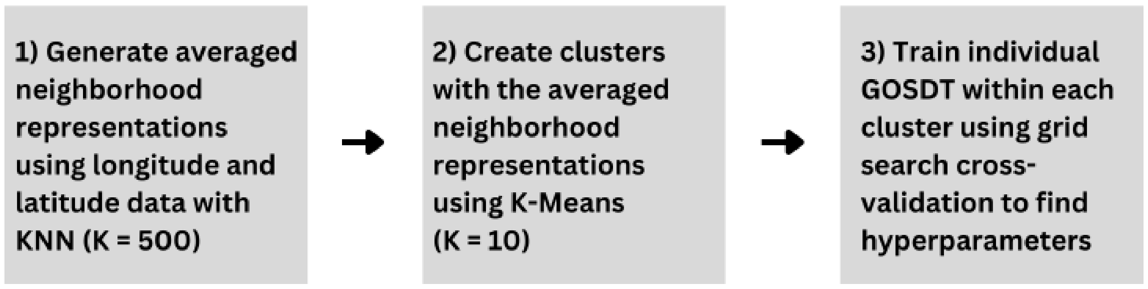
1) Generate averaged neighborhood representations using longitude and latitude data with KNN (K = 500)
2) Create clusters with the averaged neighborhood representations using K-Means (K = 10)
3) Train individual GOSDT within each cluster using grid search cross-validation to find hyperparameters

Figure 3: Pipeline 1

As for my predictive models, I chose to apply Generalized Optimal Sparse Decision Trees (GOSDT) within each cluster as my first approach primarily due to the high interpretability of this method and the provable optimality of GOSDT under a sparsity constraint. Since the clustering removes the nuance of having to split many times on latitude and longitude to isolate particular geographic regions, I felt that it would be more viable to predict using a single tree. GOSDT provides a single tree for each cluster and is unlikley to overfit because it incorporates a penalty for additional leaves in its loss function. A single tree provides clear rationale for why a property is predicted to be priced in a certain way. I utilized the GOSDT library, created by Ilias Karimalis, which I found to be very slow, likely due to the poor parallelizability of the GOSDT training process.

Pipeline 1 trains a Single GOSDT on each of 10 clusters generated using the clustering procedure described in 1.2.

My choice for a less interpretable but more performant model was Extreme Gradient Boosting (XGBoost). I wanted to attempt to use a boosting technique because I feel that it offers a logical next step to enhancing single tree methods with methodology that is still easily understandable. I selected XGBoost over AdaBoost due to its higher computational efficiency and flexibility in handling larger datasets. XGBoost uses gradient-based optimization to minimize the loss function directly, which generally results in faster convergence and better performance compared to AdaBoost's iterative adjustment of weights on misclassified samples. The xgboost library, created by Tianqi Chen and Carlos Guestrin, provides an efficient implementation and makes it straightforward to tune parameters and integrate into workflows.

It was initially unclear to me whether an XGBoost Classifier or XGBoost Regressor with

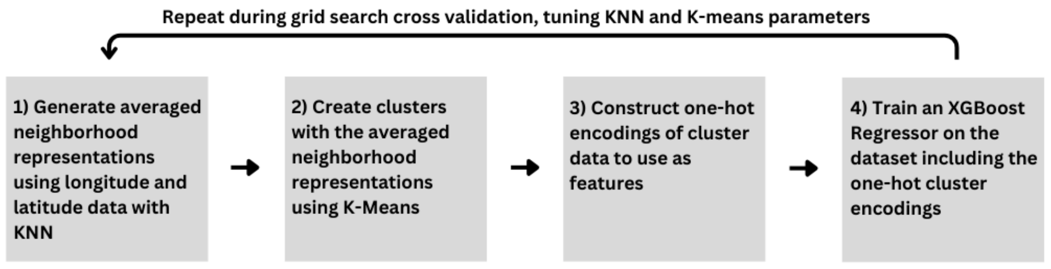
Repeat during grid search cross validation, tuning KNN and K-means parameters
1) Generate averaged neighborhood representations using longitude and latitude data with KNN
2) Create clusters with the averaged neighborhood representations using K-Means
3) Construct one-hot encodings of cluster data to use as features
4) Train an XGBoost Regressor on the dataset including the one-hot cluster encodings

Figure 4: Pipeline 2

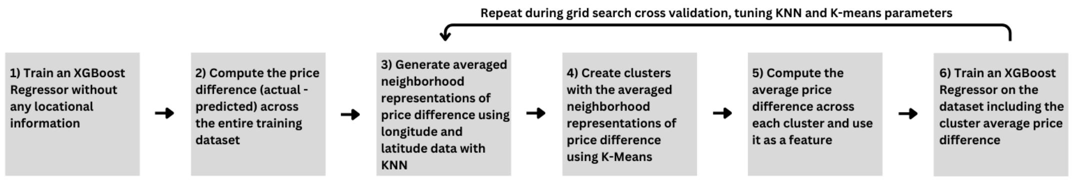
Repeat during grid search cross validation, tuning KNN and K-means parameters
1) Train an XGBoost Regressor without any locational information
2) Compute the price difference (actual - predicted) across the entire training dataset
3) Generate averaged neighborhood representations of price difference using longitude and latitude data with KNN
4) Create clusters with the averaged neighborhood representations of price difference using K-Means
5) Compute the average price difference across each cluster and use it as a feature
6) Train an XGBoost Regressor on the dataset including the cluster average price difference

Figure 5: Pieline 3

rounding to the nearest class would prove to be the most performant. On one hand, the data clearly falls into 6 classes (0-5). However, the fact that these price scales are ordered in sequential fashion with a higher class indicating a higher price ends up making regression with rounding to the nearest class a better performing approach. For this reason, I utilized the XGBoost Regressor for my final model.

Pipeline 2 creates one-hot encodings of cluster data and trains an XGBoost Regressor on this data. The number of neighbors used in KNN and number of clusters in k-means are optimized as if they are hyperparameters to find the beat way of incorporating the cluster data.

Motivated by a desire to reduce the number of features (see 1.2), **Pipeline 3** trains an XGBoost Regressor without any location data and then computes the difference between the value predicted by the regressor and the price label for each property in the training set. It then uses the KNN and K-means procedure to cluster properties with longitude and latitude data based on the difference between actual price and predicted price (from the model trained without location data). The average price difference for a property's cluster is then used as a single feature incorporating locational impact on price. Once more, the number of neighbors used in KNN and number of clusters in k-means are optimized as if they are hyperparameters.

3 Training

3.1 K-Nearest Neighbors

K-Nearest Neighbors (KNN) is a simple algorithm used for classification and regression, oftentimes relying on a majority vote or average label across the nearest neighbors. In this setting, KNN is simply used to find the nearest neighbors rather than to perform any kind of classification or regression. The algorithm searches for the k other properties closest to each property with the longitude and latitude coordinates, using euclidean distance as its distance metric. The sklearn implementation finds these nearest neighbors by selecting between three approaches depending on the structure of the dataset. The brute force method computes the distance between every pair of points, ensuring accuracy but scaling poorly with larger datasets. In contrast, tree-based methods like KD Tree and Ball Tree organize the data into hierarchical structures, allowing for faster neighbor searches by pruning irrelevant regions of the space. The wall time for KNN on the longitude and latitude data on the combined train and test datasets is approximately 12 seconds.

3.2 K-Means Clustering

K-Means is an unsupervised clustering algorithm that divides data into k clusters. It requires normalized data to ensure that certain features do not have a disproportional impact on the clusters. The algorithm starts by randomly selecting k centroids, then assigns each point to the nearest centroid. After assigning points, the centroids are updated as the mean of the points in each cluster. This process repeats until the centroids converge or the algorithm reaches a specified number of iterations. The choice of k is typically determined through methods like the elbow method, though in this setting k is tuned as if a hyperparameter during grid search cross validation. The wall time for K-Means averaged around 0.4 seconds on the combined train and test dataset using the sklearn implementation.

3.3 Generalized Optimal Sparse Decision Trees (GOSDT)

GOSDT finds a single sparse decision tree that minimizes the following objective:

$$R(tree, \mathbf{X}, \mathbf{Y}) = l(tree, \mathbf{X}, \mathbf{Y}) + \lambda k$$

where k is the number of leaves such that the depth is less than max depth D and l(loss) is the misclassification error. GOSDT trains using dynamic programming, treating each possible split as a subproblem and subproblems are indexed by a bit vector representing the data points involved in the subproblem. The training process is complete once the master problem is solved, but this first requires solving the smaller subproblems (subsets of the complete dataset) beneath it. A priority queue is used to order the subproblems represented by their bit vectors. A dependency graph is used to store the parent-child relationships between subproblems along with upper and lower bounds on the objective. During training, the upper and lower bounds are repeatedly updated up the dependency graph by checking if adding a new node can decrease the upper bound on the objective, until the upper bound on the objective matches the lower bound. This process is continued until all subproblems have been solved, converging on a single optimal tree for the provided objective. The wall time averages around 3 minutes for creating a single tree for classifying properties within a particular cluster.

3.4 Extreme Gradient Boosting (XGBoost)

XGBoost is an ensemble learning method that builds decision trees sequentially, where each tree corrects the errors of the previous one. It starts with a simple model, typically a constant prediction and then computes the the error (residuals) for this simple model. In the case of regression, the residuals are calculated simply by subtracting the predicted value from the actual value. An XGBoost Regressor then fits a new decision tree to the gradient of the loss function with respect to the predictions of the previous tree, aiming to minimize the error further. XGBoost combines the gradient and second derivative of the loss, found through a second-order Taylor expansion, to compute the gain (loss reduction) of a potential split. It also adds in regularization to prevent trees from becoming too complex (and thus over-fit to the training data). Each new tree's predictions are incorporated to adjust the previous tree with impact scaled by the learning rate. The XGBoost library trains with a wall time of approximately 30 seconds on the Airbnb dataset.

4 Hyperparameter Selection

In all of my models, I utilized grid-search cross validation to find the best combination of hyperparameters. Since the competition was scored on Root Mean Squared Error (RMSE), I selected the hyperparameters by looking for the combination that produces the lowest RMSE.

As elaborated on in 5, I split the data set into five folds and rotated which fold served as the validation dataset, training on the other four folds. I then averaged the RMSE across these 5 arrangements to gather a more accurate picture of the RMSE for a given parameter combination.

4.1 Pipeline 1

Pipeline 1 trains individual decision trees within each cluster, meaning that the grid search process occurs within each cluster. Efficiency limitations on the GOSDT library greatly limited the size of the hyperparameter grid and required that the max depth be limited to 3 for all trees to make runtime reasonable (this grid still takes around 11 hours to run locally on all of the 10 clusters).

Parameter	**Values**	**Parameter**	**Values**
`Regularization`	$\{0.01, 0.03, 0.05\}$	`Balance`	$\{$True, False$\}$

Table 1: Grid of parameter combinations for Pipeline 1.

As for the best parameters, every cluster achieved its lowest RMSE with 0.01 Regularization and no balance (False).

4.2 Pipeline 2

For Pipeline 2, the number of neighbors used to form the averaged neighborhood representations and the number of clusters used to cluster on the averaged neighborhood representations were treated as if they were hyperparameters to find the best way to represent the cluster data. These variables were changed in addition to the most consequential standard hyperparameters for XGBoost (learning rate, max depth, and number of estimators), yielding the following table of 324 hyperparameter combinations.

Parameter	**Values**	**Parameter**	**Values**
`knn_neighbors`	$\{100, 400, 1000\}$	`kmeans_clusters`	$\{1, 5, 10, 20\}$
`learning_rates`	$\{0.01, 0.1, 0.3\}$	`max_depths`	$\{8, 10, 15\}$
`n_estimators`	$\{100, 500, 1000\}$		

Table 2: Grid of parameter combinations for Pipeline 2.

The hyperparameter combination that resulted in the lowest RMSE is displayed in the table below.

Parameter	**Value**
`knn_neighbors`	1000
`kmeans_clusters`	10
`learning_rate`	0.1
`max_depth`	8
`n_estimators`	500
Final Best RMSE	0.7845074004346797

Table 3: Best hyperparameters for Pipeline 2.

Below are charts of the functional relation between predictive accuracy and each hyperparameter while keeping the other hyperparameters constant at their best value. Accuracy is found by splitting the training dataset 70-30, training on the larger portion, and predicting on the smaller portion.

While standard hyperparameters (learning rate, max depth, and number of estimators) appear to peak at a value in the middle of the tested range, suggesting a local maximum, the number of neighbors used in the KNN to find averaged neighbor representations and number of clusters found with K-means don't demonstrate a clear trend. This result is likely due to the delicate balance between adding features (one-hot encoding columns) and supplying more information to the model.

4.3 Pipeline 3

Initial experimentation led me to slightly modify the hyperparameter grid for Pipeline 3. This pipeline still refines the clustering process but now the clustering data is stored in a single feature as previously described.

Parameter	**Values**	**Parameter**	**Values**
`knn_neighbors`	{100, 400, 600, 800}	`kmeans_clusters`	{5, 10, 20, 40}
`learning_rates`	{0.01, 0.1, 0.3}	`max_depths`	{8, 10}
`n_estimators`	{500, 1000}		

Table 4: Grid of parameter combinations for Pipeline 3.

The hyperparameter combination that resulted in the lowest RMSE is displayed in the table below.

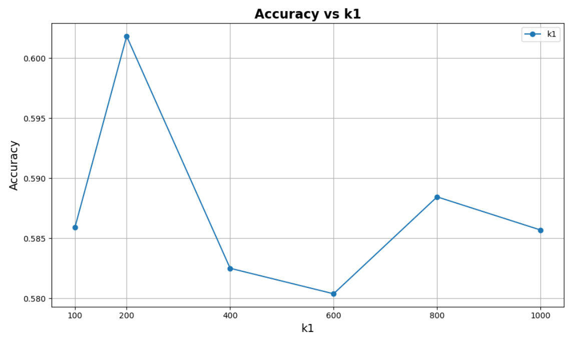

Figure 6: Accuracy vs Number of Neighbors

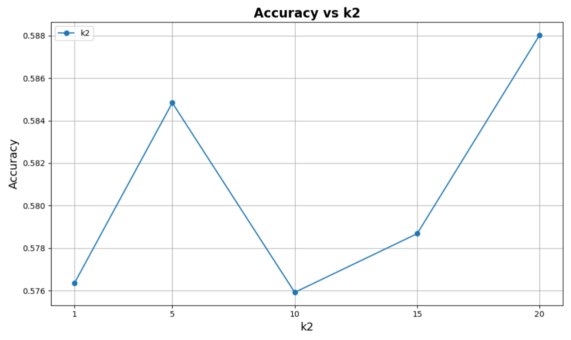

Figure 7: Accuracy vs Number of Clusters

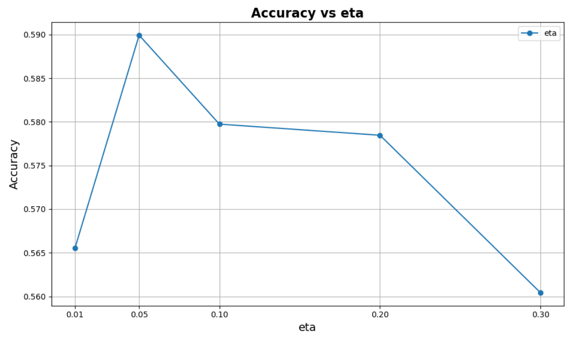

Figure 8: Accuracy vs Learning Rate

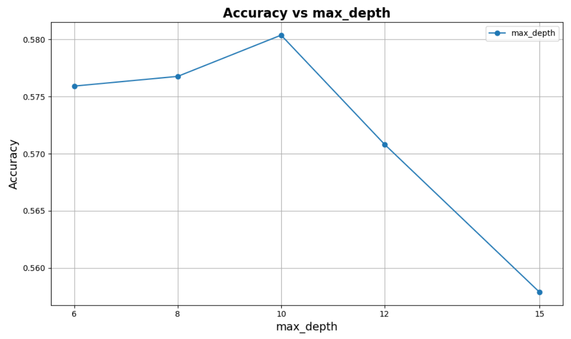

Figure 9: Accuracy vs Max Depth

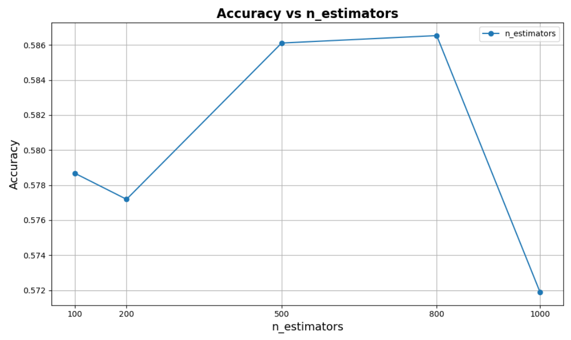

Figure 10: Accuracy vs # of Estimators

Parameter	**Value**
`knn_neighbors`	600
`kmeans_clusters`	40
`learning_rate`	0.1
`max_depth`	8
`n_estimators`	500
Final Best RMSE	0.7899443813479874

Table 5: Best hyperparameters for Pipeline 3.

5 Data Splits

During cross validation for each pipeline, I split the data into 5 folds (equal random partitions). For each parameter combination being tested, the model is trained on four of these folds (the training set) and validated on the remaining fold (the validation set). This process is repeated five times, with each fold serving as the validation set exactly once.

For each pipeline, this ensures that every data point in the dataset contributes to both the training and validation processes, reducing the risk of overfitting to a specific subset of the data. Finally, the RMSE is averaged across all five iterations to obtain a more robust estimate of the model's effectiveness for the given parameter combination.

When I trained each model to predict on the test dataset, I trained with all of the training data to reduce overfitting and maximize the model's performance.

6 Reflection on Progress

One of the most challenging aspects of this project was working with the GOSDT library. I was very excited to try implementing GOSDT as I was motivated by my success in clustering to think it was feasible to create a single sparse tree to model pricing within each clsuter. However, the library was very difficult to integrate with the sklearn framework and extremely long runtimes meant that I could not explore as large of a hyperparameter grid as I wanted. In particular, trees with a max depth of 4 or greater proved too computational expensive to find locally, forcing me to constrain all trees to a max depth of 3. I think if there were a better equipped GOSDT library, it would be possible to generate trees within each cluster with more comparable performance to other methods.

Another challenging aspect of this project was finding the best way to integrate the clustering information found through the neighborhood analysis described in 1.2. In Pipeline 2, I deploy one-hot encoding of the cluster information while still supplying the longitude and latitude data. Since I feared that this distributed locational information across too many features, I attempted to combine locational influence on price into a single feature in pipeline 2, but I found this to perform less well likely due to the information loss by reducing the cluster information to price influence. I still think that there is a better way to create clusters and incorporate geographical data.

7 Predictive Performance

Kaggle username: zachrobers

7.1 Pipeline 1

As expected, this purely interpretable approach performed the worst of the pipelines. However, this is certainly in part due to library efficiency limitations. Trees with depth greater than 3 would have yielded lower RMSE scores for each cluster. Nevertheless, the RMSE scores for the best hyperparameters on each cluster in the table below are remarkably comparable to less interpretable techniques.

Cluster	**Best RMSE**
Cluster 0	1.4189
Cluster 1	1.2910
Cluster 2	1.5074
Cluster 3	1.2561
Cluster 4	1.4527
Cluster 5	1.3552
Cluster 6	1.2137
Cluster 7	1.2487
Cluster 8	1.2076
Cluster 9	1.1337

Table 6: Best RMSE: GOSDT for each cluster

Interestingly, there is a positive correlation between the number of properties in a cluster and the RMSE, meaning that the GOSDT classifiers perform worse with larger training sets. This confirms the hypothesis that sparse decision trees would be more capable of capturing the logic behind property pricing when confined to a neighborhood of shared property characteristics.

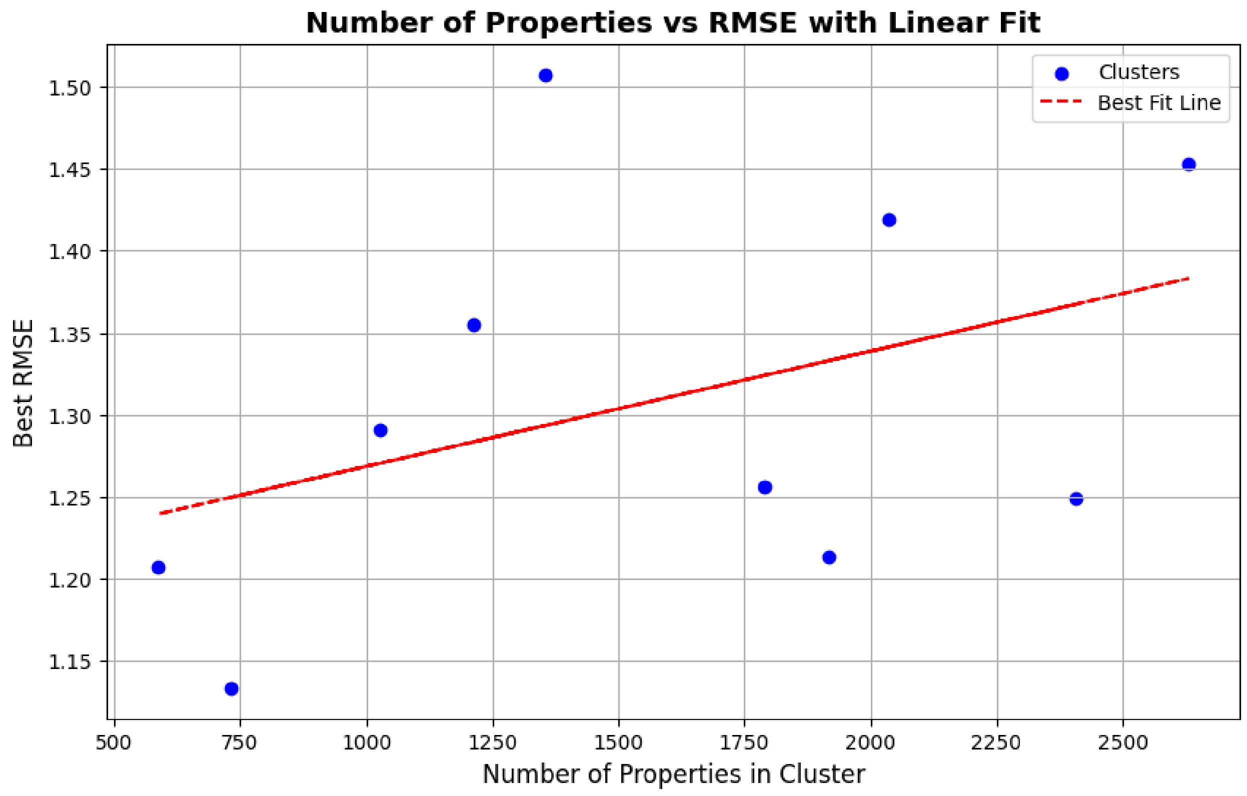

Figure 11: Number of Properties in Each Cluster vs Root Mean Squared Error

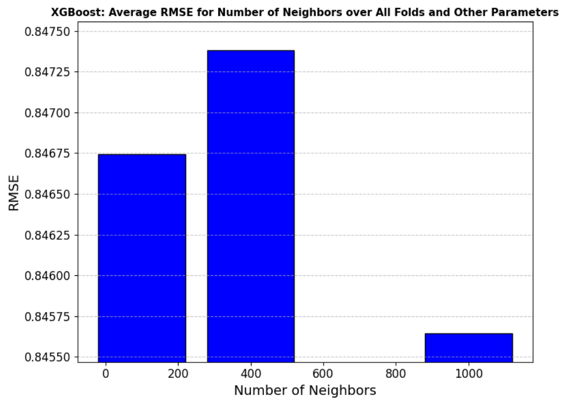

Figure 12: Number of Neighbors vs RMSE

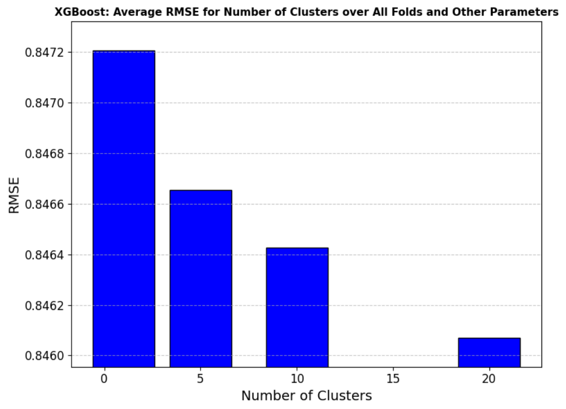

Figure 13: Number of Clusters vs RMSE

7.2 Pipeline 2

Pipeline 2 was the best performing of these approaches and accordingly will be analyzed most extensively here. With the best hyperparameters, Pipeline 2 achieves a root mean squared error of 0.7845 averaged across the five folds. The bar charts below show relationship between the clustering hyperparameters and RMSE.

Unlike the accuracy charts (??), these charts show a more clear trend of the impact of clustering. It appears that the added information from creating more clusters, although this information creates more features for the one-hot cluster encodings, causes the model to perform better.

The next figure displays a 6x6 confusion matrix comparing the predictions of the model with the best hyperparameters to the actual data using a 70-30 train-test split.

As expected, the diagonals, representing correct predictions, are the most frequent followed by predictions that are off by one in either direction. It seems that the model struggles most with correctly predicting on some of the middle price classes, often overestimating the price of properties in classes 1 and 2.

7.3 Pipeline 3

This attempt at reducing the feature count while still incorporating locational data proved largely unsuccessful. Pipeline 3 achieved a minimum root mean squared error slightly above that of Pipeline 2 with a score of 0.7899 with the best hyperparameters. This means that the lost information from no longer expressing the clusters themselves as one-hot encodings but as the average price difference within a cluster proved more damaging then any gain in

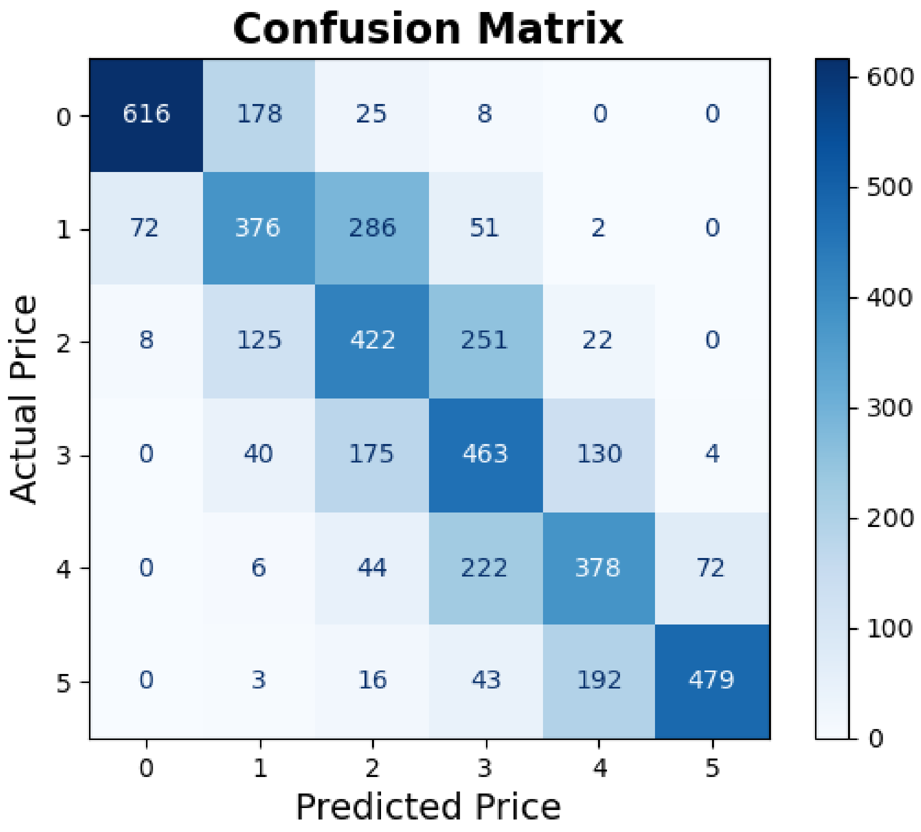

Figure 14: Confusion Matrix for Pipeline 2 with Best Hyperparameters

performance from reducing the feature count.

7.4 Feature Importance Analysis on Most Performant Model

For the most performant model (Pipeline 2 with hyperparameters given in 4.2), the most important features according to gain (or the reduction in loss across all splits in which the feature is used) are displayed in the table below.

Feature	**Importance (Gain)**
room_Private room	325.27
room_Entire home/apt	64.37
minimum_nights	47.26
calculated_host_listings_count	9.63
accommodates	7.50
room_Shared room	6.88
bathrooms	6.25
host_total_listings_count	6.16
bedrooms	6.11
beds	4.62

Table 7: Top 10 Most Important Features

For the most part, these features fall nicely into the four key factors for Airbnb pricing I identified before approaching the dataset (Size, Location, Timing, and Quality). The room type (private room vs entire home/apt vs shared room) is a direct indicator of the quality of one's stay. It is much more comfortable to have privacy, explaining why there is such a profound impact on pricing between these room types. Furthermore, the number of people the Airbnb accommodates and the number of bathrooms, bedrooms, and beds directly incorporate information on the size of the property. Most often, people are willing to pay more for a larger property. The minimum nights and host listing features are slightly more puzzling, but I suspect that they are indirect indicators of both quality and timing.

Notably absent from the top 10 features is any information on the location of the property. For that reason, I display the importance of the one-hot encodings for each cluster type in the table below.

I think the primary reason that the clusters have lower importance than some of the other features is that the locational information is spread out across all 10 features. Furthermore, I suspect the disparity of importance between cluster encodings has to do with whether there

Cluster Feature	**Importance (Gain)**
cluster_4	3.79
cluster_7	2.13
cluster_5	1.30
cluster_6	1.13
cluster_0	1.06
cluster_2	0.79
cluster_9	0.79
cluster_1	0.74
cluster_8	0.64
cluster_3	0.61

Table 8: Cluster Importance Features

is a pricing premium or discount for that cluster. For instance, a cluster centered in the upper east side likely contains more information as properties in that area are significantly more expensive than the median property, meaning that cluster has greater impact on the label.

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.

9 Logistics

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

References

[1] Hickey, J.W., Becker, W.R., Nevins, S.A. et al. Organization of the human intestine at single-cell resolution. *Nature*, **619**, 572–584 (2023). https://doi.org/10.1038/s41586-023-05915-x

[2] Rudin, Cynthia. *Intuition for the Algorithms of Machine Learning.* Self-published, eBook, 2020.

[3] Chen, T., & Guestrin, C. XGBoost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–794 (2016).