Hands-on-ML with R

Koji Mizumura

October 29th,2018 - November 29th,2018

Table of Contents

# Random forest model

**Random forests** are modification of decision trees and bagging that builds a large collection of *de-correlated* trees to reduce overfitting (aka variance). They have become a very popular “out -of-the-box” learning algorithm that enjoys good predicive performance and easy hyperparameter tuning. Many modern implementations of random forests algorithms exist: however, Leo Breiman’s algorithm [@breiman2001random] has largely become the authoritative procedure. This chapter will cover the fundamentals of random forests.

## Prerequisites {rf-requirements}

**Notes** Any tutorial on random forests (RF) should also include a review of decision trees, as these are models that are ensembled together to create the random forest model 窶? or put another way, the 窶徼rees that comprise the forest.窶? Much of the complexity and detail of the random forest algorithm occurs within the individual decision trees and therefore it窶冱 important to understand decision trees to understand the RF algorithm as a whole. Therefore, before proceeding, it is recommended that you read through <http://uc-r.github.io/regression_trees> prior to continuing.

library(rsample)  
library(ranger)  
library(h2o)  
library(vip)  
library(ggplot2)  
library(dplyr)  
library(pdp)

## Advantanges & Disadvantages

**Advantages:**

* 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.
* Does not overfit.
* No data pre-processing required - often works great with categorical and numerical values as is.
* Robust to outliers.
* Handles missing data - imputation not required.
* Provide automatic feature selection.

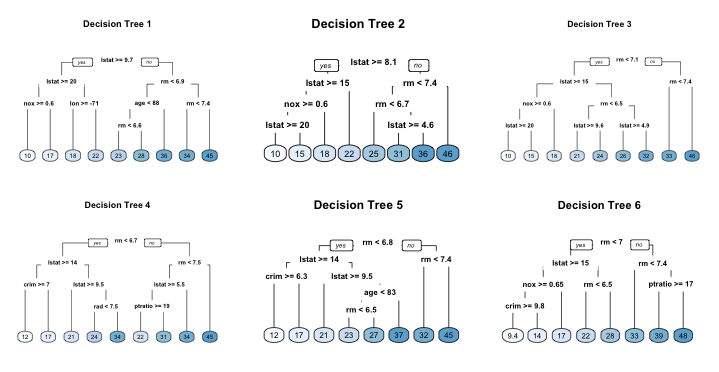
**Disadvantages:**

* Can become slow on large data sets.
* Although accurate, often cannot compete with the accuracy of advanced boosting algorithms.
* Less interpretable although this is easily addressed with various tools (variable importance, partial dependence plots, LIME, etc.).

## The Idea

Random forests are built on the same fundamendtal principles as decision trees and bagging (check out this [tutorial](http://uc-r.github.io/regression_trees) if you need a refresher on these techniques). Bagging trees introduces a random component in to the tree building process that reduces the variance of a single tree’s prediction and improves predictive performance. However, the trees in bagging are not completely independent of each other since all the original predictors are considered at every split of every tree. Rather, trees from different bootstrap samples typically have similar structure to each other (especially at the top of the tree) due to underlying relationships.

For example, if we create six decision trees with different bootstrapped samples of the [Boston housing data]((http://lib.stat.cmu.edu/datasets/boston)) [@harrison1978hedonic], we see that 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.



Six decision trees based on different bootstrap samples

This characteristics is known as \*\*tree correlation\* and prevents bagging from optimally reducing variance of the precictive values. In order to reduce variance further, we need to minimize the amount of correlation between the trees. This can be achieved by injecting more randomess into 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 *somewhat* decorrelates them.
2. **Split-variable randomization**: each time a split is to be performed, the search for the split variable is limited to a random subset of *m* of the *p* variables. Typical default values are (is limited to a random subset of *m* (regression trees) and (classification trees) but this should be considered a tuning parameter. When , the randomization amounts to using only step 1 and is the same as *bagging*.

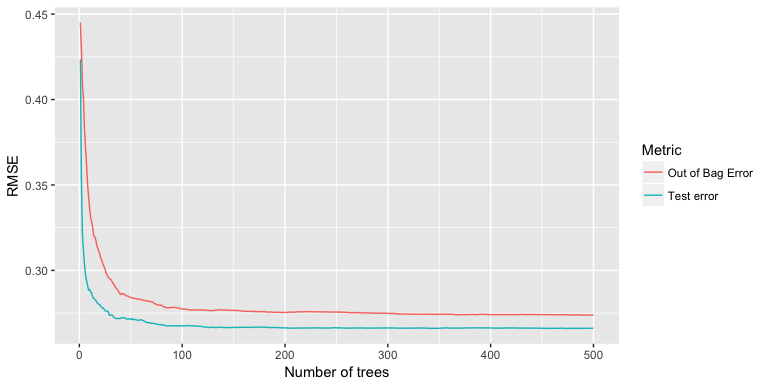
The basic algorithm for a regression or classification random forest can be generalized to the following:

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 or classification 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 typical tree model stopping criteria to determine when a tree is complete (but do not prune)  
12. end

Since the algorithm randomly selects a bootstrap sample to train on **and** predictors to use at each split, tree correlation will be lessened beyond bagging trees.

### OOB error vs test set error

Similar to bagging, a natural benefit of the bootstrap resampling process is that random frests have na out-of-bag (OOB) that provides an efficient and reasonable approximation of the test error. This provides a built-in validation set without any extra work on your part, and you do not need to sacrifice any of your training data to use for validation. This makes identifying the number of trees required to stablize the error rate during tuning more efficient; however, as illustrated below some difference between the OOB error and test error are expected.



Random forest out-of-bag error versus validation error.

Furthermore, 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. Also, although technically it is possible to compute certain metrics such as root mean squared logarithmic error (RMSLE) on the OOB sample, 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.

### Tuning

## State Space Model

DLM KFAS: non-gaussian distribution dealt

observation model

state model

library(KFAS)  
  
mod <- SSModel(Weight~SSMtrend(1,Q=NA),H=NA)  
fit <- fitSSM(mod, numeric(2),method="BFGS")  
# set method for improvming calculation speed - not required  
kfs <- KFS(fit$model)  
kfs  
  
# confidential interval  
alphahatconf <- predict(fit$model, interval="confidence", level=0.95)  
  
  
# impute missing NA  
WeightNA <- Weight[c(1:20), rep(NA,20),41:6]  
  
# visualization - forecast() package  
library(forecast)  
forecast::autoplot(Weight)+  
 autolayer(alphahatconf)

## Multivariable local level model

variance-covariance matrix is assumed for the error term distribution. shows the covariance between two errors. Thus, we need to set up a matrix. However, the covariance is identical between and .

library(tidyverse)  
modSUTSE <- SSModel(cbind(Weight, Bodyfat)~  
 SSMtrend(1,Q=matrix(NA,2,2)), H=matrix(NA,2,2))  
  
fitSUTSE <- fitSSM(modSUTSE, numeric(6), method="BFGS")  
kfsSUTSE <- KFS(fitSUTSE$model)  
  
alphahatconf <- predict(fitSUTSE$model, interval="confidence", level=0.95)  
alphahatconf  
  
library(forecast)  
forecast::autoplot(Weight)+  
 autolay  
 autolayer(alphahatconf, facets=TRUE)

local level model Cons: - prediction is horizontal based, not accurate for upward/downward trend data

Linear trend model Cons: - dependent on analysis period. and not good for prediction