QTM 385 Intro to Statistical Learning

Team members: Annie Luo, Billy Ge, Latifa Tan

Instructor: Dr. McAlister

Due date: Dec. 14, 2022

Atlanta House Rental Price Prediction Final Project

Introduction

Based on our summer experience with the Data Think research program and the machine learning tools gained from the Intro to Statistical Learning class, our goal for this final project is to apply those methodologies in Python in order to conduct predictions on rental house prices. The main dataset in our project is *SingleFamilyRentals.* Throughout the project, we take Price as the dependent variable and the following as independent variables: SqFt, Acreage, Beds, Baths, Latitude, and Longitude.

Data Cleaning

To start off, we conducted data cleaning on the raw dataset. We first looked at the dependent variable, Price. As shown in Chart 1, because the distribution of Price is heavily skewed, we decided to remove the outliers, reducing our dataset size from 6111 to 5469. As such, Chart 2 plots the histogram of Price, which displays a normal distribution. That is our response variable in every model. Next, we found some null entries in the predictors, which requires us to apply some imputation methods.

Chart 1 (Raw Price)

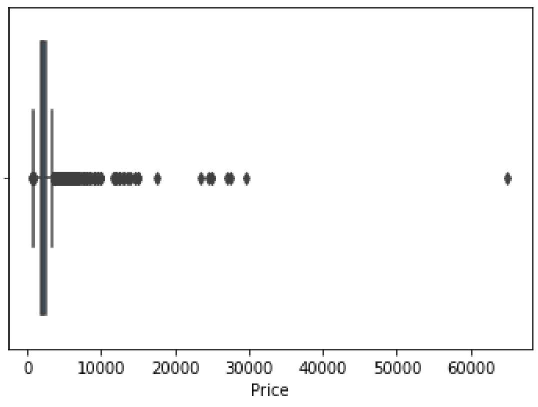
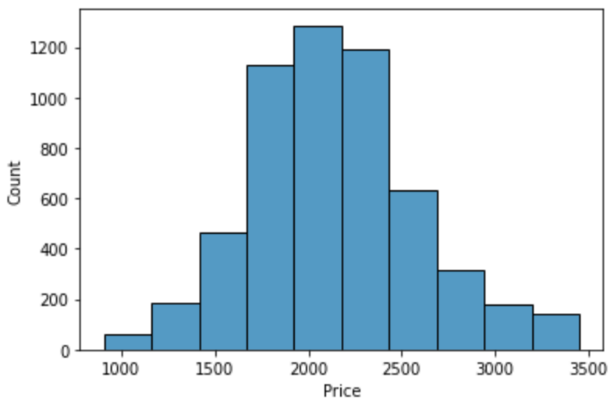


Chart 2 (Cleaned Price)



NA Imputers

We applied three main methods to filling the null values: taking the averages of the columns (Simple imputer), KNN imputer, and Random Forest imputer. For each method, we first imputed the missing entries and then applied the GAM model to evaluate how well the imputation method performs on prediction. The reason behind using the GAM model for performance comparison is its fairly precise and low computation cost nature. In both the KNN and random Forest imputer methods, the algorithms treat the column of missing values as the response variable and the rest of the columns as the predictors. For the KNN imputer, we tuned the k (number of nearest neighbors) in the range 0 to 500 and finalized the model with k = 100 according to the GCV scores from the GAM model. The performance of each imputation method is summarized in Table 1 below. We concluded that the Random Forest imputer performs the best, and we decided to use it as our predictor space moving on.

TABLE 1: Null Value Imputers Performance

Imputer Method	GCV score from GAM
Simple Imputer	289.1362
KNN Imputer	282.8076
Random Forest Imputer	282.7039

Methodologies

With the filled dataset ready, we applied and tuned various price prediction models. We took four main approaches: KNN, Random Forest, Boosted Trees, and Neural Networks. In KNN, we used cross validation on out-of-sample errors to measure model performance. In the rest of the models, since each of them takes a lot of computation time, we decided to split the dataset into training and testing subsets based on the 70/30 split rule. We know doing so may weaken the precision of our model evaluations, but we believe the computational cost is much greater and unnecessary. The following table shows the performance results of all the models we implemented in the project.

TABLE 2: Performance of Price Prediction Models

Prediction Model	Out-of-Sample Error
KNN (non-scaled)	337.141 (CV)
KNN (scaled)	321.512 (CV)
Random Forest	267.38
AdaBoost	266.38
Gradient Boost	265.98
MLP Neural Network	271.31

KNN model

We came out with two different models for KNN. Their difference lies within the predictor space, when one model is built on a scaled predictor space whereas the other model is built on the original predictor space as it is. Their performance also differs. As shown in Table 2, the scaled KNN model wins in price prediction. As far as we understand, we believe it is because our original independent variables vary a lot in scale. For example, the Latitude and Longitude elements may differ only by 0.5 between two vectors when SqFt may change by more than a few hundred. Since the KNN model makes predictions based on the distance between point vector estimates, it is dominated by SqFt and ignores geographic locations if we build the model on the original predictor space. Therefore, when we scaled all predictors by the MinMaxScaler, the KNN model performed better.

Every KNN model needs tuning in k (number of nearest neighbors). Our optimal k in the non-scaled KNN model is 300 compared to 14 in the scaled KNN model.

Random Forest model

Since all Random Forest implementations share the same underlying logic, we prefer not to be verbose here. However, rather than tuning just two parameters (number of trees and number of random predictors drawn in each tree split), we had to tune one more parameter in sklearn (minimum number of samples at a leaf node). It is because, the default setting for the min_samples_leaf parameter is 1, which can easily lead to overfitting.

Our best Random Forest model has 500 trees, random predictors (the entire predictor space) = 6, and min_samples_leaf = 3. It confirmed our overfitting hypothesis that tuning the min_samples_leaf parameter is necessary. We assumed 500 trees is enough to give us a fairly precise performance measure, since tuning it takes more computation cost.

Boosted Trees model

There are two Boosted Trees models that we learned and implemented, adaboost and gradient boost respectively. Based on our current knowledge, these two models are boosted in the sense that the trees are built in an sequential order. The next tree always tries to improve predictions on samples performed worst by the previous trees. That is the largest innovation that separates Boosted Trees from Random Forest. Both models are also ensemble methods, meaning that the predictions are averages over a number of weak learners that individually do not perform well. However, adaboost and gradient boost differ in their internal approaches. On one hand, adaboost initially assigns equal weight to the input samples and updates those weights in a sequential order by placing heavier weight to samples that failed to be predicted well by the previous trees. On the other hand, gradient boost initially assigns the mean of the response variable to every sample, subtracts the true estimates from the mean, and uses their difference as the response variable in the next sequential tree.

Since Boosted Trees are more complicated than Random Forest, we had to tune a few more parameters, especially the learning rate and the maximum depth of each individual tree. As shown in Table 2, both Boosted Trees models perform slightly better than Random Forest but their computation costs are much greater (Table 3).

One detail worth explaining is the maximum depth parameter. It is counter-intuitive that an individual weak learner tree has a maximum depth of 10 in the adaboost model and 6 in the gradient boost model. From our summer research, we saw that geographic locations can largely determine the price of a rental house. However, the correlation between Latitude and Longitude is so complex that any shallow trees cannot find it. That requires the weak learner trees to grow deeper. To further illustrate the abnormality, we added the contour plot of Price on Latitude and Longitude holding the rest of the predictors constant to our project deliverable. It is obvious that geographic locations have complicated associations with Price.

Neural Networks

Aiming to take a peek at the popular neural networks world, we implemented the entry level MLPRegressor from sklearn. Here are our gains from the learning process. First, we now understand why neural networks are so costly computationally, because there are simply so many parameters that need to be tuned. Second, we understand why the model is essentially a linear regression, since that is its underlying logic. Neural networks start from the list of input predictors, assign weights to them, connect them to the hidden layers and nodes (tuned parameters), output predictions and go back to update the weights iteratively. Third, we saw why neural networks, similar to Random Forest, are black box models. But, different from Random Forest which is black box in the tree structures, neural networks are black box in the hidden layers and nodes.

Unfortunately, our best Neural Networks model performs worse than the tree-based methods. However, we believe that given enough computation power, Neural Networks can be better tuned. Our current best model has 1 hidden layer and 90 nodes within, and it has a maximum iteration of 10000.

Computation Costs

Below is the summary table for computation costs of all Price prediction models. As we can see, KNN is the fastest model, which matches what we have learned in class. Random Forest is not as costly because we did not tune the number of trees parameter, and the size of the predictor space is not large in our dataset. Boosted Trees and Neural Networks models are the most costly models in terms of computation time, because they require us to tune many of the parameters. At the same time, when tuned property, these models can perform very well.

TABLE 3: Computation Time of Price Prediction Models

Prediction Model	Computation Time (in min.)
KNN	1
Random Forest	5
AdaBoost	24
Gradient Boost	21
MLP Neural Network	32

Conclusion

To sum up, in this project, we learned how to implement in Python the models we are already familiar with as well as more advanced prediction models such as Boosted Trees and Neural Networks. In addition, we learned scaler and imputation methods (KNN and Random Forest) to solve the prevalent flawed data problem.

Although our learnings are yet elementary from all perspectives, we believe that it is important to know the most popular machine learning tools so that we are not scared by them. Being able to implement the models from two different softwares, R and Python, hopefully will also help us gain an edge among our competitors in the job market.

Comparing our Price prediction performance to Matt's model performance from the summer, we are uncertain as to which model is strictly better. There are two reasons. First, we did not perform cross validation estimates on our best models. Second, we do not have the exact out-of-sample measurement from Matt. However, one thing is pretty clear: there may not be large improvements beyond the 265 bottleneck. As data researchers, we need to think whether performing slightly better at the expense of much greater computation time is worth the effort.