Bioconductor

Open source bioinformatics using R

by Robert Gentleman and Vincent Carey

Introduction

One of the interesting statistical challenges of the early 21st century is the analysis of genomic data. This field is already producing large complex data sets that hold the promise of answering important questions about how different organisms function. Of particular interest is the ability to study diseases at a new level of resolution.

While the rewards appear substantial there are large hurdles to overcome. The size and complexity of the data may prevent many statisticians from getting involved. The need to understand a reasonable (or unreasonable) amount of biology in order to be effective will also be a barrier to entry for some.

We have recently begun a new initiative to develop software tools to make the analysis of these data sets easier and we hope to substantially lower the barrier to entry for statisticians. The project is called Bioconductor, www.bioconductor.org. We chose the name to be suggestive of cooperation and we very much hope that it will be a collaborative project developed in much the same spirit as R.

In this article we will consider three of the Bioconductor packages and the problems that they were designed to address. We will concentrate on the analysis of DNA microarray data. This has been the focus of our initial efforts. We hope to expand our offerings to cover a much wider set of problems and data in the future. The problems that we will consider are:

- data complexity: the basic strategy here is to combine object oriented programming with appropriate data structures. The package is **Biobase**.
- utilizing biological meta data: the basic strategy here is to use databases and hash tables to link symbols. The package is **annotate**.
- gene selection: given a set of expression data (generally thousands of genes) how do we select a small subset that might be of more interest? This is equivalent to ranking genes or groups of genes in some order. The package is **genefilter**.
- documentation: as we build a collection of interdependent but conceptually distinct packages, how do we ensure durable interoperation? We describe a *vignette* documentation concept, implemented using `Sweave` (a new function in R contributed by F. Leisch).

An overview of this area, provided by the European Bioinformatics Institute, may be found at http://industry.ebi.ac.uk/~brazma/Biointro/biology.html Also, a recent issue of the Journal of the American Medical Association (November 14, 2001) has a useful series of pedagogic and interpretive papers.

Handling data complexity

Most microarray experiments consists of some number of samples, usually less than 100, on which expression of messenger RNA (mRNA) has been measured. While there are substantial interesting statistical problems involving the design and processing of the raw data we will concentrate on the data that have been processed to the point that we have comparable expression data for a set of genes for each of our samples. For simplicity of exposition we will consider a clinical setting where the samples correspond to patients.

Our primary genomic data resource then consists of an expression array which has several thousand rows (representing genes) and columns representing patients. In addition we have data on the patients, such as age, gender, disease status, duration of symptoms and so on. These data are conceptually and often administratively distinct from the expression data (which may be managed in a completely different database). We will use the term *phenotypic* data to refer to this resource, acknowledging that it may include information on demographics or therapeutic conditions that are not really about phenotype.

What are efficient approaches for coordinating access to genomic and phenotypic data as described here? It is possible to couple the genomic and phenotypic data for each patient as a row in a single data frame, but this seems unnatural. We recognize that the general content and structure of expression data and phenotype data may evolve in different ways as biological technology and research directions also change, and that the corresponding types of information will seldom be treated in the symmetric fashion implied by storing them in a common data frame. Our approach to data structure design for this problem preserves the conceptual independence of these distinct information types and exploits the *formal methods and classes* in the package **methods** which was developed by J. M. Chambers.

Class `exprSet` and its methods

The **methods** package provides a substantial basis for object-oriented programming in R. Object-oriented programming has long been a tool for deal-

ing with complexity. If we can find a suitable representation for our data then we can think of it as an object. This object is then passed to various routines that are designed to deal with the special structure in the objects.

An object has *slots* that represent its different components. To coordinate access to the genomic and phenotypic data generated in a typical microarray experiment, we defined a class of objects called `exprSet`. An instance of this class has the following slots:

exprs An array that contains the estimated expression values, with columns representing samples and rows representing genes.

se.exprs An array of the same size as `exprs` containing estimated standard errors. This may be `NULL`.

phenoData An instance of the `phenoData` class which contains the sample level variables for this experiment. This class is described subsequently.

description A character variable describing the experiment. This will probably change to some form of documentation object when and if we adopt a more standard mechanism for documentation.

annotation This is a character string indicating the name of the annotation data that can be used for this `exprSet`.

notes A character string for notes regarding the analysis or for any other purpose.

Once we have defined the class we may create instances of it. A particular data set stored in this format would be called an instance of the class. While we should technically always say something like: *x is an instance of the* `exprSet` *class*, we will often simply say that *x is an* `exprSet`.

We have implemented a number of methods for the `exprSet` and `phenoData` classes. Most of these are preliminary and some will evolve as our understanding and the data analytic processing involved mature. The reader is directed to the documentation in our packages for definitive descriptions.

We would like to say a few words here about the subsetting methods. By implementing special subsetting methods we are able to ensure that the data remain correctly aligned. So, if `x` is an instance of an `exprSet`, we may ask for `x[,1:5]`. The second index in this subscripting expression refers to tissue samples. The value of this expression is an `exprSet` whose contents are restricted to the first five samples in `x`. The new `exprSet`'s `exprs` element contains the first five columns of `x`'s `expr` array. The new `exprSet`'s `se.exprs` array is similarly restricted. But the `phenoData` must also have a subset made and for it we want the first five *rows* of `x`'s phenoData data frame, since it is in the more standard format where columns are variables and rows are cases.

Notice that by representing our data with a formal class we have removed a great deal of the complexity associated with the data. We believe that this representation relieves the data analyst of complex tasks of coordination and checking of diverse data resources and makes working with `exprSets` much simpler than working with the raw data.

We have also defined the `$` operator to work on exprSets. This operator selects variables of the appropriate name from the `phenoData` component of the `exprSet`.

The `phenoData` class was designed to hold the phenotypic (or sample level) data. Instances of this class have the following slots:

pData This is a `data.frame` with samples as the rows and the phenotypic variables as the columns.

varLabels This is a list with one element per phenotypic variable. Names of list elements are the variable names; list element values are character strings with brief textual descriptions of the associated variables.

We find it very helpful to keep descriptions of the variables associated with the data themselves. Such a construct could be helpful for all `data.frames`.

Example

The **golubEsets** package includes `exprSets` embodying the leukemia data of the celebrated Science paper of T. Golub and colleagues (Science 286:531-537, 1999). Upon attaching the `golubTrain` element of the package, we perform the following operations.

Show the data. This gives a concise report.

```
> golubTrain
Expression Set (exprSet) with
 7129 genes
 38 samples
  phenoData object with 11 variables and 38 cases
  varLabels
   Samples: Sample index
   ALL.AML: Factor, indicating ALL or AML
   BM.PB: Factor, sample from marrow or \
     peripheral blood
   T.B.cell: Factor, T cell or B cell leuk.
   FAB: Factor, FAB classification
   Date: Date sample obtained
   Gender: Factor, gender of patient
   pctBlasts: pct of cells that are blasts
   Treatment: response to treatment
   PS: Prediction strength
   Source: Source of sample
```

Subset expression values. This gives a matrix. Note the gene annotation, in Affymetrix ID format.

```
> exprs(golubTrain)[1:4,1:3]
               [,1] [,2] [,3]
AFFX-BioB-5_at -214 -139  -76
AFFX-BioB-M_at -153  -73  -49
AFFX-BioB-3_at  -58   -1 -307
AFFX-BioC-5_at   88  283  309
```

Tabulate stratum membership. This exploits the redefined $ operator.

```
> table(golubTrain$ALL.AML)

ALL AML
 27  11
```

Restrict to the ALL patients. We obtain the dimensions after restriction to patients whose phenoData indicates that they have acute lymphocytic leukemia.

```
> print(dim(exprs(golubTrain[ ,
             golubTrain$ALL.AML=="ALL"])))

[1] 7129   27
```

Other tools for working with `exprSets` are provided, including methods for iterating over genes with function application, and sampling from patients in an `exprSet` (with or without replacement). The latter method returns an `exprSet`.

Annotation

Relating the genes to various biological data resources is essential. While there is much to be learned and many of the biological databases are very incomplete it is never the less essential to start employing these data. Again our approach is quite simple but it has proven effective and we are able to both analyse the data we encounter and to provide resources to others.

Our approach has been to divide the process into two components. One process is the building and collation of annotation data from public databases, and the other process is the extraction and formatting of the collated data for analysis reporting. Data analysts using Bioconductor will typically simply obtain a set of annotation tables for the chip data they are using. Given the appropriate tables they can select genes according to certain conditions such as functional group, or biological process or chromosomal location.

The process of building annotation data sets is carried out by the **AnnBuilder** package. We distribute this but most users will not want to build their own annotation. They will want instead to have access to specific annotation for their chip. **AnnBuilder** relies on several R packages (available from CRAN) including D. Temple Lang's **XML** and T. Keitt's **RPgSQL**.

Typically the annotation data available are a complete set for all genomes or a complete set for a specific genome, such as yeast or humans. Since this is typically much larger than what is currently available on any microarray and in the interest of performance we have found that producing annotation collections at the level of a chip has been quite successful.

The data analyst simply needs to ensure that they have an annotation collection suitable for their chip. A number of these are available from the Bioconductor web page and given the appropriate reference data we can provide custom made collections within a few days.

Currently these collections arrive as a set of comma separated files. The first entry in each row is the identification name or label for the gene. This should correspond to the row names of the `expr` value in the `exprSet` that is being analysed. The annotation filenames should have their first five letters identical to the value in the `annotation` slot of the `exprSet` since this is used to align the two. This package will become more object oriented in the near future and the distribution mechanism will change.

Affymetrix Inc. produces several chips for the human genome. The most popular in current use are the U95v2 chips (with version A being used most often). The probes that are arrayed here have Affymetrix identifiers. We provide functions that map the Affymetrix identifiers to a number of other identifiers such as the Locus Link value, the GenBank value. In addition we have mapped most probes to their Genome Ontology (GO) values http://www.geneontology.org. GO is an attempt to provide standardized descriptions of the biological relevance of genes into the following three categories:

- Biological process
- Cellular component
- Molecular function

Within a category the set of terms forms a directed acyclic graph. Genes typically get a specific value for each of these three categories. We trace each gene to the top (root node) and report the last three nodes in its path to the root of the tree. These top three nodes provide general groupings that may be used to examine the data for related patterns of expression.

Additional data, such as chromosomal location, is also available. As data become available we will be adding it to our offerings. Readers with knowledge of data that we have not included are invited to submit this information to us.

The annotation files are read in to R as environments and used as if they were hash tables. Typically the name they are stored under is the identifier (but that does not need to be the case). The

value is can then be accessed using `get`. Since we often want to get or put multiple values (if we have 100 genes we would like to get chromosome location for all of them) there are functions `multiget` and `multiassign`.

Often researchers are interested in finding out more about their genes. For example they would like to look at the different resources at NCBI. We can easily produce HTML pages with active links to the different online resources. `ll.htmlpage` is an example of a function designed to provide links to the Locus Link web page for a list of genes.

It is also possible using connections and the **XML** package to open http connections and read the data from the web sites directly. It could then be processed using other R tools. For example if abstracts or full text searchable articles were available these could be downloaded and searched for relevant terms.

The remainder of this section involves reference to Web sites and R functionality that may change in the future. This portion of the project and the underlying data are in a state of near constant change. An up-to-date version of the material in this section, along with relevant scripts and data images, will be maintained at http://www.bioconductor.org/rnews0102.html. If you have problems with any of the commands described below, please consult this URL.

To illustrate how one can get started analyzing publicly distributed expression data with R, the following commands lead to a matrix `numExprs` containing the Golub training data, and a character vector of Affymetrix expression tags `accTags`:

```
golURL <-
  url(paste("http://www-genome.wi.mit.edu/",
            "mpr/data_set_ALL_AML_train.txt",
            sep = ""),
      "r")
golVec <- scan(golURL, sep = "\t", what = "")
## next command patches up a missing blank
golVec <- c(golVec[1:78], "", golVec[-(1:78)])
golMat <- t(matrix(golVec, nr = 79))
accTags <- golMat[-1, 2]
cExprs <- golMat[-1, seq(3, 78, 2)]
numExprs <- t(apply(cExprs, 1, as.numeric))
```

Let's see what the Affymetrix tags look like:

```
t(t(accTags[c(10,20,30)]))
     [,1]
[1,] "AFFX-BioB-5_st"
[2,] "AFFX-DapX-5_at"
[3,] "AFFX-ThrX-M_at"
```

Using the `annotate` package of Bioconductor, we can map these tags to GenBank identifiers:

```
library(annotate)
library(Biobase)
HGu952genBank() # set up map
multiget(accTags[c(10,20,30)],
         env=HGu95togenBank)
```

The result is a list of GenBank identifiers:

```
$"AFFX-BioB-5_st"
[1] "J04423"

$"AFFX-DapX-5_at"
[1] "L38424"

$"AFFX-ThrX-M_at"
[1] "X04603"
```

Upon submitting the tag "J04423" to GenBank, we find that this gene is from E. coli, and is related to biotin biosynthesis.

A long range objective of the Bioconductor project is to facilitate the production and collation of interpretive biological information of this sort at any stage of the process of filtering and modeling expression data.

Gene filtering

In this section we consider the process of selecting genes that are associated with clinical variables of interest. The package **genefilter** is one approach to this problem. We think of the selection process as the sequential application of filters. If a gene passes all of the filters then we deem it interesting and select it.

For example, genes are potentially interesting if they show some variation across the samples that we have. They are potentially interesting if they have estimated expression levels that are high enough for us to believe that the gene is actually expressed. These are both examples of non-specific filters; we have not made any reference to a covariate. Covariate-based filters specify conditions of magnitude or variation in expression within or between strata defined by covariates that must be satisfied by a gene in order that it be retained for subsequent filtering and analysis.

Closures for non-specific filters

A filter function will usually have some context-specific parameters to define its behavior. To make it easy to associate the correct settings with a filter function we have used R lexical scoping rules. Here is the function `kOverA` that filters genes by requiring that at least k of the samples have expression values larger than A.

```
kOverA <- function(k, A=100, na.rm = TRUE) {
  function(x) {
    if(na.rm)
      x <- x[!is.na(x)]
    sum( x > A ) > k
  }
}
```

To define a filter that requires a minimum of five samples to have expression level at least 150, we simply evaluate the expression

```
myK <- kOverA(5, 150)
```

Now myK is a function, since that is what kOverA returns. myK takes a single argument, x, which will be a vector of gene expressions (one expression level for each sample) and evaluates the body of the inner function in kOverA. In that body A, k and na.rm are unbound symbols. They will obtain their values as those that we specified when we created the closure (the coupling of the function body with its enclosing environment) myK. If es is an exprSet, the result of apply(exprs(es),1,myK) is a logical vector with gth element TRUE if gene g (with expression levels stored in row g of the exprs slot of es) has expression level at least 150 in 5 of the samples, and FALSE otherwise.

Another more interesting non-specific filter is the gap filter. It selects genes using a function that has three parameters. The first is the gap, the second is the IQR and the third is a proportion. To apply this filter the (positive) expression levels for the gene are sorted and the proportion indicated are removed from both tails. If in the remainder of the data there is a gap between adjacent values of at least the amount specified by the gap parameter then the gene will pass the filter. Otherwise, if the IQR based on all values is larger than the specified IQR the gene passes the filter. The purpose of this filter is to select genes that have some variation in their expression values and hence might have something interesting to say.

Covariate-dependent filters

An advantage to this method is that any statistical test can be implemented as a filter. Functions implementing covariate-dependent filters are built through closures, have the same appearance as simpler non-specific filters, and are apply'd to expression level data in precisely the same way as described above.

Here is a filter-building function that uses the partial likelihood ratio to test for an association between a censored survival time and gene expression levels.

```
coxfilter <- function (surt, cens, p)
{
  autoload("coxph", "survival")
  function(x) {
    srvd <- try(coxph(Surv(surt, cens) ~ x))
    if (inherits(srvd, "try-error"))
      return(FALSE)
    ltest <- -2 * (srvd$loglik[1] -
                        srvd$loglik[2])
    pv <- 1 - pchisq(ltest, 1)
    if (pv < p)
      return(TRUE)
    return(FALSE)
  }
}
```

To use this method, we must create the closure by establishing bindings for the survival time, censoring indicator, and significance level for declaring an association. Ordinarily, the survival data will be captured in the phenoData slot of an expression set, say es, so we will see something like

```
myCox <- coxfilter( es$stime, es$event, 0.05 )
```

Now apply(exprs(es), 1, myCox) is a logical vector with element TRUE at index g whenever gene g is associated with survival, and FALSE otherwise.

We have provided other simple filters such as a t-test filter and an ANOVA filter. One can simply set up the test and require that the p-value of the test when applied to the appropriate covariate (from the phenoData slot) is smaller than some prespecified value.

In one designed experiment we used testing within gene-specific non-linear mixed effects models to measure the association between gene expression and a continuous outcome. The amount of programming required was minimal. Filtering in this simple way provides us with a very natural paradigm and tool in which to carry out gene selection. Note however, that in some situations we will need to deal with more than one gene at a time.

Equipped with the lexical scoping rules, we have separated the processes of generic filter specification (filter building routines supplied in the **genefilter** package), selection of detailed filter parameters (binding of parameters when the closure is obtained from the filter builder), and application of filters over large numbers of genes. For the latter process we have given examples of "manual" use of apply, but more efficient tools for sequential filtering are supplied through the filterfun and genefilter functions in the package.

A word of warning is in order for those who have not worked extensively with function closures. It is easy to forget that the function is a closure and has some variables captured. Seemingly correct results can obtain if you are not careful. The alternative is to specify all parameters and strata at filtering time. But we find that makes the filtering code much more complicated and less intuitive.

Documentation

We are embarked on the production of packages and protocols to facilitate greater involvement of statisticians in bioinformatics, and to support smoother collaboration between statisticians, biologists and programmers when working with genomic data. Documentation of data structures, methods, and analysis procedures must be correct, thorough, and accessible if this project is to succeed.

R has a number of unique and highly effective protocols for the creation (prompt) and use of documents at the function and package level that facilitate simple interactive demonstration of function behaviors (the example function, which applies to help

topics) and unit testing of functions and packages at build time (the testing carried out by `R CMD check`).

We have introduced two new documentation protocols as we grow the Bioconductor project.

promptClass

A revised `promptClass` function has been developed which produces a template that illustrates instantiation of classes (using the `new` function) and searches through the searchlist to obtain information on formal methods that apply to the class being documented. Thus the creator of a new class is motivated to describe not only the structure of the class being created, but also the family of methods that may be used to work with this class.

The *vignette* protocol

The intent of the `CMD check` unit testing protocols of R is to establish that the elements of a package function in accordance with the description in the manual pages. The Bioconductor project has two additional unit testing requirements. First, packages must work together in well-defined ways: examples involving multiple packages and diverse data structures must interoperate correctly and continuously while evolving. Second, users will require 'higher-level' views of programming processes than are typically afforded by package man pages. Users will need to be taken through the steps of data structure creation, searchlist extension, and then through the details of any particular data processing or inference procedure, with considerable narrative guidance.

Sweave allows integrated session narration, shell monitoring, and graphics incorporation in documents that are specified in LaTeX and are renderable using PDF. We propose that Sweave documents (with suffix '.Rnw') and PDF compilations of them be kept in the 'inst/doc' subdirectory of packages related to the Bioconductor project.

Conclusions

Analytical computing for bioinformatics has been characterized to date by the proliferation of separated binaries or scripts that implement various analysis methods in very easy to use formats (e.g., MicroSoft Excel modules, Java applets). This paradigm has a number of drawbacks, including frequent reliance upon idiosyncratic data input and output formats, high costs of establishing interoperability, difficulties of incorporating new methodological insights into existing programs, minimal infrastructure to verify procedure portability, and non-standard documentation.

The Bioconductor project is based on the award-winning interactive programming language S, as deployed in the open-source statistical computing environment R. An interactive programming language provides an ideal platform for prototyping implementations of new ideas and comparing the new approaches to well-established older approaches. The language basis confers well-defined paths to extension and interoperation of any routines. Procedures that are too intensive to work routinely in the interactive environment may be coded in any higher-level language and loaded dynamically in R for higher performance with no change to the user interface. The object-oriented programming facilities provide access to state-of-the-art methods in data structure and software design that help to control code complexity and reduce obstacles to smooth interoperation of diverse procedures. The intersystems interfacing initiative at `www.omegahat.org` provides tools to permit smooth interoperation of Bioconductor routines with completely foreign systems such as relational databases, browsers, or other virtual machines.

The role of R in bioinformatics research has been substantial, with contributions already present on CRAN (**sma**, **GeneSOM**, etc.) and its use in many projects for functional genomics. It is our hope that the Bioconductor project will aid in the federation of programming efforts to support progress in many different areas of biology and clinical research. Those who are interested in this project should visit our web site and consider subscribing to the bioconductor mailing list.

Acknowledgment

Vincent Carey's work on Bioconductor is supported in part by US National Heart Lung and Blood Institute Grant HL66795, Innate Immunity in Heart Lung and Blood Disease.

Robert Gentlemans's work is supported in part by NIH/NCI Grant 2P30 CA06516-38.

Robert Gentleman
DFCI
`rgentlem@jimmy.harvard.edu`

Vincent Carey
Channing Lab
`stvjc@channing.harvard.edu`