Random Forests

Joshua F. Wiley

2019-05-14

Contents

1 *SETUs*

Please complete the SETUs for PSY6103 / PSY6161 here: `https://monash.bluera.com/monash`. Your feedback and evaluation is critical to understanding what worked and where we need to improve in this unit. As these classes are small, every evaluation is really critical. Please help us by completing it now.

2 *Random Forests*

Random forests (RFs) are a type of machine learning (ML) model that are very good at building accurate, prediction models under many circumstances.

The foundation of RFs are classification and regression trees (CART). The figure that follows shows an example of a CART for happiness. At the top is the first decision rule, if true, you move to the left, if false, you move to the right.

CARTs have many strengths.

- flexible, allowing non-linear trends to be captured
- relatively easy to follow and interpret
- utilize continuous or categorical predictors easily
- incorporate interactions between variables naturally (lower branches on the tree are interactions with higher branches)

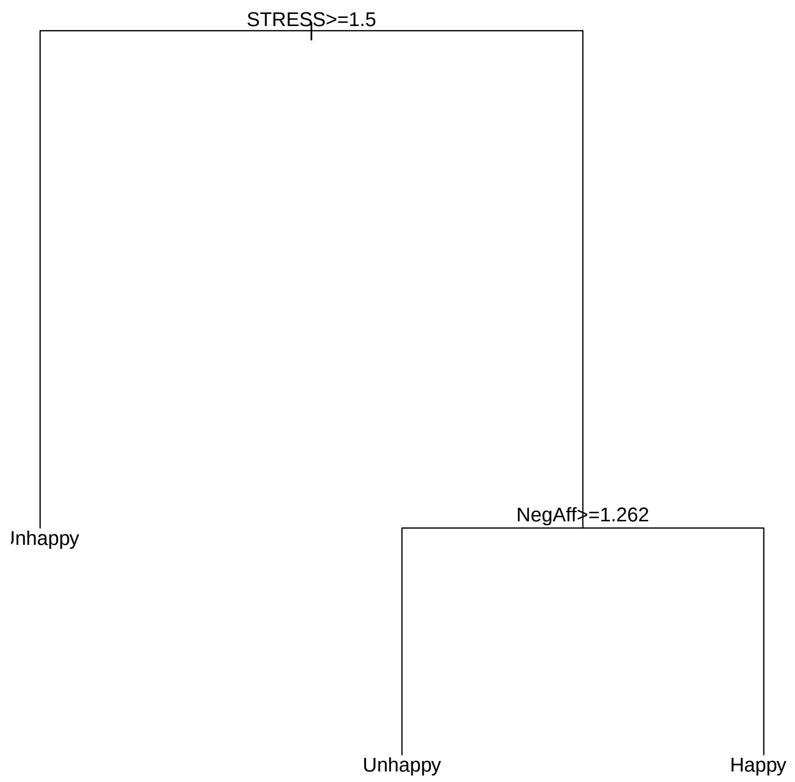


Figure 1: CART decision tree example for happy.

However, CARTs also have drawbacks. Specifically, as you move down a tree, the sample gets smaller and smaller and often it is easy to overfit data. Often, CARTs result in a large, complex tree and research found that these trees typically did not replicate well when taken to new samples, suggesting that they were overfitting the data[1].

[1] Generally in statistics or in machine learning, our goal is to build models that predict or model the "true" associations or patterns. Due to a variety of reasons, we often assume there is some "noise" in our data, due to measurement error, quirks of specific people/observations, etc. We do not want to model this noise. The idea of **overfitting** is that sometimes our models, especially complex ones in smaller samples, may fit or "learn" about the noise in the data rather than the true signal. When this occurs, a model may fit well in the data it is built on, but when taken to a new sample, fit poorly.

One solution used in the past was to "prune" CARTs. First, a large CART would be grown, and then it would be systematically pruned to remove branches that did not provide sufficient improvement in classification or variance explained. However, this approach still was not well able to achieve both high predictive accuracy and stable models.

Random forests (RFs) are built on CARTs but differ in several key ways. The basic conceptual idea is that much of the reason CARTs are unstable is that they can grow to be too larger and large, complex models more easily overfit data. RFs rely on small CARTs, by controlling the number of layers each tree can grow to. As their name suggets, RFs involve a "forest" and forests have many trees. Instead of trying to build one, large tree that captures everything, RFs rely on many small trees to build accurate prediction models.

To make overall predictions, RFs aggregate the results from each tree. Essentially, every little tree "votes" on the outcome and these votes are counted up to decide what the overall RFs prediction is.

A few final conceptual notes about machine learning generally. Often, building one model is relatively easy. Much of the work in machine learning and RFs is around creating the best model and validating it. RFs do not have p-values directly or measures for statistical significance. Typically, one does not evaluate individual predictors but the overall model performance. Common steps include creating completely independent training and testing datasets (e.g., by taking one study dataset and splitting 80% of samples for training, 20% for testing). Bootstrapping and cross-validation also often are used. Cross-validation involves splitting a dataset into k folds. A model is trained on $k-1$ of the folds and then tested on the unused kth fold. The process is repeated k times, so that every fold is used for testing once. You can use cross-validation even if splitting data into training and testing as within the training data, you may wish to **tune** the model, which is when the optimal settings are chosen based on model performance (e.g., how many trees should be in the RF, and other features that control how the RF performs).

3 Random Forests in R

3.1 Setup (Activity)

Open RStudio and open a project for this class as we did in earlier weeks when using R. If you want, you also can create a new project (File -> New Project...).

Download the raw R markdown code here https://jwiley.github.io/MonashDoctoralStatistics/RandomForests.rmd.

Download the data for today from here https://jwiley.github.io/MonashDoctoralStatistics/aces_daily_sim_processed.RDS.

Make sure to place both the code file and data in the project directory. Then, open the code in RStudio.

We use several R packages:

- data.table for general data management
- ggplot2 for graphing and visualization
- randomForest for fitting random forest machine learning models
- caret which facilitates preparing data and model building
- DALEX package to help explain "black box" machine learning models
- spdep package which is required for the DALEX package

The code below loads the packages. First, try loading each package. For any package that is not installed, you can install it by typing: install.packages("randomForest", type = "binary") or whatever the package name is. If possible, do not install from source, just use existing binary files.

```r
options(digits = 2)
options(install.packages.check.source = "no")

library(data.table)
library(ggplot2)
library(rpart)
library(randomForest)
library(caret)
library(spdep)
library(DALEX)

## read in the dataset
d <- readRDS("aces_daily_sim_processed.RDS")
## make a copy, remove missing and create
## 'Happy' as a binary variable
d2 <- copy(d)[, .(EDU, SES_1, Age, BornAUS, Female,
    STRESS, NegAff, PosAff, Survey, Weekend = as.integer(weekdays(SurveyDay) %in%
        c("Saturday", "Sunday")))]
d2 <- na.omit(d2)
d2[, ‘:=‘(Happy, factor(as.integer(PosAff > 3),
    levels = 0:1, labels = c("Unhappy", "Happy")))]

## split data into 80% for training and use the
## remaining and ’testing’ data (20%)
set.seed(12345)
trainIndex <- createDataPartition(d2$Happy, p = 0.8,
    list = FALSE, times = 1)

## subset rows that should be in the training
## data
dtrain <- d2[trainIndex]
## subset to EXCLUDE rows in the training data,
## using ’-’
dtest <- d2[-trainIndex]
```

We can have `R` train a RF relatively easily. The `caret` package provides a general interface to hundreds of machine learning methods. Before we actually train the model, however, we specify some options. The `trainControl()` function from the `caret` package is used to control how machine learning models including RFs are tuned. Tuning is a process where different options of the models are optimized based on which choice yields the best performance. The following code shows two examples. In class, we will use `simpleTuning` as it is much faster to run. However, for actual modeling, the `betterTuning` option is a better, albeit slower, choice.

```
## 2-fold cross validation normally at least 5,
## using 2 for fast demonstration
simpleTuning <- trainControl(method = "cv", number = 2,
    classProbs = TRUE, summaryFunction = twoClassSummary)

## 10-fold cross validation, repeated 10 times
betterTuning <- trainControl(method = "repeatedcv",
    number = 10, repeats = 10, classProbs = TRUE,
    summaryFunction = twoClassSummary)
```

The `train()` function from the `caret` package is a generic function that allows training many machine learning models. It requires a dataset with features for prediction, `x` and an outcome variable, `y`. We also must tell it what method to use, we use “rf” for random forests. Next is how should the best tuning parameters be chosen. We use “ROC” but others are possible. We specify the number of trees in the forest. Here we use 100. Often, 500, 1,000, or sometimes many thousands are used. We use 100 just for speed. The `nodesize` controls the minimum number of observations allowed in a terminal node. Higher values mean that a tree cannot be grown that has nodes with too few people in them.

The tuning control is specified using `trControl` and we set it equal to the `simpleTuning` we defined above. The argument, `tuneGrid` controls what tuning parameters are varied and optimized. We just do `mtry` here which is how many variables should `R` randomly try for each tree in the random forest (we are just trying 2 and 5, any number up to the number of predictors is possible). Finally `importance = TRUE` is an option to have it store information that will allow us to calculate how important each predictor variable was to the overall model performance.

```
set.seed(12345)
rfModel <- train(x = dtrain[, .(Age, BornAUS,
    Survey, Weekend, EDU, STRESS)], y = dtrain$Happy,
    method = "rf", metric = "ROC", ntree = 100,
    nodesize = 10, trControl = simpleTuning, tuneGrid = data.frame(mtry = c(2,
        5)), importance = TRUE)

rfModel

## Random Forest
##
## 4959 samples
##    6 predictor
##    2 classes: 'Unhappy', 'Happy'
##
## No pre-processing
## Resampling: Cross-Validated (2 fold)
## Summary of sample sizes: 2479, 2480
## Resampling results across tuning parameters:
##
##   mtry  ROC   Sens  Spec
##   2     0.73  0.77  0.57
##   5     0.71  0.75  0.57
##
## ROC was used to select the optimal
##  model using the largest value.
## The final value used for the model was
##  mtry = 2.

rfModel$finalModel
```

```
##
## Call:
##  randomForest(x = x, y = y, ntree = 100, mtry = param$mtry, nodesize = 10,      importance = TRUE)
##               Type of random forest: classification
##                     Number of trees: 100
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 30%
## Confusion matrix:
##         Unhappy Happy class.error
## Unhappy    2337   664        0.22
## Happy       846  1112        0.43
```

We can get many performance measures using the `confusionMatrix()` function. Assuming the table below, we have:

	Event	No Event
Event	A	B
No Event	C	D

Accuracy is the proportion of events correctly classified.

$$Accuracy = \frac{A+D}{A+B+C+D}$$

Sensitivity is the proportion of events are actually predicted to be an event.

$$Sensitivity = \frac{A}{A+C}$$

Specificity is the proportion of **non** events actually predicted to be **non** events.

$$Specificity = \frac{D}{B+D}$$

Prevalence is the actual proportion of people with an event (compared to everyone).

$$Prevalence = \frac{A+C}{A+B+C+D}$$

Positive Predictive Value (PPV) is a measure of the value of the prediction for indicating that someone is a positive case.

$$PPV = \frac{sensitivity \cdot prevalence}{(sensitivity \cdot prevalence) + ((1 - specificity) \cdot (1 - prevalence))}$$

Negative Predictive Value (NPV) is a measure of the value of the prediction for indicating that someone is a negative case.

$$NPV = \frac{specificity \cdot (1 - prevalence)}{((1 - sensitivity) \cdot prevalence) + ((specificity) \cdot (1 - prevalence))}$$

Detection rate is the proportion of all people correctly predicted as having an event.

$$DetectionRate = \frac{A}{A + B + C + D}$$

Detection prevalence is the proportion of all people predicted to have an event (correctly and incorrectly).

$$DetectionPrev = \frac{A + B}{A + B + C + D}$$

Balanced accuracy is an attempt to get a balanced measure that equally weights sensitivity and specificity.

$$BalancedAccuracy = \frac{sensitivty + specificity}{2}$$

```
confusionMatrix(data = predict(rfModel, newdata = dtest),
    reference = dtest$Happy, positive = "Happy")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Unhappy Happy
##    Unhappy     564   218
##    Happy       186   271
##
##                Accuracy : 0.674
##                  95% CI : (0.647, 0.7)
##     No Information Rate : 0.605
##     P-Value [Acc > NIR] : 3.38e-07
##
##                   Kappa : 0.31
##  Mcnemar's Test P-Value : 0.123
##
##             Sensitivity : 0.554
##             Specificity : 0.752
##          Pos Pred Value : 0.593
##          Neg Pred Value : 0.721
##              Prevalence : 0.395
##          Detection Rate : 0.219
##    Detection Prevalence : 0.369
##       Balanced Accuracy : 0.653
##
```

```
##        'Positive' Class : Happy
##
```

3.2 *Break*

3.3 *Reading (Weihs et al)*

Available from: `https://doi.org/10.1002/pon.4472`.

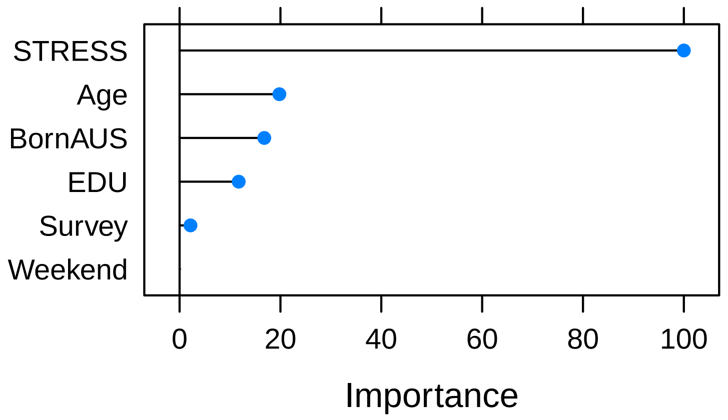
3.4 *Explaining Models*

Model performance is very important for RFs as the goal commonly is just to build the most accurate possible model. However, increasingly there is a focus on trying to explain or understand these "black box" models. The `DALEX` package has many functions to help with this. To begin with, we can calculate the relative importance of each variable. These importances are scaled by default so that the most important variable is 100% and everything else is less than that. Variable importances can help identify what are key features for your model's performance and also potential variables/features that could be dropped with minimal loss. The figure below suggests that survey (whether a survey was completed in morning, afternoon, or evening) and weekend contribute quite little.

```
varImp(rfModel)
```

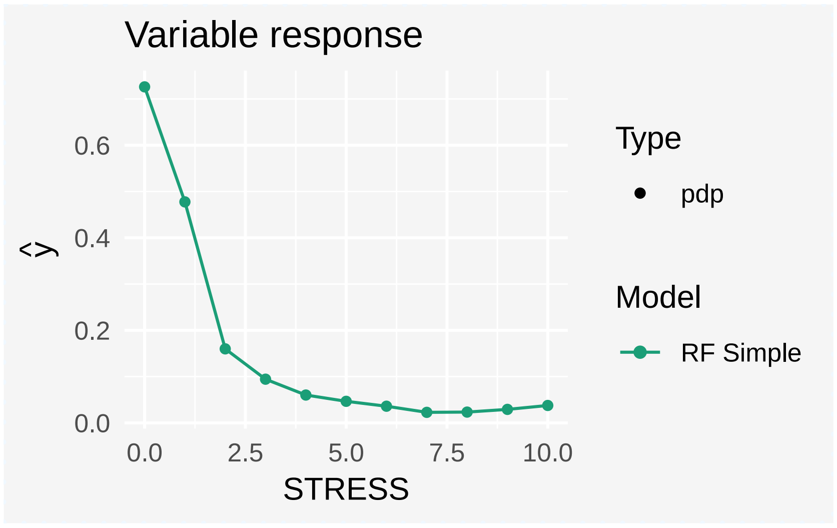
```
## rf variable importance
##
##         Importance
## STRESS      100.00
## Age          19.79
## BornAUS      16.80
## EDU          11.73
## Survey        2.16
## Weekend       0.00
```

```
plot(varImp(rfModel))
```

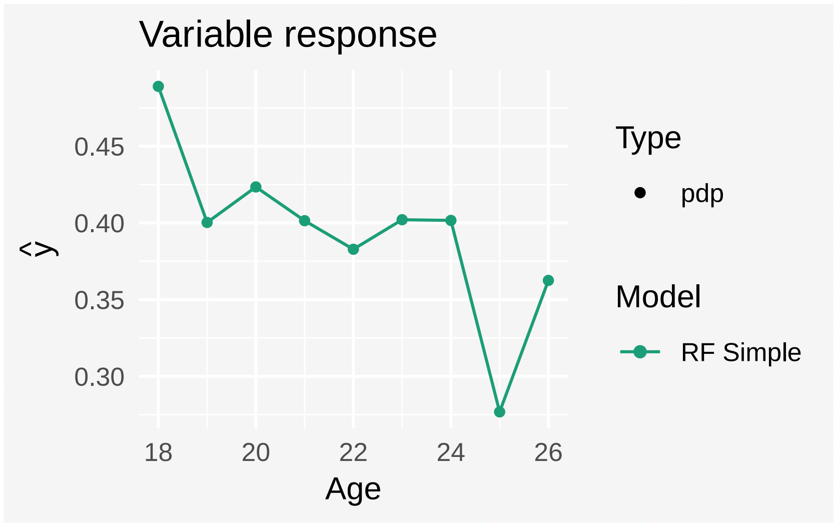


RFs and most machine learning models do not have regression coefficients or other simple parameters that can be easily interpretted. For example, with RFs, even though they are comprised of CARTs which can be used and interpretted, there often are hundreds of trees. The splits themselves are not easily interpretted by a human. However, it is possible to generate predictions from the models across a range of values. These partial dependence plots (PDP) attempt to plot the isolated impact of changing the level of one variable on the model predictions and they allow us to "see" how the model overall makes choices. We specify a custom predict function so that we are looking at something like the predicted probability that someone will be happy. These plots show suggest that people are only happy at fairly low stress values. Once stress gets above around 2-3, people become fairly unlikely to be happy. The results for age do not show any simple trend, although perhaps a trend towards lower probability of happiness with older age.

```
rfExplain <- explain(model = rfModel, data = dtrain,
    label = "RF Simple", y = dtrain$Happy, predict_function = function(model,
        x) predict(model, x, type = "prob")[,
        2])

rfPDP <- variable_response(explainer = rfExplain,
    variable = "STRESS", type = "pdp")

plot(rfPDP)
```



```
rfPDP2 <- variable_response(explainer = rfExplain,
    variable = "Age", type = "pdp")

plot(rfPDP2)
```



3.5 *In Class Activity (Workbook)*

This activity also will form the workbook activity for the week. Using the code above, change some of the predictor variables. You can either use different predictors then we did in class or you can add in more predictors from the dataset. Just use the variables already in the training dataset, which are shown here:

```
names(dtrain)

## [1] "EDU"     "SES_1"   "Age"     "BornAUS"
##  [5] "Female"  "STRESS"  "NegAff"  "PosAff"
##  [9] "Survey"  "Weekend" "Happy"
```

Once you have trained the model, use the `confusionMatrix()` function to look at how well it performs in the testing (not training) data.

Then plot the relative variable importances. Look at these to pick the two most important variables and for these, create partial dependence plots showing how the probability of being happy is expected to change based on those two variables.

```
## train a RF model using some other predictors
## in place of or in addition to the predictors
## we used in the previous example
set.seed(12345)




## evaluate the overall model performance on
## the **test** data (as we did)



## plot the variable importances



## create partial dependence plots for your
## **two** most important predictors
```

4 Extra Resources and Materials

There are many excellent resources on random forests. One of the best introductions to machine learning including random forests is:

- James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning*. New York: Springer. *[note: this is a very good book written by well-respected experts. It also is about as gentle of an introduction as you will get for machine learning]*
- Website for the `caret` package: `https://topepo.github.io/caret/`

For more in depth learning and some articles, see:

- Breiman, L. (2001). Random forests. *Machine learning*, 45(1), 5-32. *[note: this is basically the classic paper on random forests]*

- Strobl, C., Malley, J., & Tutz, G. (2009). An introduction to recursive partitioning: rationale, application, and characteristics of classification and regression trees, bagging, and random forests. *Psychological methods, 14*(4), 323. *[note: this is aimed at a psychology audience, albeit with experience in methods]*
- Friedman, J., Hastie, T., & Tibshirani, R. (2001). *The elements of statistical learning*. New York: Springer series in statistics. *[note: this is quite an extensive introduction to various aspects of machine learning, but assumes more familiarity with statistics and some math and programming than does the other book I recommended]*