Textual Description of Biobase

February 10, 2006

Introduction

Biobase is part of the Bioconductor project. It is meant to be the location of any reusable (or non–specific) functionality. Biobase will be required by most of the other Bioconductor libraries.

1 Data Structures

Part of the Biobase functionality is the standardization of data structures for genomic data. Currently we have designed some data structures to handle microarray data.

The *exprSet* class has the following slots:

exprs A matrix of expression levels. Arrays are columns and genes are rows.

se.exprs A matrix of standard errors for expressions if they are available. It will have length 0 if they are not.

phenoData An object of class `phenoData` that contains phenotypic and/or experimental data.

description A description of the experiment (object of class MIAME)

annotation A character string indicating the base name for the associated annotation.

notes A set of notes describing aspects or features of the data, the analysis, processing done, etc.

These data are extremely large and complex. To deal with them effectively we will need better tools for combining data and documentation. The `exprSet` class represents an initial attempt by the Bioconductor project to provide better tools for documenting and handling these large and complex data sets.

The expression data represent experimentally derived data. In most cases these data will benefit from making use of biologically relevant meta-data. The meta-data are very large and diverse. In order to facilitate interactions and explorations we have taken the approach of constructing a specialized meta-data package for each chip or instrument. Many of these packages are available from the Bioconductor web site. These packages contain information such as the gene name, symbol and chromosomal location. There are other meta-data packages that contain the information that is provided by other initiatives such as GO and KEGG. These data can then be linked to the `exprSet` via the `annotation` slot.

The *annotate* package provides basic data manipulation tools for the meta-data packages.

The `phenoData` class has the following slots:

pData A dataframe where the rows are cases and the columns are variables.

varLabels A list of labels and descriptions for the variables represented by the columns of `pData`.

varMetaData A *data.frame* defining meta-data for the variables contained in the `pData` slot.

Instances of this class are essentially `data.frame`'s with some additional documentation on the variables stored in the `varLabels` and `varMetaData` slots.

A mechanism for ensuring that the elements of the `phenoData` slot of an instance of `exprSet` are in the same order as the columns of the `exprs` array is needed. It is important that these be properly aligned since analyses will require this and automatic tools for checking will probably be better than ad hoc ones.

In addition to the class definitions a number of special methods (or functions) have been defined to operate on instances of these classes. Some particular attention has been paid to subsetting operations. Instances of both `phenoData` and `exprSet` are closed under subsetting operations. That is, any subset of one of these objects retains its class. There are also specialized print methods for objects of both classes.

We consider an instance of an `exprSet` to be an expression array with some additional information. Thus there are two subscripts, one for the rows and one for the columns. For that reason subsetting works in the following ways:

- If the first subscript is given then the appropriate subset of rows from `exprs` and `se.exprs` is taken. All the data in `phenoData` is propagated since no subset of cases was made.
- If the second subscript is given then the appropriate set of columns from `exprs` and `se.exprs` is taken. At the same time the corresponding set of *rows* of `phenoData` are taken.

1.1 An exprSet Vignette

In the data directory for Biobase there is a small anonymized data set. It consists of expression level data for 500 genes on 26 patients. The data can be accessed with the command `data(geneData)`. There are three artificial covariates provided as well. These can be accessed using `data(geneCovariate)` once the Biobase library is attached.

The following vignette shows how to read in these data and to create an instance of the `exprSet` class using those data. The first step is to create an object of `phenoData` calss. This object is used to store the phenotype data. We will use the example data frame `geneCovariate` to create such an object.

```
> data(geneCovariate)
> head(geneCovariate)
```

```
     sex    type score
A Female Control  0.75
B   Male    Case  0.40
C   Male Control  0.73
D   Male    Case  0.42
E Female    Case  0.93
F   Male Control  0.22
```

```
> phenoD <- new("phenoData", pData = geneCovariate, varLabels = list(sex = "Female/Male",
+     type = "Case/Control", score = "Testing Score"))
> phenoD
```

```
        phenoData object with 3 variables and 26 cases
        varLabels
               sex: Female/Male
               type: Case/Control
               score: Testing Score
```

We also need a `MIAME` object to represent the background information, such as the names of the experimenter and laboratory, for this data set.

```
> descr <- new("MIAME", name = "Pierre Fermat", lab = "Francis Galton Lab",
+     contact = "pfermat@lab.not.exist", title = "Smoking-Cancer Experiment",
+     abstract = "An example object of expression set (exprSet) class",
+     url = "www.lab.not.exist", other = list(notes = "An example object of expression set (exprSet) c
> descr

Experimenter name: Pierre Fermat
Laboratory: Francis Galton Lab
Contact information: pfermat@lab.not.exist
Title: Smoking-Cancer Experiment
URL: www.lab.not.exist

A 8 word abstract is available. Use 'abstract' method.
```

Moreover, we need a `data.frame` object to group the reporters (probes) at this data set (chip). The definition of probe types is applied from *Affymetrix GeneChip Expression Analysis Data Analysis Fundamentals* (`http://www.affymetrix.com/Auth/support/downloads/manuals/data_analysis_fundamentals_manual.pdf`), and the result is stored in the `data.frame` object `reporter`. This is a one-column data frame. The rows represent the probe ids in the data set, and the values in the data frame are the predefined types associated with each probe.

```
> data(reporter)
> head(reporter)

               probeType
AFFX-MurIL2_at         QC
AFFX-MurIL10_at        QC
AFFX-MurIL4_at         QC
AFFX-MurFAS_at         QC
AFFX-BioB-5_at         QC
AFFX-BioB-M_at         QC
```

The data frame `geneData`, containing 500 probes and 26 samples, is a subset of real expression data from an Affymetrix U95v2 chip. The `seD` is a *matrix* object containing the standard errors of `geneData`. Finaly, we can put all objects together to create the `exprSet` object.

```
> data(geneData)
> data(seD)
> exSet <- new("exprSet", exprs = geneData, se.exprs = seD, phenoData = phenoD,
+     annotation = "hgu95av2", description = descr, notes = descr@other$notes,
+     reporterInfo = reporter)
> exSet

Expression Set (exprSet) with
        500 genes
        26 samples
                phenoData object with 3 variables and 26 cases
         varLabels
                sex: Female/Male
                type: Case/Control
                score: Testing Score
```

If instead, you have the data stored in files, e.g. `.csv` files and want to read it in using `read.table` you will need to do some conversion. `read.table` reads data into a `data.frame`, and this is fine for the phenotypic data, but not for the expression data. You will need to convert the `data.frame` into a matrix. The code example gives a short example of the commands that you would need to execute if you have the expression data stored in a file named `myexprs.csv` and the phenotypic data in a file named `mycovs.csv`.

```
myExprs = read.csv("myexprs.csv")
myExprsmat = as.matrix(myExprs)
myCovs = read.csv("mycovs.csv")

if( ncol(myExprsmat) != nrow(myCovs) )
  print("ERROR: these should be the same")
```

You should probably also check to ensure that you have the appropriate samples names used as row names for `myCovs` and as column names for `myExprsmat`, and they must be in the same order. So, again some pseudo-code that gives the essential ingredients is given next. If you are unclear on the difference between a `matrix` and a `data.frame` then you should consult one of the very many introductory texts on using R as well as the manual pages and documents that come with R.

```
 if( !all.equal( row.names(myCovs), dimnames(myExprsmat)[[2]] ) )
   print("ERROR: samples names are wrong")
```

Now, we are almost ready to create an instance of an `exprSet` object. The missing piece is a list of descriptions for the covariates that constitute the phenotypic data. Suppose that we have two variables, one the age of the patient in years and the other the particular translocation they are known to have. And suppose that in the `data.frame` `myCovs` the column names are `age` and `translocation`. Then we could make up the covariate descriptions as follows.

```
covDesc = list(age="age of the patient in years",
   translocation="known translocation")
```

The purpose of this level of documentation is to help others who might want to re-analyze your data, or you, if at some time in the future you need to go back and see what was done. By carefully annotating the data, in a self-describing way, you increase the chances that you will understand what was done. Below is the simplest form for creating an `exprSet`. We note that typically you would fill in more details, like the name of the chip used and some additional data describing the experiment itself.

```
myphenoData = new("phenoData", pData=myCovs, varLabels=covDesc)

myEset = new("exprSet", exprs=myExprsmat, phenoData=myphenoData)
```

2 Aggregate

When performing an interative computation such as cross–validation or bootstrapping it is often useful to be able to aggregate certain intermediate results. The `Aggregate` functions (and soon the Aggregate class) provide some simple tools for doing this.

The strategy employed is to maintain the summary statistics in an environment. This is passed to the iterative function. It does not need to be returned since environments have a *pass–by–reference* semantic. Once the function has finished the environment can be queried for the summary statistics.

One simple task that people often want to carry out is to determine in a cross–validation calculation which genes are selected the most often. In some sense these genes may form a more stable basis for inference. Achieving that using an Aggregator is very straight forward.

At each iteration we will pass the names of the selected genes to the Aggregator. It has two functions, one for initializing and one for updating (or aggregating). The aggregator also has an environment. This environment stores the data that is being aggregated.

For our cross–validation example the process goes as follows:

1. At each iteration Aggregate is called with the list of genes selected.
2. For each gene in that list we check to see if it was selected before.
   (a) If not then `initfun` is called with that gene name. The value returned by `initfun` is then associated with the gene name in the aggregation environment.
   (b) If so, then the current value is obtained and `agfun` is called with the the gene name and the current value. This returns a new value that is then associated with the gene name in the aggregation environment.

Basically we are using this as a form of updating hash table. At the same time we are slightly subverting R's usual pass–by–value semantics.

3 Session Information

The version number of R and packages loaded for generating the vignette were:

- R version 2.2.1, 2005-12-20, `i686-apple-darwin8.4.1`
- Base packages: base, datasets, grDevices, graphics, methods, stats, tools, utils
- Other packages: Biobase 1.8.0