Modern Data Analysis in S-PLUS

Brian Ripley and Bill Venables

Workshop on October 7, 1998,
International S-PLUS User Conference,
Washington DC

ripley@stats.ox.ac.uk
http://www.stats.ox.ac.uk/~ripley

Bill.Venables@adelaide.edu.au

Programme

- visualization
- density estimation and smoothing
- using model-fitting functions
- beyond linear regression
  neural networks (I)

— lunch —

- tree-based methods
  neural networks (II)
- developing the `lda` function
- case studies, summary

Visualization, Density Estimation and Smoothing

Visualization

Challenge is to explore data in more than two or three dimensions.

via projections

Principal components is the most obvious technique: kD projection of data with largest variance matrix (in several senses). Usually 'shear' the view to give uncorrelated axes.

Lots of other projections looking for 'interesting' views, for example groupings, outliers, clumping.

Implementation via XGobi for Unix, and (now) for Windows using an X server.

multidimensional scaling

Aim is to represent distances between points well. Suppose we have distances (d_{ij}) between all pairs of n points, or a *dissimilarity* matrix. Classical MDS plots the first k principal components, and minimizes

$$\sum_{i \neq j} d_{ij}^2 - \widetilde{d}_{ij}^2$$

where $(\widetilde{d}_{ij})$ are the Euclidean distances in the kD space.

More interested in getting small distances right. Sammon (1969) proposed

$$\min E(d, \widetilde{d}) = \frac{1}{\sum_{i \neq j} d_{ij}} \sum_{i \neq j} \frac{(d_{ij} - \widetilde{d}_{ij})^2}{d_{ij}}$$

Implemented in function `sammon` in library `MASS`.

Kruskal and Shepard proposed only to preserve the ordering of distances, minimizing

$$STRESS^2 = \frac{\sum_{i \neq j} \left[\theta(d_{ij}) - \widetilde{d}_{ij}\right]^2}{\sum_{i \neq j} \widetilde{d}_{ij}^2}$$

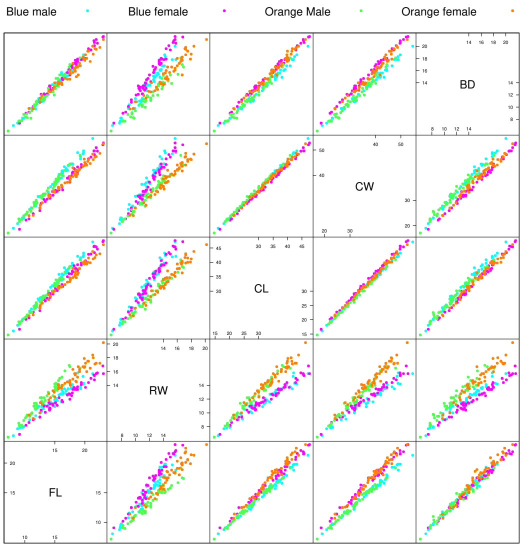
over both the configuration of points and an increasing function θ. Implemented in function `isoMDS` in library `MASS`. (This method is *isotonic*, hence the function name.)

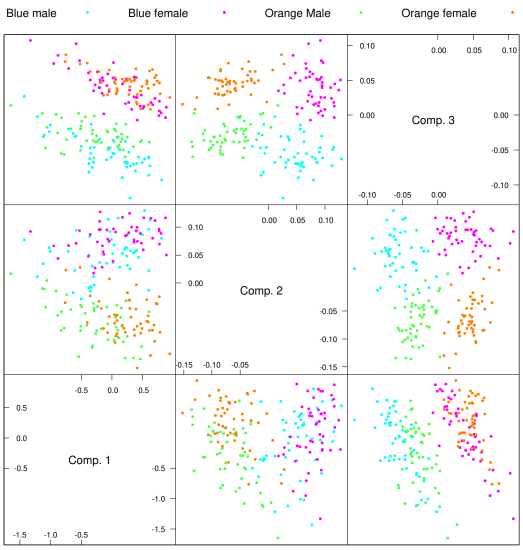
The optimization task is quite difficult and this can be slow.

Leptograpsus variegatus Crabs

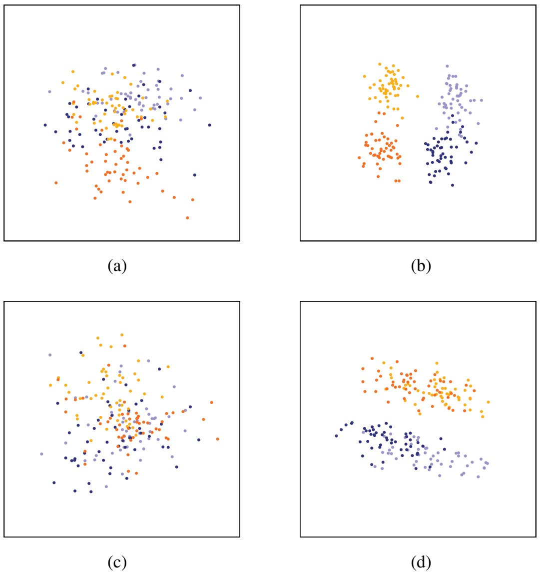
200 crabs from Western Australia. Two colour forms, blue and orange; collected 50 of each form of each sex.

Measurements of carapace (shell) length `CL` and width `CW`, the size of the frontal lobe `FL` and rear width `RW`.



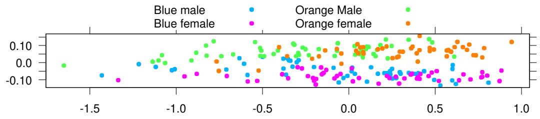


First three principal components on log scale.

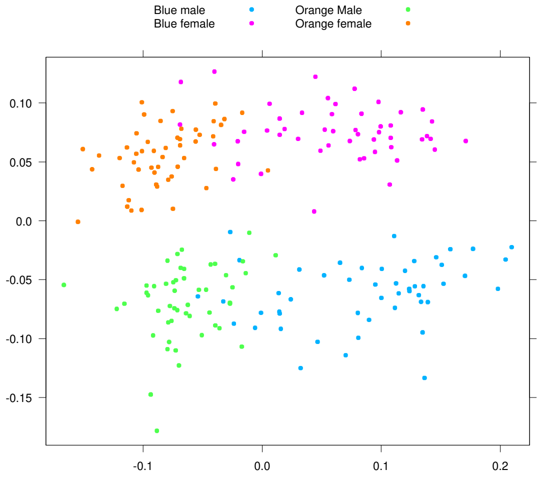


Projections of the *Leptograpsus* crabs data found by projection pursuit. View (a) is a random projection. View (b) was found using the natural Hermite index, view (c) by the Friedman–Tukey index and view (d) by Friedman's (1987) index.

Multidimensional scaling



`isoMDS` plot of raw data.



After re-scaling to (approximately) constant carapace area.

Density estimation

Closer look at data in one or perhaps two dimensions. Not many uses of density estimation other than visualization.

Kernel methods

are very well known.

$$\hat{f}(x) = \frac{1}{b}\sum_{j=1}^{n} K\left(\frac{x - x_j}{b}\right)$$

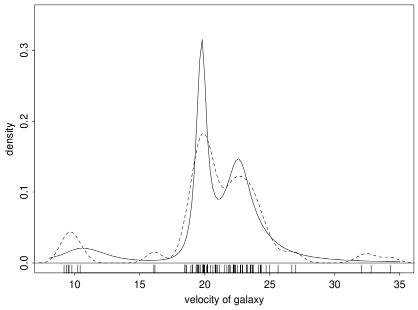
for a sample $x_1, \ldots, x_n$, a fixed kernel $K()$ and a bandwidth b.

Main issue is the choice of bandwidth b. State-of-the-art methods in libraries `MASS` and `KernSmooth`.

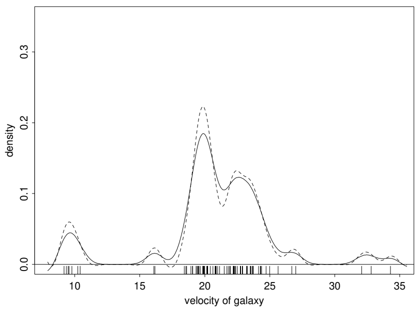
Boundary effects can be severe: no sophisticated S implementations (?)

Local polynomial methods

Do density estimation via smoothing: effectively take a fine grid and count the number of points in each interval, then smooth that.



Logspline (solid line) and kernel density (dashed) estimates for the `galaxies` data.



Local polynomial estimates by `locpoly` with linear (solid) and quadratic (dashed) local fits.

Smooth log-densities

There are several closely-related proposals to use a univariate density estimator of the form

$$f(y) = \exp g(y; \theta)$$

for a parametric family $g(\cdot; \theta)$ of smooth functions, most often splines. The fit criterion is maximum likelihood, possibly with a smoothness penalty.

It is necessary to ensure that the estimated density has unit mass, most conveniently by

$$f(y) = \exp g(y; \theta) / \int \exp g(y; \theta)\, \mathrm{d}y$$

The library `logspline` by Charles Kooperberg implements one variant on this theme. This uses a cubic spline for g, with smoothness controlled not by a penalty (as in smoothing splines) but by the number of knots selected. There is an AIC-like penalty; the number of the knots is chosen to maximize

$$\sum_{i=1}^{n} g(y_i; \widehat{\theta}) - n \log \int \exp g(y; \widehat{\theta})\, \mathrm{d}y - a \times \#\text{params}$$

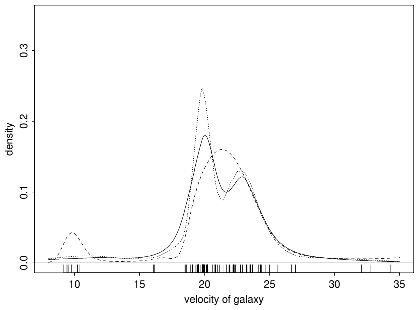
The default value of a is $\log n$ (sometimes known as BIC)

Local likelihood methods

Use a parametric model for $g(y) = \log f(y)$ (e.g. quadratic) fitted locally at x:

$$\sum_{i=1}^{n} K\left(\frac{y_i - x}{b}\right) g(y_i; \theta(x)) - n \log \int K\left(\frac{y-x}{b}\right) \exp g(y; \theta(x))\, \mathrm{d}y$$

Once again have to choose the bandwidth b, possibly depending on x.



`locfit` density estimates for the `galaxies` dataset. The solid line is the default, the dotted line has a variable bandwidth chosen by AIC, and the dashed line uses a surrogate Poisson model.

Multidimensional analogues

All these generalize easily to 2 or more dimensions, but

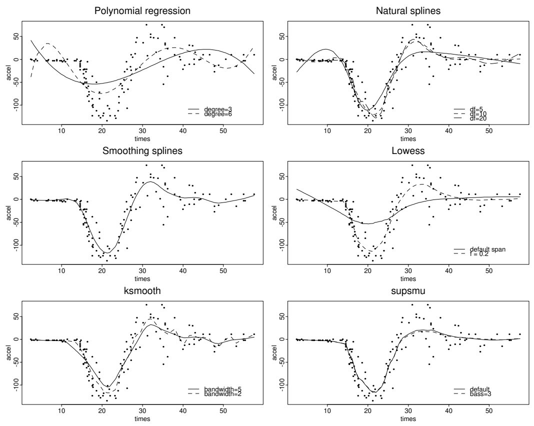
- 'bandwidth' now involves shape as well as size
- 'curse of dimensionality' means 'local' is a rather large neighbourhood.
- computation can be very slow.
- what are you going to do with this?

Survival data

Typically has partially observed data, e.g. $X > t$ is all that is known. Straightforward for likelihood-based methods, but localization is less clear-cut.

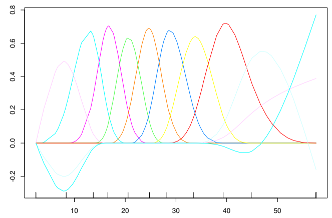
Smoothing methods

Lots of built-in methods:



Built-in

- Regression methods – polynomials, regression splines.



Splines created by `ns(times, df=10)`.

- Smoothing splines. A *smoothing spline* minimizes a compromise between the fit and the degree of smoothness of the form

$$\sum w_i [y_i - f(x_i)]^2 + \lambda \int (f''(x))^2 \, \mathrm{d}x$$

over all (measurably twice-differentiable) functions f. A piecewise cubic with breaks at the x_i, but does not interpolate the data points for $\lambda > 0$ and the degree of fit is controlled by λ. How smooth? Various forms of cross-validation.

- Local polynomial fitting.
  Locally constant: Nadaraya-Watson kernel estimation.

  $$\hat{y}_i = \sum_{j=1}^{n} y_i K\left(\frac{x_i - x_j}{b}\right) \Big/ \sum_{j=1}^{n} K\left(\frac{x_i - x_j}{b}\right)$$

  Locally linear: `supsmu`

  Locally linear or locally quadratic: `lowess` and `loess`.

3rd-party

One obvious omission is local polynomial fitting with constant-bandwidth kernels. This together with the choice of bandwidth is available from Matt Wand's library `KernSmooth` and Clive Loader's `locfit`, and `KernSmooth` does density estimation too.

Jim Ramsay's `Pspline` and `Lspline` libraries do more general spline smoothing.

Beyond Linear Regression and Neural Networks I

Overview

Response Y, predictors $X_1, \dots, X_p$.

Linear regression:

$$Y = \alpha + \sum_{j=1}^{p} \beta_j X_j + \epsilon$$

Additive models:

$$Y = \alpha + \sum_{j=1}^{p} f_j(X_j) + \epsilon$$

Response transformation models:

$$\theta(Y) = \alpha + \sum_{j=1}^{p} f_j(X_j) + \epsilon$$

for invertible smooth $\theta()$.

Projection pursuit regression:

$$Y = \alpha_0 + \sum_{j=1}^{M} f_j(\boldsymbol{\alpha}_j^T \boldsymbol{X}) + \epsilon$$

Back-fitting

Re-write additive model as

$$Y - \sum_{j \neq k} f_j(X_j) = \alpha + f_k(X_k) + \epsilon$$

so if we knew $f_j(), j \neq k$ we have a smoothing problem to choose f_k. Update smooth terms one at a time. May converge!

Not the only approach — Grace Wahba's school fits parametric models (smoothing splines) for (f_j) simultaneously. Difficulty is choosing the degrees of smoothing for more than a few terms.

Backfitting is quite closely tied to least-squares fitting.

Implementations

`gam`: clumsy choice of smoothness via `step.gam`.

`bruto`: in library `mda` (Hastie/Tibshirani). Cubic smoothing splines, choose bandwidth for each term as part of backfitting.

`mars`: in library `mda`. By default linear splines. Also allows a limited class of tensor splines.

`ace, avas`. Include response transformation. Smoothing choice by `supsmu`.

`ppreg`: projection pursuit regression, using `supsmu`.

`ppr`: in library `ppr`. Projection pursuit regression, using `supsmu` or smoothing splines, choose bandwidth for each term as part of backfitting.

Feed-forward neural networks

Projection pursuit regression was

$$Y = \alpha_0 + \sum_{j=1}^{M} f_j(\boldsymbol{\alpha}_j^T \boldsymbol{X}) + \epsilon$$

Now restrict f_j to be a single function (the 'squashing function'), usually logistic or tanh. This is the simplest form of neural network with just one output.

Both PPR and neural networks can have m outputs:

$$Y_i = \alpha_{0i} + \sum_{j=1}^{M} \beta_i f_j(\boldsymbol{\alpha}_j^T \boldsymbol{X}) + \epsilon$$

where for PPR we need some restrictions on (f_j) for identifiability (mean zero, variance one?).

Think of these as very flexible functions of p inputs and m outputs. May want to have an explicit linear part ('skip-layer connections').

These are universal approximators: for large enough M they can approximate any continuous function arbitrarily closely in many strong senses (e.g. uniformly on compacta).

We can use a neural network just about anywhere a linear predictor is used: in linear regression, GLMs, proportional hazard models, proportional odds models, ...

'training' a neural network

We have a parametric model, albeit with many parameters, so can use maximum likelihood.

All of these methods have a tendency to over-fit, to model the noise as well as the signal. Need some way to control this.

- Controlling number of parameters.
- Roughness penalties.

Analogous to knot selection in regression splines *vs* smoothing splines. Statisticians (with a few honourable exceptions) have been fixated on the first. All the theory favours the second.

For a neural network f_j is fixed, so we can have sharp changes in slope only if some components in $\boldsymbol{\alpha}_j$ are large. Usual penalty is *weight decay*, proportional to the sum of squares of the weights. Makes scaling of inputs important.

Tree-based Methods and Neural Networks II

Trees

A completely different extension is to piecewise-constant fits. Again, a universal approximator for a fine enough partition of the space. Combinatorial problems seem to be more difficult, so common to use simple axis-parallel splits or *rules*.

Good for explaining things: commonly used in botany, medical diagnosis, ...

Issues

- Class of allowable splits?
- How to choose a good split?
- How large to grow the tree?
- How large a tree to retain?

1.	Leaves subterete to slightly flattened, plant with bulb	*2.*
	Leaves flat, plant with rhizome	*4.*
2.	Perianth-tube > 10mm	**I. $\times$ hollandica**
	Perianth-tube < 10mm	*3.*
3.	Leaves evergreen	**I. xiphium**
	Leaves dying in winter	**I. latifolia**
4.	Outer tepals bearded	**I. germanica**
	Outer tepals not bearded	*5.*
5.	Tepals predominately yellow	*6.*
	Tepals blue, purple, mauve or violet	*8.*
6.	Leaves evergreen	**I. foetidissima**
	Leaves dying in winter	*7.*
7.	Inner tepals white	**I. orientalis**
	Tepals yellow all over	**I. pseudocorus**
8.	Leaves evergreen	**I. foetidissima**
	Leaves dying in winter	*9.*
9.	Stems hollow, perianth-tube 4–7mm	**I. sibirica**
	Stems solid, perianth-tube 7–20mm	*10.*
10.	Upper part of ovary sterile	*11.*
	Ovary without sterile apical part	*12.*
11.	Capsule beak 5–8mm, 1 rib	**I. enstata**
	Capsule beak 8–16mm, 2 ridges	**I. spuria**
12.	Outer tepals glabrous, many seeds	**I. versicolor**
	Outer tepals pubescent, 0–few seeds	**I. $\times$ robusta**

Key to British species of the genus *Iris*.

Splits

Most studies recommend using only binary splits: too hard to balance value of a binary and 10-way split.

For continuous and ordinal variates $x \gtreqless x_c$.

For unordered factors, all $2^{K-1} - 1$ splits of the K levels into two groups.

Values of a split

For regression problems, low within-cells sum of squares.

For classification problems, either Gini or entropy indices of impurity, both of which favour cells with examples of just one class.

Search for best possible split, with just a few computational short cuts.

May want *surrogates*, in case of missing values.

Stopping rules

There are many: few are satisfactory. Generally best to grow a large tree, down to small cells.

Pruning

Backwards selection is better than forwards selection. Choose a subtree *either*

to minimize error or deviance on a validation set

or

to minimize a penalized error or deviance on the training set, say

$$\sum_{\text{leaves}} R(t) + \alpha \times \text{size}$$

There are lots of subtrees: fortunately easy to find the minimizers for all possible α.

Choose α by V-fold cross-validation.

Implementations

`tree` from AT&T S (Daryl Pregibon and Linda Clark). Original pruning routines replaced from S-PLUS 3.4.

`Tree` by BDR. Adds Gini index, weights, case splitting of missing values. Enhanced `treefix` with V&R2 libraries.

`RPART` by Terry Therneau and Beth Atkinson. Uses cross-validation intrinsically, has Gini index, surrogate splits, Poisson and (one version of) survival trees. Closest to CART™.

If you are new to this, start with `rpart`.

Prediction *vs* Explanation

Linear regressions are often assumed to give explanations, and subset selection is usually done to provide simple explanations.

Ridge regression may provide better predictions: possibly also better explanations.

Neural networks are completely black boxes.

Small classification trees provide some explanation: large classification trees and regression trees little or none.

Additive models may appear to provide explanation, but it is often very fragile.

Refining

`lda`

Linear Discriminant Analysis

Underlying model is K classes, with data from class k giving $X \sim N(\boldsymbol{\mu}_k, \Sigma)$.

Classical approach (Rao) is to estimate the parameters by maximum likelihood (possibly with a bias correction), then estimate posterior probabilities by

$$p(C = c \mid X = \boldsymbol{x}) = \frac{\widehat{p}_c(\boldsymbol{x})\, \widehat{\pi}_c}{\sum_k \widehat{p}_k(\boldsymbol{x})\, \widehat{\pi}_k}$$

If desired, partition region according to the highest posterior probability.

Coding

There is a function `discr` in S, but hard to follow and not much functionality.

First function (eventually in V&R1) was a single function with call

```
lda <- function(x, grouping, prior, tol = 0.0001)
```

generating objects of class `"lda"` and a `predict` method.

Next step is to make this generic with the old function as the default method:

```
lda <- function(x, ...)
{
  if(is.null(class(x))) class(x) <- data.class(x)
  UseMethod("lda", x, ...)
}
lda.default <- function(x, grouping, prior,
                        tol = 0.0001, ...)
    ....
```

Note the ... to mop up extra arguments from other methods.

When working with model formulae it is helpful to store the function call as component `call` of the output list. There is a standard way to do this using the function `match.call`, so the return value of `lda.default` becomes

```
structure(list(prior = prior, counts = counts,
    means = group.means, scaling = scaling,
    lev = levels(g), svd = X.s$d[1:rank], N = n,
    call = match.call()), class = "lda")
```

Once this is done the function `update` can be used to update the object just as for linear models.

Note that it is not possible to record the call of the generic function, as `UseMethod` does not return but merely implements method dispatch.

First argument of `lda` may be either a matrix-like object or a formula. Since matrices do not necessarily have a class assigned, `lda` assigns a class via the function `data.class`. The method despatch mechanism will then use the functions `lda.matrix`, `lda.data.frame` and `lda.formula` as appropriate.

`lda.data.frame` just calls the matrix method and then fixes up the recorded call.

`lda.matrix` works with `subset` and `na.action` then calls the defaults.

`lda.formula` constructs a matrix and grouping factor via `model.frame` and `model.matrix`, calls the default method and adjusts the recorded call.

`predict` method

We may have to extract the data matrix `x` from a new data frame. We made life easy by storing the `terms` attribute in the `lda` object, so we can use code like

```
  if(!is.null(Terms <- object$terms)) { #
    # formula fit
    if(missing(newdata))
      newdata <- model.frame.lm(object)
    else
      newdata <-
        model.frame(as.formula(delete.response(Terms)),
                    newdata, na.action=na.pass)
    x <- model.matrix(delete.response(Terms), newdata)
    xint <- match("(Intercept)", dimnames(x)[[2]],
                  nomatch=0)
    if(xint > 0) x <- x[, -xint, drop=F]
  } else { #
    # matrix or data-frame fit
    if(missing(newdata)) {
      xname <- eval(substitute(object$call$x))
      if(!is.null(sub <- object$call$subset))
        newdata <-
          eval(parse(text=paste(deparse(xname),"[",
               deparse(sub),",]")), sys.parent())
      else
        newdata <- eval(parse(text=deparse(xname)),
                        sys.parent())
      if(!is.null(nas <- object$call$na.action))
        newdata <- do.call(nas, list(newdata))
    }
    if(is.null(dim(newdata)))    # a row vector
      dim(newdata) <- c(1, length(newdata))
    x <- as.matrix(newdata)  # to cope with dfs
  }
```

S4–style

First decide on the class structure.

```
setClass("lda",
         representation(prior = "named",
                        counts = "named",
                        means = "matrix",
                        scaling = "matrix",
                        lev = "character",
                        svd = "numeric",
                        N = "integer",
                        call = "call")
         )
```

Note that we have to specify the classes of the slots very carefully: numeric and integer slots do not have names attributes, but slots of class `"named"` do. We do not include a slot for `terms`, as `"terms"` is an old-style class with no fixed representation.

Our next decision is the set of arguments for all the methods. We had some choice here, and experimented with naming all the possible arguments for each method or making use of '...'. We settled on

```
lda = function(x, y, ...)
  stop("lda not implemented for class ", class(x))
```

as a function that will become the default method.

Since the previous `lda.default` is not going to be one of the methods, we rename it to `lda1`.

We can now begin to set up methods. Every object has a class. We could rely on inheritance, or use

```
lda.matrix = function(x, y, ...) {
  x = as(x, "matrix")
  dots = list(...)
  if(hasArg(subset)) {
    subset = dots$subset
    x = x[subset, , drop=F]
    y = y[subset]
  }
  if(hasArg(na.action)) {
    na.action = dots$na.action
    dfr = na.action(data.frame(g=y, x=x))
    y = dfr$g
    x = dfr$x
  }
  res = lda1(x=x, grouping=y, ...)
  res@call = match.call()
  res
}

lda.data.frame = function(x, y, ...) {
  x = as.matrix(x)
  res = callGeneric()
}
```

```
lda.Matrix = function(x, y, ...) {
  x = data.matrix(x)
  callGeneric()
}

setMethod("lda", "matrix", lda.matrix)
setMethod("lda", "Matrix", lda.Matrix)
setMethod("lda", "data.frame", lda.data.frame)
```

Note that rather than call `NextMethod`, we call the generic again with `callGeneric`.

```
lda.formula = function(x, y, ...)
{
  data = as.data.frame(y)
  m = match.call(expand.dots = F)
  m$... = NULL
  names(m)[2:3] = c("formula", "data")
  if(hasArg(subset)) m$subset = subset
  if(hasArg(na.action)) m$na.action = na.action
  m[[1]] = as.name("model.frame")
  m = eval(m, sys.parent())
  Terms = attr(m, "terms")
  y = model.extract(m, "response")
  x = model.matrix(Terms, m)
  xint = match("(Intercept)", dimnames(x)[[2]],
               nomatch=0)
  if(xint > 0) x = x[, -xint, drop=F]
  lda1(x = x, grouping = y, ...)
  res@call = match.call()
  res
}
setMethod("lda", "formula", lda.formula)
```

The argument names have changed (they have to be the same as the other methods) and so the lazy way of calling `model.frame` has to be replaced by an explicitly constructed call.

There was a lazier way to do this, by not calling `lda.formula` directly.

```
setMethod("lda", "formula",
          function (x, y, ...) {
            res = oldlda.formula(x, y, ...)
            res@call = match.call()
            res
          })

oldlda.formula =
   function(formula, data = NULL, ...,
   subset, na.action = na.fail)
{
  m = match.call(expand.dots = F)
  if(is.matrix(eval(m$data, sys.parent())))
    m$data = as.data.frame(data)
  m$... = NULL
  m[[1]] = as.name("model.frame")
  m = eval(m, sys.parent())
  Terms = attr(m, "terms")
  grouping = model.extract(m, "response")
  x = model.matrix(Terms, m)
  xint = match("(Intercept)", dimnames(x)[[2]],
               nomatch=0)
  if(xint > 0) x = x[, -xint, drop=F]
  res = lda1(x, grouping, ...)
  res@call = match.call()
  res
}
```

The snag with this approach is the that stored call (`oldlda.formula(x, y)`) is not informative unless we update the call in the wrapper function.

The changes to `lda1` are mainly in constructing the returned object. We have also to ensure that `counts` has the right class (`table` returns a one-dimensional array).

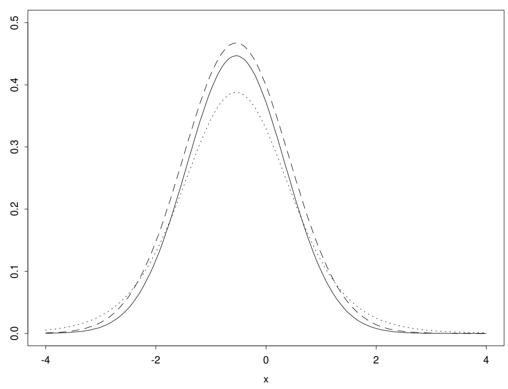
```
lda1 = function(x, grouping, prior=NULL,
   tol=1.0e-4,
   method=c("moment", "mle", "mve", "t"),
   CV=F, nu=5, ...)
{
    ....
  lev = levels(g)
  counts = as.vector(table(g))
  names(counts) = lev
    ....
  res = new("lda")
  res@prior = prior
  res@counts = counts
  res@means = group.means
  res@scaling = scaling
  res@lev = lev
  res@svd = X.s$d[1:rank]
  res@N = n
  res
}
```

We need some changes in the methods, apart from replacing $ by @ to reflect the change from a list to a representation with slots. We also need to declare a means of displaying the object, as `show` will not automatically call `print.lda` but rather would list the slots.

```
setMethod("show", "lda",
   function(object) print.lda(object))

print.lda = function(x, ...)
{
  if(!is.null(cl <- x@call)) {
    cat("Call:\n")
    dput(cl)
  }
  cat("\nPrior probabilities of groups:\n")
  print(x@prior, ...)
  cat("\nGroup means:\n")
  print(x@means, ...)
  cat("\nCoefficients of linear discriminants:\n")
  print(x@scaling, ...)
  svd = x@svd
  names(svd) = dimnames(x@scaling)[[2]]
  if(length(svd) > 1) {
    cat("\nProportion of trace:\n")
    print(round(svd^2/sum(svd^2), 4), ...)
  }
  invisible(x)
}
```

More changes can be seen on the handout.



Estimates of the density based on a random sample of size 10 from $N(0,1)$. The 'plug-in' estimate is shown with a solid line, the predictive estimate with a dotted line, and the unbiased estimate on log scale with a dashed line.

- For testing would like to do cross-validation. There are updating formulae for leave-one-out cross-validation, which we implemented, even though 10-fold cross-validation would be more reliable.

Extensions

- Robustify. Either use robust estimates of $\boldsymbol{\mu}_k$ and Σ, or use maximum likelihood, replacing multivariate normal by multivariate t. Latter proves to be a simple iterative calculation.
- MLEs of $\boldsymbol{\mu}_k$ and Σ *may* be good estimates of those quantities, even though they are traditionally bias-corrected, but is $N(\widehat{\boldsymbol{\mu}}_k, \widehat{\Sigma})$ a good estimate of $\widehat{p}_k(\boldsymbol{x})$? Certainly biased. We can seek a unbiased estimator of $p_k(\boldsymbol{x})$ or of $\log p_k(\boldsymbol{x})$. The latter is more appealing, and easily implemented.
- We are ignoring the variability in the parameter estimates. A better choice for $\widehat{p}_k(\boldsymbol{x})$ is the predictive estimate

  $$\tilde{p}_k(\boldsymbol{x}) = \int p_k(\boldsymbol{x};\theta)p(\theta \mid \text{data})\,\mathrm{d}\theta$$

  for $\theta = (\boldsymbol{\mu}_k, \Sigma)$. This needs a prior over θ, but calculations can be done for vague priors.

  The effects for LDA are small unless the groups are of very different sizes. However, it does affect the tail behaviour of the estimates and so the boundaries far from any data.