Analysis of Affymetrix .CEL Files

This is an affymetrix tutorial, for analysis of .CEL files Courtesey of MIT.

I've been working with microarray expression data from the Broad Connectivity Map, and of course decided to download raw data (to work with in R) over using their GUI. The raw data is in the form of ".CEL" files, which I believe are produced by the Affymetrix GeneChip System. What in the world? Basically, we can evaluate mRNA expression of thousands of genes at once, which are put on one of those cool chips. I'm not going to claim to know anything about the preparation of the DNA - that's for the wet lab folk :) What is important to me is that this machine can scan the array, analyze the images, and spit out expression profiles For more details, here is the reference I was reading. For Affymetrix methods (implemented in Bioconductor), I will reference this document

Preparing for Analysis with R

Let's get the data, from here These arrays were "hybridized to tissues from fetal and human liver and brain tissue, courtesy of MIT. I am following along directly from that linked page. You can download and zip to a directory of your choice, and fire up R.

Preprocessing steps are as follows:

1. preprocess the raw data (summarize probe measurements into one measurement for each probe)
2. normalize data from these eight chips
3. calculate Absent/Present calls which attempts to label genes that are "expressed"
4. calculate expression ratios of genes between two different tissues
5. use a common statistical test to identify differentially expressed genes
6. flag low intensity data (most probably background noise)
7. cluster a differentially expressed subset of all genes to identify those with similar expression profiles
8. try to find what functions specific groups of genes (with similar expression profiles) have in common **Note, specific to Affymetrix chips.

Preprocess the raw data

First, fire up R and bioconductor

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

```
## Bioconductor version 2.13 (BiocInstaller 1.12.0), ?biocLite for
help
```

```
biocLite()
```

```
## BioC_mirror: http://bioconductor.org
## Using Bioconductor version 2.13 (BiocInstaller 1.12.0), R
version 3.0.2.
```

```
## Warning: installed directory not writable, cannot update
packages 'boot',
##   'cluster', 'httpuv', 'lattice', 'MASS', 'Matrix', 'nlme',
'Rcpp',
##   'rpart', 'xtable'
```

```
## Old packages: 'httpuv', 'IRanges', 'oligo'
```

Set your working directory to where we downloaded tthe files

```
setwd("/home/vanessa/Documents/Work/GENE_EXPRESSION/tutorial")
library(affy)
```

```
## Loading required package: BiocGenerics
## Loading required package: parallel
## 
## Attaching package: 'BiocGenerics'
## 
## The following objects are masked from 'package:parallel':
## 
##     clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##     clusterExport, clusterMap, parApply, parCapply, parLapply,
##     parLapplyLB, parRapply, parSapply, parSapplyLB
## 
## The following object is masked from 'package:stats':
## 
##     xtabs
## 
## The following objects are masked from 'package:base':
## 
##     anyDuplicated, append, as.data.frame, as.vector, cbind,
##     colnames, duplicated, eval, evalq, Filter, Find, get,
##     intersect, is.unsorted, lapply, Map, mapply, match, mget,
##     order, paste, pmax, pmax.int, pmin, pmin.int, Position,
rank,
##     rbind, Reduce, rep.int, rownames, sapply, setdiff, sort,
##     table, tapply, union, unique, unlist
## 
## Loading required package: Biobase
## Welcome to Bioconductor
## 
##     Vignettes contain introductory material; view with
##     'browseVignettes()'. To cite Bioconductor, see
##     'citation("Biobase")', and for packages
'citation("pkgname")'.
```

Read in the .CEL files

```
files =
list.files("/home/vanessa/Documents/Work/GENE_EXPRESSION/tutorial",
    full.names = TRUE)
affy.data = ReadAffy(filenames = files)
```

normalize data from these eight chips

Summarize and normalize with MAS5

```
eset.mas5 = mas5(affy.data)
```

```
## background correction: mas
## PM/MM correction : mas
## expression values: mas
## background correcting...
```

```
##
```

```
## done.
## 12626 ids to be processed
## |                    |
## |####################|
```

This is what's going on, accoring to the script

- background correction: mas
- PM/MM correction : mas
- expression values: mas

Specifically, we calculate a background “floor” to subtract from each cell value. This is done by 1. splitting the array into K (default K=16) rectangular zones, not including control or masked cells. 2. Rank the cells, the loweset 2% are chosen as the background b for the zone. 3. The standard deviation of the lowest 2% cell intensities is calculated to estimate variability

After much digging, I finally found good documentation for PM/MM - it's referring to a probe set that includes a Perfect Match (PM) and MisMatch to a reference sequence. We calculate an average difference intensity value

Average Difference Intensity Value

using the two, meaning that if MM > PM, we get negative values.

Expression Values

The probe pair gets weighted based on the difference from the mean. We then take the mean of the weighted intensity values and that becomes the mean for our Kth set. We then correct for negative values, and the signals are scaled based on this mean, meaning that we have background adjusted values. Now we have normalized expression values, and we can look at the expression matrix! That was so darn easy.

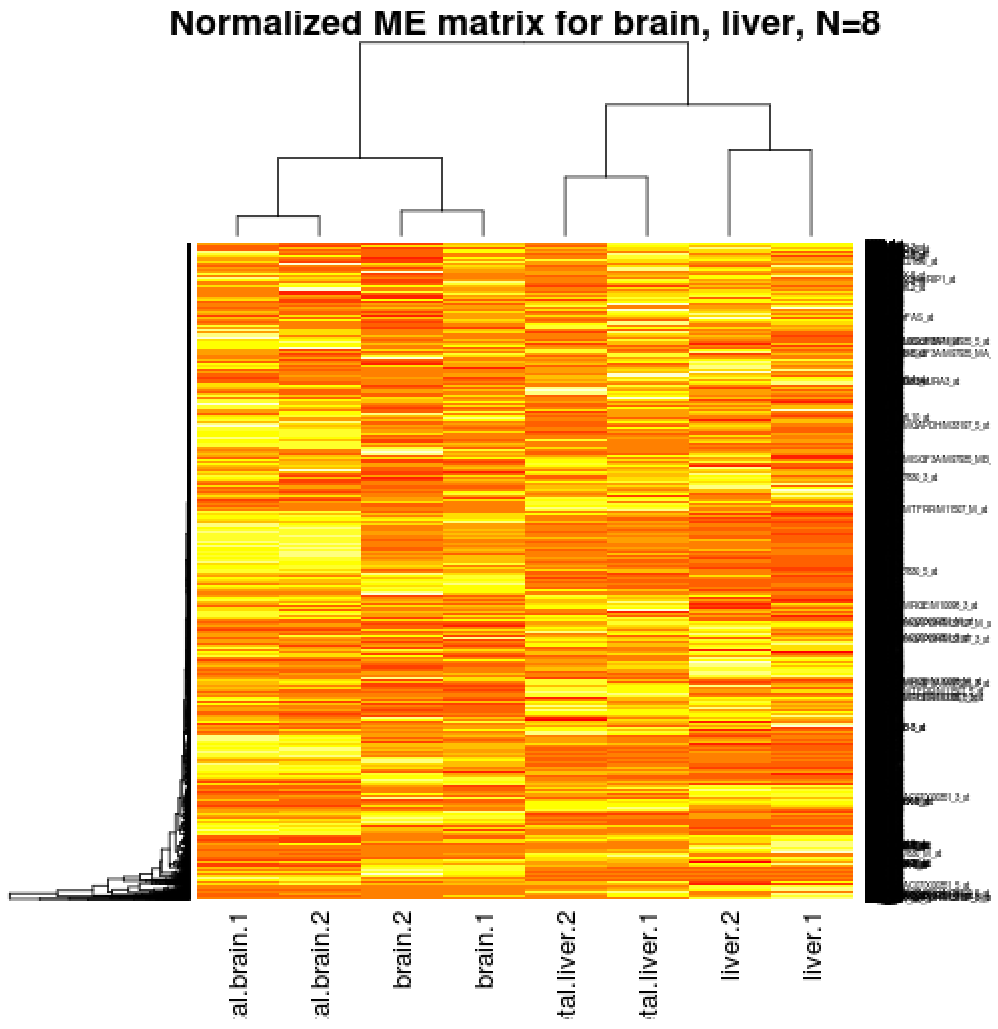
```
exprSet.nologs = exprs(eset.mas5)
# List the column (chip) names
colnames(exprSet.nologs)
```

```
## [1] "Brain_1.CEL"       "Brain_2.CEL"       "Fetal_brain_1.CEL"
## [4] "Fetal_brain_2.CEL" "Fetal_liver_1.CEL" "Fetal_liver_2.CEL"
## [7] "Liver_1.CEL"       "Liver_2.CEL"
```

```
# Rename the column names if we want
colnames(exprSet.nologs) = c("brain.1", "brain.2",
"fetal.brain.1", "fetal.brain.2",
    "fetal.liver.1", "fetal.liver.2", "liver.1", "liver.2")
```

Heatmap it up!

```
heatmap(exprSet.nologs, main = "Normalized ME matrix for brain,
liver, N=8")
```



How cool! Brain clusters with brain, and liver with liver. Our work is done! Just kidding :) Let's log transform the data so the distribution is more normal.

```
exprSet = log(exprSet.nologs, 2)
```

calculate Absent/Present calls which attempts to label genes that are “expressed”

Next let's calculate an Absent/Present Call for each probe set:

```
# Run the Affy A/P call algorithm on the CEL files we processed
above
data.mas5calls = mas5calls(affy.data)
```

```
## Getting probe level data...
## Computing p-values
## Making P/M/A Calls
```

```
# Get the actual A/P calls
data.mas5calls.calls = exprs(data.mas5calls)
```

This is a matrix of values “P” and “A” to indicate present and absent.

calculate expression ratios of genes between two different tissues

From the clustering above, it's easy to see that there are differences in the values between the two tissues. But now let's calculate expression ratios of genes between the two tissues. We start by calculating the mean of each pair of replicated experiments (since we have 2 tissue types across 8 chips, this means that we will get 4 means (brain - liver for each))

```
brain.mean = apply(exprSet[, c("brain.1", "brain.2")], 1, mean)
fetal.brain.mean = apply(exprSet[, c("fetal.brain.1",
"fetal.brain.2")], 1,
    mean)
liver.mean = apply(exprSet[, c("liver.1", "liver.2")], 1, mean)
fetal.liver.mean = apply(exprSet[, c("fetal.liver.1",
"fetal.liver.2")], 1,
    mean)
```

Each of these means is a vector of length 12626, because we have a mean for each gene across each of the tissues, respectively. Now we can calculate the ratios (for both brain and liver). This means that we are taking four experiments, and coming up with two rations:

- liver fetal vs liver adult
- brain fetal vs brain adult

Remember logarithms from high school? I don't either. But I do remember that log(A / B) == log(A) - log(B), so since these are log transformed, we can take advantage of that:

```
brain.fetal.to.adult = fetal.brain.mean - brain.mean
liver.fetal.to.adult = fetal.liver.mean - liver.mean
```

These are still of the same length, but now represent the expression ratios of genes between liver/brain fetal and adult. Let's ploop them into one big matrix:

```
all.data = cbind(exprSet, brain.mean, fetal.brain.mean,
liver.mean, fetal.liver.mean,
    brain.fetal.to.adult, liver.fetal.to.adult)
# Check what data we have here
colnames(all.data)
```

```
##  [1] "brain.1"              "brain.2"
"fetal.brain.1"
##  [4] "fetal.brain.2"        "fetal.liver.1"
"fetal.liver.2"
##  [7] "liver.1"              "liver.2"              "brain.mean"
## [10] "fetal.brain.mean"     "liver.mean"
"fetal.liver.mean"
## [13] "brain.fetal.to.adult" "liver.fetal.to.adult"
```

Are the differences significant?

We can now use a common statistical test to identify differentially expressed genes. There are multiple ways to do this - t-tests, fold change, and bootstrapping, and this document outlines the pros and cons of each. Let's use a t-test, specifically Welch's test (doesn't assume equal variance between samples) and two tailed, because we have up and down regulated genes. Here we specify dataset1, dataset2, and 2 tailed, but ONLY for the first gene:

```
dataset.1 = exprSet[1, c("brain.1", "brain.2")]
dataset.2 = exprSet[1, c("fetal.brain.1", "fetal.brain.2")]
t.test.gene.1 = t.test(dataset.1, dataset.2, "two.sided")
dataset.1
```

```
## brain.1 brain.2
##   9.367   9.502
```

```
dataset.2
```

```
## fetal.brain.1 fetal.brain.2
##         7.891         7.959
```

```
cat("PValues")
```

```
## PValues
```

```
t.test.gene.1$p.value
```

```
## [1] 0.00895
```

Now let's look at… ALL genes (evil laugh)

```
brain.p.value.all.genes = apply(exprSet, 1, function(x) {
    t.test(x[1:2], x[3:4])$p.value
})
liver.p.value.all.genes = apply(exprSet, 1, function(x) {
    t.test(x[5:6], x[7:8])$p.value
})
# Sanity check - do we get the same for the first gene?
brain.p.value.all.genes[1:5]
```

```
##  100_g_at   1000_at   1001_at 1002_f_at 1003_s_at
##   0.00895   0.04942   0.52106   0.80012   0.13491
```

flag low intensity data (most probably background noise)

We now do a bit of quality checking - we want to get a pvalue that reflects confidence in a particular probe being absent or present in our samples, and Affymetrix has given us methods to do so. We want to use the “Absent/Present” calls we did previously to flag questionable genes. We want to “flag” the genes with “Absent” calls. We can do this in one fell swoop (is that the right expression?) by doing the following:

```
# Concatenate all A/P calls for brain and liver
AP = apply(data.mas5calls.calls, 1, paste, collapse = "")
```

Now we want to select ONLY rows (a row is genes across probesets) that are expressed in no brain or liver hybridizations. By creating a smaller subset, this means that we can correct for fewer hypotheses (score!)

```
# Get the probsets where the 4 calls are not 'AAAA'
genes.present = names(AP[AP != "AAAAAAAA"])
# How many probetset/genes are present?
length(genes.present)
```

```
## [1] 7924
```

```
# Get all data for probesets that are present on at least on chip.
exprSet.present = exprSet[genes.present, ]
```

Now we have to correct for multiple hypotheses, the False Discovery Rate (FDR) Here are our raw p-values (uncorrected):

```
brain.raw.pvals.present = brain.p.value.all.genes[genes.present]
liver.raw.pvals.present = liver.p.value.all.genes[genes.present]
```

Generate FDR corrected values with p.adjust, sort them to get the lowest (most significant), and check out the lowest 10:

```
brain.fdr.pvals.present = p.adjust(brain.raw.pvals.present, method
= "fdr")
liver.fdr.pvals.present = p.adjust(liver.raw.pvals.present, method
= "fdr")
brain.fdr.pvals.present.sorted =
brain.fdr.pvals.present[order(brain.fdr.pvals.present)]
liver.fdr.pvals.present.sorted =
liver.fdr.pvals.present[order(liver.fdr.pvals.present)]
brain.fdr.pvals.present.sorted[1:10]
```

```
##  1107_s_at     118_at  1194_g_at  1247_g_at    1529_at
1629_s_at
##     0.2442     0.2442     0.2442     0.2442     0.2442
0.2442
##    1903_at 32041_r_at   32059_at   32273_at
##     0.2442     0.2442     0.2442     0.2442
```

```
liver.fdr.pvals.present.sorted[1:10]
```

```
##   39035_at 32340_s_at   32612_at   33230_at   38316_at
652_g_at
##     0.2071     0.2346     0.2346     0.2346     0.2346
0.2346
##   36036_at   32590_at   35702_at   674_g_at
##     0.2753     0.2762     0.2776     0.2776
```

Nope, nothing is significant! We can't use the raw values, because we will have a high rate of false positives. Since we want to move forward with clustering, let's use the uncorrected values, and set a significance threshold of p < 0.01 for genes expressed in both samples. First, let's get a list (one for each tissue) of gene IDs with pvalues < 0.01.

```
brain.DE.probesets =
names(brain.raw.pvals.present[brain.raw.pvals.present <
    0.01])
liver.DE.probesets =
names(liver.raw.pvals.present[liver.raw.pvals.present <
    0.01])
```

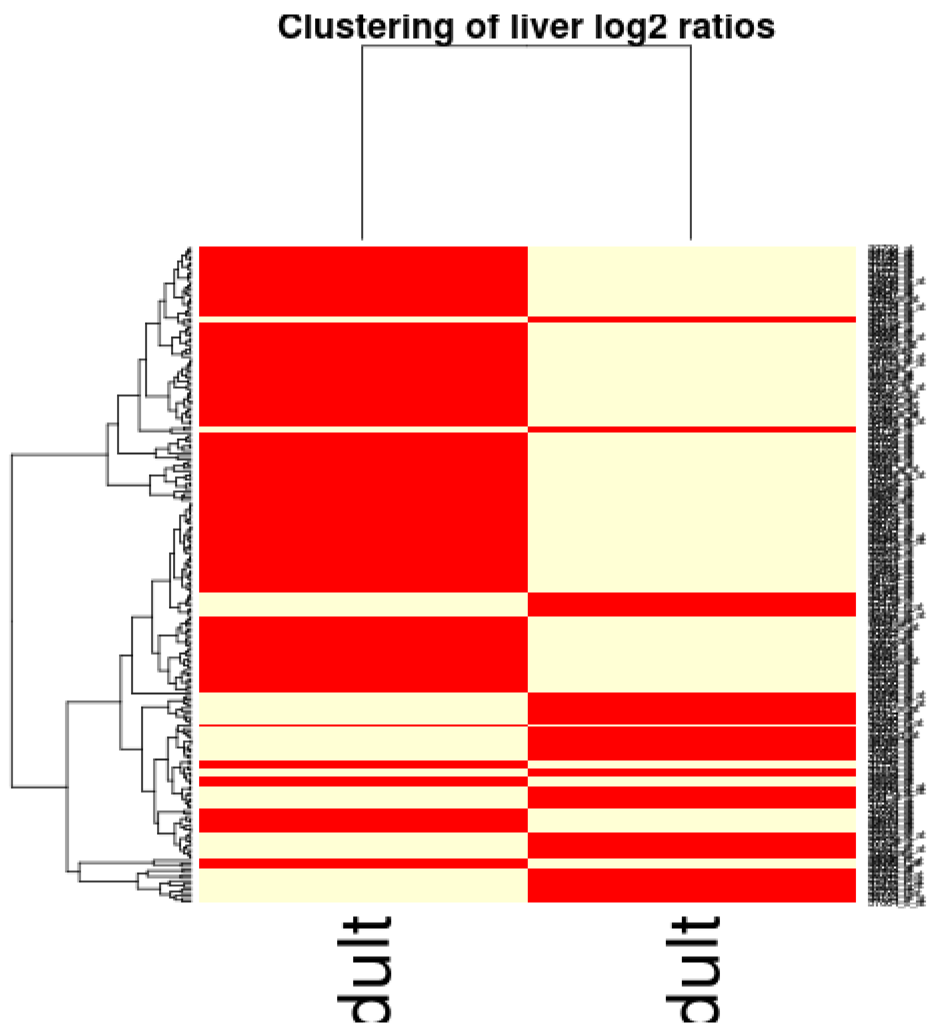
Get the expression data (log2 ratios) for these probesets (separate datasets for brain and liver).

```
brain.DE.log2.ratios = all.data[brain.DE.probesets,
c("brain.fetal.to.adult",
    "liver.fetal.to.adult")]
liver.DE.log2.ratios = all.data[liver.DE.probesets,
c("brain.fetal.to.adult",
    "liver.fetal.to.adult")]
```

cluster a differentially expressed subset of all genes to identify those with similar expression profiles

To cluster, I'm just going to use the heatmap function again, since that will automatically bicluster the data.

```
heatmap(liver.DE.log2.ratios, main = "Clustering of liver log2
ratios")
```



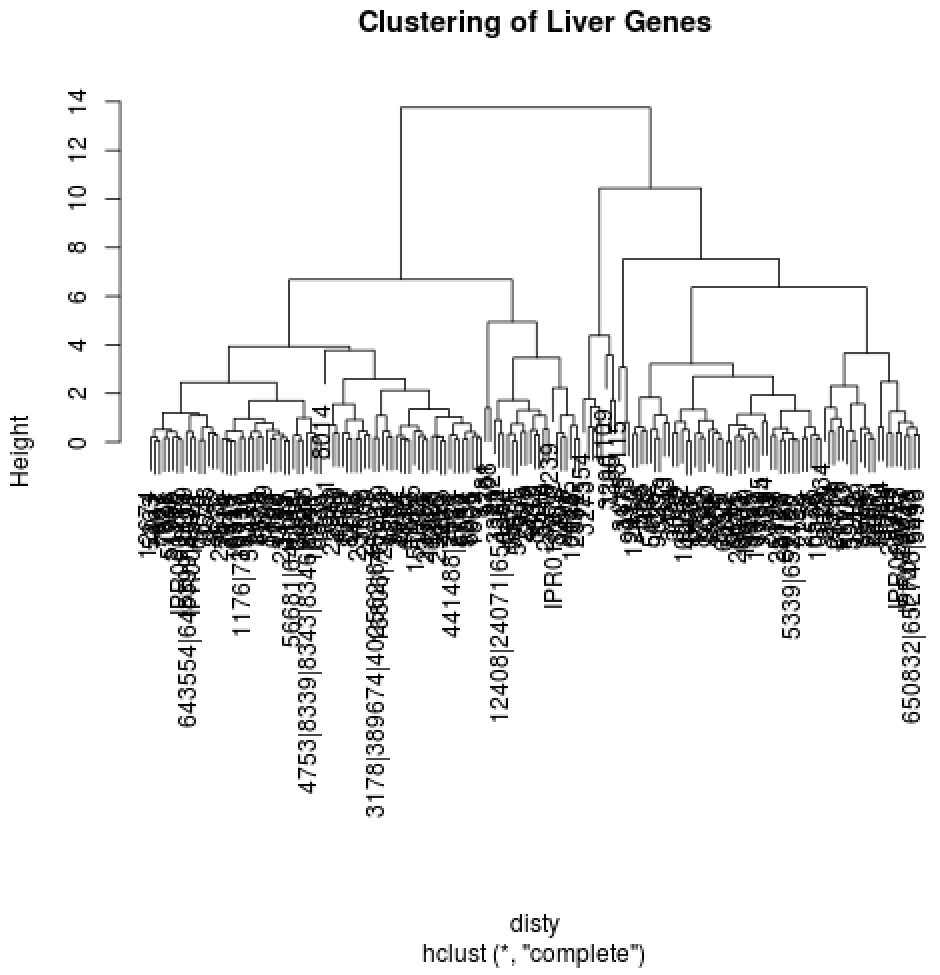
We can see our genes are grouped on the left based on similar expression profiles. At this point, we have some gene sets, and we want to understand what is interesting about those sets. This is where the gene ontology is useful (GO). Before we cluster, let's also change the gene groups to entrez IDs, for later.

```
entrez_id = read.csv("/home/vanessa/Desktop/entrezlist.txt", sep =
" ")
# 5th column is affy ID, 4th is entrez_gene
tmp = rownames(liver.DE.log2.ratios)
eid = entrez_id[which(entrez_id[, 3] %in% tmp), 1]

# Filter data to include subset we have lookups for
tmp = liver.DE.log2.ratios[which(rownames(liver.DE.log2.ratios)
%in% entrez_id[,
    3]), ]
rownames(tmp) = eid
```

Now we can do clustering with hclust:

```
disty = dist(tmp)
hc = hclust(disty)
plot(hc, main = "Clustering of Liver Genes")
```



```
groups = cutree(hc, 6)
gene_groups = list()
for (g in 1:6) {
    gene_groups = c(gene_groups, list(rownames(tmp)[which(groups
== g)]))
}
```

It's of course always problematic deciding where to cut the tree. Glancing at the plot, I'm going to give 6 clusters a try.

Now let's make a GO tree! First prepare all the packages (these need to be installed with bioconductor if you don't have them)

```
library(org.Hs.eg.db)
```

```
## Loading required package: DBI
```

```
library(GO.db)
```

```
##
```

```
library("GOstats")
```

```
## Loading required package: Category
## Loading required package: Matrix
## Loading required package: graph
##
## Attaching package: 'GOstats'
##
## The following object is masked from 'package:AnnotationDbi':
##
##     makeGOGraph
```

Get all the mapped genes from GO, first read in entrez id:

```
x <- org.Hs.egGO
entrez_object <- org.Hs.egGO
mapped_genes <- mappedkeys(entrez_object)
entrez_to_go <- as.list(x[mapped_genes])
go_object <- as.list(org.Hs.egGO2EG)
```

And here we will save a list of our annotations, we have six groups, so this will be a list of six lists.

```
GO = list()
for (t in 1:length(gene_groups)) {
    g = gene_groups[[t]]
    # Find the indices for entrez gene ids that we have in our
subset
    idx = which(mapped_genes %in% g)
    go_subset = entrez_to_go[idx]
    idx = which(g %in% mapped_genes)
    genes = as.character(g[idx])
    genes = unique(genes)
    universe = mapped_genes

    # Now perform enrichment analysis
    params <- new("GOHyperGParams", geneIds = universe,
universeGeneIds = universe,
        ontology = "BP", pvalueCutoff = 0.001, conditional = F,
testDirection = "over",
        annotation = "org.Hs.eg.db")
    hgOver <- hyperGTest(params)
    GO = c(GO, hgOver)
}
```

If we look at each of these results, we can see what functions the groups of genes have in common. We can also create a nice tree to see this visually:

```
# library('RamiGO') for (t in 1:length(GO)) { g = GO[t] color =
# rainbow(length(g))[rank(g)] pngRes <- getAmigoTree(goIDs=g,
color=color,
#
filename=paste('/home/vanessa/Desktop/group',t,'_amiGO.png',sep=''),
# picType='png', saveResult=TRUE)
```

Or if you want to just paste the GO ids into the GUI:

```
GO[1]$GOBPID
```

```
## NULL
```

Unfortunately, my groups didn't have any significant findings, so I can't show you any pretty images.