Machine Learning for Finance Applications - Tree Based Methods

Brian Clark

2019

Contents

1 Introduction **1**

2 Basic Decision Tree Algorithm **2**
2.1 CART 2
2.2 C4.5 and C5.0 5

3 Examples Using R **5**
3.1 Examples from CHapter 8 Lab of James et al. (2013) 5
3.2 Mortgage Data using CART 10
3.3 Example 2 - Mortgage Data using C4.5 and C5.0 27

4 Model Enhancements for Decision Trees **30**
4.1 Bagging 31
4.2 Random Forests 32
4.3 Boosting 33

References **34**

1 Introduction

This chapter introduces tree based methods for finance applications. The topics closely follow Chapter 8 of James et al. (2013).[1] Another resource is Chapters 9 and 10 of J. Friedman, Hastie, and Tibshirani (2001), which provides a more technical discussion.[2] Decision tree based methods have been around for decades and are commonly used in may machine learning (ML) applications. Generally speaking, they are attractive due to their simplicity and relative ease of interpretability. They are also flexible in the sense that they can be applied to regression and classification problems. The main limitation of tree-based methods is that they are high variance approaches - meaning that small changes to the inputs can result in material changes to the output. This can greatly reduce the interpretability of the models. However, they remain a "workhorse" method for many applications.

Because of the high variance nature of tree-based methods, they are often used as a base model with advanced methods overlaid. For example, the stability and performance of decision tree method can be enhanced by techniques such as bagging, random forests, and boosting. As with many ML techniques, these enhancements come at a cost in terms of computation time so the most appropriate model tends to be situation specific.

Throughout this chapter, we will use the Fannie Mae mortgage data as a sample. The data is described in the course file "Machine Learning for Finance Applications - Fannie Mae Mortgage Data."

[1] The full text is available at http://www-bcf.usc.edu/ gareth/ISL/.
[2] The full text is available at https://web.stanford.edu/ hastie/Papers/ESLII.pdf.

2 Basic Decision Tree Algorithm

Decsion trees can be used in regression and classification settings. The basic steps are as follows (see Algortihm 8.1 in James et al. (2013)):

1. Use recursive binary splitting to build a tree.
2. Apply a pruning technique to obtain a sequence of best subtrees (this is the first part of controlling ovefitting).
3. Use k-fold cross-validation (or a similar out-of sample fitting procedure) to calibrate the pruning paramter(s). Note that in many finance applications cross-validation comes with its own set of issues.
4. Select the best model based on the pruning and cross-validation results.

The above algorithm is a generic recipe for building a decision tree. In practice, there are several variants and overlays designed to enhance the performance. Two of the most popular are CART (Breiman et al. (1984)) and C4.5 (Ross Quinlan (1993)) (since revamped as C5.0).[3]

2.1 CART

2.1.1 Regression Trees

Regression trees are used to predict a continuous variable (e.g., home values). Classificaiton trees are used to classify the response (e.g, default). Using the notation in Chapter 9 of J. Friedman, Hastie, and Tibshirani (2001), assume that we start with a dataset consisting of p input variables or features and a single response variable. Let N be the number of observations in the data and (x_i, y_i) for $i = 1, 2, ..., N$ with $x_i = (x_{i1}, x_{i2}, ..., x_{iN})$. The goal of the decision tree is to split the data in to M regions, or partitions, $R_1, R_2, ..., R_M$. For simplicity, start with the assumption that the responses are modeled as a constant, c_m, in each partition so the model can be written as follows:

$$f(x) = \sum_{m=1}^{M} c_m I(x \in R_m). \tag{1}$$

To choose the constants c_m. we could minimize the residual sum of squares and simply take the average of response variable in each region,

$$\hat{c_m} = mean(y_i | x_i \in R_m) \tag{2}$$

Rather than taking the simple mean, we could alternatively model a more complex function, but this could easily be implemented as an overlay.

2.1.2 Solving the Model

Ideally, we would choose our partitions such that we minimize the sum of squares of the residuals over the entire model space but this is generally computationally infeasible. As a result, most trees choose a *greedy* method. Recall that a greedy method is an algorithm that at each step chooses a set of parameters to minimize a loss function at that step without regard for future steps. In other words, it chooses a locally optimal solution at each stage.

At each node, the CART algorithm works by choosing the combination of a splitting variable x_j and split point s that minimizes the residual sum of squares at the given node. The result is a dataset that is split into two half planes such that

[3] A brief survey of several popular tree methods is given in @singh2014comparative.

$$R_1(j,s) = \{X|X_j \leq s\} \quad and \quad R_2(j,s) = \{X|X_j > s\}. \tag{3}$$

The choice of j and s involves solving the following optimization problem at each step:

$$min_{j,s}\left[min_{c_1}\sum_{x_i \in R_1(j,s)}(y_i - c_1)^2 + min_{c_1}\sum_{x_i \in R_2(j,s)}(y_i - c_2)^2\right] \tag{4}$$

The solution to the above problem can be done efficiently becuase the inner minimizaiton problems are given by the mean values of the response variable y_i within each partition R_i. That is,

$$\hat{c_1} = mean(y_i|x_i \in R_1(j,s)) \quad and \quad \hat{c_2} = mean(y_i|x_i \in R_2(j,s)). \tag{5}$$

Essentially, this involves finding the variable x_j and correspoding level s that best segments the data to minimize the errors. To build the rest of the tree, simply repeat this process recursively.

2.1.3 Classification Trees

As stated above, most decision tree algorithms including CART can be applied to classificaiotn problems as well. The main difference is that regression trees minimize a sum of squares of errors while classifiction trees use different criteria. Again, keeping with the notation in J. Friedman, Hastie, and Tibshirani (2001) (see, equation 9.17), let node m represent a region R_m with N_m observations and define the proportion of class k observations in node m be defiend as follows:

$$\hat{p_{mk}} = \frac{1}{N_m}\sum_{x_i \in R_m} I(y_i = k). \tag{6}$$

The classifier works by classifying observations into class $k(m)$ based on the majority class in node m (i.e., $k(m) = arg\ max_k\ \hat{p_{mk}}$). Three measures of impurity, $Q_m(T)$ are as follows:

1. Misclassificaiton error:
$$\frac{1}{N_m}\sum_{i \in R_m} I(y_i \neq k(m)) = 1 - \hat{p_{mk(m)}}$$

2. Gini index:
$$\sum_{k \neq k'} \hat{p_{mk}}\hat{p_{mk'}} = \sum_{k=1}^{K} \hat{p_{mk}}(1 - \hat{p_{mk}})$$

3. Cross-entropy or deviance:
$$-\sum_{k=1}^{K} \hat{p_{mk}} log(\hat{p_{mk}})$$

In practice, the latter two critieria are more popular for two reasons. One, they are differentiable which is helpful in numerical optimization routines. Second, they are more sensitive to changes in node probabilities than the misclassification error metric (see Figure 9.3 on page 309 of J. Friedman, Hastie, and Tibshirani (2001)).

2.1.4 Stopping Criteria

For both regression and classification trees, the fitting algorithm is the same. The difference is related to the splitting metrics, as described above. The tree continues until some stopping criterion is met. If the criterion or criteria are too loose, then the tree would tend to overfit the data. Alterntively, too tight a stopping criterion would pre-maturely stop the algorithm when addtional splits would be warranted.

In practice, choosing a good stopping criteria tends to be problem specific but there are a few general rules and considerations. First, the nature of the criterion can vary by applcation. Potential stopping criterion are as follows:

1. Informational gain. Stop the algorithm based on some informational gain paramters such as a decline in R^2. One potential issue with such a stopping criterion is that the CART algorithm is greedy so that a poor split at a given node does *not* prohibit subsequent splits from performing well. As such, the tree building may be pre-maturely stopped.

2. Number of instances classified. Stop the algorithm when a minimum number of instances appears on a given node. The downside is that thisis not based on any statistical criteria and will be inherently problem-specific.

3. Number of levels. Stop the alorithm after a pre-specified number of levels. The advantage of the method is that smaller trees are generally easier interpret. However, the method is not based on any statistical criteria.

An alternative approach is described on pages 307-8 of J. Friedman, Hastie, and Tibshirani (2001). The basic strategy is to grow a large tree, T_0, and *prune* the tree to find a sub-tree that maximizes some performance statistic. Again using the notation in J. Friedman, Hastie, and Tibshirani (2001), pruning amounts to finding a subtree $T \subset T_0$ that optimizes the tradeoff between model fit and complexity.

As above, index the terminal nodes as m to represent regions R_m and let $|T|$ be the number of terminal nodes in T. Then equations 9.15 and 9.16 in J. Friedman, Hastie, and Tibshirani (2001) are as follows:

$$
\begin{aligned}
N_m &= \#\{x_i \in R_m\} \\
\hat{c_m} &= \frac{1}{N_m} \sum_{x_i \in R_m} y_i \\
Q_m(T) &= \frac{1}{N_m} \sum_{x_i \in R_m} (y_i - \hat{c_m})^2
\end{aligned} \tag{7}
$$

and the cost-complexity criterion is:

$$
C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|. \tag{8}
$$

The parameter α governs the bias-variance tradeoff in the sense that large values of α punish complexity and result in small trees. Small values of α result in larger trees. In the `rpart()` implementation of CART described below, this parameter is set as option `cp=.`. For each value of α, there is a tree T_α that minimizes $C_\alpha(T)$. The optimal value of α involves a cross-validation or similar approach.

2.2 C4.5 and C5.0

An alternative to the CART model is the C4.5 algorithm (Ross Quinlan (1993)) which has since been replaced by C5.0. Both algorithms are enhancements of the earlier ID3 algorithm.

The differences between CART and C5.0 (and C4.5) are outlined **here** (which is a nice summary of 10 popular ML algorithms). Some key differences are as follows:

1. The splitting decisions are based on informaiton gain (entropy).
2. The pruning procedure is done in a single pass using binomial confidence limits. (CART uses an iterative method based on the cost-complexity criterion given by Eq. (8).)
3. C4.5/C5.0 allow for more than two splits at a given node (CART uses only binary splits).
4. C4.5/C5.0 handles missing values using a probabilistic approach. Essentially, instances with missing values are assigned to the most probabilistic outcome.
5. The tuning parameters are:
    - The pruning procedure is based on a pessimistic estimate of the error rate associated with the set of the N classes.
    - The tree is built until a minimum number of instances are on a terminal leaf.

3 Examples Using R

3.1 Examples from CHapter 8 Lab of James et al. (2013)

The code in this section re-produces the examples given in the chapter 8 lab in (**???**) we use the `caret` package. The `caret` packge is a wrapper that calls underlying packages such as `tree` or `C5.0` which are used by James et al. (2013). In other words, we will replicate the lab in Section 9.6 of James et al. (2013) using the `caret` wrapper. The reason is that `caret` is a wrapper for many ML algorithms so getting used to its syntax will undoubtedly save time and frustration in the long run.

First load the Carseats data from the `tree` package.

```
rm(list=ls())
library("tree")
library("ISLR")
attach(Carseats)
High <- ifelse(Sales <= 8, "No", "Yes")

# Make the data frame:
Carseats <- data.frame(Carseats, High)
```

The James et al. (2013) lab uses the `tree` package to fit a CART tree but we will do this via a `caret` wrapper. The model we will use reqiores teh `rpart` package and invokes `method = 'rpart'`.

```
library("rpart")
library("caret")
```

```
## Loading required package: lattice
```

```
## Loading required package: ggplot2
```

```
# Fit a model using 10-fold cross validation:
trControl <- trainControl(method="repeatedcv",number=10,repeats=10)
```

```r
# Set the tuning paramters
grid <- expand.grid(.cp=0.01)

# Fit the model for all variables EXCEPT SALES:
Carseats <- Carseats[,-1]
cart.fit <- train(High~., data=Carseats, trControl=trControl,
                  method="rpart",
                  tuneGrid=grid)
# List the variables in the summary:
names(cart.fit)
```

```
##  [1] "method"       "modelInfo"    "modelType"    "results"
##  [5] "pred"         "bestTune"     "call"         "dots"
##  [9] "metric"       "control"      "finalModel"   "preProcess"
## [13] "trainingData" "resample"     "resampledCM"  "perfNames"
## [17] "maximize"     "yLimits"      "times"        "levels"
## [21] "terms"        "coefnames"    "contrasts"    "xlevels"
```

```r
# Show the final model
cart.fit$finalModel
```

```
## n= 400
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##   1) root 400 164 No (0.59000000 0.41000000)
##     2) ShelveLocGood< 0.5 315  98 No (0.68888889 0.31111111)
##       4) Price>=92.5 269  66 No (0.75464684 0.24535316)
##         8) Advertising< 13.5 224  41 No (0.81696429 0.18303571)
##          16) CompPrice< 124.5 96   6 No (0.93750000 0.06250000) *
##          17) CompPrice>=124.5 128  35 No (0.72656250 0.27343750)
##            34) Price>=109.5 107  20 No (0.81308411 0.18691589)
##              68) Price>=126.5 65   6 No (0.90769231 0.09230769) *
##              69) Price< 126.5 42  14 No (0.66666667 0.33333333)
##               138) Age>=49.5 22   2 No (0.90909091 0.09090909) *
##               139) Age< 49.5 20   8 Yes (0.40000000 0.60000000) *
##            35) Price< 109.5 21   6 Yes (0.28571429 0.71428571) *
##         9) Advertising>=13.5 45  20 Yes (0.44444444 0.55555556)
##          18) Age>=54.5 20   5 No (0.75000000 0.25000000) *
##          19) Age< 54.5 25   5 Yes (0.20000000 0.80000000) *
##       5) Price< 92.5 46  14 Yes (0.30434783 0.69565217)
##        10) Income< 57 10   3 No (0.70000000 0.30000000) *
##        11) Income>=57 36   7 Yes (0.19444444 0.80555556) *
##     3) ShelveLocGood>=0.5 85  19 Yes (0.22352941 0.77647059)
##       6) Price>=142.5 12   3 No (0.75000000 0.25000000) *
##       7) Price< 142.5 73  10 Yes (0.13698630 0.86301370) *
```

We can plot the decision boundary using Michael Hahsler's `decisionplot()` function which can be found **here**.

```r
# Define the function:
decisionplot <- function(model, data, class = NULL, predict_type = "class",
  resolution = 100, showgrid = TRUE, ...) {

  if(!is.null(class)) cl <- data[,class] else cl <- 1
```

```r
  data <- data[,1:2]
  k <- length(unique(cl))

  plot(data, col = as.integer(cl)+1L, pch = as.integer(cl)+1L, ...)

  # make grid
  r <- sapply(data, range, na.rm = TRUE)
  xs <- seq(r[1,1], r[2,1], length.out = resolution)
  ys <- seq(r[1,2], r[2,2], length.out = resolution)
  g <- cbind(rep(xs, each=resolution), rep(ys, time = resolution))
  colnames(g) <- colnames(r)
  g <- as.data.frame(g)

  ### guess how to get class labels from predict
  ### (unfortunately not very consistent between models)
  # p <- predict(model, g, type = predict_type)
  p <- predict(model,newdata=g)
  if(is.list(p)) p <- p$class
  p <- as.factor(p)

  if(showgrid) points(g, col = as.integer(p)+1L, pch = ".")

  z <- matrix(as.integer(p), nrow = resolution, byrow = TRUE)
  contour(xs, ys, z, add = TRUE, drawlabels = FALSE,
    lwd = 2, levels = (1:(k-1))+.5)

  invisible(z)
}
```

Call the above function to plot the decision boundary. Note that the above function had to be amended slightly from the one directly given on Michael Hahsler's webpage. In particular, the line `## p <- predict(model, g, type = predict_type)` was replaced with `## p <- predict(model,newdata=g)`. The reason is that the original function was not intended for use with `caret`. This change *should* make the `decisionplot()` function more robust in the sense that it will work for various types of models when using `caret`, but I have yet to verify.

Note that to use the above funciton, we would need to fit a model of only two variables. We could for example re-fit a sub-model using only the two most important variables from the full model.

```r
# Find the most impotrtant variables:
cart.fit$finalModel$variable.importance
```

```
##         Price ShelveLocGood           Age   Advertising     CompPrice
##    39.3458304    28.9918954    13.0761815    12.7110483    10.2253806
##        Income    Population     Education         USYes
##     6.2611750     3.1669718     0.9670985     0.1411614
```

Now let's make a subset of the data and store it as `Carseats.2` and re-fit the model so we can plot it. Alternatively, we could have changed the plotting function to make it more robust and for example vary two features while holding the other variables constant using the mean values. Note also that we need to select numeric classes. Based on the above variable importance scores, we will choose *Age* and *Price*.

```r
Carseats.2 <- Carseats[,c("Price","Age","High")]
head(Carseats.2)
```

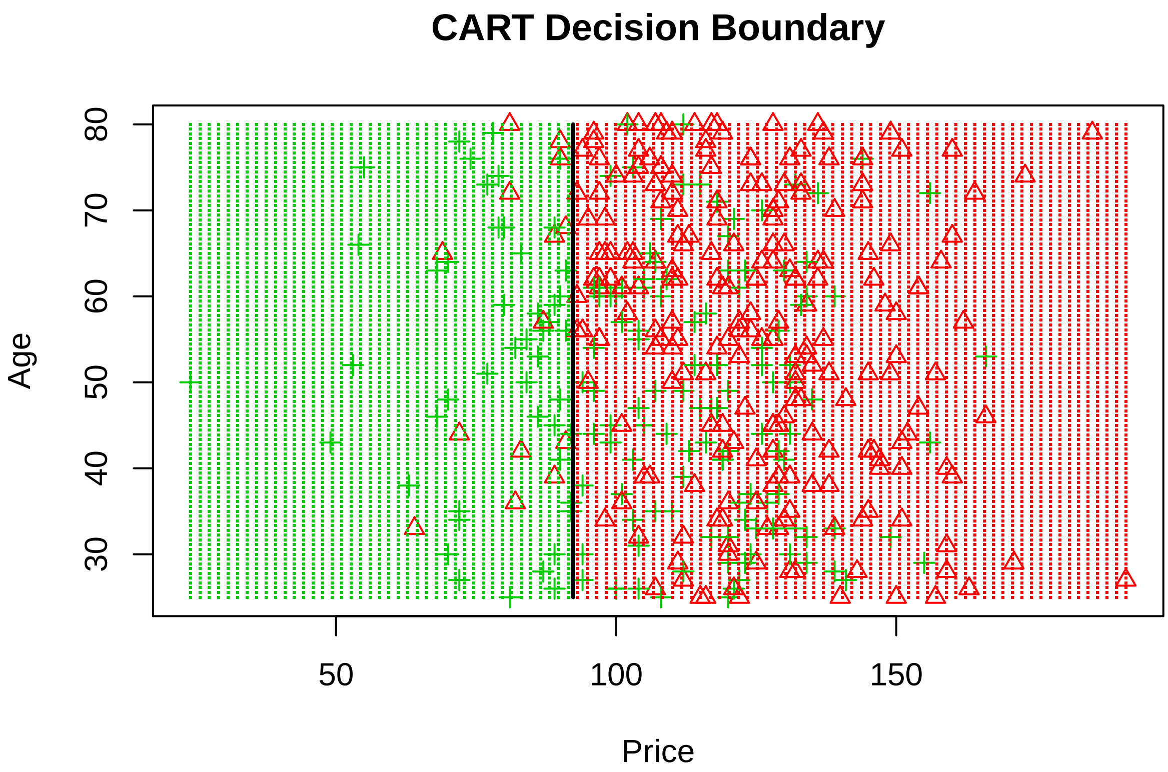
```
##   Price Age High
```

```
## 1   120  42  Yes
## 2    83  65  Yes
## 3    80  59  Yes
## 4    97  55   No
## 5   128  38   No
## 6    72  78  Yes
```

```r
# Set the tuning paramters
grid <- expand.grid(.cp=0.1)

# Fit eh small model
cart.fit.2 <- train(High~., data=Carseats.2, trControl=trControl,
                 method="rpart",
                 tuneGrid=grid)

# Show the final model
cart.fit.2$finalModel
```

```
## n= 400
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 400 164 No (0.5900000 0.4100000)
##   2) Price>=92.5 338 116 No (0.6568047 0.3431953) *
##   3) Price< 92.5 62  14 Yes (0.2258065 0.7741935) *
```

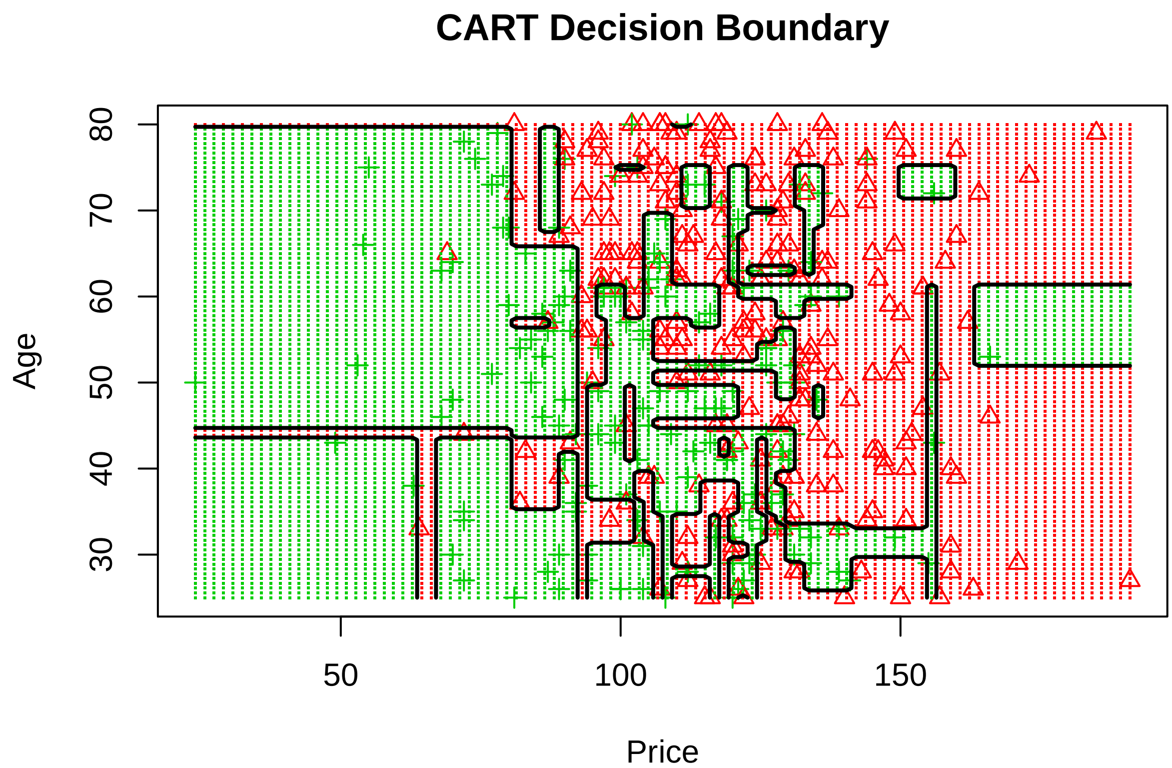
```r
# Call the amended function for the cost=10 model
decisionplot(cart.fit.2,data=Carseats.2,
             class="High",
             main="CART Decision Boundary")
```



We could repeat the analysis for a more overfit example by setting the complexity parameter to a very small value.

```
# Set the tuning paramters
grid <- expand.grid(.cp=0.000001)

# Fit the small model
cart.fit.2 <- train(High~., data=Carseats.2, trControl=trControl,
                  method="rpart",
                  tuneGrid=grid,
                  control=rpart.control(minsplit = 2))

# Show the final model
# cart.fit.2$finalModel
```

```
# Call the amended function for the cost=10 model
decisionplot(cart.fit.2,data=Carseats.2,
             class="High",
             main="CART Decision Boundary")
```



3.2 Mortgage Data using CART

We will use the Fannie Mae mortgage data to test the CART method. The CART model can be implemented using the **rpart** package in R. Another resource to understand CART is **here**. The `rpart` documentation is **here**.

First, initialize the R session and load the data.

```
rm(list=ls()) # clear the memory
setwd("C:/Users/CLARKB2/Documents/Classes/ML Course")
library("ggplot2")
library("reshape")
library("plm")
library("rpart")
library("zoo")
library("plyr")
library("dplyr")
library("stringr")
library("reshape2")
library("ggplot2")
library("pander")
library("DataCombine")
library("plm")
library("quantmod")

# Import the mortgage data:
```

```r
load("Mortgage_Annual.Rda")
```

The next step is to process the data. Within the .Rda file is a data frame called `p.mort.dat.annual`. As a matter of preference, rename `p.mort.dat.annual` as `df`. Set the data as a panel dataset (`pdata.frame()`) based on `LOAN_ID` and `year`.

```r
# Rename the data (matter of preference):
df <- p.mort.dat.annual
rm(p.mort.dat.annual)

df <- pdata.frame(df, index=c("LOAN_ID","year"),
                  stringsAsFactors = F)

# Print the class of the variable:
class(df)
```

```
## [1] "pdata.frame" "data.frame"
```

Next, we can generate variables that we need. First, define default as the first instance of 180+ days delinquent, which is given in the data by `F180_DTE` (to refer to the variable in `df`, use `df$F180_DTE`). `F180_DTE` is the date in which a loan first becomes 180+ delinquent. If the loan never becomes delinquent, it is missing (i.e., `NA`). To make the default variable, first find the indices where `df$F180_DTE == df$date` and save it as a vector, `tmp`. The line `df$def[tmp] <- 1` sets the new variable `def = 1` for the year in which the loan defaults.

```r
# Generate Variables we want:
# 1. Default 1/0 indicator (180+ DPD):
df$def <- 0
# Save the indices (rows) of
tmp <- which(df$F180_DTE == df$date)
df$def[tmp] <- 1
```

We may want to generate some other variables. For example, the variable `NUM_UNIT` gives the number of units is a house. Use `table(df$NUM_UNIT)` to print a frequency table of values of the number of units per house. Then, define a new variable to be a dummy if the number of units is greater than one (`MULTI_UN`).

```r
# 2. Replace NUM_UNIT with MULTI_UNIT dummy:
table(df$NUM_UNIT)
```

```
##
##      1      2      3      4
## 305028   6452   1051   1085
```

```r
df$MULTI_UN <- 0
tmp <- which(df$NUM_UNIT > 1)
df$MULTI_UN[tmp] <- 1
```

Finally, we can compress the data down to a single observation per loan. If you wanted to conduct a time-series or panal data analysis, you would skip this step. First, print the number of unique loans.

```r
# 3. Count the number of loans:
print(length(unique(df$LOAN_ID)))
```

```
## [1] 66704
```

Next, compress the data down to a single observation per loan. First, we need to generate a variable equal to one if the loan ever defaulted. We can do this using the `plm` package and grouping the data by `LOAN_ID`. Make a new dataset `df.annual` with a few additional variables: i) `def.max` and ii) `n` which is the row number to be used for keeping observaitons.

```r
# Compress the data to single loans:
df.annual <-df %>%
  group_by(LOAN_ID) %>%
  mutate(def.max = max(def)) %>%
  mutate(n = row_number()) %>%
  ungroup()
```

```
## Warning: `as_dictionary()` is soft-deprecated as of rlang 0.3.0.
## Please use `as_data_pronoun()` instead
## This warning is displayed once per session.

## Warning: `new_overscope()` is soft-deprecated as of rlang 0.2.0.
## Please use `new_data_mask()` instead
## This warning is displayed once per session.

## Warning: The `parent` argument of `new_data_mask()` is deprecated.
## The parent of the data mask is determined from either:
##
##   * The `env` argument of `eval_tidy()`
##   * Quosure environments when applicable
## This warning is displayed once per session.

## Warning: `overscope_clean()` is soft-deprecated as of rlang 0.2.0.
## This warning is displayed once per session.
```

```r
# Print the variable names in df.annual
names(df.annual)
```

```
##  [1] "year"              "ZIP_3"             "V1"
##  [4] "LOAN_ID"           "ORIG_CHN"          "Seller.Name"
##  [7] "ORIG_RT"           "ORIG_AMT"          "ORIG_TRM"
## [10] "ORIG_DTE"          "FRST_DTE"          "OLTV"
## [13] "OCLTV"             "NUM_BO"            "DTI"
## [16] "CSCORE_B"          "FTHB_FLG"          "PURPOSE"
## [19] "PROP_TYP"          "NUM_UNIT"          "OCC_STAT"
## [22] "STATE"             "MI_PCT"            "Product.Type"
## [25] "CSCORE_C"          "MI_TYPE"           "RELOCATION_FLG"
## [28] "Monthly.Rpt.Prd"   "Servicer.Name"     "LAST_RT"
## [31] "LAST_UPB"          "Loan.Age"          "Months.To.Legal.Mat"
## [34] "Adj.Month.To.Mat"  "Maturity.Date"     "MSA"
## [37] "Delq.Status"       "MOD_FLAG"          "Zero.Bal.Code"
## [40] "ZB_DTE"            "LPI_DTE"           "FCC_DTE"
## [43] "DISP_DT"           "FCC_COST"          "PP_COST"
## [46] "AR_COST"           "IE_COST"           "TAX_COST"
## [49] "NS_PROCS"          "CE_PROCS"          "RMW_PROCS"
## [52] "O_PROCS"           "NON_INT_UPB"       "REPCH_FLAG"
## [55] "TRANSFER_FLAG"     "CSCORE_MN"         "ORIG_VAL"
## [58] "PRIN_FORG_UPB"     "MODTRM_CHNG"       "MODUPB_CHNG"
## [61] "Fin_UPB"           "modfg_cost"        "C_modir_cost"
## [64] "C_modfb_cost"      "Count"             "LAST_STAT"
## [67] "lpi2disp"          "zb2disp"           "INT_COST"
## [70] "total_expense"     "total_proceeds"    "NET_LOSS"
## [73] "NET_SEV"           "Total_Cost"        "Tot_Procs"
## [76] "Tot_Liq_Ex"        "LAST_DTE"          "FMOD_DTE"
## [79] "FMOD_UPB"          "FCE_DTE"           "FCE_UPB"
## [82] "F180_DTE"          "F180_UPB"          "VinYr"
```

```
## [85] "ActYr"               "DispYr"              "MODIR_COST"
## [88] "MODFB_COST"          "MODTOT_COST"         "d.HPI"
## [91] "date"                "n"                   "n.obs"
## [94] "n.year"              "n.year.max"          "def"
## [97] "MULTI_UN"            "def.max"
```

Finally, we can save only one observation per loan.

```
# keep one obs per loan:
tmp <- which(df.annual$n == 1)
df.annual <- df.annual[tmp,]
dim(df.annual)
```

```
## [1] 66704    98
```

Notice that the number of rows is equal to the number of unique loans as shown above. Now, retain only the variables needed for the analysis.

```
# Keep only relevant variables for default analysis:
my.vars <- c("ORIG_CHN","ORIG_RT",
             "ORIG_AMT","ORIG_TRM","OLTV",
             "DTI","OCC_STAT",
             "MULTI_UN",
             "CSCORE_MN",
             "ORIG_VAL",
             "VinYr","def.max")
df.model <- subset(df.annual,select=my.vars)
names(df.model)
```

```
##  [1] "ORIG_CHN"  "ORIG_RT"   "ORIG_AMT"  "ORIG_TRM"  "OLTV"
##  [6] "DTI"       "OCC_STAT"  "MULTI_UN"  "CSCORE_MN" "ORIG_VAL"
## [11] "VinYr"     "def.max"
```

```
# Print the number of defaults/non-defaults
table(df.model$def.max)
```

```
##
##     0     1
## 64397  2307
```

```
tmp <- table(df.model$def.max)
df.rate <- tmp[2]/sum(tmp)*100
message(sprintf("The default rate is: %4.2f%%",df.rate))
```

```
## The default rate is: 3.46%
```

The last line prints the default rate. You can control the formatting just as you would in Matlab. The next step is to plot the data. We will use the `ggplot2()` package.

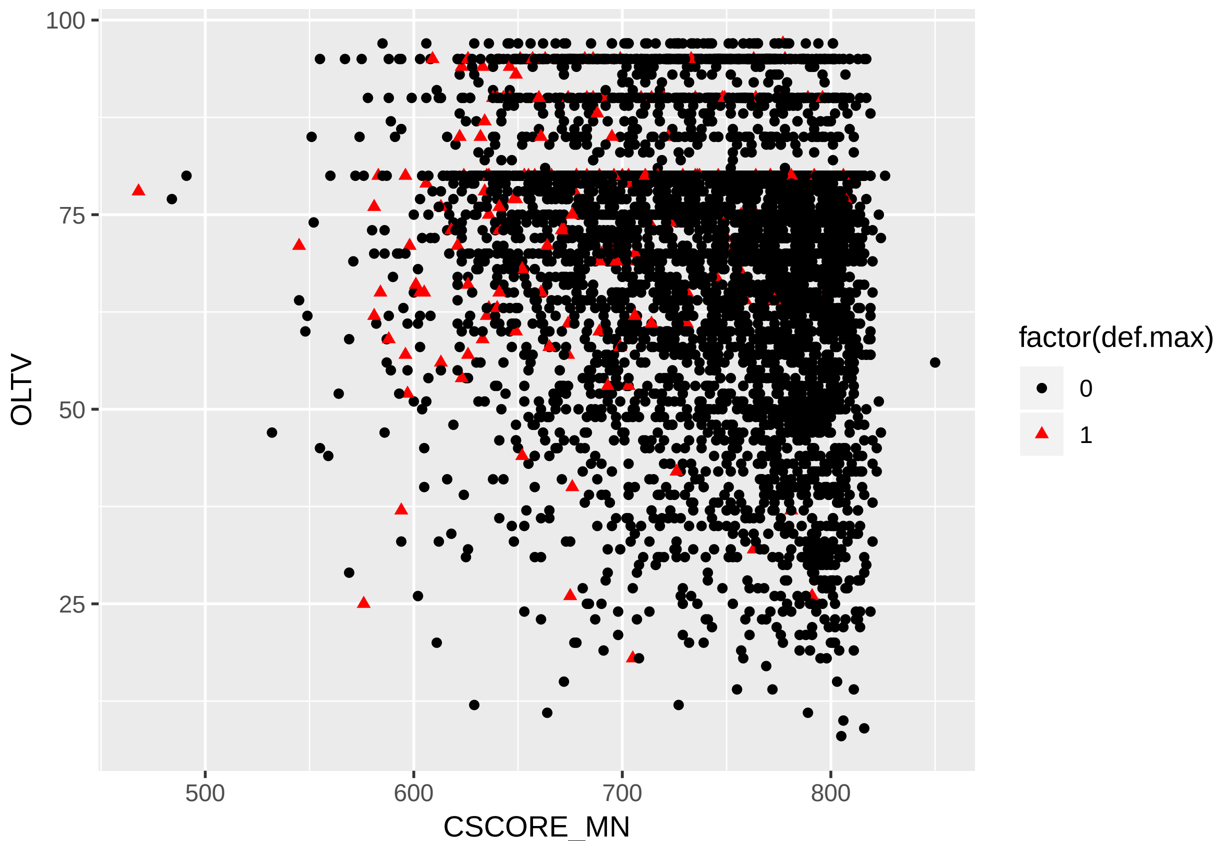
```
# --------------------
# Plot the data:
# Set the colors for the points:
mycolor <- c("black","red")

# Generate a small sample for plotting:
df.model.small <- df.model[sample(dim(df.model)[1],5000),]

# Plot Defaults vs. CSCORE_MN and OLTV:
sp <- ggplot(df.model.small, aes(x=CSCORE_MN, y=OLTV,
```

```r
                        color=factor(def.max))) +
  geom_point(aes(shape=factor(def.max))) +
  scale_fill_manual(values=mycolor) +
  scale_colour_manual(values=mycolor)
```

The above code saves the plot object as `sp`. To show the plot, simply print it. `ggplot()` has a nice feature that we can simply add to plots (similar to `hold on` in Matlab) as follows. As an example, we can make the size of the plotted points larger for defaulted loans (i.e., make the red dots bigger).

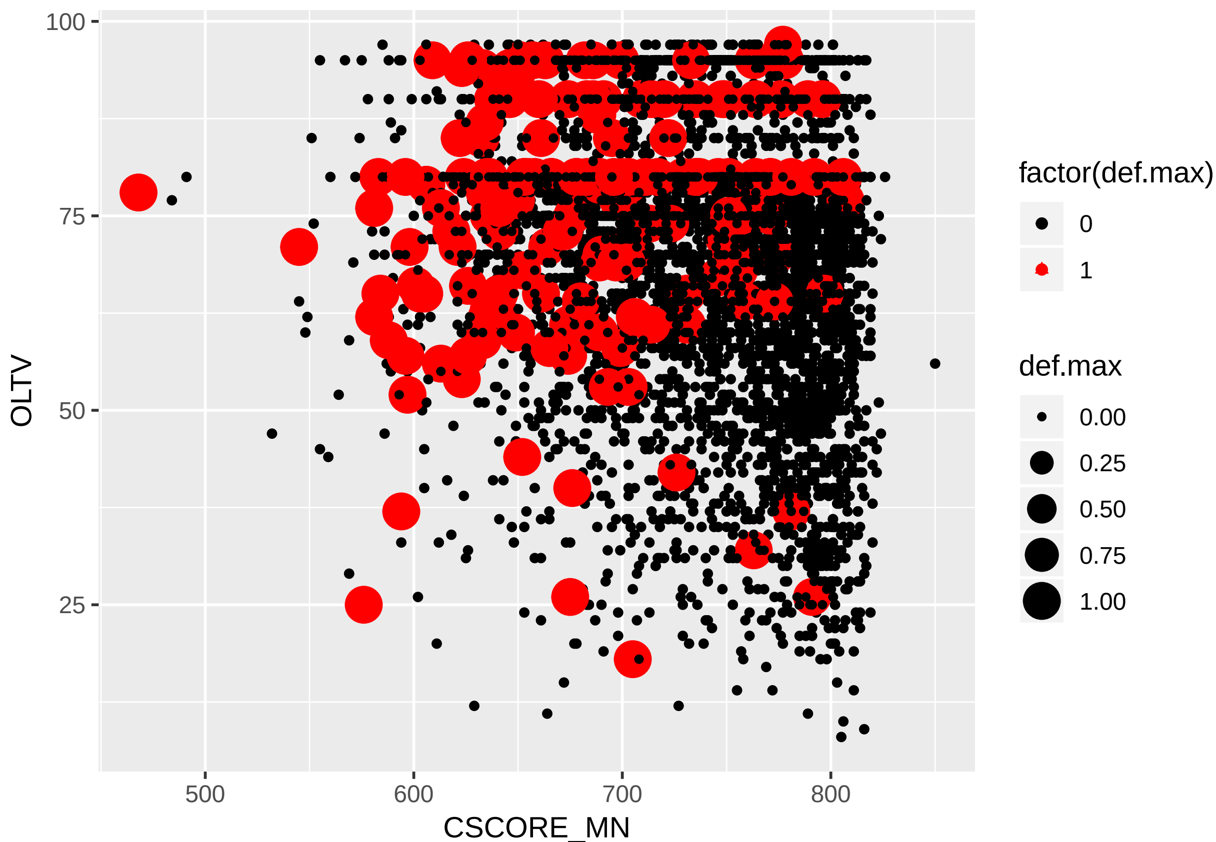
```r
print(sp)
```

```
## Warning: Removed 11 rows containing missing values (geom_point).
```




```r
# Add some new formatting:
sp <- sp + geom_point(aes(size=def.max))
print(sp)
```

```
## Warning: Removed 11 rows containing missing values (geom_point).

## Warning: Removed 11 rows containing missing values (geom_point).
```



Notice that the defaults tend to cluster more hevily for higher original LTV's and lower credit scores (note that `CSCORE_MN` is the minimum credit score of the borrowers). Finally, we can remove all unnecessary ojects.

```
# Print the objects in memory:
ls()
```

```
## [1] "df"              "df.annual"      "df.model"       "df.model.small"
## [5] "df.rate"         "my.vars"        "mycolor"        "sp"
## [9] "tmp"
```

```
# Remove all but df.model
rm(list=setdiff(ls(), "df.model"))
ls()
```

```
## [1] "df.model"
```

Now we only have a single object in memory - `df.model`. Note that in many ML applicaitons, memory becomes crucial. For this example, the data is pretty small so memory isn't a concern but in "big data" applications, the above commands could prove useful.

Finally, we have a dataset that we can use to fit some models. First, we will implement the CART algorithm which can be accessed as part of the `rpart` package (which was loaded above).

Start by fitting a large tree by setting `cp=1e-6` (i.e., set the complexity parameer to a small value). The essentially means teh tree will keep splitting even if there is very little informaitonal gain. Directly from `help("rpart.contol")`:

"`cp: complexity parameter. Any split that does not decrease the overall lack of fit by a factor of cp is not attempted. For instance, with anova splitting, this means that the overall R-squared must increase by cp at each step. The main role of this parameter is to` save computing time by pruning off splits that are obviously not worthwhile. Essentially,the user informs the program that any split which does not improve the fit by cp will likely be pruned off by cross-validation, and that hence the program need not pursue it."

We also set `minsplit=5` which means that the tree will only stop splitting nodes if there are less than 5 observations (instances) on a given node. Essentially, the combination of `cp=1e-6` and `minsplit=5` make the tree huge!

```
# --------------------------------
# Fit some models:
# Fit the CART tree using rpart:
cart.tree <- rpart(def.max ~ ., data = df.model, minsplit=5, cp=1e-6)
# 1. Model is defiend with def as the repsonse and all vars as features
# 2. the data is df.model, which is a data.frame
```

We have the model saved as `cart.tree`. Let's first look into the object and see what's there.

```
# Let's see what is in the model:
str(cart.tree)
```

```
## List of 15
##  $ frame              :'data.frame': 5053 obs. of  8 variables:
##   ..$ var       : Factor w/ 12 levels "<leaf>","CSCORE_MN",..: 12 2 12 2 12 7 11 6 9 2 ...
##   ..$ n         : int [1:5053] 66704 51114 43811 40964 28891 19247 19190 13936 13838 13766 ...
##   ..$ wt        : num [1:5053] 66704 51114 43811 40964 28891 ...
##   ..$ dev       : num [1:5053] 2227.2 707.9 329.5 239.6 93.7 ...
##   ..$ yval      : num [1:5053] 0.03459 0.01405 0.00758 0.00588 0.00325 ...
##   ..$ complexity: num [1:5053] 0.041422 0.005762 0.000813 0.000394 0.000206 ...
##   ..$ ncompete  : int [1:5053] 4 4 4 4 4 4 4 4 4 4 ...
##   ..$ nsurrogate: int [1:5053] 0 3 0 3 4 1 4 0 0 0 ...
##  $ where              : Named int [1:66704] 4235 67 1241 471 67 156 3972 99 4814 13 ...
##   ..- attr(*, "names")= chr [1:66704] "1" "2" "3" "4" ...
##  $ call               : language rpart(formula = def.max ~ ., data = df.model, minsplit = 5, cp = 1e
##  $ terms              :Classes 'terms', 'formula'  language def.max ~ ORIG_CHN + ORIG_RT + ORIG_AMT +
##   .. ..- attr(*, "variables")= language list(def.max, ORIG_CHN, ORIG_RT, ORIG_AMT, ORIG_TRM, OLTV, D
##   .. ..- attr(*, "factors")= int [1:12, 1:11] 0 1 0 0 0 0 0 0 0 0 ...
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:12] "def.max" "ORIG_CHN" "ORIG_RT" "ORIG_AMT" ...
##   .. .. .. ..$ : chr [1:11] "ORIG_CHN" "ORIG_RT" "ORIG_AMT" "ORIG_TRM" ...
##   .. ..- attr(*, "term.labels")= chr [1:11] "ORIG_CHN" "ORIG_RT" "ORIG_AMT" "ORIG_TRM" ...
##   .. ..- attr(*, "order")= int [1:11] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(def.max, ORIG_CHN, ORIG_RT, ORIG_AMT, ORIG_TRM, OLTV, DT
##   .. ..- attr(*, "dataClasses")= Named chr [1:12] "numeric" "character" "numeric" "numeric" ...
##   .. .. ..- attr(*, "names")= chr [1:12] "def.max" "ORIG_CHN" "ORIG_RT" "ORIG_AMT" ...
##  $ cptable            : num [1:699, 1:5] 0.04142 0.02859 0.00629 0.00576 0.00345 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:699] "1" "2" "3" "4" ...
##   .. ..$ : chr [1:5] "CP" "nsplit" "rel error" "xerror" ...
##  $ method             : chr "anova"
##  $ parms              : NULL
##  $ control            :List of 9
##   ..$ minsplit      : num 5
##   ..$ minbucket     : num 2
```

```
##   ..$ cp            : num 1e-06
##   ..$ maxcompete    : int 4
##   ..$ maxsurrogate  : int 5
##   ..$ usesurrogate  : int 2
##   ..$ surrogatestyle: int 0
##   ..$ maxdepth      : int 30
##   ..$ xval          : int 10
##  $ functions          :List of 2
##   ..$ summary:function (yval, dev, wt, ylevel, digits)
##   ..$ text   :function (yval, dev, wt, ylevel, digits, n, use.n)
##  $ numresp            : int 1
##  $ splits             : num [1:17496, 1:5] 66704 66519 66704 65386 66704 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:17496] "VinYr" "CSCORE_MN" "ORIG_RT" "DTI" ...
##   .. ..$ : chr [1:5] "count" "ncat" "improve" "index" ...
##  $ csplit             : int [1:2425, 1:18] 1 1 1 1 3 2 2 2 2 2 ...
##  $ variable.importance: Named num [1:11] 497 478 443 307 293 ...
##   ..- attr(*, "names")= chr [1:11] "ORIG_VAL" "ORIG_AMT" "CSCORE_MN" "DTI" ...
##  $ y                  : Named num [1:66704] 0 0 0 0 0 0 0 0 0 0 ...
##   ..- attr(*, "names")= chr [1:66704] "1" "2" "3" "4" ...
##  $ ordered            : Named logi [1:11] FALSE FALSE FALSE FALSE FALSE FALSE ...
##   ..- attr(*, "names")= chr [1:11] "ORIG_CHN" "ORIG_RT" "ORIG_AMT" "ORIG_TRM" ...
##  - attr(*, "xlevels")=List of 3
##   ..$ ORIG_CHN: chr [1:3] "B" "C" "R"
##   ..$ OCC_STAT: chr [1:3] "I" "P" "S"
##   ..$ VinYr   : chr [1:18] "1999" "2000" "2001" "2002" ...
##  - attr(*, "class")= chr "rpart"
```

That is a lot of information. Alternatively, we could simply print the names of the objects in `cart.tree`.

```
# Or a less detailed view:
names(cart.tree)
```

```
##  [1] "frame"               "where"               "call"
##  [4] "terms"               "cptable"             "method"
##  [7] "parms"               "control"             "functions"
## [10] "numresp"             "splits"              "csplit"
## [13] "variable.importance" "y"                   "ordered"
```

```
# above is a list of the objects in the model object (cart.tree)
```

Now let's see how big the tree is by listing the number of splits. A list of splits is stored in `cart.tree$splits`.

```
# View the tree splits:
dim(cart.tree$splits)
```

```
## [1] 17496     5
```

There are 17,496 splits! This highlights the need for pruning and is a prime example of over-fitting. Let's print the first 20 splits.

```
# there are 12 splits, so print them all:
print(cart.tree$split[c(1:20),])
```

```
##           count ncat     improve       index         adj
## VinYr     66704   18 0.041421512      1.0000 0.0000000000
## CSCORE_MN 66519    1 0.032240793    679.5000 0.0000000000
## ORIG_RT   66704   -1 0.024042736      5.4995 0.0000000000
```

```
## DTI       65386   -1 0.011039656     45.5000 0.0000000000
## ORIG_TRM  66704   -1 0.004848382    341.0000 0.0000000000
## CSCORE_MN 50955    1 0.018242071    679.5000 0.0000000000
## VinYr     51114   18 0.010370260      2.0000 0.0000000000
## ORIG_RT   51114   -1 0.009633554      5.3625 0.0000000000
## ORIG_VAL  51114    1 0.005101081 130810.1473 0.0000000000
## ORIG_AMT  51114    1 0.002874539 105500.0000 0.0000000000
## ORIG_AMT    159    1 0.856755961  14500.0000 0.0005477201
## ORIG_RT       0   -1 0.856716711      9.8125 0.0002738601
## ORIG_VAL      0    1 0.856716711  17222.2222 0.0002738601
## VinYr     43811   18 0.005495357      3.0000 0.0000000000
## ORIG_RT   43811   -1 0.004094349      4.8700 0.0000000000
## CSCORE_MN 43652    1 0.003818525    737.5000 0.0000000000
## ORIG_VAL  43811    1 0.001741480 125342.7230 0.0000000000
## DTI       43154   -1 0.001628954     43.5000 0.0000000000
## CSCORE_MN 40825    1 0.002869262    737.5000 0.0000000000
## ORIG_RT   40964   -1 0.002668329      4.8700 0.0000000000
```

A few items to note from the above list:

- The variables that show up seem to make sense - credit score, vintage, interest rate, debt-to-income, etc.
- The index column gives the splitting value.
- 'ncat' gives the number of categories in the splitting variable, equal to +/-1 for continuous variables with the direction of the split given by the sign.

Print the variable importance scores.

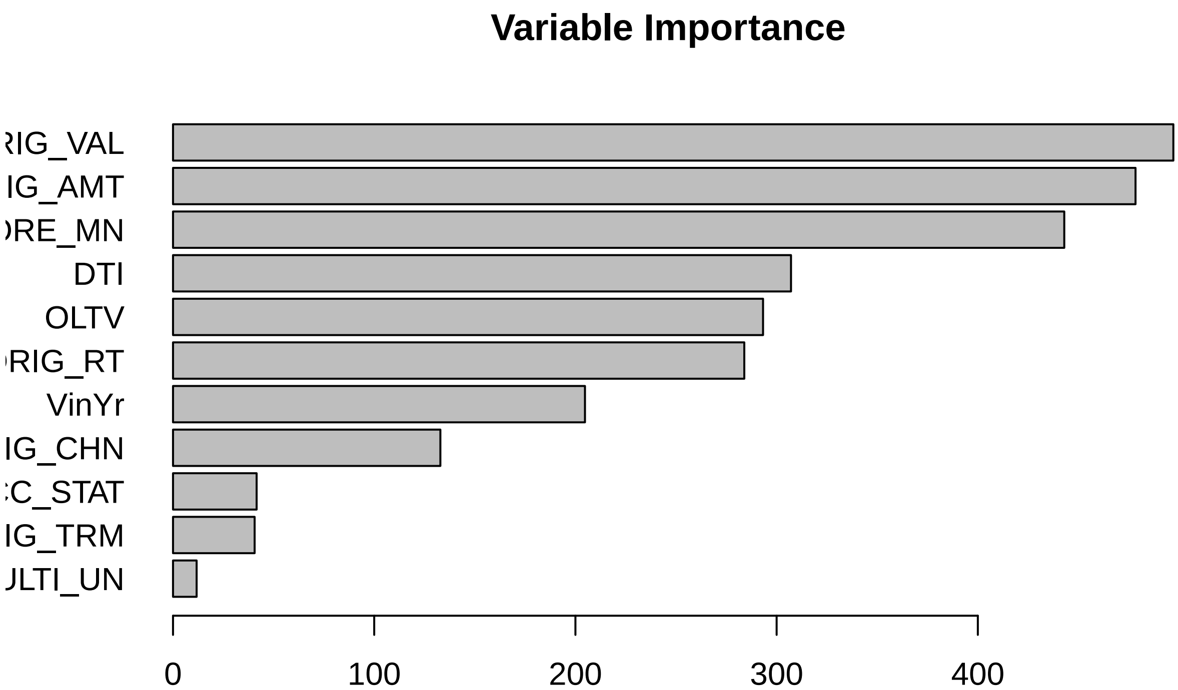
```
# Print the variable importance:
print(cart.tree$variable.importance)
```

```
##  ORIG_VAL  ORIG_AMT CSCORE_MN       DTI      OLTV   ORIG_RT     VinYr
## 497.17125 478.37037 442.95827 307.13892 293.25102 283.92725 204.74191
##  ORIG_CHN  OCC_STAT  ORIG_TRM  MULTI_UN
## 132.88458  41.55525  40.55269  11.73865
```

```
class(cart.tree$variable.importance)
```

```
## [1] "numeric"
```

```
barplot(rev(cart.tree$variable.importance),
        main="Variable Importance", horiz=T,
  names.arg=rev(labels(cart.tree$variable.importance)),
  las=1)
```




```
# Summarize the tree:
# summary(cart.tree)

# print(cart.tree)

# Save a graphical summary:
# png("cart.tree.png",width=1200, height=800)
# post(cart.tree, file="", title. = "Classifying Defaults")
#
# print(cart.tree$variable.importance)
```

3.2.1 Pruning

The above tree us unpruned and will overfit the data. We could instead prune the tree in an attempt to enhance out of sample (test) performance. Within the results, the object `.$cptable` tells us about the tree for different values of the tuning parameter `cp`.

```
dim(cart.tree$cptable)
```

```
## [1] 699   5
```

```
# Since the table is so long, print just the first 20 rows:
print(cart.tree$cptable[c(1:20),])
```

```
##              CP nsplit rel error    xerror       xstd
## 1  0.041421512      0 1.0000000 1.0000242 0.01972419
## 2  0.028586744      1 0.9585785 0.9594381 0.01822668
```

```
## 3  0.006288526      2 0.9299917 0.9335316 0.01756570
## 4  0.005761536      3 0.9237032 0.9282499 0.01736010
## 5  0.003450683      4 0.9179417 0.9224952 0.01711815
## 6  0.002479570      5 0.9144910 0.9206858 0.01708062
## 7  0.002436986      6 0.9120114 0.9192043 0.01700504
## 8  0.002026797      7 0.9095744 0.9172038 0.01695307
## 9  0.001965756      8 0.9075476 0.9151894 0.01687931
## 10 0.001884332      9 0.9055819 0.9140946 0.01685996
## 11 0.001820174     10 0.9036976 0.9142400 0.01686092
## 12 0.001807378     11 0.9018774 0.9137165 0.01685598
## 13 0.001410308     12 0.9000700 0.9136389 0.01685378
## 14 0.001303926     13 0.8986597 0.9138711 0.01685551
## 15 0.001290312     15 0.8960518 0.9147213 0.01685210
## 16 0.001168722     16 0.8947615 0.9140518 0.01683654
## 17 0.001166366     20 0.8900866 0.9145825 0.01683232
## 18 0.001155935     22 0.8877539 0.9146521 0.01683252
## 19 0.001150717     24 0.8854420 0.9150422 0.01683781
## 20 0.001058370     25 0.8842913 0.9151297 0.01683829
```

```
# And the bottom 20 rows:
tmp <- dim(cart.tree$cptable)
print(cart.tree$cptable[c((tmp[1]-19):tmp[1]),])
```

```
##                CP nsplit rel error   xerror       xstd
## 680 5.967263e-05   2458 0.2693386 1.582151 0.02401723
## 681 5.645248e-05   2468 0.2687419 1.582684 0.02401856
## 682 5.612401e-05   2472 0.2684900 1.582643 0.02401719
## 683 5.490392e-05   2473 0.2684339 1.582642 0.02401719
## 684 5.489051e-05   2475 0.2683241 1.583106 0.02401752
## 685 5.472091e-05   2477 0.2682143 1.583106 0.02401752
## 686 5.428387e-05   2479 0.2681049 1.582927 0.02401452
## 687 5.426867e-05   2481 0.2679963 1.582975 0.02401453
## 688 5.380961e-05   2483 0.2678878 1.582988 0.02401477
## 689 5.365702e-05   2485 0.2677801 1.582987 0.02401477
## 690 5.287044e-05   2487 0.2676728 1.582987 0.02401477
## 691 5.261081e-05   2489 0.2675671 1.582987 0.02401477
## 692 4.958443e-05   2504 0.2666696 1.583099 0.02401493
## 693 4.898641e-05   2507 0.2665209 1.583376 0.02401528
## 694 4.836082e-05   2510 0.2663739 1.583376 0.02401528
## 695 4.768706e-05   2513 0.2662288 1.583376 0.02401528
## 696 3.713468e-05   2516 0.2660858 1.583376 0.02401528
## 697 3.707586e-05   2519 0.2659744 1.583497 0.02401607
## 698 1.496640e-05   2522 0.2658632 1.583497 0.02401607
## 699 1.000000e-06   2526 0.2658033 1.583382 0.02401636
```

`rel-error` is the relative error is the ratio of $\sum_{m=1}^{|T|} Q_m(T)$ to a single root tree. Note it is always decreasing in `cp`. `xerror` is the 10-fold cross-validation error.

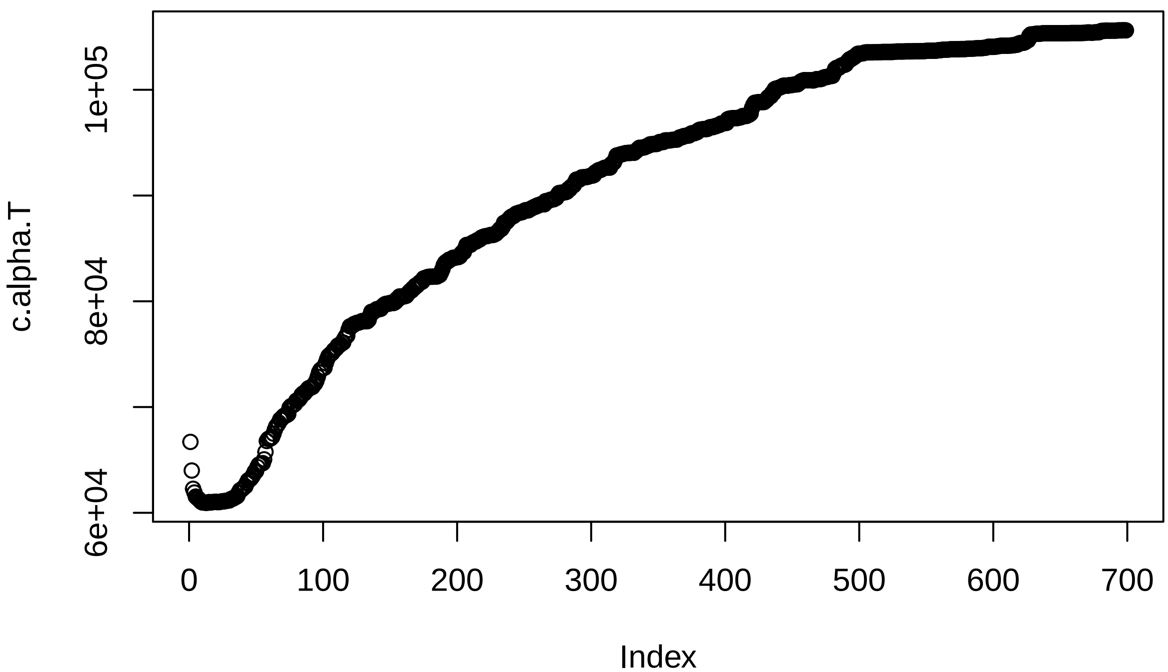
As discussed above, we want to choose `cp` (i.e., α in Eq. (8)) to maximize the cost-complexity criterion. We could either manually select a pruning level or select the one to maximize Eq. (8). To find the best tree size, we can use the information in `.$cptable` to define a metric to approximate $C_\alpha(T)$ and then choose the minimum value.

One such metric given in **Pekelis (2013)** which is as follows:

$$C_\alpha(T) \approx N \sum_{m=1}^{|T|} Q_m(T) + \alpha|T|, \tag{9}$$

where $\sum_{m=1}^{|T|} Q_m(T)$ is the relative error in `.$cptable`, N is the number of trianing instances, and $|T|$ is the total number of terminal nodes - or two times the number of splits. However, rather than using the relative error, we probably should consider cross-validated error, or `.$xerror`.

```
# Estimate c_alpha(T):
N <- dim(df.model)[1]
c.alpha.T <- N * cart.tree$cptable[,4] +
              (cart.tree$cptable[,2] + 1)*2*cart.tree$cptable[,1]
plot(c.alpha.T)
```




```
# Choose the minimum of c.alpha.T:
min.C.alpha.T <- cart.tree$cptable[which.min(c.alpha.T),2]
message(sprintf("The best tree size has %d splits",min.C.alpha.T))
```

```
## The best tree size has 12 splits
```

We could select and save that tree as a pruned tree as follows.

```
# Save the pruned tree
cart.tree.best <- prune(cart.tree,
                        cart.tree$cptable[which.min(c.alpha.T),1])
```

Finally, we could print the results:

```r
print(cart.tree.best)
```

```
## n= 66704
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##   1) root 66704 2227.21100 0.034585630
##     2) VinYr=1999,2000,2001,2002,2003,2004,2009,2010,2011,2012,2013,2014,2015,2016 51114  707.91420 0
##       4) CSCORE_MN>=679.5 43811  329.48410 0.007578006 *
##       5) CSCORE_MN< 679.5 7303  365.59800 0.052854990
##        10) VinYr=1999,2000,2001,2002,2010,2011,2012,2013,2014,2015,2016 5292  183.17840 0.035903250 *
##        11) VinYr=2003,2004,2009 2011  176.89710 0.097463950 *
##     3) VinYr=2005,2006,2007,2008 15590 1427.04200 0.101924300
##       6) CSCORE_MN>=674.5 12100  762.70050 0.067603310
##        12) CSCORE_MN>=738.5 7137  269.01500 0.039232170 *
##        13) CSCORE_MN< 738.5 4963  479.67960 0.108402200
##          26) OLTV< 59.5 945   36.47196 0.040211640 *
##          27) OLTV>=59.5 4018  437.78000 0.124440000
##            54) DTI< 36.5 1607  123.65900 0.084007470 *
##            55) DTI>=36.5 2411  309.74280 0.151389500 *
##       7) CSCORE_MN< 674.5 3490  600.67310 0.220916900
##        14) VinYr=2005,2006,2008 2461  379.62130 0.190572900
##          28) OLTV< 48.5 244   14.07787 0.061475410 *
##          29) OLTV>=48.5 2217  361.02930 0.204781200
##            58) CSCORE_MN>=650.5 956  125.08790 0.154811700 *
##            59) CSCORE_MN< 650.5 1261  231.74460 0.242664600
##             118) ORIG_RT< 6.275 650   99.10154 0.187692300 *
##             119) ORIG_RT>=6.275 611  128.58920 0.301145700 *
##        15) VinYr=2007 1029  213.36640 0.293488800
##          30) ORIG_TRM< 324 122   13.15574 0.122950800 *
##          31) ORIG_TRM>=324 907  196.18520 0.316427800 *
```

Or summarize the tree:

```r
summary(cart.tree.best)
```

```
## Call:
## rpart(formula = def.max ~ ., data = df.model, minsplit = 5, cp = 1e-06)
##   n= 66704
##
##              CP nsplit rel error    xerror       xstd
## 1  0.041421512      0 1.0000000 1.0000242 0.01972419
## 2  0.028586744      1 0.9585785 0.9594381 0.01822668
## 3  0.006288526      2 0.9299917 0.9335316 0.01756570
## 4  0.005761536      3 0.9237032 0.9282499 0.01736010
## 5  0.003450683      4 0.9179417 0.9224952 0.01711815
## 6  0.002479570      5 0.9144910 0.9206858 0.01708062
## 7  0.002436986      6 0.9120114 0.9192043 0.01700504
## 8  0.002026797      7 0.9095744 0.9172038 0.01695307
## 9  0.001965756      8 0.9075476 0.9151894 0.01687931
## 10 0.001884332      9 0.9055819 0.9140946 0.01685996
## 11 0.001820174     10 0.9036976 0.9142400 0.01686092
## 12 0.001807378     11 0.9018774 0.9137165 0.01685598
## 13 0.001410308     12 0.9000700 0.9136389 0.01685378
```

```
##
## Variable importance
##     VinYr CSCORE_MN      OLTV       DTI   ORIG_RT  ORIG_TRM  ORIG_VAL
##        47        42         5         2         2         2         1
##
## Node number 1: 66704 observations,    complexity param=0.04142151
##   mean=0.03458563, MSE=0.03338947
##   left son=2 (51114 obs) right son=3 (15590 obs)
##   Primary splits:
##       VinYr     splits as  LLLLLLRRRRLLLLLLLL, improve=0.041421510, (0 missing)
##       CSCORE_MN < 679.5    to the right, improve=0.032240790, (185 missing)
##       ORIG_RT   < 5.4995   to the left,  improve=0.024042740, (0 missing)
##       DTI       < 45.5     to the left,  improve=0.011039660, (1318 missing)
##       ORIG_TRM  < 341      to the left,  improve=0.004848382, (0 missing)
##
## Node number 2: 51114 observations,    complexity param=0.005761536
##   mean=0.01404703, MSE=0.01384971
##   left son=4 (43811 obs) right son=5 (7303 obs)
##   Primary splits:
##       CSCORE_MN < 679.5    to the right, improve=0.018242070, (159 missing)
##       VinYr     splits as  LLLLRR----LLLLLLLL, improve=0.010370260, (0 missing)
##       ORIG_RT   < 5.3625   to the left,  improve=0.009633554, (0 missing)
##       ORIG_VAL  < 130810.1 to the right, improve=0.005101081, (0 missing)
##       ORIG_AMT  < 105500   to the right, improve=0.002874539, (0 missing)
##   Surrogate splits:
##       ORIG_AMT < 14500    to the right, agree=0.857, adj=0.001, (159 split)
##       ORIG_RT  < 9.8125   to the left,  agree=0.857, adj=0.000, (0 split)
##       ORIG_VAL < 17222.22 to the right, agree=0.857, adj=0.000, (0 split)
##
## Node number 3: 15590 observations,    complexity param=0.02858674
##   mean=0.1019243, MSE=0.09153575
##   left son=6 (12100 obs) right son=7 (3490 obs)
##   Primary splits:
##       CSCORE_MN < 674.5    to the right, improve=0.044557860, (26 missing)
##       ORIG_RT   < 6.495    to the left,  improve=0.013452170, (0 missing)
##       DTI       < 36.5     to the left,  improve=0.011736980, (451 missing)
##       OLTV      < 58.5     to the left,  improve=0.010916620, (0 missing)
##       ORIG_TRM  < 359.5    to the left,  improve=0.009009478, (0 missing)
##   Surrogate splits:
##       ORIG_RT < 7.6375   to the left,  agree=0.776, adj=0.001, (26 split)
##
## Node number 4: 43811 observations
##   mean=0.007578006, MSE=0.007520579
##
## Node number 5: 7303 observations,    complexity param=0.00247957
##   mean=0.05285499, MSE=0.05006134
##   left son=10 (5292 obs) right son=11 (2011 obs)
##   Primary splits:
##       VinYr    splits as  LLLLRR----RLLLLLLL, improve=0.015105460, (0 missing)
##       ORIG_VAL < 83670.01 to the right, improve=0.010251490, (0 missing)
##       ORIG_RT  < 4.9325   to the left,  improve=0.009905408, (0 missing)
##       ORIG_AMT < 76500    to the right, improve=0.005531550, (0 missing)
##       OLTV     < 80.5     to the left,  improve=0.004365182, (0 missing)
##
```

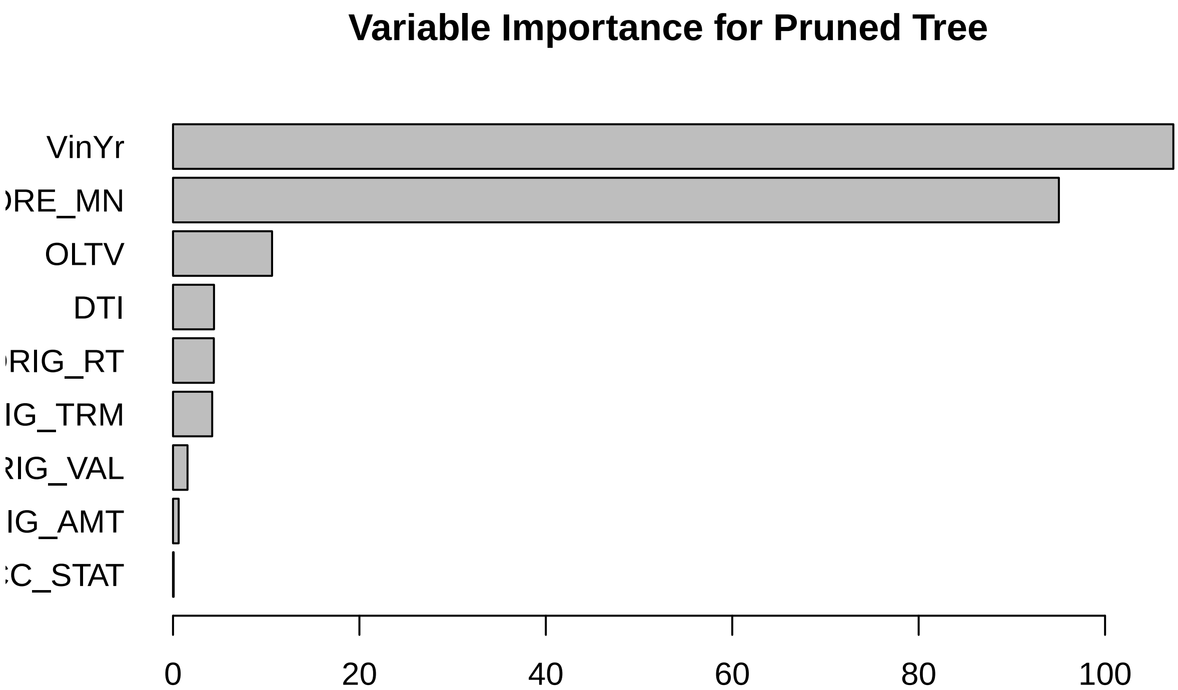
```
## Node number 6: 12100 observations,    complexity param=0.006288526
##   mean=0.06760331, MSE=0.0630331
##   left son=12 (7137 obs) right son=13 (4963 obs)
##   Primary splits:
##       CSCORE_MN < 738.5    to the right, improve=0.018401570, (26 missing)
##       OLTV      < 59.5     to the left,  improve=0.011463420, (0 missing)
##       DTI       < 36.5     to the left,  improve=0.011367100, (369 missing)
##       ORIG_RT   < 6.3875   to the left,  improve=0.007963315, (0 missing)
##       ORIG_TRM  < 359      to the left,  improve=0.006969895, (0 missing)
##   Surrogate splits:
##       OLTV     < 81.5     to the left,  agree=0.595, adj=0.015, (26 split)
##       ORIG_RT  < 6.87     to the left,  agree=0.593, adj=0.008, (0 split)
##       ORIG_AMT < 21500    to the right, agree=0.590, adj=0.001, (0 split)
##       ORIG_VAL < 38194.44 to the right, agree=0.590, adj=0.000, (0 split)
##
## Node number 7: 3490 observations,    complexity param=0.003450683
##   mean=0.2209169, MSE=0.1721126
##   left son=14 (2461 obs) right son=15 (1029 obs)
##   Primary splits:
##       VinYr    splits as  ------LLRL--------, improve=0.012794640, (0 missing)
##       ORIG_TRM < 359.5    to the left,  improve=0.012223890, (0 missing)
##       OLTV     < 46.5     to the left,  improve=0.011078980, (0 missing)
##       ORIG_RT  < 6.275    to the left,  improve=0.009441140, (0 missing)
##       DTI      < 37.5     to the left,  improve=0.008225135, (82 missing)
##   Surrogate splits:
##       ORIG_RT < 8.1875   to the left,  agree=0.706, adj=0.003, (0 split)
##       OLTV    < 11.5     to the right, agree=0.706, adj=0.003, (0 split)
##
## Node number 10: 5292 observations
##   mean=0.03590325, MSE=0.03461421
##
## Node number 11: 2011 observations
##   mean=0.09746395, MSE=0.08796473
##
## Node number 12: 7137 observations
##   mean=0.03923217, MSE=0.03769301
##
## Node number 13: 4963 observations,    complexity param=0.002436986
##   mean=0.1084022, MSE=0.09665114
##   left son=26 (945 obs) right son=27 (4018 obs)
##   Primary splits:
##       OLTV     < 59.5     to the left,  improve=0.011315220, (0 missing)
##       ORIG_TRM < 210      to the left,  improve=0.010047790, (0 missing)
##       DTI      < 31.5     to the left,  improve=0.009652204, (157 missing)
##       ORIG_RT  < 6.3725   to the left,  improve=0.009274667, (0 missing)
##       ORIG_AMT < 170500   to the left,  improve=0.004209679, (0 missing)
##   Surrogate splits:
##       ORIG_VAL < 642197.8 to the right, agree=0.834, adj=0.129, (0 split)
##       ORIG_TRM < 150      to the left,  agree=0.816, adj=0.033, (0 split)
##       ORIG_AMT < 22500    to the left,  agree=0.811, adj=0.006, (0 split)
##       ORIG_RT  < 4.6875   to the left,  agree=0.810, adj=0.003, (0 split)
##
## Node number 14: 2461 observations,    complexity param=0.002026797
##   mean=0.1905729, MSE=0.1542549
```

```
##   left son=28 (244 obs) right son=29 (2217 obs)
##   Primary splits:
##       OLTV      < 48.5     to the left,  improve=0.011891070, (0 missing)
##       CSCORE_MN < 650.5    to the right, improve=0.009860653, (0 missing)
##       ORIG_RT   < 6.275    to the left,  improve=0.008783505, (0 missing)
##       ORIG_TRM  < 359.5    to the left,  improve=0.008657372, (0 missing)
##       DTI       < 31.5     to the left,  improve=0.008277598, (57 missing)
##   Surrogate splits:
##       ORIG_VAL < 746134.8 to the right, agree=0.908, adj=0.074, (0 split)
##       ORIG_AMT < 22500    to the left,  agree=0.902, adj=0.008, (0 split)
##       ORIG_RT  < 4.8125   to the left,  agree=0.901, adj=0.004, (0 split)
##
## Node number 15: 1029 observations,    complexity param=0.001807378
##   mean=0.2934888, MSE=0.2073531
##   left son=30 (122 obs) right son=31 (907 obs)
##   Primary splits:
##       ORIG_TRM  < 324      to the left,  improve=0.01886619, (0 missing)
##       ORIG_AMT  < 359000   to the left,  improve=0.01488409, (0 missing)
##       CSCORE_MN < 623.5    to the right, improve=0.01470734, (0 missing)
##       ORIG_RT   < 7.0625   to the left,  improve=0.01205791, (0 missing)
##       OLTV      < 54.5     to the left,  improve=0.01099618, (0 missing)
##   Surrogate splits:
##       OLTV < 25.5     to the left,  agree=0.884, adj=0.025, (0 split)
##
## Node number 26: 945 observations
##   mean=0.04021164, MSE=0.03859466
##
## Node number 27: 4018 observations,    complexity param=0.001965756
##   mean=0.12444, MSE=0.1089547
##   left son=54 (1607 obs) right son=55 (2411 obs)
##   Primary splits:
##       DTI      < 36.5     to the left,  improve=0.010050940, (123 missing)
##       ORIG_RT  < 6.495    to the left,  improve=0.008277172, (0 missing)
##       ORIG_TRM < 359      to the left,  improve=0.007767584, (0 missing)
##       ORIG_CHN splits as  RLL,          improve=0.004813548, (0 missing)
##       ORIG_AMT < 203500   to the left,  improve=0.003915681, (0 missing)
##   Surrogate splits:
##       ORIG_AMT < 120500   to the left,  agree=0.606, adj=0.030, (123 split)
##       ORIG_VAL < 132466.2 to the left,  agree=0.597, adj=0.009, (0 split)
##       ORIG_RT  < 5.05     to the left,  agree=0.595, adj=0.003, (0 split)
##
## Node number 28: 244 observations
##   mean=0.06147541, MSE=0.05769618
##
## Node number 29: 2217 observations,    complexity param=0.001884332
##   mean=0.2047812, MSE=0.1628459
##   left son=58 (956 obs) right son=59 (1261 obs)
##   Primary splits:
##       CSCORE_MN < 650.5    to the right, improve=0.011624560, (0 missing)
##       ORIG_RT   < 6.275    to the left,  improve=0.009902450, (0 missing)
##       ORIG_TRM  < 359.5    to the left,  improve=0.007017941, (0 missing)
##       DTI       < 31.5     to the left,  improve=0.007014271, (50 missing)
##       ORIG_CHN  splits as  RRL,          improve=0.005404706, (0 missing)
##   Surrogate splits:
```

```
##       OCC_STAT splits as  LRR,          agree=0.576, adj=0.017, (0 split)
##       ORIG_RT  < 5.1875   to the left,  agree=0.573, adj=0.010, (0 split)
##       OLTV     < 92.5     to the right, agree=0.571, adj=0.005, (0 split)
##       ORIG_AMT < 457500   to the right, agree=0.570, adj=0.002, (0 split)
##       ORIG_VAL < 825823.5 to the right, agree=0.570, adj=0.002, (0 split)
##
## Node number 30: 122 observations
##   mean=0.1229508, MSE=0.1078339
##
## Node number 31: 907 observations
##   mean=0.3164278, MSE=0.2163012
##
## Node number 54: 1607 observations
##   mean=0.08400747, MSE=0.07695021
##
## Node number 55: 2411 observations
##   mean=0.1513895, MSE=0.1284707
##
## Node number 58: 956 observations
##   mean=0.1548117, MSE=0.130845
##
## Node number 59: 1261 observations,    complexity param=0.001820174
##   mean=0.2426646, MSE=0.1837785
##   left son=118 (650 obs) right son=119 (611 obs)
##   Primary splits:
##       ORIG_RT  < 6.275    to the left,  improve=0.017493010, (0 missing)
##       DTI      < 37.5     to the left,  improve=0.013619990, (30 missing)
##       VinYr    splits as  ------LR-R--------, improve=0.009213111, (0 missing)
##       ORIG_TRM < 359.5    to the left,  improve=0.007852151, (0 missing)
##       ORIG_CHN splits as  RLL, improve=0.006964813, (0 missing)
##   Surrogate splits:
##       VinYr     splits as  ------LR-R--------, agree=0.738, adj=0.460, (0 split)
##       ORIG_VAL  < 170557.3 to the right, agree=0.571, adj=0.115, (0 split)
##       ORIG_AMT  < 94500    to the right, agree=0.560, adj=0.092, (0 split)
##       OLTV      < 79.5     to the left,  agree=0.556, adj=0.083, (0 split)
##       CSCORE_MN < 615.5    to the right, agree=0.553, adj=0.077, (0 split)
##
## Node number 118: 650 observations
##   mean=0.1876923, MSE=0.1524639
##
## Node number 119: 611 observations
##   mean=0.3011457, MSE=0.210457
```

And finally, print the relevant variable importance scores for the optimized tree.

```r
barplot(rev(cart.tree.best$variable.importance),
        main="Variable Importance for Pruned Tree", horiz=T,
  names.arg=rev(labels(cart.tree.best$variable.importance)),
  las=1)
```



3.3 Example 2 - Mortgage Data using C4.5 and C5.0

The above example can also be carried out using the C5.0 algorithm in R. The relevant package is the `C50` package available **here** with documentation **here**. We will use the same data as the above tree. The `C5.0()` algorithm can be called from within the `caret` package (see **this link** for a nice help file). A similar approach approach is the C4.5 algorithm which is actually implemented using the `caret` wrapper by calling the underlying Weka funciton which is written in Java (so it doesn't work on the grid or OCC laptops…as of the last time I tested it).

The `C5.0()` function can handle the input data to be specified either as i) a matrix or data.frame of features, `x`, and a vector of outcomes `y` or ii) using a formula similar to `rpart`.

```
library("C50")
library("RWeka")
library("caret")
library("mlbench")

set.seed(100)

# Select all except def.max
x <- subset(df.model, select=c(-def.max,-VinYr))
y <- as.factor(df.model$def.max)
```

Start by fitting a tree without cross-validation to speed things up. The bagging, boosting, CV, etc. are set in `trainControl()`. Other options will be discussed in the sections below.

```r
# First turn the stings into factors:
x$ORIG_CHN <- as.factor(x$ORIG_CHN)
x$OCC_STAT <- as.factor(x$OCC_STAT)
# x$VinYr <- as.factor(x$VinYr)

# Set the control parameters to shut off CV:
fitControl <- trainControl(method = "none",number=1)

# Set the tuning parameters:
grid <- expand.grid(.C = 0.5,
                    .M = 2)

metric <- "Kappa"

# Fit the C4.5 Model (called J48)
J48.tree <- train(x=x,y=y,
                  tuneGrid = grid,
                  trControl = fitControl,
                  method="J48",
                  metric = metric,
                  maximize=TRUE,
                  na.action=NULL)
```

```
## Warning: Setting row names on a tibble is deprecated.
```

```r
# Summarize the results:
summary(J48.tree)
```

```
##
## === Summary ===
##
## Correctly Classified Instances       64488               96.6779 %
## Incorrectly Classified Instances      2216                3.3221 %
## Kappa statistic                          0.0842
## Mean absolute error                      0.0592
## Root mean squared error                  0.172
## Relative absolute error                 88.6445 %
## Root relative squared error             94.1225 %
## Total Number of Instances            66704
##
## === Confusion Matrix ===
##
##      a     b   <-- classified as
##  64382    15 |     a = 0
##   2201   106 |     b = 1
```

Note that the tuning confidence parameter (`C=0.5`) is set to a relatively high value for illustrative purposes (if it is set to a usual value around 0.25, the tree is empty). Also note that `metric = "Kappa"` so that the objective is to maximize (note `maximize=TRUE`) the Kappa statistic. Also note that `na.action=NULL` means that the model incorporates missing values.

One issue that arises and is contributing to the realtively poor model fit is that we have an unbalanced response variable - very few defaults. The `caret` package has two built in sampling methods to handle this:

1. Up sampling. Up sampling bootstraps with replacement the less frequent of the two classes (in this case defaults) so that the number of observations of each class is equivalent and equal to $[N_{Class_1}, N_{Class_2}]^+$.

2. Down sampling. Down sampling smaples the more frequent of the two classes (in this case non-defaults) so that the number of observations of each class is equivalent and equal to $[N_{Class_1}, N_{Class_2}]^-$.

The two functions are `downSample()` and `upSample()`. Note that the response will be stored in a new variable called `.$Class`.

```
# Tabulate the response variable
table(y)
```

```
## y
##     0     1
## 64397  2307
```

```
# Up and down sampling examples:
down_train <- downSample(x = x[, -ncol(x)],
                         y = y)
table(down_train$Class)
```

```
##
##    0    1
## 2307 2307
```

```
# Up-sampling
up_train <- upSample(x = x[, -ncol(x)],
                     y = y)
table(up_train$Class)
```

```
##
##     0     1
## 64397 64397
```

```
J48.tree.dn <- train(Class ~ ., data  = down_train,
                  tuneGrid = grid,
                  trControl = fitControl,
                  method="J48",
                  metric = metric,
                  maximize=TRUE,
                  na.action=NULL)
summary(J48.tree.dn)
```

```
##
## === Summary ===
##
## Correctly Classified Instances        3797               85.2301 %
## Incorrectly Classified Instances       658               14.7699 %
## Kappa statistic                          0.7047
## Mean absolute error                      0.2192
## Root mean squared error                  0.331
## Relative absolute error                 43.8356 %
## Root relative squared error             66.2085 %
## Total Number of Instances             4455
##
## === Confusion Matrix ===
##
```

```
##      a    b   <-- classified as
##   1836  399 |    a = 0
##    259 1961 |    b = 1
```

```r
J48.tree.up <- train(Class ~ ., data  = up_train,
                  tuneGrid = grid,
                  trControl = fitControl,
                  method="J48",
                  metric = metric,
                  maximize=TRUE,
                  na.action=NULL)
summary(J48.tree.up)
```

```
##
## === Summary ===
##
## Correctly Classified Instances      124526               99.6407 %
## Incorrectly Classified Instances       449                0.3593 %
## Kappa statistic                          0.9928
## Mean absolute error                      0.007
## Root mean squared error                  0.0592
## Relative absolute error                  1.4004 %
## Root relative squared error             11.834  %
## Total Number of Instances           124975
##
## === Confusion Matrix ===
##
##      a     b   <-- classified as
##  62543   449 |     a = 0
##      0 61983 |     b = 1
```

```r
# J48.tree.up$finalModel
```

Note that the models predict much better - at least in sample. This is to be expected but likely overstates the model performance - especially when tested out of sample. For example, consider the up-sampling. We have on average $r_{default}^{-1}$ observations of each loan in the model. As such the features associated with those defaults will be overly important.

4 Model Enhancements for Decision Trees

Like many other algorithms, decision trees can be improved using several techniques are common across ML applications. There are three main enhancements which are as follows (see Chapter 8 of James et al. (2013)):

1. Bagging
2. Random Forests
3. Boosting

In this section, we will demonstrate each of these features, which are readily accesible using the `caret` package.

4.1 Bagging

Bagging is an application of bootstrapping. It involves selecting random samples and averaging the results. The basic idea stems from the Central Limit Theorem whereby variance of the mean of a set of observations decreases linearly with the number of observations n being averaged. In the context of decision tree models, we can train B trees $(\hat{f}^1(x), \hat{f}^2(x), ..., \hat{f}^B(x))$ using B separate training sets. In reality, we don't have B training sets so we repeatedly sample from a single training set and then average the predictions to get a single output,

$$\hat{f_{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f^{*b}}(x). \tag{10}$$

For the case of decision trees, the aggregation process depends on the nature of the problem. For the case of regression trees, the predictions for each instance can be the average of the predicted values across the bagged trees. For classification problems, a different rule can be applied. The most simplistic rule would be to take the majority vote of across the models (e.g., classify based on the most frequently predicted class). More advanced rules - possibly probabilistic could also be used.

Of course a disadvantage of bagging is that there are now B trees so the nice interpretability feature of the goes away. However, we can still compute the variable importance scores for each feature. For regression trees, one could compute the change in the residual sum of squares. For classification trees, we could compute the drop in the Gini index (CART) or entropy information gain (C5.0) which would occur if we were to omit a given variable.

4.1.1 Bagging Example using CART Trees

The `caret` package has several bagging routines. A great resource for the `caret` package can be found **here**. For illustartive purposes, we will use the CART algorithm which has an associated bagging routine called that can be called using the `method = 'bag'`. The `bagControl` function contains options to control the method.

```
# Set the control parameters to shut off CV:
fitControl <- trainControl(method = "none",
                           repeats = 1)
```

```
## Warning: `repeats` has no meaning for this resampling method.
```

```
# Set the tuning parameters:
grid <- expand.grid(.cp = 0.001)

metric <- "Kappa"

bagControl <- bagControl(fit = train(x=x,y=y,
                  trControl = fitControl,
                  method="rpart",
                  metric = metric,
                  maximize=TRUE,
                  na.action=NULL),
                  predict=predict)
```

```
## Warning: Setting row names on a tibble is deprecated.
```

```
# Fit the C4.5 Model (called J48)
cart.bag <- train(x=x,y=y,
                  trControl = fitControl,
                  method="rpart",
```

```
                   metric = metric,
                   maximize=TRUE,
                   na.action=NULL)
```

```
## Warning: Setting row names on a tibble is deprecated.
```

```
# Summarize the results:
print(cart.bag)
```

```
## CART
##
## 66704 samples
##    10 predictor
##     2 classes: '0', '1'
##
## No pre-processing
## Resampling: None
```

```
# Test the model:
cart.bag.pred <- predict(cart.bag,x)
p.vs.a <- data.frame(predicted=cart.bag.pred,actual=y)
head(p.vs.a,20)
```

```
##    predicted actual
## 1          0      0
## 2          0      0
## 3          0      0
## 4          0      0
## 5          0      0
## 6          0      0
## 7          0      0
## 8          0      0
## 9          0      0
## 10         0      0
## 11         0      0
## 12         0      0
## 13         0      0
## 14         0      0
## 15         0      0
## 16         0      0
## 17         0      0
## 18         0      0
## 19         0      0
## 20         0      0
```

```
dim(p.vs.a)
```

```
## [1] 66704     2
```

```
num.1 <- length(which(p.vs.a$predicted == p.vs.a$actual))
num.0 <- length(which(p.vs.a$predicted != p.vs.a$actual))
```

4.2 Random Forests

Random forests are similar to bagging with one key difference that has the intention of reducing the correlation between the trees. Random forests still work on randomly drawn samples of the training data. The difference from bagging is that for each node, only a subsample of the p features are considered as potential splitting variables. James et al. (2013) suggests that a typical value of parameters is $m \approx \sqrt{p}$. If we set $m = p$ then it is equivalent to bagging. The reason that random forests work is that the trees tend to look very different as compared to bagged trees. Random forests can be applied to regression or classification to different tree algorithms (e.g., CART of C5.0).

4.3 Boosting

Boosting is a third enhancement to decision tree models. The main difference between boosting and bagging is that boosted trees are build sequentially. The boosted trees are not build upon bootstrapped samples, rather they are build on a modified version of the original data. The method works by inferring information on the residuals at each step and using this information to improve the next iteration.

A basic algorithm is given on page 323 of James et al. (2013):

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all $i \in X$.

2. For $b = 1, 2, ..., B$, repeat:
   (a) Fit a tree $\hat{f}^b(x)$ with d splits to the training data (X, r).
   (b) Update $\hat{f}$ by adding in a shrunken version of the new tree:
   $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x). \tag{11}$$
   (c) Update the residuals,
   $$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i). \tag{12}$$

3. Output the boosted model,
   $$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x). \tag{13}$$

There are three tuning parameters in terms fo boosting decision trees:

1. The number of trees, B, which can be chosen using cross-validation. Boosting can overfit the data is B is too large.

2. The shrinkage parameter λ which controls the learning rate. Smaller values mean the algorithm learns slower because each sequential tree $\hat{f}^b(x)$ is less influential (see step 2 of the above algorithm).

3. The number of splits per tree, d. Typically a small value works best.

References

Breiman, Leo, Jerome H Friedman, Richard A Olshen, and Charles J Stone. 1984. "Classification and Regression Trees. Belmont, ca: Wadsworth." *International Group*, 432.

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. 2001. *The Elements of Statistical Learning*. Vol. 1. 10. Springer series in statistics New York.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An Introduction to Statistical Learning*. Vol. 112. Springer.

Ross Quinlan, J. 1993. "C4. 5: Programs for Machine Learning." *Mach. Learn* 16 (3): 235–40.