Chapter 1

Bioconductor Tools for Microarray Analysis

SIMON COCKELL

Bioinformatics Support Unit

Newcastle University

Newcastle upon Tyne, UK

Email: simon.cockell@newcastle.ac.uk

MATTHEW BASHTON

Bioinformatics Support Unit

Newcastle University

Newcastle upon Tyne, UK

Email: matthew.bashton@newcastle.ac.uk

COLIN S. GILLESPIE

School of Mathematics & Statistics
Newcastle University
Newcastle upon Tyne, UK
Email: colin.gillespie@newcastle.ac.uk

1.1 Introduction

1.1.1 What is Bioconductor?

Bioconductor [1] is an open source software project for the R statistical computing language. The main aim of the project is to provide R packages to facilitate the analysis of DNA microarray, sequencing, SNP and other genomic data. In addition to various packages for analysis of data, they also distribute meta-data packages that provide useful annotation. The stable non-development version of Bioconductor is normally released twice annually[1]. In addition to providing packages the Bioconductor site (bioconductor.org) also provides documentation for each package, often a brief vignette and a more comprehensive user guide.

1.1.2 Illumina microarrays

The Illumina BeadArray^{TM} microarray platform makes use of BeadChips, which, as their name suggests are composed of tiny beads. These beads are made of silica and are just 3μm in diameter, and are covered in hundreds of thousands of 79 nucleotide long oligonucleotide sequences; individual beads each having a different probe sequence. A unique property of BeadChips, which distinguish them from Affymetrix arrays, is that every array is unique, the beads self assemble into microwells found on the slide. As a consequence of this, most probes will be present on the array $\tilde{3}0$ times [2]. An additional consequence is that each array has to be decoded during its manufacture so that the positions of each probe are known; the first 29 nucleotides of the probe sequence are reserved for decoding the array during manufacture. An Illumina BeadChip can support the analysis of multiple samples having up to 12 copies of each array on a chip. The HumanHT-12 v4.0 Expression BeadChip has over 47,000 probes which are based on RefSeq 38 and Unigene[2]. Being randomly assembled means that Illumina arrays dont suffer so much from spatially localized artifacts [3] additionally the presence of multiple copies of each bead can be used as technical replicate information for Variance Stabilizing Transformation (VST) [4].

1.1.3 lumi

The `lumi` package [5], available in Bioconductor[3], is specifically designed to process BeadArrray data providing VST and quality control steps specifically tuned for this technology. `lumi` assumes that data has been pre-background corrected in Illumina's BeadStudio or GenomeStudio, although if it has not they provide the `lumiB` function which mimics the background correction found therein. Should you want to investigate processing the raw image files from the scanner directly, in order to have control over the

[1] http://www.bioconductor.org/news/bioc_2_11_release/

[2] http://www.illumina.com/Documents/products/datasheets/datasheet_gene_exp_analysis.pdf

[3] http://www.bioconductor.org/packages/release/bioc/html/lumi.html

background correction and image processing steps then the Bioconductor package `beadarray` needs to be used. This however, depends on the BeadScan Illumina microarray scaner software being set to output raw TIFF files, which is not a default setting, at the time the arrays are processed. Consequently we are focusing on analysis downstream of BeadStudio or GenomeStudio. Two key features of `lumi` which give it an advantage over other methods, which are largely based on Affymetrix data processing methods are: VST which takes advantage of multiple beads per probe when transforming the data and outperforms log2 based transformations [4] and Robust Spline Normalisation (RSN) which is designed for Illumina data and combines the advantageous features of both quantile normalisation (fast, gene rank order preserving) and loess normalization which is continuous[4].

1.1.4 Limma

Whilst `lumi` handles the transformation, normalisation and quality control of Illumina microarray data, in order to obtain a list of differentially expressed genes you need to make use of the `limma` package. This has advantages over t-tests since it makes use of linear models and empirical Bayes methods in order to determine significant differentially expressed genes [6][7]. The starting point for analysis in `limma` is a matrix of expression values from `lumi` this is accessed via the `exprs` method provided by the core `Biobase` package of Bioconductor.

1.2 Loading Microarray Data into R

1.2.1 Installing Bioconductor packages

Provided that R is already installed, than installing packages from the Bioconductor repository is straightforward. First we source the installation script:

```
source("http://bioconductor.org/biocLite.R")
```

and then run the downloaded function, `biocLite`, to install the standard packages:

```
biocLite()
```

This will install the `Biobase`, `IRanges` and `AnnotationDbi` packages (and their dependencies). Additional packages can be installed by directly specifying their name:

[4] http://www.bioconductor.org/packages/2.11/bioc/vignettes/lumi/inst/doc/lumi.pdf

Package	Version	Package	Version	Package	Version
ArrayExpress	1.18.0	arrayQualityMetrics	3.14.0	Biobase	2.18.0
GEOquery	2.24.1	GOstats	2.24.0	gplots	2.11.0
limma	3.14.4	lumi	2.10.0	lumiHumanAll.db	1.18.0
lumiHumanIDMapping	1.10.0	RamiGO	1.4.0		

Table 1.1: Packages (and versions) used in this chapter.

```
## From Bioconductor
biocLite(c("ArrayExpress", "arrayQualityMetrics", "GOstats", "GEOquery",
  "lumi", "lumiHumanAll.db", "lumiHumanIDMapping", "RamiGO"))
## From CRAN
install.packages("gplots")
```

Table 1.1 gives an overview of the packages used in this chapter. A text file of the commands used can be found at

https://github.com/csgillespie/illuminia-analysis/

1.2.2 Loading and normalising the data

The data for this chapter can be downloaded (within R) directly from the Array Express website. We simply load the `ArrayExpress` and `lumi` packages

```
library("ArrayExpress")
library("lumi")
```

and use the `getAE` function to download the data file. Since the data is Illumina BeadArray, the standard `ArrayExpress` method will make some false assumptions, so we use `getAE` instead

```
ae = getAE("E-MTAB-1593", type = "full")
raw_data = lumiR(ae)
```

The `raw_data` object contains all the information associated with this microarray experiment. For example, the raw data, information on the protocol and details of the `MIAME` metadata. It is an S4 R `LumiBatch` object. This object is specifically used to contain and describe Illumina data within R. It extends `ExpressionSet`, one of the key Bioconductor classes. To investigate the `LumiBatch` object (or any R object), we can use the `str` command to obtain a detailed overview of its structure:

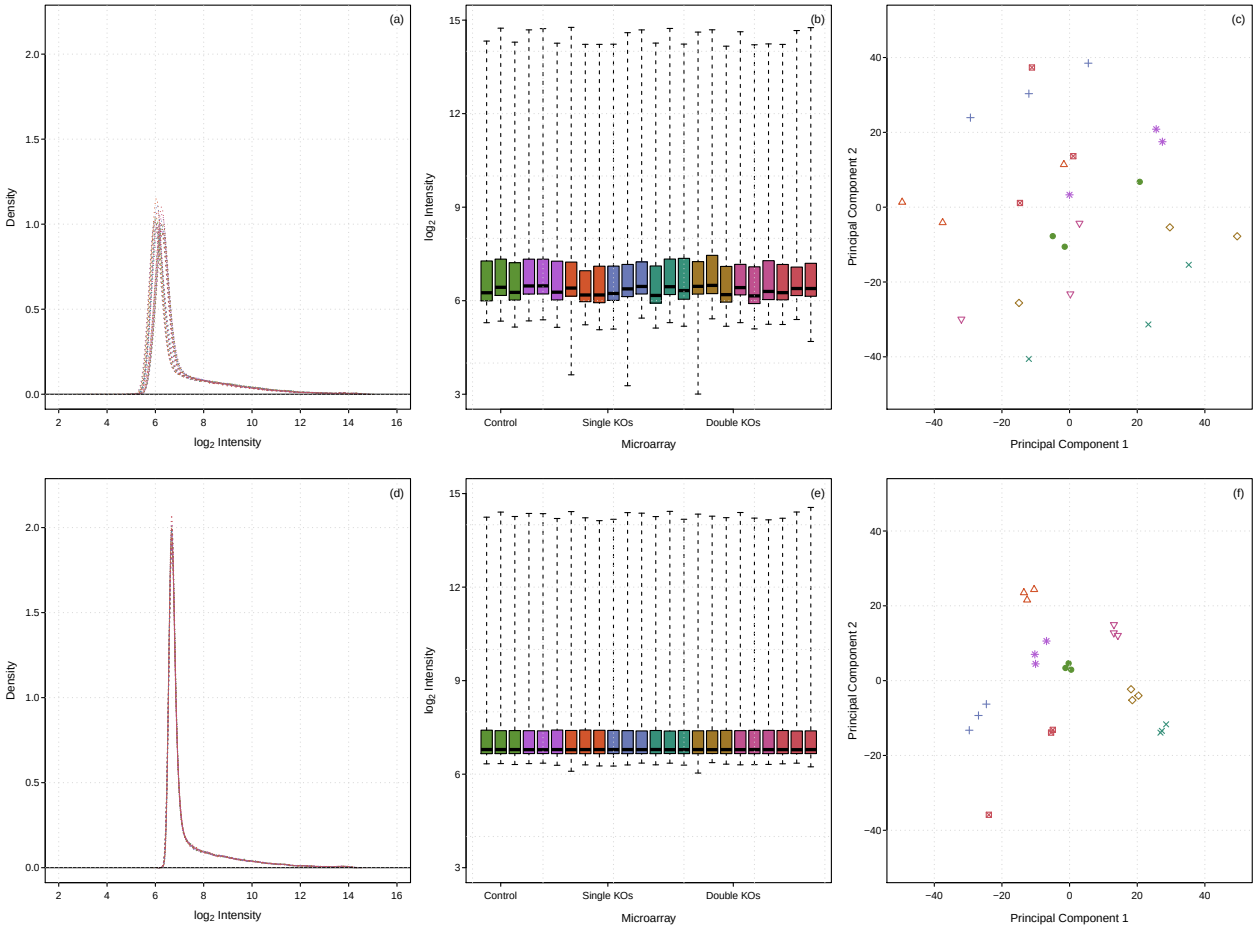

Figure 1.1: Top row: raw data. Bottom row: normalised. (a) Density plots. (b) Boxplots (c) PCA plots.

```
str(raw_data)
```

Alternatively, we can get a short summary using `print(raw_data)`.

It is also desirable to use the `plot()` method to investigate some of the properties of the individual arrays. This quality control measure allows any obviously aberrant arrays to be detected and removed. There are a range of possible plots, including density plots of intensity (line graphs or boxplots), plots of pairwise correlation and a plot of the coefficient of variance.

```
## Substitute "density" for one of the other plot types:
## "boxplot", "pair", "MA" or "cv"
plot(raw_data, what="density")
```

Plots of the raw and normalised data are given in figure 1.1. As the density plots show, the intensity across the microarrays in our experiment can be quite varied (figure 1.1a). We can make a valid assumption here that this variance is experimental, and not biological. The result of this large amount of experimental variability is that it masks the biological variance in which we are interested. We therefore have to treat the data

obtained from the arrays in such a way that this systematic variability is masked, while the important and interesting biological variability is maintained. Normalisation is intended to achieve this purpose by ensuring all the samples in an experiment follow the same underlying statistical distribution, but the variances within observations of a particular probe should remain, and therefore be discoverable.

There is no *single* method for normalising microarray data sets. In this chapter, we will use one of the standard methods. Normalisation is carried out over two steps. First, a transformation is applied to stabilise the variance across probes[4]:

```
vst_data = lumiT(raw_data)
```

The variance stabilization transformation exploits the within-array technical replicates (i.e. the bead-level data) generated from Illumina microarrays to model the relationship between the mean and the variance. Using VST means the differential expression of more genes can be detected at the same time as reducing false positives[4]. Other (less sophisticated) methods are a $\log_2$ and cubic root transformation.

The second step is to normalise between chips

```
## Use robust spline normalisation (rsn)
rsn_data = lumiN(vst_data, method = "rsn")
```

This normalisation step uses the `rsn` method and forces the intensity values for different samples (microarrays) to have the same distribution[5].

When the raw data is loaded, the `lumiQ` function is automatically called to proved a list of summary data about the arrays that stored in the QC slot of the `LumiBatch` object. Following normalisation this function needs to be called again to update the data in the QC slot to reflect changes in the underlying data, viz.

```
qc_data = lumiQ(rsn_data)
```

Information held in the QC slot includes the mean, standard deviation, detectable probe ratios and sample correlation data, in addition to control probe information for each array. The subsequent S4 plot methods used to render QC plots are also dependent on the data in the QC slot of the `LumiBatch` object.

The raw and transformed data can be easily compared via plotting:

```
plot(qc_data)
plot(raw_data)
```

1.2.3 Quality Control

The `arrayQualityMetrics` package is a generic set of quality control routines that can be applied to many types of array data. It produces a report of quality metrics, which can be used to make assessments about overall array quality, and diagnose batch effects. To generate the report, we load the package

```
library(arrayQualityMetrics)
```

then run the `arrayQualityMetrics` function

```
arrayQualityMetrics(expressionset=qc_data, outdir="qc")
```

You can view the output from the `arrayQualityMetrics` function at

`https://github.com/csgillespie/illuminia-analysis/`

Examining the *distance between arrays* metric and the principal component plot from the report (the latter figure is given in figure 1.1), strongly suggests that array twenty-three (`TF3TF4B`) is an outlier - so we will remove this array from any further analysis. We have to do this removal in the raw data, since the removal of any whole sample will effect our between-array normalisation, so this has to be repeated with the subset of twenty-three arrays that passed quality control:

```
raw_data_post = raw_data[, -23]
vst_data_post = lumiT(raw_data_post)
rsn_data_post = lumiN(vst_data_post, method = "rsn")
qc_data_post = lumiQ(rsn_data_post)
```

The *array intensity distributions* (not shown) metric suggests that array eleven may also be an outlier, although in this case it's variation from the other arrays is less convincing, so we will retain the array in the analysis.

Now that we can be confident the arrays we have retained are of sufficient quality, we want to ensure the probes we are analysing on those arrays also pass a stringent quality check. We begin by extracting the data matrix from the `LumiBatch` object:

```
exprs_data = exprs(qc_data_post)
treatments = c("Ctrl", "TF1", "TF2", "TF3", "TF4", "TF1TF4",
                                                  "TF2TF4", "TF3TF4")
array_names = rep(treatments, each=3)[1:23]
colnames(exprs_data) = array_names
```

Then we use the `detectionCall` to find the probes that are below a detection threshold.

```
present_count = detectionCall(raw_data_post)
select_data = exprs_data[present_count > 0, ]
```

This method exploits the detection p-value found in the raw data to determine whether or not a probe is detected above a threshold level in each of the samples of our experiment. If the detection p-value is less than 0.01 (by default, this can be changed by passing the `Th=` parameter to `detectionCall`), then the probe is found to be detected. The filter that is applied at this stage removes a probe from all the samples if it is not detected on any of the twenty-tree arrays. A probe that is detected on at least one array is retained.

Overall, this procedure has removed approximately 50% of probes:

```
nrow(select_data)/nrow(exprs_data)
## [1] 0.4709
```

1.3 Differentially expressed genes

1.3.1 Array annotation

Oligonucleotide identifiers provided by array manufacturers are technology specific, proprietary and mutable. It was felt that an external identifier would be beneficial to solve the problem of identifier permanency, and enable cross-platform data integration. nuIDs are one implementation of just such a strategy, each nuID is a string of letters and numbers which encode, via a variation of Base64, a lossless compression of the probes sequence in addition to a checksum [8]. The probe names on our Illumina arrays can be mapped to nuIDs, and these can then be used to stably map the probes to other identifiers (in this case, gene symbols and gene names).

This annotation is useful for much of the downstream analysis, providing human readable identifiers for genes that can be included in the output of the differential gene detection to come.

```
library(lumiHumanAll.db)
library(annotate)
probe_list = rownames(select_data)
nuIDs = probeID2nuID(probe_list)[, "nuID"]
symbol = getSYMBOL(nuIDs, "lumiHumanAll.db")
name = unlist(lookUp(nuIDs, "lumiHumanAll.db", "GENENAME"))
```

To avoid mixing-up order of the IDs and symbols, we combine everything in an R data frame

```
anno_df = data.frame(ID = nuIDs, probe_list, symbol, name)
```

1.3.2 Using limma to detect differentially expressed genes

The `limma` package has become the de-facto way of analysing microarrays for differentially expressed genes. By using an empirical Bayes method to moderate the variance of observations across the microarrays, `limma` tends to give more robust gene-lists than those generated by standard Student's t-tests. We set our samples up by defining the design of our experiment. The in-built R method `model.matrix` can be used to do this, the resulting design matrix tells `limma` which samples belong in which groups. The `lmFit` method then fits a linear model for each gene in the group of arrays:

```
library(limma)
design = model.matrix(~0 + factor(array_names, levels = treatments))
colnames(design) = treatments
num_parameters = ncol(design)
fit = lmFit(select_data, design)
```

Now we can set up the constrasts, or comparisons, that we want to analyse. In the case of this experiment we have eight sample groups: a control, four individual siRNA knock-downs and three combination knock-downs. We want to analyse the effect of each transcription factor knock-down separately and also detect the presence of any interaction effects. To detect main effect differences, we would have a contrast of the form

`TF1 - Ctrl`

To investigate interactions, the contrast would be slightly more complicated

`(TF1TF4-Ctrl) - (TF1 - Ctrl) - (TF4 - Ctrl) = TF1TF4 - TF1 - TF4 + Ctrl`

Using the `makeContrasts` method provided by `limma`, we set up the following contrast matrix:

```
cont_mat = makeContrasts(TF1-Ctrl, TF2-Ctrl, TF3-Ctrl, TF4-Ctrl,
                  TF1TF4-TF1-TF4+Ctrl, TF2TF4-TF2-TF4+Ctrl,
                  TF3TF4-TF3-TF4+Ctrl, levels=treatments)
fit2  = contrasts.fit(fit, contrasts=cont_mat)
```

We then fit these contrasts to the `limma` model of the data, and apply the empirical Bayes statistics to derive moderated t-statistics of differential expression for all the probes on the arrays:

Rank	Gene Symbol	Mean Expression	$\log_2$ Fold Change	Adjusted p-value
1	NDRG1	9.51	-1.64	5.5×10^{-16}
2	SCARNA11	7.49	1.73	1.1×10^{-14}
3	ZDHHC7	10.53	-1.21	1.1×10^{-13}
4	PAICS	11.10	-1.25	2.3×10^{-13}
5	C1orf85	10.55	-1.59	2.5×10^{-13}

Table 1.2: Top five differentially expression genes for the contrast `TF1 - Ctrl`.

```
fit2 = eBayes(fit2)
fit2$genes = anno_df
```

To enable other functions, notably `topTable`, to access the gene names we append the `anno_df` data frame to the `fit2` list.

The final stage of the `limma` analysis is to apply `topTable` in order to produce a list of differentially expressed genes. `topTable` takes a number of arguments which allow for precise filtering of the gene list by a number of criteria, including applying a cut-off to the $\log_2$ fold change and the p-value of the moderated t-tests performed. `topTable` also allows us to apply a multiple testing correction to the test statistics produced. This is important because utilising "raw" p-values in over 20,000 independent statistical tests will lead to an unreasonably high error rate when predicting differentially expressed genes (by definition, a rate of 1 in 20, or around 1,000 false positives per contrast for our arrays). Applying a multiple testing correction allows this error rate to be controlled for, and a lower false positive rate to be guaranteed.

```
## Filter by fold change (1.5x) and p-value (0.05) cutoffs
## Adjusted using Benjimini-Hochberg False Discovery Rate
topTable(fit2, coef="TF1 - Ctrl", p.value=0.05, lfc=log2(1.5))
```

1.4 Visually investigating differences

When dealing with so many data points, it can be useful to get a visual overview to quickly identify particularly interesting genes. In this section, we consider three standard graphical methods.

1.4.1 Volcano plots

When looking for interesting genes it can be helpful to restrict attention to differentially expressed genes that are both statistically significant and of potential biological interest. This objective can be achieved by

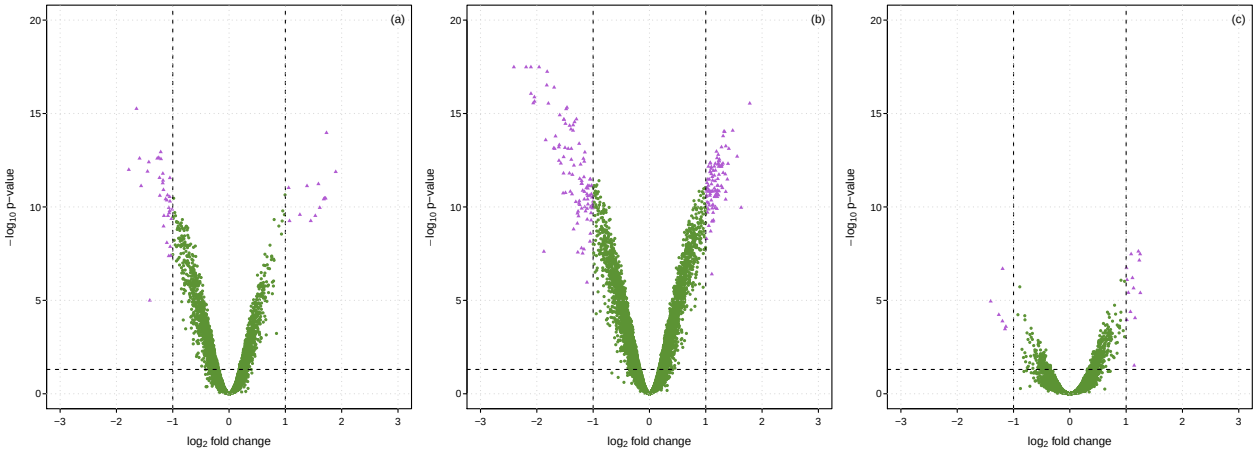

Figure 1.2: Volcano plots. The purple triangles indicate genes whose $\log_2$ absolute fold change is greater than one and has an adjust p-value greater than 0.05. The figures above are volcano plots for the contrasts (a) `TF1 - Ctrl` (b) `TF4 - Ctrl` (c) `TF1TF4 - TF1 - TF4 + Ctrl`.

considering only significant genes which show, say, at least a two-fold change in their expression level. A volcano plot can be used to summarise the fold change and p-value information produced by `limma`, the log fold change is plotted on the x-axis and the negative $\log_{10} p$-value is plotted on the y-axis.

To construct the plot, extract all genes, with associated p-value and fold change for a particular contrast,

```
gene_list = topTable(fit2, coef = "TF1 - Ctrl", number = nrow(anno_df))
```

The volcano plot is created using the standard plotting function

```
plot(gene_list$logFC, -log10(gene_list$adj.P.Val),
     col=1+(abs(gene_list$logFC) > 1 & gene_list$adj.P.Val < 0.05))
```

In this plot, we use the `col` argument to colour points where the adjusted p-value was less than 0.05 and the absolute log fold change was above one. Figure 1.2 shows volcano plots for three particular contrasts.

1.4.2 Venn Diagrams

Typically when comparing interesting genes over different contrasts, we often are interested in the gene overlap. The function `classifyTestsF` from the `limma` package classifies each gene as being up, down or not significant. In this example, we choose a p-value cut-off of 0.01

```
## P-values are adjusted for multiple testing
results = classifyTestsP(fit2, p.value = 0.01, method = "fdr")
```

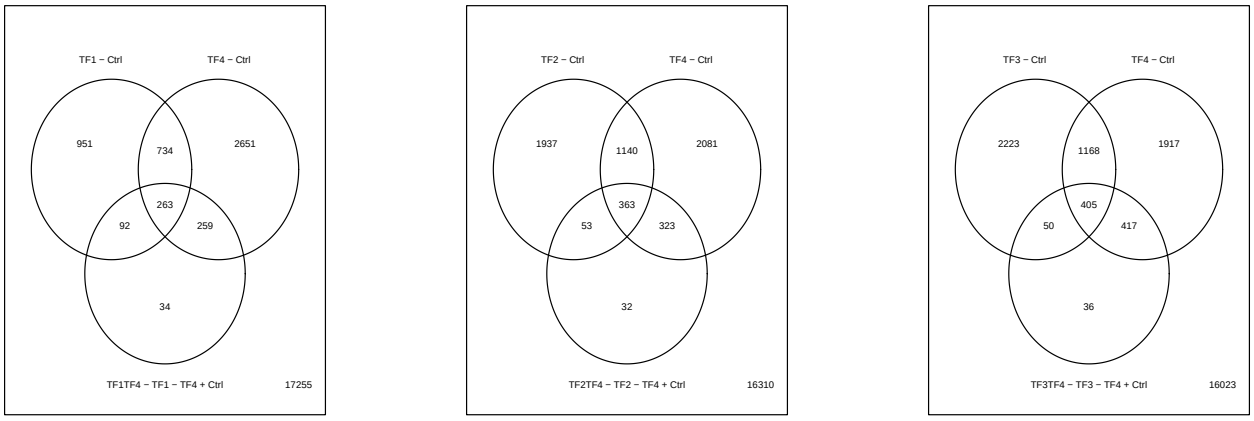

Figure 1.3: Venn diagrams showing the number over overlapping genes between contrasts.

When classifying the tests, we control for multiple testing using the FDR correction[9]. The results can then be visualised using a Venn diagram

```
vennDiagram(results[, c("TF1 - Ctrl", "TF4 - Ctrl",
                        "TF1TF4 - TF1 - TF4 + Ctrl")])
```

This commands generates figure 1.3a.

1.4.3 Heatmaps

Lists of genes that change expression are of limited value taken on their own. What is needed to provide more value to these lists is context. We can look at how the genes that change in one of our contrasts is affected in all the others by constructing a heatmap of the fold changes found in each of the contrasts. This visual tools allows us to quickly identify genes that are similarly regulated across conditions, or those that have opposing regulation. Due to the probe-level clustering that is applied during heatmap construction, we can also spot groups of genes that are regulated in similar ways across the constrasts. Genes that share regulation in this way are often functionally related.

In order to construct this plot, we first need the full results of the four contrasts we are plotting.

```
tf1_table = topTable(fit2, coef="TF1 - Ctrl", n=length(probe_list),
                     sort.by="logFC")
tf2_table = topTable(fit2, coef="TF2 - Ctrl", n=length(probe_list),
                     sort.by="logFC")
tf3_table = topTable(fit2, coef="TF3 - Ctrl", n=length(probe_list),
                     sort.by="logFC")
tf4_table = topTable(fit2, coef="TF4 - Ctrl", n=length(probe_list),
```

```
                        sort.by="logFC")
```

Now we construct a list of the top ten genes from each contrast (sorted on fold change). From each gene table we take the top ten genes, then select the unique probes, since the same probe may occur in more than one list

```
all_signames = unique(c(rownames(tf1_table[1:10,]),
                        rownames(tf2_table[1:10,]),
                        rownames(tf3_table[1:10,]),
                        rownames(tf4_table[1:10,])))
```

Now that we have a list of genes, we extract these genes from each data frame

```
full = data.frame(tf1_table[all_signames,]$logFC,
                  tf2_table[all_signames,]$logFC,
                  tf3_table[all_signames,]$logFC,
                  tf4_table[all_signames,]$logFC,
                  row.names=all_signames)
colnames(full) = paste0("TF", 1:4)
```

Another couple of manipulations are necessary, to provide information necessary for the rendering of the heatmap. We need to know the smallest and largest fold change values, so the extent of the range of values represented can be passed to the heatmap function, and we can keep zero (i.e. no change) in the centre of the colour spectrum. We also filter out unannotated genes at this stage, since they are of limited interest.

```
row_names = as.character(tf1_table[rownames(full), ]$symbol)
## Filter out unannotated genes
hm_data = full[!is.na(row_names), ]
row_names = row_names[!is.na(row_names)]
```

Finally, the heatmap itself is plotted (figure 1.4). The `heatmap.2` function from the `gplots` package is responsible for a number of data manipulations while drawing the heatmap. It rearranges the data rows (i.e. the probes) according to the results of a clustering step (using a Euclidean distance matrix, by default), and it colours the cells of the heatmap according to the fold change value for that probe in the contrast that the column represents. The `breaks` argument defines the scope of the colour gradient, ensuring the the most ”blue” cell is the one with the biggest negative fold change, that the most ”red” cell is the one with the largest positive fold change, and that white (the centre of the colour gradient) is set at zero.

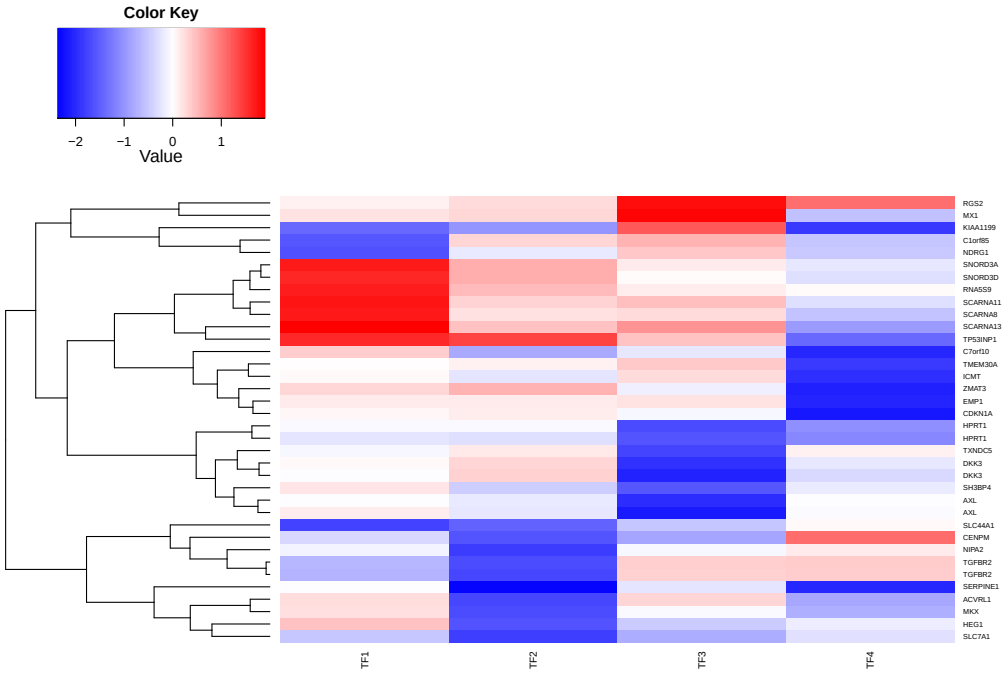

Figure 1.4: Clustering of the top ten differentially expressed genes from each main contrast. Red and blue correspond to up- and down-regulation respectively.

```
library("gplots")
breaks = c(seq(min(full), 0, length.out=128),
           seq(0, max(full), length.out=128))
heatmap.2(as.matrix(hm_data), dendrogram='row', Colv=FALSE,
              col=bluered(255), key=TRUE, labRow=row_names,
              breaks=breaks, symkey=FALSE, density.info="none",
              trace="none", cexRow=0.5, cexCol=0.75)
```

The resulting heat map is shown in figure 1.4. It is worth noting that many heat maps are constructed using a red-green colour scheme. However, since 5%-10% of the male population are red-green colour blind, this is a particularly poor choice of colours.

1.5 GOstats

The functional relationship of the genes in our lists can be studied more directly by using the `GOstats` package[10] to find Gene Ontology[11] terms that are statistically over-represented among the genes in the lists. To begin, we load the package

```
library(GOstats)
```

and then select the ”top genes”

```
sig_values = tf1_table[tf1_table$adj.P.Val < 0.05 &
                         abs(tf1_table$logFC) > log2(1.5), ]
sig_probes = as.character(sig_values$probe_list)
```

Then we map our list significant probes to Entrez gene identifiers (as these are the identifiers GOstats uses to find GO terms)

```
## Map probe id to Entrez gene identifiers
entrez = unique(unlist(lookUp(nuIDs[sig_probes],
                              "lumiHumanAll.db", "ENTREZID")))
entrez = as.character(entrez[!is.na(entrez)])
```

We also need to define a ”gene universe”, that is, the global set of terms among which we want to look for over-representation. If a particular GO term is not present in the genes represented on our microarray, then there is no point in testing to see if it is over-represented. In this case, we construct our universe from the probes on the arrays that passed our probe-level quality control step 1.2.3.

```
## Determine the universe of possible entrez ids
entrez_universe = unique(unlist(
                    lookUp(nuIDs, "lumiHumanAll.db", "ENTREZID")))
entrez_universe = as.character(entrez_universe[!is.na(entrez_universe)])
```

To test whether a particular probe is over-represented, we use a hyper-geometric test; this is provided by the `GOstats` package. We create an S4 object that contains all the necessary information

```
params = new("GOHyperGParams",
 geneIds=entrez,
 universeGeneIds=entrez_universe,
 annotation="lumiHumanAll.db",
 ontology="BP",
 pvalueCutoff= 0.01,
 conditional=FALSE,
 testDirection="over")
```

GO Identifier	Description	Occurrences (Universe size)	p-value
GO:0022403	Cell cycle phase	62(684)	3.1×10^{-19}
GO:0022402	Cell cycle process	67(857)	7.3×10^{-18}
GO:0000278	Mitotic cell cycle	57(641)	2.2×10^{-17}
GO:0007049	Cell cycle	73(1133)	2.8×10^{-15}
GO:0051301	Cell division	38(380)	1.8×10^{-12}

Table 1.3: Top five gene ontology identifiers with a brief description.

The `params` object is then passed to the `hyperGTest` function

```
hyperg_result = hyperGTest(params)
```

which returns a `GOHyperGResult` object. Printing this object, provides a summary of the results

```
print(hyperg_result)
## Gene to GO BP  test for over-representation
## 2766 GO BP ids tested (190 have p < 0.01)
## Selected gene set size: 214
##     Gene universe size: 9594
##     Annotation package: lumiHumanAll
```

Since we have performed multiple tests, we use the adjusted p-values

```
## Adjust p-values for multiple test (FDR)
go_fdr = p.adjust(pvalues(hyperg_result), method="fdr")
```

We can then select the significant GO terms

```
## Select the Go terms with adjusted p-value less than 0.01
sig_go_id = names(go_fdr[go_fdr < 0.01])
## Retrieve significant GO terms for BP (Molecular Function)
sig_go_term = getGOTerm(sig_go_id)[["BP"]]
```

The top five gene ontology identifiers are given in table 1.3.

Once we have a list of over-represented terms, we can use the `RamiGO` package to visualise them. `RamiGO` takes a list of terms and constructs a graph from them, and their parent terms.

```
# Visualize the enriched GO categories
library(RamiGO)
amigo_tree = getAmigoTree(sig_go_id)
```

The resulting graph can be examined for clusters of related enriched terms, which may be more important for the context of the experiment than isolated over-represented terms would be. The graph can be found on this chapter's website.

As we can see from the top GO terms for this contrast in table 1.3 (and the Biological Process graph produced by RamiGO), knocking down the expression of TF1 has had a demonstrable effect on a number of processes related to the progression of the cell cycle. We may simply hypothesise, therefore, that TF1 is an important regulator of these processes. Similar analyses with the other contrasts we have performed allow us to draw similar conclusions as to the function of the other knocked-down transcription factors. The top GO terms for each of these other transcription factor knockdowns also contain a lot of terms indicating involvement of those transcription factors in the cell cycle, suggesting that all of the transcription factors being studied are involved with this important process.

`GOstats` can also be used to investigate statistical enrichment of other functional classifications, for example, the other categories of the Gene Ontology (Molecular Function and Cellular Component) or pathway data, taken from the Kyoto Encyclopedia of Genes and Genomes (KEGG). It is therefore possible to build up a very rich picture of the functions of our lists of genes in this way.

1.6 Summary

Bioconductor provides us with all the tools we need to go from raw data in a public repository to deriving novel conclusions about the function of groups of genes, via data quality control and detection of differentially expressed genes. We have only scratched the surface of the 671 packages available (at time of writing, see `http://bioconductor.org` for an up-to-date list). Import and processing of Affymetrix microarrays is provided by the `affy`[12] and `simpleaffy` packages[13]. Other methods of differential expression analysis are available, in packages such as `RankProd`[14]. Batch effects, systematic errors that can cause many problems with microarray analysis, can be dealt with using the methods available in the `sva` package[15], especially `ComBat`[16]. Bioconductor also doesn't just provide packages for analysing expression microarrays, there is also extensive support for different types of arrays, such as methylation arrays, arrayCGH, SNP arrays etc. Beyond microarrays, there are also many packages for the analysis of proteomics data and latterly next-generation sequencing analysis. Coupled with generic packages for inferring biological meaning, and visualising large and complex datasets, Bioconductor is increasingly a

'one-stop' solution for many bioinformatics analyses.

Acknowledgements

We'd like to thank Neil Perkins (Newcastle University) for the kind permission to use his microarray data for this chapter.

Bibliography

[1] R. C. Gentleman, V. J. Carey, D. M. Bates, B. Bolstad, M. Dettling, S. Dudoit, B. Ellis, L. Gautier, Y. Ge, J. Gentry, K. Hornik, T. Hothorn, W. Huber, S. Iacus, R. Irizarry, F. Leisch, C. Li, M. Maechler, A. J. Rossini, G. Sawitzki, C. Smith, G. Smyth, L. Tierney, J. Y. Yang, and J. Zhang, "Bioconductor: open software development for computational biology and bioinformatics," *Genome Biology*, vol. 5, no. 10, p. R80, 2004.

[2] K. L. Gunderson, "Decoding Randomly Ordered DNA Arrays," *Genome Research*, vol. 14, pp. 870–877, May 2004.

[3] K. Kuhn, "A novel, high-performance random array platform for quantitative gene expression profiling," *Genome Research*, vol. 14, pp. 2347–2356, Nov. 2004.

[4] S. M. Lin, P. Du, W. Huber, and W. A. Kibbe, "Model-based variance-stabilizing transformation for illumina microarray data," *Nucleic Acids Research*, vol. 36, no. 2, p. e11, 2008.

[5] P. Du, W. A. Kibbe, and S. M. Lin, "lumi: a pipeline for processing illumina microarray," *Bioinformatics*, vol. 24, no. 13, pp. 1547–1548, 2008.

[6] G. K. Smyth, "Linear models and empirical bayes methods for assessing differential expression in microarray experiments.," *Statistical Applications in Genetics and Molecular Biology*, vol. 3, p. Article3, 2004.

[7] G. K. Smyth, J. Michaud, and H. S. Scott, "Use of within-array replicate spots for assessing differential expression in microarray experiments," *Bioinformatics*, vol. 21, pp. 2067–2075, May 2005.

[8] P. Du, W. A. Kibbe, and S. M. Lin, "nuID: a universal naming scheme of oligonucleotides for Illumina, Affymetrix, and other microarrays.," *Biology Direct*, vol. 2, p. 16, 2007.

[9] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 289–300, 1995.

[10] S. Falcon and R. Gentleman, "Using gostats to test gene lists for go term association," *Bioinformatics*, vol. 23, no. 2, pp. 257–258, 2007.

[11] M. Ashburner, C. A. Ball, J. A. Blake, D. Botstein, H. Butler, J. M. Cherry, A. P. Davis, K. Dolinski, S. S. Dwight, J. T. Eppig, M. A. Harris, D. P. Hill, L. Issel-Tarver, A. Kasarskis, S. Lewis, J. C. Matese, J. E. Richardson, M. Ringwald, G. M. Rubin, and G. Sherlock, "Gene ontology: tool for the unification of biology. The Gene Ontology Consortium.," *Nature Genetics*, vol. 25, pp. 25–29, May 2000.

[12] L. Gautier, L. Cope, B. M. Bolstad, and R. A. Irizarry, "affy—analysis of affymetrix genechip data at the probe level," *Bioinformatics*, vol. 20, no. 3, pp. 307–315, 2004.

[13] C. L. Wilson and C. J. Miller, "Simpleaffy: a bioconductor package for affymetrix quality control and data analysis," *Bioinformatics*, vol. 21, no. 18, pp. 3683–3685, 2005.

[14] F. Hong, R. Breitling, C. W. McEntee, B. S. Wittner, J. L. Nemhauser, and J. Chory, "Rankprod: a bioconductor package for detecting differentially expressed genes in meta-analysis," *Bioinformatics*, vol. 22, no. 22, pp. 2825–2827, 2006.

[15] J. T. Leek, W. E. Johnson, H. S. Parker, A. E. Jaffe, and J. D. Storey, "The sva package for removing batch effects and other unwanted variation in high-throughput experiments," *Bioinformatics*, 2012.

[16] W. E. Johnson, C. Li, and A. Rabinovic, "Adjusting batch effects in microarray expression data using empirical bayes methods," *Biostatistics*, vol. 8, no. 1, pp. 118–127, 2007.