Template for In-Class Kaggle Competition Writeup

CompSci 671

Due: Dec 5th 2023

Kaggle Competition Link

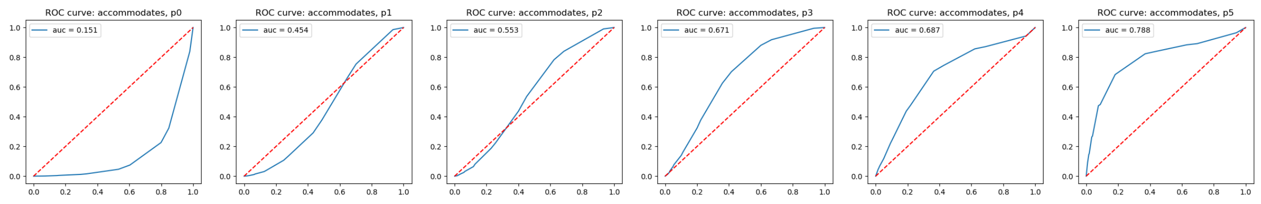
Your Kaggle ID (on the leaderboard): Haiyan Wang

1 Exploratory Analysis

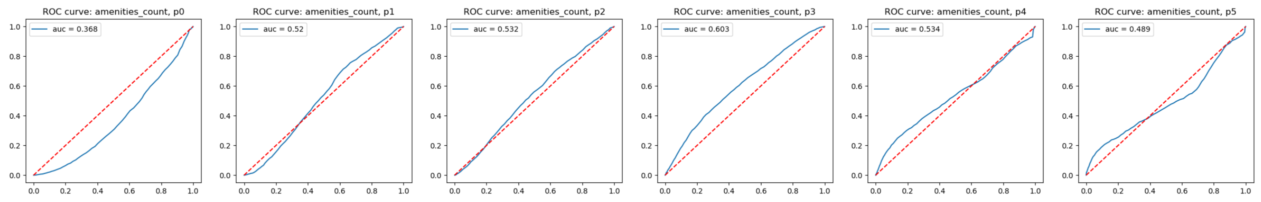
How did you make sense of the provided dataset and get an idea of what might work? Did you use histograms, scatter plots or some sort of clustering algorithm? Did you do any feature engineering? Describe your thought process in detail for how you approached the problem and if you did any feature engineering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

I began by analyzing the raw data feature by feature to determine if any features were significantly correlated with price. For each numeric or boolean feature, I plotted six ROC curves, one for each price point, treating it as a one-vs-all binary classification problem. Thresholds were defined between each pair of realized values and used as classification thresholds. For certain categorical features, numerical features could be extracted (e.g. for the 'bathrooms_text' column) or the categories were concise enough to make one-hot-encoding practical (e.g. for the 'room_type' column). These features were also analyzed using ROC curves. Where necessary, data was cleaned for simplicity (e.g. dropping NaN values). Additionally, due to the size of some of the features and the number of realized values, ROC curves were sometimes plotted with a stepsize parameter k wherein thresholds were defined for every k realized values (to accelerate the plotting process).

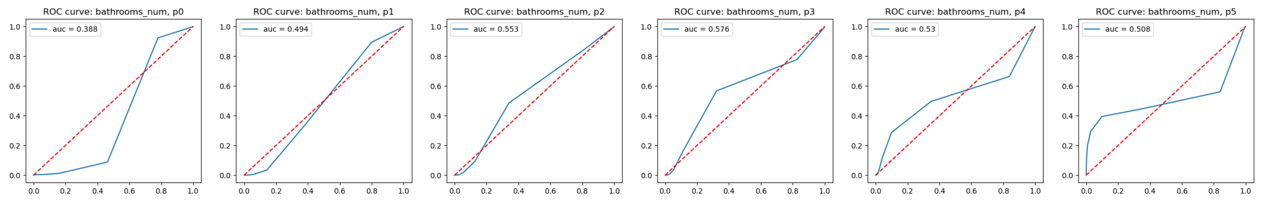
An AUC statistic outside of 0.5 ± 0.1 for any price point was considered "significant," and AUC statistics outside of 0.5 ± 0.2 were noted as particularly impactful predictors. For significant features, when relevant, a histogram or bar chart was created to visualize the distribution of values within the feature. A total of 18 features (or associated extracted features) were identified as important based on this AUC statistic criterion: ’host_listings_count,’ ’host_total_listings_count,’ ’calculated_host_listings_count,’ ’calculated_host_listings_count_entire_h ’calculated_host-listings_count_private_rooms,’ ’room_type’ (separated into three one-hot-encoded columns denoting entire properties, private rooms, and other), ’accommodates,’ ’bathrooms_text’ (separated into a number of bathrooms a boolean shared bathroom column), ’beds,’ ’amenities’ (total count of amenities offered), ’longitude,’ ’number_of_reviews,’ ’number_of_reviews-ltm,’ ’number_of_reviews_l30d,’ ’availability_30,’ ’availability_60,’ ’availability_90,’ and ’availability_365.’ These features naturally fall under several overarching categories describing the property - host information, property information, review data, and availability information - so they were grouped as such and correlation analysis of features within each of these categories was conducted using the Pearson coefficient. From sets of linearly correlated features, a single representative feature was selected at random and linearly uncorrelated features were maintained. An example of this exploratory data analysis process is outlined in the figures below, and the complete analysis is available in the notebook entited eda.ipynb.



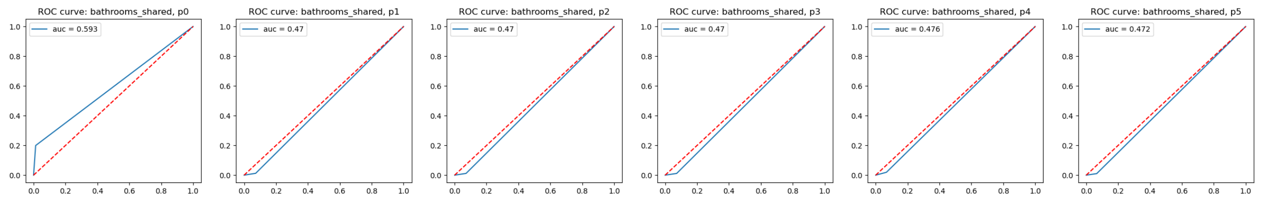
ROC curves for price by number of people accommodated



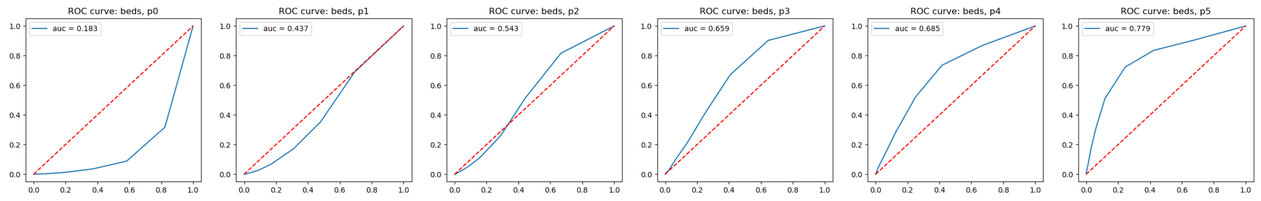
ROC curves for price by number of amenities listed



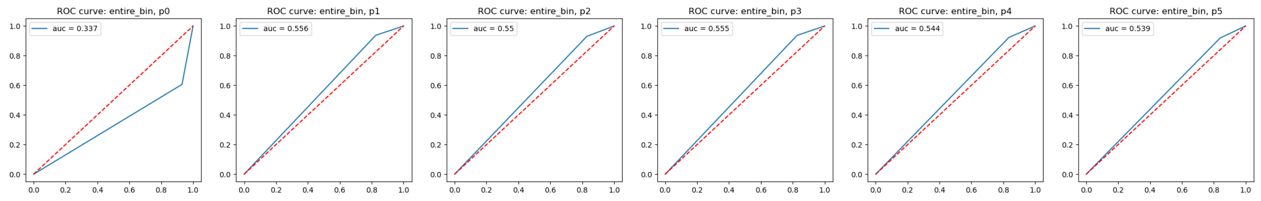
ROC curves for price by number of bathrooms



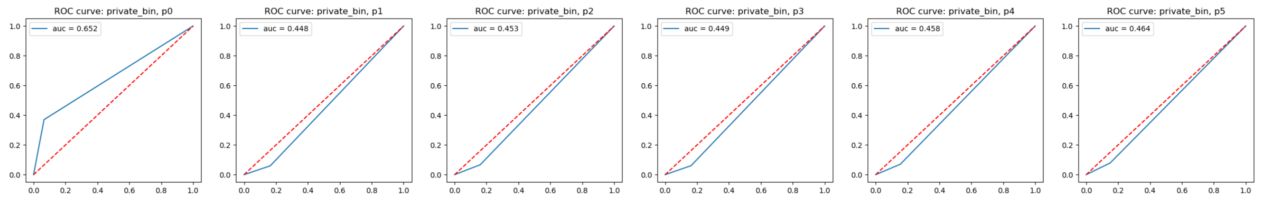
ROC curves for price by whether the bathroom(s) are shared or private



ROC curves for price by number of beds

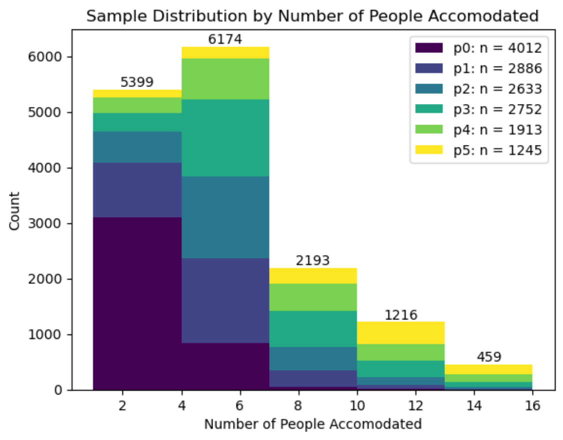


ROC curves for price by whether the guest lives in the entire property

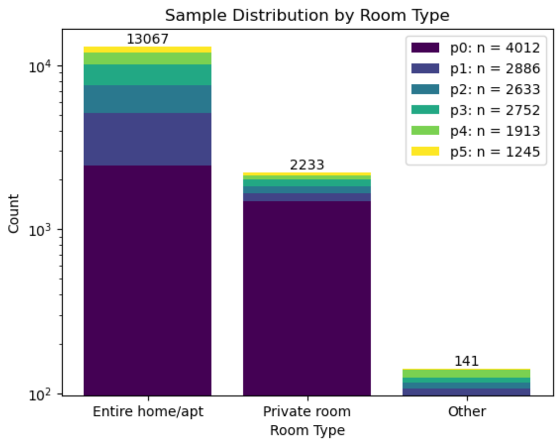


ROC curves for price by whether the guest lives in a private room

Each of the features depicted were predictively relevant for at least one of the price points by the AUC metric defined previously. The distributions of number of people accommodated and room type are shown below, and again, all other generated figures are available in the notebook.

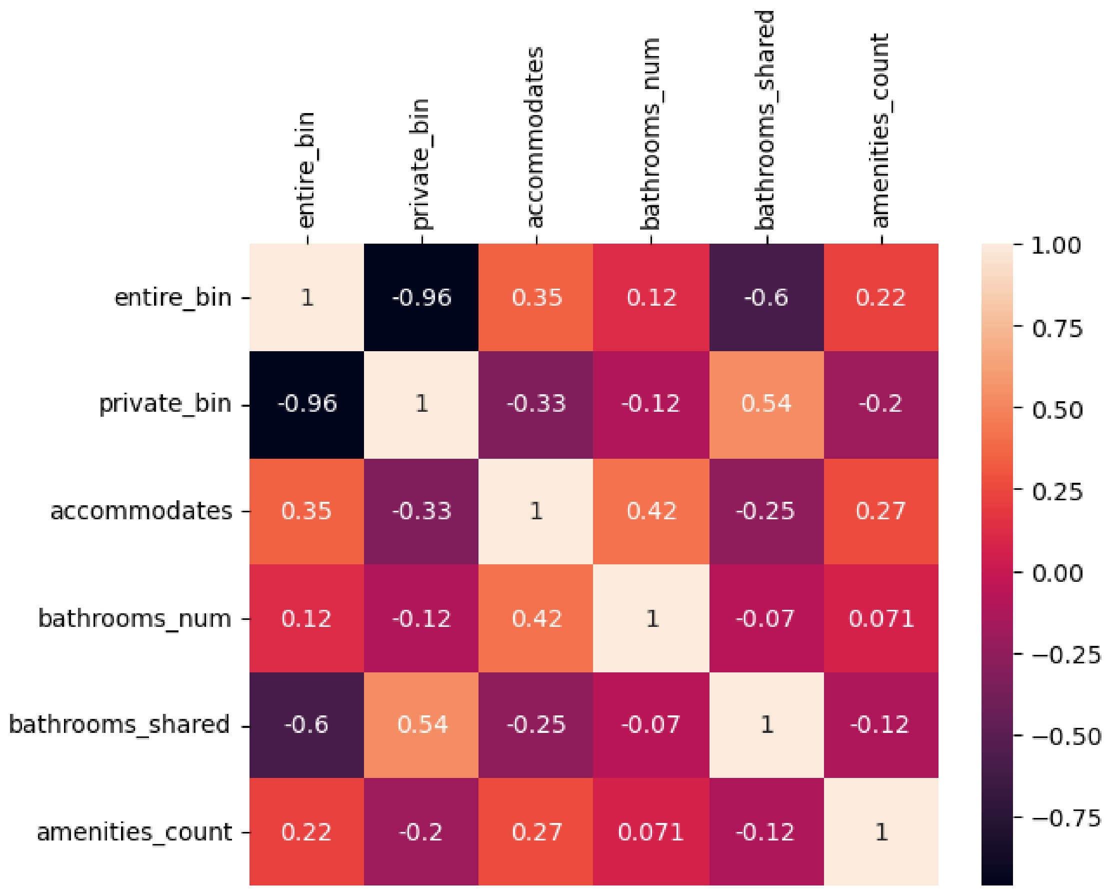


Distribution by number of people accomodated



Distribution by room type

All the features above fall under the property information category, for which a correlation matrix was generated.

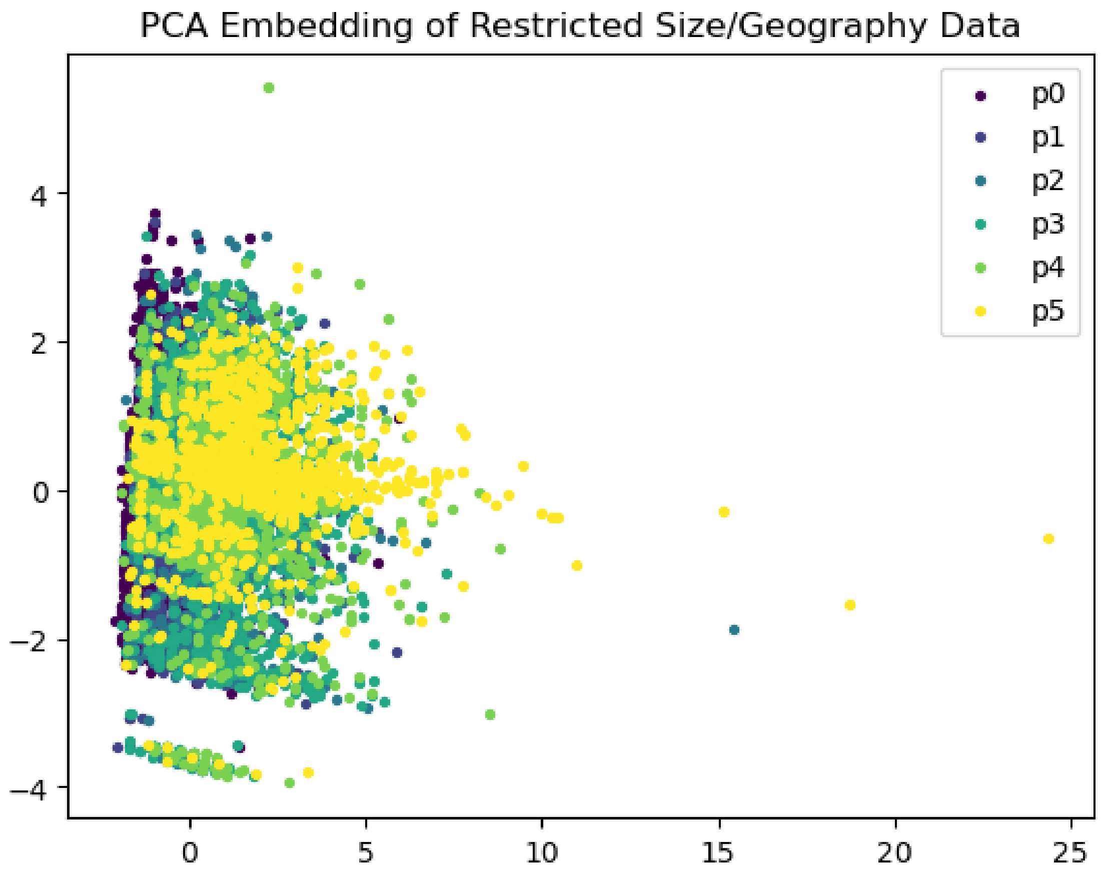


	entire_bin	private_bin	accommodates	bathrooms_num	bathrooms_shared	amenities_count
entire_bin	1	-0.96	0.35	0.12	-0.6	0.22
private_bin	-0.96	1	-0.33	-0.12	0.54	-0.2
accommodates	0.35	-0.33	1	0.42	-0.25	0.27
bathrooms_num	0.12	-0.12	0.42	1	-0.07	0.071
bathrooms_shared	-0.6	0.54	-0.25	-0.07	1	-0.12
amenities_count	0.22	-0.2	0.27	0.071	-0.12	1

Pairwise Pearson correlations of property information features

Following the identification of predictively significant features, I applied several dimensionality reduction methods to better understand the relationship between subsets of multiple features and pricing. Among the methods used were PCA, IsoMap, and tSNE, each of which I'll explain briefly below.

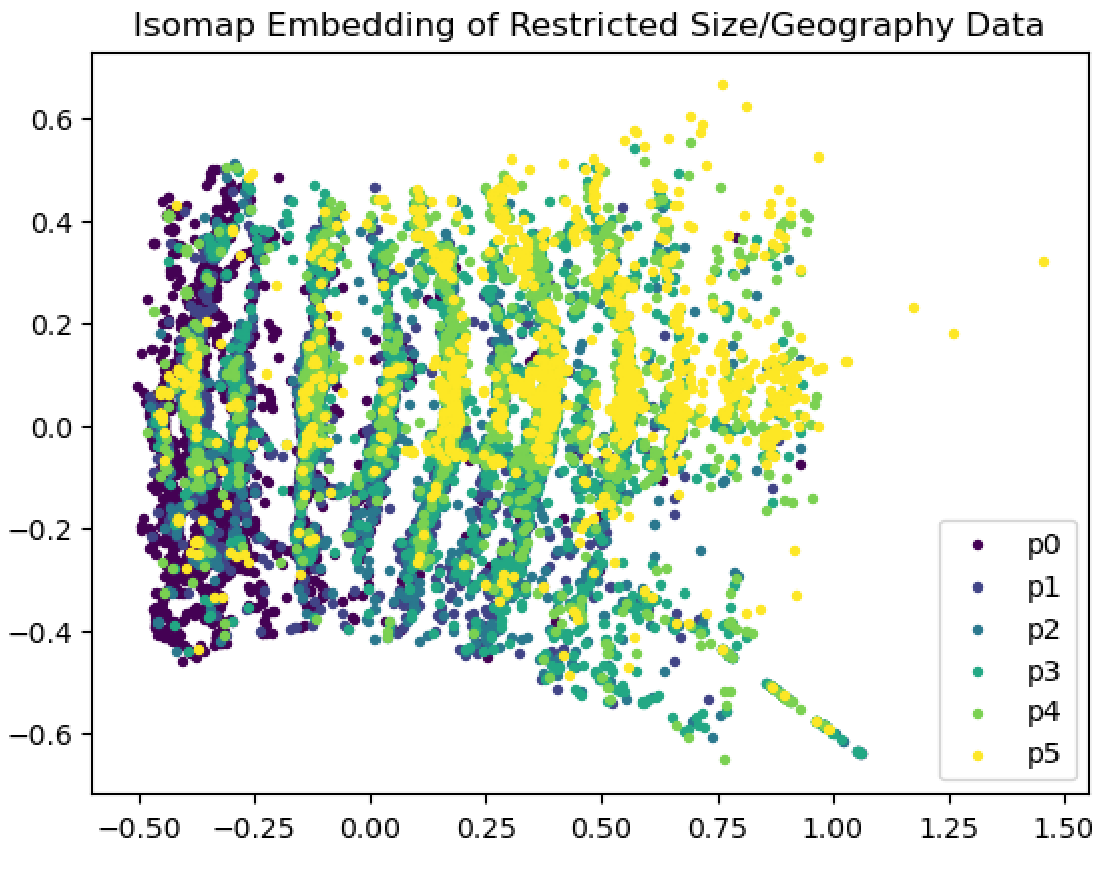
PCA, short for principal component analysis, is a linear dimensionality reduction that aims to project high dimensional data vectors into a lower dimensional subspace while maintaining as much variance from the original data as possible. It finds this subspace by calculating the eigendecomposition of the data covariance matrix or the singular value decomposition of the original matrix. When the resulting eigenvectors or right singular vectors are sorted in descending order by their respective eigenvalues or singular values, the first k said vectors span an appropriate subspace of dimension k.[1]



PCA on property information and geographic coordinate data

[1] Pearson, K. (1901). LIII. On lines and planes of closest fit to systems of points in space. https://doi.org/10.1080/14786440109462720

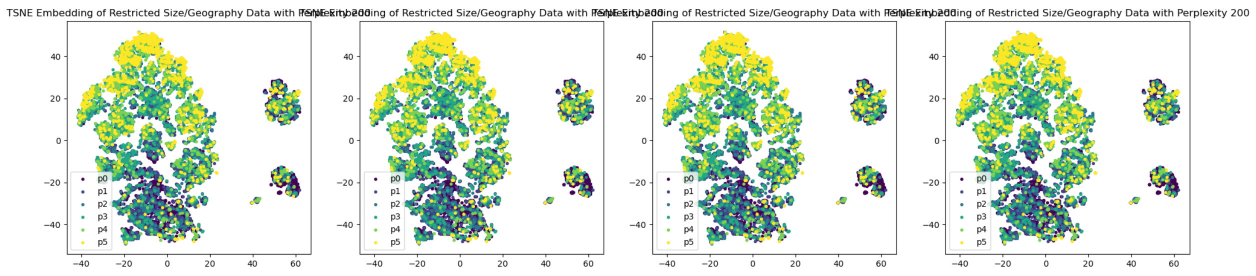
IsoMap is a nonlinear dimensionality reduction and manifold learning technique. Like the rest of the dimension reduction methods described here, it rests on the manifold hypothesis which assumes that the samples in the dataset lie on some low dimensional manifold in high dimensional space. As a result, the samples can be described by the coordinate system of the manifold rather than the coordinate system of the space itself, resulting in a dimensionality reduction. In particular, IsoMap is an extension of multidimensional scaling, which generates low-dimensional embeddings by attempting to preserve relative distances between points in high dimensional space. Rather than preserving Euclidean distances between points, it attempts to learn the intrinsic geometry of the manifold by calculating geodesic distances between points. To do so, it constructs a nearest-neighbor graph between samples and calculates distance between samples with some shortest-path graph traversal algorithm. Multidimensional scaling can then generate a low-dimensional embedding.[2]



IsoMap with 15 neighbors on property information and geographic coordinate data

[2] Joshua B. Tenenbaum et al. ,A Global Geometric Framework for Nonlinear Dimensionality Reduction.Science290,2319-2323(2000).DOI:10.1126/science.290.5500.2319

tSNE, or t-distributed stochastic neighbor embedding, is also a nonlinear dimensionality reduction technique, but rather than consider pairwise or geodesic distance, it measures pairwise affinities from a probabilistic standpoint. The algorithm creates a low dimensional embedding wherein points that were nearby in high dimensional space are also nearby in the embedding with high probability, while points that were distant in high dimensional space are also distant in the embedding with high probability. tSNE is a slightly modified implementation of SNE, which functions by centering probability distributions (Gaussians) on every point in both the original data and the generated embeddings (which are initialized randomly). It then optimizes the embedding by comparing the distributions over the original data to the distributions in the embedding. Distributions are considered similar if they preserve the probabilities of observing similar and distant points described above. Specifically, this optimization process occurs through gradient descent on the Kullback-Leibler divergence of the low and high dimensional distributions.[3] tSNE modifies SNE by replacing the low dimensional Gaussian distributions with t-distributed functions.[4]



tSNE with various perplexity values on property information and geographic coordinate data

2 Models

You are required to use at least two different algorithms for generating predictions. 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

[3] Hinton, G.E., & Roweis, S.T. (2002). Stochastic Neighbor Embedding. Neural Information Processing Systems.

[4] Maaten, L.V., & Hinton, G.E. (2008). Visualizing Data using t-SNE. Journal of Machine Learning Research, 9, 2579-2605.

the code (make sure to cite all references and figures that you use if someone else designed them). Try to be adventurous!

I first used a k-nearest neighbors model as part of my feature engineering. I grouped the training data by city and property type and calculated the mean and median price for each city and property type that occurred more than 20 times in the training data. I then mapped these values to the testing data, but NaNs were generated in samples whose city and/or property type either didn't occur in the training data or didn't occur more than 20 times. I imputed these values by considering their 15 nearest neighbors based on the already existing features, taking the mean or median value of the generated columns for those neighbors. I chose this approach for a few reasons. I had originally imputed these values by taking a global mean/median (i.e. the mean/median price over the entire training set), but I thought considering nearest neighbors would be a better estimate of the calculated features because the nearest neighbors would be found through a consideration of geographic coordinates (approximating city) and property information (approximating the price of a similarly sized property nearby). KNN is also very efficient, running in log-linear time due to the implementation of kd- and ball-trees for space partitioning, and it is easy to access and use through scikit-learn.

When I first began fitting predictive models, I immediately tried KNN, linear and logistic regression, decision trees, and AdaBoost. There was no particular reasoning behind why I chose these outside of the obvious (that it is possible to apply them to multiclass problems). They were simply easy to access through scikit-learn (one import statement and two lines to fit and predict), and I had an intuitive understanding of how and why they worked. These models weren't cross validated, they just gave me a rough understanding of what baseline to expect.

The models I focused most on and submitted predictions from were XGBoost and random forest. I tried random forest first because, based on the preliminary analysis I had already done, I realized that the given features could be very noisy and I wanted a method that could account for this and provide reasonably accurate predictions without being significantly influenced in training by large outliers. Additionally, random forests are especially resilient against overfitting. Implementing a random forest first gave me a good idea of what prediction accuracies I should expect in that they were unlikely to be significantly erroneous or overfit.

I next used XGBoost to generate predictions. I had seen XGBoost applied successfully in the past to multiclass problems, including on Kaggle, and I also read articles about the application of XGBoost to very similar datasets (such as hotel or AirBnB price prediction, albeit with a different set of features).

These models also share many desirable traits in common. Both are highly efficient to train, and by virtue of being ensemble methods that are based on trees, offer a high number of tunable parameters and a significant resistance to overfitting.

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.

Random forests are built on a very conceptually simple algorithm. The algorithm draws a bootstrap resample of size n from the training data and grows a tree from it. At each node, m features are selected at random and the ideal feature (based on some metric of information gain such as the Gini Index) is used to split the node. In addition to the number of trees in the forest, the splitting metric, and the amount of training data used to train each tree, random forests have many other tunable parameters including but not limited to the maximum depth of the tree, the maximum number of leaf nodes allowed, and the minimum number of samples required in a node to continue splitting. Many of these parameters are in place to reduce overfitting in individual trees and to control sparsity. When the entire forest has been generated, in the case of classification problems, the mode of the forest's predictions generated by each tree is taken as the classification. The Gini Index of a tree node n is defined

$$\text{Gini}_n = 2p(1-p) \tag{1}$$

where p is the proportion of positive samples flowing into the node and $1-p$ is the proportion of negative samples. If the Gini Index is low, it indicates that the proportion of one class or the other is very high, meaning the node makes predictions with high confidence. Decisions trees built on the Gini index seek to maximize Gini reduction, or the decrease in the Gini index after a split where the post-split index is calculated as the average Gini index across all children weighted by the number of samples in each child.

XGBoost, short for Extreme Gradient Boosting, is an implementation of gradient-boosted decision trees. It sequentially builds decision trees to greedily correct errors made by previous trees in the model. In particular, it defines an additive model composed of a sum of predictions made by trees and minimizes the objective function defined by a convex loss and a regularization term that depends on the L2 norm of the parameters defining each tree. However, this objective function can only be optimized and trained in an additive manner. Each new tree is added to minimize the loss given the predictions of previous trees, and this minimization problem is solved efficiently by instead considering the second order approximation (a Taylor polynomial) of the loss function.

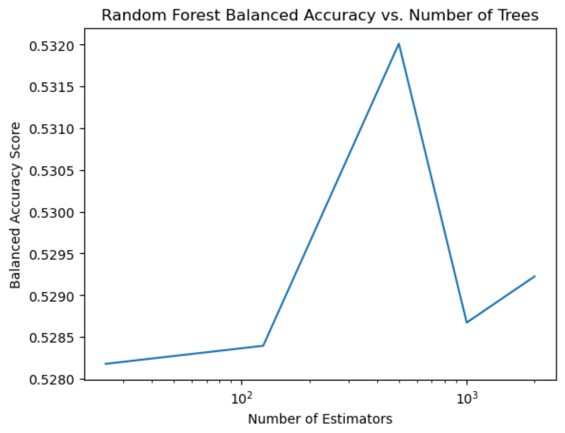
Training the models themselves was relatively fast (less than a minute for both XGBoost and Random Forest, for all combinations of hyperparameters tested). Generally, both these algorithms took between 20-40 seconds to train. The most time consuming parts of training were in feature generation (particularly for clustering where tSNE, IsoMap, and spectral embedding took more than 20 minutes on the split training fold alone and more than 40 minutes in the final model) and cross validation (anywhere from 10 minutes to hours).

4 Hyperparameter Selection

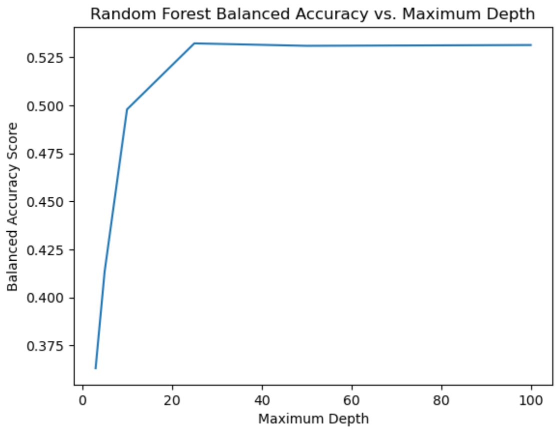
You also need to explain how the model hyperparameters were tuned to achieve some degree of optimality. Examples of what we consider hyperparameters are the number of trees used in a random forest model, the regularization parameter for LASSO or the type of activation / number of neurons in a neural 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.

For some reason, GridSearchCV consistently ran incredibly slowly on my machine (on the order of hours), so I wasn't able to tune as many hyperparameters as I would have liked, and in some cases I had to resort to manually changing them to approximately evaluate how well my models were performing with certain hyperparameter configurations. That said, I ran cross validation on XGBoost with the number of estimators (500, 750, 1000, 2000), maximum depth (2, 3, 5, 7, 8), regularization parameter (0.3, 0.5, 0.7), and boosters 'gbtree' and 'dart,' arriving at an optimal selection of 'gbtree,' 500 trees, regularization parameter 0.7, and maximum depth 8. This result took several hours and my computer died after it finished so I wasn't able to display the results in my notebook, but the code is there. The other parameters were a result of manual changes, though they seem to have at least slightly impacted performance. I changed the learning rate to see if I was potentially descending the gradient too quickly and missing the optimal solution as well as the sample selection proportions (for splitting), though I later decided to leave the default values of 1 because I wasn't noticing significant performance changes and wanted to ensure diverse trees.

For the random forest, I similarly ran cross vaidation on the number of trees and the maximum depth of each tree, finding that 500 trees was optimal. I also found that not specifying maximum depth led to reasonably strong results (after increasing max_depth multiple times outside of the range in my cross validation).



Random forest performance by number of estimators



Random forest performance by maximum depth

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.

After cleaning my data (dropping NaNs, extracting certain features from the original data for each sample), I used train_test_split to split the given training data into 80-20 training and validation sets. I ensured the extraction process didn't lead to data leakage (all extraction happened by sample, no interaction between samples).

6 Errors and Mistakes

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'm writing this when I don't have enough time to train another model, but I'm realizing that my embeddings introduced a lot of data leakage. I ran embedding algorithms using both the train and test set because its unsupervised, and my original reasoning for why it didn't introduce leakage was because I wasn't actually revealing anything about the labels of the testing set (i.e. if I had been given the true labels of the test set and fit a KNN classifier to both sets, that obviously would have introduced leakage, but not knowing the labels, embedding was just another form of feature extraction like counting the number of bathrooms from a text column). To be honest, I'm still not entirely sure if this qualifies at leakage. In a real world scenario where this data was coming in real time (as in a new listing was posted and I wanted to predict the price), I feel I could still attempt that classification problem by running clustering on that new sample along with all the samples I already have and know the prices for. In a sense, I'm extracting a feature from a previously unseen sample by observing where it lies in relation to to previously seen samples. All of this can happen without the model knowing what the label of the new sample is, and the model is fit also without this knowledge.

The hardest part of the competition for me was the initial analysis. I didn't see the Ed post with the spreadsheet containing information about the features until much later, so I spent a lot of time early on exploring feature importance and correlation without really knowing what the features meant. This doesn't change my interpretations of the results, but it definitely made feature selection much harder because it can often be helpful to have an intuitive understanding of which features are likely important and which aren't, but that doesn't happen if I'm not sure what the features mean.

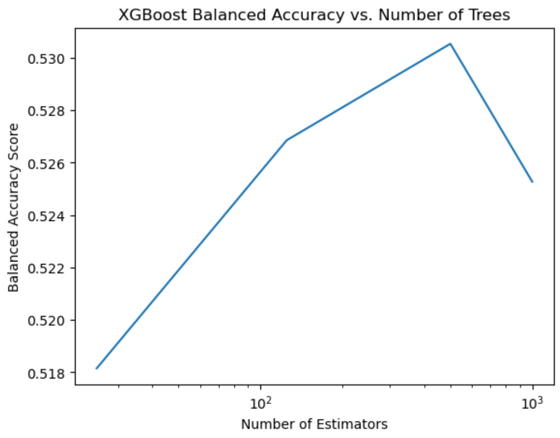
Another significant challenge was finding where to go after initially training models. I'm coming to realize that there's a significant gap between understanding how these algorithms work from a technical standpoint and actually being able to apply them. I realize that, in many cases, I can choose the right algorithms and tune them correctly, but I don't know where to go from there even though I know there are better ways (by virtue of seeing the leaderboard).

Runtimes are also a humongous pain.

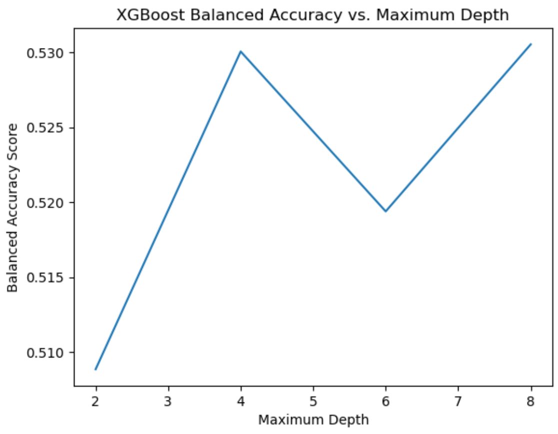
7 Predictive Accuracy

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 class-balanced accuracy 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. Scoring in the 90th percentile and above will give 10 points while being in the bottom 10 percent will give one point. 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.

My Kaggle username is Haiyan Wang. The below results are calculated using scikit-learn's balanced_accuracy_score metric (which I believe is similar to the one used on Kaggle). Ideally, I would have tracked each of my submissions and the model and parameters I used, but I didn't think to do so until later. The figures below show further results of hyperparameter tuning that I used to arrive at my optimal models.



XGBoost performance by number of estimators



XGBoost performance by maximum depth

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 Dec 5th. Good luck!