Decision Trees, Random Forests, and Nearest-Neighbor classifiers

Shane T. Mueller [shanem@mtu.edu](mailto:shanem@mtu.edu)

2019-03-05

# Decision Trees, Forests, and Nearest-Neighbors classifiers

The classic statistical decision theory on which LDA and QDA, and logistic regression are highly model-based. We assume the features are fit by some model, we fit that model, and use inferences from that model to make a decision. Using the model means we make assumptions, and if those assumptions are correct, we can have a lot of success. Not all classifiers make such strong assumptions, and three of these will be covered in this section: Decision trees, random forests, and K-Nearest Neighbor classifiers.

# Decision trees

Decision trees assume that the different predictors are independent and combine together to form a an overall likelihood of one class over another. However, this may not be true. Many times, we might want to make a classification rule based on a few simple measures. The notion is that you may have several measures, and by asking a few decisions about individual dimensions, end up with a classification decision. For example, such trees are frequently used in medical contexts. If you want to know if you are at risk for some disease, the decision process might be a series of yes/no questions, at the end of which a ‘high-risk’ or ‘low-risk’ label would be given:

1. Decision 1: Are you above 35?

* Yes: Use Decision 2
* No: Use Decision 3

1. Decision 2: Do you have testing scores above 1000?

* Yes: High risk
* No: Low risk

1. Decision 3: Do you a family history?

* Yes: etc.
* No: etc.

Notice that if we were trying to determine a class via LDA, we’d create a single composite score based on all the questions and make a decision at the end. This allows us to consider interactions between features conditionally, and so it can be very powerful. Tree-based decision tools can be useful in many cases:

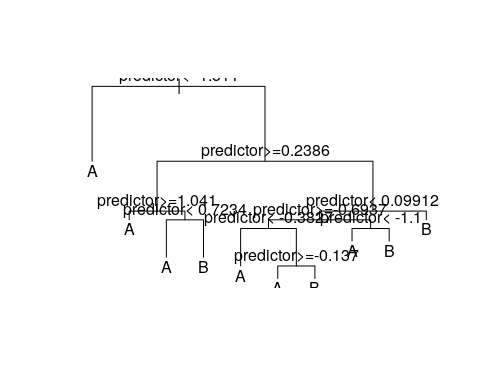
* When we want simple decision rules that can be applied by novices or people under time stress.
* When the structure of our classes are dependent or nested, or somehow hierarchical. Many natural categories have a hierarchical structure, so that the way you split one variable may depend on the value of another. For example, if you want to know if someone is ‘tall’, you first might determine their gender, and then use a different cutoff for each gender.
* When many of our observables are binary states. To classify lawmakers we can look at their voting patterns–which are always binary. We may be able to identify just a couple issues we care about that will tell us everything we need to know.
* When you have multiple categories. There is no need to restrict the classes to binary, unlike for the previous methods we examined.

To make a decision tree, we essentially have to use heuristic processes to determine the best order and cutoffs to use in making decisions, while identifying our error rates. Even with a single feature, we can make a decision tree that correctly classifies all the elements in a set (as long as all feature values are unique). So we also need to understand how many rules to use, in order to minimize over-fitting.

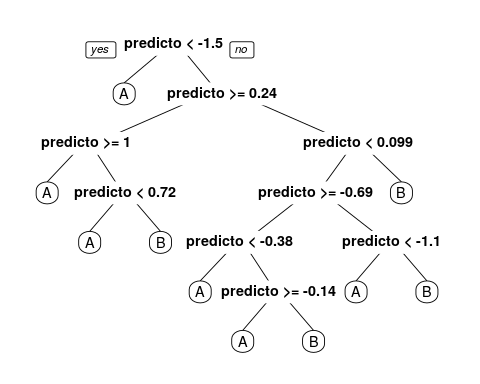
There are a number of software packages available for classification trees. One commonly-used package in R is called rpart.

’Let’s start with a simple tree made to classify elements on which we have only one measure:

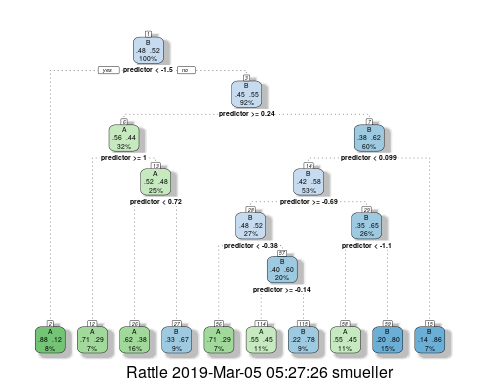
library(rpart)  
library(rattle) #for graphing   
library(rpart.plot) #also for graphing  
library(DAAG)  
classes <- sample(c("A", "B"), 100, replace = T)  
predictor <- rnorm(100)  
r1 <- rpart(classes ~ predictor, method = "class")  
  
plot(r1)  
text(r1)



prp(r1)



fancyRpartPlot(r1) #better visualization

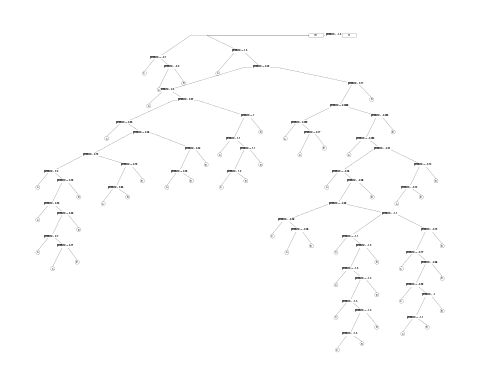


confusion(classes, predict(r1, type = "class"))

Overall accuracy = 0.7   
  
Confusion matrix   
 Predicted (cv)  
Actual [,1] [,2]  
 [1,] 0.812 0.188  
 [2,] 0.404 0.596

BNotice that even though the predictor had no true relationship to the categories, we were able to get 71% accuracy. We could in fact do better, just by adding rules:

r2 <- rpart(classes ~ predictor, method = "class", control = c(minsplit = 1,   
 minbucket = 1, cp = -1))  
prp(r2)



# this tree is a bit too large for this graphics method: fancyRpartPlot(r2)  
confusion(classes, predict(r2, type = "class"))

Overall accuracy = 1   
  
Confusion matrix   
 Predicted (cv)  
Actual [,1] [,2]  
 [1,] 1 0  
 [2,] 0 1

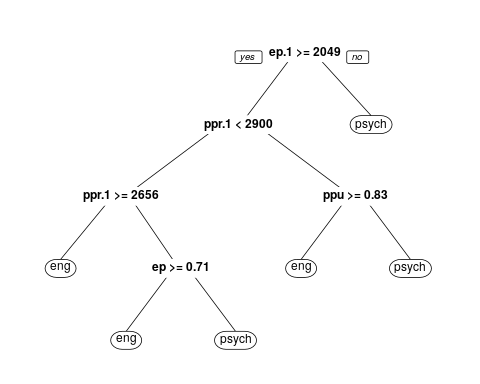
Now, we have completely predicted every item by successively dividing the line. There were three parameters we adjusted to do this. First, we set the minsplit and minbucket to 1–this determines the smallest size of the leaf nodes. Then, we set the cp argument to be negative (it defaults to .01). cp is a complexity parameter, that is a criterion for how much better each model must be before splitting a leaf node. A negative value will mean that any additional node is better. Notice that by default, rpart must choose the best number of nodes to use. This is a critical decision for a decision tree, as it impacts the complexity of the model, and how much it overfits the data.

How will this work for a real data set? Let’s re-load the engineering data set. In this data, we have asked both Engineering and Psychology students to determine whether pairs of words from Psychology, Engineering go together, and measured their time and accuracy.

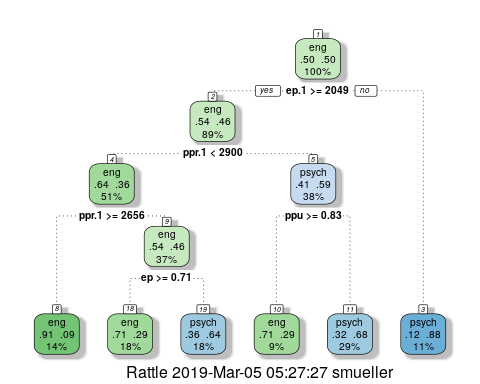
joint <- read.csv("eng-joint.csv")  
joint$eng <- as.factor(c("psych", "eng")[joint$eng + 1])  
  
  
## This is the partitioning tree:  
  
r1 <- rpart(eng ~ ., data = joint, method = "class")  
r1

n= 76   
  
node), split, n, loss, yval, (yprob)  
 \* denotes terminal node  
  
 1) root 76 38 eng (0.50000000 0.50000000)   
 2) ep.1>=2049.256 68 31 eng (0.54411765 0.45588235)   
 4) ppr.1< 2900.149 39 14 eng (0.64102564 0.35897436)   
 8) ppr.1>=2656.496 11 1 eng (0.90909091 0.09090909) \*  
 9) ppr.1< 2656.496 28 13 eng (0.53571429 0.46428571)   
 18) ep>=0.7083333 14 4 eng (0.71428571 0.28571429) \*  
 19) ep< 0.7083333 14 5 psych (0.35714286 0.64285714) \*  
 5) ppr.1>=2900.149 29 12 psych (0.41379310 0.58620690)   
 10) ppu>=0.8333333 7 2 eng (0.71428571 0.28571429) \*  
 11) ppu< 0.8333333 22 7 psych (0.31818182 0.68181818) \*  
 3) ep.1< 2049.256 8 1 psych (0.12500000 0.87500000) \*

prp(r1, cex = 0.75)



# text(r1,use.n=TRUE)  
fancyRpartPlot(r1)



confusion(joint$eng, predict(r1, type = "class"))

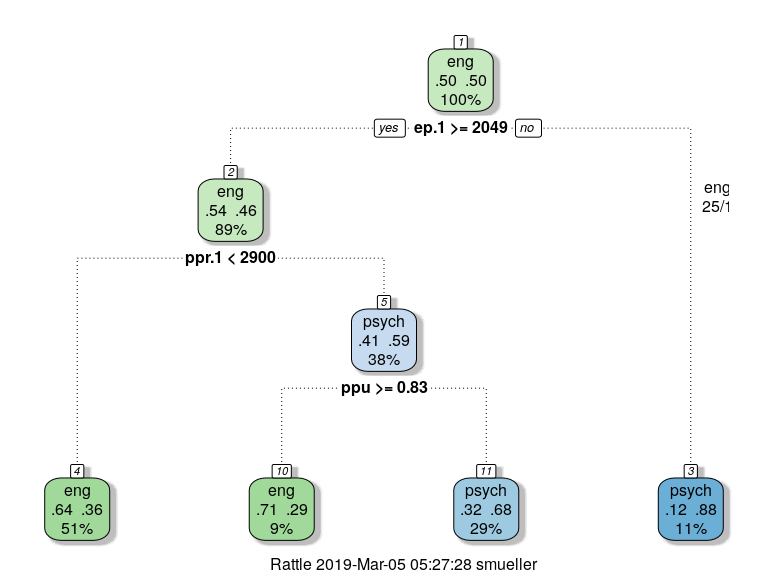
Overall accuracy = 0.737   
  
Confusion matrix   
 Predicted (cv)  
Actual eng psych  
 eng 0.658 0.342  
 psych 0.184 0.816

With the default full partitioning, we get 73% accuracy.. But the decision tree is fairly complicated, and we might want something simpler. Let’s only allow it to go to a depth of 2, by controlling maxdepth:

r2 <- rpart(eng ~ ., data = joint, method = "class", control = c(maxdepth = 3))  
r2

n= 76   
  
node), split, n, loss, yval, (yprob)  
 \* denotes terminal node  
  
 1) root 76 38 eng (0.5000000 0.5000000)   
 2) ep.1>=2049.256 68 31 eng (0.5441176 0.4558824)   
 4) ppr.1< 2900.149 39 14 eng (0.6410256 0.3589744) \*  
 5) ppr.1>=2900.149 29 12 psych (0.4137931 0.5862069)   
 10) ppu>=0.8333333 7 2 eng (0.7142857 0.2857143) \*  
 11) ppu< 0.8333333 22 7 psych (0.3181818 0.6818182) \*  
 3) ep.1< 2049.256 8 1 psych (0.1250000 0.8750000) \*

fancyRpartPlot(r2)  
text(r2, use.n = T)



confusion(joint$eng, predict(r2, type = "class"))

Overall accuracy = 0.684   
  
Confusion matrix   
 Predicted (cv)  
Actual eng psych  
 eng 0.789 0.211  
 psych 0.421 0.579

Here, we are down to 68% ‘correct’ classifications.

##Looking deeper If we look at the summary of the tree object, it gives us a lot of details about goodness of fit and decision points.

summary(r2)

Call:  
rpart(formula = eng ~ ., data = joint, method = "class", control = c(maxdepth = 3))  
 n= 76   
  
 CP nsplit rel error xerror xstd  
1 0.15789474 0 1.0000000 1.263158 0.1106647  
2 0.13157895 1 0.8421053 1.210526 0.1121371  
3 0.07894737 2 0.7105263 1.078947 0.1143498  
4 0.01000000 3 0.6315789 1.026316 0.1146681  
  
Variable importance  
 ep.1 ppr.1 ppu.1 ppu eer.1 eeu.1 ep   
 25 19 16 14 13 11 1   
  
Node number 1: 76 observations, complexity param=0.1578947  
 predicted class=eng expected loss=0.5 P(node) =1  
 class counts: 38 38  
 probabilities: 0.500 0.500   
 left son=2 (68 obs) right son=3 (8 obs)  
 Primary splits:  
 ep.1 < 2049.256 to the right, improve=2.514706, (0 missing)  
 eeu.1 < 2431.336 to the right, improve=2.326531, (0 missing)  
 ppu.1 < 4579.384 to the right, improve=2.072727, (0 missing)  
 eer.1 < 4687.518 to the right, improve=1.966874, (0 missing)  
 ppu < 0.8333333 to the right, improve=1.719298, (0 missing)  
 Surrogate splits:  
 ppu.1 < 1637.18 to the right, agree=0.947, adj=0.500, (0 split)  
 eeu.1 < 1747.891 to the right, agree=0.934, adj=0.375, (0 split)  
 eer.1 < 1450.41 to the right, agree=0.921, adj=0.250, (0 split)  
 ppr.1 < 1867.892 to the right, agree=0.921, adj=0.250, (0 split)  
  
Node number 2: 68 observations, complexity param=0.1315789  
 predicted class=eng expected loss=0.4558824 P(node) =0.8947368  
 class counts: 37 31  
 probabilities: 0.544 0.456   
 left son=4 (39 obs) right son=5 (29 obs)  
 Primary splits:  
 ppr.1 < 2900.149 to the left, improve=1.717611, (0 missing)  
 ppu < 0.8333333 to the right, improve=1.553072, (0 missing)  
 ppu.1 < 4579.384 to the right, improve=1.535294, (0 missing)  
 eer.1 < 4687.518 to the right, improve=1.529205, (0 missing)  
 ep.1 < 2754.762 to the left, improve=1.013682, (0 missing)  
 Surrogate splits:  
 eer.1 < 3007.036 to the left, agree=0.750, adj=0.414, (0 split)  
 ep.1 < 2684.733 to the left, agree=0.647, adj=0.172, (0 split)  
 ppu.1 < 2346.981 to the left, agree=0.632, adj=0.138, (0 split)  
 eeu.1 < 2806.34 to the left, agree=0.618, adj=0.103, (0 split)  
 ep < 0.7916667 to the left, agree=0.603, adj=0.069, (0 split)  
  
Node number 3: 8 observations  
 predicted class=psych expected loss=0.125 P(node) =0.1052632  
 class counts: 1 7  
 probabilities: 0.125 0.875   
  
Node number 4: 39 observations  
 predicted class=eng expected loss=0.3589744 P(node) =0.5131579  
 class counts: 25 14  
 probabilities: 0.641 0.359   
  
Node number 5: 29 observations, complexity param=0.07894737  
 predicted class=psych expected loss=0.4137931 P(node) =0.3815789  
 class counts: 12 17  
 probabilities: 0.414 0.586   
 left son=10 (7 obs) right son=11 (22 obs)  
 Primary splits:  
 ppu < 0.8333333 to the right, improve=1.6663680, (0 missing)  
 ppr.1 < 4858.184 to the right, improve=1.6663680, (0 missing)  
 eeu.1 < 3128.98 to the right, improve=1.5785810, (0 missing)  
 ep < 0.625 to the left, improve=1.0584390, (0 missing)  
 eer.1 < 3323.144 to the right, improve=0.8880131, (0 missing)  
 Surrogate splits:  
 ppu.1 < 6007.29 to the right, agree=0.828, adj=0.286, (0 split)  
 eer.1 < 4516.42 to the right, agree=0.793, adj=0.143, (0 split)  
 eeu.1 < 3768.95 to the right, agree=0.793, adj=0.143, (0 split)  
 ep.1 < 5327.593 to the right, agree=0.793, adj=0.143, (0 split)  
  
Node number 10: 7 observations  
 predicted class=eng expected loss=0.2857143 P(node) =0.09210526  
 class counts: 5 2  
 probabilities: 0.714 0.286   
  
Node number 11: 22 observations  
 predicted class=psych expected loss=0.3181818 P(node) =0.2894737  
 class counts: 7 15  
 probabilities: 0.318 0.682

This model seems to fit a lot better than our earlier LDA models, which suggest that it is probably overfitting. Cross-validation can be done within the control parameter :

r3 <- rpart(eng ~ ., data = joint, method = "class", control = c(maxdepth = 3,   
 xval = 10))  
r3

n= 76   
  
node), split, n, loss, yval, (yprob)  
 \* denotes terminal node  
  
 1) root 76 38 eng (0.5000000 0.5000000)   
 2) ep.1>=2049.256 68 31 eng (0.5441176 0.4558824)   
 4) ppr.1< 2900.149 39 14 eng (0.6410256 0.3589744) \*  
 5) ppr.1>=2900.149 29 12 psych (0.4137931 0.5862069)   
 10) ppu>=0.8333333 7 2 eng (0.7142857 0.2857143) \*  
 11) ppu< 0.8333333 22 7 psych (0.3181818 0.6818182) \*  
 3) ep.1< 2049.256 8 1 psych (0.1250000 0.8750000) \*

confusion(joint$eng, predict(r1, type = "class"))

Overall accuracy = 0.737   
  
Confusion matrix   
 Predicted (cv)  
Actual eng psych  
 eng 0.658 0.342  
 psych 0.184 0.816

confusion(joint$eng, predict(r3, type = "class"))

Overall accuracy = 0.684   
  
Confusion matrix   
 Predicted (cv)  
Actual eng psych  
 eng 0.789 0.211  
 psych 0.421 0.579

Accuracy goes down a bit, but the 74% accuracy is about what we achieved in the simple lda models.

Clearly, for partitioning trees, we have to be careful about overfitting, because we can always easily get the perfect classification.

#Random Forests

One advantage of partitioning/decision trees is that they are fast and easy to make. They are also considered interpretable and easy to use. A tree like this can be used by a doctor or a medlical counselor to help understand the risk for a disease, by asking a few simple questions.

The downsides of a decision tree is that they are often not very good, especially once they have been trimmed to avoid over-fitting. Recently, researchers have been interested in combining the results of many small (and often not-very-good) classifiers to make one better one. This often is described as ‘boosting’, or `ensemble’ methods, and there are a number of ways to achieve this. Doing this in a particular way with decision trees is referred to as a ‘random forest’ (see Breiman and Cutler).

Random forests can be used for both regression and classification (trees can be used in either way as well), and the classification and regression trees (CART) approach is a method that supports both. A random forest works as follows:

* Build trees (where N may be hundreds), where each tree is built from a random subset of features/variables. That is, on each step, choose the best variable to divide the branch based on a random subset of variables.

For each tree:

1. Pick a subset of the variables.
2. Build a tree

* Then, to classify any item, have each tree determine its best guess, and then take the most frequent outcome (or give a probabilistic answer based on the balance of evidence).

The randomForest package in R supports building these models. Because these can be a bit more difficult to comprehend, there is a companion package randomForestExplainer that is also handy in digging down on the types of forests derived.

library(randomForest)  
library(randomForestExplainer)  
  
rf <- randomForest(x = joint[, -1], y = joint$eng, proximity = T, ntree = 5)  
rf

Call:  
 randomForest(x = joint[, -1], y = joint$eng, ntree = 5, proximity = T)   
 Type of random forest: classification  
 Number of trees: 5  
No. of variables tried at each split: 3  
  
 OOB estimate of error rate: 63.77%  
Confusion matrix:  
 eng psych class.error  
eng 12 22 0.6470588  
psych 22 13 0.6285714

# printRandomForests(rf) ##look at all the RFs  
confusion(joint$eng, predict(rf))

Overall accuracy = 0.362   
  
Confusion matrix   
 Predicted (cv)  
Actual eng psych  
 eng 0.353 0.647  
 psych 0.629 0.371

For whatever reason, this model is below chance at making this classification. Looking at the trees, they are quite complex as well, with tens of rules. We can get a printout of individual subtrees with getTree:

(getTree(rf, 1))

left daughter right daughter split var split point status prediction  
1 2 3 4 0.5000000 1 0  
2 4 5 9 4220.4438418 1 0  
3 6 7 9 2892.7790434 1 0  
4 8 9 9 2799.4585854 1 0  
5 0 0 0 0.0000000 -1 2  
6 10 11 10 1916.4265009 1 0  
7 12 13 6 3439.3742499 1 0  
8 14 15 3 0.6250000 1 0  
9 16 17 2 0.8333333 1 0  
10 18 19 9 1830.9169578 1 0  
11 20 21 8 2719.4552605 1 0  
12 22 23 9 6355.0945496 1 0  
 [ reached getOption("max.print") -- omitted 23 rows ]

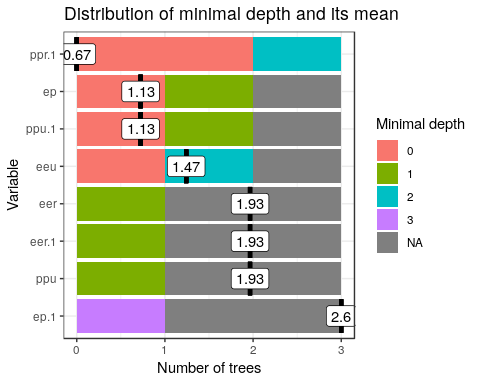
(getTree(rf, 5))

left daughter right daughter split var split point status prediction  
1 2 3 9 5065.2031024 1 0  
2 4 5 9 3528.8856466 1 0  
3 0 0 0 0.0000000 -1 1  
4 6 7 10 4563.6048647 1 0  
5 8 9 5 0.8333333 1 0  
6 10 11 7 2807.8686333 1 0  
7 12 13 9 1738.5203110 1 0  
8 0 0 0 0.0000000 -1 2  
9 14 15 6 4001.6659174 1 0  
10 16 17 9 1729.7592005 1 0  
11 18 19 10 3962.3749991 1 0  
12 20 21 10 5739.1531797 1 0  
 [ reached getOption("max.print") -- omitted 23 rows ]

rf <- randomForest(eng ~ ., data = joint, proximity = T, replace = F, mtry = 3,   
 ntree = 5, keep.inbag = T, sampsize = 10)  
  
confusion(joint$eng, predict(rf))

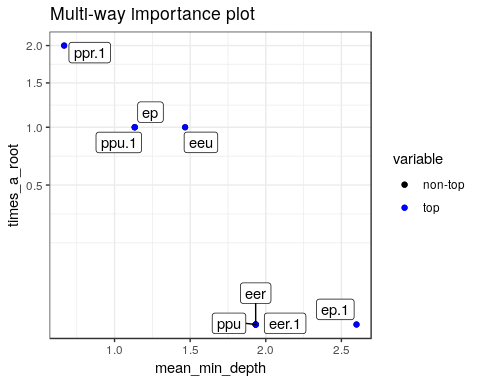
Overall accuracy = 0.526   
  
Confusion matrix   
 Predicted (cv)  
Actual eng psych  
 eng 0.368 0.632  
 psych 0.316 0.684

plot\_min\_depth\_distribution(rf)



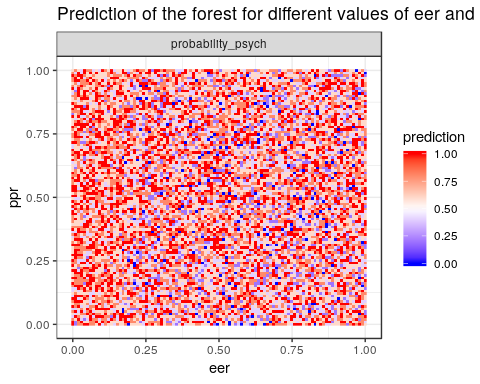
plot\_multi\_way\_importance(rf)

[1] "Warning: your forest does not contain information on local importance so 'accuracy\_decrease' measure cannot be extracted. To add it regrow the forest with the option localImp = TRUE and run this function again."

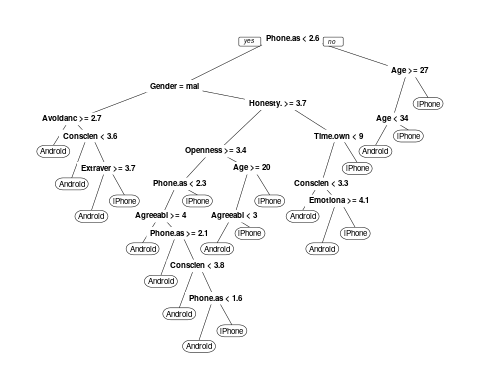


This plots the prediction of the forest for different values of variables:

plot\_predict\_interaction(rf, joint, "eer", "ppr")



phone <- read.csv("data\_study1.csv")  
phone.dt <- rpart(Smartphone ~ ., data = phone)  
prp(phone.dt, cex = 0.5)



confusion(phone$Smartphone, predict(phone.dt)[, 1] < 0.5)

Overall accuracy = 0.783   
  
Confusion matrix   
 Predicted (cv)  
Actual Android iPhone  
 Android 0.658 0.342  
 iPhone 0.129 0.871

phone.rf <- randomForest(Smartphone ~ ., data = phone)  
phone.rf

Call:  
 randomForest(formula = Smartphone ~ ., data = phone)   
 Type of random forest: classification  
 Number of trees: 500  
No. of variables tried at each split: 3  
  
 OOB estimate of error rate: 38.19%  
Confusion matrix:  
 Android iPhone class.error  
Android 98 121 0.5525114  
iPhone 81 229 0.2612903

confusion(phone$Smartphone, predict(phone.rf))

Overall accuracy = 0.618   
  
Confusion matrix   
 Predicted (cv)  
Actual Android iPhone  
 Android 0.447 0.553  
 iPhone 0.261 0.739

This does not seem to be better than the other models for the iphone data, but at least is is comparable.

# K Nearest-Neighbor classifiers

Another non-model-based classifier are nearest-neighbor methods. These methods require you to develop a similarity or distance space–usually on the basis of a set of predictors or features. Once this distance space is defined, we determine the class by finding the nearest neighbor in that space, and using that label. Because this could be susceptible to noisy data, we usually find a set of neighbors and have them vote on the outcome. These are K-nearest neighbor classifiers, or KNN.

## Choosing K.

The choice of can have a big impact on the results. If is too small, classification will be highly impacted by local neighborhood; if it is too big (i.e., if K = N), it will always just respond with the most likely response in the entire population.

## Distance Metric

The distance metric used in KNN easily be chosen poorly. For example, if you want to decide on political party based on age, income level, and gender, you need to figure out how to weigh and combine these. By default, you might add differences along each dimension, which would combine something that differs on the scale of dozens with something that differs on the scale of thousands, and so the KNN model would essentially ignore gender and age.

## Normalization selection

A typical approach would be to normalize all variables first. Then each variable has equal weight and range.

## Using KNN

Unlike previous methods where we train a system based on data and then make classifications, for KNN the trained system is just the data.

## Libraries and functions.

There are several options for knn classifers. Library: Class Function: knn knn.cv

### Library: kknn

Function: kknn This provides a ‘weighted k-nearest neighbor classifier’. Here, weighting is a weighting kernel, which gives greater weights to the nearer neighbors.

### Library: klaR

Function: sknn A ‘simple’ knn function. Permits using a weighting kernel as well.

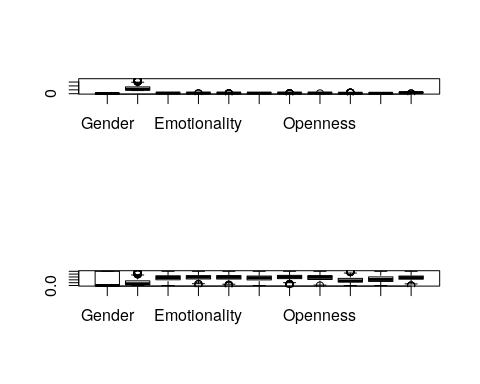
### Function: nm

This provides a ‘nearest mean’ classifier

## Example

In general, we need to normalize the variables first.

# Normalizing the values by creating a new function  
normal = function(x) {  
 xn = (x - min(x))/(max(x) - min(x))  
 return(xn)  
}  
  
phone2 <- phone  
for (i in 2:ncol(phone2)) {  
 phone2[, i] = normal(as.numeric(phone2[, i]))  
}  
  
  
# checking the range of values after normalization  
par(mfrow = c(2, 1))  
boxplot(phone[, 2:12])  
boxplot(phone2[, 2:12])

 ##fitting a simple model

library(class)  
m1 = knn(train = phone2[, -1], test = phone2[, -1], cl = phone2$Smartphone,   
 k = 10)  
confusion(phone2$Smartphone, m1)

Overall accuracy = 0.696   
  
Confusion matrix   
 Predicted (cv)  
Actual Android iPhone  
 Android 0.566 0.434  
 iPhone 0.213 0.787

That is pretty good, but what if we expand k

m1 = knn(train = phone2[, -1], test = phone2[, -1], cl = phone2$Smartphone,   
 k = 25)  
confusion(phone2$Smartphone, m1)

Overall accuracy = 0.686   
  
Confusion matrix   
 Predicted (cv)  
Actual Android iPhone  
 Android 0.534 0.466  
 iPhone 0.206 0.794

It looks we the choice of k is not going to matter much. What we could do is play with cross-validation though–right now we are testing on the same set as we are ‘training’ with, which will boost our accuracy (since the correct answer will always be one of the elements.) We can also try to do variable selection to change the similarity space to something that works better.

phone2 <- phone2[order(runif(nrow(phone2))), ] #random sort  
phone.train = phone2[1:450, 2:12]  
phone.test = phone2[451:nrow(phone2), 2:12]  
phone.train.target = phone2[1:450, 1]  
phone.test.target = phone2[451:nrow(phone2), 1]  
  
train <- sample(1:nrow(phone2), 300)  
m2 <- knn(train = phone.train, test = phone.test, cl = phone.train.target, k = 10)  
confusion(phone.test.target, m2)

Overall accuracy = 0.62   
  
Confusion matrix   
 Predicted (cv)  
Actual Android iPhone  
 Android 0.541 0.459  
 iPhone 0.310 0.690

# Additional resources

K-Nearest Neighbor Classification: MASS: Chapter 12.3, p. 341 <https://rstudio-pubs-static.s3.amazonaws.com/123438_3b9052ed40ec4cd2854b72d1aa154df9.html>

Relevant library/functions in R:

Decision Trees: Faraway, Chapter 13

Library: rpart (function rpart)

Random Forest Classification:

Reference: <https://www.r-bloggers.com/predicting-wine-quality-using-random-forests/> <https://www.stat.berkeley.edu/~breiman/RandomForests/>

Library: randomForest (Function randomForest)