Random Forests

Jan-Philipp Kolb

28 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.

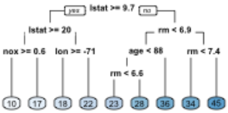
Extending the bagging technique

- 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.

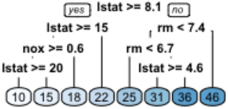
Similar trees - tree correlation

- 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.

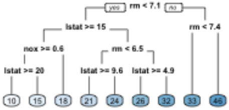
Decision Tree 1



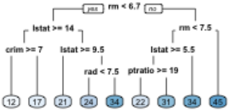
Decision Tree 2



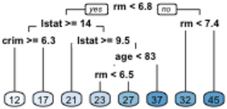
Decision Tree 3



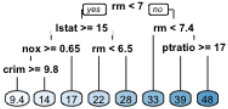
Decision Tree 4



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:

```
Given training data set
Select number of trees to build (ntrees)
for i = 1 to ntrees do
|  Generate a bootstrap sample of the original data
|  Grow a regression tree to the bootstrapped data
|  for each split do
|  | Select m variables at random from all p variables
|  | Pick the best variable/split-point among the m
|  | Split the node into two child nodes
|  end
| Use tree model stopping criteria to determine: tree comple
end
```

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

Characteristics

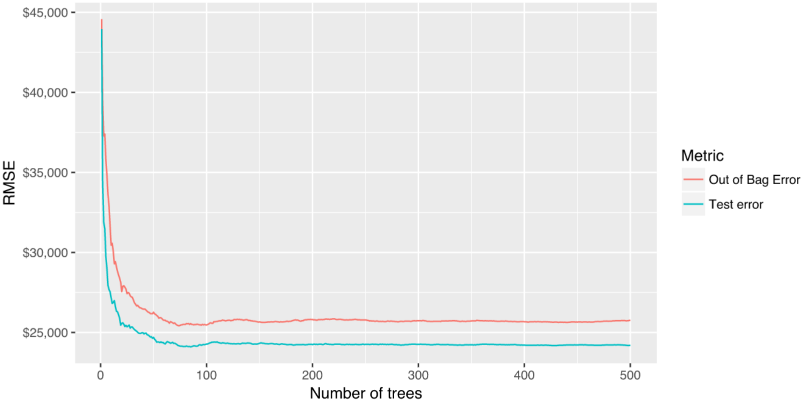
- Since bootstrap samples and sample of features are selected randomly at each split, a more diverse set of trees are produced which tends to lessen tree correlation beyond bagged trees and often dramatically increase predictive power.

OOB error vs. test set 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 stablize the error rate

Out-of-bag error vs. validation error

- Some difference between the OOB error and test error are expected.



Scoring models - metrics

- Many packages do not keep track of which observations were part of the OOB sample for a given tree and which were not.
- If you are comparing multiple models to one-another, you'd want to score each on the same validation set to compare performance.
- It is possible to compute certain metrics such as root mean squared logarithmic error (RMSLE) on the OOB sample, but it is not built in to all packages.
- So if you are looking to compare multiple models or use a slightly less traditional loss function you will likely want to still perform cross validation.

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

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 packages `ranger` and `h2o` packages will be presented in the tuning section.

randomForest::randomForest

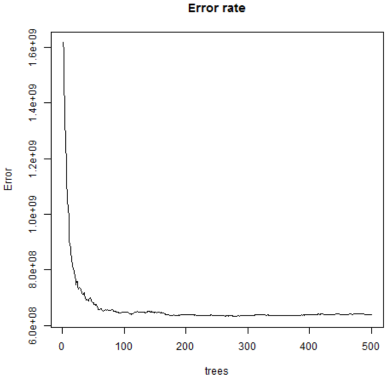
- randomForest can use the formula or separate x, y matrix notation for specifying our model.
- Below we apply the default randomForest model using the formulaic specification.
- The default random forest performs 500 trees and $\frac{\text{features}}{3} = 26$ randomly selected predictor variables at each split.
- Averaging across all 500 trees provides an OOB MSE=659550782 (RMSE=25682).

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

Plotting the model

- Plotting the model will illustrate the error rate as we average across more trees and shows that our error rate stabalizes with around 100 trees but continues to decrease slowly until around 300 or so trees.

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



The plotted error rate

- The plotted error rate above is based on the OOB sample error and can be accessed directly at `m1$mse`.
- We can find which number of trees providing the lowest error rate, which is 344 trees providing an average home sales price error of $25,673.

```
which.min(m1$mse)
sqrt(m1$mse[which.min(m1$mse)])
```

A validation set to measure predictive accuracy

- randomForest also allows us to use a validation set to measure predictive accuracy if we did not want to use the OOB samples.
- Here we split our training set further to create a training and validation set.
- The validation data is in xtest and ytest.

```
set.seed(123)
valid_split <- initial_split(ames_train, .8)
# training data
ames_train_v2 <- analysis(valid_split)
# validation data
ames_valid <- assessment(valid_split)
x_test <- ames_valid[setdiff(names(ames_valid), "Sale_Price")]
y_test <- ames_valid$Sale_Price
```

Run the model

- With the command randomForest

```
rf_oob_comp <- randomForest(formula=Sale_Price ~ .,
  data=ames_train_v2,xtest = x_test,ytest=y_test)
```

COMPARE ERROR RATES

- Extract OOB & validation errors

```
tibble::tibble(
  `Out of Bag Error` = oob,
  `Test error` = validation,
  ntrees = 1:rf_oob_comp$ntree
) %>%
  gather(Metric, RMSE, -ntrees) %>%
  ggplot(aes(ntrees, RMSE, color = Metric)) +
  geom_line() +
  scale_y_continuous(labels = scales::dollar) +
  xlab("Number of trees")
```

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

Tuning parameters (I)

number of trees

- `ntree` - We want enough trees to stabalize 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`=pthe 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

- `sampsize` - 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 allows 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 patters and relatonships 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 `mtry = 5` and increases by a factor of 1.5 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 = 1.5,
  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 and 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

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

RANGER SPEED

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

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.

```r
# 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
```

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$sampe_size[i],
    seed            = 123
  )
    # add OOB error to grid
  hyper_grid$OOB_RMSE[i] <- sqrt(model$prediction.error)
}
```

The results

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

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 26,000-27,000.
- Our top 10 performing models all have RMSE values right around 26,000 and the results show that models with slighly 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

```r
# one-hot encode our categorical variables
one_hot <- dummyVars(~ ., ames_train, fullRank = FALSE)
ames_train_hot<-predict(one_hot,ames_train)%>%as.data.frame()
# 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

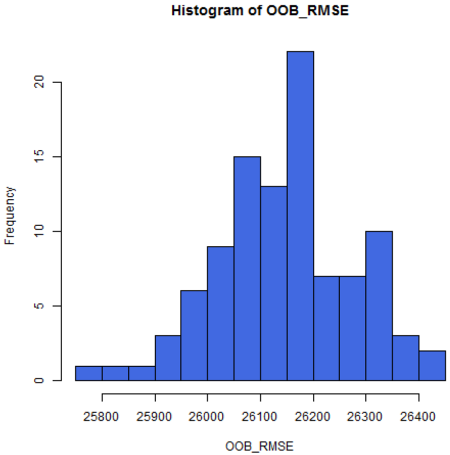
- repeat the model to get a better expectation of error rate.
- as expected error ranges between ~25,800-26,400

Random forests with ranger

```r
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

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

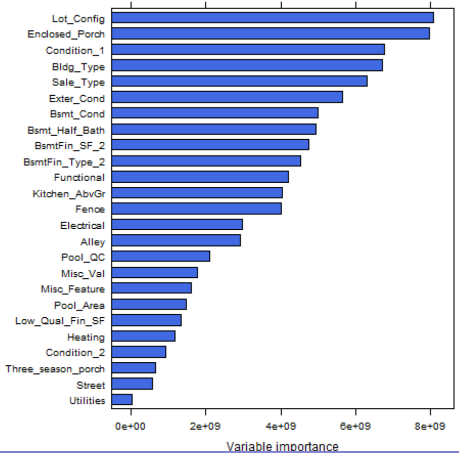


Variable importance

- 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 imporant 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, ranger, and h2o); although the outputs differ slightly.

```
# randomForest
pred_randomForest <- predict(ames_randomForest, ames_test)
head(pred_randomForest)
# ranger
pred_ranger <- predict(ames_ranger, ames_test)
head(pred_ranger$predictions)
# h2o
pred_h2o <- predict(best_model, ames_test.h2o)
head(pred_h2o)
```

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 forrests

- 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 forrests

- 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