Textual Description of Biobase

October 28, 2003

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`.

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

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(geneCov)` 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.

```
> data(geneCov)
> data(geneData)
> covN <- list(cov1 = "Covariate 1; 2 levels", cov2 = "Covariate 2; 2 levels",
+     cov3 = "Covariate 3; 3 levels")
> pD <- new("phenoData", pData = geneCov, varLabels = covN)
> eSet <- new("exprSet", exprs = geneData, phenoData = pD)
```

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.