genBart package Vignette

Jacob Cardenas

2017-12-07

BART (Biostatistical Analysis Reporting Tool) is a user friendly, point and click, R shiny application. With the genBart package, biostatisticians/bioinformaticians analyze high throughput biological data obtained from RNA-Seq, Microarray, Flow Cytometry, and metabolomic experiments and upload the result into BART. The tool allows the R programmer who conducted the analysis and their colleagues to efficiently examine the results. BART provides users the ability to easily view, modify and download the tables and figures generated by the app. BART offers one reporting source for all analyses in a project workflow.

- Sample information (e.g. technical, biological, clinical, and demographics) summary statistics
- Sample quality control metrics
- Unsupervised analysis (heatmap and cluster analysis)
- Differential expression (sortable genelists, venn diagrams)
- Gene set analysis
- Correlation analysis within and between biological data types

So that BART is readily available, we introduce the genBart package. genBart is a series of functions that transforms R objects generated from statistical and modeling packages such as limma, DESeq2, and edgeR into files that are uploaded into BART. In the following sections, we illustrate how to use the functions available in genBart by walking through the analysis of a longitudinal microarray study tracking gene expression changes in 38 healthy cynomolgus macaques infected with *M. tuberculosis* (Skinner et al.). Briefly, this study consisted of data measurements taken at baseline (before infection) and at 11 additional timepoints, up to 6 months post infection. Each macaque was identified as developing symptomatic (active) TB or asymptomatic (latent) infection. For the purposes of illustration and time, a subset of the microarray data was chosen for analysis. We randomly selected 4000 probes and 10 macaques and include 3 time points (days 0, 20, and 42). For the flow, data was collected on 19 of the 38 macaques and 13 variables are analyzed. In addition to microarray data, the study also has flow data that will be incorporated to demonstrate the flexibility of the BART app.

Summary of genBart package

Table 1: Functions in the genBart package ordered to demonstrate a typical analysis workflow.

Function	Purpose
`metaData()`	Aligns design and expression files. Defines elements of design for further analyses.
`normalizeData()`	Normalizes data.
`clusterData()`	Clusters normalized expression data.
`genModScores()`	Generates sample level module scores for plotting.
`genModelResults()`	Processes and formats differential expression modeling results.
`qBart()|rBart()`	Runs gene set analysis using QuSAGE or ROAST algorthms.
`crossCorr()`	Runs pairwise cross platform correlations (e.g. metabolites vs gene expression).
`genFile()`	Generates and saves BART ready R data file.
`updateFile()`	Updates/adds analyses to existing BART file.
`runBart()`	Runs BART shiny app.

The genBart package contains 11 functions, each of which is listed in the table above. Greater detail on each of these functions, their parameters, and how they are integrated into a typical workflow is available via the help commands in R.

Design Information

Before running any analysis, the first step to generate a BART ready file is defining the design and sample information elements. Keep in mind that not every element is necessary for the rest of the genBart functions to run, so in practice, much of this section is left to user discretion and the particular project at hand.

```
library(genBart)
library(limma)
```

Below are the first six lines of the design file from the TB microarray study described in the introduction.

```
head(tb.design)
#>                    columnname monkey_id timepoint sample_id clinical_status
#> 1  AVG_Signal__7196763044_K        M1         0   M1-Pre1          Active
#> 5  AVG_Signal__6303256034_B        M1        20    M1-D20          Active
#> 7  AVG_Signal__6303256020_I        M1        42    M1-D42          Active
#> 22 AVG_Signal__7196763078_I       M11         0  M11-Pre1          Latent
#> 26 AVG_Signal__7196771011_D       M11        20   M11-D20          Latent
#> 28 AVG_Signal__7196763087_B       M11        42   M11-D42          Latent
#>    timerange    Group
#> 1        Pre  Active0
#> 5      Early Active20
#> 7     Middle Active42
#> 22       Pre  Latent0
#> 26     Early Latent20
#> 28    Middle Latent42
```

For this study, the function call would be:

```
meta <- metaData(y = tb.expr, design = tb.design, data.type = "microarray",
                 columnname = "columnname", long = TRUE, sample.id = "sample_id",
                 subject.id = "monkey_id", time.var = "timepoint",
                 baseline.var = "timepoint", baseline.val = 0)
```

We proceed similarly for the flow data:

```
meta.flow <- metaData(y = tb.flow, design = tb.flow.des, data.type = "flow",
                      columnname = "columnname", long = TRUE, sample.id = "sample_id",
                      subject.id = "monkey_id", time.var = "timepoint",
                      baseline.var = "timepoint", baseline.val = 0)
```

Please refer to the R help for `metaData` for more detail on function parameters.

Expression Normalization and Clustering

Since BART's primary function is to serve as a reporting tool for analyzed data, we normalize and cluster the data beforehand so that users can efficiently sort through various heat maps without having to wait for the computationally intensive task of clustering thousands of genes. In genBart, normalization and hierarchical clustering are completed by two simple functions. First, the object generated by `metaData` containing the expression and design information is input into `normalizeData`. This function normalizes the data in various ways, depending on the study design specified in `metaData`. In our TB microarray example with repeated measurements, `normalizeData` returns three normalized datasets: all samples normalized to the mean or median (specified by `norm.method` argument), baseline samples normalized to the mean or median, and all samples normalized to baseline. For baseline normalization, instead of subtracting by the mean or median of the baseline samples, each sample's own baseline is subtracted. The list returned by `normalizeData` is then input into `clusterData`, which performs hierarchical clustering on each of the normalized datasets and returns a list of dendrograms. For more detail on the various normalization and hierarchical clustering methods, please refer to the R help for `normalizeData` and `clusterData`.

```
norm.data <- normalizeData(meta = meta, norm.method = "mean")
cluster.data <- clusterData(norm.data = norm.data, dist.method = "euclidean", agg.method = "complete")
#> [1] "clustering genes from baseline samples normalized to mean..."
#> [1] "clustering genes from all samples normalized to mean..."
#> [1] "clustering genes from all samples normalized to baseline..."
```

Generate Gene Set Figures

It is often of interest to examine the behavior of genes that have been grouped together based on similar biological function or other metric. The `genModScores` function calculates the percentage of genes within a gene set that are up or down regulated with respect to baseline and/or a set of controls. These *up* or *down* percentages, referred to as a *module scores*, are calculated for every gene set and sample in the study and are plotted within BART as a heatmap. As an example, we formed gene sets by creating a list of 10 clusters formed through hierarchical clustering. The function call to generate the module scores is below.

```
mod.scores <- genModScores(meta = meta, gene.sets = clusters, sd.lim = 2)
```

Refer to the R help for `genModScores` for more detail on function parameters and how the module scores are calculated.

Sample Differential Expression Analysis with limma

Now we demonstrate a quick walkthrough of differential expression analysis of microarray and flow cytometry data using limma. For more details about linear modeling in limma, refer to the limma User's Guide.

Microarray

Create a grouping factor by combining clinical status and time point.

```
tb.design$Group <- paste(tb.design$clinical_status, tb.design$timepoint, sep = "")
grp <- factor(tb.design$Group)
```

Create a design matrix that includes separate coefficients for each level within the grouping factor so that the desired differences can be extracted as contrasts.

```
design2 <- model.matrix(~0+grp)
colnames(design2) <- levels(grp)
```

Since there are repeated measurements on each monkey, we treat monkey_id as a random effect and estimate the correlation between measurements on the same subject. In limma, this is done using duplicateCorrelation.

```
dupcor <- duplicateCorrelation(tb.expr, design2, block = tb.design$monkey_id)
```

Fit model and extract contrasts of interest.

```
fit <- lmFit(tb.expr, design2, block = tb.design$monkey_id, correlation = dupcor$consensus.correlation)
contrasts <- makeContrasts(A_20vsPre = Active20-Active0, A_42vsPre = Active42-Active0,
                           L_20vsPre = Latent20-Latent0, L_42vsPre = Latent42-Latent0,
                           levels=design2)

fit2 <- contrasts.fit(fit, contrasts)
fit2 <- eBayes(fit2, trend = FALSE)
```

Flow Cytometry

Analysis of flow data in limma proceeds similarly.

```r
tb.flow.des$Group <- paste(tb.flow.des$clinical_status, tb.flow.des$timepoint, sep = "")
grp <- factor(tb.flow.des$Group)

design2 <- model.matrix(~0+grp)
colnames(design2) <- levels(grp)
dupcor <- duplicateCorrelation(tb.flow, design2, block = tb.flow.des$monkey_id)

fit.flow <- lmFit(tb.flow, design2, block = tb.flow.des$monkey_id, correlation = dupcor$consensus.correl

# Add more contrasts since we have all timepoints available. Additional contrasts could be added as wel
contrasts <- makeContrasts(A_20vsPre = Active20-Active0, A_30vsPre = Active30-Active0,
                           A_42vsPre = Active42-Active0, A_56vsPre = Active56-Active0,
                           L_20vsPre = Latent20-Latent0, L_30vsPre = Latent30-Latent0,
                           L_42vsPre = Latent42-Latent0, L_46vsPre = Latent56-Latent0,
                           levels=design2)

fit2.flow <- contrasts.fit(fit.flow, contrasts)
fit2.flow <- eBayes(fit2.flow, trend = FALSE)
```

Generate Model Results File

The next step in the process is generating a model results file that is formatted specifically for BART. This can be done using `genModelResults`, which takes as input the expression data used for modeling, as well as model result output from limma, DESeq2, or edgeR. In our example, we ran linear models using limma for both the microarray and flow data.

```r
mod.results <- genModelResults(y = tb.expr, data.type = "microarray", object = fit2, lm.Fit = fit, metho
mod.results.flow <- genModelResults(y = tb.flow, data.type = "flow", object = fit2.flow, lm.Fit = fit.fl
```

Gene Set Analysis: qBart

Next we run gene set analysis by incorporating functions available in the `qusage` package, which tests whether the average log2 fold change of the genes within a gene set are different than 0. `qBart` simplifies the process by requiring two input parameters, the model results object produced by `genModelResults` and a list of gene sets.

```r
qus <- qBart(model.results = mod.results, gene.sets = modules)
#> Aggregating gene data for gene sets.Done.
#> Calculating VIF's on residual matrix.
#> Q-Gen analysis complete.Aggregating gene data for gene sets.Done.
#> Calculating VIF's on residual matrix.
#> Q-Gen analysis complete.Aggregating gene data for gene sets.Done.
#> Calculating VIF's on residual matrix.
#> Q-Gen analysis complete.Aggregating gene data for gene sets.Done.
#> Calculating VIF's on residual matrix.
#> Q-Gen analysis complete.
```

Correlations

In addition to providing tools for standard analysis, BART also offers a tool that allows users to visualize and sort through potentially hundreds of thousands pairwise correlations across multiple platforms (e.g. gene expression vs metabolites) or versus clinical outcomes. `crossCorr` takes two numeric data frames and outputs a long format file of all pairwise correlations. The function has an option to run correlations by a grouping factor and various additional parameters for labeling in BART. We walk through an example in which we wish to correlate module activity scores with flow variables. Since this is a longitudinal setting, we correlate by time.

```
# Create time variable
time <- module.as$time
module.as$time <- NULL

# Format flow data to run correlations and match flow samples with module.as
flow <- data.frame(t(tb.flow))
flow <- flow[match(rownames(module.as), rownames(flow), nomatch = 0), ]

# Run correlations formatted for BART
corrs <- crossCorr(x = module.as, y = flow, by = time, by.name = "days",
                   description = "Mod.Act.Score vs Flow", x.var = "Mod.Act.Score",
                   y.var = "Flow", method = "spearman")
```

Generate BART file

The last step in the process is to generate the final file that can be uploaded into BART. This is done through the function `genFile`, which takes as its arguments the objects generated from the previous functions. Not every object is required to generate a BART file. For example, if gene set analysis and correlations are not run, `metaData`, `clusterData`, `genModScores`, and `genModelResults` objects can still be run through `genFile`. BART will only show the results that are input.

```
genFile(meta = list(meta, meta.flow), module.scores = mod.scores, dendrograms = cluster.data,
        model.results = list(mod.results, mod.results.flow), project.name = "BART example")
```

In our example, we did run gene set analysis and correlations. Adding these results into BART is made simple through the `updateFile` function. The user must simply provide the path to the BART file that needs to be updated and add the new additions in the same way that `genFile` takes them.

```
path <- paste(getwd(), "/", "BART example", sep = "")
updateFile(load.path = path, qusage.results = qus, corr.results = list(corrs))
```

Run BART App

Now that a BART file has been created, it is now time to begin using the app by running the function `runBart`.

```
runBart()
```