# Name the Price: Analysis and Prediction of House Prices with Machine Learning

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 the Kaggle data science competition. I learned a lot during and after the competition. I believe in the mantra "learning by doing" (coined by the famous American philosopher *John Dewey*). So I have been working on this project by making multiple changes, trying different algorithms, and eventually improving my modeling techniques and prediction power.

## 2 Data

The data are free to download from the Kaggle website. I downloaded the data a while ago and have 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 bedrooms, number of bathrooms, latitude, and so on) and (2) Testing data containing only the characteristics of the house. My goal simply is to predict the house prices in the testing data. In order to do that, first I analyze 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 variables (one could argue it is a continuous variable, but I stay away from that argument in this analysis). So this could be considered a regression problem. From my rigorous training in Economics, I can immediately identify a problem with the data. Endogeneity (the explanatory variables may be correlated with the error term); 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 processes 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. My primary objective of this work is to use supervised learning methods to predict 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.

## 2.1 Variable description

The training data set contains the following variables, and testing data contains the same variables except *price* which is our variable of interest for prediction.

- 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 the house
- bathrooms - number of bathrooms in the house
- sqft_living - square footage of the house
- sqft_lot - lot area
- floors - total floors/levels in the house

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

```
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)
```

```
# 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)
```

# 3 Preliminary analysis:

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

## 3.1 Simple correlation analysis:

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

## 3.2 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 significance. From ANOVA results, I conclude that almost all the explanatory variables are highly significant in explaining the house price. However, the variable importance method concludes that bedrooms, sqft_living, waterfront, grade, yr_built, and lat are the

most important variables. By further analysis, I find other variables are important as well.

### 3.3 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 the primary features 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 selected sqft_living, grade, lat, long, sqft_living15, yr_built, and waterfront as the most important variables.

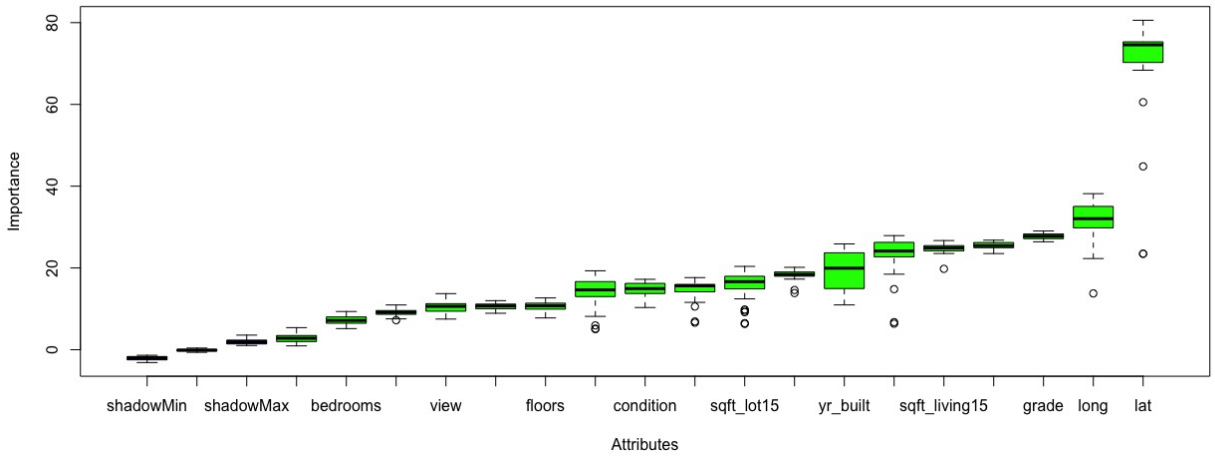
### 3.4 Boruta feature selection algorithm to determine the variable importance:

```
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 which include bathrooms, bedrooms, condition, floors, grade, and 13 more. No attributes are 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. So I keep all the variables for prediction. Now, I read the description of each variable and use my judgment to drop, modify, or change the data type. I make the following changes in both training and testing data.

1. I drop *date* from the data since all houses sold are in the same year.
2. I modify the data type of *zipcode* to *factor* variable since it is not a integer/numeric and also not a continuous variable.
3. I modify the data type of *waterfront* to *factor* variable since it's data type is binary (0 or 1).
4. I modify the data type of *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.

```
housePrice <- train_data$price
# Here I drop date from the data since the houses sold are
# in the same year.
train_in <- train_data %>% select(bedrooms:sqft_lot15)
house_train <- cbind(housePrice,train_in) %>% as.data.frame()

# I change zipcode to factor variable since it is not an
# integer/numeric variable
house_train$zipcode <- as.factor(house_train$zipcode)

# I change the waterfront and condition variable to factor variable.
# Waterfront is a binary variable and condition is the rating variable.
# So we change them to factor.
house_train$waterfront <- as.factor(house_train$waterfront)
house_train$condition <- as.factor(house_train$condition)

# I drop sqft_living, sqft_lot,sqft_above, and sqft_basement.
# sqft_living15 and sqft_lot15 contain most recent information
# about the sqft information
house_train <- house_train %>% select(-sqft_above, -sqft_basement,
                                      -sqft_living, -sqft_lot)

# Replicating the same changes in test data set
house_test <- test_data %>% select(bedrooms:sqft_lot15)
house_test$zipcode <- as.factor(house_test$zipcode)
house_test$waterfront <- as.factor(house_test$waterfront)
house_test$condition <- as.factor(house_test$condition)
```

```
house_test <- house_test %>% select(-sqft_above, -sqft_basement,
                                    -sqft_living, -sqft_lot)
```

# 4 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 (CV) with repeats. Of course, repeated cross-validation will increase the execution time. But I prefer good results over time. With the availability of increased computing power, we can use HPC or even use parallel computing to execute the code. So repeated cross-validation is used with 2 repeats. Since I am doing this on 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. Simply I center and scale the data such that I don't have to think about the units of the variables included in the modeling.

In this exercise, the metrics I choose for the best fit are Root Mean Squared Error (RMSE) and R-Square. A low RMSE and a high R-square are considered the best fit for the exercise.

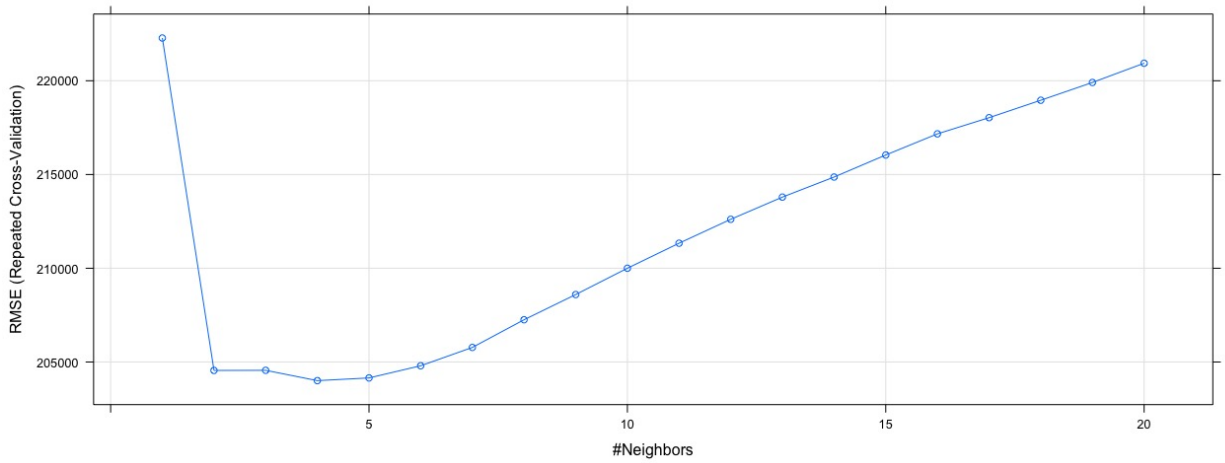
## 4.1 k-Nearest Neighbour (kNN)

The tune grid I provide for kNN is from 1 to 20. I give the algorithm to choose nearest neighbors from 1 to 20.

```
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
```

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.

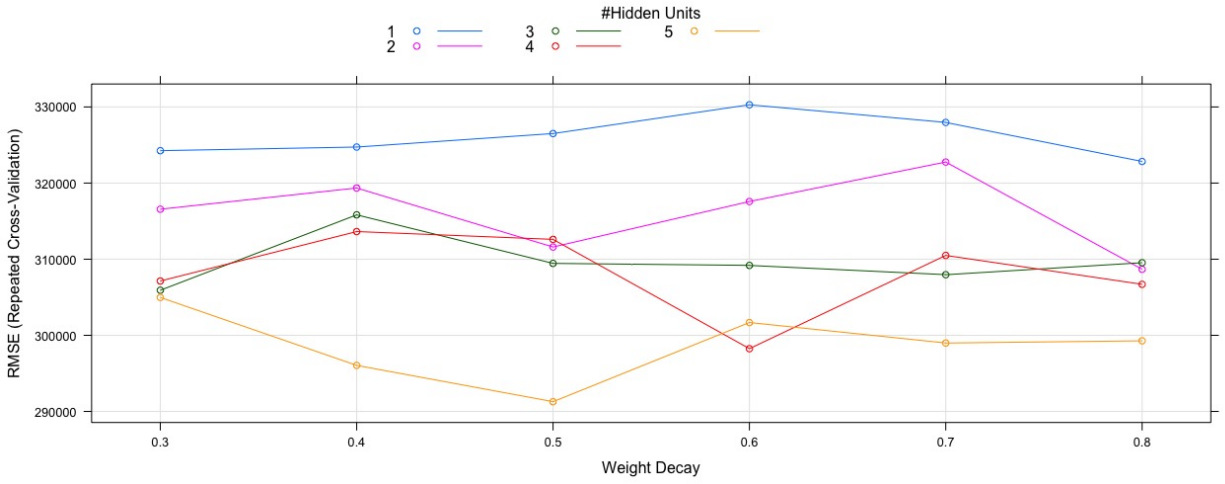
## 4.2 Neural Network

First I 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 the global optimum. The values chosen for the size and decay are widely practiced in the literature. *Size* is the number of 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*.

## 4.3 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 |

## 4.4 Random Forest (RF)

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

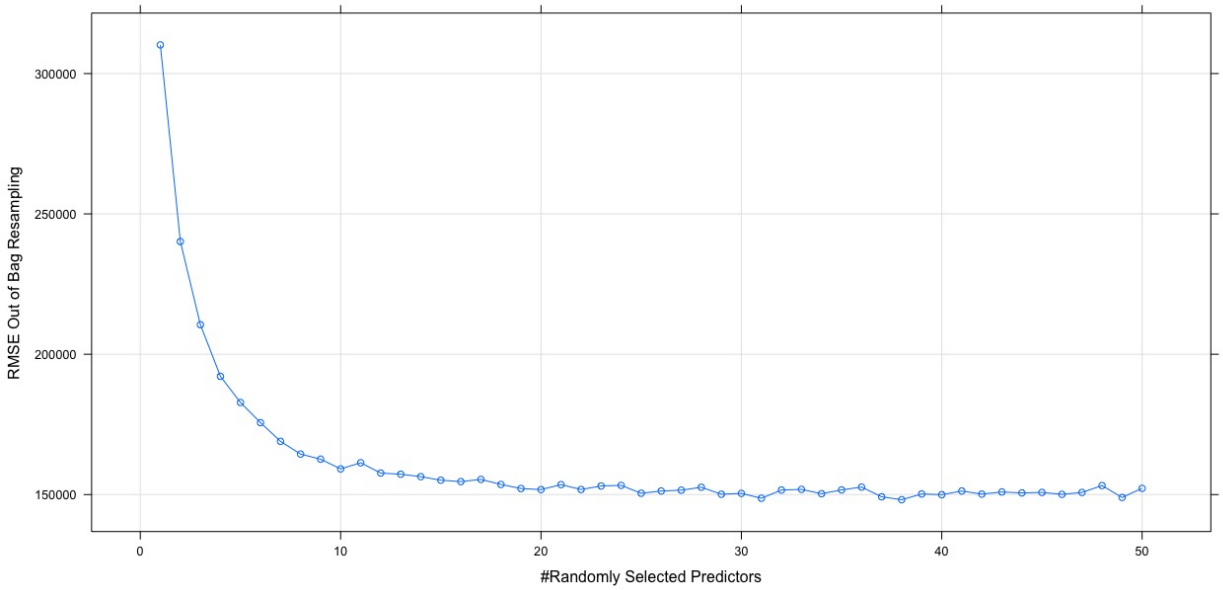
In the random forest model, the original training data is randomly sampled with replacement generating small subsets of data. These subsets are widely known as bootstrap samples. For the tune grid, I provide *mtry*, which is the number of variables randomly sampled as candidates at each split. I also provide *ntree*, which is basically 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 random forest model.

At each bootstrap, the algorithm leaves out some rows and trains with leftover data. Once the training is finished, the model is fed the left out rows and asked to predict the outcome. This

is repeated multiple times and the OOB score is computed by simply adding up the number of correctly predicted rows. We can also use a 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]


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



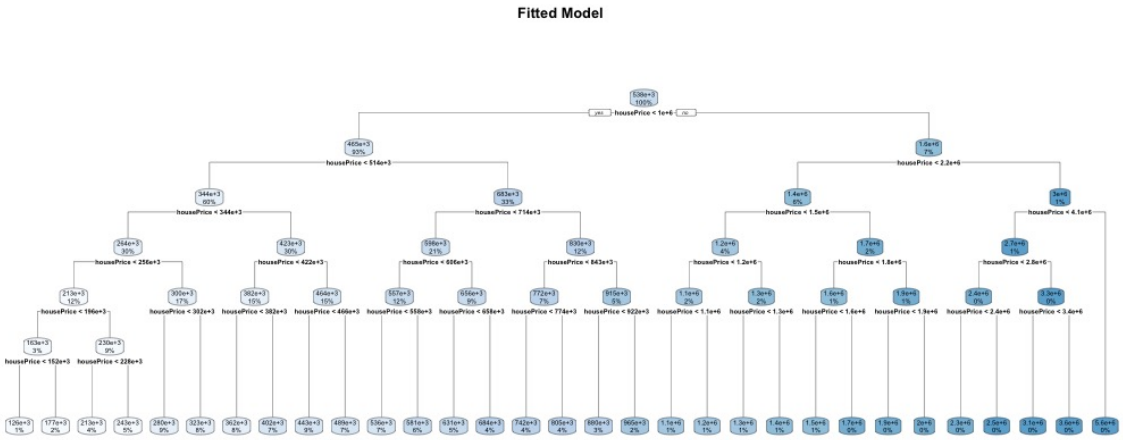

Under *mtry=38*, the model achieved the lowest RMSE.

## 4.5 Decision Tree

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

```
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. These are mostly used in almost every machine learning competition, data science competition, and also in most businesses.

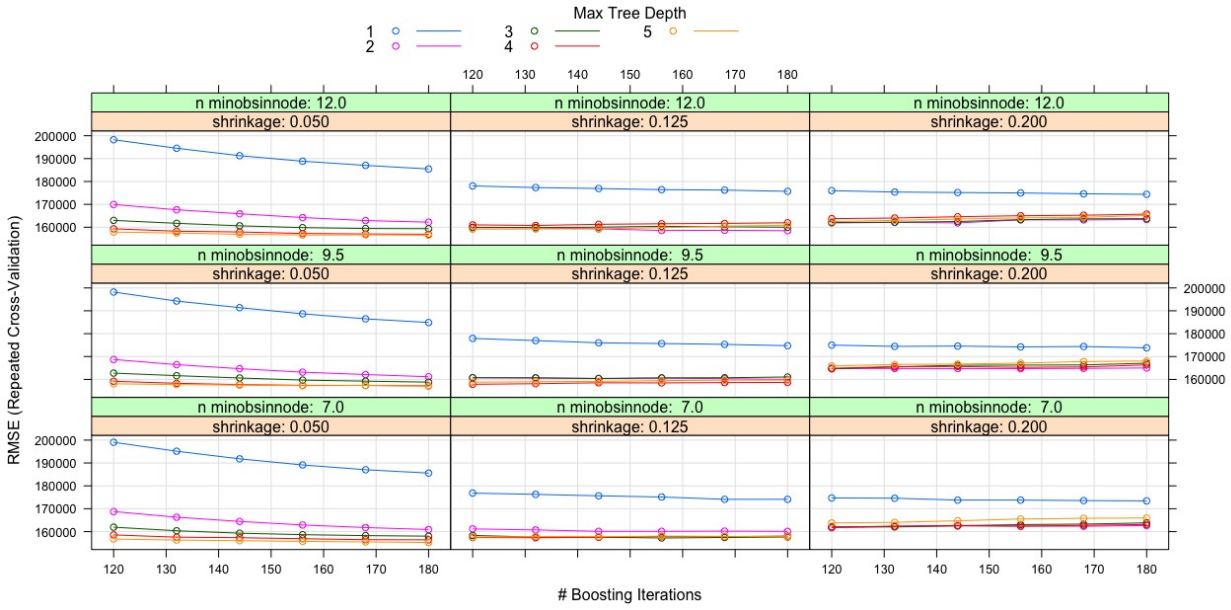
## 4.6 Gradient Boosting Machine (GBM)

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

```
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. So I do not increase the interaction depth.

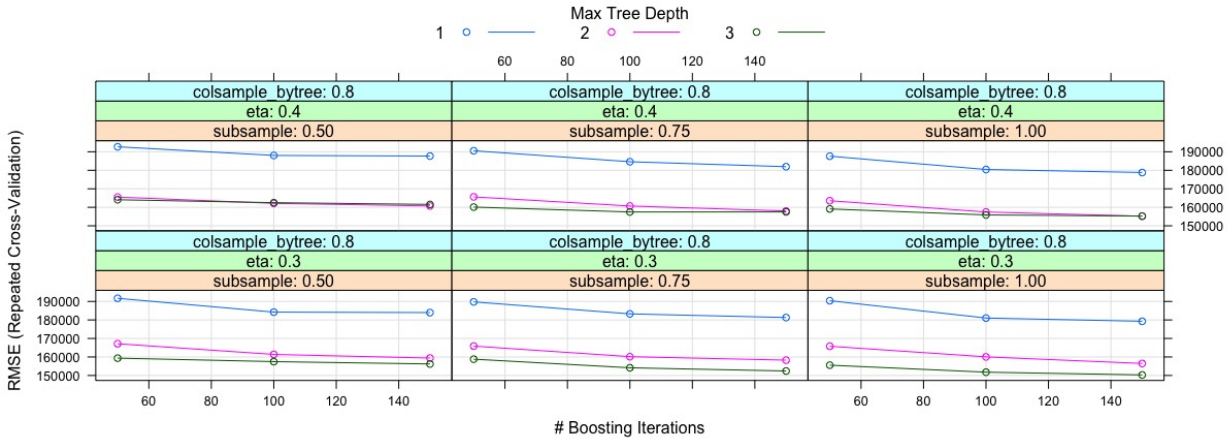
## 4.7 Extreme Gradient Boosting Machine (XGBoost)

XGBoost is another state-of-the-art, popular boosting algorithm. This is 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.

```
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




The minimum RMSE is achieved with nrounds = 150, max_depth = 3, $\eta$ = 0.3, $\gamma$ = 0, colsample_bytree = 0.8, min_child_weight = 1, and subsample = 1. One could argue that the $\gamma$ and *min_child_weight* hyperparameters are kept constant and you could improve the prediction by changing them. Yes, you could change them, but for the sake of simplicity, I'll use a constant value for them.

Now we come to the not so well known supervised learning models. These are not discussed

much in the community. There's nothing wrong with them. Maybe because of their simple names (:D) they are not discussed much. My goal is to get to know different learning models. Therefore I make them available in this exercise.

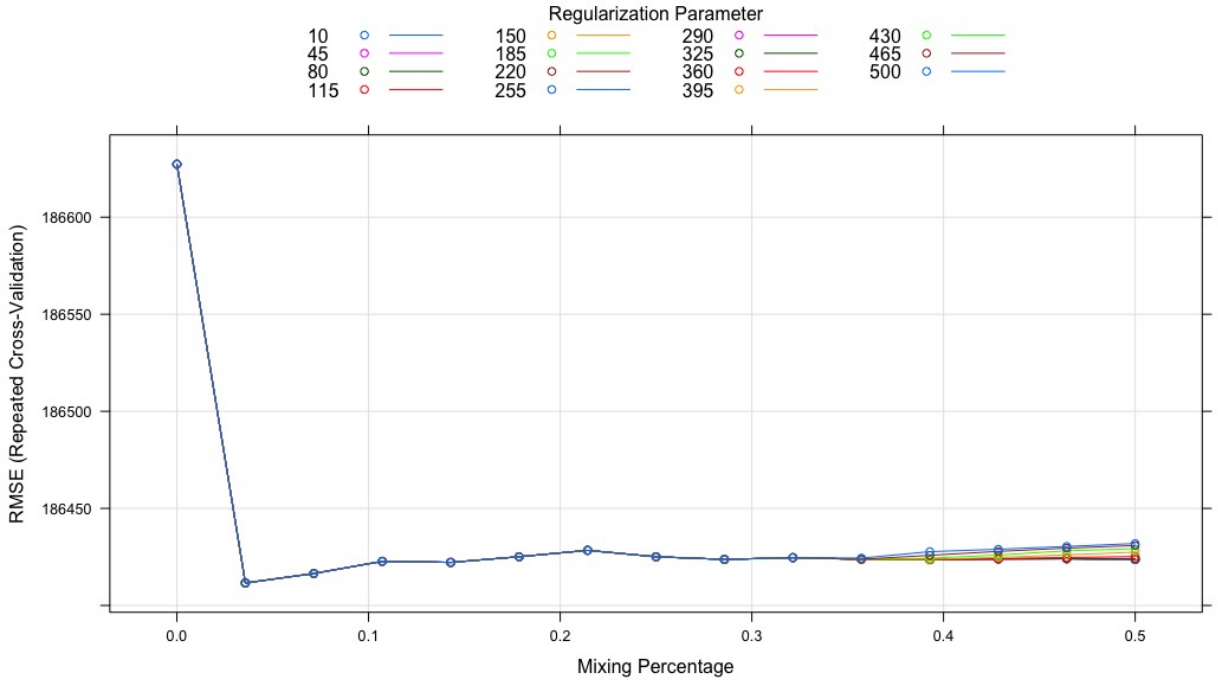
## 4.8 Elastic Net (ENet)

This method is similar to lasso and ridge regression. In fact, this is a combination of lasso and ridge regression. $\alpha$ and $\lambda$ which are the penalty parameters of lasso and ridge regression respectively are passed as the grid in the modeling.

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

Enet_grid <- expand.grid(alpha = seq(0,.5,length.out=15),
                         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 $\lambda = 500$.

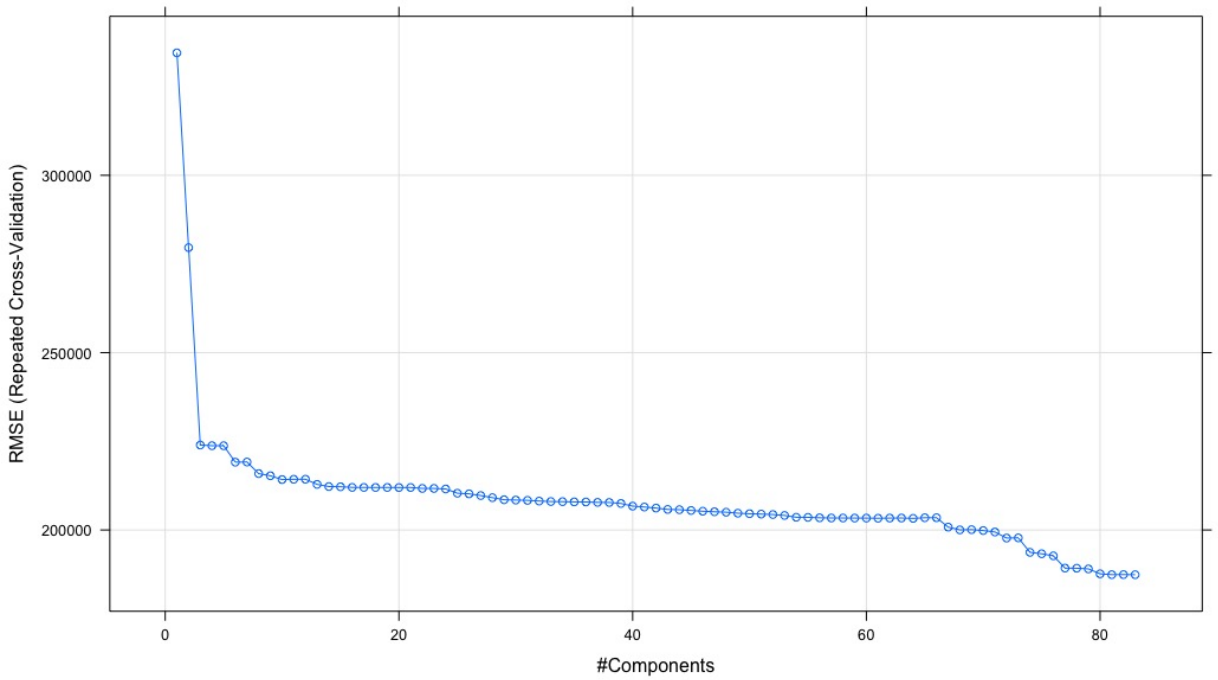
## 4.9 Principal Component Regression (PCR)

PCR is based on Principal Component Analysis (PCA).

```
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

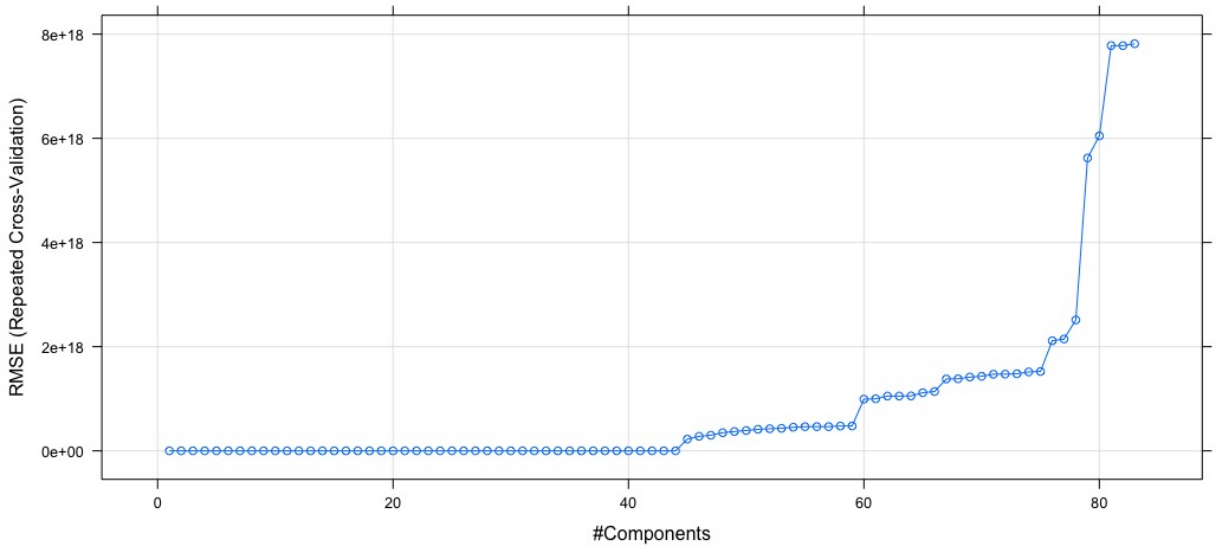



## 4.10 Partial Least Squares (PLS)

```
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
# 22

plsResults <- plsTune_housePrice$results

plot(plsTune_housePrice)
```

Figure 10: PLS Tuned Plot

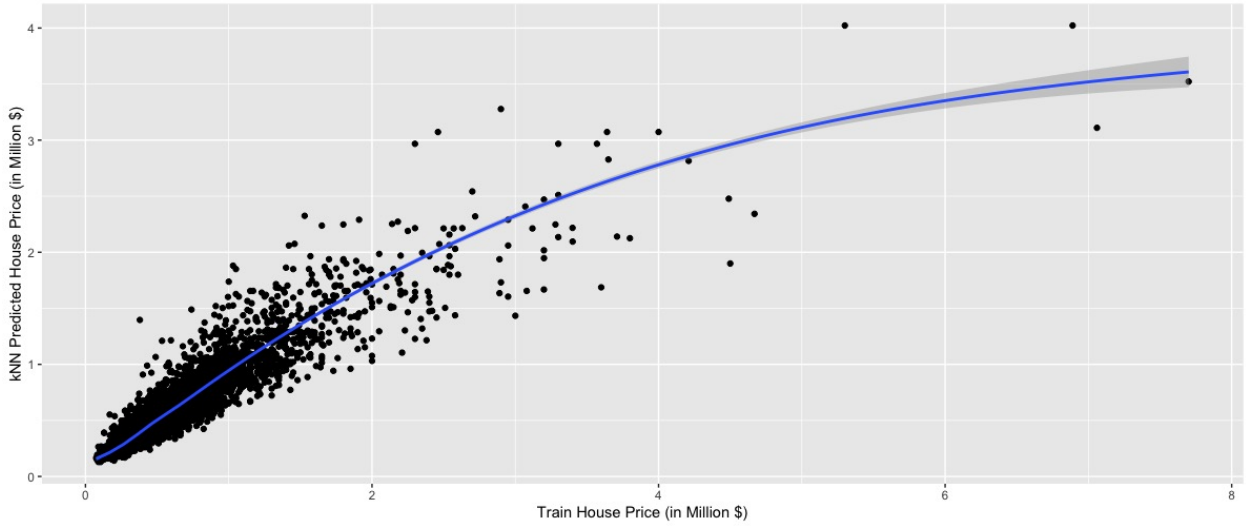


# 5 Prediction plots of the above fitted models

Now to see how the above-supervised learning models performed, 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.[1] I perform this small exercise to visualize the model performance and later select the models for stacking purposes (stacking is explained in the next subsequent sections).

## 5.1 k-Nearest Neighbour Prediction

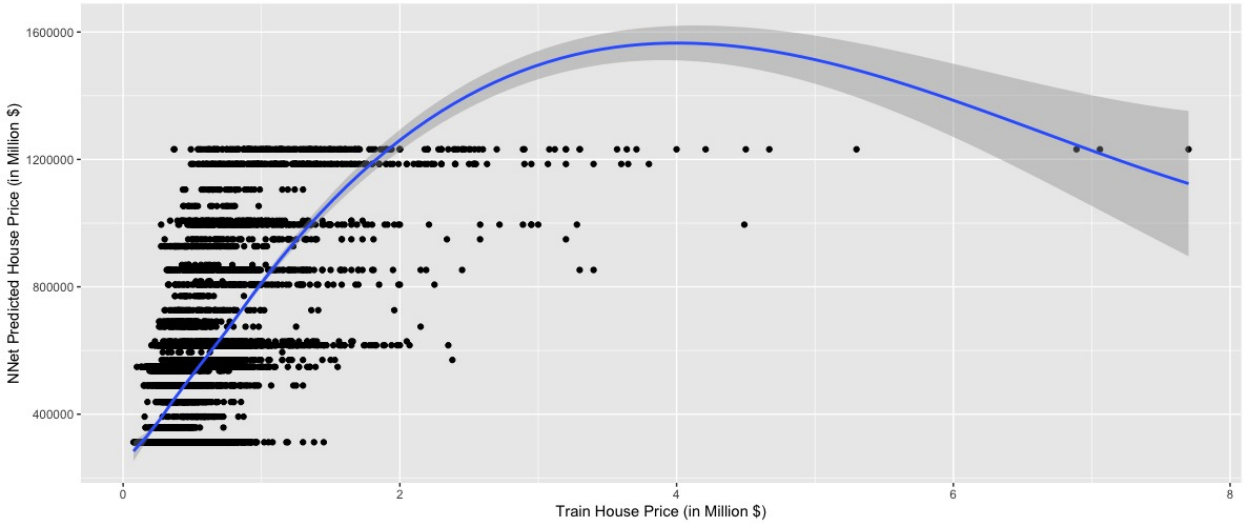
Figure 11: kNN Prediction Plot




[1] All the models are fit either by repeated cross-validation or with out-of-bag methods.

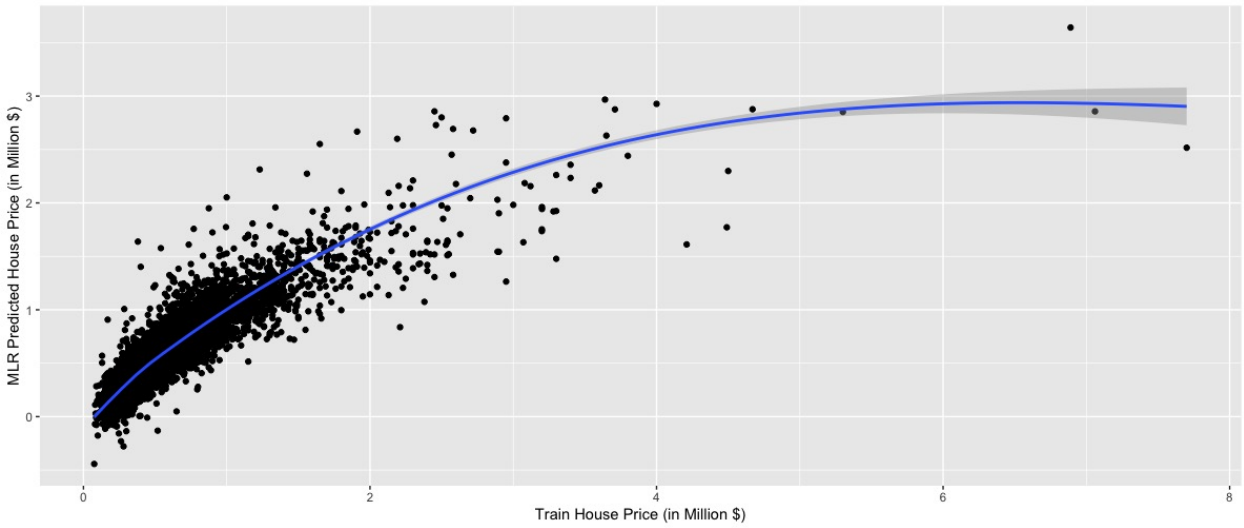
## 5.2 Neural Network Prediction

Figure 12: NNet Prediction Plot



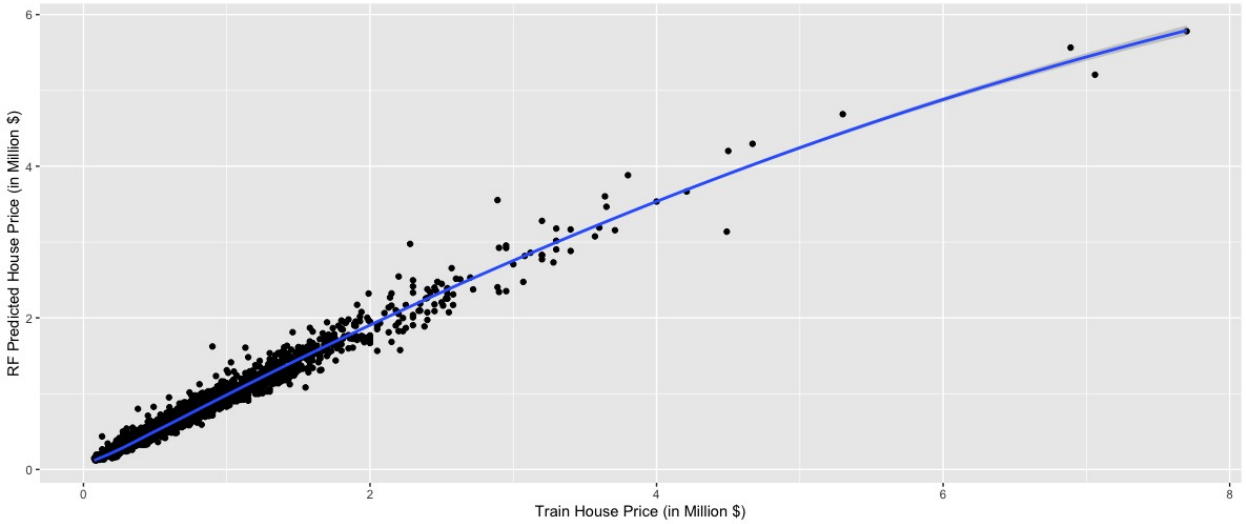

## 5.3 Multiple Linear Regression (MLR) Prediction

Figure 13: MLR Prediction Plot



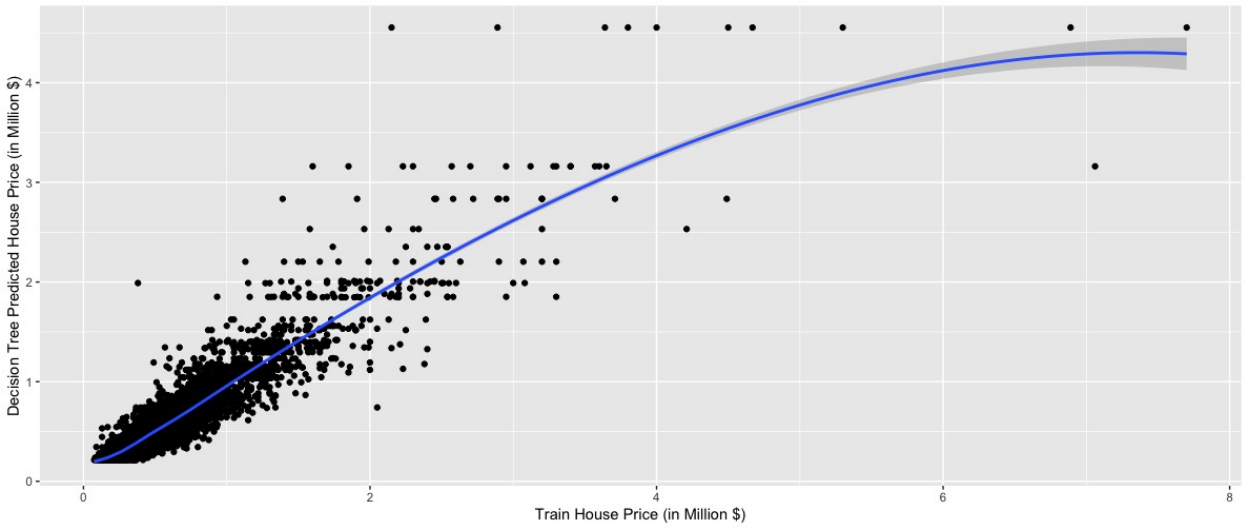
## 5.4 Random Forest (RF) Prediction

Figure 14: RF Prediction Plot



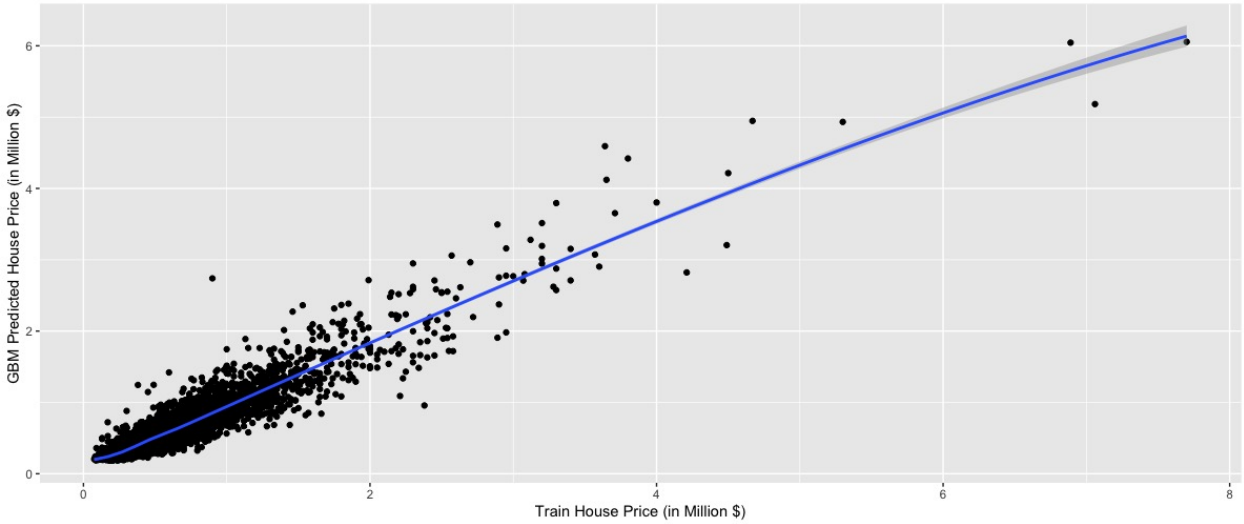

## 5.5 Decision Tree Prediction

Figure 15: Decision Tree Prediction Plot



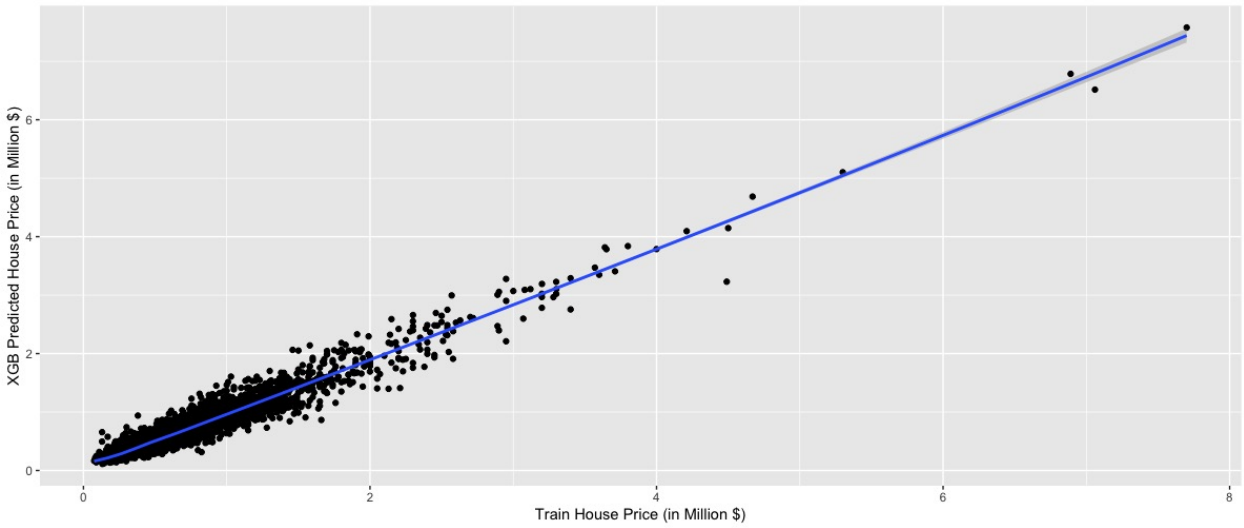
## 5.6 Gradient Boosting Machine (GBM) Prediction

Figure 16: GBM Prediction Plot



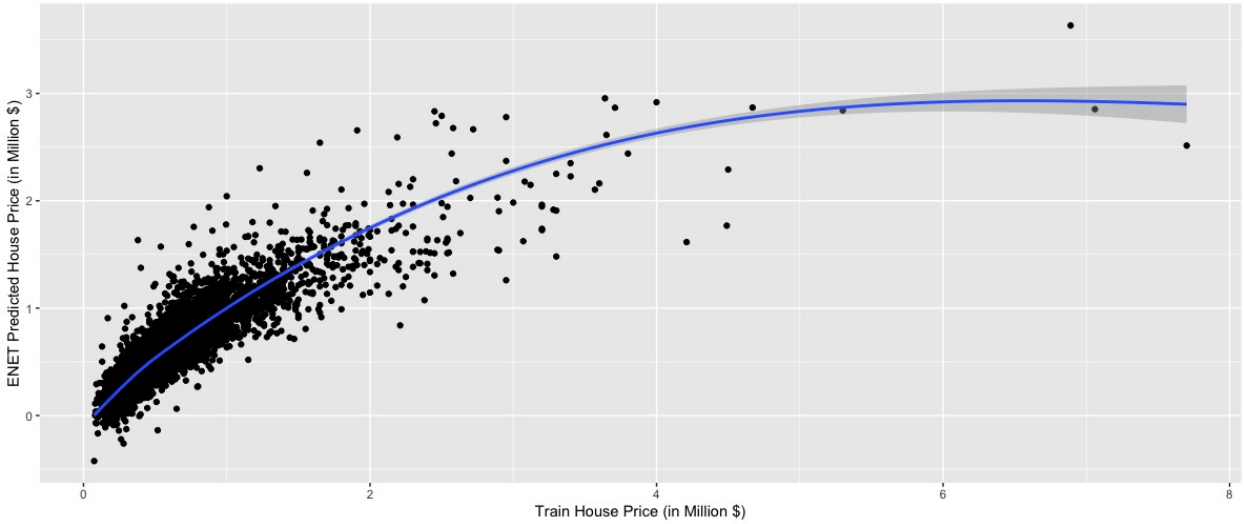

## 5.7 Extreme Gradient Boosting Machine (XGBoost) Prediction

Figure 17: XGBoost Prediction Plot



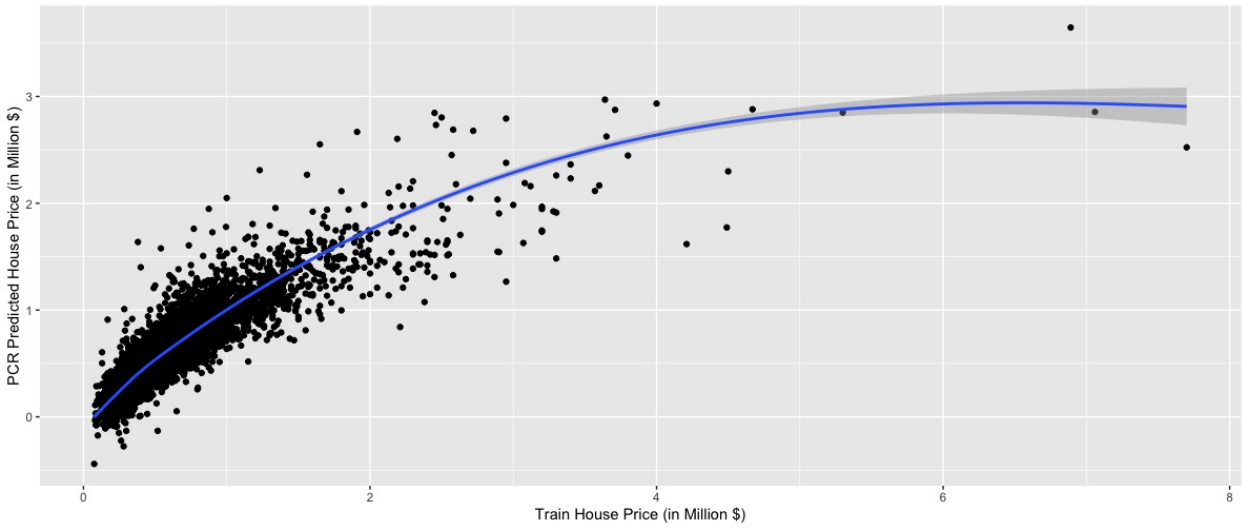
## 5.8 Elastic Net (ENet) Prediction

Figure 18: ENet Prediction Plot



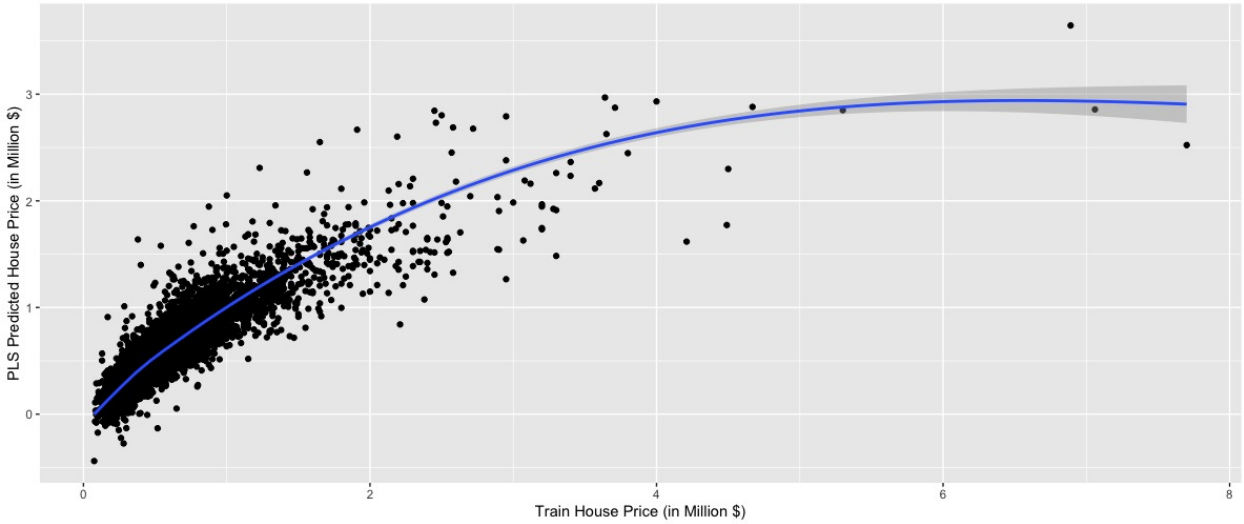

## 5.9 Principal Component Regression (PCR) Prediction

Figure 19: PCR Prediction Plot



### 5.10 Partial Least Squares (PLS) Prediction

Figure 20: PLS Prediction Plot




# 6 Stacking

Stacking is one of the ensemble methods that has become popular in recent years and became a go-to method to use for any supervised learning method. Simply put, it uses a meta-learning algorithm to combine and learn from different base learning models. Base learning models include all the methods I provided above. I use stacking to improve and increase my predictive power. First, I pick the base models to include in the meta-learning algorithm. I have to be extra careful when picking the base learning models. I simply cannot use all the above methods. From the prediction plots above it is apparent that some base models are very poor in predicting the house prices. So using them in the meta-learning algorithm is not ideal. Along with the prediction plots, I also use other metrics such as Root Mean Squared Error (RMSE) and R-squared to pick the base models. A linear relationship in the above plots, a low RMSE, and a high R-squared are used to pick the base models for meta-learning. By carefully analyzing the metrics, I picked: Extreme Gradient Boosting Machine (XGBoost), Random Forest (RF), gradient Gradient Boosting Machine (GBM), and k-Nearest Neighbour (kNN) for meta-learning.

Before jumping into meta-learning, I get the best tune of the base methods and arrange them.

```
# First I get the best tune from the above mentioned models
xgbBestTune_meta <- xgbBestTune
forestBestTune_meta <- forestBestTune
gbmBestTune_meta <- gbm_BestTune
knnBestTune_meta <- knnBestTune

# I arrange all the best tune and the models for stacking
method_meta <- c("xgbTree", "rf", "gbm", "knn")
parametersTuned_meta <-
  list(xgbBestTune_meta, forestBestTune_meta, gbmBestTune_meta,
       knnBestTune_meta)
```

In the code snippet below, I arrange the data for stacking by simply using the model fit to get the prediction. *metaPred1*, *metaPred2*, *metaPred11*, and *metaPred21* contain those data.

```
metaPred1<- matrix( NA, nrow = dim(house_test)[1],
                    ncol = length(method_meta) + 2)
metaPred2<- matrix( NA, nrow = dim(house_train)[1],
                    ncol = length(method_meta))

metaPred11<- matrix( NA, nrow = dim(house_train)[1],
                     ncol = length(method_meta) + 2)
metaPred21<- matrix( NA, nrow = dim(house_train)[1],
                     ncol = length(method_meta))

train <- house_train
test <- house_test
```

```
for (j in 1:length(method_meta)){

  modelfit <- train(housePrice~.,
                    data   = house_train,
                    method = method_meta[j],
                    preProc = c("center","scale"),
                    trControl = trainControl(method="none"),
                    tuneGrid = parametersTuned_meta[[j]])


  metaPred1[,j] <- predict(modelfit, newdata = house_test)
  metaPred2[,j] <- predict(modelfit, newdata = house_train)

  metaPred11[,j] <- predict(modelfit, newdata = house_train)
  metaPred21[,j] <- predict(modelfit, newdata = house_train)
}
```

Now I use *metaPred2* (which contains predictions from base models with the best tune), to fit another base model. I use a linear regression model (OLS), RF, and kNN to get the weights and best tunes.

```
emfit1_OLS <- lm(train$housePrice ~ metaPred2)

emfit2_RF <- train(y = train$housePrice,
                x = as.data.frame(metaPred2),
                tuneGrid = data.frame(mtry=1:50),
                method = "rf", ntree = 150,
```

```
                trControl = trainControl(method="oob"))

nnet.gridMeta <- expand.grid(size = seq(from = 1, to = 6, length.out = 6),
                             decay = seq(from = .3, to = .8, length.out = 6))

emfit2_kNN <- train(y = train$housePrice,
                    x = as.data.frame(metaPred21),
                    method = "knn",
                    preProcess = c("center","scale"),
                    tuneGrid = data.frame(.k=1:20),
                    trControl =
                      trainControl(method = "repeatedcv", repeats = 3, number = 10))

plot(emfit2_RF)

plot(emfit2_kNN)
```

Figure 21: RF Meta-Learning Tuned Plot

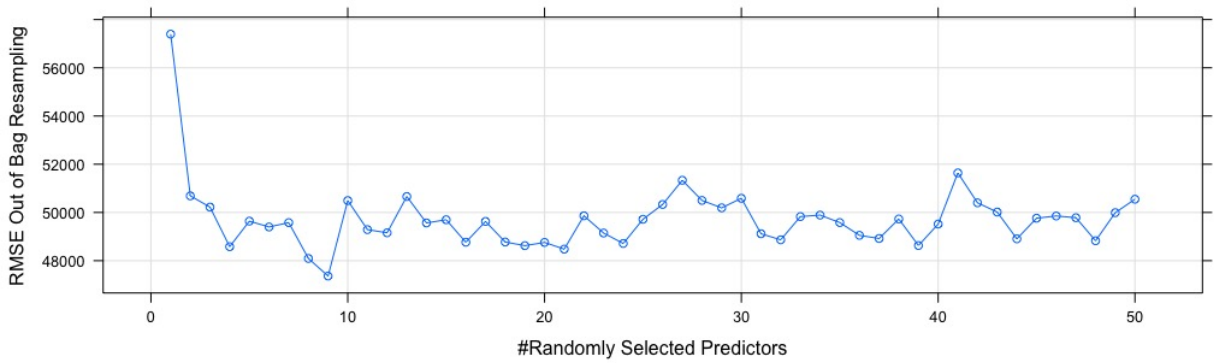
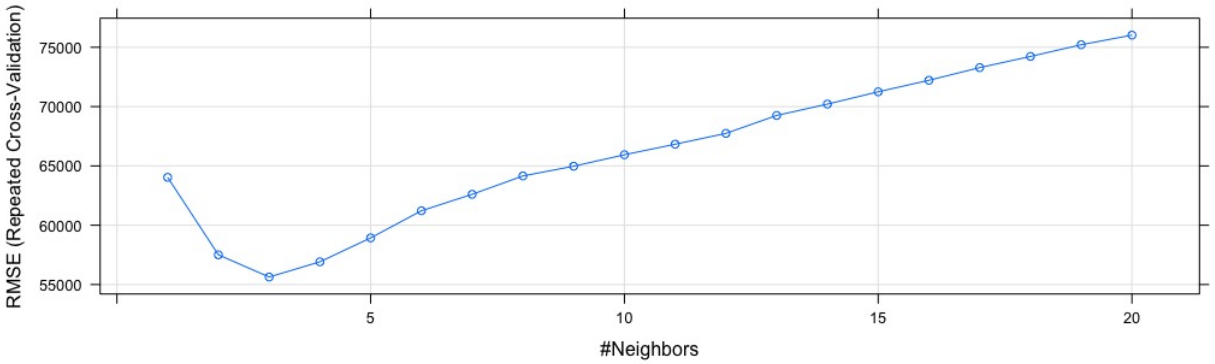


Figure 22: kNN Meta-Learning Tuned Plot




From the above best tune plots the lowest RMSE is achieved for *mtry=9* and *k=3* for RF and kNN respectively.

Finally, I perform the following procedure to get the meta-algorithm prediction: 1. Use the coefficients from the linear regression model as weights on the predictions of base models to predict the first meta price. 2. Use the best tune from kNN meta-model on the predictions of base models and the meta prediction in step 1. 3. Use the best tune from RF meta-model on the predictions of base models, the meta prediction in step 1, and the meta prediction in step 2.

The following code snippet is used perform the above procedure and arrange the predictions in a dataframe.

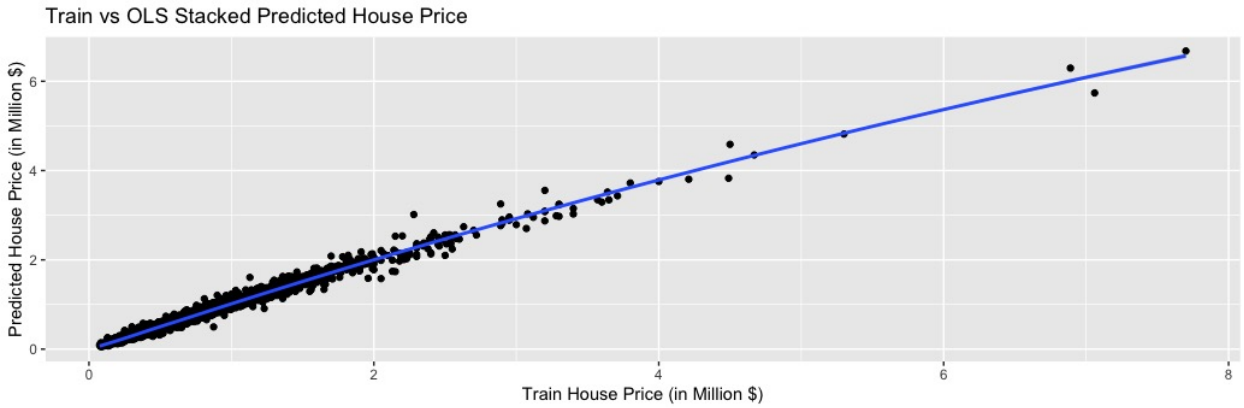
```
# Using the best fit of LM to assign weights for base models and predict
metaPred11[,5] <- cbind( matrix(1,nrow=dim(house_train)[1],ncol=1) ,
                         metaPred11[,-c(5,6)] ) %*% coef(emfit1_OLS)


# Using the best tune of RF on base models plus meta LM model and predict
metaPred11[,6] <- predict( emfit2_kNN, as.data.frame(metaPred11))
metaStack1<- as.data.frame(metaPred11)
colnames(metaStack1)[1:4]<- method_meta
```

```
colnames(metaStack1)[5:6]<- c("enOLS","enKNN")


# Using the best tune of kNN on base models + meta LM + meta RF
metaStack1$enRF <- predict(emfit2_RF, as.data.frame(metaPred11))
```

*metaStack1* contains the predictions from the base models and meta models. Below, I provide plots similar to the base model prediction.

## 6.1 OLS Stacked Prediction

Figure 23: OLS Stacked Prediction Plot




## 6.2 k-Nearest Neighbour Stacked Prediction

Figure 24: KNN Stacked Prediction Plot

## 6.3 Random Forest (RF) Stacked Prediction

Figure 25: RF Stacked Prediction Plot


# 7 Predicting the price of test data

I use all the lessons learned in the above sections to predict the house price for the test data. Note: In the test data we only have house characteristics. Using those house characteristics I have to predict the price. I use the best tune parameters from the base models and from the meta-learnings models to predict the price.

*metaPred1* already contains the house predictions made by the base models XGBoost, RF, GBM, and kNN. Now I use the weights from the linear regression, kNN, and RF meta models to predict the house price for test data.

```
metaPred1[,5] <- cbind( matrix(1,nrow=dim(house_test)[1],ncol=1) ,
                        metaPred1[,-c(5,6)] ) %*% coef(emfit1)
metaPred1[,6] <- predict( emfit2_kNN, as.data.frame(metaPred1))
metaStack<- as.data.frame(metaPred1)
colnames(metaStack)[1:4]<- method_meta
colnames(metaStack)[5:6]<- c("enOLS","enKNN")
metaStack$enRF <- predict( emfit2_RF, as.data.frame(metaPred1))
```

```
metaStack %>% head()
```

*metaStack* contains the price predicted from the supervised learning models.

The following code snippet writes the predicted price in *.csv* file and saves them in the current directory.[2]

```
names(metaStack) <- c("priceXGB", "priceRF", "priceGBM",
                      "priceKNN", "priceEnOLS", "priceEnKNN", "priceEnRF")


testPredictedHousePrice <- cbind(metaStack, house_test)


write.csv(testPredictedHousePrice,file="testPredictedHousePrice.csv")
```

# 8 Conclusion

In this project, I used several supervised learning methods to analyze and predict house prices based on the characteristics of the house. Extreme Gradient Boosting Machine, Random Forest, Gradient Boosting Machine, and k Nearest Neighbours are some of the base models that perform better in terms of RMSE, R-squared, and linear relationship between the predicted and tested price. Using the best base models I implement a meta-learning algorithm (stacking) by using Ordinary Least Squares, k Nearest Neighbours, and Random Forest. Stacking improved the predictive power and gave me good results (evident from the plots I provided). Finally, I use the best tunes of the base models and meta-models to predict the house price using the house characteristics in the test data.

[2]One can always give a specific directory name to save the file. In order to do that first a local directory must be created and must be passed in *write.csv* function.