Regression and Classification Trees

1 Regression Trees

The basic idea behind regression trees is the following: Group the n subjects into a bunch of groups based solely on the explanatory variables. Prediction for a future subject is then done in the following way. Look at the explanatory variable values for the future subject to figure which group he belongs to. Then predict his response value by the mean response for his group.

The main thing to understand here is how the grouping is constructed. Finding the *best* grouping is a computationally challenging task. In practice, a greedy algorithm, called *Recursive Partitioning*, is employed which produces a reasonable albeit not the best grouping.

1.1 Recursive Partitioning for Regression Trees

Recursive Partitioning is done in R via the function *rpart* from the library *rpart*. The grouping produced by recursive partitioning can be nicely represented by a tree. The tree is relatively easy to interpret.

Let us first use the *rpart* function to fit a regression tree to the bodyfat dataset.

```
dataDir <- "../../finalDataSets"
body = read.csv(file.path(dataDir, "bodyfat_short.csv"),
    header = T)
```

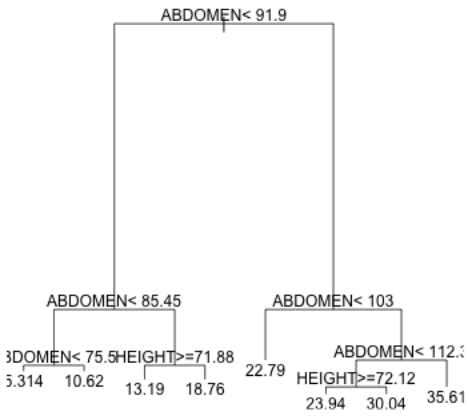
Let us fit a regression tree to Bodyfat percentage in terms of the explanatory variables Age, Weight, Height, Chest, Abdomen, Hip and Thigh.

```
library(rpart)
rt = rpart(BODYFAT ~ AGE + WEIGHT + HEIGHT + CHEST +
    ABDOMEN + HIP + THIGH, data = body)
rt
```

```
## n= 252
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 252 17578.99000 19.150790
##    2) ABDOMEN< 91.9 132  4698.25500 13.606060
##      4) ABDOMEN< 85.45 66  1303.62400 10.054550
##        8) ABDOMEN< 75.5 7   113.54860  5.314286 *
##        9) ABDOMEN>=75.5 59  1014.12300 10.616950 *
##      5) ABDOMEN>=85.45 66  1729.68100 17.157580
##       10) HEIGHT>=71.875 19   407.33790 13.189470 *
##       11) HEIGHT< 71.875 47   902.23110 18.761700 *
##    3) ABDOMEN>=91.9 120  4358.48000 25.250000
##      6) ABDOMEN< 103 81  1752.42000 22.788890 *
##      7) ABDOMEN>=103 39  1096.45200 30.361540
##       14) ABDOMEN< 112.3 28   413.60000 28.300000
##         28) HEIGHT>=72.125 8    89.39875 23.937500 *
##         29) HEIGHT< 72.125 20   111.04950 30.045000 *
##       15) ABDOMEN>=112.3 11   260.94910 35.609090 *
```

We shall interpret the output above a little later. But before that, let us plot the regression tree. This is easily done as follows.

```
plot(rt)
text(rt)
```



How to interpret this tree? At each node of the tree, there is a clause involving a variable and a cut-off. If the clause is met, then we go left and if it is not met, we go right. Here is how prediction works. Suppose we need to predict the bodyfat percentage of an individual who is 30 years of age, 180 pounds in weight, 70 inches tall and whose chest circumference is 95 cm, abdomen circumference is 90 cm, hip circumference is 100 cm and thigh circumference is 60 cm. The clause at the top of the tree is "ABDOMEN < 91.9" which is met for this person, so we move left. We then encounter the clause "ABDOMEN < 85.45" which is not met so we move right. This leads to the clause "HEIGHT $>= 71.88$" which is true for this person. So we move left. We then hit a "terminal node" where the displayed value is 13.19. We therefore predict this person's bodyfat as 13.19%.

How does *rpart* construct this tree? At each node of the tree, there is a clause involving a variable and a cut-off. How does *rpart* choose the variable and the cut-off.

Let us first understand how the clause is selected at the top of the tree (i.e., the first clause). Given a variable X_j and a cut-off c, we can divide the subjects into two groups: G_1 given by $X_j \leq c$ and G_2 given by $X_j > c$. The RSS corresponding to this split is defined as:

$$RSS(j, c) := \sum_{i \in G_1} (y_i - \bar{y}_1)^2 + \sum_{i \in G_2} (y_i - \bar{y}_2)^2$$

where $\bar{y}_1$ and $\bar{y}_2$ denote the mean values of the response in the groups G_1 and G_2 respectively.

The values of j and c for which $RSS(j, c)$ is the smallest give the best first split. The quantity $\min_{j,c} RSS(j, c)$ should be compared with $TSS = \sum_i (y_i - \bar{y})^2$. The ratio $\min_{j,c} RSS(j, c)/TSS$ is always smaller than 1 and the smaller it is, the greater we are gaining by the split.

For example, for the bodyfat dataset, the total sum of squares here is 17578.99. After spliitting based on "Abdomen < 91.9", one gets two groups with residuals sums of squares given by 4698.255 and 4358.48. Therefore the reduction in the sum of squares is:

```
(4698.255 + 4358.48)/17578.99
```

```
## [1] 0.5152022
```

The reduction in error due to this split is therefore 0.5152. This is the greatest reduction possible by splitting the data into two groups based on a variable and a cut-off.

The recursive partitioning algorithm for constructing the regression tree proceeds as follows: Find j and c (or S) such that $RSS(j, c)$ (or $RSS(j, S)$) is the smallest. Use the jth variable and the cut-off c (or the subset S) to divide the data into two groups: G_1 given by $X_j \leq c$ and G_2 given by $X_j > c$ (or G_1 given by $X_j \in S$ and G_2 given by $X_j \notin S$). Repeat this process within each group separately.

The depth of the branches in the tree are proportional to the reduction in error (residual sum of squares) due to the split. In the bodyfat dataset, the reduction in sum of squares due to the first split was 0.5152. For this dataset, this is apparently a big reduction compared to subsequence reductions and this is why it is plotted by a big branch.

Note that the tree here only uses the variables Abdomen and Height. The other variables are not being used; therefore *rpart*() automatically does some model selection.

How are categorical explanatory variables dealt with? For a categorical explanatory variable, it clearly does not make sense to put a cut-off across its value. For such variables, the groups (or splits) are created in the following way. Suppose X_j is a categorical variable that takes k values: $\{a_1, \ldots, a_k\}$. Then for every subset $S \subseteq \{a_1, \ldots, a_k\}$, the data are split into the two groups: G_1 given by $X_j \in S$ and G_2 given by $X_j \notin S$.

Here is an example with categorical explanatory variables. This is the college dataset.

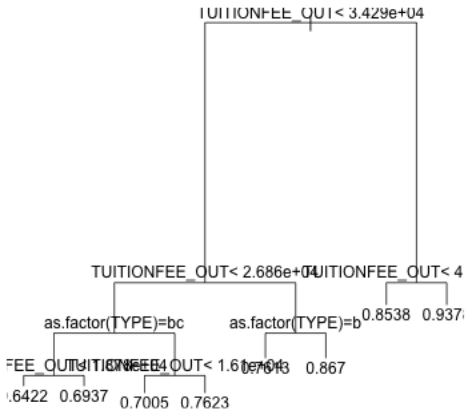
```
scorecard = read.csv(file.path(dataDir, "college_short.csv"))
names(scorecard)
```

```
## [1] "SAT_AVG_ALL"    "AVGFACSAL"      "TUITIONFEE_IN"  "TUITIONFEE_OUT"
```

```
## [5] "UGDS"           "RET_FT4"        "PCTFLOAN"       "PFTFAC"
## [9] "TYPE"
```

Note that TYPE is a categorical variable. Let us fit a regression tree with retention rate as the response and tuition fee and type as the explanatory variables.

```
req = rpart(RET_FT4 ~ TUITIONFEE_OUT + as.factor(TYPE),
    data = scorecard)
plot(req)
text(req)
```




```
req
```

```
## n= 1241
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 1241 16.8642000 0.7609155
##    2) TUITIONFEE_OUT< 34290 1005 11.1054700 0.7312472
##      4) TUITIONFEE_OUT< 26855 731  8.1311670 0.7128963
##        8) as.factor(TYPE)=2,3 345  4.0915040 0.6787971
##         16) TUITIONFEE_OUT< 18785 100  1.3470310 0.6421740 *
##         17) TUITIONFEE_OUT>=18785 245  2.5556030 0.6937453 *
##        9) as.factor(TYPE)=1 386  3.2799700 0.7433736
##         18) TUITIONFEE_OUT< 16097.5 118  1.0737510 0.7004661 *
```

```
##         19) TUITIONFEE_OUT>=16097.5 268  1.8933230 0.7622657 *
##       5) TUITIONFEE_OUT>=26855 274  2.0713890 0.7802051
##        10) as.factor(TYPE)=2 225  1.4400930 0.7613084 *
##        11) as.factor(TYPE)=1,3 49  0.1820255 0.8669755 *
##     3) TUITIONFEE_OUT>=34290 236  1.1070330 0.8872572
##       6) TUITIONFEE_OUT< 41516 142  0.5717404 0.8537951 *
##       7) TUITIONFEE_OUT>=41516 94  0.1361027 0.9378064 *
```

Note the clauses "as.factor(TYPE) = bc" and "as.factor(TYPE) = b" appearing in the tree. How does one interpret this tree?

It may be noted that *rpart* gives an error if you try to put in interaction terms. Interaction is automatically included in regression trees.

For every regression tree T, we can define its RSS in the following way. Let the final groups generated by T be $G_1, \ldots, G_k$. Then the RSS of T is defined as

$$RSS(T) := \sum_{j=1}^{m} \sum_{i \in G_j} (y_i - \bar{y}_j)^2$$

where $\bar{y}_1, \ldots, \bar{y}_m$ denote the mean values of the response in each of the groups.

We can also define a notion of R^2 for the regression tree as:

$$R^2(T) := 1 - \frac{RSS(T)}{TSS}.$$

```
rt = rpart(BODYFAT ~ AGE + WEIGHT + HEIGHT + CHEST +
    ABDOMEN + HIP + THIGH, data = body)
1 - (sum(residuals(rt)^2))/(sum((body$BODYFAT - mean(body$BODYFAT))^2))
```

```
## [1] 0.7354195
```

Let us now briefly look at the biggest and most complicated issue for trees. How large should the tree be grown? How is *rpart()* deciding when to stop growing the regression tree? Very large trees obviously lead to over-fitting. One strategy is to stop when $\min_j \min_{c \text{ or } S} RSS(j, c \text{ or } S)/TSS$ for a group is not small. This would be the case when we are not gaining all that much by splitting further. This is actually not a very smart strategy. Why? Because incremental improvements due to each expansion of the tree may not necessarily always be decreasing.

Regression and classification trees were invented by Leo Breiman from UC Berkeley. He also had an idea for the tree size issue. He advocates against stopping the

recursive partitioning algorithm at some step. Instead, he recommends growing a **full tree** or a very large tree, $T_{\max}$. Now, given a parameter $\alpha \geq 0$ (called the *complexity parameter* and referred to as *cp*), Breiman recommends choosing $T(\alpha)$ as the *subtree* of $T_{\max}$ which minimizes

$$R_\alpha(T) := RSS(T) + \alpha(TSS)|T|$$

where $|T|$ is the number of terminal nodes of the tree T. This is done via a slightly complicated algorithm involving *weakest link cutting*. Because the number of possible subtrees of $T_{\max}$ is finite, it follows that the set of possible trees $\{T(\alpha), \alpha \geq 0\}$ has to be finite as well. Breiman showed that one can obtain $\alpha_1 > \alpha_2 > \ldots$ and *nested* subtrees $T(\alpha_1) < T(\alpha_2) < \ldots$ such that

$$T(\alpha) = T(\alpha_k) \text{ if } \alpha_k \leq \alpha < \alpha_{k-1}.$$

Here we take α_0 to be $+\infty$. After obtaining this sequence of trees $T(\alpha_1), T(\alpha_2), \ldots,$ the default choice in R is to take $\alpha = 0.01$ and then generating the tree corresponding to $T(\alpha_k)$ for which $\alpha_k \leq 0.01 < \alpha_{k-1}$. The parameter α is referred to as *cp* in R.

The *printcp*() function in R gives the values of $\alpha_1, \alpha_2, \ldots$ (up to the actual α) and also gives the number of splits of the tree $T(\alpha_k)$ for each k.

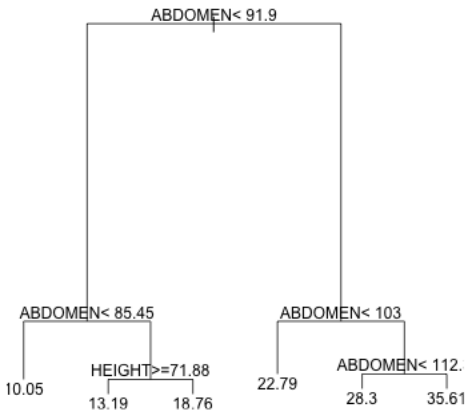
```
printcp(rt)
```

```
##
## Regression tree:
## rpart(formula = BODYFAT ~ AGE + WEIGHT + HEIGHT + CHEST + ABDOMEN +
##     HIP + THIGH, data = body)
##
## Variables actually used in tree construction:
## [1] ABDOMEN HEIGHT
##
## Root node error: 17579/252 = 69.758
##
## n= 252
##
##         CP nsplit rel error  xerror     xstd
## 1 0.484798      0   1.00000 1.00435 0.081140
## 2 0.094713      1   0.51520 0.57129 0.048806
## 3 0.085876      2   0.42049 0.51532 0.044394
## 4 0.024000      3   0.33461 0.39068 0.034624
## 5 0.023899      4   0.31061 0.39926 0.035027
## 6 0.012125      5   0.28672 0.36550 0.030966
## 7 0.010009      6   0.27459 0.37344 0.030195
## 8 0.010000      7   0.26458 0.38311 0.030910
```

The rows in the *printcp* output are different trees corresponding to their α values. Also given in the *printcp*() output are three other quantities: *relerror*, *xerror* and *xstd*. These mean the following. *relerror* for a tree T is simply $RSS(T)/TSS$. Because more deep trees have smaller RSS, this quantity will always decrease as we go down the column. The *xerror* term is an accuracy measure calculated by 10-fold cross validation (and then divided by TSS). This quantity will be random (i.e., different runs of *rpart*() will result in different values for *xerror*); this is because 10-fold cross-validation relies on randomly partitioning the data into 10 parts and the randomness of this partition results in *xerror* being random. The quantity *xstd* provides a standard deviation for the random quantity *xerror*. If we do not like the default choice of 0.01 for cp, we can choose a higher value of cp using *xerror* and *xstd*.

For this particular run, the xerror seems to be smallest at $cp = 0.012125$ and then this seems to increase. So we can use this value of cp instead of the default $cp = 0.01$. We will then get a smaller tree.

```
rt = rpart(BODYFAT ~ AGE + WEIGHT + HEIGHT + CHEST +
    ABDOMEN + HIP + THIGH, data = body, cp = 0.0122)
plot(rt)
text(rt)
```



Now we get a tree with 5 splits or 6 terminal nodes.

1.2 Classification Trees

The idea behind classification trees is similar. Just as in the case of regression trees, we can run recursive partitioning as follows. Given a variable X_j and a cut-off c, the subjects are divided into the two groups G_1 where $X_j \leq c$ and G_2 where $X_j > c$. The efficiency of this split is measured by the RSS:

$$RSS(j, c) := \sum_{i \in G_1} (y_i - \bar{y}_1)^2 + \sum_{i \in G_2} (y_i - \bar{y}_2)^2$$

where $\bar{y}_1$ and $\bar{y}_2$ denote the mean values of the response in the Groups G_1 and G_2 respectively. In classification problems, the response values are 0 or 1. Therfore $\bar{y}_1$ equals the proportion of ones in G_1 while $\bar{y}_2$ equals the proportion of ones in G_2. It is better to denote $\bar{y}_1$ and $\bar{y}_2$ by $\bar{p}_1$ and $\bar{p}_2$ respectively.

The formula for $RSS(j, c)$ then becomes:

$$RSS(j, c) = n_1 \bar{p}_1 (1 - \bar{p}_1) + n_2 \bar{p}_2 (1 - \bar{p}_2).$$

This quantity is also called the *Gini index* of the split corresponding to the jth variable and cut-off c. The function $p(1-p)$ takes its largest value at $p = 1/2$ and it is small when p is close to 0 or 1. Therefore the quantity $n_1 \bar{p}_1 (1 - \bar{p}_1)$ is small if either most of the response values in the group G_1 are 0 (in which case $\bar{p}_1 \approx 0$) or when most of the response values are 1 (in which case $\bar{p}_1 \approx 1$). A group is said to be pure if either most of the response values in the group are 0 or if most of the response values are 1. Thus the quantity $n_1 \bar{p}_1 (1 - \bar{p}_1)$ measures the impurity of a group. If $n_1 \bar{p}_1 (1 - \bar{p}_1)$ is low, then the group is pure and if it is high, it is impure. The group is maximally impure if $\bar{p}_1 = 1/2$.

The Gini Index, $RSS(j, c)$, is the sum of the impurities of the groups given by $X_j \leq c$ and $X_j > c$. The recursive partitioning algorithm determines j and c such that $RSS(j, c)$ is the smallest. This divides the data into two groups with $X_j \leq c$ and $X_j > c$. The process is then continued within each of these groups separately.

The quantity $n_1 \bar{p}_1 (1 - \bar{p}_1)$ is not the only function used for measuring the impurity of a group in classification. The key property of the function $p \mapsto p(1-p)$ is that it is symmetric about $1/2$, takes its maximum value at $1/2$ and it is small near the end points $p = 0$ and $p = 1$. Two other functions having this property are:

1. **Cross-entropy or Deviance**: Defined as $-2n_1 \left(\bar{p}_1 \log \bar{p}_1 + (1 - \bar{p}_1) \log(1 - \bar{p}_1) \right)$. This also takes its smallest value when $\bar{p}_1$ is 0 or 1 and it takes its maximum value when $\bar{p}_1 = 1/2$.

2. **Misclassification Error**: This is defined as $n_1 \min(\bar{p}_1, 1 - \bar{p}_1)$. This equals 0 when $\bar{p}_1$ is 0 or 1 and takes its maximum value when $\bar{p}_1 = 1/2$. If we classify all

the response values in the group G_1 by the majority value, then the number of response values misclassified in G_1 equals $n_1 \min(\bar{p}_1, 1 - \bar{p}_1)$. This explains the name misclassification error.

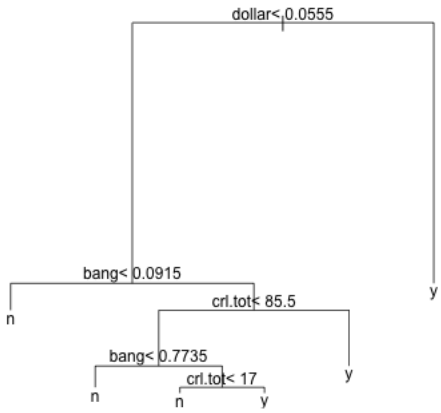
One can use Deviance or Misclassification error instead of the Gini index while growing a classification tree. The default in R is to use the Gini index.

Let us apply the classificationt tree to the email spam dataset.

```
library(DAAG)
data(spam7)
spam = spam7
```

The only change to the *rpart* function to classification is to use $method = "class"$.

```
library(rpart)
sprt = rpart(yesno ~ crl.tot + dollar + bang + money +
    n000 + make, method = "class", data = spam)
plot(sprt)
text(sprt)
```




```
sprt
```

```
## n= 4601
```

```
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 4601 1813 n (0.6059552 0.3940448)
##    2) dollar< 0.0555 3471  816 n (0.7649092 0.2350908)
##      4) bang< 0.0915 2420  246 n (0.8983471 0.1016529) *
##      5) bang>=0.0915 1051  481 y (0.4576594 0.5423406)
##       10) crl.tot< 85.5 535  175 n (0.6728972 0.3271028)
##         20) bang< 0.7735 418  106 n (0.7464115 0.2535885) *
##         21) bang>=0.7735 117   48 y (0.4102564 0.5897436)
##           42) crl.tot< 17 43   12 n (0.7209302 0.2790698) *
##           43) crl.tot>=17 74   17 y (0.2297297 0.7702703) *
##       11) crl.tot>=85.5 516  121 y (0.2344961 0.7655039) *
##    3) dollar>=0.0555 1130  133 y (0.1176991 0.8823009) *
```

The tree construction works exactly as in the regression tree. We can look at the various values of the α (cp) parameter and the associated trees and errors using the function *printcp*().

```
printcp(sprt)
```

```
##
## Classification tree:
## rpart(formula = yesno ~ crl.tot + dollar + bang + money + n000 +
##     make, data = spam, method = "class")
##
## Variables actually used in tree construction:
## [1] bang    crl.tot dollar
##
## Root node error: 1813/4601 = 0.39404
##
## n= 4601
##
##         CP nsplit rel error  xerror     xstd
## 1 0.476558      0   1.00000 1.00000 0.018282
## 2 0.075565      1   0.52344 0.54937 0.015408
## 3 0.011583      3   0.37231 0.38445 0.013414
## 4 0.010480      4   0.36073 0.37286 0.013246
## 5 0.010000      5   0.35025 0.37397 0.013262
```

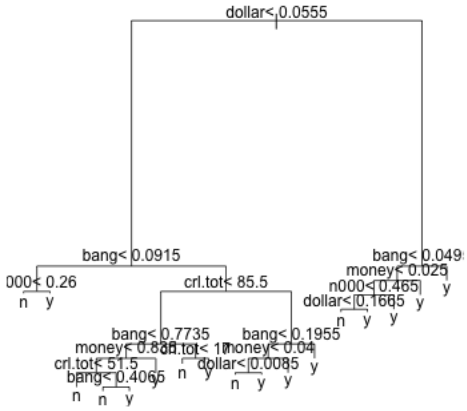
Notice that the *xerror* seems to decrease as *cp* decreases. We might want to set the *cp* to be lower than 0.01 so see how the *xerror* changes:

```
sprt = rpart(yesno ~ crl.tot + dollar + bang + money +
    n000 + make, method = "class", cp = 0.001, data = spam)
printcp(sprt)
```

```
##
## Classification tree:
## rpart(formula = yesno ~ crl.tot + dollar + bang + money + n000 +
##     make, data = spam, method = "class", cp = 0.001)
##
## Variables actually used in tree construction:
## [1] bang    crl.tot dollar  money   n000
##
## Root node error: 1813/4601 = 0.39404
##
## n= 4601
##
##           CP nsplit rel error  xerror     xstd
## 1  0.4765582      0   1.00000 1.00000 0.018282
## 2  0.0755654      1   0.52344 0.55268 0.015442
## 3  0.0115830      3   0.37231 0.38224 0.013382
## 4  0.0104799      4   0.36073 0.37176 0.013229
## 5  0.0063431      5   0.35025 0.35907 0.013040
## 6  0.0055157     10   0.31660 0.34970 0.012896
## 7  0.0044126     11   0.31109 0.33701 0.012696
## 8  0.0038610     12   0.30667 0.33480 0.012661
## 9  0.0027579     16   0.29123 0.32929 0.012572
## 10 0.0022063     17   0.28847 0.32819 0.012554
## 11 0.0019305     18   0.28627 0.32708 0.012536
## 12 0.0016547     20   0.28240 0.32653 0.012527
## 13 0.0010000     25   0.27413 0.32377 0.012482
```

Now the minimum *xerror* seems to be the tree with 16 splits (at $cp = 0.0027$). A reasonable choice of *cp* here is thereore 0.0028. We can refit the classification tree with this value of *cp*:

```
sprt = rpart(yesno ~ crl.tot + dollar + bang + money +
    n000 + make, method = "class", cp = 0.0028, data = spam)
plot(sprt)
text(sprt)
```



Let us now talk about getting predictions from the classification tree. Prediction is obtained in the usual way using the *predict*() function. The *predict*() function results in predicted probabilities (not 0-1 values). Suppose we have an email where $crl.tot = 100$, $dollar = 3$, $bang = 0.33$, $money = 1.2$, $n000 = 0$ and $make = 0.3$. Then the predicted probability for this email being spam is given by:

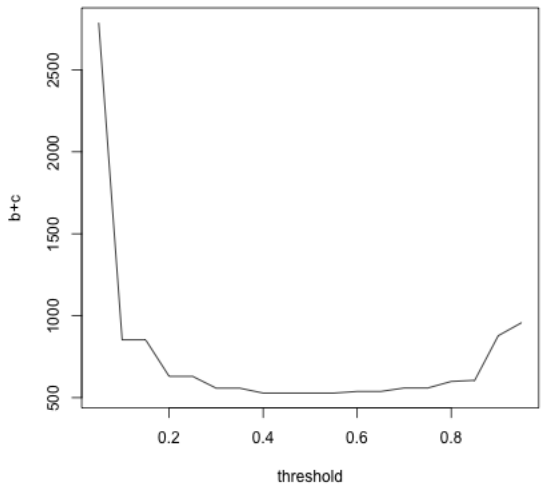
```
x0 = data.frame(crl.tot = 100, dollar = 3, bang = 0.33,
    money = 1.2, n000 = 0, make = 0.3)
predict(sprt, x0)
```

```
##            n        y
## 1 0.04916201 0.950838
```

The predicted probability is 0.950838. If we want to conver this into a 0-1 prediction, we can do this via a confusion matrix in the same way as for logistic regression.

```
y.tr = predict(sprt, spam)[, 2]
confusion <- function(y, yhat, thres) {
    n <- length(thres)
    conf <- matrix(0, length(thres), ncol = 4)
    colnames(conf) <- c("a", "b", "c", "d")
    for (i in 1:n) {
        a <- sum((!y) & (yhat <= thres[i]))
        b <- sum((!y) & (yhat > thres[i]))
        c <- sum((y) & (yhat <= thres[i]))
        d <- sum((y) & (yhat > thres[i]))
        conf[i, ] <- c(a, b, c, d)
```

```
    }
    return(conf)
}
v = seq(0.05, 0.95, by = 0.05)
y = as.numeric(spam$yesno == "y")
tree.conf = confusion(y, y.tr, v)
plot(v, tree.conf[, 2] + tree.conf[, 3], xlab = "threshold",
    ylab = "b+c", type = "l")
```



It seems that it is reasonable to choose a cut-off of 0.5. This would give the following precision and recall.

```
precision = 1449/(271 + 1449)
recall = 1449/(364 + 1449)
c(precision, recall)
```

```
## [1] 0.8424419 0.7992278
```

2 Random Forests

Random forests is another technique for regression and classification problems. They are best suited for problems where the main target is prediction (like most Kaggle

competitions). They are based on classification and regression trees. Essentially a random forest is a collection of (classification or regression) trees.

We shall use the R function *randomForest()* (in the package *randomForest()*) for constructing random forests. The algorithm underlying this function is the following. A large number *ntree* (default choice is 500) of trees (regression trees if it is a regression problem or classification trees if it is a classification problem) are constructed in the following manner. Two parameters *mtry* (whose default choice is $p/3$) and *nodesize* (whose default size is 5) are used in the tree construction process.

The i^{th} tree (for $i = 1, \ldots, ntree$) is constructed in the following manner:

1. Generate a *new* dataset having n observations by resampling uniformly at random with replacement from the existing set of observations. This resampling is the same as in bootstrap. Of course, some of the original set of observations will be repeated in this bootstrap sample (because of with replacement draws) while some other observations might be dropped altogether. The observations that do not appear in the bootstrap are referred to as *out of bag* (o.o.b) observations.

2. Construct a tree (regression tree if it is a regression problem or classification tree if it is a classification problem) based on the bootstrap sample. This tree construction is almost the same as the construction underlying the *rpart()* function but with two important differences.

   (a) At each stage of splitting the data into groups, *mtry* number of variables are selected at random from the available set of p variables and only splits based on these *mtry* variables are considered (by putting various thresholds across them). In contrast, in *rpart*, the best split is chosed by considering splits corresponding to all explanatory variables and all thresholds. This consequently makes random forests computationally fast. The *mtry* variables are chosen uniformly at random at each step of growing the tree. So it can happen, for example, that the first split in the tree is based on variables $1, 2, 3$ (here $mtry = 3$) resulting in two groups G_1 and G_2. In the group G_1, the first split might be based on variables $4, 5, 6$ and in the group G_2, the first split might be based on variables $1, 5, 6$ and so on.

   (b) The second important difference between tree construction in random forests versus *rpart*() is that in random forests, the trees are grown to full size. There is no pruning involved. More precisely, each tree is grown till the number of observations in each terminal node is no more than *nodesize*. This, of course, means that each individual tree will overfit the data. However, each individual tree will overfit in a different way and, finally, when we average the predictions due to the different trees, the overfitting will be removed.

At the end of the tree construction process, we will have *ntree* trees. These *ntree* trees will all be different (because each tree will be based on a different bootstrapped dataset and also because of randomness in the choice of variables to split) although they may be similar. Prediction now works in the following natural way. Given a new observation with explanatory variable values $x_1, \ldots, x_p$, each tree in our forest will yield a prediction for the response of this new obervation. Our final prediction will simply take the average of the predictions of the individual trees in case of regression or the majority vote of the predictions of the individual trees in case of classification.

Let us now see how random forests work for regression in the bodyfat dataset.

```
body = read.csv(file.path(dataDir, "bodyfat_short.csv"),
    header = T)
```

The syntax for the *randomForest* function works as follows:

```
library(randomForest)
ft = randomForest(BODYFAT ~ AGE + WEIGHT + HEIGHT +
    CHEST + ABDOMEN + HIP + THIGH, data = body, importance = TRUE)
ft
```

```
##
## Call:
##  randomForest(formula = BODYFAT ~ AGE + WEIGHT + HEIGHT + CHEST +      ABDOMEN + H
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##           Mean of squared residuals: 23.38991
##                     % Var explained: 66.47
```

R tells us that *ntree* is 500 and *mtry* (number of variables tried at each split) is 2. We can change these values if we want. The square of the mean of squared residuals roughly indicates the size of each residual. These residuals are slightly different from the usual residuals in that for each observation, the fitted value is computed from those trees where this observation is out of bag. But you can ignore this detail. The percent of variance explained is similar to R^2. The $importance = TRUE$ clause inside the randomForest function gives some variable importance measures. These can be seen by:

```
importance(ft)
```

```
##           %IncMSE IncNodePurity
## AGE      8.445399      1044.474
## WEIGHT  12.834322      2129.998
## HEIGHT  11.518788      1220.295
## CHEST   17.105418      3342.776
## ABDOMEN 35.321460      5775.380
## HIP     14.497659      2077.743
## THIGH   11.898637      1302.661
```

The exact meaning of these importance measures is nicely described in the help entry for the function importance. Basically, large values indicate importance. The variable *Abdomen* seems to be the most important (this is unsurprising given our previous experience with this dataset) for predicting bodyfat.

Now let us come to prediction with random forests. The R command for this is exactly the same as before. Suppose we want to the body fat percentage for a new individual whose AGE = 40, WEIGHT = 170, HEIGHT = 76, CHEST = 120, ABDOMEN = 100, HIP = 101 and THIGH = 60. The prediction given by random forest for this individual's response is obtained via

```
x0 = data.frame(AGE = 40, WEIGHT = 170, HEIGHT = 76,
    CHEST = 120, ABDOMEN = 100, HIP = 101, THIGH = 60)
predict(ft, x0)
```

```
##        1
## 24.14096
```

Now let us come to classification and consider the email spam dataset. The syntax is almost the same as regression.

```
library(DAAG)
data(spam7)
spam = spam7
sprf = randomForest(as.factor(yesno) ~ crl.tot + dollar +
    bang + money + n000 + make, data = spam)
sprf
```

```
##
```

```
## Call:
##  randomForest(formula = as.factor(yesno) ~ crl.tot + dollar +      bang + money +
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 11.69%
## Confusion matrix:
##      n    y class.error
## n 2646  142  0.05093257
## y  396 1417  0.21842250
```

Unlike classification tree, we do not have to explicitly mention $method = class$; the fact the response is given as a factor variable tells R to use a classification forest as opposed to a regression forest. The output is similar to the regression forest except that now we are also given a confusion matrix as well as some estimate of the misclassification error rate. Prediction is obtained in exactly the same was as regression forest via:

```
x0 = data.frame(crl.tot = 100, dollar = 3, bang = 0.33,
    money = 1.2, n000 = 0, make = 0.3)
predict(sprf, x0)
```

```
## 1
## y
## Levels: n y
```

Note that unlike logistic regression and classification tree, this directly gives a binary prediction (instead of a probability). So we don't even need to worry about thresholds.