Supervised Machine Learning in R: Example 1

Carlos Calvo Hernandez

2/15/2019

Welcome to Machine Learning in R!

The purpose of this report is to two-fold to show an example of an **R Markdown** report, and show an example of Supervised Machine Learning algorithms. Two algorithms are used here: Decision Trees and Random Forests.

Setup and Data

We are gonna start by creating and running a model that predicts the price of real estate in Melbourne, Australia.

Let's load in the packages, and data we'll use.

```r
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------------------------------- tidyverse 1.2.
```

```
## v ggplot2 3.1.0     v purrr   0.3.0
## v tibble  2.0.1     v dplyr   0.7.8
## v tidyr   0.8.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.3.0
```

```
## -- Conflicts ------------------------------------------------------------------ tidyverse_conflicts(
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```r
library(rpart)
library(randomForest)
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
## The following object is masked from 'package:ggplot2':
##
##     margin
```

```r
melbourne_data <- read_csv("~/Documents/RStudio/Kaggle-DS_R/Data/melb_data.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Suburb = col_character(),
##   Address = col_character(),
##   Type = col_character(),
##   Method = col_character(),
##   SellerG = col_character(),
##   Date = col_character(),
##   CouncilArea = col_character(),
##   Regionname = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

Let's print a summary of the data to analyze it.

```r
summary(melbourne_data)
```

```
##     Suburb            Address              Rooms             Type          
##  Length:13580       Length:13580       Min.   : 1.000   Length:13580      
##  Class :character   Class :character   1st Qu.: 2.000   Class :character  
##  Mode  :character   Mode  :character   Median : 3.000   Mode  :character  
##                                        Mean   : 2.938                     
##                                        3rd Qu.: 3.000                     
##                                        Max.   :10.000                     
##                                                                           
##      Price            Method             SellerG         
##  Min.   :  85000   Length:13580       Length:13580      
##  1st Qu.: 650000   Class :character   Class :character  
##  Median : 903000   Mode  :character   Mode  :character  
##  Mean   :1075684                                        
##  3rd Qu.:1330000                                        
##  Max.   :9000000                                        
##                                                         
##      Date              Distance        Postcode       Bedroom2     
##  Length:13580       Min.   : 0.00   Min.   :3000   Min.   : 0.000  
##  Class :character   1st Qu.: 6.10   1st Qu.:3044   1st Qu.: 2.000  
##  Mode  :character   Median : 9.20   Median :3084   Median : 3.000  
##                     Mean   :10.14   Mean   :3105   Mean   : 2.915  
##                     3rd Qu.:13.00   3rd Qu.:3148   3rd Qu.: 3.000  
##                     Max.   :48.10   Max.   :3977   Max.   :20.000  
##                                                                    
##     Bathroom          Car           Landsize         BuildingArea  
##  Min.   :0.000   Min.   : 0.00   Min.   :     0.0   Min.   :    0  
##  1st Qu.:1.000   1st Qu.: 1.00   1st Qu.:   177.0   1st Qu.:   93  
##  Median :1.000   Median : 2.00   Median :   440.0   Median :  126  
##  Mean   :1.534   Mean   : 1.61   Mean   :   558.4   Mean   :  152  
##  3rd Qu.:2.000   3rd Qu.: 2.00   3rd Qu.:   651.0   3rd Qu.:  174  
##  Max.   :8.000   Max.   :10.00   Max.   :433014.0   Max.   :44515  
##                  NA's   :62                         NA's   :6450   
##    YearBuilt    CouncilArea          Lattitude        Longtitude   
##  Min.   :1196   Length:13580       Min.   :-38.18   Min.   :144.4  
##  1st Qu.:1940   Class :character   1st Qu.:-37.86   1st Qu.:144.9  
##  Median :1970   Mode  :character   Median :-37.80   Median :145.0  
##  Mean   :1965                      Mean   :-37.81   Mean   :145.0  
##  3rd Qu.:1999                      3rd Qu.:-37.76   3rd Qu.:145.1  
##  Max.   :2018                      Max.   :-37.41   Max.   :145.5  
##  NA's   :5375                                                      
##   Regionname        Propertycount  
##  Length:13580       Min.   :  249  
##  Class :character   1st Qu.: 4380  
##  Mode  :character   Median : 6555
##                     Mean   : 7454
##                     3rd Qu.:10331
##                     Max.   :21650
##
```

For this example we'll get rid of the observations that have missing values in the data.

```
melbourne_data <- na.omit(melbourne_data)
```

Running the model

Choosing the prediction target.

Since we want to predict the price of houses in Melbourne, then our "prediction target" should be the "Price" variable.

Choosing predictors

Next we need to choose the predictors. Let's start with the numeric variables of the data set. These are: the number of rooms and bathrooms, the size of the lot and the builiding, the year it eas built, and the location of the lot.

Building the model

We need to go through a set of steps to buil and use a model:

Define: Choose the type of model. Fit: Capture patterns from the data. This is the heart of the model. *Predict: The names says it all.* Evaluate: Determine the accuracy of the model's predictions.

We are going to create a decision tree. For that we'll use the `rpart()` function from the `rpart` package. This is where our earlier choices will come in. The `rpart()` uses a specific syntax that looks like this:

> prediction_target ~ predictor1 + predictor2 + predictor3

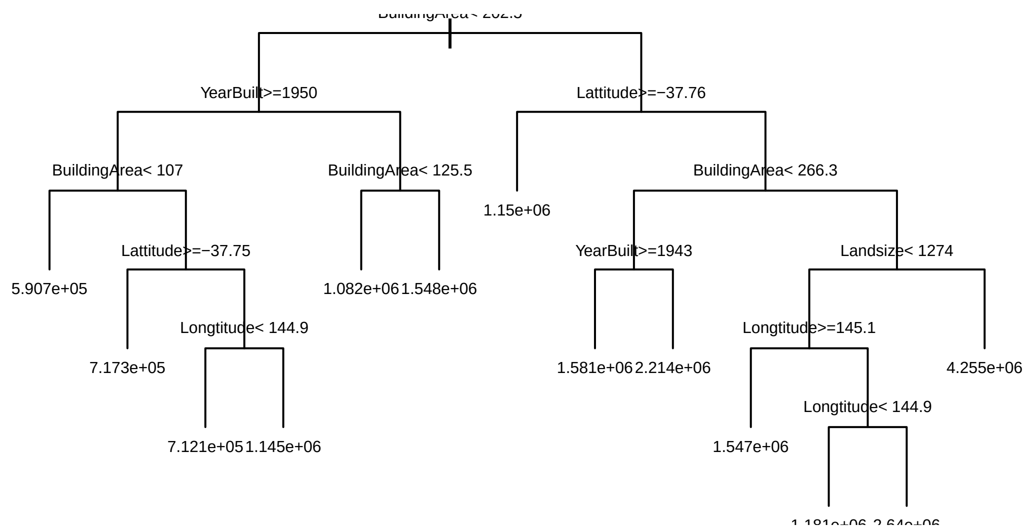
This syntax tells the function that we want to predict the "prediction_target" variable based on the values of "predictor1", "predictor2", and "predictor3".

Let's train the decision tree with the Melbourne data set.

```
fit <- rpart(Price ~ Rooms + Bathroom + Landsize + BuildingArea + YearBuilt + Lattitude + Longtitude, da
```

To actually see what the model is doing:

```
plot(fit, uniform = TRUE)
text(fit, cex = .5) # this line of code adds text labels and makes them 50% of their
```



```
                    # default size
```

Now, we can use our chosen fitted model to predict the prices of some house, we use the `predict()` function.

```
## [1] "Making predictions for the following 6 houses:"

## # A tibble: 6 x 21
##   Suburb Address Rooms Type   Price Method SellerG Date  Distance Postcode
##   <chr>  <chr>   <dbl> <chr>  <dbl> <chr>  <chr>   <chr>    <dbl>    <dbl>
## 1 Abbot~ 25 Blo~     2 h     1.03e6 S      Biggin  4/02~      2.5     3067
## 2 Abbot~ 5 Char~     3 h     1.46e6 SP     Biggin  4/03~      2.5     3067
## 3 Abbot~ 55a Pa~     4 h     1.60e6 VB     Nelson  4/06~      2.5     3067
## 4 Abbot~ 124 Ya~     3 h     1.88e6 S      Nelson  7/05~      2.5     3067
## 5 Abbot~ 98 Cha~     2 h     1.64e6 S      Nelson  8/10~      2.5     3067
## 6 Abbot~ 10 Val~     2 h     1.10e6 S      Biggin  8/10~      2.5     3067
## # ... with 11 more variables: Bedroom2 <dbl>, Bathroom <dbl>, Car <dbl>,
## #   Landsize <dbl>, BuildingArea <dbl>, YearBuilt <dbl>,
## #   CouncilArea <chr>, Lattitude <dbl>, Longtitude <dbl>,
## #   Regionname <chr>, Propertycount <dbl>

## [1] "The predictions are"

##       1       2       3       4       5       6
## 1082360 1547898 1145040 2214155 1082360 1082360

## [1] "Actual price"

## [1] 1035000 1465000 1600000 1876000 1636000 1097000
```

How do we know our model is good?

How good is the model we've just built? This question should be answered for almost every model you build. In most, though not in all, applications the rekevant measure of model quality is predictive accuracy. In other words, the idea is to know if the model's predictions are close to reality.

One way to do this is by making predictions with the same data used to train the model. Then we could compare those predictions to the actual target values in the training data. There's a critical shortcoming to this approach that we'll tackle in the following sections.

Even with this simple approach, we'll need to summarize the model quality into a form that humans can understand. If we end up with 10000 predicted home values, inevitably, there's gonna be a mix of good and bad predictions. Looking through such a long list would be time consuming and difficult.

There are many metrics for summarizing model quality, we'll start with **Mean Absolute Error** (MAE). This metric is a measure of difference between two continuos variables that observe the same phenomenon.

The prediction error for each house is:

> error = actual - predicted

This is straightforward, if a house costs $150,000 and the model predicted it would cost $100,000, then the error is $50,000.

With the MAE metric, we take the absolute value of each error. This converts each error to a positive number. We then take the average of those absolute errors. This is our measure of model quality. In other words:

> On average, our predictions are off by X

We can get the MAE for our model using the `mae()` function from the `modelr` package. The `mae()` function takes in a model and the dataset to test it againts.

```r
library(modelr)

mae(model = fit, data = melbourne_data)
```

```
## [1] 276287.9
```

The problem with "in-sample" scores

The MAE result we just computed is known as an "in-sample" score. In our model, we used a single set of houses (as our data set) for building the model **and** for calculating the MAE score. This is not ideal.

Imagine that, in the large real estate market, door color is unrelated to home price. However, in the sample of data you used to build the model, it may be that all homes with green doors were very expensive. The model's job is to find patterns that predict home prices, so it will see this pattern, and it will always predict high prices for homes with green doors.

Since this pattern was originally derived from the training data, the model will appear accurate in the training data. But this pattern likely won't hold when the model sees new data, and the model would be very inaccurate.

Models' practical value come from making predictions on new data, so we should measure performance on data that wasn't used to build the model. The most straightforward way to do this is to exclude some of our data from the model-building process, and then use those to test the model's accuracy on data it hasn't seen before. This data is called *validation data*.

We can split our dataframe into testing and training data very easily using the `resample_partition()` function from the `modelr` package.

```r
# split our data so that 30% is in the test set and 70% is in the training set
splitData <- resample_partition(melbourne_data, c(test = 0.3, train = 0.7))

# how many cases are in test & training set?
lapply(splitData, dim)
```

```
## $test
## [1] 1858   21
##
## $train
## [1] 4338   21
```

We can now fit a new model using the training data and test it using the testing data.

```
# fit a new model to our training set
fit2 <- rpart(Price ~ Rooms + Bathroom + Landsize + BuildingArea +
             YearBuilt + Lattitude + Longtitude, data = splitData$train)

# get the mean average error for our new model, based on our test data
mae(model = fit2, data = splitData$test)
```

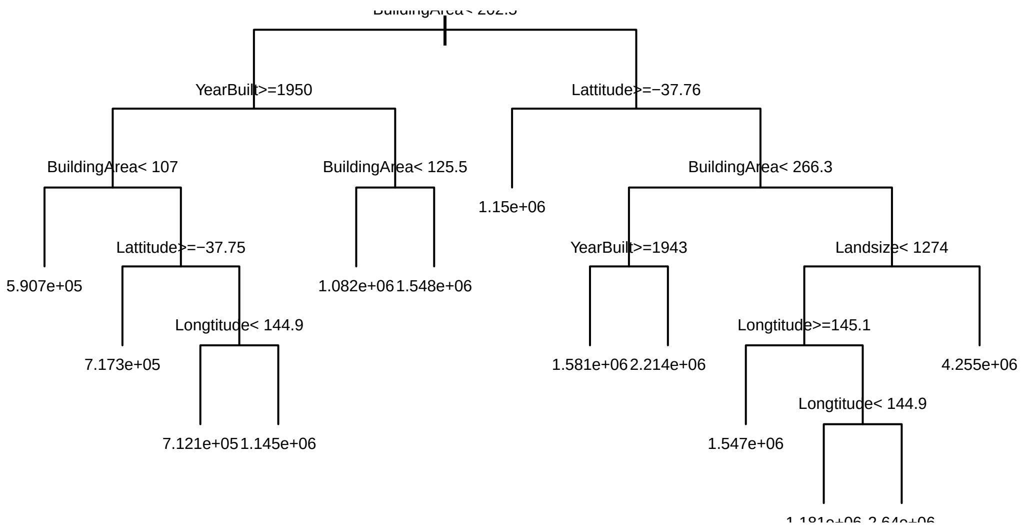
```
## [1] 283068.2
```

The error is now even larger than before, but we now know that it's not artificially lowered by testing on the training data.

Improving the model

There is a phenomenon called **overfitting**, in our "tree" model scenario, is when the model matches the data almost perfectly, but does poorly in validation and other new data. This would mean that there is a "leaf" of the tree for almost every house in the model.

At the other extreme end, a model that fails to capture important distinctions and patterns in the data so that it performs poorly even with training data is called **underfitting**. In our model, this would be like only having two or three "leaves" so that every leaf contains a wide variety of houses, that would make predictions to turn up far off for most houses.



We can use a utility function to help compare MAE scores from different values for the tree's max depth. A function to get the MAE for a given max depth.

```
# You should pass in the target as the name of the target column and the predictors
# as vector where each item in the vector is the name of the column

get_mae <- function(maxdepth, target, predictors, training_data, testing_data){

    # turn the predictors & target into a formula to pass to rpart()
    predictors <- paste(predictors, collapse="+")
    formula <- as.formula(paste(target,"~",predictors,sep = ""))

    # build our model
    model <- rpart(formula, data = training_data,
                   control = rpart.control(maxdepth = maxdepth))
    # get the mae
    mae <- mae(model, testing_data)
    return(mae)
}
```

Now, let's use a for-loop to compare the accuracy of models built with different values for maxdepth. In this case, the lowest MAE is 5.

```
# target & predictors to feed into our formula
target <- "Price"
predictors <-  c("Rooms","Bathroom","Landsize","BuildingArea",
                 "YearBuilt","Lattitude","Longtitude")

# get the MAE for maxdepths between 1 & 10
for(i in 1:10){
    mae <- get_mae(maxdepth = i, target = target, predictors = predictors,
                  training_data = splitData$train, testing_data = splitData$test)
    print(glue::glue("Maxdepth: ",i,"\t MAE: ",mae))
}
```

```
## Maxdepth: 1   MAE: 393678.333444824
## Maxdepth: 2   MAE: 355289.443450616
## Maxdepth: 3   MAE: 304689.049306818
## Maxdepth: 4   MAE: 289946.176913848
## Maxdepth: 5   MAE: 283068.165755048
## Maxdepth: 6   MAE: 283068.165755048
## Maxdepth: 7   MAE: 283068.165755048
## Maxdepth: 8   MAE: 283068.165755048
## Maxdepth: 9   MAE: 283068.165755048
## Maxdepth: 10  MAE: 283068.165755048
```

You can notice that after a certain depth the MAE levels out. This is because given this data set and our current stopping condition, 6 is the maximum number of nodes that `rpart` will use to create a tree. `rpart` has some built-in safeguards to prevent overfitting that won't generate a deeper tree for this data set.

The takeaway is that models can suffer from either:

- Overfitting: capturing spurious patterns that won't recur in the future, leading to less accurate predictions, or
- Underfitting: failing to capture relevant patterns, again leading to less accurate predictions.

We use validation data, which isn't used in model training, to measure a candidate model's accuracy. This lets us try many candidate models and keep the best one.

A new type of model: Random Forests

Decision trees leave you with a difficult decision. What type of tree do I use? A deep tree with lots of leaves that will overfit because each prediction is coming from historical data from only the few houses at its leaf, or a shallow tree with few leaves that will perform poorly because it fails to capture as many distinctions in the raw data.

Even today's most sophisticated modeling techniques face this tension between underfitting and overfitting. Nevertheless, many models have clever ideas that can lead to better performance. One example is the cleverly named **Random Forest**.

The random forest creates many trees, and it makes a prediction by averaging the predictions of each component tree. It generally has much better predictive accuracy than a single decision tree and it works well with default parameters. If you keep modeling, you can learn more models with even better performance, but many of those are sensitive to getting the right parameters.

One of the nice things about R is that the syntax you use to build models across different packages is pretty consistent. All we need to change in order to use a random forest instead of a plain decision tree is to load in the correct library & change the function we use from `rpart()` to `randomForest()`, like so:

```r
# fit a random forest model to our training set
fitRandomForest <- randomForest(Price ~ Rooms + Bathroom + Landsize + BuildingArea +
             YearBuilt + Lattitude + Longtitude, data = splitData$train, importance = TRUE)

# get the mean average error for our new model, based on our test data
mae(model = fitRandomForest, data = splitData$test)
```

```
## [1] 183839.8
```

As you can see, this is a big improvement over our previous best decision tree, which was around $320,000 off. There are parameters that allow you to change the performance of the Random Forest much as we changed the maximum depth of the single decision tree. But one of the best features of Random Forest models is that they generally work reasonably even without this tuning.

```r
fitRandomForest
```

```
##
## Call:
##  randomForest(formula = Price ~ Rooms + Bathroom + Landsize +      BuildingArea + YearBuilt + Lattitu
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##           Mean of squared residuals: 93379474509
##                     % Var explained: 79.19
```

We can tune the model by changing the number of trees (`ntree`), the number of variables randomly sampled (`mtry`). By default the `ntree` is set to 500 and `mtry` is 2. The percentage of the variance explained by the model is 77.25%.

```r
fitRandomForest2 <- randomForest(Price ~ Rooms + Bathroom + Landsize + BuildingArea +
             YearBuilt + Lattitude + Longtitude, data = splitData$train, ntree = 500, mtry = 6 , importa
```

```r
mae(model = fitRandomForest2, data = splitData$test)
```

```
## [1] 176923.9
```

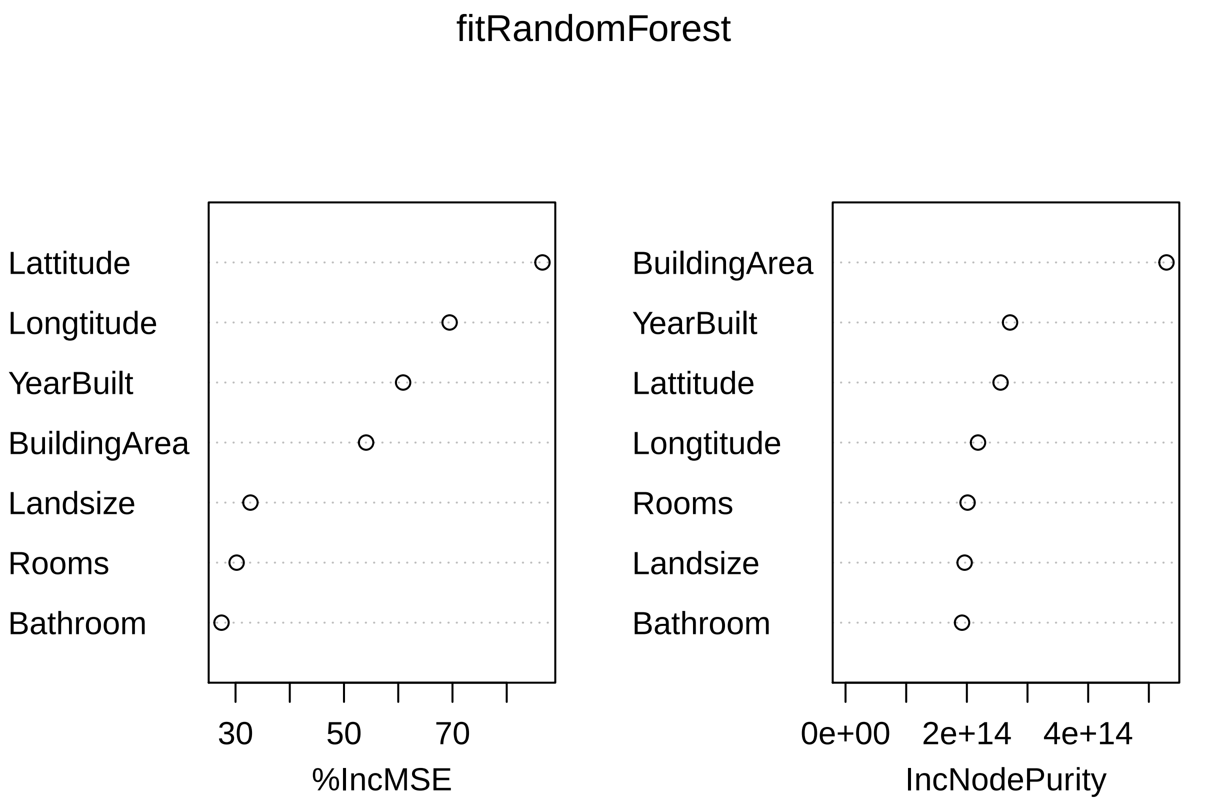
```r
fitRandomForest2
```

```
##
## Call:
##  randomForest(formula = Price ~ Rooms + Bathroom + Landsize +      BuildingArea + YearBuilt + Lattitu
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 6
##
##           Mean of squared residuals: 88027989389
##                     % Var explained: 80.38
```

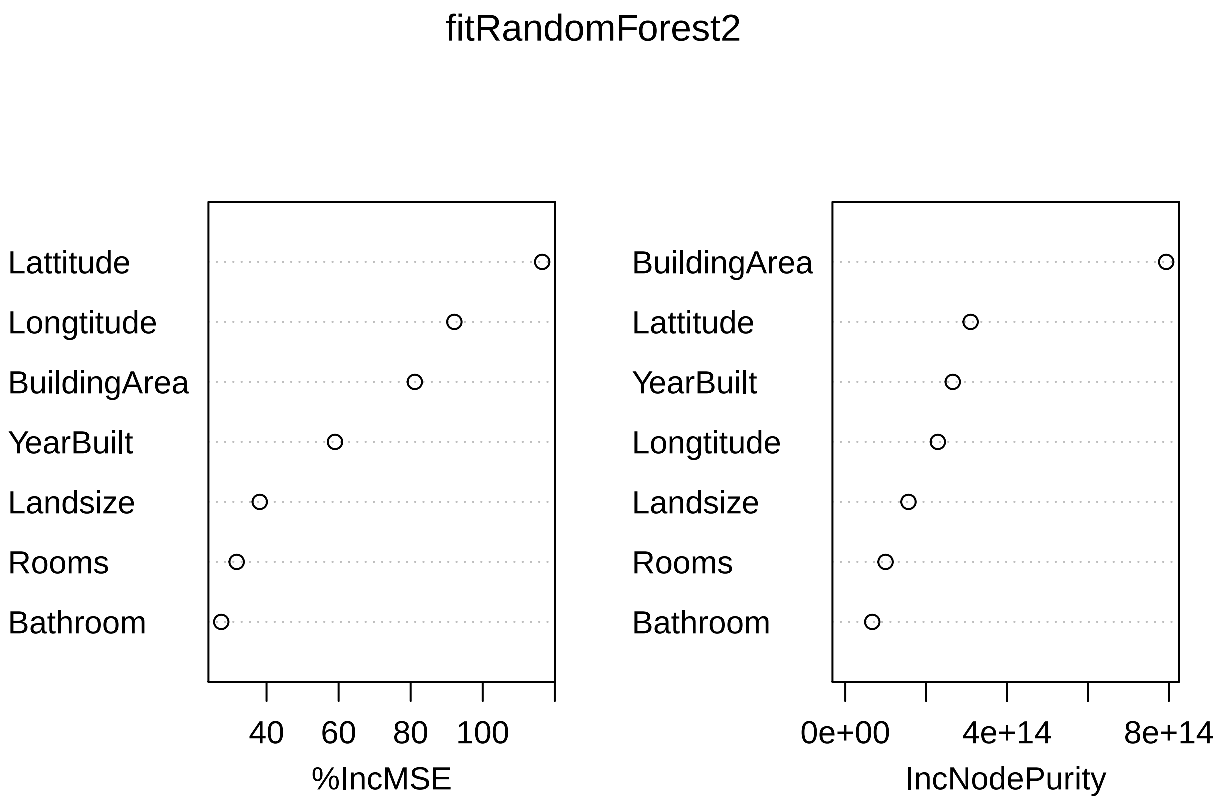
When we increase the `mtry` from 2 to 6, the variance explained increased from 77.25% to 78.52% and the MAE decreased by $ 6915.9794575.

Below, we can see the drop in mean accuracy for each of the variables in each of the models.

```r
varImpPlot(fitRandomForest)
```



```r
varImpPlot(fitRandomForest2)
```



We will now use a for-loop and check for different values of `mtry`.

```r
for (i in 2:7){
  fitRandomForest3 <- randomForest(Price ~ Rooms + Bathroom + Landsize + BuildingArea +
             YearBuilt + Lattitude + Longtitude, data = splitData$train, ntree = 500, mtry = i , importa
  rf_mae <- mae(model = fitRandomForest3, data = splitData$test)
  print(fitRandomForest3)
  print(glue::glue("mtry: ",i,"\t MAE: ", rf_mae))
}
```

```
##
## Call:
##  randomForest(formula = Price ~ Rooms + Bathroom + Landsize +      BuildingArea + YearBuilt + Lattitu
##               Type of random forest: regression
##                     Number of trees: 500
## No. of variables tried at each split: 2
##
##           Mean of squared residuals: 91360855310
##                     % Var explained: 79.64
## mtry: 2   MAE: 184100.245006768
##
## Call:
##  randomForest(formula = Price ~ Rooms + Bathroom + Landsize +      BuildingArea + YearBuilt + Lattitu
##               Type of random forest: regression
##                     Number of trees: 500
## No. of variables tried at each split: 3
##
##           Mean of squared residuals: 88150491214
##                     % Var explained: 80.36
## mtry: 3   MAE: 178920.174003426
##
## Call:
##  randomForest(formula = Price ~ Rooms + Bathroom + Landsize +      BuildingArea + YearBuilt + Lattitu
##               Type of random forest: regression
##                     Number of trees: 500
## No. of variables tried at each split: 4
##
##           Mean of squared residuals: 86900286421
##                     % Var explained: 80.64
## mtry: 4   MAE: 176575.869369749
##
## Call:
##  randomForest(formula = Price ~ Rooms + Bathroom + Landsize +      BuildingArea + YearBuilt + Lattitu
##               Type of random forest: regression
##                     Number of trees: 500
## No. of variables tried at each split: 5
##
##           Mean of squared residuals: 87104360494
##                     % Var explained: 80.59
## mtry: 5   MAE: 176502.429489577
##
## Call:
##  randomForest(formula = Price ~ Rooms + Bathroom + Landsize +      BuildingArea + YearBuilt + Lattitu
##               Type of random forest: regression
##                     Number of trees: 500
## No. of variables tried at each split: 6
##
##           Mean of squared residuals: 88337552217
##                     % Var explained: 80.32
## mtry: 6   MAE: 176342.665964434
##
## Call:
##  randomForest(formula = Price ~ Rooms + Bathroom + Landsize +      BuildingArea + YearBuilt + Lattitu
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 7
##
##           Mean of squared residuals: 91408466777
##                     % Var explained: 79.63
## mtry: 7   MAE: 177291.322021375
```

We can see how the variance explained by the model increases every iteration until the maximum accuracy level of `mtry` (8).

Conclusion

I hope this helps realize the power that RStudio posses when creating reports with embedded code, and how ML algorithms can be used and explained through these types of reports.

Sources

- Kaggle - Learn - R Module
- R-bloggers: How to implement random forests in R by Perceptive Analytics