Additional Topics and Case Study

High-Performance Computing

Motivation

Often, computations take ("too") long. What can we do?

Hard answer: **avoid inefficient computations**.

- Avoid recomputing things already computed.
- Avoid making and keeping unnecessary copies of (large) data sets or other R objects.
- Avoid storing things not actually needed.
- Be clever when reading in (large) data sets, and subsequently keep these in a format R can handle more efficiently than CSV.

Motivation

- Avoid loops? As always, "it depends" . . .
  Many computations in R are already vectorized (i.e., work on sequences), for these iteration over individual elements will take (much) longer. E.g.,

```
R> x <- 1 : 123456
R> system.time(s1 <- sum(x^2))

   user  system elapsed
  0.001   0.000   0.001

R> system.time({
+       s2 <- 0
+       for(i in seq_along(x)) s2 <- s2 + x[i]^2
+ })

   user  system elapsed
  0.018   0.000   0.018

R> s1 == s2

[1] TRUE
```

Motivation

- Avoid computations that unnecessarily need to reallocate storage, in particular appending to sequences:

```
R> B <- 123456
R> system.time({
+      x <- numeric()
+      for(i in 1 : B) x[i] <- i^2
+ })
   user  system elapsed 
  0.028   0.008   0.036 
R> system.time({
+      x <- numeric(B)
+      for(i in 1 : B) x[i] <- i^2
+ })
   user  system elapsed 
  0.008   0.000   0.008 
```

  Of course, one should really do `x <- (1 : B)^2`.

Motivation

- Use better/faster implementations of learning methods in add on packages (e.g., package `ranger` instead of `RandomForest`, package `xgboost` instead of `gbm`, ...???)

Easy answer: **use more available resources**.

On modern computers, many computations can be done "in parallel".

At a high level, can use this for "coarse-grained parallelization" for "embarrasingly parallel computations" (e.g., applying the same function on many different data sets [as we do for bootstrapping or cross-validation]).

In R provided by package `parallel`.

Package `parallel`

Basic computational model:

(a) Start up M 'worker' processes, and do any initialization needed on the workers.
(b) Send any data required for each task to the workers.
(c) Split the task into M roughly equally-sized chunks, and send the chunks (including the R code needed) to the workers.
(d) Wait for all the workers to complete their tasks, and ask them for their results.
(e) Repeat steps (b–d) for any further tasks.
(f) Shut down the worker processes.

Package `parallel` provides functions `mclapply` and `parLapply` as near-drop-in replacements for `lapply`.

Different model: split into $M_1 > M$ chunks, send the first M chunks to the workers, then repeatedly wait for any worker to complete and send it the next remaining task: **load balancing**.

Package `parallel`

Workers are (currently) implemented as full processes, created in one of three ways:

1. Via launching new R processes on the current machine or a similar one, and using sockets for communication.
   Should be available on all R platforms (at least when using the current machine).
2. Via **forking**: available on all platforms except Windows. Creates new processes as complete copies of the master process (including workspace and random number streams).
3. Via special OS facilities to send tasks to other members of a group of machines (MPI, . . .).

Numbers of CPUs or cores

Determining "available resources" can be a bit tricky.

These days, all physical CPUs contain two or more cores that run more-or-less independently.

However, cores may be able to run several tasks simultaneously, and Windows has the concept of "logical CPUs".

```
R> require("parallel")
R> detectCores()
```

```
[1] 8
```

Where possible (Windows), this gives logical (e.g., hyperthreaded) CPUs.

Parallel variants of `lapply`

The basic functions in `parallel` are:

```
parLapply(cl, x, FUN, ...)
mclapply(X, FUN, ..., mc.cores)
```

`mclapply()` uses forking (hence only makes sense on non-Windows systems) to set up a pool of `mc.cores` workers just for the desired computation.

`parLapply()` uses sockets to communicate with a pool of workers ("clusters", also "SNOW [Simple Network of Workstations] clusters") specified by `cl` as started by `makeCluster()` (and stopped eventually by `stopCluster()`. Workflow:

```
cl <- makeCluster(<size of pool>)
## One ore more calls to parLapply()
stopCluster(cl)
```

Example 1

Suppose we want to simulate B means of samples of size n from the standard normal distribution:

```
R> B <- 123
R> fun <- function(b) mean(rnorm(10000))
```

One at a time:

```
R> system.time(lapply(1 : B, fun))
```

```
   user  system elapsed
  0.075   0.000   0.075
```

If not on Windows, can parallelize using `mcparallel()`:

```
R> system.time(mclapply(1 : B, fun, mc.cores = 4L))
```

```
   user  system elapsed
  0.002   0.004   0.027
```

Example 1

Everywhere, can parallelize using `parLapply()`.

```
R> system.time({
+     cl <- makeCluster(4)
+     parLapply(cl, 1 : B, fun)
+     stopCluster(cl)
+ })

   user  system elapsed
  0.011   0.001   0.207
```

Why is this soooo slow?

Example 1

Well, most of the time is spent setting up the cluster:

```
R> cl <- makeCluster(4)
R> system.time({
+     parLapply(cl, 1 : B, fun)
+ })

   user  system elapsed 
  0.001   0.000   0.023 

R> stopCluster(cl)
```

So indeed, when using `parLapply()`, we should set up the cluster once and re-use for subsequent parallelized computations (and stop eventually).

Note that still we are not seeing linear speedup (noticeable communication overhead).

Example 2

Re-consider using k-fold cross-validation to estimate the predictive performance the linear regression model of Amount on Duration and Job.

Data set and helper functions:

```
R> load("german.rda")
R> MSE <- function(y, yhat) mean((y - yhat)^2)
R> folds <- function(n, k)
+     unname(split(sample(1 : n, n), cut(1 : n, breaks = k)))
```

For each fold with observation numbers `i` we need to apply:

```
R> fun <- function(i) {
+     m <- lm(Amount ~ Duration + Job, data = german[-i, ])
+     MSE(german$Amount[i], predict(m, german[i, ]))
+ }
```

Example 2

We use $k = 24$ folds.

```
R> k <- 24
```

One at a time:

```
R> system.time(lapply(folds(nrow(german), k), fun))

   user  system elapsed 
  0.066   0.004   0.070 
```

If not on Windows, can parallelize using `mcparallel()`:

```
R> system.time(mclapply(folds(nrow(german), k), fun, mc.cores = 4))

   user  system elapsed 
  0.010   0.000   0.035 
```

Example 2

Everywhere, can parallelize using `parLapply()`.

However, we cannot simply do

```
R> parLapply(cl, folds(nrow(german), k), fun)
```

as the worker processes know nothing about `german` or `MSE` (only the master process has to know `folds`).

So we either need to integrate loading data and additional functions needed into the function to be performed, or make these variables available to the worker processes using `clusterExport()`:

```
R> clusterExport(cl, c("german", "MSE"))
R> parLapply(cl, folds(nrow(german), k), fun)
```

Example 2

With this, we get:

```
R> cl <- makeCluster(4)
R> clusterExport(cl, c("german", "MSE"))
R> system.time({
+     parLapply(cl, folds(nrow(german), k), fun)
+ })
```

```
   user  system elapsed 
  0.002   0.000   0.029 
```

```
R> stopCluster(cl)
```

Again: not bad, but speedup is not linear.

Random-number generation

Suppose we want to parallelize in a reproducible way (i.e., via setting the random seed). Reusing Example 1:

```
R> B <- 123
R> fun <- function(b) mean(rnorm(10000))
R> cl <- makeCluster(4)
R> set.seed(4711)
R> mpar1 <- unlist(parLapply(cl, 1 : B, fun))
R> set.seed(4711)
R> mpar2 <- unlist(parLapply(cl, 1 : B, fun))
R> table(mpar1 == mpar2)
```

```
FALSE
  123
```

```
R> stopCluster(cl)
```

Why can we not reproduce the simulated values?

Random-number generation

Well, we set the seed for the master process, but this did not get propagated to the workers. Can be accomplished using `clusterSetRNGStream()`:

```
R> cl <- makeCluster(4)
R> clusterSetRNGStream(cl, 4711)
R> mpar1 <- unlist(parLapply(cl, 1 : B, fun))
R> clusterSetRNGStream(cl, 4711)
R> mpar2 <- unlist(parLapply(cl, 1 : B, fun))
R> table(mpar1 == mpar2)
```

```
TRUE
 123
```

```
R> stopCluster(cl)
```

Yes!

Random-number generation

On the other hand, we really need reasonably independent random number streams in the workers.

Worst case: if one sets the random seed in the R startup code (e.g., in a saved workspace), one gets identical streams!

In `parallel`, this is accomplished by using an RNG which makes **streams** with seeds enough steps apart in the RNG stream.

With this RNG (`"L'Ecuyer-CMRG"`) one can then use `nextRNGStream()` to get the seed suitable for the next stream, and can then send this to the next worker.

Good news: `clusterSetRNGStream()` does this for us (for SNOW clusters), and `mclapply()` does this too (by default).

Example 3

Re-consider using the non-parametric bootstrap analysis of the coefficients of the linear regression of Amount on Duration and Job.

When doing things "by hand", bootstrap replications of the coefficients can be obtained via

```
R> fun <- function(b) {
+     n <- nrow(german)
+     i <- sample(1 : n, n, replace = TRUE)
+     coef(lm(Amount ~ Duration + Job, data = german[i, ]))
+ }
```

Example 3

We use $B = 345$ replications.

```
R> B <- 345
```

One at a time:

```
R> system.time(betas <- lapply(1 : B, fun))

   user  system elapsed
  0.569   0.008   0.577
```

Example 3

Note that (unlike `vapply()` or `replicate()` as we used before, `lapply()` does not simplify:

```
R> head(betas, 2)
[[1]]
(Intercept)    Duration     JobA172     JobA173     JobA174
  -787.3019    130.8045   1109.2676   1020.5245   2518.0866

[[2]]
(Intercept)    Duration     JobA172     JobA173     JobA174
  -357.1777    140.3728    470.6099    622.0935   2552.9782
```

Example 3

We can "bind" results into matrices using `rbind()` or `cbind()`:

```
R> betas <- do.call(rbind, betas)
R> head(betas, 2)
```

```
     (Intercept) Duration   JobA172   JobA173  JobA174
[1,]   -787.3019 130.8045 1109.2676 1020.5245 2518.087
[2,]   -357.1777 140.3728  470.6099  622.0935 2552.978
```

Example 3

Everywhere, can parallelize using `parLapply()`.

```
R> cl <- makeCluster(4)
R> clusterSetRNGStream(cl)
R> clusterExport(cl, "german")
R> system.time(betas <- parLapply(cl, 1 : B, fun))

   user  system elapsed 
  0.002   0.000   0.198 

R> stopCluster(cl)
```

(Again, still need to bind the betas into a matrix.)

Example 3

When using the boot package (to take advantage of the available methods for obtaining confidence intervals), we used

```
R> require("boot")
R> statistic <- function(d, i) {
+     coef(lm(Amount ~ Duration + Job, data = d[i, ]))
+ }
R> system.time(betas <- boot(german, statistic, B))

   user  system elapsed 
  0.552   0.000   0.553 
```

Example 3

One can parallelize the above directly (see the vignette for the boot package). However, there is a simpler way:

```
R> args(boot)
```

```
function (data, statistic, R, sim = "ordinary", stype = c("i",
    "f", "w"), strata = rep(1, n), L = NULL, m = 0, weights = NULL,
    ran.gen = function(d, p) d, mle = NULL, simple = FALSE, ...,
    parallel = c("no", "multicore", "snow"), ncpus = getOption("boot.ncpus",
        1L), cl = NULL)
NULL
```

We can ask boot to use a SNOW cluster with the desired number of CPUs/cores, and optionally also provide an already available cluster!

Example 3

```
R> cl <- makeCluster(4)
R> clusterSetRNGStream(cl)
R> system.time(betas <- boot(german, statistic, B,
+                             parallel = "snow", ncpus = 4, cl = cl))

   user  system elapsed 
  0.024   0.000   0.221 

R> stopCluster(cl)
```

Note that we really need to give the number of CPUs/cores: it does not inferred from the cluster we provide.

Summary

If we have 4 (additional) cores available, we can easily make many computations 2-3 times faster, with little additional coding effort.

For at least `boot::boot`, can directly take advantage.

However, not for `boot::cv.glm`.

Parallelization support in other packages discussed is not overwhelming:

- none in `randomForest`
- `gbm(n.cores = M)` will use `parLapply()` on a (new) SNOW cluster with `M` workers for the cross-validations (note that boosting itself is an inherently serial methodology).
- `cv.glmnet(parallel = TRUE)` can use registered parallel backends for the `foreach` package.
- Package caret can use "suitably" registered clusters "under the hood" (again using `foreach`).

Evaluation of binary classifiers

Evaluation of binary classifiers

- A **classification model** (or **classifier**) is a mapping from instances to predicted classes.
- Some classification models produce a continuous output (e.g., an estimate of an instance's class membership probability) to which different thresholds may be applied to predict class membership.
- Other models produce a discrete class label indicating only the predicted class of the instance.
- An important distinction is between metrics that are independent on the prevalence and metrics that depend on the prevalence.

Confusion matrix

		Predicted class		
		$\mathbf{p'}$	$\mathbf{n'}$	**total**
True class	**p**	True Positives	False Negatives	P
	n	False Positives	True Negatives	N
	total	P'	N'	

Common metrics

- True positive rate / Hit rate / Recall / Sensitivity:

$$\frac{\text{True Positives}}{\text{P}}$$

- False positive rate / False alarm rate:

$$\frac{\text{False Positives}}{\text{N}}$$

- Specificity: $1 -$ False positive rate.
- Precision:

$$\frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Common metrics / 2

- Accuracy:

$$\frac{\text{True Positives} + \text{True Negatives}}{\text{P} + \text{N}}$$

- F-measure:

$$\frac{2}{1/\text{Precision} + 1/\text{Recall}}$$

ROC curves

- ROC graphs are two-dimensional with
  - True positive rate on the y-axis,
  - False positive rate on the x-axis.
- A ROC graph depicts relative tradeoffs between benefits (true positives) and costs (false positives).
- Every **discrete** classifier produces an (False positive rate, True positive rate) pair.
- Classifiers appearing on the left-hand side of an ROC graph, near the x-axis, may be thought of as "conservative".
- Classifiers on the upper right-hand side of an ROC graph may be thought of as "liberal".
- The diagonal line $y = x$ represents the strategy of randomly guessing a class.
- Any classifier that appears in the lower right triangle performs worse than random guessing.

ROC curves / 2

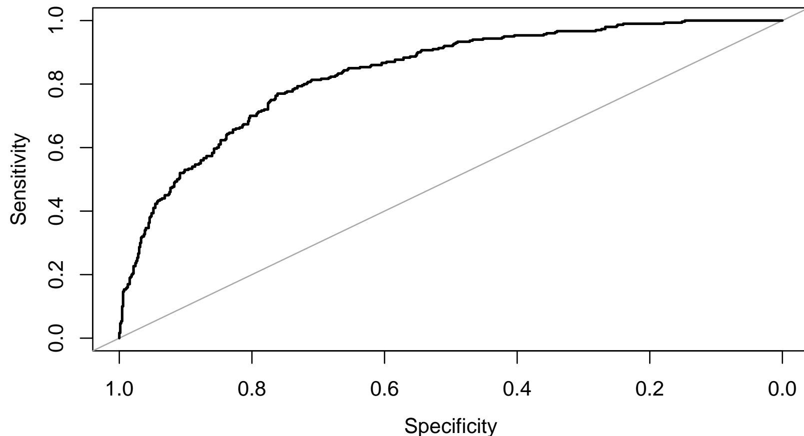
- A **scoring** classifier can be used with a threshold to produce a discrete (binary) classifier. Each threshold value produces a different point in ROC space.
- ROC graphs enable visualizing and organizing classifier performance without regard to class distributions or error costs.
- To compare classifiers ROC performance is reduced to a single scalar value representing expected performance by calculating the area under the ROC curve, abbreviated AUC.
- R packages: e.g.,
  - **pROC**

Example: German Data Set

```
R> load("german.rda")
R> model <- glm(Class ~ ., data = german, family = binomial())

R> library("pROC")
R> ROC <- roc(german$Class, predict(model))
R> auc(ROC)

Area under the curve: 0.8338
```

Example: German Data Set / 2

```
R> plot(r)
```



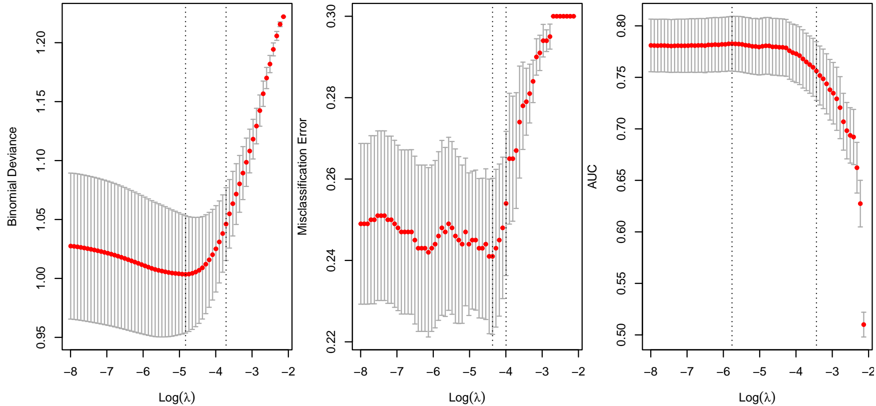
Example: German Data Set / 3

```
R> library("glmnet")
R> categorical <- c("Status_of_checking_account", "History", "Purpose",
+   "Savings", "Status_and_sex", "Other_debtors_or_guarantors", "Property",
+   "Other_installment_plans", "Housing", "Job", "Phone", "Foreign")
R> for (col in categorical) {
+   german[[col]] <- relevel(german[[col]],
+     names(which.max(table(german[[col]]))))
+ }
R> german[["Employment_since"]] <- ordered(german[["Employment_since"]])
R> mf <- model.frame(Class ~ ., data = german)
R> X <- model.matrix(Class ~ ., data = mf,
+   contrasts = list(Employment_since = MASS::contr.sdif))[, -1]
R> y <- model.response(mf)
R> foldid <- integer(nrow(german))
R> foldid[german$Class == "good"] <-
+   sample(rep(1:10, length.out = sum(german$Class == "good")))
R> foldid[german$Class == "bad"] <-
+   sample(rep(1:10, length.out = sum(german$Class == "bad")))
```

Example: German Data Set / 4

```
R> cv.model <- cv.glmnet(X, y, family = "binomial", foldid = foldid)
R> cv.model.class <- cv.glmnet(X, y, family = "binomial", foldid = foldid,
+    type.measure = "class")
R> cv.model.auc <- cv.glmnet(X, y, family = "binomial", foldid = foldid,
+    type.measure = "auc")

R> op <- par(mfrow = c(1, 3))
R> plot(cv.model)
R> plot(cv.model.class)
R> plot(cv.model.auc)
R> par(op)
```

Example: German Data Set / 5

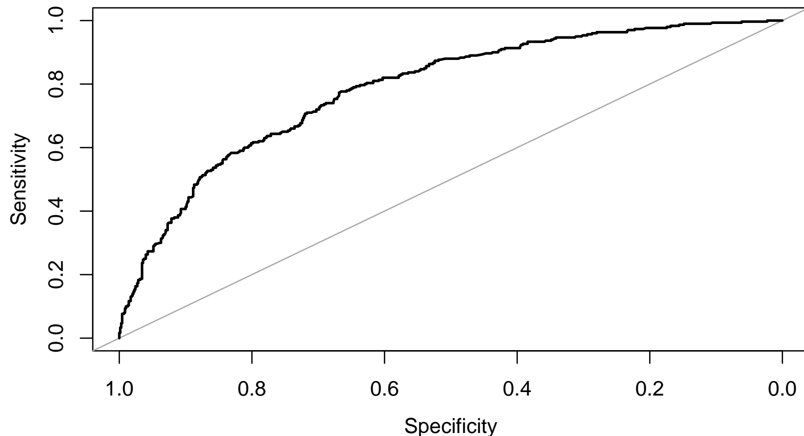


Example: German Data Set / 6

```
R> ROC <- roc(german$Class, as.numeric(predict(cv.model.auc, newx = X)))
R> auc(ROC)

Area under the curve: 0.7862
```

Example: German Data Set / 7

```
R> plot(ROC, asp = NA)
```



Case Study

Introduction

The `firms` data set contains financial ratios, stock price variables, default information as well as ratings and mapped PD scores for S&P.

The sample contains information on 3011 publicly traded US firms over the period 1985–2014.

Download for data ('firms.RData') and docs ('firms.pdf') via `https://statmath.wu.ac.at/~hornik/DTM`.

First steps

Getting started

We start by loading the data set:

```
R> load("firms.RData")
```

This gives a data frame of size:

```
R> dim(firms)
```

```
[1] 26620    62
```

Variables

The data set has the following variables:

```
R> names(firms)
 [1] "gvkey"        "conml"        "gsector"      "sic"          "datadate"
 [6] "datadate_eoy" "R1"           "R2"           "R3"           "R4"
[11] "R5"           "R6"           "R7"           "R7M"          "R8.Altman1"
[16] "R9"           "R10"          "R11"          "R11M"         "R12"
[21] "R13"          "R14"          "R15"          "R16"          "R16M.Altman4"
[26] "R17"          "R17M"         "R18"          "R18a"         "R18b"
[31] "R19"          "R20.Altman2"  "R21"          "R21a.Altman3" "R22"
[36] "R22M"         "R23"          "R24"          "R25"          "R26"
[41] "R27"          "R28.Altman5"  "R29"          "R30"          "R31"
[46] "R32"          "R33"          "R34"          "R35"          "R36"
[51] "DIVPAYER"     "lAT"          "lSALE"        "MB"           "SIGMA"
[56] "BETA"         "EXRET"        "RSIZE"        "SPR"          "SPR21"
[61] "PDscore.SPR"  "default"
```

Variables

Variables gvkey and conml give the (COMPUSTAT) id and legal name.

Variable gsector gives a two-digit GICS sector code (https://en.wikipedia.org/wiki/Global_Industry_Classification_Standard).

Variable sic gives the SIC code (https://en.wikipedia.org/wiki/Standard_Industrial_Classification).

```
R> firms[1 : 5, 1 : 4]

   gvkey                       conml gsector  sic
40 10000 Standard Motor Products Inc.      25 3690
41 10000 Standard Motor Products Inc.      25 3690
42 10000 Standard Motor Products Inc.      25 3690
43 10000 Standard Motor Products Inc.      25 3690
44 10000 Standard Motor Products Inc.      25 3690
```

Variables

Variables `R1` to `R36` are the "usual" financial ratios, including the 5 Altman variables.

Accordings to the docs, a ratio is set to zero if the denominator was close to zero, and all ratios are winsorized at 5%.

Variables `lAT` and `lSALE` are log total assets and total sales.

Variables `MB` to `RSIZE` are market variables.

Variables

Variables SPR and SPR21 are long term end of year ratings with coarsened and original granularities.

```
R> with(firms, table(SPR))

SPR
CCC/C     B    BB   BBB     A    AA   AAA
  469  4986  6597  6759  4900  1375   369

R> with(firms, table(SPR21))

SPR21
   C   CC CCC-  CCC CCC+   B-    B   B+  BB-   BB  BB+ BBB-  BBB BBB+   A-    A
   0   40   36  127  266  754 1497 2735 2832 2196 1569 2127 2664 1968 1692 2020
  A+  AA-   AA  AA+  AAA
1188  646  591  138  369
```

(Note that ratings should really be provided as **ordered** factors.)

Variables

Variable `PDscore.SPR` is a mapped logit PD score for the coarsened rating (more later).

Variable `default` is `"yes"` whenever the company files for Chapter 7 or 11 under the US Bankruptcy law or whenever at least one rating agency assigns a default rating.

```
R> with(firms, table(default))

default
   no   yes 
26182   438 

R> prop.table(with(firms, table(default)))

default
        no        yes 
0.98354621 0.01645379 
```

So definitely low-default.

Variables

Variable datadate gives the calendar date of the financial statement.

```
R> with(firms, head(datadate))
```

```
[1] "1999-12-31" "2000-12-31" "2001-12-31" "2002-12-31" "2003-12-31"
[6] "2004-12-31"
```

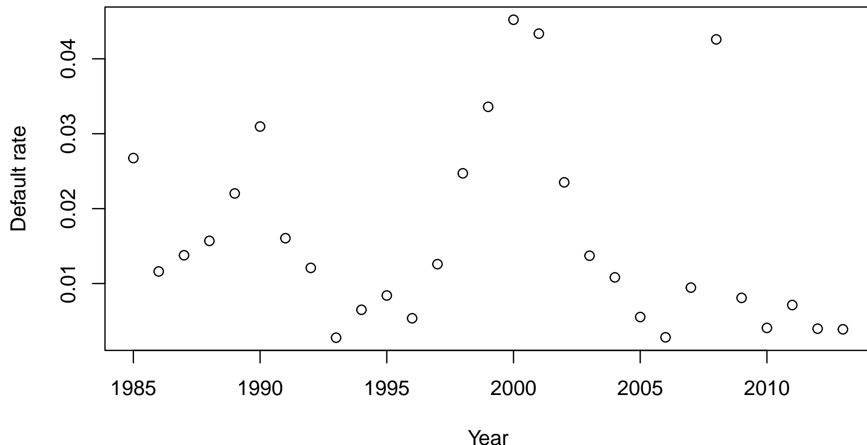
Let us extract the **year** from this and add it to the data set:

```
R> year <- with(firms, as.numeric(format(as.Date(datadate), "%Y")))
R> firms <- cbind(firms, year = year)
```

Variables

Note that the default rates vary markedly with year:

```
R> tab <- with(firms, prop.table(table(year, default), 1L))
R> plot(rownames(tab), tab[, 2L], xlab = "Year", ylab = "Default rate")
```



Variables

The data has the "usual" firm-year unbalanced panel structure:

```
R> year_by_gvkey <- with(firms, split(year, gvkey))
R> head(year_by_gvkey, 4)

$`1004`
 [1] 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001
[16] 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013

$`1020`
[1] 1985 1986 1987

$`1023`
[1] 1986 1987

$`1034`
 [1] 1989 1990 1991 1998 1999 2000 2001 2002 2003 2004 2005 2007
```

Variables

```
R> n_of_years_by_gvkey <- lengths(year_by_gvkey)
R> table(n_of_years_by_gvkey)
```

```
n_of_years_by_gvkey
  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20 
321 335 287 235 168 167 158 130  91  88  76  75  82  86  85  92  51  43  39  32 
 21  22  23  24  25  26  27  28  29 
 32  30  25  21  13  11  31  46 111 
```

Task

Discuss:

- What are the implications of this panel structure for (traditional) statistical modeling?

Task

Discuss:

- What are the implications of this panel structure for (traditional) statistical modeling?
- What are the implications of this panel structure for (modern) statistical learning, in particular estimating prediction error?

Challenges

- Data is definitely **not** i.i.d. Is that a problem?

Challenges

- Data is definitely **not** i.i.d. Is that a problem?
- Do we explicitly need to model longitudinal dependencies?

Challenges

- Data is definitely **not** i.i.d. Is that a problem?
- Do we explicitly need to model longitudinal dependencies?
- Do we explicitly need to model cross-sectional dependencies (business cycles)?

Challenges

- Data is definitely **not** i.i.d. Is that a problem?
- Do we explicitly need to model longitudinal dependencies?
- Do we explicitly need to model cross-sectional dependencies (business cycles)?
- How could/should the panel structure impact drawing folds for cross-validation exercises? Should we sample from firms or from firm-years?

Challenges

- Data is definitely **not** i.i.d. Is that a problem?
- Do we explicitly need to model longitudinal dependencies?
- Do we explicitly need to model cross-sectional dependencies (business cycles)?
- How could/should the panel structure impact drawing folds for cross-validation exercises? Should we sample from firms or from firm-years?
- Is there a selection bias so that firms with more observations have a higher chance to default?

Variables

To investigate the last:

```
R> default_by_gvkey <- with(firms, split(default == "yes", gvkey))
```

This gives a series of default indicators for each firm.

To compute the number of defaults for each firm:

```
R> n_of_defaults_by_gvkey <- vapply(default_by_gvkey, sum, 0)
R> table(n_of_defaults_by_gvkey)
```

```
n_of_defaults_by_gvkey
   0    1    2    3    4    5
2600  301   47   10    2    1
```

Variables

Do we really have firms with several (up to five) defaults in the data?

```
R> default_by_gvkey[ n_of_defaults_by_gvkey >= 4 ]
```

```
$`1990`
 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE

$`3587`
[1] TRUE TRUE TRUE TRUE

$`25767`
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Variables

Which firms are these?

```
R> gvkeys <- names(default_by_gvkey[ n_of_defaults_by_gvkey >= 4 ])
R> unique(subset(firms, gvkey %in% gvkeys, 1 : 2))

        gvkey                         conml
201105   1990 Bally Entertainment Corp
246367 25767                 Bradlees Inc
302229   3587             Crazy Eddie Inc
```

Variables

Which firm-years are these?

```
R> year_by_gvkey[ n_of_defaults_by_gvkey >= 4 ]
$`1990`
 [1] 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995

$`3587`
[1] 1987 1988 1989 1990

$`25767`
[1] 1993 1994 1995 1996 1997 1998
```

Task

Discuss:

- How should we handle firms with several defaults?

Task

Discuss:

- How should we handle firms with several defaults?
- How should we handle firms with consecutive defaults?

Variables

Now back to the original questions: are defaults more likely for firms with more firm-year observations?

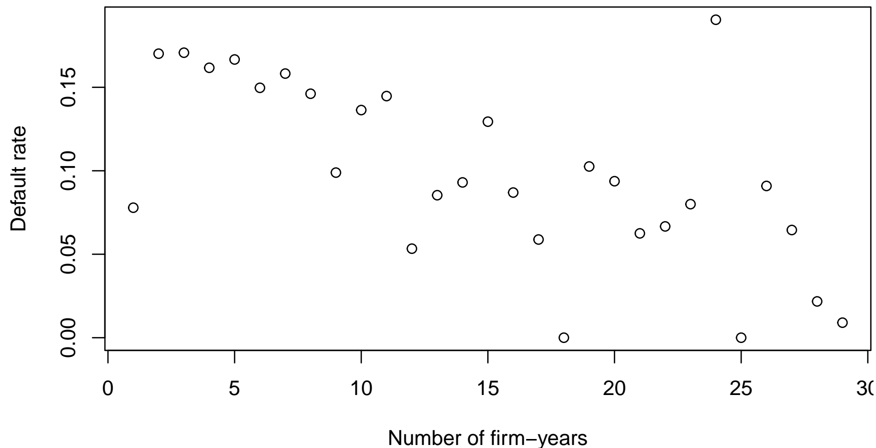
```
R> gvkey_has_any_default <- (n_of_defaults_by_gvkey > 0)
R> tab <- table(n_of_years_by_gvkey, gvkey_has_any_default)
R> prop.table(tab, 1L)[, 2L]
```

```
          1           2           3           4           5           6 
0.077881620 0.170149254 0.170731707 0.161702128 0.166666667 0.149700599 
          7           8           9          10          11          12 
0.158227848 0.146153846 0.098901099 0.136363636 0.144736842 0.053333333 
         13          14          15          16          17          18 
0.085365854 0.093023256 0.129411765 0.086956522 0.058823529 0.000000000 
         19          20          21          22          23          24 
0.102564103 0.093750000 0.062500000 0.066666667 0.080000000 0.190476190 
         25          26          27          28          29 
0.000000000 0.090909091 0.064516129 0.021739130 0.009009009 
```

Variables

Plot of default rates according to number of firm-years:

```
R> plot(prop.table(tab, 1L)[, 2L],
+       xlab = "Number of firm-years", ylab = "Default rate")
```

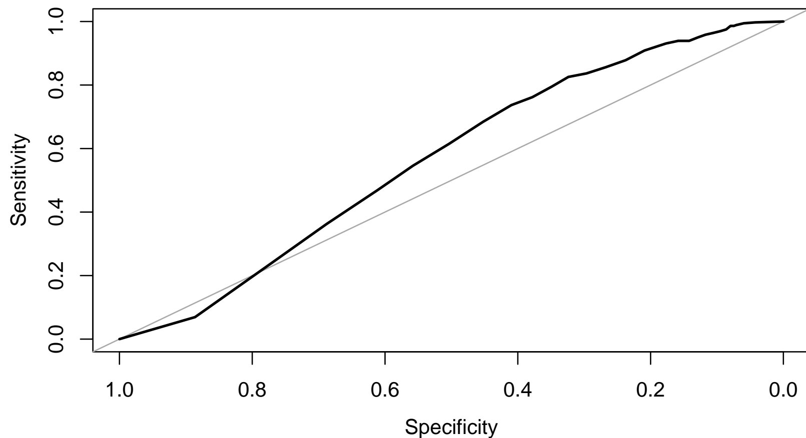


Variables

```
R> r <- pROC::roc(gvkey_has_any_default, n_of_years_by_gvkey)
R> pROC::auc(r)

Area under the curve: 0.5677
```

Variables

```
R> plot(r)
```



Task

Discuss:

- Can we really use all firm-years?

Task

Discuss:

- Can we really use all firm-years?
- Alternatively, should we down-sample to use only one year for each firm?

Modeling setup

Challenge

To find the "best" model, we would ideally have a **validation set** to compare model performance and select the best model (note that we would need another suitably independent **test set** to estimate the performance of the chosen "best" model).

Remember cross-validation from last year's course.

Note the "correct" use of terminology for training, validation and test data.

Note also
`https://en.wikipedia.org/wiki/Training,_validation,_and_test_sets`, in particular the section on "Confusion in terminology"!

Task

Discuss:

- How should we split our data into these subsets (or folds for cross-validation)?

Task

Discuss:

- How should we split our data into these subsets (or folds for cross-validation)?
- Specifically, should we split according to firm or firm-year?

Task

Discuss:

- How should we split our data into these subsets (or folds for cross-validation)?
- Specifically, should we split according to firm or firm-year?
- Should splitting try to maintain default rates, and what precisely are these?

Setup

For simplicity, let us use a single training/validation split. How?

One common idea: use the last year(s) ... clearly not appropriate here. Why?

Let us simply sample from the firm-years (e.g., 80/20), controlling the firm-year default rates.

Setup

Re-using code from earlier units:

```
R> source("folds.R")
R> folds

function (n, k, f = NULL)
{
    if (is.null(f))
        ids <- sample(rep(1:k, length.out = n))
    else {
        ids <- integer(n)
        for (l in levels(f)) {
            ind <- (f == l)
            ids[ind] <- sample(rep(1:k, length.out = sum(ind)))
        }
    }
    split(1:n, ids)
}
```

Setup

Create 5 folds, and use the last for validation:

```
R> set.seed(4711)
R> ids <- folds(NROW(firms), 5, firms$default)
```

This has 5 folds with ids:

```
R> lengths(ids)

   1    2    3    4    5
5325 5325 5324 5323 5323
```

Create an indicator for the ids in the last fold (validation set):

```
R> valid <- seq_len(NROW(firms)) %in% ids[[length(ids)]]
```

Everything else goes into the training set:

```
R> train <- !valid
```

Setup

Using positions/indices (as in the previous course) is more efficient.

Using logical values as above often more convenient:

```
R> (tab <- with(firms, table(train, default)))

       default
train      no   yes
  FALSE  5236    87
  TRUE  20946   351

R> prop.table(tab, 1L)

       default
train           no        yes
  FALSE 0.98365583 0.01634417
  TRUE  0.98351881 0.01648119
```

Modeling defaults

Lab

Try some default prediction models from Course 1, using only the financial ratios, and report their performance in the training and validation sets.

Setup

Financial ratios start with R1 in column 7 and go up to R36 in column 50.

Could create a new data set having only these columns and `default`, and then use `default ~ .` for modeling.

Better to programmatically create formulas and use the full data set.

The ratios are the variables whose names start with 'R'. So get all these names:

```
R> nms <- grep("^R[0-9]", names(firms), value = TRUE)
```

Setup

And use these for the RHS of the model formula:

```
R> fml <- as.formula(paste("default ~",
+                          paste(nms,  collapse = "+")))
R> fml
```

```
default ~ R1 + R2 + R3 + R4 + R5 + R6 + R7 + R7M + R8.Altman1 +
    R9 + R10 + R11 + R11M + R12 + R13 + R14 + R15 + R16 + R16M.Altman4 +
    R17 + R17M + R18 + R18a + R18b + R19 + R20.Altman2 + R21 +
    R21a.Altman3 + R22 + R22M + R23 + R24 + R25 + R26 + R27 +
    R28.Altman5 + R29 + R30 + R31 + R32 + R33 + R34 + R35 + R36
```

Magic . . .

Simple logit model

```
R> m1_logit <- glm(fml,
+                  data = firms[train, ],
+                  family = binomial())
```

Note that many ratios are actually not significant:

```
R> summary(m1_logit)
```

Compute in-sample confusion matrix as in Course 1:

```
R> with(firms[train, ], table(default, round(fitted(m1_logit))))
```

```
default     0     1
    no  20912    34
    yes   278    73
```

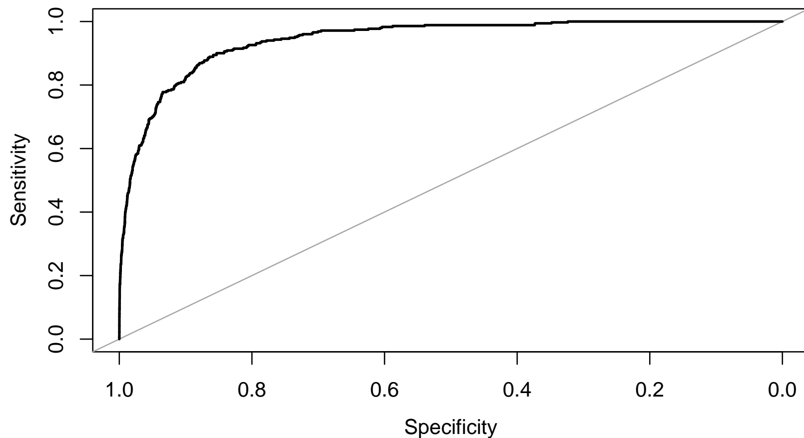
This seems **very** disappointing. Let us look at the fitted PDs more closely:

Simple logit model

```
R> pds <- fitted(m1_logit)
R> r <- pROC::roc(firms$default[train], pds)
R> pROC::auc(r)

Area under the curve: 0.9433
```

Simple logit model

```
R> plot(r)
```



Simple logit model

Of course, we should really evaluate performance on the validation set:

```
R> pds <- predict(m1_logit, firms[valid, ], type = "response")
R> with(firms[valid, ], table(default, round(pds)))
```

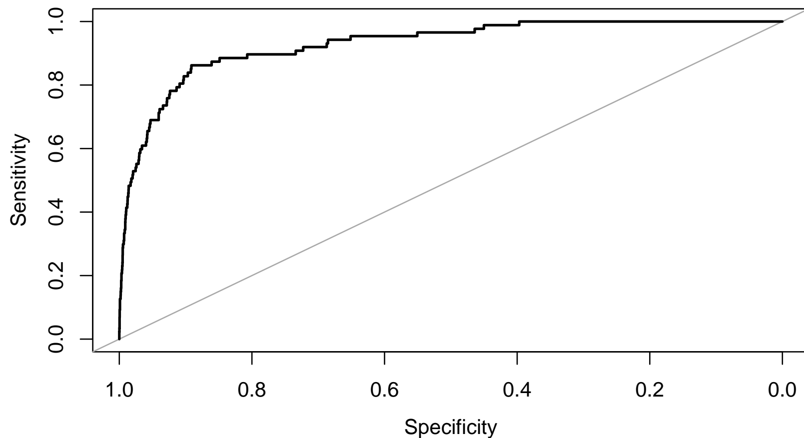
```
default    0    1
     no 5221   15
     yes  71   16
```

```
R> r <- pROC::roc(firms$default[valid], pds)
R> pROC::auc(r)
```

```
Area under the curve: 0.9289
```

Simple logit model

```
R> plot(r)
```



Small logit model

Maybe we should drop non-significant predictors?

Inspect first:

```
R> summary(m1_logit)
```

What we learned previously is "step-wise logistic regression", here via step-wise selection starting from the full model:

```
R> m1_logit_step <- step(m1_logit)
```

However, this takes rather long (although nowhere near "big data").

Small logit model

Alternatively, could be fancy and only keep significant predictors:

```
R> mat <- summary(m1_logit)$coefficients
R> (mat <- mat[mat[, 4L] <= 0.05, ])

               Estimate  Std. Error   z value     Pr(>|z|)
(Intercept)  -6.58445631 1.994814237 -3.300787 9.641415e-04
R1           17.12189083 4.069658131  4.207206 2.585472e-05
R2           -0.26255507 0.054538688 -4.814107 1.478597e-06
R7M          -6.59205650 2.616165883 -2.519739 1.174417e-02
R10           1.80205865 0.508326986  3.545078 3.924974e-04
R13           0.02348775 0.006713825  3.498415 4.680325e-04
R16          -1.60619133 0.562098665 -2.857490 4.270060e-03
R16M.Altman4  0.24737443 0.095929221  2.578718 9.916763e-03
R18a          1.27437044 0.648935955  1.963785 4.955508e-02
R22M         -5.91178813 1.847915041 -3.199167 1.378255e-03
R25           1.09405294 0.328508479  3.330364 8.673240e-04
R27          -0.04661052 0.022804677 -2.043902 4.096326e-02
```

Small logit model

Programmatically contruct the model formula:

```
R> fml_small <- as.formula(paste("default ~",
+                                paste(rownames(mat)[-1L], collapse = "+")))
R> fml_small
```

```
default ~ R1 + R2 + R7M + R10 + R13 + R16 + R16M.Altman4 + R18a +
    R22M + R25 + R27
```

Now re-fit a logit model with only the significant variables:

```
R> m1_logit_small <- glm(fml_small,
+                        data = firms[train, ],
+                        family = binomial())
```

```
R> summary(m1_logit_small)
```

Small logit model

```
R> summary(m1_logit_small)$coefficients
```

```
                Estimate  Std. Error    z value     Pr(>|z|)
(Intercept)  -3.70869315 0.249816462 -14.845672 7.420794e-50
R1           13.26873908 2.581603787   5.139727 2.751377e-07
R2           -0.25654844 0.030347328  -8.453741 2.821157e-17
R7M          -4.06197730 1.032846367  -3.932799 8.396239e-05
R10           0.45396979 0.316262166   1.435422 1.511668e-01
R13           0.02163866 0.006301834   3.433708 5.953843e-04
R16          -1.07958240 0.255611995  -4.223520 2.405159e-05
R16M.Altman4 -0.42464722 0.136556825  -3.109674 1.872939e-03
R18a          2.54084936 0.227334947  11.176677 5.303873e-29
R22M         -5.57633007 0.736744346  -7.568881 3.764536e-14
R25           0.33800520 0.195906090   1.725343 8.446568e-02
R27          -0.03902263 0.016495887  -2.365597 1.800100e-02
```

Small logit model

Evaluate performance on the validation sample:

```
R> pds <- predict(m1_logit_small, firms[valid, ], type = "response")
R> with(firms[valid, ], table(default, round(pds)))
```

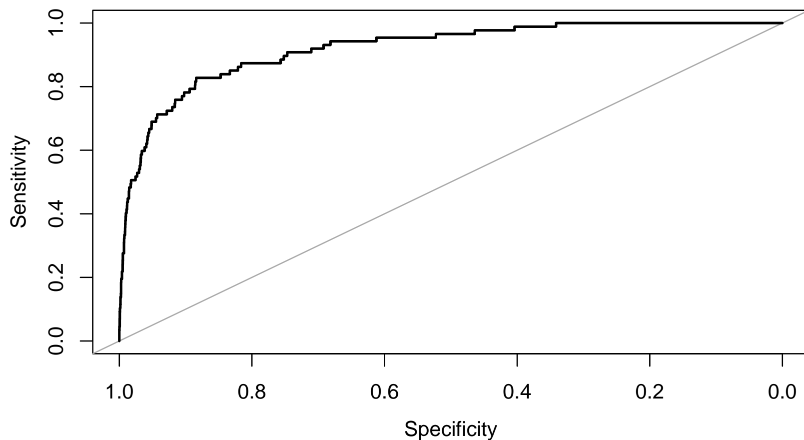
```
default    0    1
    no  5223   13
    yes   75   12
```

```
R> r <- pROC::roc(firms$default[valid], pds)
R> pROC::auc(r)
```

```
Area under the curve: 0.9202
```

Small logit model

```
R> plot(r)
```



Logit model with LASSO

```
R> X <- as.matrix(firms[train, nms])
R> y <- firms[train, "default"]
R> m1_glmnet <- glmnet::cv.glmnet(X, y, family = "binomial")
R> m1_glmnet

Call:  glmnet::cv.glmnet(x = X, y = y, family = "binomial")

Measure: Binomial Deviance

       Lambda Index Measure       SE Nonzero
min 0.0001419    60  0.1058 0.004299      35
1se 0.0027855    28  0.1099 0.004297      13
```

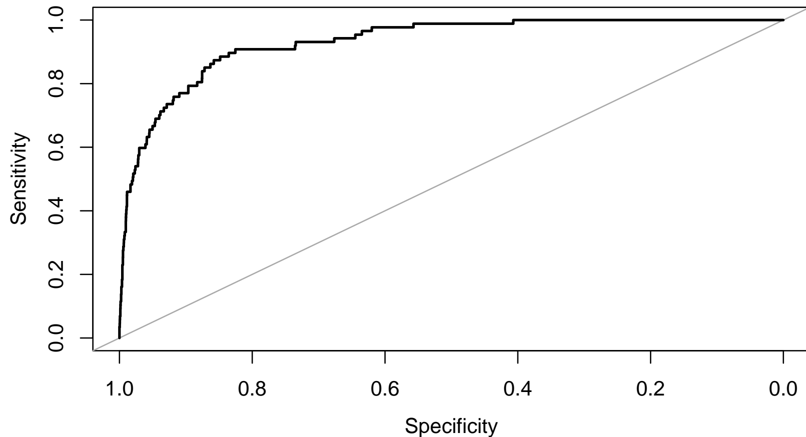
Logit model with LASSO

Evaluate performance on the validation sample:

```
R> pds <- predict(m1_glmnet, newx = as.matrix(firms[valid, nms]),
+                  type = "response")[, 1L]
R> with(firms[valid, ], table(default, round(pds)))

default    0    1
    no  5227    9
    yes   77   10

R> r <- pROC::roc(firms$default[valid], pds)
R> pROC::auc(r)

Area under the curve: 0.9314
```

Logit model with LASSO

```
R> plot(r)
```



Basic Altman model

Do a logit model with only the Altman ratios.

```
R> nms <- grep("Altman", names(firms), value = TRUE)
R> fml_Altman <- as.formula(paste("default ~",
+                                 paste(nms, collapse = "+")))
R> m1_logit_Altman <- glm(fml_Altman,
+                         data = firms[train, ],
+                         family = binomial())
```

Basic Altman model

```
R> summary(m1_logit_Altman)

Call:
glm(formula = fml_Altman, family = binomial(), data = firms[train,
    ])

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.6935  -0.1684  -0.0966  -0.0406   6.7773

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)  -2.92034    0.11816 -24.716  < 2e-16 ***
R8.Altman1   -3.96735    0.36895 -10.753  < 2e-16 ***
R16M.Altman4 -1.25568    0.13545  -9.270  < 2e-16 ***
R20.Altman2  -1.01473    0.10392  -9.764  < 2e-16 ***
R21a.Altman3 -8.80190    0.60229 -14.614  < 2e-16 ***
R28.Altman5   0.30022    0.07747   3.875 0.000106 ***
```

Basic Altman model / 2

```
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 3578.3  on 21296  degrees of freedom
Residual deviance: 2563.8  on 21291  degrees of freedom
AIC: 2575.8

Number of Fisher Scoring iterations: 9
```

Basic Altman model

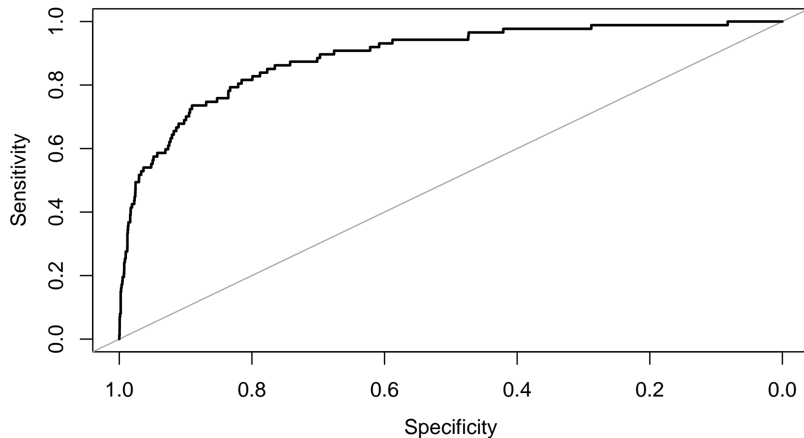
Evaluate performance on the validation sample:

```
R> pds <- predict(m1_logit_Altman, firms[valid, ], type = "response")
R> with(firms[valid, ], table(default, round(pds)))

default    0    1
    no  5227    9
    yes   80    7

R> r <- pROC::roc(firms$default[valid], pds)
R> pROC::auc(r)

Area under the curve: 0.8903
```

Ah. Finally something that does not work so well ...

Basic Altman model

```
R> plot(r)
```



Random forests

The standard `randomForest` package from the last course takes some time.

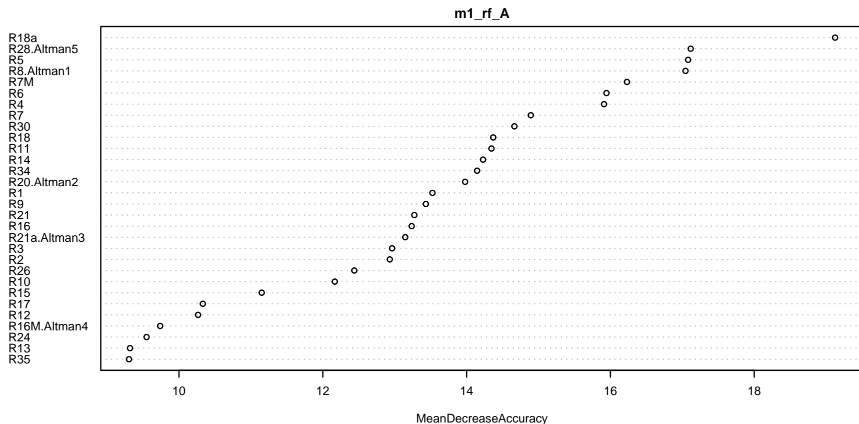
```
R> system.time(m1_rf_A <-
+                   randomForest::randomForest(fml,
+                                              data = firms[train, ],
+                                              importance = TRUE))
```

Random forests

```
R> m1_rf_A

Call:
 randomForest(formula = fml, data = firms[train, ], importance = TRUE)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 6

        OOB estimate of  error rate: 1.41%
Confusion matrix:
       no yes class.error
no  20924  22  0.00105032
yes   278  73  0.79202279
```

Random forests

```
R> randomForest::varImpPlot(m1_rf_A, type = 1)
```



Random forests

Evaluate performance on the validation sample:

```
R> pds <- predict(m1_rf_A, firms[valid, ], type = "prob")[, 2L]
R> with(firms[valid, ], table(default, round(pds)))
```

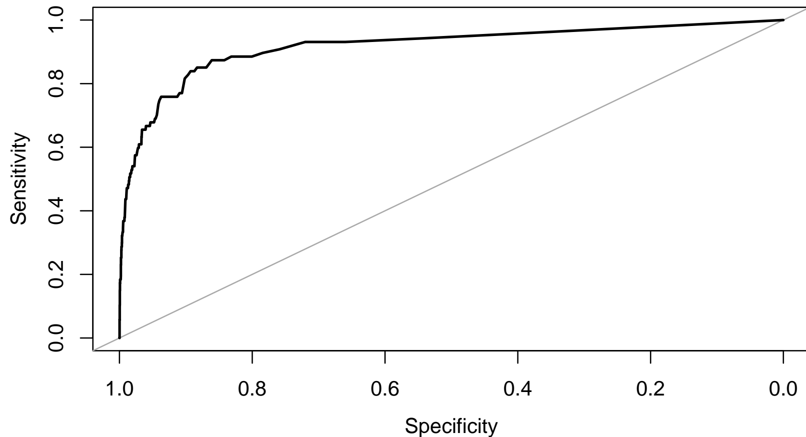
```
default    0    1
    no  5231    5
    yes   71   16
```

```
R> r <- pROC::roc(firms$default[valid], pds)
R> pROC::auc(r)
```

```
Area under the curve: 0.9168
```

Random forests

```
R> plot(r)
```



Random forests

Package ranger provides faster alternatives which can use several cores in parallel.

```
R> system.time(m1_rf_B <- ranger::ranger(fml,
+                                        data = firms[train, ],
+                                        importance = "impurity",
+                                        probability = TRUE))
```

Fast . . . apparently using all 8 cores on my system.

Random forests

```
R> m1_rf_B

Ranger result

Call:
 ranger::ranger(fml, data = firms[train, ], importance = "impurity",      probabili

Type:                             Probability estimation
Number of trees:                  500
Sample size:                      21297
Number of independent variables:  44
Mtry:                             6
Target node size:                 10
Variable importance mode:         impurity
Splitrule:                        gini
OOB prediction error (Brier s.):  0.01206362
```

Random forests

Can also provide variable importance:

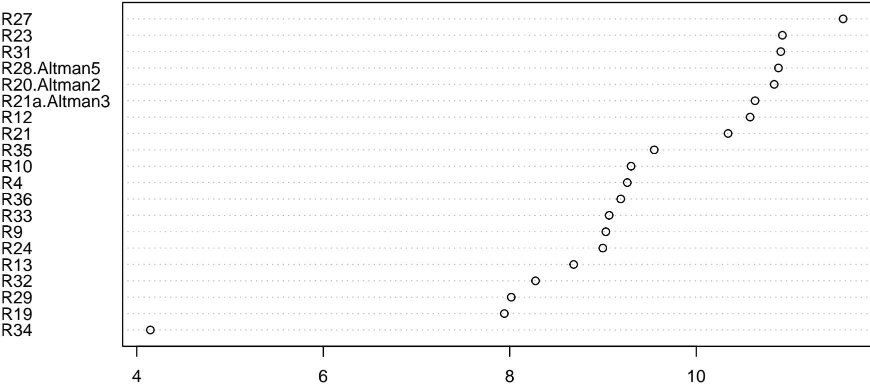
```
R> (imp <- ranger::importance(m1_rf_B))
```

```
          R1           R2           R3           R4           R5           R6 
   14.031547    17.073266    12.435896     9.260534    20.618364    14.900651 
          R7          R7M  R8.Altman1           R9          R10          R11 
   11.859824    11.982345    29.478401     9.030460     9.301042    13.307293 
        R11M          R12          R13          R14          R15          R16 
   13.293846    10.577619     8.685373    15.074470    12.855827    15.165414 
R16M.Altman4          R17         R17M          R18         R18a         R18b 
   12.664578    12.585375    13.457419    19.311803    24.672984    16.098930 
         R19  R20.Altman2          R21 R21a.Altman3          R22         R22M 
    7.941938    10.835930    10.341360    10.630763    21.201429    20.795032 
         R23          R24          R25          R26          R27  R28.Altman5 
   10.923932     8.997807    12.951479    12.604321    11.575063    10.881414 
         R29          R30          R31          R32          R33          R34 
    8.016126    12.024589    10.906699     8.276212     9.066276     4.145725 
         R35          R36 
    9.549302     9.190361 
```

But apparently no cool plot, so let's do by hand:

Random forests

```
R> dotchart(sort(imp))
```



Random forests

Evaluate performance on the validation sample:

Note that getting PD predictions is a bit fancy: we needed to fit a “probability forest” above, and now need:

```
R> pds <- predict(m1_rf_B, firms[valid, ])$predictions[, 2L]
R> with(firms[valid, ], table(default, round(pds)))
```

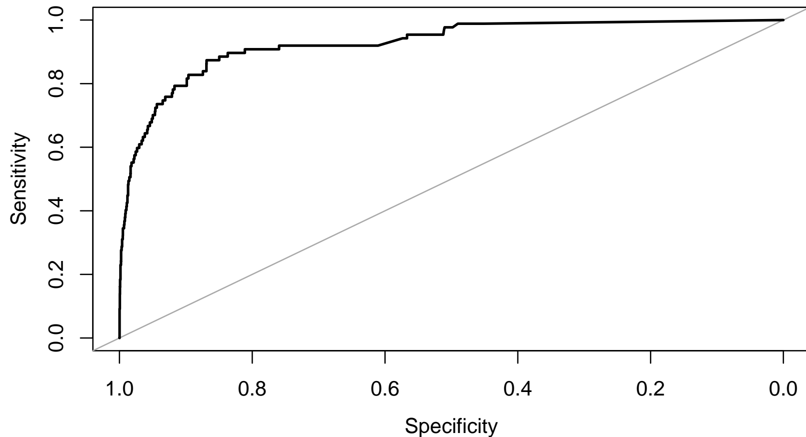
```
default    0    1
     no 5231    5
    yes   71   16
```

```
R> r <- pROC::roc(firms$default[valid], pds)
R> pROC::auc(r)
```

```
Area under the curve: 0.927
```

Random forests

```
R> plot(r)
```



Modeling ratings

Motivation

A common alternative for PD modeling in low default portfolios is building a **shadow rating** of available rating information.

E.g., ratings from at least one of the big three rating agencies.

Task

Discuss:

- What are the (possible) pros and cons of doing this?

Task

Discuss:

- What are the (possible) pros and cons of doing this?
- How can this be done from a modeling perspective?

Task

Discuss:

- What are the (possible) pros and cons of doing this?
- How can this be done from a modeling perspective?
- What if not everyone in the portfolio has a rating?

Task

Discuss:

- What are the (possible) pros and cons of doing this?
- How can this be done from a modeling perspective?
- What if not everyone in the portfolio has a rating?
- Are there other commonly available measures of creditworthiness (default risk) which could be used for modeling (instead of defaults)?

Issues

- Ratings from CRAs are on an ordinal scale, not necessarily for PDs, and typically through the cycle and not point in time.

Issues

- Ratings from CRAs are on an ordinal scale, not necessarily for PDs, and typically through the cycle and not point in time.
- There are ordered probit/logit models, but these are somewhat tricky.

Issues

- Ratings from CRAs are on an ordinal scale, not necessarily for PDs, and typically through the cycle and not point in time.
- There are ordered probit/logit models, but these are somewhat tricky.
- Perhaps one could transform/map the ordinal ratings to corresponding PD scores?

Issues

- Ratings from CRAs are on an ordinal scale, not necessarily for PDs, and typically through the cycle and not point in time.
- There are ordered probit/logit models, but these are somewhat tricky.
- Perhaps one could transform/map the ordinal ratings to corresponding PD scores?
- Of course, ideally we would use **both** rating and default infos.

PD scores

In our data set, we have a very simple mapping of the coarsened S&P ratings to PD scores via logistic regression.

```
R> with(firms, summary(PDscore.SPR))

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
-12.320  -8.635  -6.793  -6.126  -4.951  -1.266    1165
```

Note that not everyone in the data set has a rating, and that we really only have one global PD (score) for each coarse grade:

```
R> with(firms, table(PDscore.SPR))

PDscore.SPR
 -12.320145207672 -10.4777625549668 -8.63537990226158 -6.79299724955637
              369              1375              4900              6759
-4.95061459685117 -3.10823194414596 -1.26584929144076
             6597              4986               469
```

PD scores

The latter should really be improved. How?

The former implies we need to drop the firm-years with no rating when fitting models for the PD scores on the training data.

```
R> with(firms, table(train, is.na(PDscore.SPR)))
```

```
train   FALSE  TRUE
  FALSE  5077   246
  TRUE  20378   919
```

PD scores

Set things up for modeling `PDscore.SPR` in terms of the financial ratios.

```
R> nms <- grep("^R[0-9]", names(firms), value = TRUE)
R> fml <- as.formula(paste("PDscore.SPR ~",
+                          paste(nms, collapse = "+")))
```

Linear model

```
R> m2_lm_A <- lm(fml, data = firms[train, ])
```

However: this has used many coefficients to learn 7 numbers:

```
R> summary(m2_lm_A)
```

Linear model

To evaluate performance on the validation sample, need to map the predictions for the PD scores to PDs:

```
R> pds <- plogis(predict(m2_lm_A, firms[valid, ]))
R> with(firms[valid, ], table(default, round(pds)))
```

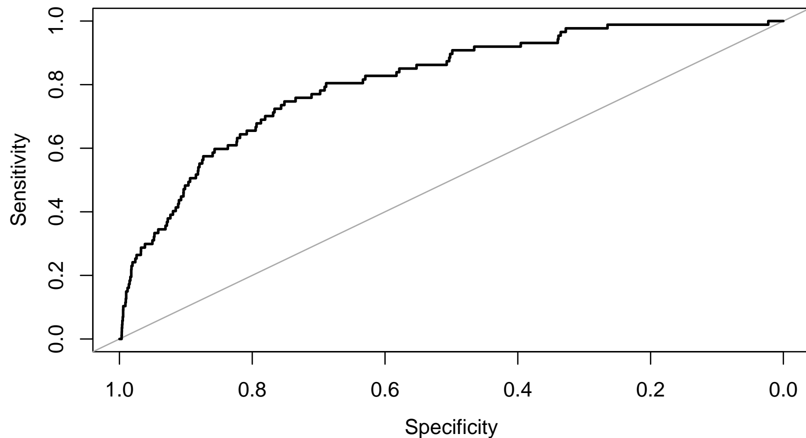
```
default    0    1
    no  5219   17
    yes   86    1
```

```
R> r <- pROC::roc(firms$default[valid], pds)
R> pROC::auc(r)
```

```
Area under the curve: 0.8103
```

Linear model

```
R> plot(r)
```



Linear model

So with the current mapping from ratings to PDs, this really does not work well.

One idea could be reducing the discreteness of the response variable which results in overfitting by “jittering”:

```
R> (delta <- diff(sort(unique(firms$PDscore.SPR))))
```

```
[1] 1.842383 1.842383 1.842383 1.842383 1.842383 1.842383
```

```
R> width <- delta[1L]
R> firms1 <- firms
R> set.seed(4711)
R> firms1$PDscore.SPR <-
+     firms$PDscore.SPR + runif(NROW(firms), - width / 2, width / 2)
R> ## Or use transform() ...
R> m2_lm_B <- lm(fml, data = firms1[train, ])
```

Linear model

However, this still does not work much better:

```
R> pds <- plogis(predict(m2_lm_B, firms[valid, ]))
R> with(firms[valid, ], table(default, round(pds)))
```

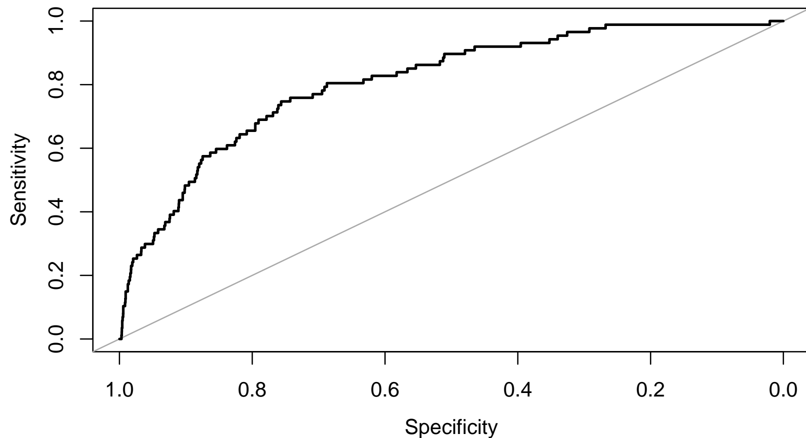
```
default    0    1
     no 5219   17
     yes  86    1
```

```
R> r <- pROC::roc(firms$default[valid], pds)
R> pROC::auc(r)
```

```
Area under the curve: 0.81
```

Linear model

```
R> plot(r)
```



Random forest

Let us try a random forest instead.

Note that this does not automatically handle missings, so we need a reduced **training** data set (missing PD scores in the validation data are not a problem):

```
R> train2 <- train & !is.na(firms$PDscore.SPR)
```

```
R> m2_rf <- ranger::ranger(fml, data = firms[train2, ])
```

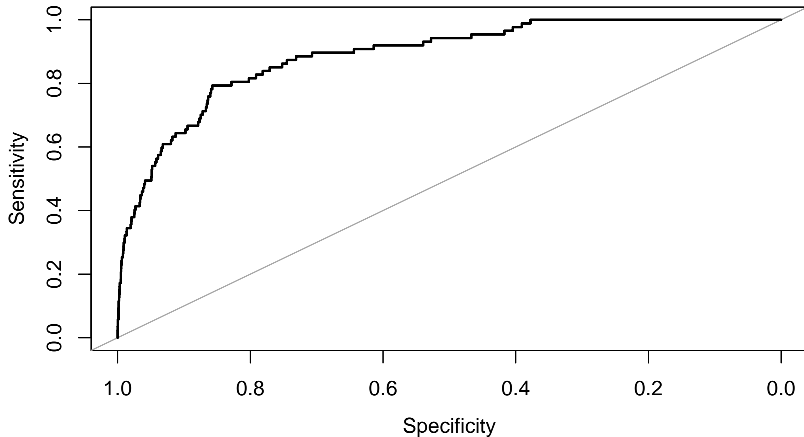
Random forest

Evaluate performance on the validation sample:

```
R> pds <- plogis(predict(m2_rf, firms[valid, ])$predictions)
R> with(firms[valid, ], table(default, round(pds)))

default    0
    no  5236
    yes   87

R> r <- pROC::roc(firms$default[valid], pds)
R> pROC::auc(r)

Area under the curve: 0.888
```

Random forest

```
R> plot(r)
```



Summary

The shadow rating approach works better with the random forest.

However, with the current very simple mapping of ratings to PDs, directly modeling the defaults seems to work better.