Module 10. Microarray Analysis

Overview:

The analysis of gene expression values is of key importance in bioinformatics. In this model we will learn the basic structure and the basic analysis of microarray data. We will study how to preprocess probe data, to filter genes, to program various visualizations, and to use gene ontology identifiers.

Learning Objectives

1. **Preprocess** probe data
2. Apply simple **statistical analysis** for analyzing the microarray data
3. Use **gene ontology** identifiers to search for gene information

Readings:

Applied Statistics for Bioinformatics using R pages 91-115.

Lesson 1: The Microarray Data and Preprocessing

Objectives

By the end of this lesson you will have had the opportunity to:

- Explain the general **structure** of the microarray data set
- **Visualize** the raw microarray data
- **Preprocess a data set**: convert raw microarray data into expression data

Overview

In this module, we will study the build-in microarray data set object in R; its structure, and how to retrieve basic information about it. We will also review how to preprocess the data set; that is, how to get expression data from raw data.

Microarray Data

The microarray technique takes advantage of hybridization properties of nucleic acids. The basic idea is that complementary molecules are attached and labeled on a solid surface, in order for a specialized scanner to measure the intensity of target molecules. Per gene, there are about twenty such measures obtained for each probe (gene). Per probe, these measures come in pairs. The intensity of the perfect match (PM) intends to measure the amount of transcripts from the gene. The intensity of the mismatch (MM) is related to non-specific binding and is often seen as a background type of noise.

The raw data from the Affymetrix scanner is stored in so-called DAT files, which are processed to so-called CEL files, which we will work with. We generally do not want to deal with this raw microarray data, and prefer to work with the expression data. In this lesson we give a brief introduction of how to get the expression data from the raw microarray data. Conversion of the raw data into the expression data is usually referred as **preprocessing**.

Microarray Data Object in R

In R, the microarray data is often stored as an AffyBatch object. You will need to install the package affy from Bioconductor. It is named by Affymetrix Inc., a manufacturer of DNA microarrays. It contains raw measurement data and all of the label information according to a special format. We illustrate the structure of the AffyBatch object using an example data set that comes from the ALLMLL package.

Demonstration: A Microarray Data Object in R

We will start with a built-in data set called MLL.B from the ALLMLL package. You will need to have the packages affy and ALLMLL installed. If needed, you can review the instructions on page 2 of Applied Statistics for Bioinformatics using R using command bioLite("package.name") to install the packages. To load the data set and to retrieve basic information, we use

```
> library(affy); data(MLL.B, package = "ALLMLL");
> MLL.B
AffyBatch object
size of arrays=712x712 features (13 kb)
cdf=HG-U133B (22645 affyids)
number of samples=20
number of genes=22645
annotation=hgu133b
notes=
```

This data set is an AffyBatch object. It contains expression data for 22645 genes in 20 samples.

Next, we will carefully review what are the samples, and then what are the genes.

Demonstration (Continued): The Pheno Data

The phenoData contains information on what the samples are.

```
> phenoData(MLL.B)
An object of class 'AnnotatedDataFrame'
  sampleNames: JD-ALD009-v5-U133B.CEL JD-ALD051-v5-U133B.CEL
...
    JD-ALD520-v5-U133B.CEL (20 total)
  varLabels: sample
  varMetadata: labelDescription
```

There are 20 samples in MLL.B, each saved in a CEL file. The names of CEL files can be found by command sampleNames(phenoData(MLL.B)). For further description of the sample names, use

```
> varMetadata(phenoData(MLL.B))
          labelDescription
sample arbitrary numbering
```

For this particular data set, there is no extra information provided on what the 20 samples are, except some arbitrary labels. This data is in fact a subset of arrays in a large acute lymphoblastic leukemia (ALL) study published in the paper Ross et al. (2003) Classification of pediatric acute lymphoblastic leukemia by gene expression profiling, Blood 102 (8): 2951-2959.

Each sample is one HG-U133B array of gene profiles for one patient with pediatric ALL. The original study contains information regarding who the patient is and what subtype of ALL he/she has, but this information is not provided in this subset of data MLL.B. Therefore, besides the fact that these samples are from 20 different patients, we do not have any extra information here.

We focus next on how to turn raw data from these 20 microarrays into gene expression data, i.e., preprocessing.

Demonstration (Continued): The Features

It is helpful to review the structure of MLL.B object again

```
> MLL.B
AffyBatch object
size of arrays=712x712 features (13 kb)
cdf=HG-U133B (22645 affyids)
number of samples=20
number of genes=22645
annotation=hgu133b
notes=
```

One microarray has 712 by 712 probes; that is, there are 506944 probes in total. As the microarray pairs the probes to take PM (perfect match) and MM (mismatch) measures, at most 253472 (506944/2) probe pairs can be used to measure gene expressions. We can use probeNames(MLL.B) to get the genes measured by each probe pair. Here, not all probes on the microarray are used.

```
> length(probeNames(MLL.B))
[1] 249502
```

So 249502 out of the 253472 probe pairs are used in this study. Those probe pairs are used to measure only 22645 genes. This is because we need multiple probe pairs measuring the same gene to get reliable answer. The gene names can be found by featureNames(MLL.B). In this study, most of the genes are measured by 11 probe pairs, but some are measured by 20 probe pairs. You can check that the first 11 probe pairs probeNames(MLL.B)[1:11] all measure the same gene 200000_s_at, and the last 20 probe pairs probeNames(MLL.B)[249502+(-19:0)] all measure the same gene AFFX-TrpnX-M_at.

We need preprocessing to combine the multiple measurements on the same gene into one expression value. Before that we first extract the raw data on the probes.

The Raw Data: Probe Intensities

The raw expression intensities for all probes can be extracted by function exprs().

```
> dim(exprs(MLL.B))
[1] 506944 20
```

From the dimensions, we can see that this extracts the probe intensities for all 506944 probes for each of the 20 microarrays. Since not all probes are used to measure the genes of interest, we need pm() and mm() functions to extract the relevant PM (perfect match) and MM (mismatch) values.

```
> dim(pm(MLL.B))
[1] 249502     20
> dim(mm(MLL.B))
[1] 249502     20
```

We can see that pm() and mm() give us the probe intensity values of the 249502 used probe pairs for each of the 20 samples. If you want to see all values related to the gene 200000_s_at, do the following

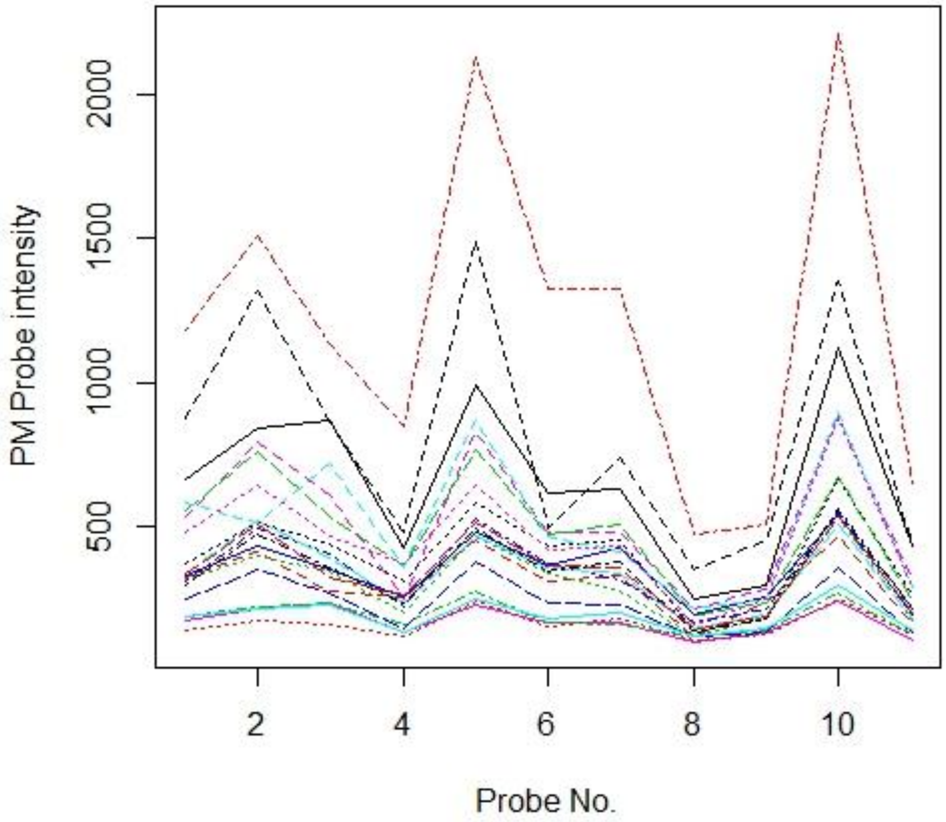
```
> pm(MLL.B,"200000_s_at")
```

Let us visualize the probe intensities raw data next.

Visualize the PM Values for One Gene

We plot the PM values for the first gene 200000_s_at by function matplot()

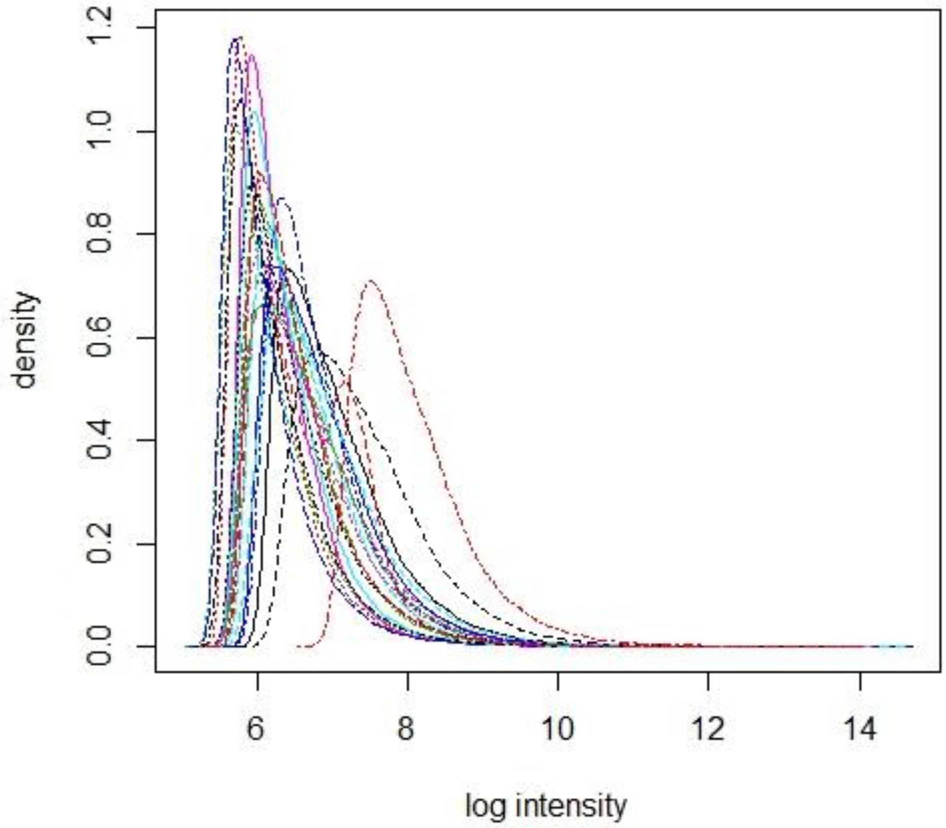
```
> matplot(pm(MLL.B,"200000_s_at"),type="l", xlab="Probe No.",
ylab="PM Probe intensity")
```



We see 20 curves, one for each sample (microarray). Each curve connects the PM density values at the 11 probes measuring the same gene 200000_s_at. We can see that there is substantial variation within the same microarray (curve) among the 11 values measuring the same quantity. The 11 values need to be **summarized** into one expression value for gene 200000_s_at. Also, we observe systematic difference among the curves, which could result from the biological differences across the samples (patients) that we are interested in. But this may also be caused by systematic measurement difference across the microarrays (samples). To remove the second effect from the analysis, we need to carry out a **background correction**.

Visualize all Probe Intensity Values

The hist(MLL.B) produces plots for the *log of the probe intensity* values. Notice here that hist() does not produce histograms, but rather density plots (which can be considered as the smoothed version of histograms).



)

We see 20 curves, one for each sample (microarray), for all logs of probe intensities within the same microarray. The density curves are quite skew to the right. There are also many variations across the curves. We expect many of the genes, and particularly the MM values, to express similarly in different patients. Therefore, the variations across the curves are likely due to systematic measurement difference across the microarrays. We remove the systematic bias (location difference among the curves) through **background correction**. Also, we remove the shape differences by **normalization**. After these two steps, all microarrays have the same probe intensity distribution (over all genes). So for the expression values processed this way, any remaining difference across microarrays for a particular gene should be due to its genetic effect, not due to systematic measurement difference.

Preprocessing Microarray Data

To convert the raw probe intensity data into gene expression data, we preprocess in three steps:

1. Background correction
2. Normalization
3. Summarization

Bioconductor provides various preprocessing methods. We will only outline a description of the available methods. The topic of optimal preprocessing is a field of intense research, so a definitive recommendation is not mandatory. Generally, we should use one of the standard implemented methods. In your future work, you should follow related literature. Or more likely, follow the instruction of your supervisor who should have expertise in this area.

Background correction

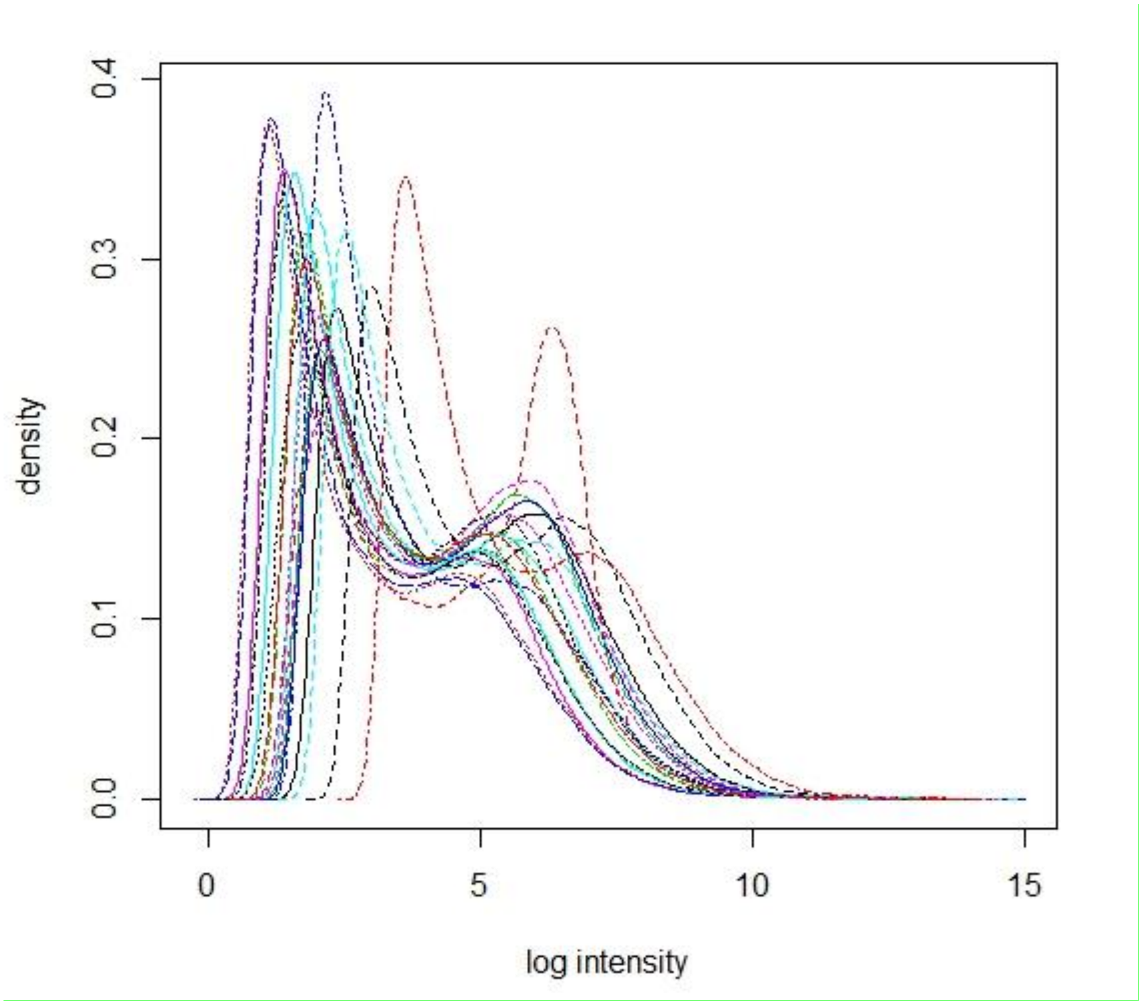
The available background correction methods can be seen in the following:

```
> bgcorrect.methods()
[1] "bg.correct" "mas"      "none"      "rma"
```

The mas background is part of the MAS Affymetrix software and is based on the 2% lowest probe values. RMA uses only the PM values, neglects the MM values totally, and is based on conditional expectation and the normality assumption of probe values.

We can see the effect of RMA background correction on the MLL.B data

```
> bgcData<-bg.correct(MLL.B,"rma")
> hist(bgcData)
```



Normalization

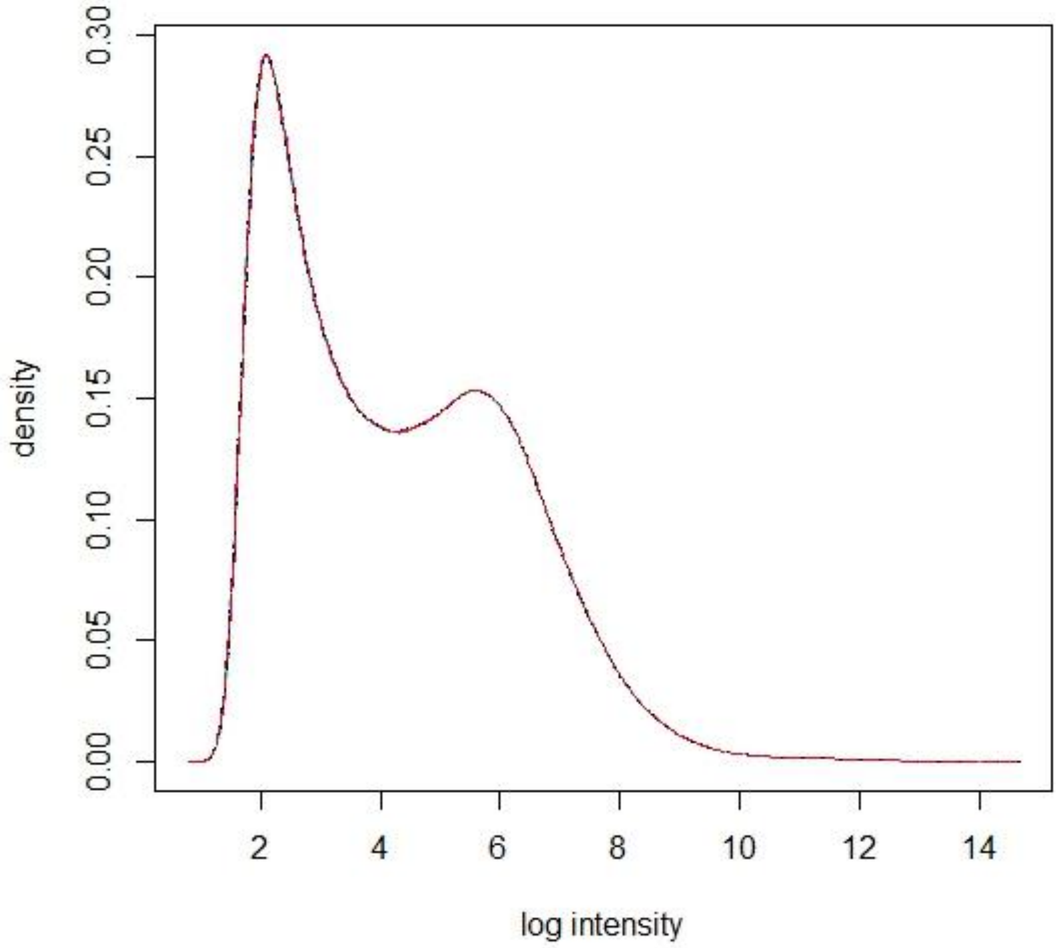
The following normalization methods are available:

```
> normalize.methods(MLL.B)
[1] "constant"      "contrasts"      "invariantset"    "loess"
[5] "methods"        "qspline"         "quantiles"       "quantiles.robust"
```

Constant is a scaling method equivalent to linear regression on a reference array although without intercept term. More general are the non-linear normalization methods such as loess, qspline, quantiles, and robust quantiles. For more detailed description, check the manual for built-in processing methods in affy package.

The normalization effect on MLL.B data can be observed with

```
> normData<-normalize(bgcData)
>hist(normData)
```



Summarization

Finally, we wish to combine the multiple probe intensities into one gene expression value. First, we generally wish to correct PM values by subtracting the corresponding MM values in the pair:

```
> pmcorrect.methods()
[1] "mas"      "methods"   "pmonly"    "subtractmm"
```

MAS subtract the MM value when possible (if MM value less than PM value) and subtract something else when not possible, by an algorithm designed by Affymetrix. PMONLY will make no adjustment to PM values. See more descriptions for built-in processing methods in affy package.

Then the combination methods are:

```
> express.summary.stat.methods()
[1] "avgdiff" "liwong" "mas" "medianpolish" "playerout"
```

The first one is just simple averaging. For descriptions of other methods, see built-in processing methods in affy package.

We now obtain the gene expression data for the first gene 200000_s_at in MLL.B data set with:

```
p<-probeset(normData,"200000_s_at")[[1]]
est<-express.summary.stat(p, pmcorrect="mas", summary="liwong")
> summary(est)
         Length Class  Mode
exprs    20     -none- numeric
se.exprs 20     -none- numeric
```

This gives the 200000_s_at gene expression values for each of the 20 samples, and the corresponding standard error for the gene expression values.

Preprocessing the Example Data Set

For preprocessing the whole MLL.B data set, we can combine the steps by:

```
eset <- expresso(MLL.B,bgcorrect.method="rma",
normalize.method="quantiles", pmcorrect.method="subtractmm",
summary.method="avgdiff")
```

This results in an ExpressionSet object “eset”

```
> summary(eset)

     Length        Class        Mode

           1 ExpressionSet         S4
```

The expression values can be extracted from the object by exprs(eset), which gives a 22645 by 20 matrix containing the expression values of 22645 genes of 20 samples. These are the same kind of expression data sets we used in previous modules.

An alternate method is to use the RMA method for preprocessing, which combines convolution background correction, quantile normalization, and summarization by the median polish algorithm.

```
eset1<- rma(MLL.B)
```

This gives us the ExpressionSet object “eset1”, and we can get the expression data set by exprs(eset1).

Our data analysis usually starts with the matrix containing #genes by #samples expression values, such as exprs(eset1). You should know how to get the ExpressionSet using given preprocessing methods from the raw data of AffyBatch, and to extract the expression values by exprs().

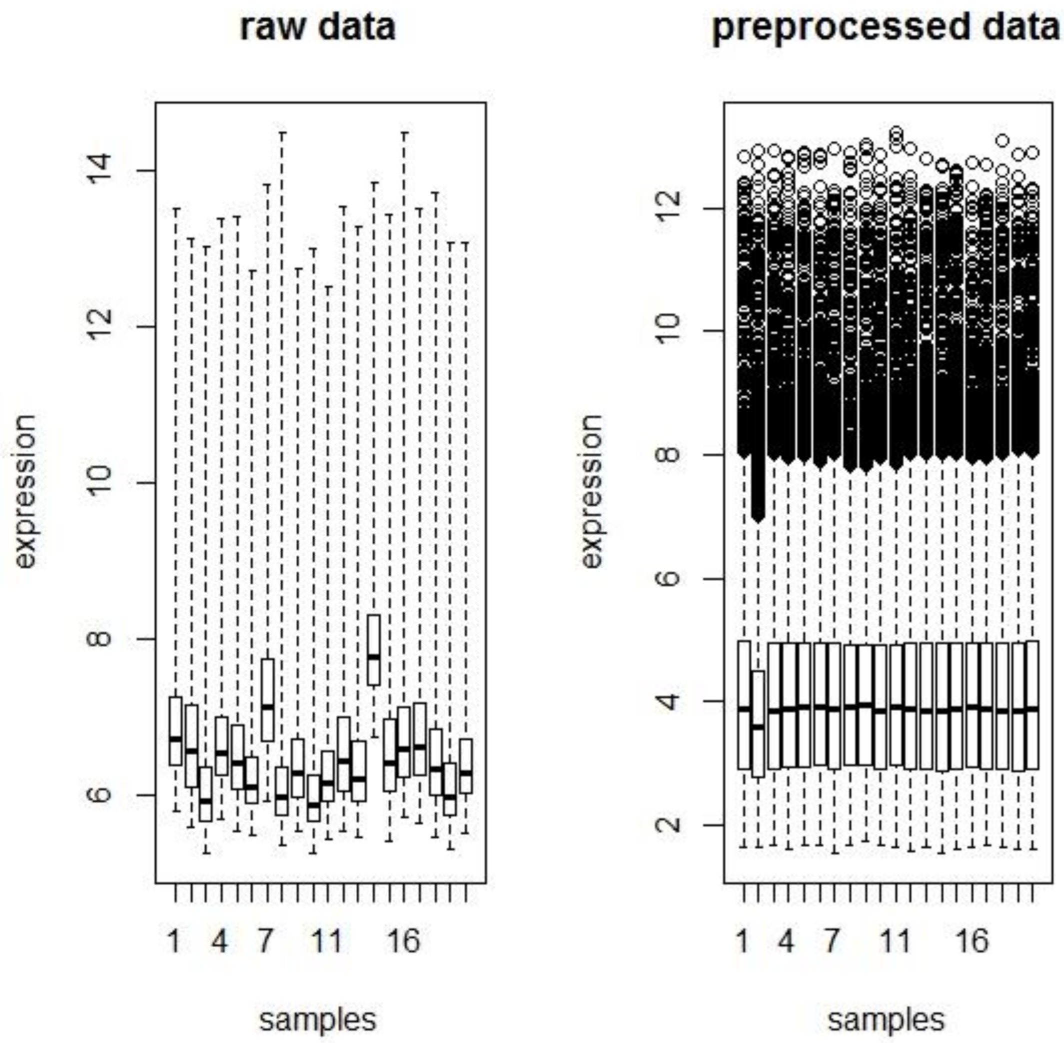
Visualize the Preprocessing Effect

We can use boxplots to compare the raw data and preprocessed data:

```
par(mfrow=c(1,2))  #Put two plots in one row (2 columns)

boxplot(MLL.B, col=MLL.B$samples, xlab="samples", ylab="expression", names=c(1:20), main="raw data")

boxplot(data.frame(exprs(eset1)), xlab="samples", ylab="expression", names=c(1:20), main="preprocessed data")
```



Further Preprocessing by Removing Patient Specific Means or Medians

After the aforementioned steps, it is often desirable to further preprocess the ExpressionSet data in order to remove patient specific means or medians. When the patient median is zero, for instance, testing for a gene to have mean expression value different from zero becomes meaningful.

We demonstrate this on the ExpressionSet eset1 we just got. We take out the first ten patients and further preprocess them: subtract the median expression of the patient and then divide by the MAD of the expressions of the patient.

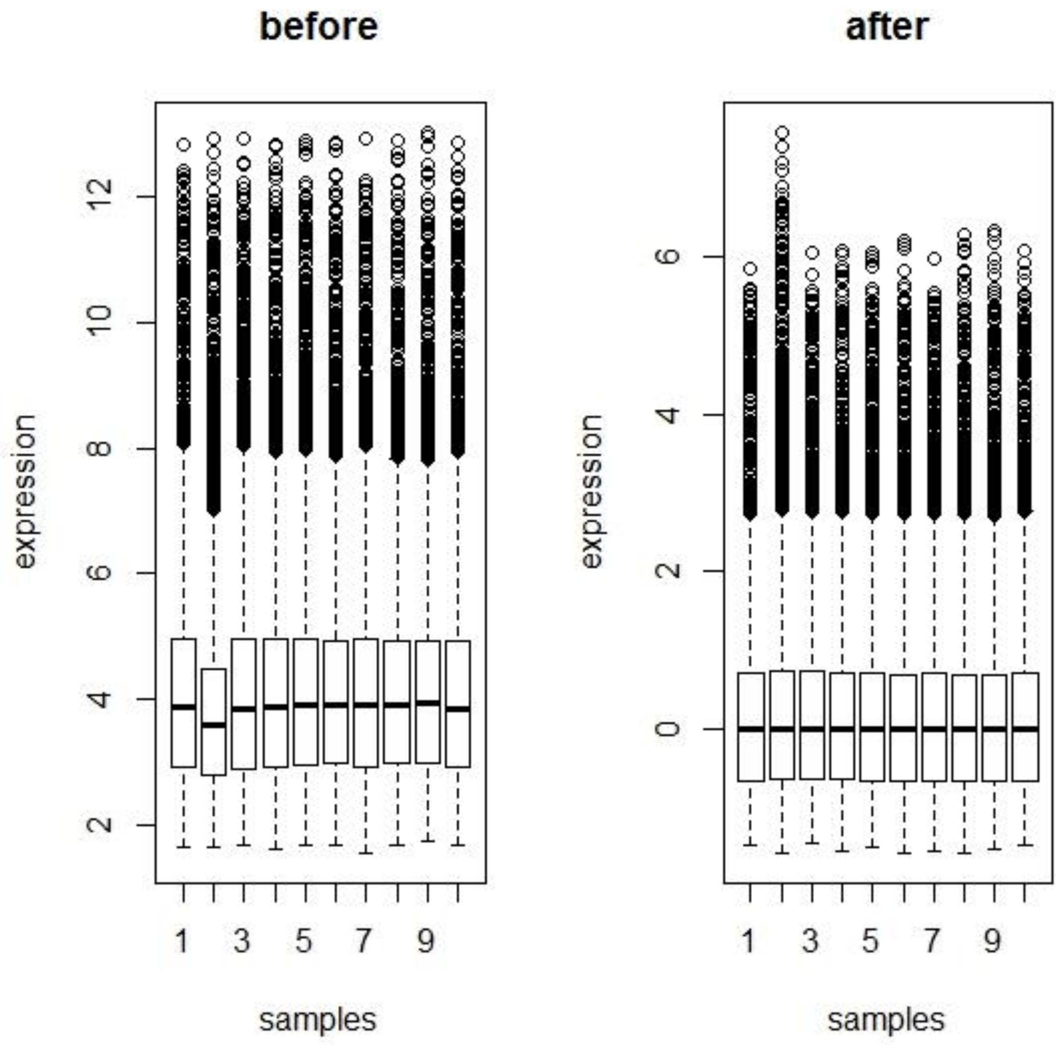
```
firstTen<-eset1[,eset1$sample<=10]  #Take first 10 samples
firstTen.copy<-firstTen    #Make a copy data set
mads <- apply(exprs(firstTen), 2, mad) #MAD for each of the ten samples
meds <- apply(exprs(firstTen), 2, median) #median for each of the ten samples
dat <- sweep(exprs(firstTen), 2, meds) #subtract the medians
exprs(firstTen.copy) <- sweep(dat, 2, mads, FUN="/") #divide MADs
```

Here, we take out the first ten patients to firstTen, and made a copy firstTen.copy. The median and MAD are calculated column-wise from firstTen, and removed from firstTen.copy by sweep() function. So firstTen contains the untreated data and firstTen.copy contains the further preprocessed data. Notice the default option in sweep() is subtraction, so division needs to be specified by FUN="/".

Visualize the Effect of Median Removal

We can use boxplots to visualize the effect of further preprocessing.

```
par(mfrow=c(1,2)) #Put two plots in one row (2 columns)
boxplot(data.frame(exprs(firstTen)),  xlab="samples", ylab="expression",
names=c(1:10),  main="before")
boxplot(data.frame(exprs(firstTen.copy)),  xlab="samples",
ylab="expression", names=c(1:10),  main="after")
```



Summary

In this lesson, we discussed the basic structure of the microarray data in R., and introduced preprocessing.

The information on different arrays is stored in phenoData. Generally, arrays correspond to different patients. They can also be related to issues from different organs of the same patient, or replicates of the same tissue sample.

You should know the three major steps of the preprocessing, and the purpose they each serve. In R, you should be able to apply existing preprogrammed methods to preprocess the raw probe intensities data into an ExpressionSet object. The gene expression values (which we usually do statistical analysis with) are then extracted by using exprs() function on the ExpressionSet object. You should also be able to replicate basic visualization of data using histograms (in Bioconductor packages they are actually the density plots) and boxplots.

In the next lesson, we will apply some simple statistical analysis to gene expression values.

Lesson 2: Simple Linear Regression

Lesson 2: Gene filtering and the application of linear models

Objectives

By the end of this lesson you will have had the opportunity to:

- **Filter** the genes in the data set
- Apply **linear model analysis** to genes

Overview

In this module, we will review how to filter genes in the data set and visualize the result. We will also discuss the application of linear models to analyze microarray data.

Gene Filtering

We can ‘filter’ genes by selecting genes that meet some criteria. We illustrate the process of gene filtering here.

It is wise to keep in mind that there are statistical as well as biological criteria for filtering genes and that a combination of these often gives the most satisfactory results.

We demonstrate the gene filtering on the data set “ALL” that comes with the package “ALL” from Bioconductor. This data set is an ExpressionSet object. So it has already been preprocessed.

Combining Several Filters

It is often desired to combine several filters. Of course it is possible to program filters completely on your own, however, we may conveniently use the function filterfun() to combine them into one filter. The following script combines six filters together into ff(), which we can apply to a data set.

```
library("genefilter")

        f1 <- function(x)(IQR(x)>0.5)

        f2 <- pOverA(.25, log2(100))

        f3 <- function(x) (median(2^x) > 300)

        f4 <- function(x) (shapiro.test(x)$p.value > 0.05)

        f5 <- function(x) (sd(x)/abs(mean(x))<0.1)

        f6 <- function(x) (sqrt(10)* abs(mean(x))/sd(x) > qt(0.975,9))

        ff <- filterfun(f1,f2,f3,f4,f5,f6)  #combine above 6 functions into one filter
```

The following are circumstances under which functions return TRUE:

- The first function returns TRUE if the interquartile range is larger than 0.5
- The second if 25% of the gene expression values is larger than log2(100)=6.643856
- The third if the median of the expression values taken as powers to the base two is larger than 300
- The fourth if it passes the Shapiro-Wilk normality test at 0.05 level
- The fifth if the coefficient of variation is smaller than 0.1
- The sixth if the one-sample *t*-value is significant at 0.05 level.

Then filterfun() combines the six filter functions into ff(). This combined filter can be applied to a data set by

```
        genefilter(DataSet, ff)  #apply the filter “ff” on dataset named “DataSet”
```

The genefilter() returns a logical vector indicating whether the gene passed all the six filters.

If you do not have the package “genefilter” on your machine yet, install it from Bioconductor. Use command biocLite("genefilter") (see page 2 of Applied Statistics for Bioinformatics using R).

Example : Combined Filter on the ALL Data Set

We apply the combined filter to the preprocessed ALL dataset from the ALL package. (You should have run the previous script so that ff() is defined in your session.)

```
> library("ALL"); data(ALL)

> selected <- genefilter(exprs(ALL[,ALL$BT=="B"]), ff) #use “ff” filter on “B” stage patients’ expression values

> sum(selected)

[1] 317
```

We can see that 317 genes pass all six filters. We can then study those 317 genes further.

Comments on the Combined Filter

In order to use these filter steps properly we should first think them through. Some filters may be focusing on very similar properties.

```
library("genefilter")

f1 <- function(x)(IQR(x)>0.5)

f2 <- pOverA(.25, log2(100))

f3 <- function(x) (median(2^x) > 300)

f4 <- function(x) (shapiro.test(x)$p.value > 0.05)

f5 <- function(x) (sd(x)/abs(mean(x))<0.1)

f6 <- function(x) (sqrt(10)* abs(mean(x))/sd(x) > qt(0.975,9))

ff <- filterfun(f1,f2,f3,f4,f5,f6) #combine above 6 functions into one filter
```

Since the IQR divided by 1.349 is a robust estimator of the standard deviation, the first filter in fact selects genes with a certain minimal standard deviation.

Notice that $2^x > 300$ is equivalent to $x > \log_2(300) = 8.228819$, so that the third filter is highly similar to the second filter but with a bigger cutoff value.

Furthermore, $s/|\bar{x}| < 0.1$ is equivalent to $\sqrt{10}|\bar{x}|/s > 1/\sqrt{10}$, so that the last two filters are highly similar but with different cutoff values.

There really is a lot redundancy in these six filters, and we could use only four of them to obtain the same results.

Example: T-test and Normality Filter on the ALL Data Set

Here we consider selecting genes that express differently for B-cell ALL versus T-cell ALL using a two-sample *t*-test. Since t-test requires the normality assumption, we may opt to select only those genes that also pass the Shapiro-Wilk normality test. The normality test will be applied separately for the B-cell ALL patients and for the T-cell ALL patients.

```
library("genefilter");library("ALL"); data(ALL)

patientB <- factor(ALL$BT %in% c("B","B1","B2","B3","B4")) #An indicator variable =1 for B-cell patients and =0 for T-cell patients

f1 <- function(x) (shapiro.test(x)$p.value > 0.05)  #pass normality test

f2 <- function(x) (t.test(x ~ patientB)$p.value < 0.05) #pass 2-sample t-test

sel1 <- genefilter(exprs(ALL[,patientB==TRUE]), filterfun(f1)) #B-cell patients pass normality tests

sel2 <- genefilter(exprs(ALL[,patientB==FALSE]), filterfun(f1)) #T-cell patients pass normality tests

sel3 <- genefilter(exprs(ALL), filterfun(f2)) #pass t-tests

selected <- sel1 & sel2 & sel3 #pass if passing all 3 filters
```

We create a logical factor patientB indicating patients with B-cell ALL (TRUE) and with T-cell ALL (FALSE). We then select the genes passing Shapiro-Wilk normality test (p-value>0.05) in B-cell ALL and T-cell ALL, and store results in sel1 and sel2. The genes passing the Welch two-sample *t*-test (p-value<0.05) is stored in sel3. Then we keep those genes passing all three tests in selected. The expression values for the filtered genes can be found by

```
ALLs <- ALL[selected,] #Data for genes passing the 3 filters
```

This results in1817 genes which pass the three filters.

Example (Continued): Visualize Gene Selection

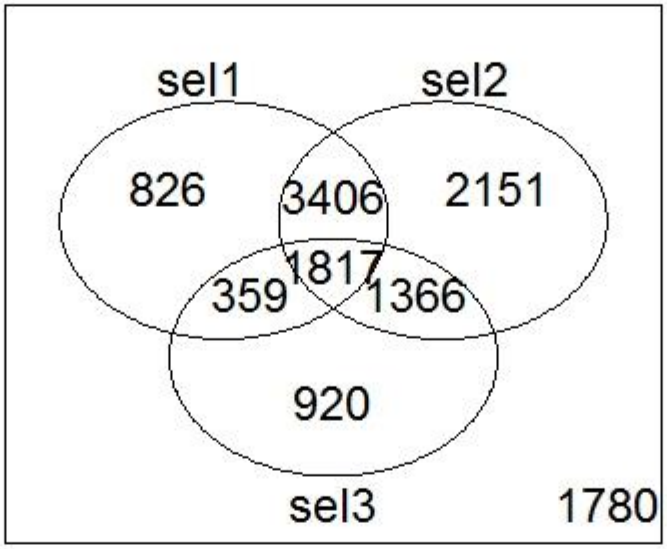
A common approach to visualize how genes are divided among filters is through construction of a Venn diagram. This can conveniently be done using functions from the “limma” package (install it from Bioconductor if not already installed).

```
library(limma)

x <- apply(cbind(sel1,sel2,sel3), 2, as.integer) #Combine 3 filter results (as 3 columns) and convert entries (TRUE/FALE) to integers (0 and 1)

vc <- vennCounts(x, include="both") #Calculate counts for Venn Diagram

vennDiagram(vc) #Draw Venn diagram
```



From the resulting Venn diagram it can be seen that 1817 genes pass all three filters, 1780 genes pass none, 3406 genes pass the normality tests but not the *t*-test filter, etc.

Applications of Linear Models

The limma package is frequently used for analyzing microarray data through linear models, such as ANOVA.

Example: Here, we follow Example 1 on page 101 of the Applied Statistics for Bioinformatics using R. We illustrate the application of ANOVA on the ALL data set. We will use only three groups for the purposes of illustration: patients with B-cell leukemia in early stages B, B1 and B2.

There are 60 patients total in the three stages. We can extract their gene expressions in a 12625 by 60 matrix. For each of the 12625 genes, that is a sample of 60 observations in three groups. We can apply an ANOVA analysis to each gene as in an earlier module.

Here, we follow Example 1 on page 101 of the Applied Statistics for Bioinformatics using R. That is, we fit the ANOVA through linear model by explicitly defining its design matrix (review the lecture notes in ANOVA module if you need to review what a design matrix is). The fits are in fact **not** those in the usual ANOVA models; we will explain that in more detail on the next page.

It is important to note that instead of 12625 separate ANOVAs, the linear model fits are adjusted through an empirical Bayes procedure. The main idea is that the within group noise variance across 60 samples should be similar for different genes as they are on the same microarrays. To use this information, the empirical Bayes procedure adjusts the linear model fits to adapt the gene specific variances with a global variance estimator (Smyth, 2004). We will not review any details of the empirical Bayes procedure. It is preprogrammed in R and we will just use it directly to adjust the linear model fits. The following R script does the work for us.

```
library("limma"); library("ALL"); data(ALL, package = "ALL");

allB <- ALL[,which(ALL$BT %in% c("B","B1","B2"))] #Patients in 3 stages

design.ma <- model.matrix(~ 0 + factor(allB$BT)) #Design matrix for ANOVA without intercept

colnames(design.ma) <- c("B","B1","B2") #variable names

fit <- lmFit(allB, design.ma) #fit linear model with the design matrix

fit <- eBayes(fit) #apply Empirical Bayes adjustment.
```

Example: ANOVA on B-cell ALL Focusing on B1 Group

The 12625 ANOVA results are saved in the “fit” object. Since the design matrix has zero intercept, the ANOVA coefficients are simply group means (adjusted by empirical Bayes method). They are stored in the fit$coef, a 12625 by 3 matrix. As in textbook, we can print out the top five genes selected by p-values adjusted by false discovery rate (FDR). This is done using topTable().

```
> print( topTable(fit, coef=2, number=5, adjust.method="fdr"), digits=4) #Print the top 5 selected for 2nd coefficient with FDR adjustment, show only 4 digits.
```

```
                  logFC AveExpr     t       P.Value adj.P.Val     B
AFFX-hum_alu_at 13.42   13.50       326.0 3.165e-99 3.996e-95 207.5
32466_at        12.68   12.70       306.3 1.333e-97 8.413e-94 204.8
32748_at        12.08   12.11       296.3 9.771e-97 3.616e-93 203.4
35278_at        12.44   12.45       295.5 1.146e-96 3.616e-93 203.3
34593_g_at      12.64   12.58       278.0 4.431e-95 1.119e-91 200.6
```

Let us denote the true means for groups B, B1 and B2 as μ_0, μ_1 and μ_2. By coef=2, we are looking at the second coefficient μ_1. That is, we are testing if the second group (“B1”) mean expression is zero $H_0 : \mu_1 = 0$. Since we are testing this null hypothesis for multiple genes (12625 of them) in the data set, the false discovery rate adjustment is needed. Recall that the “adjusted p-value” here is in fact the q-value covered in Module 6.

We can see that the first gene AFFX-hum_alu_at has mean expression of 13.42 in the B1 stage patients, and overall mean of 13.5 in all patients in this three groups. The extremely small adjusted p-value means that we reject the null hypothesis, and the AFFX-hum_alu_at gene is expressed in B1 group (nonzero mean). This does not imply that AFFX-hum_alu_at gene express in B1 group differently from in the other two groups (B and B2), which we will test using contrasts later.

We can also focus on finding the expressed genes in any of the three groups, instead of only B1 group. This is done by removing the coef=2 from above R command.

Example: ANOVA on B-cell ALL Focusing on Three Groups

Here we look at another print out using the same function without coef=2 option:

```
> print( topTable(fit, number=5,adjust.method="fdr"), digits=4) #Print the top 5 selected for overall model with FDR adjustment, show only 4 digits.

                     B    B1    B2    AveExpr      F   P.Value  adj.P.Val
AFFX-hum_alu_at 13.54 13.42 13.54   13.50 113249 1.370e-112 1.729e-108
32466_at           12.65 12.68 12.72   12.70  99047 7.641e-111 4.823e-107
32748_at           12.05 12.08 12.14   12.11  92909 5.212e-110 2.101e-106
35278_at           12.40 12.44 12.47   12.45  92156 6.655e-110 2.101e-106
34593_g_at         12.49 12.64 12.56   12.58  80660 3.630e-108 9.166e-105
```

Here we use the F-test to testing the null hypothesis $H_0 : \mu_0 = \mu_1 = \mu_2 = 0$. This finds genes that expresses in at least one group. Notice that this is not the usual ANOVA null hypothesis $H_0 : \mu_0 = \mu_1 = \mu_2$.

Caution: The textbook is stating that the linear model fit is the usual ANOVA testing $H_0 : \mu_0 = \mu_1 = \mu_2$ (similarly expressed in three groups). It is not.

To do the usual ANOVA to find genes that express differently among the three groups, we need to use contrasts. When we test two null hypothesis $\mu_0 - \mu_1 = 0$ and $\mu_1 - \mu_2 = 0$, together they become the ANOVA null hypothesis $H_0 : \mu_0 = \mu_1 = \mu_2$.

Next, we test two contrasts $\mu_0 - \mu_1$ and $\mu_1 - \mu_2$.

Example: (Continued) Contrasts on B-cell ALL

We consider two contrasts here: mean difference between the first two groups $\mu_0 - \mu_1$ and mean difference between the last two groups $\mu_1 - \mu_2$. These two can be put in a contrast matrix.

```
> cont.ma <- makeContrasts(B-B1,B1-B2, levels=factor(allB$BT)) #design
matrix built from the two contrasts

> cont.ma

Contrasts

Levels B - B1 B1 - B2

B        1        0

B1      -1        1

B2       0       -1
```

Implement this, and we can find the genes differ most between B1 and B2 groups. Testing null hypothesis $H_0 : \mu_1 = \mu_2$

```
> cont.ma <- makeContrasts(B-B1,B1-B2, levels=factor(allB$BT)) #design matrix
built from the two contrasts

> fit1 <- contrasts.fit(fit, cont.ma) #fit linear model with the design matrix

> fit1 <- eBayes(fit1) #apply Empirical Bayes adjustment.

> print( topTable(fit1, coef=2, number=5,adjust.method="fdr"), digits=4) #Print
the top 5 selected for 2nd coefficient with FDR adjustment, show only 4 digits.

                 logFC  AveExpr    t      P.Value       adj.P.Val     B

33358_at        1.4890   5.260    7.374  5.737e-10   7.242e-06 12.272

1389_at        -1.7852    9.262   -7.081 1.816e-09   9.744e-06 11.220

1914_at         2.0976    4.939    7.019  2.315e-09   9.744e-06 10.997

36873_at        1.8646   4.303     6.426 2.361e-08   7.452e-05  8.871

37471_at        0.8701   6.551    6.106 8.161e-08   2.061e-04  7.733
```

We can see that this ranks the genes with largest mean difference between groups in the top, comparing to the ones with largest mean values (such as AFFX-hum_alu_at) in the earlier table without contrasts.

Example: (Continued) Contrasts on B-cell ALL for ANOVA

When we consider both contrasts $\mu_0 - \mu_1$ and $\mu_1 - \mu_2$ together, then the null hypothesis becomes the usual ANOVA null hypothesis $H_0 : \mu_0 = \mu_1 = \mu_2$. We can see the top 5 genes selected as

```
> print( topTable(fit1, number=5,adjust.method="fdr"), digits=4) #Print the top 5
selected for overall model with FDR adjustment, show only 4 digits.

            B...B1 B1...B2 AveExpr   F   P.Value     adj.P.Val

33358_at    -1.502   1.489   5.260    28.44 2.042e-09 2.578e-05

1914_at      -2.244   2.098   4.939    26.20 6.615e-09 4.176e-05

1389_at       0.974  -1.785   9.262    25.18 1.146e-08 4.824e-05

32977_at     2.022  -1.241   8.750    22.70 4.542e-08 1.433e-04

36873_at    -1.930   1.865   4.303    21.75 7.878e-08 1.989e-04
```

Here we no longer rank in top the AFFX-hum_alu_at gene which has large but similar means in all three groups. Rather we are ranking in top the genes with most significant differences among group means. Notice that the top second 1914_at gene actually has larger absolute differences among group means than the top 33358_at gene. This is because the variance of the gene expression is also taken into account in statistical test. The 33358_at gene is statistically more significantly different (so it must have smaller variance than 1914_at gene).

It is critically important to know which hypothesis you are testing with those R commands. The textbook is rather unclear on this point, and has mistakes in exercise solutions. So carefully review this example.

Next page we further show a problem of not using contrasts on this data set.

Example: (Continued) B-cell ALL: How Many Genes Are Expressed?

Recall the first linear model fit without contrasts was testing the null hypothesis $H_0 : \mu_0 = \mu_1 = \mu_2 = 0$. This seems to be testing for genes that expresses in at least one groups. So how many expressed genes do this test find? Let us use a false discovery rate of 0.05. Then

```
> dim( topTable(fit, number=Inf, p.value=0.05,adjust.method="fdr"))
```

```
[1] 12625     7
```

All of the 12625 genes are expressed!!!

Is this true? When we look at the data set more carefully,

```
> min(exprs(allB)) #minimum expression value of allB data
```

```
[1] 1.984919
```

All the expression values are positive!! So this data set has **not** been preprocessed to center the samples.

Generally, even after centering the samples, we still have too many genes rejecting the null hypothesis $H_0 : \mu_0 = \mu_1 = \mu_2 = 0$. These tests just found expression differences among genes which occur naturally without relating to the disease (pheno data) of interest. Hence the differentially expressed genes really need to be found using contrasts in this linear model.

Not centering each sample does not affect the contrasts testing among groups. So the testing for the difference among groups with contrasts is still valid. We find 787 genes express differently among the three groups B, B1, B2.

```
> dim( topTable(fit1, number=Inf, p.value=0.05, adjust.method="fdr"))
```

```
[1] 787   7
```

In many experiments, there are some microarrays in the “control” group. Then the differentially expressed genes are found by testing contrasts with the control group.

Example: (Continued) ANOVAs on B-cell ALL

We can also run the usual ANOVAs on each gene.

```
> allB.exp<-exprs(allB) #get expression values

> panova <- apply(allB.exp, 1, function(x) anova(lm(x ~
factor(allB$BT)))$Pr[1]) #apply ANOVA on each row (gene), save p-value

> p.fdr <-p.adjust(p=panova, method="fdr") #FDR adjusted p-values

> ord<-order(p.fdr , decreasing=F)[1:5] #5 largest q-values

> as.matrix(p.fdr[ord]) #Show top 5 q-values

                [,1]

33358_at     6.32e-05

1914_at      1.03e-04

1389_at      1.14e-04

32977_at      3.08e-04

36873_at     3.81e-04
```

We can see that the usual ANOVAs select same top genes as the contrasts fit above. Those genes are not the same top genes selected through the linear model fit. This is understandable, as the usual ANOVA selects genes with large differences in group means, rather than genes with large group mean values.

However, this code is much more time-consuming than the earlier one, and it does not allow the use of the empirical Bayes procedure to correct gene specific variance from other genes' information. Notice the p-values shown on this page (using the usual ANOVAs on each gene) are different from the p-values from the fit with contrasts. That one adjusts the gene specific variance using the empirical Bayes procedure.

In practice, we want to use the limma package to do the linear model fit with contrasts.

Lesson Summary

This lesson covered gene filtering. You should now be able to program gene filters, and to combine several filters together using “genefilter” package.

We also reviewed how to apply the linear model fit in the limma package. You should now be able to explain the hypotheses tested by the linear fit model, and use the model to filter genes. In addition, you should be able to discuss the similarities and differences between the linear model and ANOVA.

In the next lesson, we will discuss using annotation to relate your selected genes to their biological meanings.

Lesson 3: Annotation and the GO numbers

Objectives

By the end of this lesson you will have had the opportunity to:

- Search an **annotation package** to obtain information pertaining to a gene
- Use annotation to **search literature**
- Search **GO numbers**, parents and children
- Use GO numbers to **filter genes**

Overview

In this module, our analysis relates to obtaining biological information pertaining to genes. We will search an annotation package to obtain such biological information. Next, we will search the gene in literature. After that, we will review what is known as GO numbers, and use them to filter genes that are related to a specific biological term.

Annotation

We have learned how to get the expression data set from the raw data, and perform some analysis to select some genes. However, finding a gene “200000_s_at” is really no help to any researcher, unless we understand what that gene is. The raw data as an AffyBatch object also contains annotation information tells us what the genes are. The annotation can be extracted from the MLL.B object as

```
> annotation(MLL.B)
[1] "hgu133b"
```

This tells us the information about the gene identifiers, as such as “200000_s_at”, is contained in the annotation environment “hgu133b”. We next learn how to use the corresponding annotation package to obtain this information. For “hgu133b” annotation, the info is contained in the annotation package “hgu133b.db”.

Annotation Package

We demonstrate the use of annotation packages on the ALL data set. First we find its annotation.

```
> library("ALL"); data(ALL)

> annotation(ALL)

[1] "hgu95av2"
```

Hence, the info is contained in the corresponding annotation package "hgu95av2.db". Install it from Bioconductor using biocLite("hgu95av2.db") if not previously installed. You can use library(help=hgu95av2.db) to find out what the package contains.

The package contains many contains many environments (hash tables), each mapping the gene manufacturer identifiers such as "1000_at" to various types of information about the genes. For example, we see "hgu95av2CHR" maps manufacturer IDs to chromosomes. You can use commands such as ?hgu95av2CHR to get detailed help on this environment. An easy manner to make the content of an environment available is by converting it into a list and to print part of it to the screen.

```
> library(hgu95av2)

> ChrNrOfProbe <- as.list(hgu95av2CHR)

> ChrNrOfProbe[1:2]

$`1000_at`

[1] "16"

$`1001_at`

[1] "1"
```

We see that the genes 1000_at and 1001_at are on the 16th and 1st chromosome respectively.

Searching Annotation Package

We can use get() function to retrieve needed information from the environments. We demonstrate this for the gene with the manufacturer's identifier "1389_at". Below we obtain the Entrez Gene identifier, the gene abbreviation, gene name, and the UniGene identifier, respectively.

```
> get("1389_at", env = hgu95av2ENTREZID)
[1] 4311
> get("1389_at", env = hgu95av2SYMBOL)
[1] "MME"
> get("1389_at", env = hgu95av2GENENAME)
[1] "membrane metallo-endopeptidase (neutral endopeptidase,
enkephalinase, CALLA, CD10)"
> get("1389_at", env = hgu95av2UNIGENE)
[1] "Hs.307734"
```

Frequently we are interested in their chromosome location, and, specifically, starting position(s).

```
> get("1389_at", env = hgu95av2CHRLOC)
        3         3         3         3
154797436 154797705 154797953 154798079
```

Its cytoband location can also be obtained.

```
> get("1389_at", env = hgu95av2MAP)
[1] "3q25.2"
```

Hence, we see that the gene is on Chromosome 3 at q arm band 25 subband 2.

GeneBank Information

We can get the GenBank accession number by

```
> get("1389_at", env = hgu95av2ACCNUM)
[1] "J03779"
```

Let us use the GenBank accession number to search its nucleotide data base.

```
> library(annotate)
> genbank("J03779",disp="browser", type="accession")
```

From this we obtain the corresponding Gene ID number (4311), which can also be used to obtain a complete XML document.

```
> genbank(4311, disp="data", type="uid")
```

Using annotation to search literature

Given the manufactures probe identifier we can also search literature by collecting Pubmed ID's and to use these to collect relevant articles.

```
library(hgu95av2.db);library(annotate); library(ALL); data(ALL)

pmid <- get("1389_at",env=hgu95av2PMID)

pubmed(pmid,disp="browser")
```

Another possibility is to collect a list containing PubMed ID, authors, abstract text, title, journal, and publication date.

```
absts <- pm.getabst("1389_at", "hgu95av2")

pm.titles(absts)
```

The list can obviously be searched for regular expressions.

```
ne <- pm.abstGrep("neutral endopeptidase",absts[[1]])
```

You can then construct an HTML table with the titles.

```
pmAbst2HTML(absts[[1]][ne],filename="pmon1389_at.html")
```

You can see the file pmon1389_at.html in your working directory. The file contains papers related to neutral endopeptidase.

Searching GO Numbers and Evidence

By the phrase "ontology" we mean a structured language about some conceptual domain. The gene ontology consortium defines three ontologies:

- A Molecular Function (MF) describes a phenomenon at the biochemical level such as "enzyme", "transporter", or "ligand"
- A Biological Process (BP) may coordinate various related molecular functions such as "DNA replication" or "signal transduction"
- A Cellular Component (CC) is a unit within a part of the cell such as "chromosome", "nucleus", or "ribosome"

Each term is identified by a unique GO number. To find GO numbers and their dependencies we extract a list from the annotation files hgu95av2GO for example. From the latter we extract a list and use an apply type of function to extract another list containing GO identification numbers.

```
> go1389 <- get("1389_at", env = hgu95av2GO)

> idl <- lapply(go1389,function(x) x$GOID)

> idl[[1]]

[1] "GO:0006518"
```

The list idl contains 30 members of which only the first is printed to the screen. By changing GOID into Ontology more specific information pertaining to ontology is extracted. From the annotate package we may now select the GO numbers which are related to a biological process.

Demonstration (Cont'd): Searching Evidence for 1389_at

We now select the gene 1389_at GO numbers which are related to a biological process. This yields 12 of them.

```
> library(annotate)

> getOntology(go1389,"BP")

[1] "GO:0006518" "GO:0019233" "GO:0050435" "GO:0002003"
"GO:0044267"

 [6] "GO:0006508" "GO:0071345" "GO:0071492" "GO:0071493"
"GO:0046449"

[11] "GO:0001822" "GO:0090399"
```

There are various types of evidence such as that inferred from sequence or structural similarity (ISS), traceable author statement (TAS), inferred from electronic annotation (IEA), etc. See the documentation at http://geneontology.org/page/guide-go-evidence-codes. Per GO identifier the type of evidence can be obtained. (We only display the first five to save space here.)

```
> getEvidence(go1389)[1:5]

GO:0006518 GO:0019233 GO:0050435 GO:0002003 GO:0044267

    "ISS"      "ISS"      "ISS"      "TAS"      "TAS"
```

When we want to select the GO numbers with evidence of a traceable author statement, we can use the subset function to create a list.

```
go1389TAS <- subset(go1389, getEvidence(go1389)=="TAS")
```

A manner to extract information from this list is by using an apply type of function.

```
sapply(go1389TAS,function(x) x$GOID)

sapply(go1389TAS,function(x) x$Evidence)

sapply(go1389TAS,function(x) x$Ontology)
```

Searching GO Parents and Children

The term "transmembrane receptor protein-tyrosine kinase" is more specific and therefore a 'child' of the more general term parent term "transmembrane receptor".

We can obtain parents and children from a GO identifier. We will need the package GO.db (install it from Bioconductor).

```
>  library("GO.db")

> GOMFPARENTS$"GO:0003700"

      is_a

"GO:0001071"

> GOMFCHILDREN$"GO:0003700"

      is_a         is_a         is_a        is_a          is_a

"GO:0000981" "GO:0001010" "GO:0001011" "GO:0001034"
"GO:0001073"

      is_a         is_a         is_a        is_a          is_a

"GO:0001130" "GO:0001142" "GO:0001167" "GO:0001199"
"GO:0098531"
```

In case of a list of GO identifiers you may want to collect the ontology, parents, and children identifiers in a vector.

```
go1389 <- get("1389_at", env = hgu95av2GO) #GOID for 1389_at probe
gonr <- getOntology(go1389, "BP") #biological processes GO numbers
gP <- getGOParents(gonr) #parents for GO numbers
gC <- getGOChildren(gonr) #children for GO numbers
gPC <- c(gonr,gP,gC)        #combine the 3 lists
pa <- sapply(gP,function(x) x$Parents) #extract GO numbers for parents
ch <- sapply(gC,function(x) x$Children) #extract GO numbers for children
gonrc <- unique(c(gonr, unlist(pa), unlist(ch))) # GO numbers of (genes, parents, children)
```

You can inspect the resulting vector gonrc. Note that we applied unique() function to this list to remove redundant genes. In this particular example, there is no redundancy, so that the result is same whether applying unique() or not. However, it is possible for different genes to have the same parent genes, creating redundancy in the list. (On the other hand, different genes should have different children genes. You may think about why.) So you should remember to do it to remove redundant genes from the list.

Gene Filtering by a Biological Term

An application of working with GO numbers is to filter for genes which are related to a biological term. From a biological point of view it is most interesting to select genes which are related to a certain biological process to be specified by a term such as "transcriptional repression". We combine this with the previous filter. First we define a function (Gentleman, et al., 2005, p.123) to collect appropriate GO numbers from the environment GOTERM.

```
library("GO.db"); library("annotate"); library("hgu95av2.db")

GOTerm2Tag <- function(term) {

    GTL <- eapply(GOTERM, function(x) {grep(term, x@Term,
value=TRUE)}) #grep GOTERM on each element in list

    Gl <- sapply(GTL, length) #count how many are found

   names(GTL[Gl>0]) #return those nonempty lists (length>0)

}

> GOTerm2Tag("transcriptional repressor")

[1]  "GO:0017053"
```

In the above, we use function eapply() search an environment like GOTERM by grep() for matches of a specified term. A precaution is taken to select only those names which are not empty. This is done by sapply the length() function to find how many matches we get in each name.

Combine Statistical Filtering and Filtering by a Biological Term

We now combine the filtering by a biological term above with a previous statistical filtering in Lesson 2. We first reproduce the data set ALLs in the earlier filtering through t-tests and normality tests.

```
library("genefilter"); library("ALL"); data(ALL)
patientB <- factor(ALL$BT %in% c("B","B1","B2","B3","B4"))
f1 <- function(x) (shapiro.test(x)$p.value > 0.05)
f2 <- function(x) (t.test(x ~ patientB)$p.value < 0.05)
sel1 <- genefilter(exprs(ALL[,patientB==TRUE]), filterfun(f1))
sel2 <- genefilter(exprs(ALL[,patientB==FALSE]), filterfun(f1))
sel3 <- genefilter(exprs(ALL), filterfun(f2))
selected <- sel1 & sel2 & sel3
ALLs <- ALL[selected,]
```

Then we search the GO term "GO:0017053" which can be translated to probe of the ALLs data.

```
tran <- hgu95av2GO2ALLPROBES$"GO:0017053" #probes for this GO number
inboth <- tran %in% row.names(exprs(ALLs)) #ALLs probes for this GO number
ALLtran <- ALLs[tran[inboth],]  #pick out probes data for this GO number
```

The GO translated probe names are intersected with the row names of the data, giving the logical variable inboth. The variable tran[inboth] gives the ids by which genes can be selected. Next, gene ids for which inboth equals TRUE are selected and the corresponding data are collected in the data frame ALLtran. More information can be obtained by GOTERM$"GO:0017053". By dim(exprs(ALLtran)) it can be observed that 9 genes which passed the normality and t-test filter are related to "transcriptional repression".

Significance per Chromosome Example

After a statistical analysis designed to filter and order genes, it is often quite useful to do post analysis on the results. In particular, after collecting *p*-values from a *t*-test, one may wonder whether genes with significant *p*-values occur more often within a certain chromosome. To test for such over or under representation the Fisher test is very useful.

We illustrate this on the ALL data set. On the expression values of the ALL data we perform a two sample *t*-test using the patient groups (a) for which remission was achieved and (b) for which it was not achieved. Per chromosome, it can be tested whether the odds ratio differs from 1 or, equivalently, whether there is independence. The data for the test consist of the number of significant probes on Chromosome 19, the number of non-significant probes on Chromosome 19, the number of remaining significant probes, and the number of remaining non-significant probes.

```
> library("ALL"); data(ALL); library("hgu95av2.db")

> rawp <- apply(exprs(ALL), 1, function(x) t.test(x ~
ALL$remission)$p.value) #p-values for t-test on each gene

> xx <- as.list(hgu95av2CHR) #Chromosome

> AffimIDChr <- names(xx[xx=="19"]) #those on 19th chromosome

> names(rawp) <- featureNames(ALL) #gene names

> f <- matrix(NA,2,2) #a matrix to save 2 by 2 table

> f[1,1] <- sum(rawp[AffimIDChr]<0.05); f[1,2] <-
sum(rawp[AffimIDChr]>0.05) #counts of reject/accept by t-test on 19th
chromosome

> f[2,1] <- sum(rawp<0.05) - f[1,1] ; f[2,2] <- sum(rawp>0.05) - f[1,2]
#counts of reject/accept by t-test on other chromosomes

> print(f)

     [,1]  [,2]

[1,]  106   603

[2,]  832 11084
```

Significance per Chromosome Example (Cont’d)

We now apply the Fisher’s test on the 2 by 2 table of significant genes on Chromosome 19.

```
> fisher.test(f)  #apply Fisher’s exact test on 2 by 2 table ‘f’

        Fisher's Exact Test for Count Data


data:  f

p-value = 1.651e-12

alternative hypothesis: true odds ratio is not equal to 1

95 percent confidence interval:

 1.864352 2.919902

sample estimates:

odds ratio

  2.341648
```

The number of significant probes is larger for Chromosome 19 resulting in an odds ratio of 2.34. The hypothesis of independence to Chromosome is rejected.

Lesson Summary

This lesson covered how to find related biological information for genes. We first reviewed annotation packages and how to search them for information. Next, we reviewed how a literature search is conducted.

We also reviewed GO numbers, and used them to filter genes which are related to a specific biological term.

Finally, we learned how to combine the biological filtering with statistical filtering.

Module Summary

This module covered analysis of microarray data set. You should now be able to preprocess probe data, and to use statistical analysis including the linear model to filter genes. You should also be able to explain annotations, and to combine the biological information of the genes with statistical analysis.

In the next module, we will study clustering methods in grouping data.