PreProcessing with OOMPA

Kevin R. Coombes

January 4, 2007

Contents

1 Introduction

OOMPA is a suite of object-oriented tools for processing and analyzing large biological data sets, such as those arising from mRNA expression microarrays or mass spectrometry proteomics. This vignette documents a low-level package that supplies utilities and preprocessing routines for the higher-level pieces that come later.

2 Getting Started

As usual, you begin by loading the package

```
> library(PreProcess)
```

2.1 Matrix manipulations

Now we can play with some of the simple utilities. The first two are matrix operations borrowed from MATLAB. To see how they work, start with a small matrix:

```
> mat <- matrix(1:12, 3, 4)
> mat
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Now flip it left-to-right:

```
> fliplr(mat)
```

```
     [,1] [,2] [,3] [,4]
[1,]   10    7    4    1
[2,]   11    8    5    2
[3,]   12    9    6    3
```

Next, flip it up-to-down:

```
> flipud(mat)
```

```
     [,1] [,2] [,3] [,4]
[1,]    3    6    9   12
[2,]    2    5    8   11
[3,]    1    4    7   10
```

Mathematically, a left-right flip followed by an up-down flip is a 180° rotation.

```
> flipud(fliplr(mat))
```

```
     [,1] [,2] [,3] [,4]
[1,]   12    9    6    3
[2,]   11    8    5    2
[3,]   10    7    4    1
```

2.2 Plotting Ellipses

We also have a utility that adds ellipses to plots. Figure 1 contains some examples.

2.3 Concordance

We also include a few statistical utilities, the most interesting of which is probably a computation of the "concordance correlation coefficient". As the name suggests, this quantity is similar to the Pearson correlation coefficient. However, instead of measuring whether the points track any straight line, this quantity is specifically designed to test if the values track the identity line. For example, consider the following three variables:

```
> x <- rnorm(30)
> y <- x + rnorm(30, sd = 0.1)
> z <- 3 + 2 * x + rnorm(30, sd = 0.1)
```

```
> x <- rnorm(1000, 1, 2)
> y <- rnorm(1000, 1, 2)
> plot(x, y)
> ellipse(1, 1, col = "red", type = "l", lwd = 2)
> ellipse(3, 2, col = "green", type = "l", lwd = 2)
> ellipse(3, 2, x0 = 2, y0 = 2, col = "blue", type = "l", lwd = 2)
```

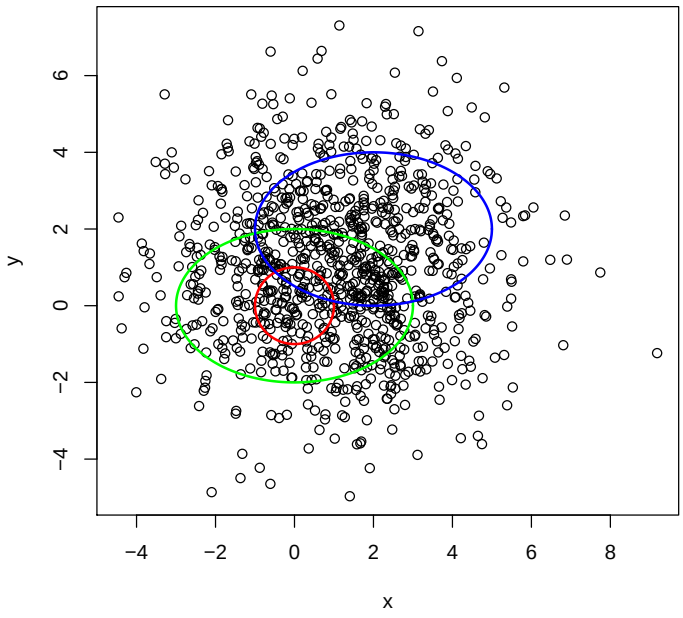

Figure 1: Independent normal noise with overlaid ellipes.

By construction, all three are highly correlated.

```
> cor(x, y)

[1] 0.9970514

> cor(x, z)

[1] 0.999084

> cor(y, z)

[1] 0.9956913
```

But only x and y are concordant.

```
> f.cord(x, y)

[1] 0.9967042

> f.cord(x, z)

[1] 0.264887

> f.cord(y, z)

[1] 0.2704955
```

2.4 Color Coded Graphs

We frequently find ourselves producing multiple figures with a common color scheme, where each color or each symbol is used to denote samples or genes with a particular property (in the simplest case, "cancer" versus "normal"). Because we got tired of continually cutting and pasting `plot` and `points` commands and making sure the color legends stayed synchronized, we developed the `ColorCoding` and `ColorCodedPair` classes to encapsulate this notion.

We can simulate some data as an example.

```
> x <- matrix(rnorm(100 * 3), nrow = 100, ncol = 3)
> class1 <- class2 <- rep(FALSE, 100)
> class1[sample(100, 20)] <- TRUE
> class2[sample(100, 20)] <- TRUE
> class3 <- !(class1 | class2)
> codes <- list(ColorCoding(class1, "red", 16), ColorCoding(class2,
+     "blue", 15), ColorCoding(class3, "black", 17))
```

```
> par(mfrow = c(2, 1))
> plot(ColorCodedPair(x[, 1], x[, 2], codes), xlab = "Coord1",
+     ylab = "Coord2")
> plot(ColorCodedPair(x[, 1], x[, 3], codes), xlab = "Coord1",
+     ylab = "Coord3")
> par(mfrow = c(1, 1))
```

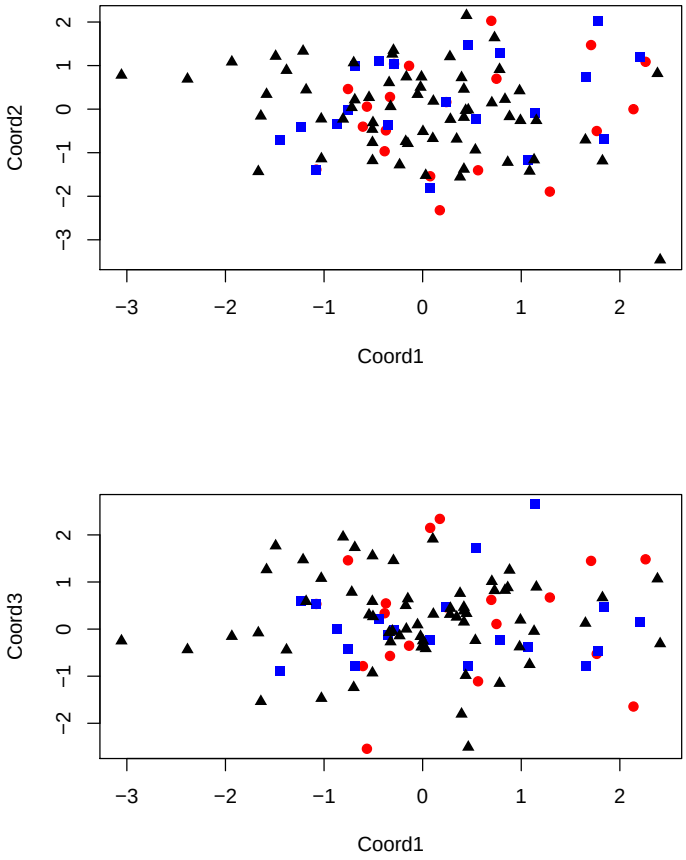

Figure 2: Color coded plots of three (simulated) related variables.

3 Processing in Pipelines

We have been analyzing microarray data at M.D. Anderson for more than seven years now. We have continually been looking for better ways to process the data, since we are still not convinced that any of the existing methods is truly "best" in all circumstances. As a result, we have often tinkered with our default processing method, changing it when we find something that we think works better.

That approach is common in academic settings. However, we also have a service role to perform, in that we have to analyze data for grant proposals and articles in preparation by the biologists and clinicians at M.D. Anderson. In particular, we often find ourselves going back to analyses that were performed six months ago (possibly by a statistical analyst who has since moved on to another position) and trying to figure out exactly how we processed the data.

That question is much, *much*, **much** harder than it sounds. The problem is that, just because there is an '.Rdata' workspace and an R script sitting in the directory that contains the raw data, there is no assurance that the objects in that workspace were actually processed using the code in that script. The *Sweave* package goes a long way toward solving this problem, but only if one adds some additional discipline. For example, some unknown commands may have been entered at the command line; multiple scripts may have been invoked, and even the code within the script may have been executed in some order other than the one that has been preserved.

The approach we are working on for this problem involves objects that have a "history" slot so they remember what was done to them. In order for this to work, however, we have to convert simple processing functions into full-fledged objects that can update the history slot in the right way. We do this with `Processor`s and `Pipeline`s.

To see how this might work, we start by simulating one `Channel` of a microarray experiment:

```
> nc <- 100
> nr <- 100
> v <- rexp(nc * nr, 1/1000)
> b <- rnorm(nc * nr, 80, 10)
> s <- sapply(v - b, max, 1)
> ct <- ChannelType("user", "random", nc, nr, "fake")
> subbed <- Channel(name = "fraud", parent = "", type = ct, x = s)
> rm(ct, nc, nr, v, b, s)
> summary(subbed)

fraud, a microarray channel object
Parent object: NA
Microarray type: user random
Labeled with: fake
Design size: 100 by 100
Design information object:
```

```
History:
         Channel(parent = "", name = "fraud", type = ct, x = s)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.0   209.0   616.7   918.4  1303.0  9562.0
```

As you can see from the summary, this object remembers the function call that created it.

The *PreProcess* package include an object called the `PIPELINE.STANDARD` that represents one simple, standard way to process microarray data. We can invoke it and see what happens:

```
> processed <- process(subbed, PIPELINE.STANDARD)
> summary(processed)

log normalized fraud, a microarray channel object
Parent object: NA
Microarray type: user random
Labeled with: fake
Design size: 100 by 100
Design information object:
History:
         Channel(parent = "", name = "fraud", type = ct, x = s)
         Default channel processing (using pipeline: PIPELINE.STANDARD)
         Global normalization (using object: proc) with parameter = 0
         Truncated below (using object: proc) with parameter = 0
         Log transformation (using object: proc) with parameter = 2

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-0.3818  7.3260  8.8870  8.0460  9.9660 12.8400
```

Now the summary tells us that this object was processed using the standard pipeline. It also knows that this means that we started by performing global normalization, we then truncated below to remove any values below 0, and we then transformed by computing the base two logarithm.

Each `Pipeline` is just a list of `Processor`s that are applied in sequence, so we can mix-and-match simple pieces inside the `Pipeline`. The `Processor`s that are defined inside the *PreProcess* package are

- `PROC.SUBTRACTOR`: subtract a global constant (default: 0).
- `PROC.THRESHOLD`: truncate below (default: at 0).
- `PROC.LOG.TRANSFORM`: Compute logarithms (default base: 2)
- `PROC.GLOBAL.NORMALIZATION`: divide by a global constant (default: 75^{th} percentile).

- `PROC.MEDIAN.EXPRESSED.NORMALIZATION`: divide by the median of the expressed values, where expressed means that the signal is greater than 0.
- `PROC.SUBSET.NORMALIZATION`: divide by the median of a specified subset of spots (default: all).
- `PROC.SUBSET.MEAN.NORMALIZATION`: divide by the mean of a specified subset of spots (default: all).