

RANDOM FORESTS

Jan-Philipp Kolb

31 Mai, 2019

RANDOM FORESTS

- ▶ Thus, **bagging** can turn a single tree model with high variance and poor predictive power into a fairly accurate prediction function.
- ▶ But bagging suffers from **tree correlation**, which reduces the overall performance of the model.
- ▶ **Random forests** are a modification of bagging that builds a large collection of de-correlated trees
- ▶ It is a very popular **out-of-the-box**) learning algorithm that enjoys good predictive performance.

EXTENDING THE BAGGING TECHNIQUE

- ▶ Bagging introduces a random component in to the tree building process
- ▶ The trees in bagging are not completely independent of each other since all the original predictors are considered at every split of every tree.
- ▶ Trees from different bootstrap samples have similar structure to each other (especially at the top of the tree) due to underlying relationships.

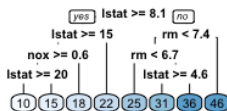
SIMILAR TREES - TREE CORRELATION

- ▶ If we create six decision trees with different bootstrapped samples of the Boston housing data, the top of the trees all have a very similar structure.
- ▶ Although there are 15 predictor variables to split on, all six trees have both `lstat` and `rm` variables driving the first few splits.

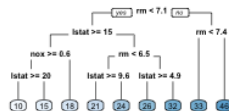
Decision Tree 1



Decision Tree 2



Decision Tree 3



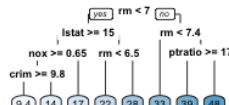
Decision Tree 4



Decision Tree 5



Decision Tree 6



TREE CORRELATION

- ▶ Tree correlation prevents bagging from optimally reducing variance of the predictive values.
- ▶ To reduce variance further, we need to minimize the amount of correlation between the trees.
- ▶ This can be achieved by injecting more randomness into the tree-growing process.

RANDOM FORESTS ACHIEVE THIS IN TWO WAYS:

1) Bootstrap:

- ▶ Similar to bagging, each tree is grown to a bootstrap resampled data set, which makes them different and decorrelates them.

2) Split-variable randomization:

- ▶ For every split, the search for the split variable is limited to a random subset of m of the p variables.
- ▶ For regression trees, typical default values are $m = p/3$ (tuning parameter).
- ▶ When $m = p$, the randomization is limited (only step 1) and is the same as bagging.

BASIC ALGORITHM

The basic algorithm for a regression random forest can be generalized:

1. Given training data set
2. Select number of trees to build (ntrees)
3. for i = 1 to ntrees do
4. | Generate a bootstrap sample of the original data
5. | Grow a regression tree to the bootstrapped data
6. | for each split do
7. | | Select m variables at random from all p variables
8. | | Pick the best variable/split-point among the m
9. | | Split the node into two child nodes
10. | end
11. | Use tree model stopping criteria to determine: tree complete
12. end

The algorithm randomly selects a bootstrap sample to train and predictors to use at each split.

CHARACTERISTICS

- ▶ Since bootstrap samples and features are selected randomly at each split, we create a more diverse set of trees, which tends to lessen tree correlation beyond bagged trees and often dramatically increase predictive power.

OUT-OF-BAG ERROR

- ▶ Similar to bagging, a natural benefit of the bootstrap resampling process is that random forests have an **out-of-bag** (OOB) sample that provides an efficient and reasonable approximation of the test error.
- ▶ This provides a built-in validation set without any extra work, and you do not need to sacrifice any of your training data to use for validation.
- ▶ We are more efficient identifying the number of trees required to stabilize the error rate

PREPARATION - RANDOM FORESTS

- ▶ The following slides are based on UC Business Analytics R Programming Guide on **random forests**

```
library(rsample)      # data splitting
library(randomForest) # basic implementation
library(ranger)       # a faster implementation of randomForest
# an aggregator package for performing many
# machine learning models
library(caret)
```

THE AMES HOUSING DATA

```
set.seed(123)
ames_data <- AmesHousing::ames_raw

set.seed(123)
ames_split <- rsample::initial_split(ames_data, prop=.7)
ames_train <- rsample::training(ames_split)
ames_test  <- rsample::testing(ames_split)
```

BASIC IMPLEMENTATION

- ▶ There are over 20 random forest packages in R.
- ▶ To demonstrate the basic implementation we illustrate the use of the `randomForest` package, the oldest and most well known implementation of the random forest algorithm in R.
- ▶ As your data set grows in size `randomForest` does not scale well (although you can parallelize with `foreach`).
- ▶ To explore and compare a variety of tuning parameters we can also find more effective packages.
- ▶ The package `ranger` will be presented in the tuning section.

RANDOMFOREST::RANDOMFOREST

- ▶ `randomForest` can use the formula or separate x, y matrix notation for specifying the model.
- ▶ Below we apply the default `randomForest` model using the formal specification.
- ▶ The default random forest performs 500 trees and $\frac{\text{features}}{3} = 26$ randomly selected predictor variables at each split.

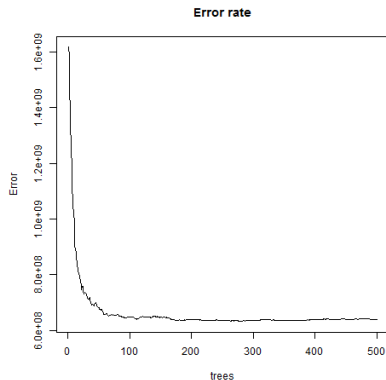
```
set.seed(123)
# default RF model
(m1 <- randomForest(formula = Sale_Price ~ ., data=ames_train))

##
## Call:
## randomForest(formula = Sale_Price ~ ., data = ames_train)
##               Type of random forest: regression
##               Number of trees: 500
## No. of variables tried at each split: 26
##
```

PLOTTING THE MODEL

- ▶ The error rate stabilizes with around 100 trees but continues to decrease slowly until around 300 trees.

```
plot(m1,main="Error rate")
```



RANDOM FORESTS - OUT-OF-THE-BOX ALGORITHM

- ▶ Random forests perform remarkably well with very little tuning.
- ▶ We get an RMSE of less than 30K Dollar without any tuning
- ▶ This is more than 6K Dollar RMSE-reduction compared to a fully-tuned bagging model
- ▶ and 4K dollar reduction to to a fully-tuned elastic net model.
- ▶ We can still seek improvement by tuning our random forest model.

TUNING RANDOM FORESTS

- ▶ Random forests are fairly easy to tune since there are only a handful of tuning parameters.
- ▶ The primary concern at the beginning is tuning the number of candidate variables to select from at each split.
- ▶ A few additional hyperparameters are important.

TUNING PARAMETERS (I)

- ▶ The argument names may differ across packages, but these hyperparameters should be present:

NUMBER OF TREES

- ▶ `ntree` - We want enough trees to stabilize the error but using too many trees is inefficient, esp. for large data sets.

NUMBER OF VARIABLES

- ▶ `mtry` - number of variables as candidates at each split. When `mtry=p` the model equates to bagging.
- ▶ When `mtry=1` the split variable is completely random, all variables get a chance but can lead to biased results. Suggestion: start with 5 values evenly spaced across the range from 2 to p .

TUNING PARAMETERS (II)

NUMBER OF SAMPLES

- ▶ `sampsiz` - Default value is 63.25% since this is the expected value of unique observations in the bootstrap sample.
- ▶ Lower sample sizes can reduce the training time but may introduce more bias. Increasing sample size can increase performance but at risk of overfitting - it introduces more variance.
- ▶ When tuning this parameter we stay near the 60-80% range.

TUNING PARAMETERS (III)

MINIMUM NUMBER OF SAMPLES WITHIN THE TERMINAL NODES:

- ▶ `nodesize` - Controls the complexity of the trees.
- ▶ Smaller node size allow for deeper, more complex trees
- ▶ This is another bias-variance tradeoff where deeper trees introduce more variance (risk of overfitting)
- ▶ Shallower trees introduce more bias (risk of not fully capturing unique patterns and relationships in the data).

MAXIMUM NUMBER OF TERMINAL NODES

- ▶ `maxnodes`: A way to control the complexity of the trees.
- ▶ More nodes equates to deeper, more complex trees.
- ▶ Less nodes result in shallower trees.

INITIAL TUNING WITH RANDOMFOREST

- ▶ If we just tune the `mtry` parameter we can use `randomForest::tuneRF` for a quick and easy tuning assessment.
- ▶ We start with 5 candidate variables (`mtryStart=5`) and increase by a factor of 2 until the OOB error stops improving by 1 per cent.
- ▶ `tuneRF` requires a separate `x y` specification.
- ▶ The optimal `mtry` value in this sequence is very close to the default `mtry` value of $\frac{\text{features}}{3} = 26$.

```
features <- setdiff(names(ames_train), "Sale_Price")  
  
set.seed(123)  
m2<-tuneRF(x= ames_train[,features],  
  y= ames_train$Sale_Price, ntreeTry = 500,  
  mtryStart = 5, stepFactor = 2,  
  improve = 0.01, trace=FALSE)
```

FULL GRID SEARCH WITH `RANGER`

- ▶ To perform a larger grid search across several hyperparameters we'll need to create a grid, loop through each hyperparameter combination and evaluate the model.
- ▶ Unfortunately, this is where `randomForest` becomes quite inefficient since it does not scale well.
- ▶ Instead, we can use `ranger` which is a C++ implementation of Breiman's random forest algorithm and is over 6 times faster than `randomForest`.

ASSESSING THE SPEED

RANDOMFOREST SPEED

```
system.time(  
  ames_randomForest <- randomForest(  
    formula = Sale_Price ~ .,  
    data     = ames_train,  
    ntree    = 500,  
    mtry     = floor(length(features) / 3)  
  )  
)
```

```
#      User      System    elapsed  
#      145.47      0.09      152.48
```

RANGER SPEED

```
system.time(  
  ames_ranger <- ranger(formula=Sale_Price ~ .,  
    data      = ames_train,num.trees = 500,  
    mtry      = floor(length(features) / 3))  
)  
  
##      user  system elapsed  
##    8.05    0.05     3.03
```

THE GRID SEARCH

- ▶ To perform the grid search, we construct our grid of hyperparameters.
- ▶ We search across 96 different models with varying `mtry`, minimum node size, and sample size.

```
# hyperparameter grid search
hyper_grid <- expand.grid(
  mtry          = seq(20, 30, by = 2),
  node_size     = seq(3, 9, by = 2),
  sampe_size    = c(.55, .632, .70, .80),
  OOB_RMSE      = 0
)

nrow(hyper_grid) # total number of combinations

## [1] 96
```

LOOP - HYPERPARAMETER COMBINATION (I)

```
for(i in 1:nrow(hyper_grid)) {  
  # train model  
  model <- ranger(  
    formula      = Sale_Price ~ .,  
    data         = ames_train,  
    num.trees    = 500,  
    mtry         = hyper_grid$mtry[i],  
    min.node.size = hyper_grid$node_size[i],  
    sample.fraction = hyper_grid$sample_size[i],  
    seed         = 123  
  )  
  # add OOB error to grid  
  hyper_grid$OOB_RMSE[i] <- sqrt(model$prediction.error)  
}
```

THE RESULTS

```
hyper_grid %>%  
  dplyr::arrange(OOB_RMSE) %>%  
  head(10)
```

##	mtry	node_size	sampe_size	OOB_RMSE
## 1	26	3	0.8	25404.60
## 2	28	3	0.8	25405.92
## 3	28	5	0.8	25459.46
## 4	26	5	0.8	25493.80
## 5	30	3	0.8	25528.26
## 6	22	3	0.7	25552.73
## 7	26	9	0.8	25554.31
## 8	28	7	0.8	25578.45
## 9	20	3	0.8	25581.23
## 10	24	3	0.8	25590.73

LOOP - HYPERPARAMETER COMBINATION (I)

- ▶ We apply 500 trees since our previous example illustrated that 500 was plenty to achieve a stable error rate.
- ▶ We set the random number generator seed. This allows us to consistently sample the same observations for each sample size and make the impact of each change clearer.
- ▶ Our OOB RMSE ranges between 25000 - 26000.
- ▶ Our top 10 performing models all have RMSE values right around 25500 and the results show that models with slightly larger sample sizes (70-80 per cent) and deeper trees (3-5 observations in an terminal node) perform best.
- ▶ We get a full range of `mtry` values showing up in our top 10 - not over influential.

HYPERPARAMETER GRID SEARCH - CATEGORICAL VARIABLES

- ▶ We use **one-hot encoding** for our categorical variables which produces 353 predictor variables versus the 80 we were using above.

```
# one-hot encode our categorical variables
```

```
(one_hot <- dummyVars(~ ., ames_train, fullRank = FALSE))
```

```
## Dummy Variable Object
```

```
##
```

```
## Formula: ~.
```

```
## 81 variables, 46 factors
```

```
## Variables and levels will be separated by '.'
```

```
## A less than full rank encoding is used
```

MAKE A DATAFRAME OF DUMMY VARIABLE OBJECT

```
ames_train_hot<-predict(one_hot,ames_train)%>%as.data.frame()
```

```
ames_train_hot[1:8,1:8]
```

```
## MS_SubClass.One_Story_1946_and_Newer_All_Styles
```

```
## 1 1
```

```
## 2 1
```

```
## 3 1
```

```
## 4 0
```

```
## 5 0
```

```
## 6 0
```

```
## 7 0
```

```
## 8 1
```

```
## MS_SubClass.One_Story_1945_and_Older
```

```
## 1 0
```

```
## 2 0
```

```
## 3 0
```

```
## 4 0
```

```
## 5 0
```

HOT ENCODING AND HYPERGRID

```
# make ranger compatible names
names(ames_train_hot) <- make.names(names(ames_train_hot),
                                     allow_ = FALSE)
ames_train_hot <- predict(one_hot, ames_train) %>%
  as.data.frame()
# --> same as above but with increased mtry values
hyper_grid_2 <- expand.grid(
  mtry      = seq(50, 200, by = 25),
  node_size = seq(3, 9, by = 2),
  sampe_size = c(.55, .632, .70, .80),
  OOB_RMSE  = 0
)
```

THE BEST MODEL

THE BEST RANDOM FOREST MODEL:

- ▶ retains columnar categorical variables
- ▶ `mtry = 24`,
- ▶ terminal node size of 5 observations
- ▶ sample size of 80%.

HOW TO PROCEED

- ▶ repeat the model to get a better expectation of error rate.
- ▶ as expected error ranges between ~25,800-26,400

RANDOM FORESTS WITH RANGER

```
OOB_RMSE <- vector(mode = "numeric", length = 100)

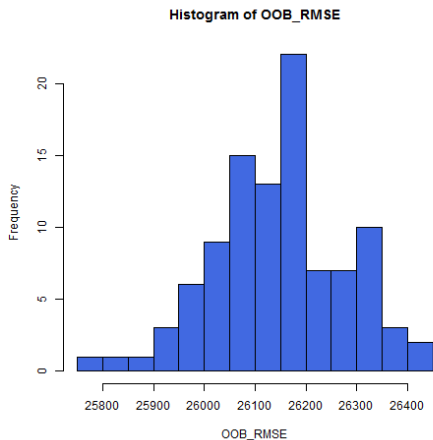
for(i in seq_along(OOB_RMSE)) {

  optimal_ranger <- ranger(
    formula      = Sale_Price ~ .,
    data         = ames_train,
    num.trees    = 500,
    mtry         = 24,
    min.node.size = 5,
    sample.fraction = .8,
    importance    = 'impurity'
  )

  OOB_RMSE[i] <- sqrt(optimal_ranger$prediction.error)
}
```

A HISTOGRAM OF OOB RMSE

```
hist(OOB_RMSE, breaks = 20,col="royalblue")
```

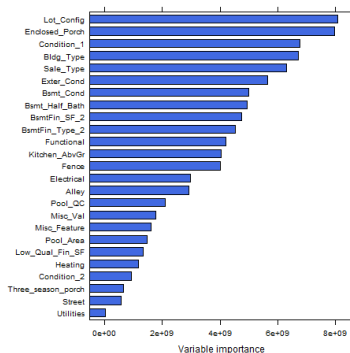


VARIABLE IMPORTANCE

- ▶ We set `importance = 'impurity'`, which allows us to assess variable importance.
- ▶ **Variable importance** is measured by recording the decrease in MSE each time a variable is used as a node split in a tree.
- ▶ The remaining error left in predictive accuracy after a node split is known as node impurity and a variable that reduces this impurity is considered more important than those variables that do not.
- ▶ We accumulate the reduction in MSE for each variable across all the trees and the variable with the greatest accumulated impact is considered the more important, or impactful.
- ▶ We see that `Overall_Qual` has the greatest impact in reducing MSE across our trees, followed by `Gr_Liv_Area`, `Garage_Cars`, etc.

PLOT THE VARIABLE IMPORTANCE

```
varimp_ranger <- optimal_ranger$variable.importance  
lattice::barchart(sort(varimp_ranger)[1:25], col="royalblue")
```



PREDICTING

- ▶ With the preferred model we can use the traditional predict function to make predictions on a new data set.
- ▶ We can use this for all our model types (randomForest and ranger); although the outputs differ slightly.

```
# randomForest
```

```
pred_randomForest <- predict(ames_randomForest, ames_test)
head(pred_randomForest)
```

```
##           1           2           3           4           5           6
## 113543.1 185556.4 259258.1 190943.9 179071.0 480952.3
```

```
# ranger
```

```
pred_ranger <- predict(ames_ranger, ames_test)
head(pred_ranger$predictions)
```

```
## [1] 129258.1 186520.7 265628.2 197745.5 175517.6 392691.7
```

SUMMARY - RANDOM FORESTS

- ▶ Random forests provide a very powerful out-of-the-box algorithm that often has great predictive accuracy.
- ▶ Because of their more simplistic tuning nature and the fact that they require very little, if any, feature pre-processing they are often one of the first go-to algorithms when facing a predictive modeling problem.

ADVANTAGES & DISADVANTAGES

ADVANTAGES - RANDOM FORESTS

- ▶ Typically have very good performance
- ▶ Remarkably good “out-of-the box” - very little tuning required
- ▶ Built-in validation set - don't need to sacrifice data for extra validation
- ▶ No pre-processing required
- ▶ Robust to outliers

DISADVANTAGES - RANDOM FORESTS

- ▶ Can become slow on large data sets
- ▶ Although accurate, often cannot compete with advanced boosting algorithms
- ▶ Less interpretable

LINKS

These slides are mainly based on

- ▶ A UC Business Analytics R Programming Guide - section **random forests**
- ▶ and on the **chapter on random forests** in the e-book of Brad Boehmke and Brandon Greenwell - Hands-on Machine Learning with R
- ▶ **Rpubs tutorial** - random forests
- ▶ Random Forests in R
- ▶ Boston Dataset-Tree Family Part-1