Working with R

A University of Queensland Advanced Workshop

Session 9:

Machine Learning Approaches

Bill Venables, CSIRO/Data61, Dutton Park

Rhetta Chappell, Griffith University

2–5 February, 2021

Contents

1 The Swiss credit card data set

For this section, we use one data set which we prepare from original sources, namely the file `creditCard.csv`.

- Response: *credit.card.owner*, binary with levels *c("no","yes")*.
- Predictors: mostly numeric, to do with banking activity.
  - *profession* is a sparse factor which we re-group into three densely populated levels (as factor *pclass*).
  - *nationality* which we re-group into two levels, *"Swiss"* and *"Foreigner"*.
  - *sex* is the only other factor.
- Entries have a numerical *ident* which we zero-fill and move to the *rownames* rather than have it as a potentially active variable.

Read in the data and

```
library(WWRCourse)
CCf <- system.file("extdata", "creditCard.csv", package = "WWRCourse")
CC <- read.csv(CCf, na.strings="", stringsAsFactors = TRUE)
row.names(CC) <- fill0(CC$ident)

with(CC, c(cases = nrow(CC), known = sum(!is.na(credit.card.owner))))

cases known
10893  2085
```

The data is mostly *unknown*: for building predictive models we only have two thousand and eighty-five *at most* from ten thousand eight hundred and ninety-three cases.

```r
CC <- CC %>% within({
  p2 <- c("doctor", "engineer", "lawyer", "professor", "business")
  p1 <- c("teacher", "police", "service", "chemist", "nurse",
          "postman", "physical")
  p0 <- "none"
  pclass <- rep("", length(profession))
  pclass[profession %in% p0] <- "p0"
  pclass[profession %in% p1] <- "p1"
  pclass[profession %in% p2] <- "p2"
  pclass <- factor(pclass, levels = paste0("p", 0:2))
  swiss <- ifelse(nationality == "CH", "Swiss", "Foreigner")
  sex <- factor(sex)
  swiss <- factor(swiss)
  pclass <- factor(pclass)
  ident <- profession <- nationality <- p0 <- p1 <- p2 <- NULL
})
CCKnown <- na.omit(CC)
dim(CCKnown)
```

```
[1] 1620   55
```

For model building we have just one thousand six hundred and twenty cases and fifty-four potential predictors.

For demonstration purposes we split these into *Training* and `Test` data sets.

```
set.seed(20200202)    ## for reproducibility
train <- sample(nrow(CCKnown), nrow(CCKnown)/2)

CCTrain <- CCKnown[train, ]
CCTest <- CCKnown[-train, ]
Store(CC, CCKnown, CCTrain, CCTest, train, remove = FALSE)  ## for saftey
```

2 Trees and forests

- A technique that developed in machine learning and now widely used in data mining.
- The model uses *recursive partitioning* of the data and is a greedy algorithm.
- The two main types of tree models are
  - ***Regression trees*** — response is a continuous variable and fitting uses a least squares criterion,
  - ***Classification trees*** — response is a factor variable and fitting uses an entropy (multinomial likelihood) criterion.

- Model fitting is easy. Inference poses more of a dilemma.
- The tree structure is very unstable. *boosting* and *bagging* (random forests) can be useful ways around this.
- Two packages for tree models: `rpart` (which is part of **R** itself) and the older `tree`, (Ripley., 2012), which has an **S-PLUS** flavour and a few advantages for teaching.

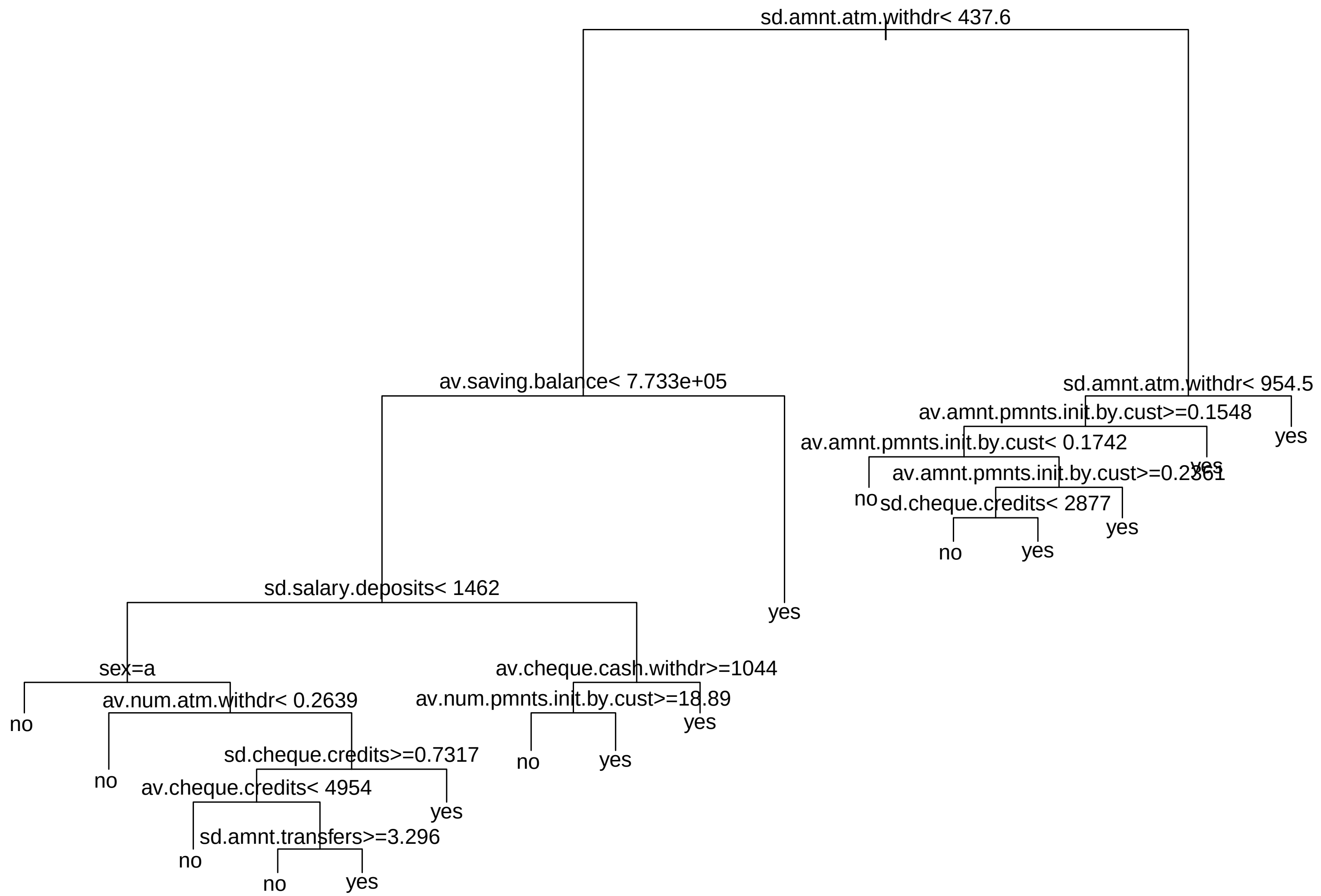
  Use `rpart` in practice.

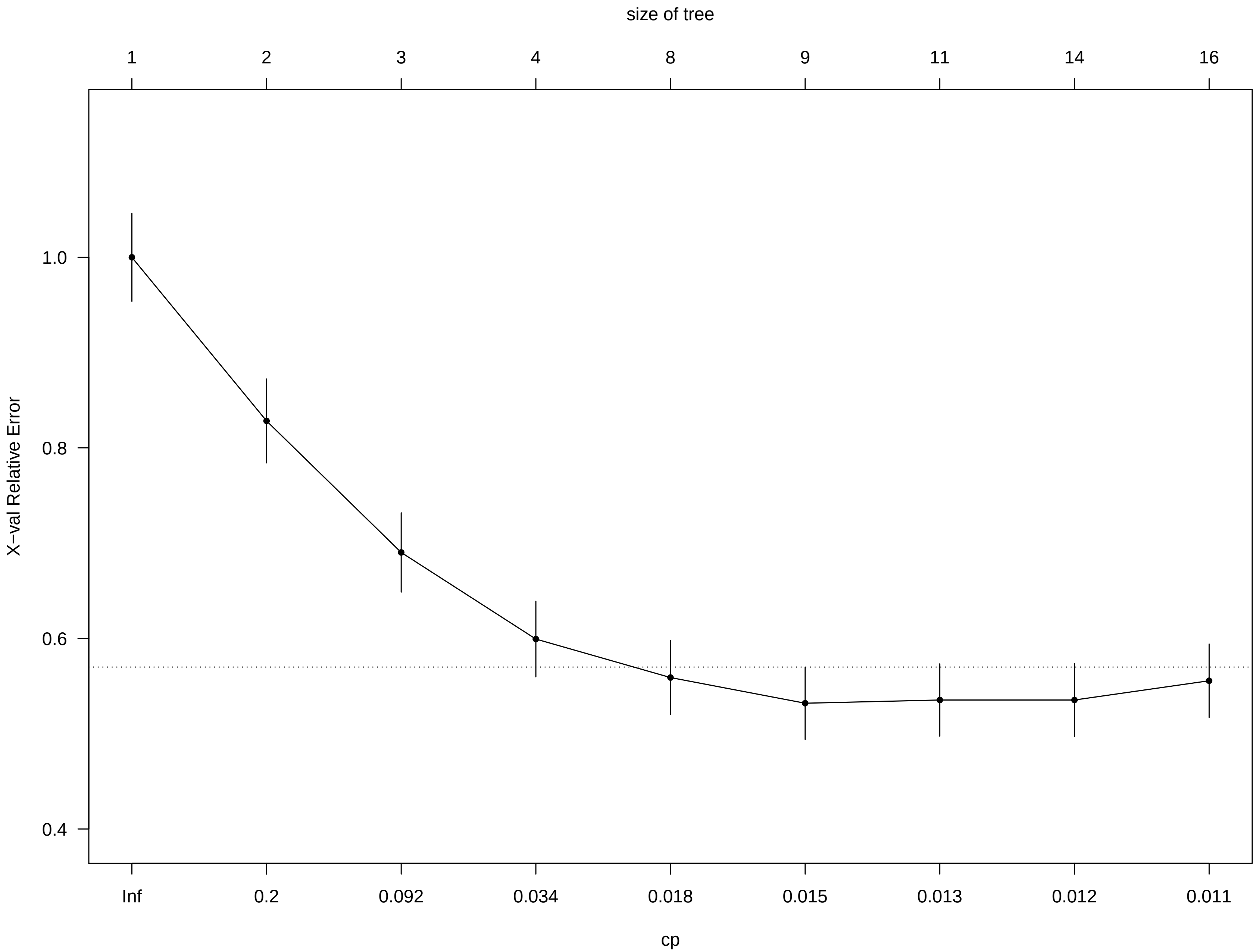
3 Do you want a credit card?

Our plan is to build predictive models on the *Training* set, and use the *Test* set to see how well they fared. We employ a number of techniques, mostly of the "black box" kind.

3.1 An initial tree model

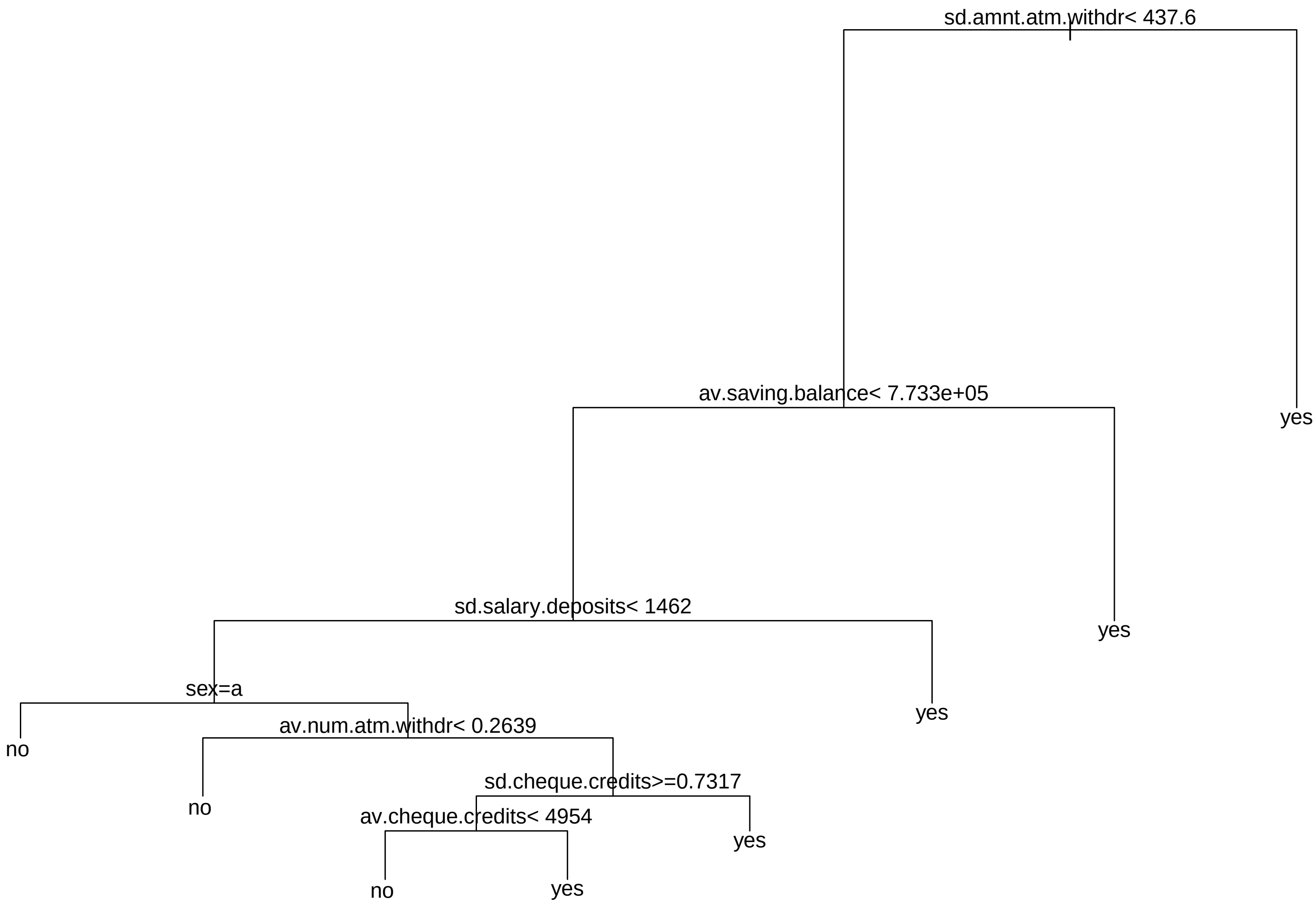
```
requireData(rpart)
CCTree <- rpart(credit.card.owner ~ ., CCTrain)
Store(CCTree, remove = FALSE)

## The first graphic shows the initial fitted tree:

plot(CCTree)
text(CCTree, xpd = NA)

## The next graphic is to check for the need to prune:

plotcp(CCTree)
```


sd.amnt.atm.withdr< 437.6
av.saving.balance< 7.733e+05
sd.amnt.atm.withdr< 954.5
av.amnt.pmnts.init.by.cust>=0.1548
av.amnt.pmnts.init.by.cust< 0.1742
av.amnt.pmnts.init.by.cust>=0.2361
sd.cheque.credits< 2877
sd.salary.deposits< 1462
sex=a
av.num.atm.withdr< 0.2639
sd.cheque.credits>=0.7317
av.cheque.credits< 4954
sd.amnt.transfers>=3.296
av.cheque.cash.withdr>=1044
av.num.pmnts.init.by.cust>=18.89
no
yes


size of tree
1
2
3
4
8
9
11
14
16
X−val Relative Error
1.0
0.8
0.6
0.4
Inf
0.2
0.092
0.034
0.018
0.015
0.013
0.012
0.011
cp

Pruning is suggested by the “one standard error” rule. Get the pruned tree:

```
oneSERule(CCTree)                   ## optimal tuning paramater

[1] 0.01683502

CCPTree <- prune(CCTree, cp = oneSERule(CCTree))
Store(CCPTree, remove = FALSE)

plot(CCPTree)                 ## should have 10 terminal nodes
text(CCPTree, xpd = NA)
```


sd.amnt.atm.withdr< 437.6
av.saving.balance< 7.733e+05
yes
sd.salary.deposits< 1462
yes
sex=a
yes
no
av.num.atm.withdr< 0.2639
no
sd.cheque.credits>=0.7317
av.cheque.credits< 4954
yes
no
yes

The "one standard error rule" function(s) are listed here for completeness. (The coding details are not important for our primary purpose here.)

```
WWRCourse::oneSERule

function (tree, f, ...)
{
    UseMethod("oneSERule")
}
<bytecode: 0x5641497423b8>
<environment: namespace:WWRCourse>

methods("oneSERule")

[1] oneSERule.rpart*
see '?methods' for accessing help and source code

WWRCourse:::oneSERule.rpart

function (tree, f = 1, ...)
{
    cp <- data.frame(tree$cptable)
    imin <- with(cp, which(xerror == min(xerror))[1])
    with(cp, CP[which(xerror <= xerror[imin] + f * xstd[imin])[1]])
}
<bytecode: 0x56414d4708c0>
```

```
<environment: namespace:WWRCourse>
```

3.2 Simple bagging

"Bootstrap aggregation" — invented by Leo Breimann as a device to stabilize tree methods and improve their predictive capacity. Very much a "black box" technique.

- Grow a forest of trees using bootstrap samples of the training data.
- For predictions average over the forest:
  - For classification trees, take a majority vote,
  - For regression trees, take an average.

'Random forests', (Liaw and Wiener, 2002), is a development of bagging with extra protocols imposed.

Consider bagging "by hand".

```
bagRpart <- local({
  bsample <- function(dataFrame) # bootstrap sampling
    dataFrame[sample(nrow(dataFrame), rep = TRUE),  ]
  function(object, data = eval.parent(object$call$data),
           nBags=200, type = c("standard", "bayesian"), ...) {
    type <- match.arg(type)
    bagsFull <- vector("list", nBags)
    if(type == "standard") {
      for(j in 1:nBags)
          bagsFull[[j]] <- update(object, data = bsample(data))
      } else {
        nCases <- nrow(data)
        for(j in 1:nBags)
            bagsFull[[j]] <- update(object, data = data, weights = rexp(nCases))
        }
    class(bagsFull) <- "bagRpart"
    bagsFull
  }
})

## a prediction method for the objects (somewhat tricky!)
```

```r
predict.bagRpart <- function(object, newdata, ...) {
  X <- sapply(object, predict, newdata = newdata, type = "class")
  candidates <- levels(predict(object[[1]], type = "class"))
  X <- t(apply(X, 1, function(r) table(factor(r, levels = candidates))))
  factor(candidates[max.col(X)], levels = candidates)
}
Store(bagRpart, lib = .Robjects)
```

An alternative coding using *replicate*:

```
bagRpart <- local({
###
  bsample <- function(dataFrame) # bootstrap sampling
    dataFrame[sample(nrow(dataFrame), rep = TRUE),  ]
###
  function(object, data = eval.parent(object$call$data),
           nBags=200, type = c("standard", "bayesian"), ...) {
    type <- match.arg(type)
    bagsFull <- if(type == "standard") {
      replicate(nBags, update(object, data = bsample(data)),
                simplify = FALSE)
    } else {
      nCases <- nrow(data)
      replicate(nBags,  update(object, data = data, weights = rexp(nCases)),
                simplify = FALSE)
    }
    class(bagsFull) <- "bagRpart"
    bagsFull
  }
})
```

Now for an object or two:

```
set.seed(4321) ##
Obj <- update(CCTree, cp = 0.005, minsplit = 9)  ## expand the tree
(one <- rbind(standard = system.time(CCSBag <- bagRpart(Obj, nBags = 200)),
              Bayes = system.time(CCBBag <- bagRpart(Obj, nBags = 200,
                                                      type = "bayes"))))
```

```
         user.self sys.self elapsed user.child sys.child
standard    11.333    0.008  11.342          0         0
Bayes       11.561    0.004  11.565          0         0
```

```
Store(CCSBag, CCBBag, remove = FALSE)
```

3.3 A parallel version

Some preliminaries first:

```
## parallel backend; includes other pkgs
suppressPackageStartupMessages({
  library(doParallel)
})

(nc <- detectCores())  ## how many CPUs has your computer?

[1] 12

cl <- makeCluster(nc-1)
registerDoParallel(cl)
```

A modified version of the fitting function. Some care needed:

```
bagRpartParallel <- local({
  bsample <- function(dataFrame) # bootstrap sampling
    dataFrame[sample(nrow(dataFrame), rep = TRUE),  ]

  function(object, data = eval.parent(object$call$data),
           nBags = 200, type = c("standard", "bayesian"), ...,
           cores = detectCores() - 1, seed0 = as.numeric(Sys.Date())) {
    type <- match.arg(type)
    bagsFull <- foreach(j = idiv(nBags, chunks=cores), seed = seed0+seq(cores),
                        .combine = c, .packages = c("rpart", "stats"),
                        .inorder = FALSE, .export = c("bsample")) %dopar% {
                          ## now inside a single core
                          set.seed(seed = seed)
                          if(type == "standard") {
                            replicate(j, simplify = FALSE,
                                      update(object, data = bsample(data)))
                          } else {
                            replicate(j, simplify = FALSE,
                                      update(object, data = data,
                                             weights = rexp(nrow(data))))
                          }
```

```r
                          ## end of single core mode
                        }
    class(bagsFull) <- "bagRpart"
    bagsFull
  }
})
```

A timing comparison

```r
Obj <- update(CCTree, cp = 0.005, minsplit = 9)  ## expand the tree
rbind(one,
      standardP = system.time(CCSBagP <- bagRpartParallel(Obj, nBags = 200)),
      BayesP = system.time(CCBBagP    <- bagRpartParallel(Obj, nBags = 200,
                                                          type = "bayes")))

          user.self sys.self elapsed user.child sys.child
standard     11.333    0.008  11.342      0.000         0
Bayes        11.561    0.004  11.565      0.000         0
standardP     0.156    0.016   1.993      0.002         0
BayesP        0.152    0.008   1.924      0.003         0

rm(Obj, one)
Store(CCSBagP, CCBBagP, remove = FALSE)
```

3.4 The actual random forest

The random forest package, (Liaw and Wiener, 2002), implements this technology, and more, automatically. The number of trees is set to 500 by default. How many times does each observation get sampled if we restrict it to 200 trees?

```
n <- nrow(CCTrain)
X <- replicate(200,
       table(factor(sample(n, rep=TRUE), levels = 1:n)))
(lims <- range(rowSums(X > 0)))
```

```
[1] 104 150
```

```
rm(n, X)
```

So in this simulation the cases were sampled between one hundred and four and one hundred and fifty times. This seems about enough.

We now fit the random forest.

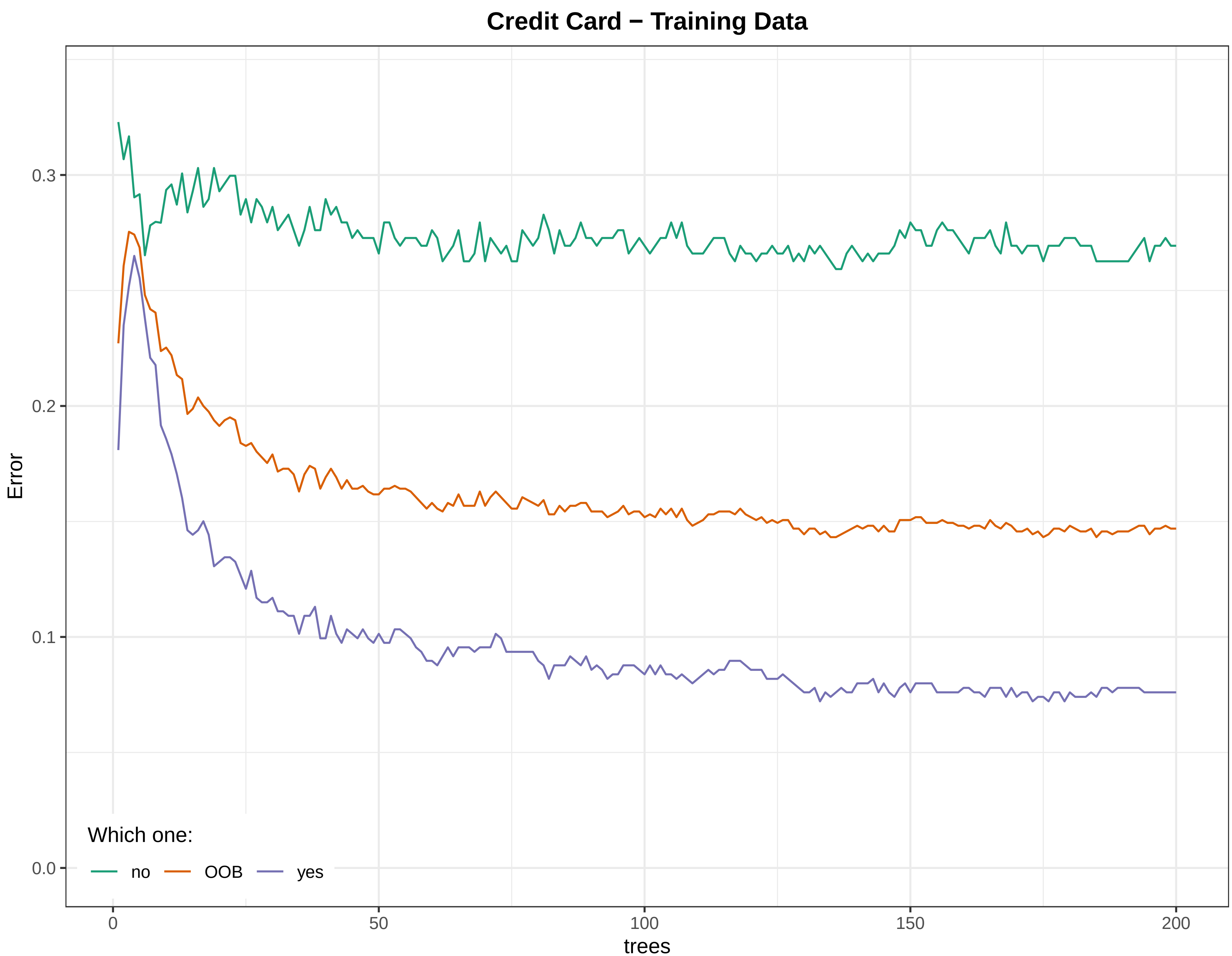
```
requireData(randomForest)
(CCRf <- randomForest(credit.card.owner ~ ., CCTrain, ntree = 200))


Call:
 randomForest(formula = credit.card.owner ~ ., data = CCTrain,      ntree = 200)
               Type of random forest: classification
                     Number of trees: 200
No. of variables tried at each split: 7

        OOB estimate of  error rate: 14.69%
Confusion matrix:
     no yes class.error
no  217  80  0.26936027
yes  39 474  0.07602339

Store(CCRf, remove = FALSE)
```

The "out of bag" (OOB) error rate is estimated as 14.69%. We check later how this compares with the observed error rate in the reserved *Test* data.

3.5 Progressive error rate

The random forest technology keeps a tally of the OOB error rate as the forest grows.

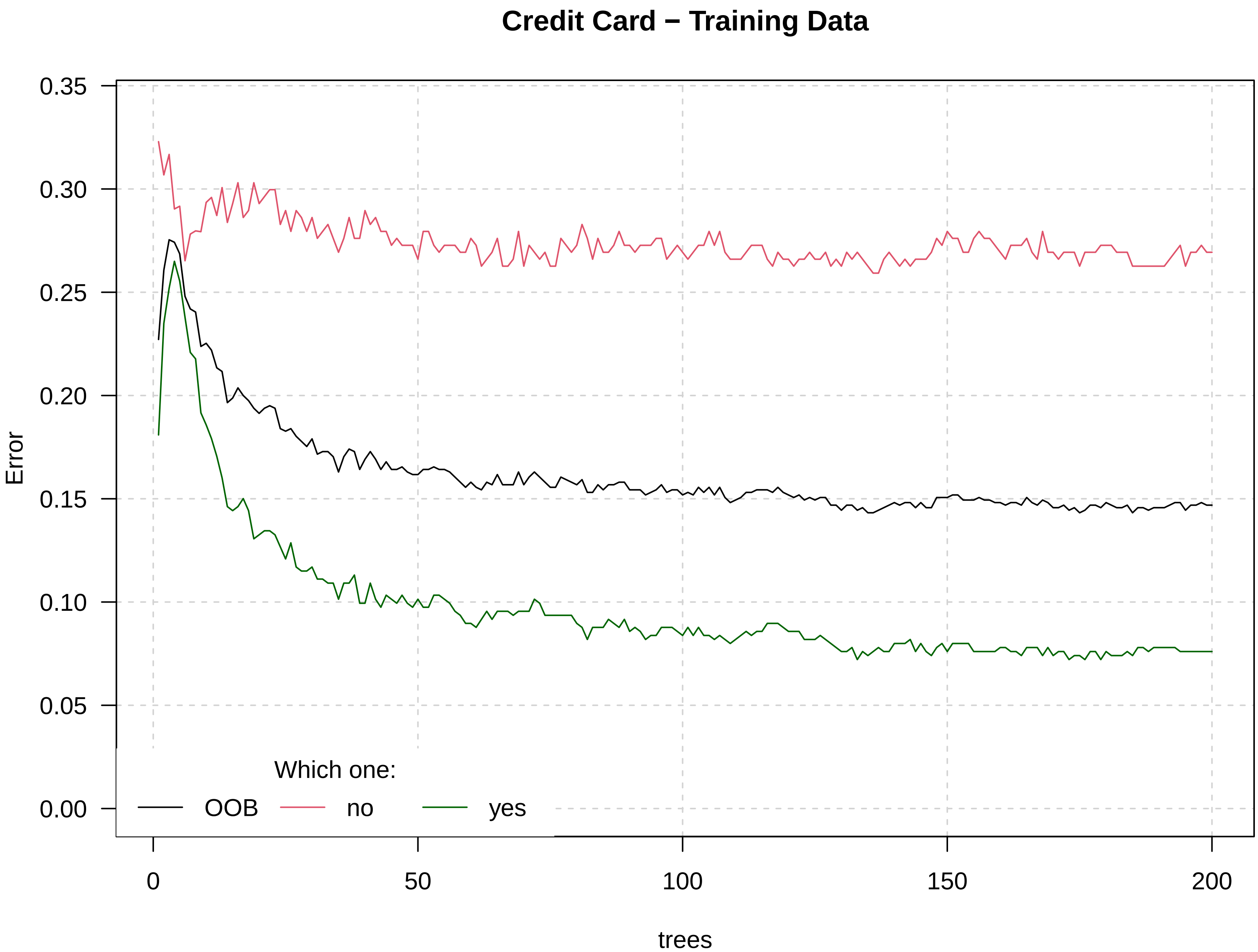
```
err <- as.data.frame(CCRf$err.rate) %>%
  rownames_to_column("trees") %>%
  mutate(trees = as.numeric(trees)) %>%
  pivot_longer(cols = -trees, names_to = "Which", values_to = "Error")
ggplot(err) + aes(x = trees, y = Error, colour = Which) + geom_line() +
  scale_colour_brewer(palette = "Dark2", name = "Which one:") +
  ylim(0, max(err$Error)*1.05) + labs(title = "Credit Card - Training Data")  +
  guides(colour = guide_legend(ncol = 3)) +
  theme_bw() + theme(legend.position = c(0, 0)+0.01,
                     legend.justification = c("left", "bottom"),
                     plot.title = element_text(hjust = 0.5, face = "bold"))
```



Credit Card − Training Data
0.0
0.1
0.2
0.3
Error
0
50
100
150
200
trees
Which one:
no
OOB
yes

We can do this more directly with the *plot* method for random forest objects. As is our custom, we add a few little aesthetic enhancements:

```
ER <- CCRf$err.rate
plot(CCRf, ylim = range(0, ER)*1.05, lty = "solid", las = 1,
     panel.first = grid(lty = "dashed"), main = "Credit Card - Training Data")
legend("bottomleft", colnames(ER), ncol = ncol(ER), bg = "white",
       col = 1:3, lty = "solid", bty = "o", box.col = "transparent",
       title = "Which one:")
```

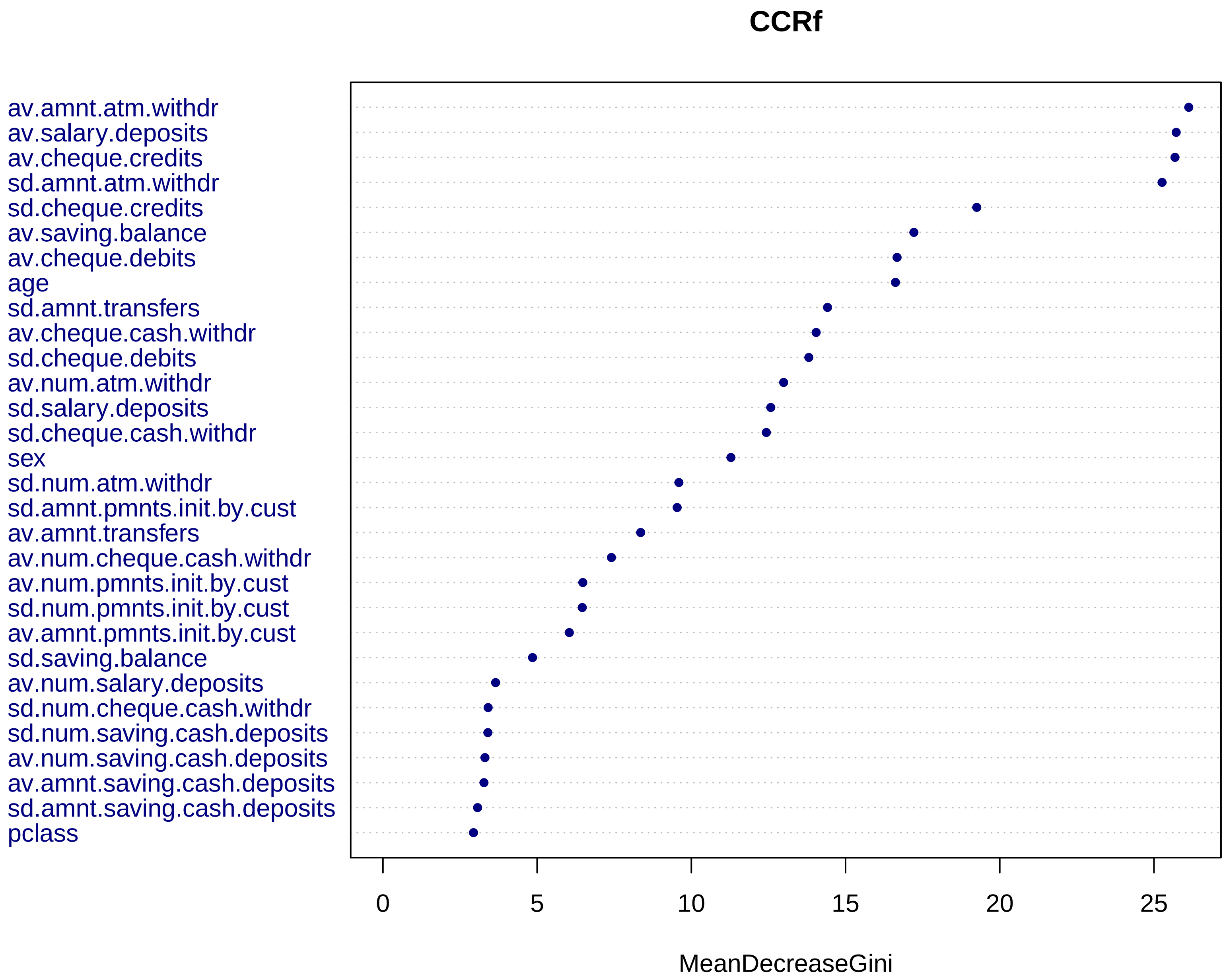


Credit Card − Training Data
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Error
0
50
100
150
200
trees
Which one:
OOB
no
yes

3.6 Variable importances

One other nice by-product is variable importances.

```
par(family="sans")
varImpPlot(CCRf, pch = 20, col = "navy")  ## causes a plot
```



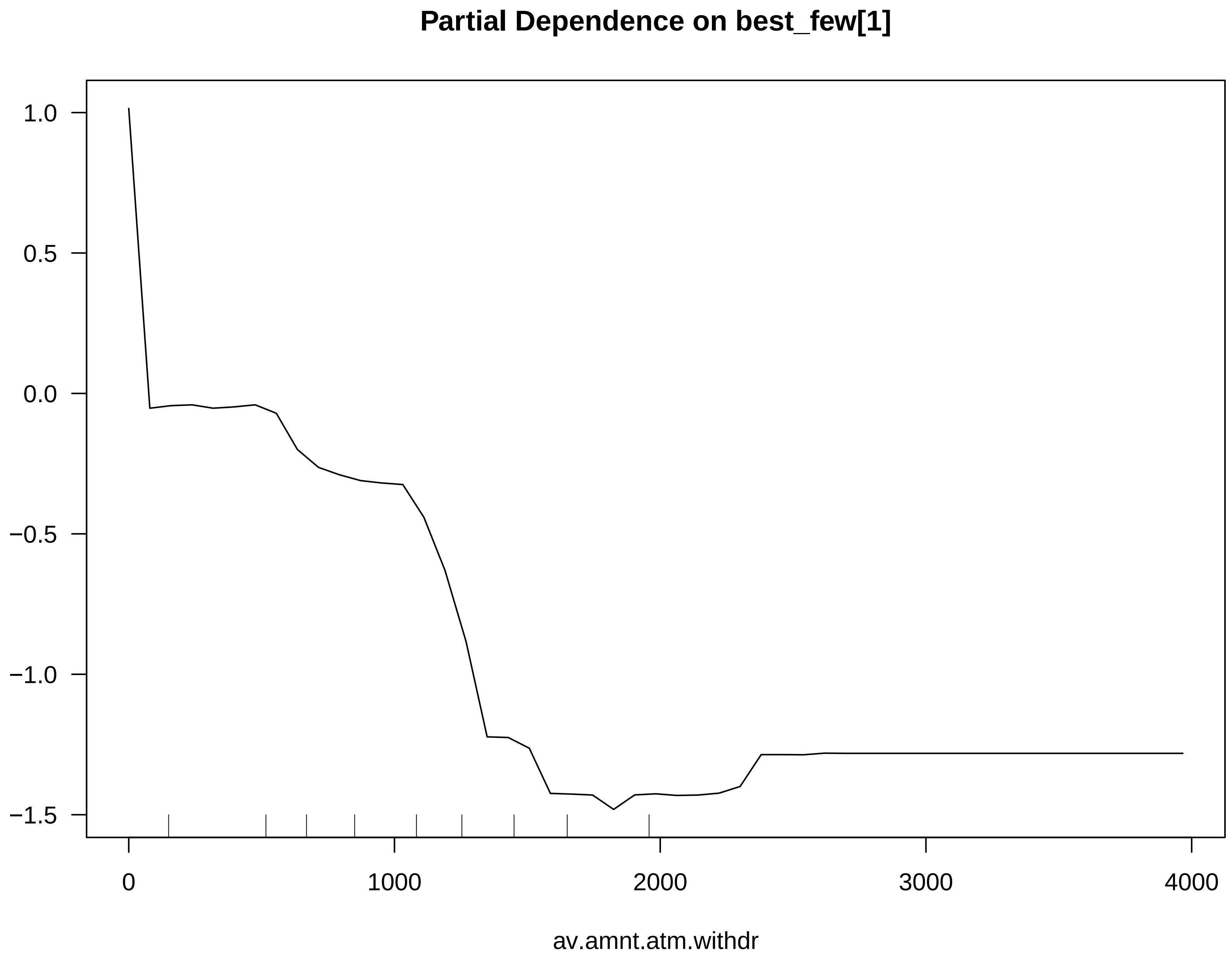
CCRf
av.amnt.atm.withdr
av.salary.deposits
av.cheque.credits
sd.amnt.atm.withdr
sd.cheque.credits
av.saving.balance
av.cheque.debits
age
sd.amnt.transfers
av.cheque.cash.withdr
sd.cheque.debits
av.num.atm.withdr
sd.salary.deposits
sd.cheque.cash.withdr
sex
sd.num.atm.withdr
sd.amnt.pmnts.init.by.cust
av.amnt.transfers
av.num.cheque.cash.withdr
av.num.pmnts.init.by.cust
sd.num.pmnts.init.by.cust
av.amnt.pmnts.init.by.cust
sd.saving.balance
av.num.salary.deposits
sd.num.cheque.cash.withdr
sd.num.saving.cash.deposits
av.num.saving.cash.deposits
av.amnt.saving.cash.deposits
sd.amnt.saving.cash.deposits
pclass
0
5
10
15
20
25
MeanDecreaseGini

```
(v <- sort(drop(importance(CCRf)), decreasing = TRUE))[1:6]

av.amnt.atm.withdr av.salary.deposits  av.cheque.credits
          26.12861           25.72058           25.68204
sd.amnt.atm.withdr  sd.cheque.credits  av.saving.balance
          25.26308           19.25392           17.21673

best_few <- names(v)[1:20] %>% print ## used later

 [1] "av.amnt.atm.withdr"            "av.salary.deposits"
 [3] "av.cheque.credits"             "sd.amnt.atm.withdr"
 [5] "sd.cheque.credits"             "av.saving.balance"
 [7] "av.cheque.debits"              "age"
 [9] "sd.amnt.transfers"             "av.cheque.cash.withdr"
[11] "sd.cheque.debits"              "av.num.atm.withdr"
[13] "sd.salary.deposits"            "sd.cheque.cash.withdr"
[15] "sex"                           "sd.num.atm.withdr"
[17] "sd.amnt.pmnts.init.by.cust" "av.amnt.transfers"
[19] "av.num.cheque.cash.withdr"  "av.num.pmnts.init.by.cust"
```

3.6.1 Partial plots for predictor variables

These look at the way predictors appear to influence the response, *mutatis mutandis*.

```
partialPlot(CCRf, pred.data = CCTrain, x.var = best_few[1], xlab = best_few[1])
```



Partial Dependence on best_few[1]
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
1000
2000
3000
4000
av.amnt.atm.withdr

3.7 Parametric models

Tree models and random forests are natural competitors to the standard parametric models, notably GLMs. We begin with a naive model based only on what appear good variables in the random forest, and then consider other modest versions, but automatically produced.

```
(form <- as.formula(paste("credit.card.owner~",
                          paste(best_few, collapse="+"))))

credit.card.owner ~ av.amnt.atm.withdr + av.salary.deposits +
    av.cheque.credits + sd.amnt.atm.withdr + sd.cheque.credits +
    av.saving.balance + av.cheque.debits + age + sd.amnt.transfers +
    av.cheque.cash.withdr + sd.cheque.debits + av.num.atm.withdr +
    sd.salary.deposits + sd.cheque.cash.withdr + sex + sd.num.atm.withdr +
    sd.amnt.pmnts.init.by.cust + av.amnt.transfers + av.num.cheque.cash.withdr +
    av.num.pmnts.init.by.cust

Call <- substitute(glm(FORM, binomial, CCTrain), list(FORM = form))
CCGlmNaive <- eval(Call)
Store(CCGlmNaive, remove = FALSE)
```

```r
upp <- paste("~", paste(setdiff(names(CCTrain),
                                "credit.card.owner"),
                        collapse="+")) %>%
  as.formula()
scope <- list(upper=upp, lower=~1)
CCGlmAIC <- step_AIC(CCGlmNaive, scope = scope)
CCGlmGIC <- step_GIC(CCGlmNaive, scope = scope)
CCGlmBIC <- step_BIC(CCGlmNaive, scope = scope)
Store(CCGlmAIC, CCGlmBIC, form, upp, remove = FALSE)
```

3.8 Boosting other models

The *mboost* package is an implementation of the boosting idea for a suite of different models. It is described in Buehlmann and Hothorn (2007). We consider two different kinds of boosted models, namely a boosted GLM and a boosted TREE version.

The package is profligate in the number of others it requires!

```
requireData(mboost)
Call <- substitute(glmboost(FORM, data = CCTrain, family=Binomial()),
                   list(FORM = form))  ## same as naive model
CCglmboost <- eval(Call)

  ## more packages needed!
CCblackboost <- blackboost(credit.card.owner ~ ., CCTrain,
                           family = Binomial())
Store(CCglmboost, CCblackboost, remove = FALSE)
```

3.9 Weird science: C5.0

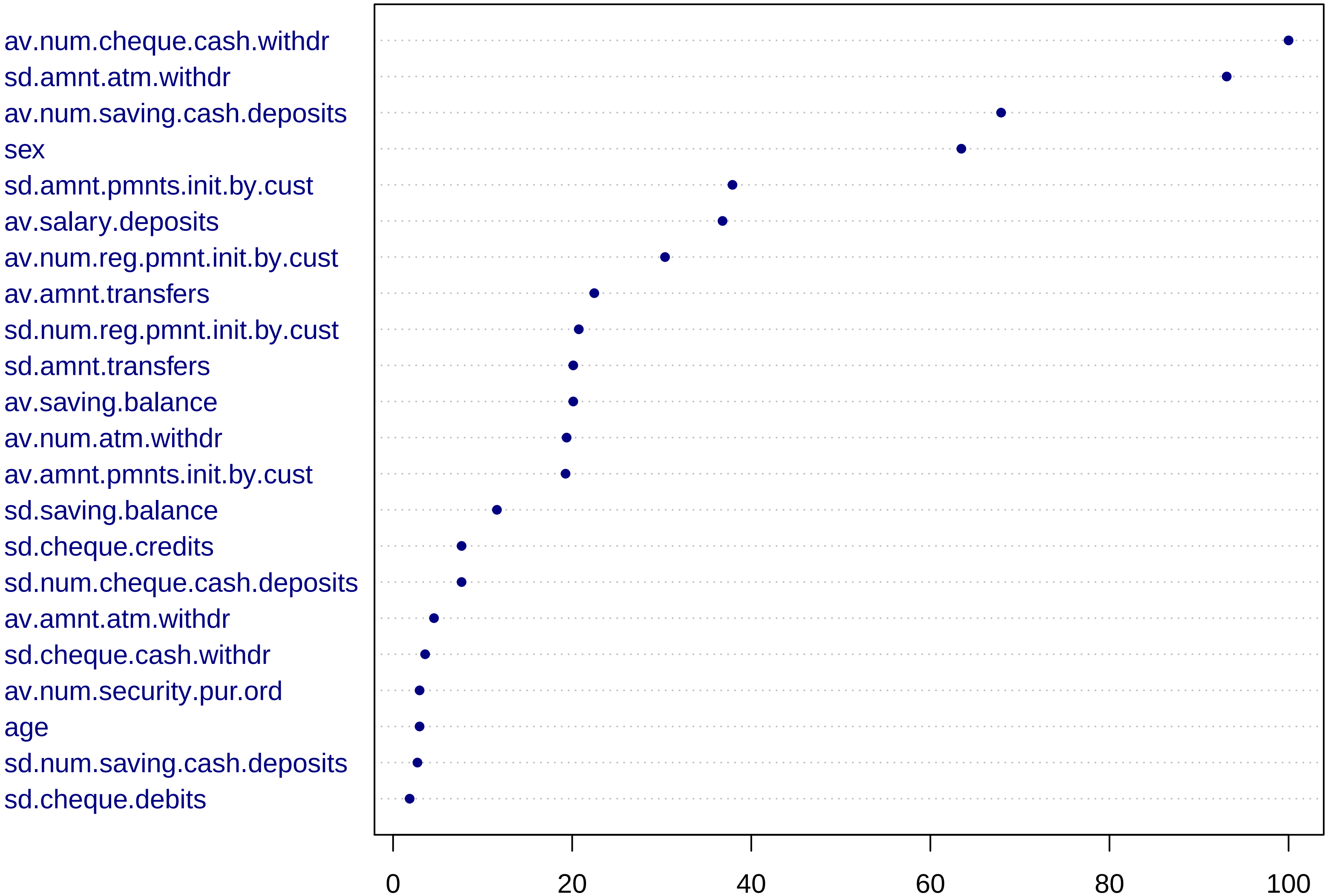
This is a development of a previously proprietary algorithm of Quinlan (1993); Kuhn and Quinlan (2020). It yields a rule-based classifier that might be considered intermediate between trees and forests.

```
requireData(C50)  ## NB C-5-Zero
(CCc50 <- C5.0(credit.card.owner ~ ., CCTrain))


Call:
C5.0.formula(formula = credit.card.owner ~ ., data = CCTrain)

Classification Tree
Number of samples: 810
Number of predictors: 54

Tree size: 37

Non-standard options: attempt to group attributes

Store(CCc50)
```

The C50 package, like randomForest, has some tools that allow the fitted model object to be investigated.

These include tools to assess *variable importance*, where the measure itself is somewhat obscure, but it expressed as a percentage.

```
obj <- C5imp(CCc50)
data.frame(importance = obj$Overall, variable = rownames(obj)) %>%
  filter(importance > 0) %>%
  arrange(importance) %>%
  with(., dotchart(importance, as.character(variable), pch = 20, col = "navy"))
rm(obj)
```


av.num.cheque.cash.withdr
sd.amnt.atm.withdr
av.num.saving.cash.deposits
sex
sd.amnt.pmnts.init.by.cust
av.salary.deposits
av.num.reg.pmnt.init.by.cust
av.amnt.transfers
sd.num.reg.pmnt.init.by.cust
sd.amnt.transfers
av.saving.balance
av.num.atm.withdr
av.amnt.pmnts.init.by.cust
sd.saving.balance
sd.cheque.credits
sd.num.cheque.cash.deposits
av.amnt.atm.withdr
sd.cheque.cash.withdr
av.num.security.pur.ord
age
sd.num.saving.cash.deposits
sd.cheque.debits
0
20
40
60
80
100

3.10 Gradient boosting models, GBM

Like C5.0 this method was proprietary but for some time now there has been an R implementation by Greg Ridgeway, (Greenwell et al., 2020).

It offers a powerful machine learning method for several classes of models, including 'bernoulli' (binary data, but **not** multinomial, unlike randomForest or even C5.0.)

Bernoulli models need to be fitted with a numeric response confined to the values $\{0, 1\}$, which can be a bit awkward:

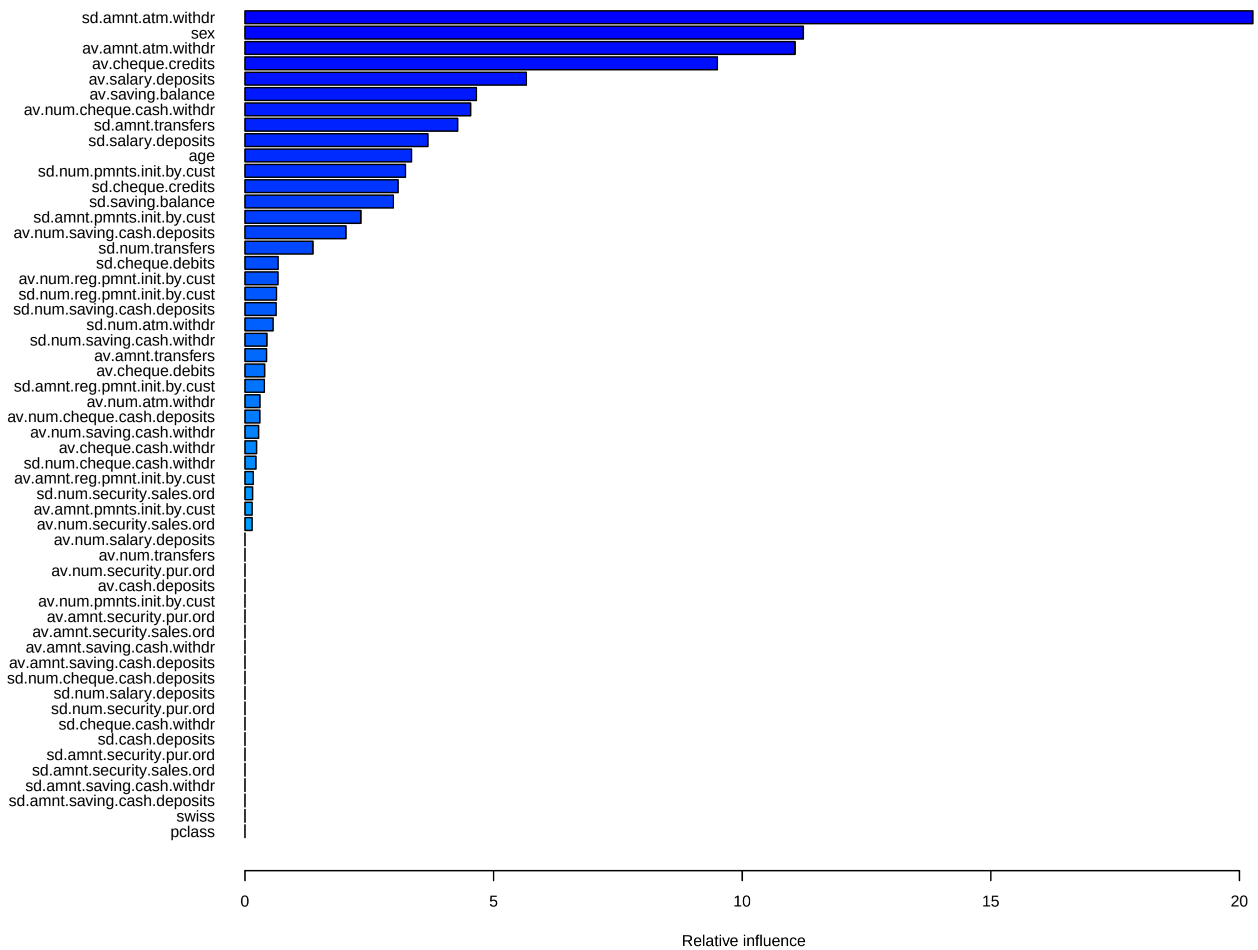
```
suppressPackageStartupMessages(library(gbm))
(CCgbm <- gbm(ifelse(credit.card.owner == "no", 0, 1) ~ ., data = CCKnown,
              distribution = "bernoulli", n.trees = 200, cv.folds = 10))

gbm(formula = ifelse(credit.card.owner == "no", 0, 1) ~ ., distribution = "bernoull
    data = CCKnown, n.trees = 200, cv.folds = 10)
A gradient boosted model with bernoulli loss function.
200 iterations were performed.
The best cross-validation iteration was 200.
There were 54 predictors of which 34 had non-zero influence.
```

There are several useful tools for investigating the fit and efficacy of a GBM. The first of these is one for looking at the relative influence of the predictors:

```
summary(CCgbm, plotit = FALSE) %>% filter(rel.inf > 0)

                                                             var    rel.inf
sd.amnt.atm.withdr                            sd.amnt.atm.withdr 20.2703539
sex                                                          sex 11.2280381
av.amnt.atm.withdr                            av.amnt.atm.withdr 11.0621992
av.cheque.credits                              av.cheque.credits  9.5013640
av.salary.deposits                            av.salary.deposits  5.6615081
av.saving.balance                              av.saving.balance  4.6562862
av.num.cheque.cash.withdr              av.num.cheque.cash.withdr  4.5393055
....
av.num.saving.cash.withdr              av.num.saving.cash.withdr  0.2743822
av.cheque.cash.withdr                      av.cheque.cash.withdr  0.2336712
sd.num.cheque.cash.withdr              sd.num.cheque.cash.withdr  0.2199467
av.amnt.reg.pmnt.init.by.cust av.amnt.reg.pmnt.init.by.cust  0.1665780
sd.num.security.sales.ord              sd.num.security.sales.ord  0.1537358
av.amnt.pmnts.init.by.cust            av.amnt.pmnts.init.by.cust  0.1441431
av.num.security.sales.ord              av.num.security.sales.ord  0.1435577
```

But wait, there's more. You can get a *barplot* of the result using same function

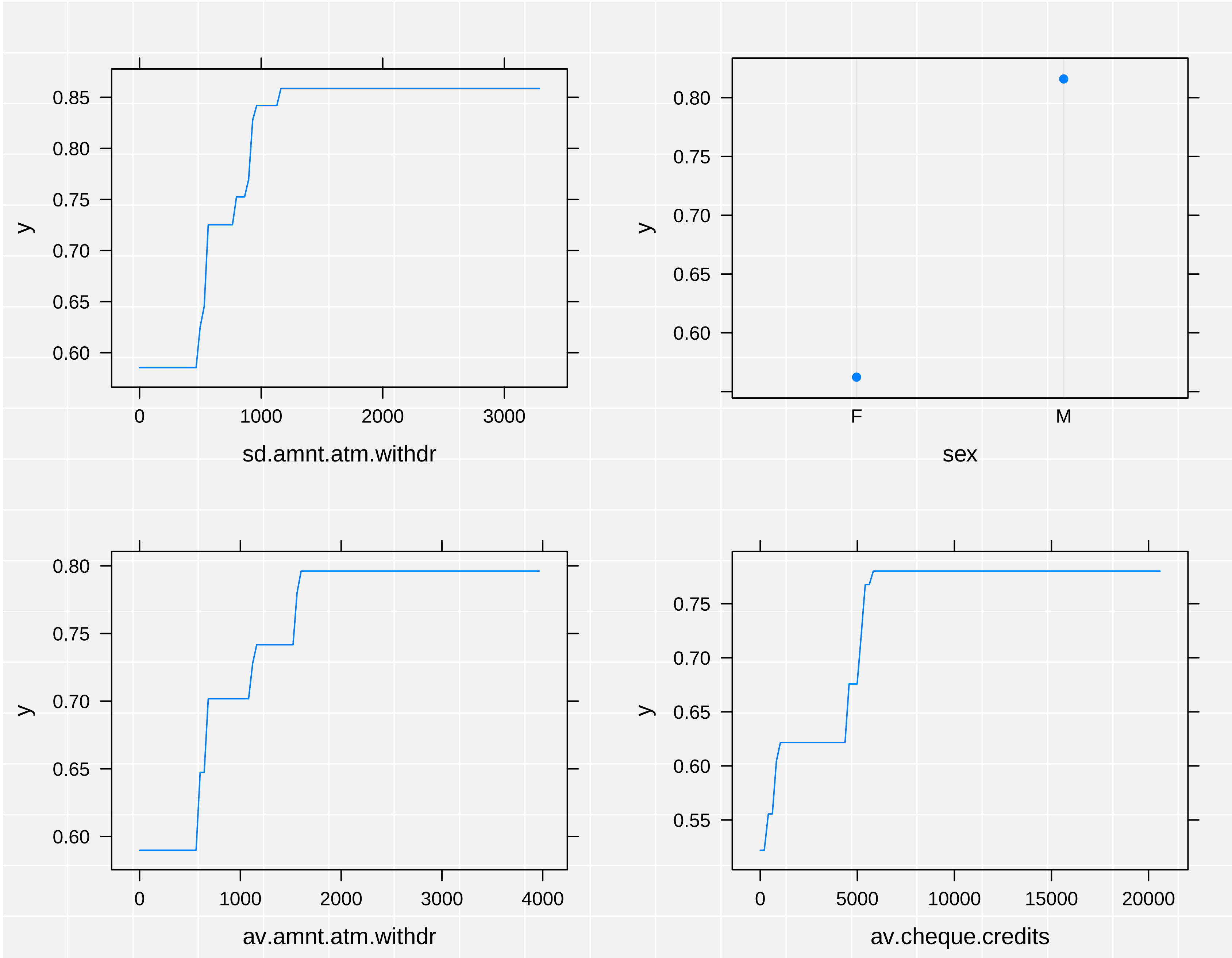
```
par(mar = c(4,12,1,1), cex = 0.7)
tmp <- summary(CCgbm)  ## tmp is now the data frame displayed above.
```



Partial dependency plots are also available.

Look at the four "most influential" variables

```
suppressPackageStartupMessages({
  library(gridExtra)
})
(vars <- tmp$var[1:4])                    ## four "most influential" variables

[1] "sd.amnt.atm.withdr" "sex"                "av.amnt.atm.withdr"
[4] "av.cheque.credits"

plots <- lapply(vars, . %>% plot(CCgbm, i.var = ., type = "response"))
grobs <- arrangeGrob(grobs = plots, layout_matrix = rbind(1:2, 3:4))
plot(grobs)
```


0.85
0.80
0.75
0.70
0.65
0.60
y
0
1000
2000
3000
sd.amnt.atm.withdr
0.80
0.75
0.70
0.65
0.60
y
F
M
sex
0.80
0.75
0.70
0.65
0.60
y
0
1000
2000
3000
4000
av.amnt.atm.withdr
0.75
0.70
0.65
0.60
0.55
y
0
5000
10000
15000
20000
av.cheque.credits

4 The final reckoning

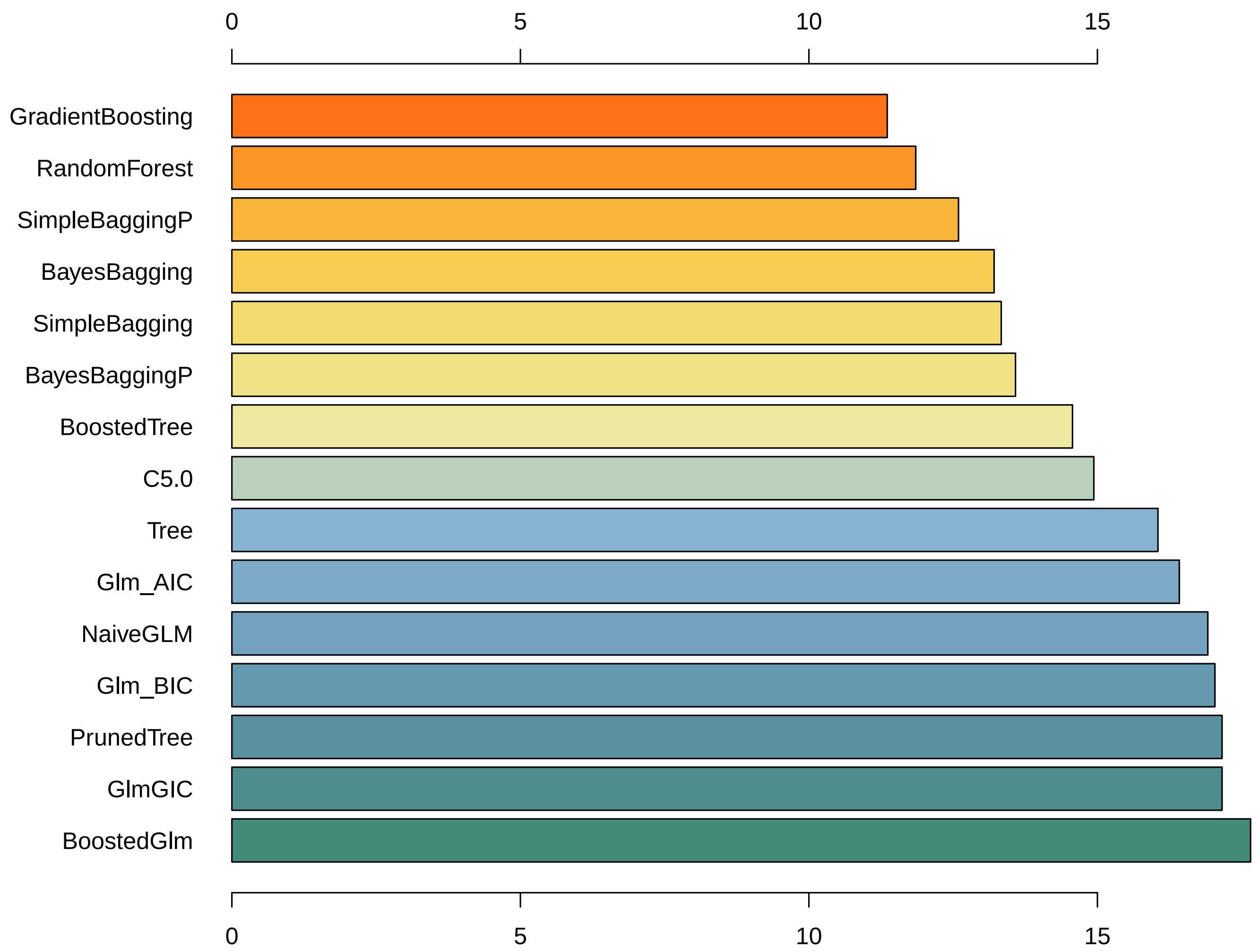
Now to see how things worked out this time. First a helper function

```
Class <- function(object, newdata, ...)
    UseMethod("Class")

Class.rpart <- function(object, newdata, ...)
    predict(object, newdata, type = "class")

Class.bagRpart <- function(object, newdata, ...)
    predict(object, newdata)

Class.randomForest <- Class.C5.0 <- predict

Class.glm <- Class.mboost <- Class.gbm <- function(object, newdata, ...) {
  ## only applies for binomial GLMs with symmetric links
  suppressMessages(predict(object, newdata) > 0)
}
```

The helper function *Class* streamlines things a bit:

```
errorRate <- function(tab) 100*(1 - sum(diag(tab))/sum(tab))
true <- CCTest$credit.card.owner  #$
(res <- sort(sapply(list(Tree = CCTree,             PrunedTree = CCPTree,
                         SimpleBagging = CCSBag,    BayesBagging = CCBBag,
                         SimpleBaggingP = CCSBagP,  BayesBaggingP = CCBBagP,
                         RandomForest = CCRf,       C5.0 = CCc50,
                         GradientBoosting = CCgbm,  BoostedGlm = CCglmboost,
                         NaiveGLM = CCGlmNaive,     BoostedTree = CCblackboost,
                         Glm_AIC = CCGlmAIC,        GlmGIC = CCGlmGIC,
                         Glm_BIC = CCGlmBIC),
                    function(x) errorRate(table(Class(x, CCTest), true)))))

GradientBoosting     RandomForest   SimpleBaggingP     BayesBagging
        11.35802         11.85185         12.59259         13.20988
   SimpleBagging    BayesBaggingP      BoostedTree             C5.0
        13.33333         13.58025         14.56790         14.93827
            Tree          Glm_AIC         NaiveGLM          Glm_BIC
        16.04938         16.41975         16.91358         17.03704
      PrunedTree           GlmGIC       BoostedGlm
        17.16049         17.16049         17.65432
```

```r
par(mar = c(3,8,3,1))
barplot(rev(res), horiz=TRUE, las=1, fill = pal_green2brown)
axis(3)
```



References

Buehlmann, P. and T. Hothorn (2007). Boosting algorithms: Regularization, prediction and model fitting (with discussion). *Statistical Science 22*, 477–505.

Greenwell, B., B. Boehmke, J. Cunningham, and G. Developers (2020). *gbm: Generalized Boosted Regression Models*. R package version 2.1.8.

Kuhn, M. and R. Quinlan (2020). *C50: C5.0 Decision Trees and Rule-Based Models*. R package version 0.1.3.1.

Liaw, A. and M. Wiener (2002). Classification and regression by `randomForest`. *R News 2*(3), 18–22.

Quinlan, R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers.

Ripley., B. (2012). *tree: Classification and Regression Trees*. CRAN. R package version 1.0-29.

Session information

Date: 2021-01-29

- R version 4.0.3 (2020-10-10), `x86_64-pc-linux-gnu`
- Running under: `Ubuntu 20.04.1 LTS`
- Matrix products: default
- BLAS: `/usr/lib/x86_64-linux-gnu/blas/libblas.so.3.9.0`
- LAPACK: `/usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.9.0`
- Base packages: base, datasets, graphics, grDevices, methods, parallel, stats, utils
- Other packages: C50 0.1.3.1, doParallel 1.0.16, dplyr 1.0.3, english 1.2-5, forcats 0.5.1, foreach 1.5.1, gbm 2.1.8, GGally 2.1.0, ggplot2 3.3.3, ggthemes 4.2.4, gridExtra 2.3, haven 2.3.1, iterators 1.0.13, knitr 1.31, lattice 0.20-41, lme4 1.1-26, Matrix 1.3-2, mboost 2.9-4, mgcv 1.8-33, microbenchmark 1.4-7, nlme 3.1-151, patchwork 1.1.1, purrr 0.3.4, randomForest 4.6-14, rbenchmark 1.0.0, Rcpp 1.0.6, readr 1.4.0, rpart 4.1-15, scales 1.1.1, SOAR 0.99-11, stabs 0.6-3, stringr 1.4.0, tibble 3.0.5, tidyr 1.1.2, tidyverse 1.3.0, visreg 2.7.0, WWRCourse 0.2.3, WWRData 0.1.0, WWRGraphics 0.1.2, WWRUtilities 0.1.2, xtable 1.8-4
- Loaded via a namespace (and not attached): assertthat 0.2.1, backports 1.2.1, boot 1.3-26, broom 0.7.3, cellranger 1.1.0, cli 2.2.0, codetools 0.2-18, colorspace 2.0-0, compiler 4.0.3, crayon 1.3.4, Cubist 0.2.3, DBI 1.1.1, dbplyr 2.0.0, digest 0.6.27, ellipsis 0.3.1, evaluate 0.14, fansi 0.4.2, farver 2.0.3, Formula 1.2-4, fractional 0.1.3, fs 1.5.0, generics 0.1.0, glue 1.4.2, grid 4.0.3, gtable 0.3.0, highr 0.8, hms 1.0.0, httr 1.4.2, inum 1.0-1, jsonlite 1.7.2, labeling 0.4.2, lazyData 1.1.0, libcoin 1.0-7, lifecycle 0.2.0, lubridate 1.7.9.2, magrittr 2.0.1, MASS 7.3-53, minqa 1.2.4, modelr 0.1.8, munsell 0.5.0, mvtnorm 1.1-1, nloptr 1.2.2.2, nnls 1.4, partykit 1.2-11, PBSmapping 2.73.0, pillar 1.4.7, pkgconfig 2.0.3, plyr 1.8.6, quadprog 1.5-8, R6 2.5.0, RColorBrewer 1.1-2, readxl 1.3.1, reprex 1.0.0, reshape 0.8.8, reshape2 1.4.4, rlang 0.4.10, rstudioapi 0.13, rvest 0.3.6, splines 4.0.3, statmod 1.4.35, stringi 1.5.3, survival 3.2-7, tidyselect 1.1.0, tools 4.0.3, vctrs 0.3.6, withr 2.4.1, xfun 0.20, xml2 1.3.2