Object Oriented Microarray and Proteomics Analysis (OOMPA)

Kevin R. Coombes

August 19, 2019

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 the base package, *oompaBase*. A critical (but invisible to the user) feature of the *oompaBase* package is that it defines a `class union` allowing you to use "numeric" or "NULL" objects in the design of an S4 class. More interesting user-visible features include alternative color schemes and vectorized matrix operations to speed the computation of row-by-row means, variances, and t-tests.

2 Getting Started

You invoke the package in the usual way:

```
> library(oompaBase)
```

3 Color Schemes

To illustrate the various color schemes, we first create a structured matrix:

```
> mat <- matrix(1:1024, ncol=1)
```

The following code is used to generate Figure 1.

```
> # windows(width=6,height=8)
> opar <- par(mfrow=c(8, 1), mai=c(0.3, 0.5, 0.2, 0.2))
> image(mat, col=jetColors(128), main='jetColors')
> image(mat, col=wheel(64, 0.5), main='wheel, half saturation')
> image(mat, col=redgreen(64), main='redgreen')
> image(mat, col=blueyellow(32), main='blueyellow')
> image(mat, col=cyanyellow(32), main='cyanyellow')
> image(mat, col=redscale(64), main='redscale')
> image(mat, col=bluescale(64), main='bluescale')
> image(mat, col=greyscale(64), main='greyscale')
> par(opar)
```

4 Row-by-row Matrix Operations

We now want to illustrate the "matrix operations" that allow for rapid computation of row-by-row means, variances, and t-tests.

We start by creating a slightly more interesting matrix full of random data. First, we make the variance larger in the second half (by column) of the data than in the first half.

```
> ng <- 10000
> ns <- 50
> dat <- matrix(rnorm(ng*ns, 0, rep(c(1, 2), each=25)), ncol=ns, byrow=TRUE)
```

Next, we shift the mean for the first 500 "genes" (rows).

```
> dat[1:500, 1:25] <- dat[1:500, 1:25] + 2
```

In order to compute t-tests, we also assign arbitrary labels separating the "sample columns" into two groups.

```
> clas <- factor(rep(c('Good', 'Bad'), each=25))
```

Here we compute the row-by-row means.

```
> a0 <- proc.time()
> myMean <- matrixMean(dat)
> used0 <- proc.time() - a0
```

For comparison purposes, we perfom the same computation using `apply`.

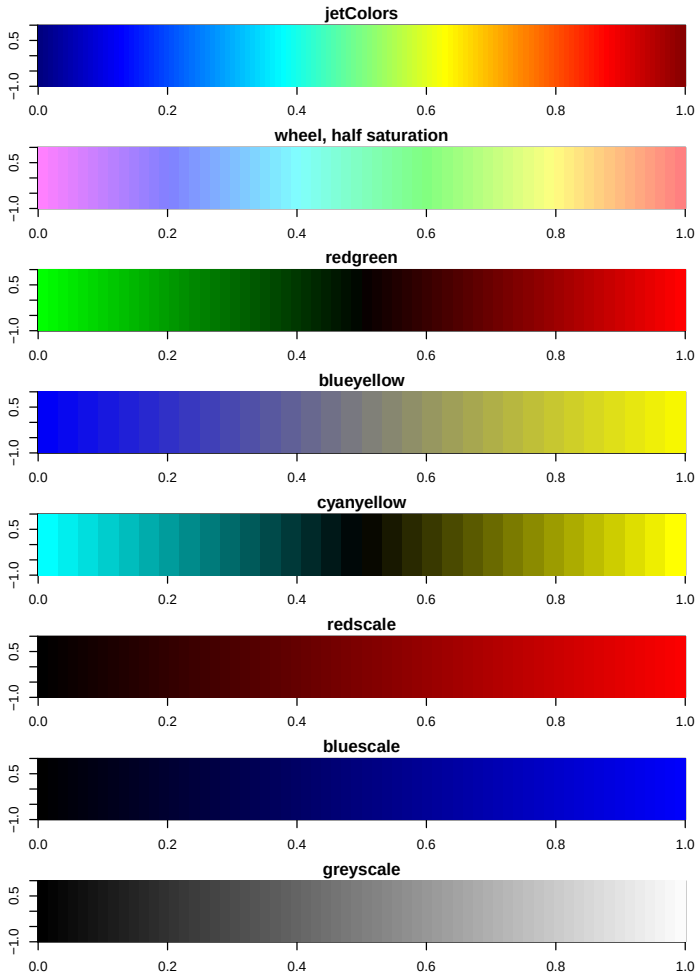

Figure 1: Eight color schemes.

```
> a1 <- proc.time()
> mm <- apply(dat, 1, mean)
> used1 <- proc.time() - a1
```

The results are the same, to within round-off error.

```
> summary(as.vector(myMean-mm))
```

```
      Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
-8.882e-16 -2.776e-17  0.000e+00  8.616e-19  2.776e-17  6.661e-16
```

There is a measurable (although not really user-perceptible) difference in the time for the two methods.

```
> used0
```

```
   user  system elapsed
  0.001   0.001   0.002
```

```
> used1
```

```
   user  system elapsed
  0.097   0.007   0.104
```

Here we compute the variances using two different methods.

```
> a0 <- proc.time()
> myVar  <- matrixVar(dat, myMean)
> a1 <- proc.time()
> vv <- apply(dat, 1, var)
> a2 <- proc.time()
```

Again, the values are the same:

```
> summary(as.vector(myVar - vv))
```

```
      Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
-3.109e-15 -4.441e-16  0.000e+00 -7.883e-18  4.441e-16  2.665e-15
```

However, the time savings is substantially larger.

```
> a1 - a0
```

```
   user  system elapsed
  0.003   0.003   0.007
```

```
> a2 - a1
```

```
   user  system elapsed
  0.225   0.001   0.225
```

Not surprisingly, there is an even bigger time savings when computing (equal variance) t-statistics.

```
> t0 <- proc.time()
> myT <- matrixT(dat, clas)
> t1 <- proc.time()
> tt <- sapply(1:nrow(dat), function(i) {
+   t.test(dat[i,clas=="Bad"], dat[i, clas=="Good"], var.equal=T)$statistic
+ })
> t2 <- proc.time()

> summary(as.vector(tt - myT))

      Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
-9.770e-15 -1.110e-16  0.000e+00  6.796e-18  1.110e-16  6.217e-15

> t1 - t0

   user  system elapsed
  0.014   0.003   0.018

> t2 - t1

   user  system elapsed
  3.247   0.004   3.251
```

5 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))
```

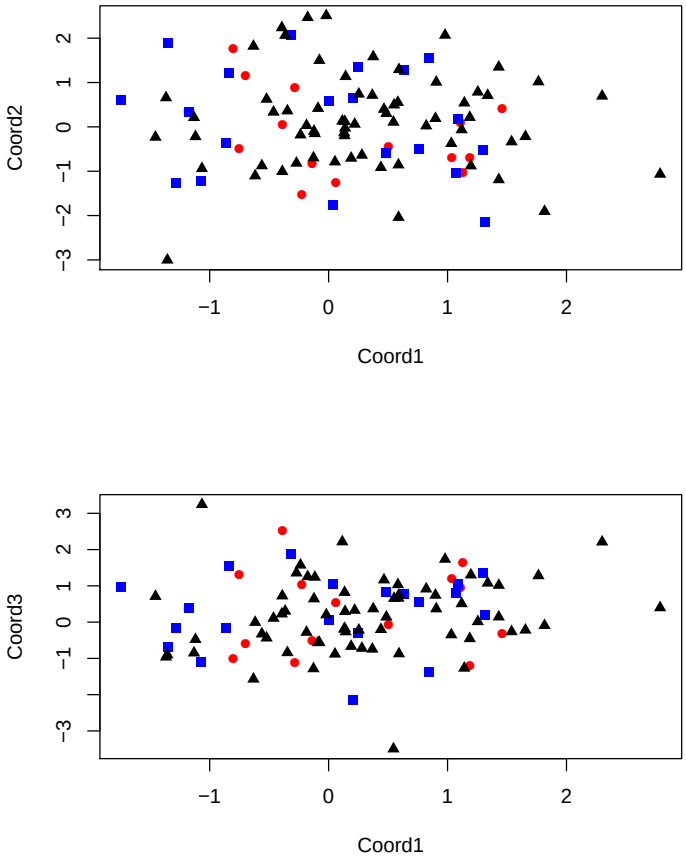

Figure 2: Color coded plots of three (simulated) related variables.