R / Bioconductor: A Short Course

James H. Bullard
Sandrine Dudoit
Division of Biostatistics, UC Berkeley
www.stat.berkeley.edu/~bullard
www.stat.berkeley.edu/~sandrine

Cuernevaca, Mexico
January 21-25, 2008

Introduction to Bioconductor

1 Background
2 Getting Started
3 Environments
4 Annotation or MetaData

Bioconductor

- Biological experiments are continuously generating more and more data!
- It has become nearly impossible to analyze a biological experiment without statistics and bioinformatics.
- Different research groups are constantly rewriting essentially the same software for only slightly different purposes.
- There are some exceptions out there, such as Blast, but often we find ourselves writing software which has probably already been written (usually at least twice).
- Bioconductor is an open-source / open-development set of tools which can be widely employed in a number of genetic and biomedical settings.

Bioconductor: Overview

- Bioconductor was started in the Fall of 2001
- The core maintainers and hosts of the Bioconductor websites are located at Fred Hutchinson Cancer Research Center
- A new version of Bioconductor is released twice-anually and is linked to the release of R
- Bioconductor makes heavy use of the S4 class system of R. This often makes it more complicated than simpler R packages to work with.
- Bioconductor has been most successful in the analysis of Microarray data - R is written almost exclusivey by Statisticians; although it is a very general environment for computing most often we think of R as a statistical programming environment.

Bioconductor: Goals

1. Provide access to statistical and graphical tools to perform analysis in a number of bioinformatics settings.
2. Provide a framework for extending the components of the system to customize the environment for particular settings.
3. Provide a comprehensive set of documents describing the system and how to extend/interact with the components.
4. Provide tools to interact with publicly available databases as well as other sources of meta-data. [1]

[1] These goals were adapted from W. Huber's slides

Website

Lets take a small website tour to see whats there.
`bioconductor.org`

Bioconductor: Getting Started

The core bioconductor package is: Biobase - There are some "sample" data sets in the package - these are good for testing some of our knowledge of the core structures[2]

```
> library(Biobase)
> library(help = "Biobase")
> data(package = "Biobase")
> data(sample.ExpressionSet)
> print(sample.ExpressionSet)
```

[2] Much of this introduction is bases on: "Lab: Using R and Bioconductor", by R. Gentleman, F. Hahne, and P. Murrell

ExpressionSet

- The core class for microarray analysis in Bioconductor is the ExpressionSet
- This class packages a number of common objects that are commonly encountered in microarray experiments

```
> slotNames("ExpressionSet")
[1] "assayData"
[2] "phenoData"
[3] "featureData"
[4] "experimentData"
[5] "annotation"
[6] ".__classVersion__"
```

ExpressionSet: History

- Unfortunately, the ExpressionSet has evolved from some earlier classes with some strikingly similar names (exprSet).
- This class still exists in the system but its use is deprecated - we often see pdfs or websites using this class, we should strive to use only the new classes.
- The ExpressionSet extends the eSet - we will rarely encounter this class, but lets have a quick look.

```
> extends("ExpressionSet")
```

```
[1] "ExpressionSet"
[2] "eSet"
[3] "VersionedBiobase"
[4] "Versioned"
```


ExpressionSet: History

```
> slotNames("eSet")
```

```
[1] "assayData"
[2] "phenoData"
[3] "featureData"
[4] "experimentData"
[5] "annotation"
[6] ".__classVersion__"
```


ExpressionSet

The ExpressionSet class is derived from eSet. The main difference between these two classes is that an ExpressionSet provides an exprs method which accesses the expression matrix.

```
> hasMethod("exprs", "eSet")
```

```
[1] FALSE
```

```
> hasMethod("exprs", "ExpressionSet")
```

```
[1] TRUE
```


PhenoData

- The slot phenoData from the ExpressionSet is of type AnnotatedDataFrame - this class essentially contains "meta-data" of the experiment which is often data on the samples which were hybridized to the microarray.
- We access the phenoData slot using the accessor function phenoData
- The phenoData slot is an AnnotatedDataFrame, this is essentially a data.frame with a slot: varMetadata which contains information about the phenotypic data stored in the class.

```
> pData <- phenoData(sample.ExpressionSet)
> varMetadata(pData)
```

PhenoData

```
      labelDescription
sex         Female/Male
type       Case/Control
score     Testing Score
```

```
> pData$sex
```

```
 [1] Female Male   Male   Male   Female
 [6] Male   Male   Male   Female Male
[11] Male   Female Male   Male   Female
[16] Female Female Male   Male   Female
[21] Male   Female Male   Male   Female
[26] Female
Levels: Female Male
```


ExperimentData

- The ExperimentData object contains information about the experiment

```
> experimentData(sample.ExpressionSet)
> class(experimentData(sample.ExpressionSet))
```

1. Find out a little about the MIAME class?
2. What slots does it have?
3. What does the show method do?

ExpressionSet: Expression Matrix

So we have information about the experiment, information about the subjects or samples from the experiment, and we have information about the probes/probesets from the microarray - these can be found in using `exprs(sample.ExpressionSet)`

- The expression matrix has dimension $N_{reporters} \times N_{arrays}$
- "reporters" is the terminology used for probes/probesets or other such signal producing elements from a microarray

Looking at the Data

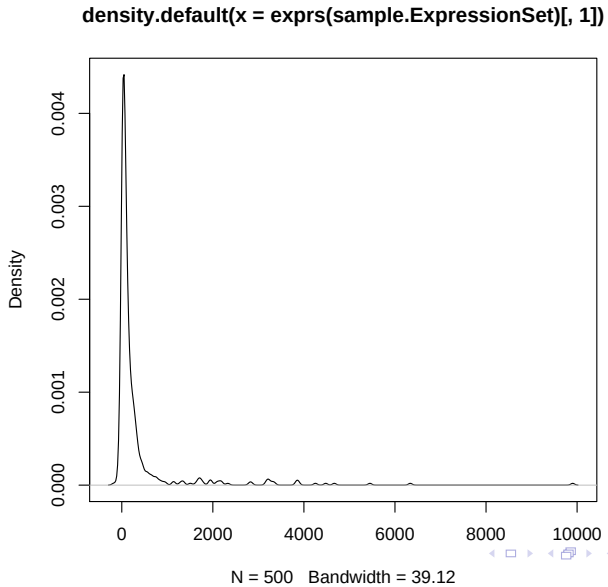
Since we are stressing looking at the data - how do we look at the data

```
> dim(exprs(sample.ExpressionSet))
```

```
[1] 500  26
```

```
> plot(density(exprs(sample.ExpressionSet)[,
+     1]))
```

Looking at the Data



Subsetting ExpressionSets

We can subset the ExpressionSet object just as we could a matrix. In the example and most ExpressionSets we have the samples in the columns and so we will subset based on some phenotypic characteristic (the columns). If we want to subset the reporters (probes or probesets) then we subset the rows - Note: things are kind of backwards in the microarray we have the subjects in the columns and the covariates (genes) in the rows.

```
> pData <- phenoData(sample.ExpressionSet)$type
> cases <- grep("Case", pData)
> controls <- grep("Control", pData)
> casesEx <- sample.ExpressionSet[,
+     cases]
```


Subsetting ExpressionSets

```
> controlsEx <- sample.ExpressionSet[,
+     controls]
```

What is the class of casesEx and controlsEx?

Accessing the ExpressionSet

- Accessing the relevant data involves calling accessor functions. We should try to avoid ever accessing the data directly with the "@" accessor because it is less future-proof. Unlike many object oriented programming languages R does not provide a mechanism for protecting data, such as "private" member variables in many languages.
- Have a look at `?ExpressionSet` to see what other methods are available.

```
> featureNames(sample.ExpressionSet)
> sampleNames(sample.ExpressionSet)
```

A Technical Note

- R is a pass by value language - meaning that when we pass an argument into a function we actually pass a copy of that argument.
- This can be exactly what we do not want when dealing with large data sets.
- Bioconductor classes try to optimize for this by using Environments.

```
> class(assayData(sample.ExpressionSet))
```

```
[1] "environment"
```


A Technical Note

```
> exprsTmp <- exprs(sample.ExpressionSet)
> centerSamples <- function(es, sel = 1:10) {
+     es <- es[, 1:10]
+     exprs(es) <- (exprs(es) - colMeans(exprs(es)))
+     return(es)
+ }
> nExprSet <- centerSamples(sample.ExpressionSet)
```

What does this code do? If I call `all(exprsTmp == exprs(sample.ExpressionSet))` immediately after the last line is it TRUE or FALSE? What about `all(exprs(sample.ExpressionSet) == exprs(nExprSet))`?

Environments

- Environments are essentially hashtables - When implementing a language the key-value mechanism is often called an environment and thus the name in R.
- We can use as an environment as a hashtable although things get much more confusing unless we understand some of the nuances of environments.
- The main confusion is the "chaining" of environments, i.e. an environment has a pointer to its parent environment - by default the environment where it was created.

```
> a <- new.env()
> b <- new.env(parent = emptyenv())
> NAME <- "jim"
> ls(a)
```


Environments

```
character(0)
```

```
> ls(b)
```

```
character(0)
```

```
> get("NAME", a)
```

```
[1] "jim"
```

```
> tryCatch(get("NAME", b), error = function(e) {
+     print("couldn't find the name!")
+ })
```

```
[1] "couldn't find the name!"
```

Now what happens with function calls ...

Environments

```
> x <- new.env(parent = emptyenv())
> setValEnv <- function(name, value,
+     e) {
+     assign(name, value, e)
+ }
> setValEnv("x.1", 10, x)
> y <- list()
> setValList <- function(name, value,
+     l) {
+     l[[name]] <- value
+ }
> setValList("y.1", 10, y)
> get("x.1", x) == y[["y.1"]]
```

Super Extra Credit: Is this TRUE, FALSE, or something else?

Environments

- The take home message is that environments do not behave exactly as most R data structures.

Back to Bioinformatics

- Gene Ontology information is often a necessary component of any bioinformatics microarray analysis.
- A common case is that we have performed our microarray analysis - done some statistics to obtain a list of genes and now we want to do something with those genes.
- One important thing that might come to mind is what do the genes do?

hgu95av2 is an R package which contains a number of environments that provide mappings between manufacturer identifiers (Affymetrix ProbeSet Identifiers) and the various IDs for differing annotation.

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("hgu95av2")
```


Back to Bioinformatics

Here we use the environments "hgu95av2GO" and "hgu95av2ENTREZID" to map our Affymetrix probeset identifiers into GO ids and EntrezGene identifiers.

```
> library("GO")
> library("hgu95av2")
> psNames <- featureNames(sample.ExpressionSet)
> cProbes <- grep("AFFX", psNames)
> sExprSet <- sample.ExpressionSet[-cProbes,
+     ]
> goIDs <- Filter(function(x) !is.null(x),
+     lapply(mget(featureNames(sExprSet),
+         hgu95av2GO), names))
> entrezGeneIDs <- mget(featureNames(sExprSet),
+     hgu95av2ENTREZID)
```

Back to Bioinformatics

Now we can have a look using the GOTERMS environment.

```
> mget(goIDs[[1]], GOTERM)
```

```
$`GO:0006955`
An object of class "GOTerms"
Slot "GOID":
[1] "GO:0006955"

Slot "Term":
[1] "immune response"

Slot "Ontology":
[1] "BP"

Slot "Definition":
```


Back to Bioinformatics

```
[1] "Any immune system process that functions in the

Slot "Synonym":
character(0)

Slot "Secondary":
character(0)


$`GO:0009887`
An object of class "GOTerms"
Slot "GOID":
[1] "GO:0009887"

Slot "Term":
```


Back to Bioinformatics

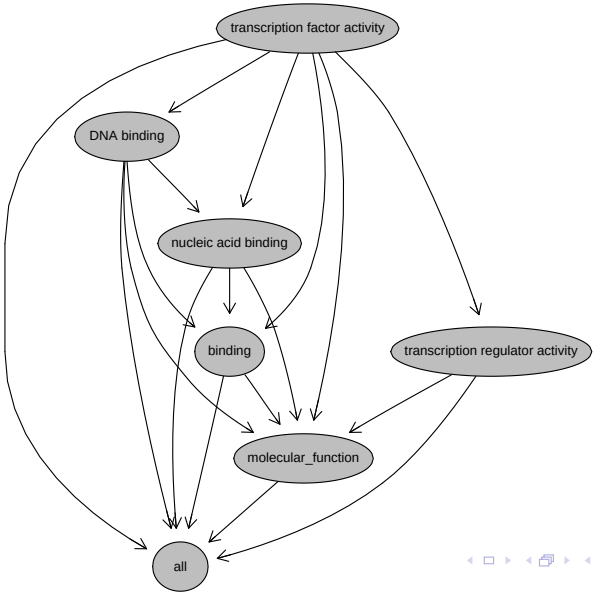
```
[1] "organ morphogenesis"

Slot "Ontology":
[1] "BP"

Slot "Definition":
[1] "Morphogenesis of an organ. An organ is defined a

Slot "Synonym":
[1] "histogenesis and organogenesis"

Slot "Secondary":
character(0)
```


The GO Graph

- The GO graph is a DAG - we can visualize the dag for a particular GO ID using the Bioconductor package Rgraphviz and the Bioconductor package GOstats. [3]

```
> library(GOstats)
> library(Rgraphviz)
> gGraph <- GOGraph("GO:0003700",
+     GOMFANCESTOR)
> terms <- eapply(GOTERM, Term)
> mt <- match(nodes(gGraph), names(terms))
> nattr <- makeNodeAttrs(gGraph,
+     label = terms[mt], shape = "ellipse",
+     fillcolor = "grey", fixedsize = FALSE)
> plot(gGraph, nodeAttrs = nattr)
```

The GO Graph


transcription factor activity
DNA binding
nucleic acid binding
binding
transcription regulator activity
molecular_function
all