Supervised Learning - Analysis and prediction of house prices

Dinesh R Poddaturi

1 Introduction

In this project I use multiple supervised learning methods to carefully analyze and predict house prices. Originally this project was a part of Kaggle data science competition. Although I didn't win the competition, I learned a lot during and after the competition. I believe in the mantra "learning by doing" (coined by a famous American philosopher *John Dewey*). So I have been working on this project by making multiple changes, trying different algotithms, eventually improving the prediction power.

2 Data and Methodology

The data are free to download from Kaggle website. I downloaded the data a while ago and has been working with the same data. There are two different files of data; (1) Training data containing the price of the house and house characteristics (e.g., the year it was constructed, number of bed rooms, number of bathrooms, latitude and so on) and (2) Testing data containing only the characteristics of the house. Our goal simply is to predict the house prices in the testing data. In order to do that, first we analyse the training data using multiple supervised learning methods and use the models to predict the house price in the testing data.

One could immediately say this is not a classification problem. Note that the house prices are not boolean i.e., 0 or 1, instead they are discrete variable (one could argue it is a continuous variable, but I stay away from that argument in this analysis). So this could be considered as a regression problem. From my rigorus training in Economics, I can immediately identify a problem with the data. Endogeneity; People choose to live in a particular area. It could depend on the school district (people with kids), location (near to a metro or a mall), clean air, less noise pollution and so on. We cannot observe all these in the data. Although we have latitude and zip code, we cannot observe individual decision process and on what basis an individual makes choices. Hence, there could be endogeneity in the data. In this work, I do not focus on that issue. The primary objective of this work is to use supervised learning methods to predict the house prices. Furthermore, this is not a causal study (I am not claiming any causality and not claiming a certain variable causes the house price to increase or decrease). So, I stay away from endogeneity and self selection issues. This is a pure supervised learning exercise.

3 Variable description

- id - (in the test set only) number of the test case
- property - a unique identifier for a house
- date - date house was sold (YYYYMMDD)
- price - house selling price (prediction target)
- bedrooms - number of bedrooms in house
- bathrooms - number of bathrooms in the house
- sqft_living - square footage of house
- sqft_lot - lot area
- floors - total floors/levels in the house
- waterfront - indicator that the house has a view to a waterfront
- condition - an overall condition rating

- grade - an overall grade given to the housing unit per a King County grading system
- sqft_above - square footage apart from basement
- sqft_basement - square footage of basement
- yr_built - year of initial construction
- yr_renovated - year when house was renovated
- zipcode - US Postal Service zipcode
- lat - latitude
- long - longitude
- sqft_living15 - square footage of house in 2015
- sqft_lot15 - lot size in 2015

```r
# install.packages("librarian")


librarian::shelf(stringr, Matrix, glmnet, xgboost, randomForest,
                 caret, scales, e1071, corrplot,
                 psych, tidyverse, lubridate, pls,
                 gdata, graphics, rpart, gbm, earth,
                 Boruta, ggcorrplot, Metrics, rpart.plot)
```

```r
# Reading training and testing data
train_data <- read.csv("./Data/train.csv")
test_data <- read.csv("./Data/test.csv")


### Converting date to ymd format
train_data$date <- ymd(train_data$date)
train_data <- train_data %>% select(price:sqft_lot15)
```

4 Preliminary analysis:

In this section, I perfom some preliminary analysis. Preliminary analysis include carefully studying each variable (house characteristic), it's data type (boolean, string, integer, character etc.), each variables correlation with outcome, importance of each variable, and whether the data type of a variable needs to be modified.

Simple correlation analysis:

```
housePrice <- train_data$price
correlationMatrix <- cor(cbind(housePrice,train_data %>% select(-price)))
correlationMatrix_sorted <- as.matrix(sort(correlationMatrix[,'housePrice'], decreasing
correlationMatrix_sorted
```

From the correlation matrix the predictors sqft_living, grade, sqft_above, sqft_living15, and bathrooms are highly correlated and Zipcode, longitude, condition, and yr_built are less correlated with the price.

Multiple Linear Regression (MLR) to find the important variables:

```
mreg <- lm(price ~ . , data=train_data)


an <- anova(mreg)


impVar_lm <- varImp(mreg)
```

The above code snippet simply performs MLR using training data, and using the fitted model gives variable importance and their significance. From anova results, I conclude that almost all the explanatory variables are significant explaining the house price. However, the variable importance method concludes that bedrooms, sqft_living, waterfront, grade, yr_built, lat are the most important variables. By close analysis I find other variables important as well.

Using eXtreme Gradient Boosting (XG Boost) to further analyze variable importance:

```
designX <- train_data %>% select(-price)

xgBoost_fit <- xgboost(data.matrix(designX), train_data$price, nrounds=800)

xgBoost_Imp <- xgb.importance(model = xgBoost_fit, feature_names = names(designX))

xgBoost_Imp
```

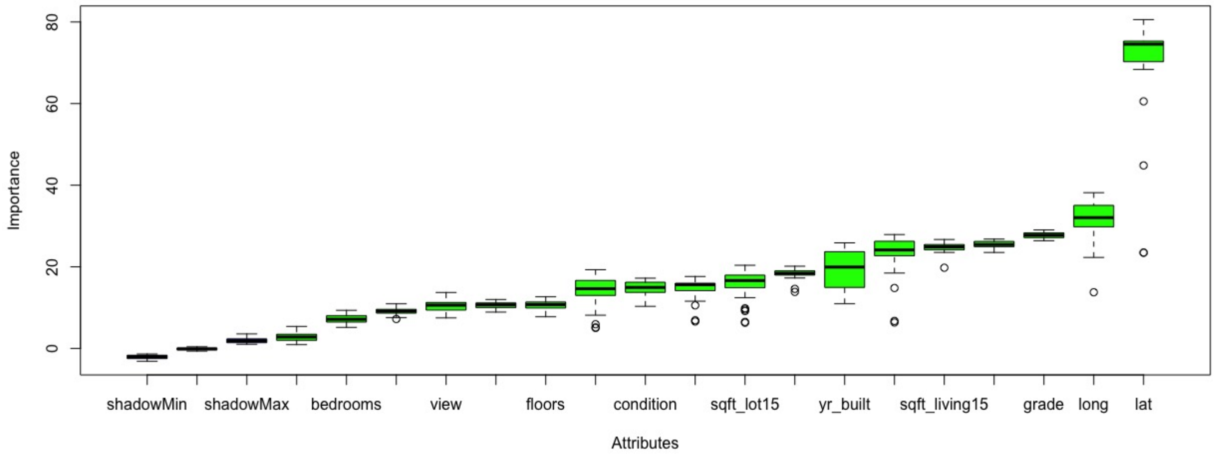
Recently XGB has become a go-to algorithm in machine learning predictions and also in data science competitions. Execution speed and model performance are primary feature of this algorithm. I use this algorithm later to train the data. The above code snippet fits the training data and gives the important variables to consider (not remove). The XG boost selectd sqft_living, grade, lat, long, sqft_living15, yr_built, and waterfront as the most important variables. Similar to the variable importance from MLR, XG boost selected similar variables as important.

Boruta feature selection algorithm to determine the variable importance:

```
boruta_fit <- Boruta(price~. , data = train_data , doTrace = 2)

print(boruta_fit)

plot(boruta_fit)

colMeans(boruta_fit$ImpHistory)
```

Boruta feature selection algorithm is also widely used to find the variable importance. The above Boruta performed 68 iterations in 15.8 mins, and 18 attributes confirmed important including bathrooms, bedrooms, condition, floors, grade and 13 more. No attributes deemed unimportant from the fit. Boruta also provides a simple visualization of the variable importance.

Figure 1: Boruta fit plot



From all the tests for variable importance above, I conclude that almost all the variables are important. Now, I read description of each variable and use my judgement to drop, modify, or change the data type.

1. I remove *date* from the data since the houses sold are in the same year.
2. I change *zipcode* to *factor* variable since it is not an integer/numeric.
3. I change *waterfront* to *factor* variable since it's data type is binary (0 or 1).
4. I change *condition* to *factor* variable since it is a rating.
5. I drop *sqft_living*, *sqft_lot*, *sqft_above*, and *sqft_basement*. This is because *sqft_living15* and *sqft_lot15* contain the most recent information about the house. Although more information is better. Including all the variables might make our design matrix a singular matrix. Although the algorithms we are going to use would drop those variables, I drop them to make the data as clean and informative as possible.

The above changes are made in both training and testing data.

5 Implementing supervised learning algorithms

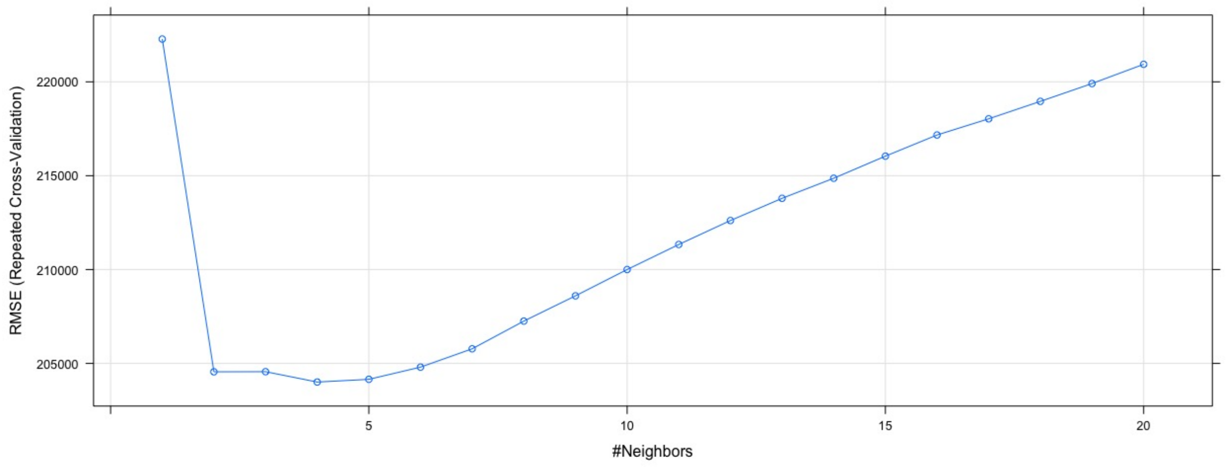
For every algorithm, I provide a tune grid and also ask the algorithm to perform cross validation. In particular, I choose cross validation with repeats. Of course repeated cross validation will increase the execution time. But I prefer good results over time. With increased computing power we have now, we can use HPC or even use parallel computing to execute the code. So a repeated cross validation is used with 2 repeats. Since I am doing this in my machine I chose 2 repeats. If it were to be executed on HPC I would choose 20 repeats. I also standardize the data whenever necessary. Basically I center and scale the data such that I don't have to think about the units of the variables included in the modeling.

5.1 k-Nearest Neighbour (kNN)

The tune grid I provide for kNN is from 1 to 20.

```r
knnTune_housePrice <- train(y = house_train[,1], x = house_train[,2:13],
                            method = "knn",
                            preProcess = c("center","scale"),
                            tuneGrid = data.frame(.k=1:20),
                            trControl = trainControl(method = "repeatedcv",
                                                     repeats = 2, number = 10))

plot(knnTune_housePrice)
knnBestTune <- knnTune_housePrice$bestTune
#### From the plot and bestTune above, the model selected k = 4
# knnTune_housePrice$results
```

Figure 2: kNN Tuned Plot



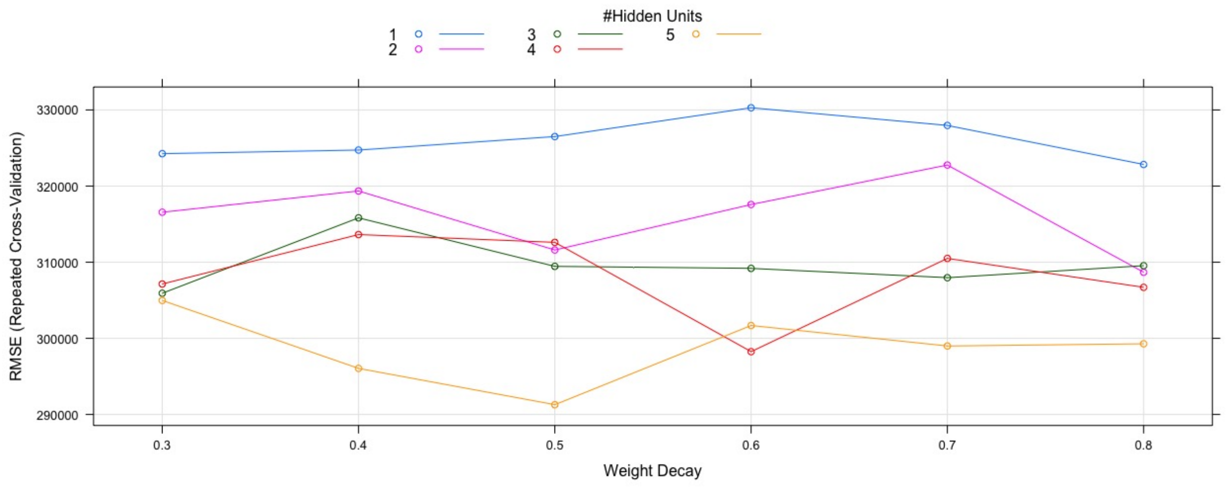
From the plot above, the algorithm picked $k = 4$ as the best tune. From the plot, we can observe that for $k = 4$ the Root Mean Squared Error (RMSE) is the lowest.

5.2 Neural Network

First we give the algorithm a grid of size and decay to look over. This is to give some direction to the algorithm and to make sure I get global optimum. The values chosen for the size and decay are widely practiced in the literature. *Size* is the nuber of units hidden layers and *Decay* is the regularization parameter to avoid over-fitting.

```
nnet.grid <- expand.grid(size = seq(from = 1, to = 5, length.out = 5),
                         decay = seq(from = .3, to = .8, length.out = 6))
nnetTune_housePrice <- train(y = house_train[,1], x = house_train[,2:13],
                             method = "nnet", trace = FALSE,
                             preProc = c("center","scale"),
                             linout = TRUE, tuneGrid = nnet.grid,
                             maxit = 50,
                             trControl = trainControl(method = "repeatedcv",
                                                  repeats = 2, number = 10) )
plot(nnetTune_housePrice)
nnetBestTune <- nnetTune_housePrice$bestTune
# size decay
# 5   0.5
nnetResults <- nnetTune_housePrice$results
```

Figure 3: Neural Net Tuned Plot



The lowest RMSE is attained for *size=5* and *decay=0.5*.

5.3 Multiple Linear Regression (MLR)

There is no tuning for MLR. However, the best tune provided under cross validation would be either intercept or no-intercept.

```
lmTune_housePrice <- train(housePrice~., data = house_train,
                           method = "lm",
                           trControl = trainControl(method = "repeatedcv",
                                                    repeats = 2, number = 10))
```

```
lmBestTune <- lmTune_housePrice$bestTune


lmResults <- lmTune_housePrice$results
```

The MLR results are provided in the table 1

Table 1: MLR best results

Intercept	RMSE	Rsquared	MAE
TRUE	186853.2	0.7581584	106488.8

5.4 Random Forest (RF)

For random forest algorithm, I must provide the design matrix. This is because, we have factor variables in the data and we need a seperate column for them.

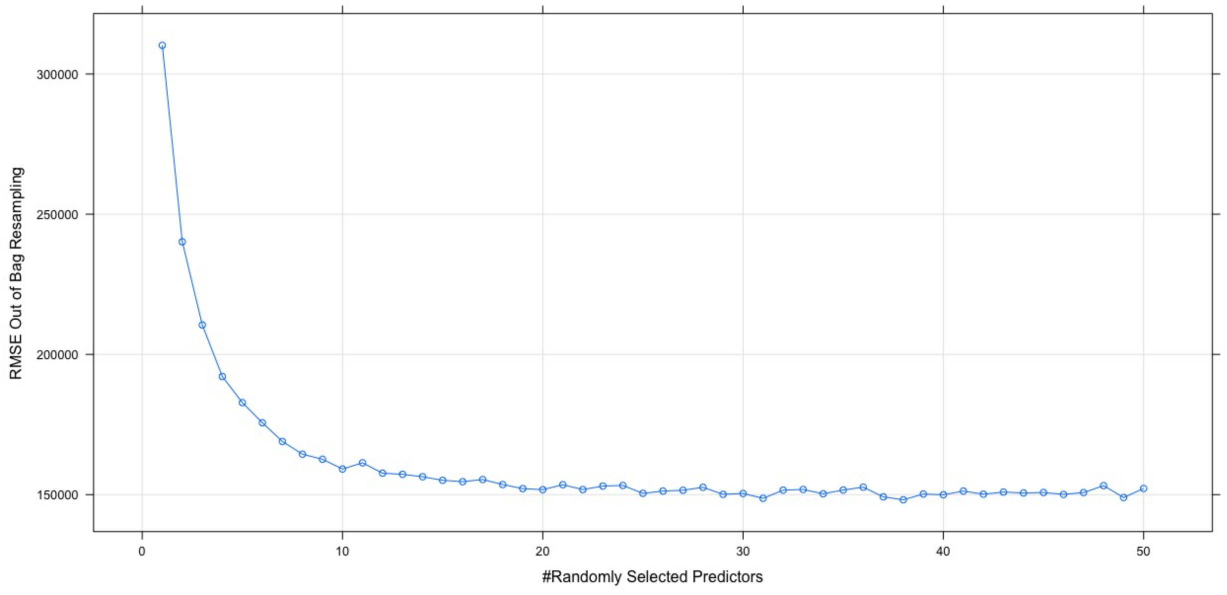
In the random forest model, the original training data is randomly sampled-with-replacement generating small subsets of data. These subsets are widely known as bootstrap samples. For tune grid I provide *mtry*, which is number of variables randomly sampled as candidates at each split. I also provide *ntree*, which is bacially telling the model the number of trees to grow. Instead of repeated cross validation I use **Out Of Bag** (oob) which is a method to validate the ranndom forest model.

At each bootstrap, the algorithm leaves out some rows and trains with kept data. Once the training is finished, the model is fed the left out rows and asks to predict the outcome. This is repeated multiple times and oob score is computed by simply adding up the number of correctly predicted rows. We can also use repeated CV as well. However, OOB is a well known method and is recognized well in the literature

```
designMat <- model.matrix(lm(housePrice~.,data=house_train))
designMatRF <- designMat[,-1]
```

```r
forestTune_housePrice <- train(y = house_train[,1], x = designMatRF,
                               tuneGrid = data.frame(mtry=1:50),
                               method = "rf", ntree = 150,
                               trControl = trainControl(method="oob"))

forestBestTune <- forestTune_housePrice$bestTune
plot(forestTune_housePrice)
```

Figure 4: Random Forest Tuned Plot



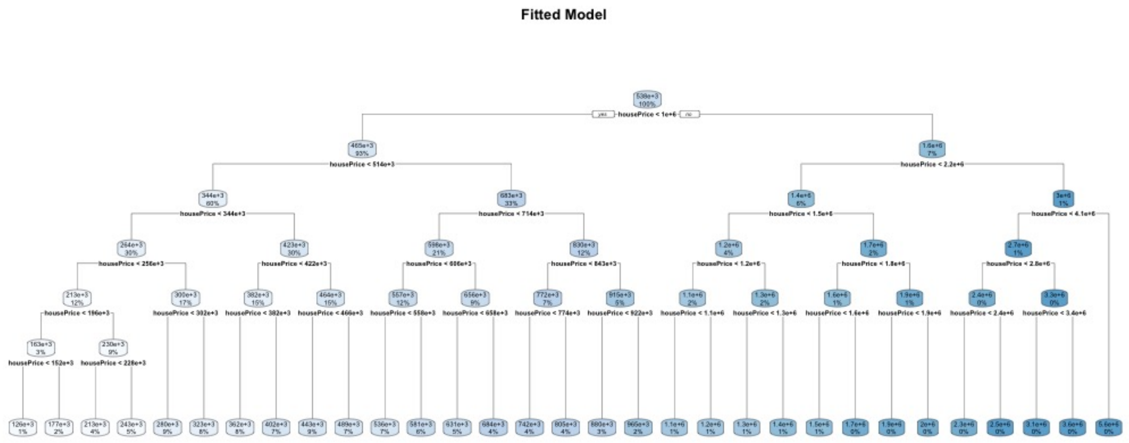
Under *mtry=38*, the model achieved the lowest RMSE.

5.5 Decision Tree

I specify the grid of complexity parameter for tuning. It is a stopping parameter. I simply helps speed up the search for splits, it identified the splits that don't meet the specified critetia and prune those splits before going deep.

```r
TreeTune_housePricing <- train(y = house_train[,1], x = house_train[,2:13],
                               method = "rpart",
                               tuneGrid = data.frame(cp = seq(from = .0001,
                                                              to = .1, length.out = 50)),
                               trControl = trainControl(method = "repeatedcv",
                                                        repeats = 50, number = 10))



plot(TreeTune_housePricing)


treeBestTune <- TreeTune_housePricing$bestTune
# cp (complexity parameter)
# 1 1e-04


fit.model <- rpart(house_train[,1]~., data = house_train, cp = treeBestTune)


fittedModelPlot <- rpart.plot(fit.model, main = "Fitted Model")
```

Figure 5: Decision Tree Tuned Plot



Now we get into the state of the art supervised learning models widely used in business settings. These are mostly used in almost every machine learning competitions, data science competitions, and also in most of the businesses.

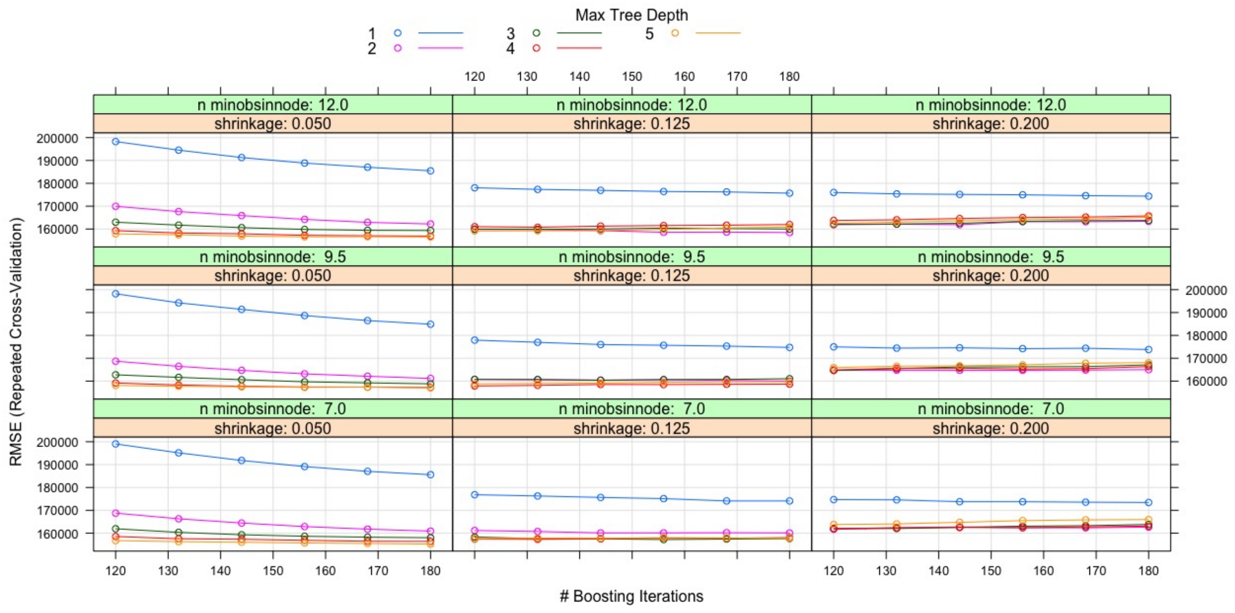
5.6 Gradient Boosting Machine (GBM)

By combining predictions from multiple decision trees, a GBM generates the final predictions. The tune grid for GBM contain number of trees; interaction depth simply put number of leaves; shrinkage which is learning rate; minimum observations in a node which is minumum number of observations at the terminal node of the tree. This algorithm will take some time to execute.

```r
gbm.grid <- expand.grid(n.trees = seq(from = 120, to = 180,
                                      length.out = 6),
                        interaction.depth = seq(1,5),
                        shrinkage = seq(from = .05, to = 0.2,
                                        length.out = 3),
                        n.minobsinnode = seq(from = 7, to = 12,
                                             length.out = 3))

cv.control_house <- trainControl(method = "repeatedcv",
                                 repeats = 2, number = 10)

GbmTune_housePricing <- train(y = house_train[,1], x = house_train[,2:13],
                              tuneGrid = gbm.grid,
                            method = "gbm",
                            trControl = cv.control_house)

gbmBestTune <- GbmTune_housePricing$bestTune
```

```
# n.trees interaction.depth shrinkage n.minobsinnode

# 180                     5       0.05             7


plot(GbmTune_housePricing)
```

Figure 6: GBM Tuned Plot



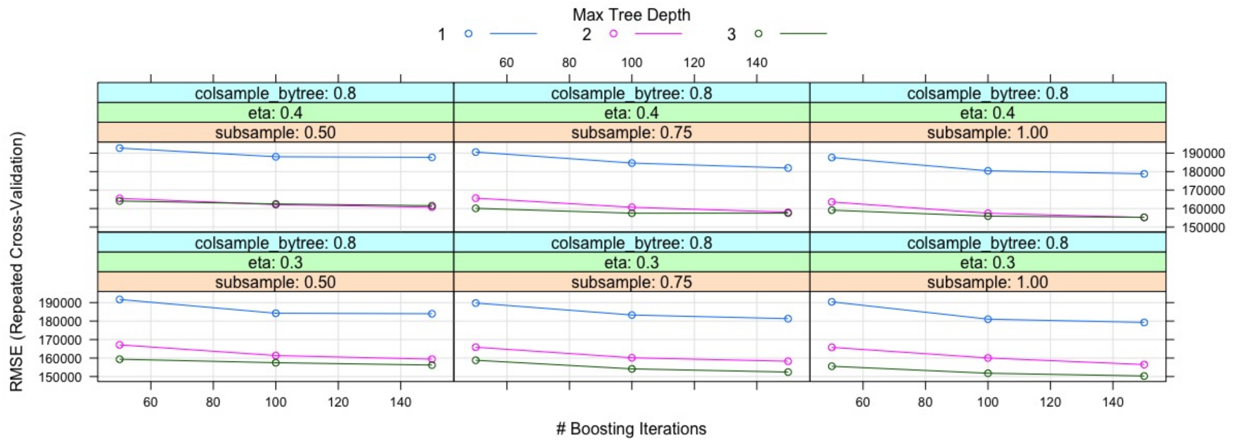
A minimum RMSE is achieved under 180 trees, 5 leaves for each tree, shrinkage of 0.05, and 7 observations at the terminal node of each tree. I could increase the leaves for each node and re-run the code. When I closely observe the plot 6, *interaction.depth=5* is converging.

5.7 Extreme Gradient Boosting Machine (XGBoost)

XGBoost is another state of the art, popular boosting algorithm. This is simply an advanced improved version of the simple GBM. The algorithm of XGBoost is similar to GBM but with sequential tree building and correcting the errors made by the previous trees.

```r
designMatXGB <- model.matrix(lm(housePrice~.,data=house_train))
designMatXGB <- designMatXGB[,-1]

# set up the cross-validated hyper-parameter search
xgb_grid <- expand.grid(eta = seq(from = 0.01, to = 0.2,
                                  length.out = 3),
                        max_depth = seq(from = 3, to = 10,
                                        length.out = 3),
                        colsample_bytree = seq (from = 0.5, to = 1,
                                                length.out = 3),
                        nrounds = seq(from = 100, to = 500,
                                      length.out = 3),
                        gamma = 0,  min_child_weight = 1,
                        subsample = seq(from = 0.5, to = 1,
                                        length.out = 3))


xgbTune_housePrice <- train(y = house_train[,1], x = designMatXGB,
                           method = "xgbTree",
                           trControl = trainControl(method="repeatedcv",
                                                    repeats = 2, number = 10),
                           tuneGrid = xgb_grid)


xgbBestTune <- xgbTune_housePrice$bestTune


xgbresults <- xgbTune_housePrice$results


plot(xgbTune_housePrice)
```

Figure 7: XGBoost Tuned Plot



Least RMSE is achived under nrounds = 150, max_depth = 3, η = 0.3, γ = 0, colsample_bytree = 0.8, min_child_weight = 1, and subsample = 1. One could argue that hyperparameters γ and *min_child_weight* are kept constant and I could improve the prediction by changing them. Yes, I could change them but for simplicity I use a constant value for them.

Now we get into not so well known supervised learning models. These are not discussed a lot in the community. There is nothing wrong with them. Perhaps they are not discussed a lot because of non-fancy names (:-D). My goal is to learn about different learning models. So I provide them in this exercise.

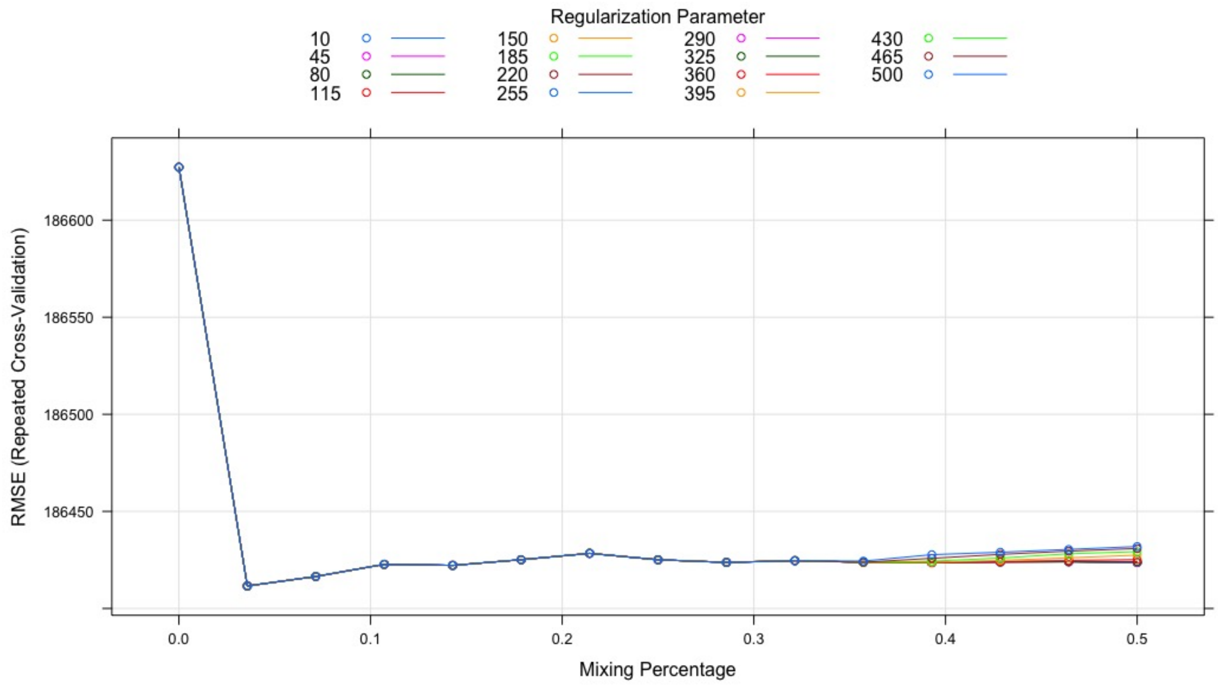
5.8 Elastic Net (ENet)

This method is similar to lasso and ridge regression. In fact, this is a combination of lasso and ridge regression. α and λ which are the penalty parameters of lasso and ridge regression respectiveky are passed as grid in the modeling.

```
designMatENET <- model.matrix(lm(housePrice~.,data=house_train))
designMatENET <- designMat[,-1]


Enet_grid <- expand.grid(alpha = seq(0,.5,length.out=15),
```

```r
                               lambda = seq(10,500,length.out=15))

ENetTune_housePrice <- train(y = house_train[,1], x = designMatENET,
                           method = "glmnet", tuneGrid = Enet_grid,
                           trControl = trainControl(method="repeatedcv",
                                                    repeats = 2, number = 10))

ENetBestTune <- ENetTune_housePrice$bestTune
#          alpha lambda
# 0.03571429    500

ENetResults <- ENetTune_housePrice$results

plot(ENetTune_housePrice)
```

Figure 8: ENet Tuned Plot



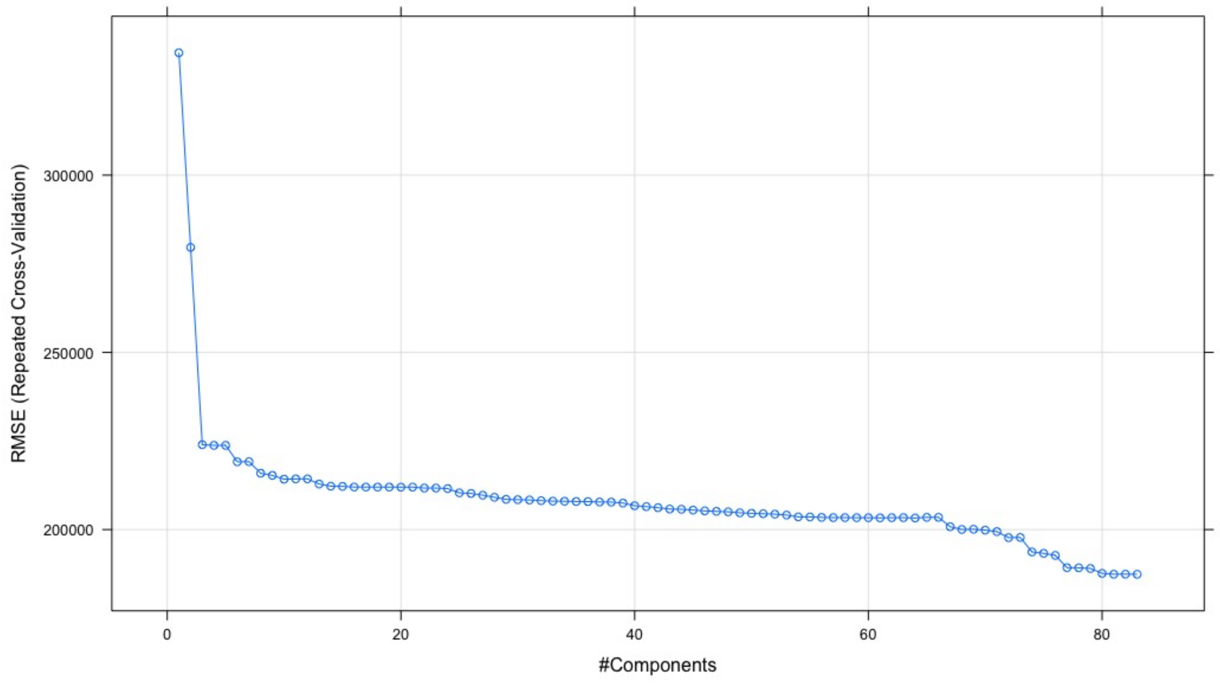
From the above plot 8, it is clear that RMSE is minimized under $\alpha = 0.0357$ and $\lambda = 500$.

5.9 Principal Component Regression (PCR)

PCR is based on Principal Component Analysis (PCA).

```r
designMatPCR <- model.matrix(lm(housePrice~.,data=house_train))
designMatPCR <- designMat[,-1]

pcrTune_housePrice <- train(y = house_train[,1], x = designMatPCR,
                            method = "pcr",
                            preProcess = c("center","scale"),
                            trControl = trainControl(method = "repeatedcv",
                                                     repeats = 2, number = 10),
                            tuneLength = 120)

pcrBestTune <- pcrTune_housePrice$bestTune
# ncomp
# 83
pcrResults <- pcrTune_housePrice$results

plot(pcrTune_housePrice)
```

Figure 9: PCR Tuned Plot

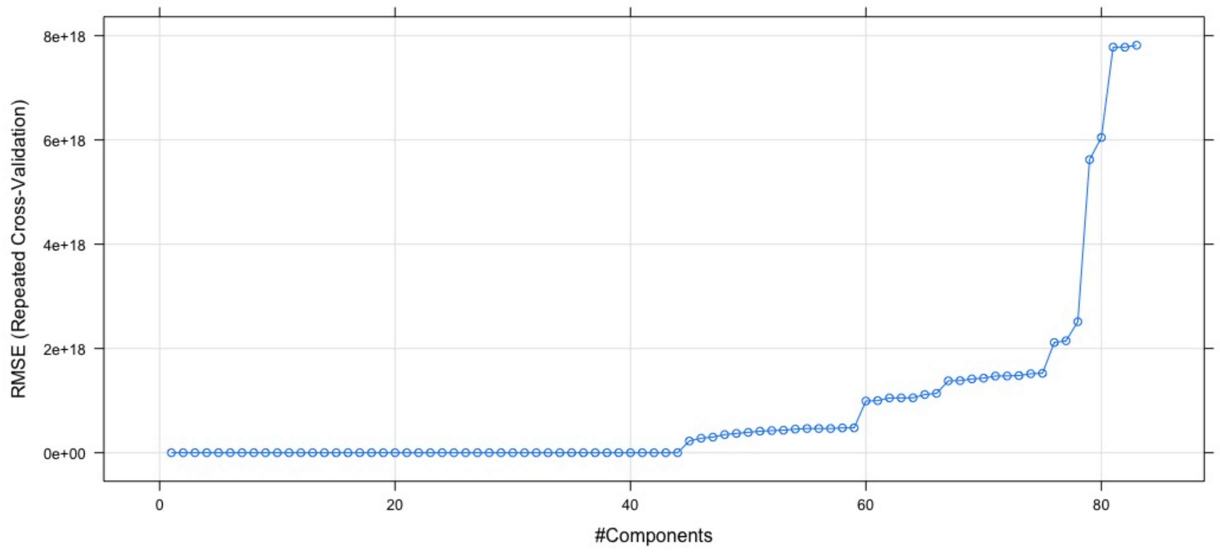


5.10 Partial Least Squares (PLS)

```
designMatPLS <- model.matrix(lm(housePrice~.,data=house_train))
designMatPLS <- designMat[,-1]


plsTune_housePrice <- train(y = house_train[,1], x = designMatPLS,
                            method = "pls",
                            preProcess = c("center","scale"),
                            trControl = trainControl(method = "repeatedcv",
                                                     repeats = 2, number = 10),
                            tuneLength = 120)


plsBestTune <- plsTune_housePrice$bestTune
# ncomp
```

```r
# 22

plsResults <- plsTune_housePrice$results

plot(plsTune_housePrice)
```

Figure 10: PLS Tuned Plot

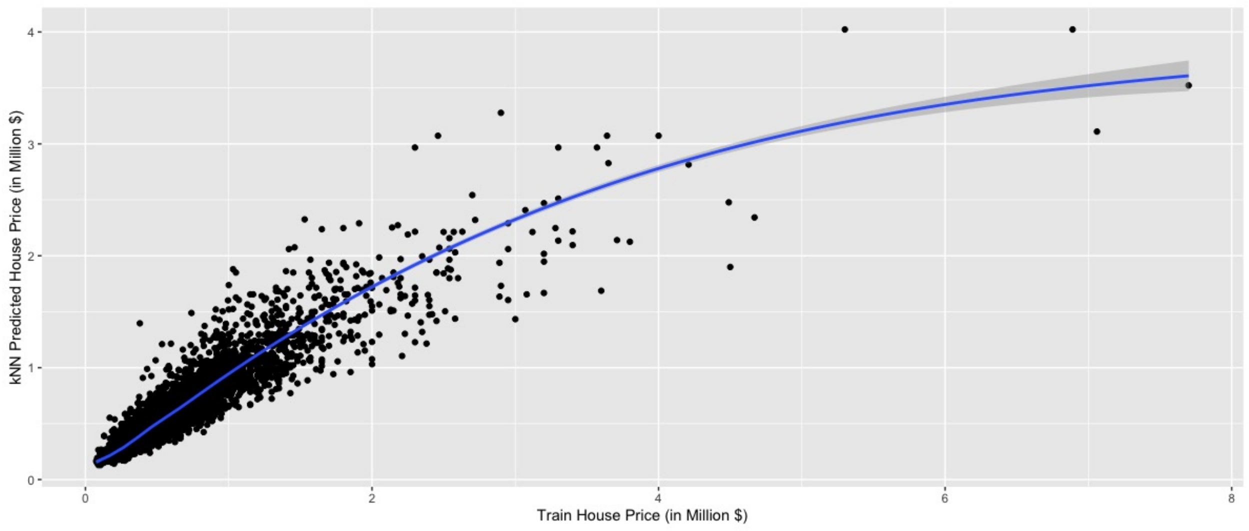


6 Prediction plots of the above fitted models

Now to see how the above supervised learning models perform, I predict the training set price with the fitted models. I argue that it is okay to predict the same training price with the model built on the same data. I perform this small exercise to visualize the model performance and later select the models for stacking purpose (stacking is explained in next subsequent sections).

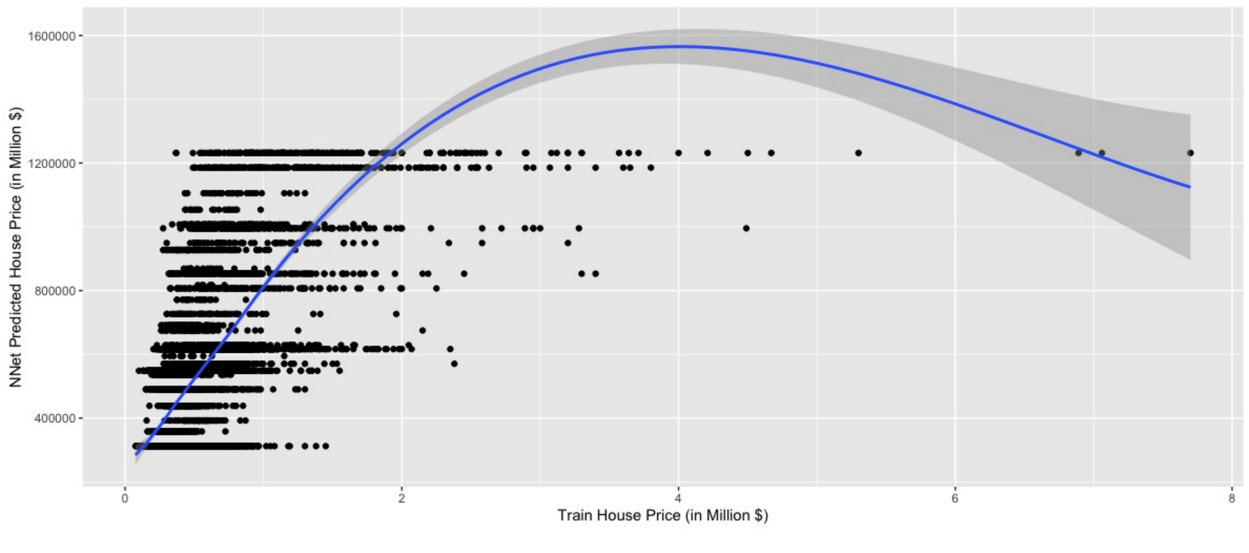
6.1 k-Nearest Neighbour Prediction

Figure 11: kNN Prediction Plot



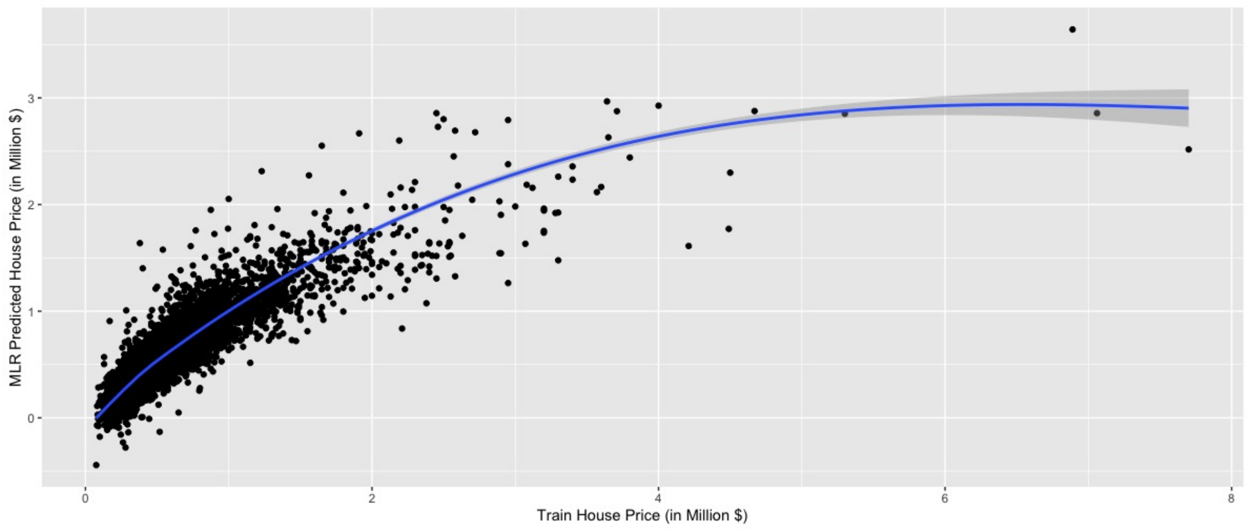
6.2 Neural Network Prediction

Figure 12: NNet Prediction Plot



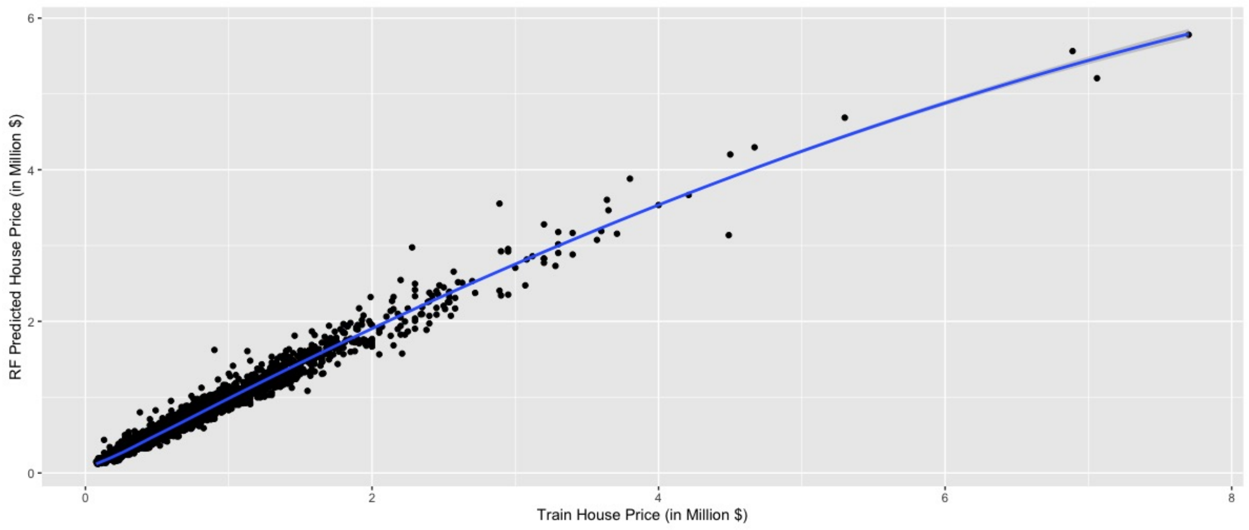
6.3 Multiple Linear Regression (MLR) Prediction

Figure 13: MLR Prediction Plot



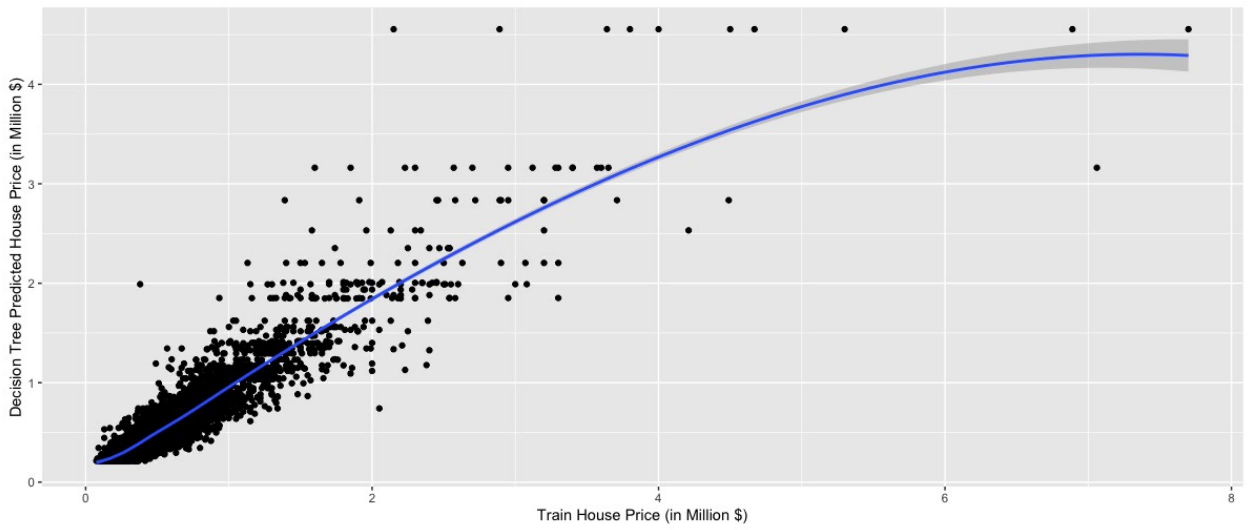
6.4 Random Forest (RF) Prediction

Figure 14: RF Prediction Plot



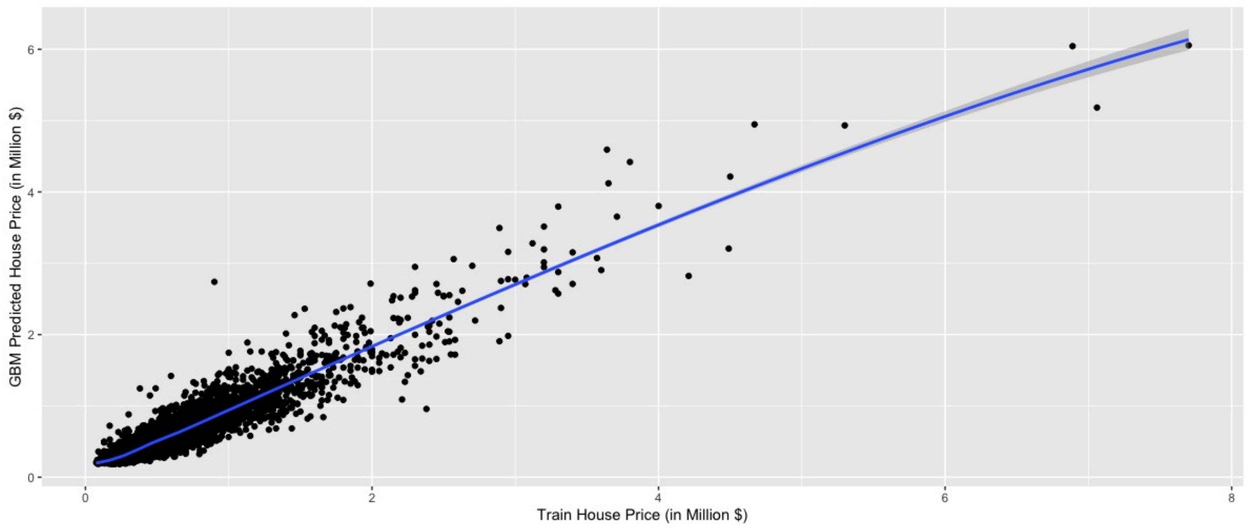
6.5 Decision Tree Prediction

Figure 15: Decision Tree Prediction Plot



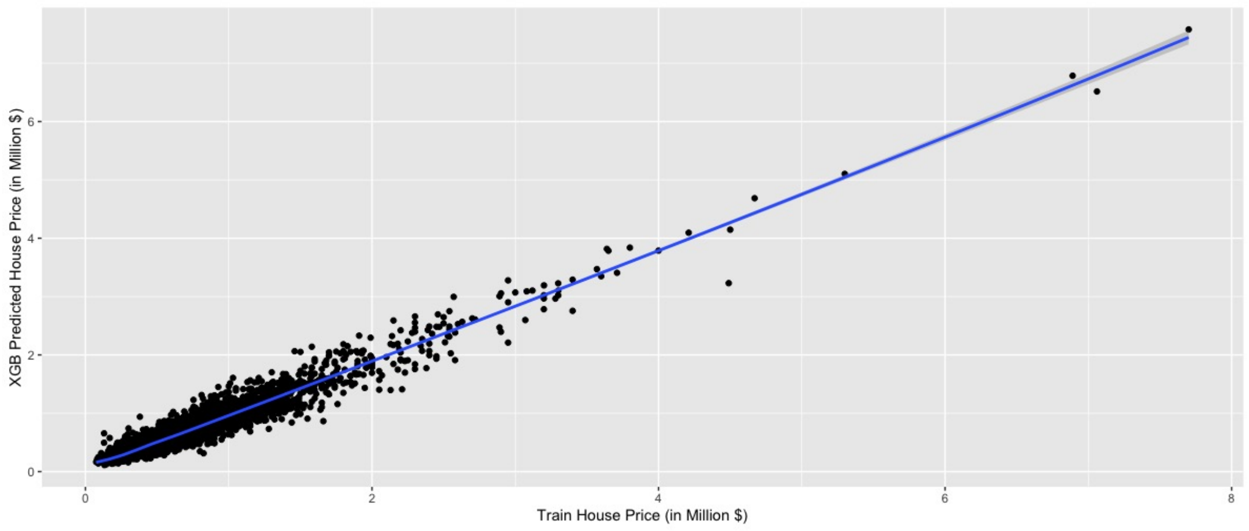
6.6 Gradient Boosting Machine (GBM) Prediction

Figure 16: GBM Prediction Plot



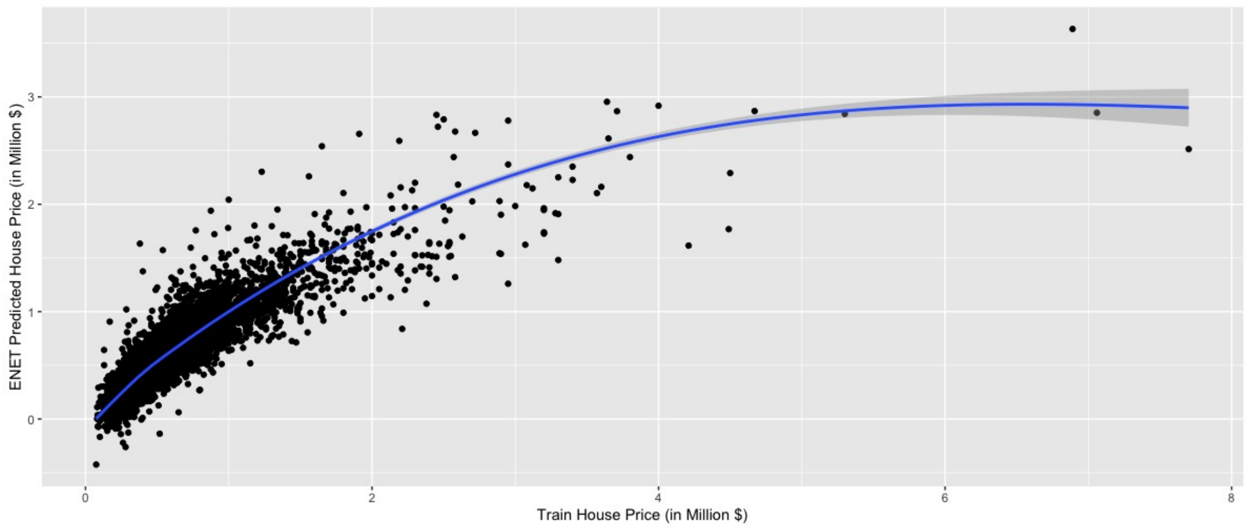
6.7 Extreme Gradient Boosting Machine (XGBoost) Prediction

Figure 17: XGBoost Prediction Plot



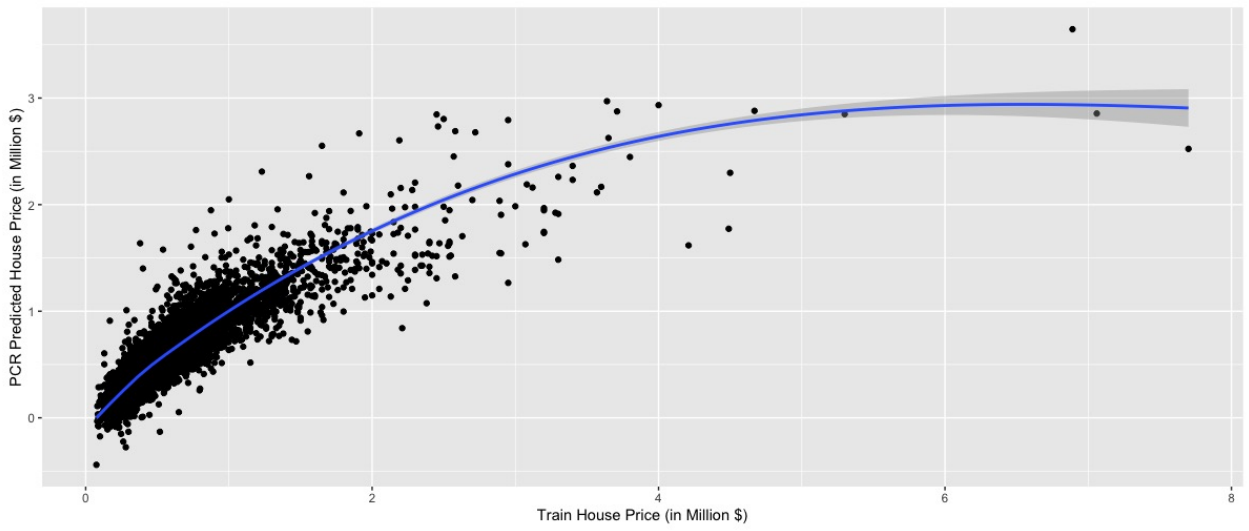
6.8 Elastic Net (ENet) Prediction

Figure 18: ENet Prediction Plot



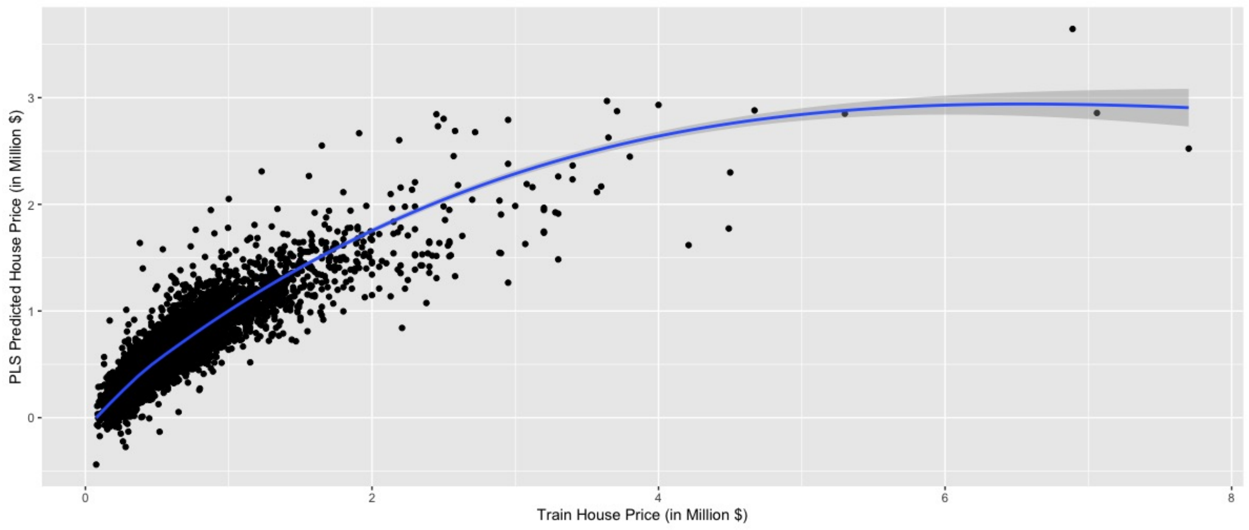
6.9 Principal Component Regression (PCR) Prediction

Figure 19: PCR Prediction Plot



6.10 Partial Least Squares (PLS) Prediction

Figure 20: PLS Prediction Plot


PLS Predicted House Price (in Million $)
Train House Price (in Million $)