1 Initial comments

If you have used the NGSexpressionSet from my GitHub account and are now moving to this updated summary class instead, you have to change the package atribute for your old objects like that before you can use them again!

```
## not run
if ( FALSE) {
        attr(class(oldObject),'package') <- 'StefansExpressionSet'
}
```

1.1 Install

You can install the package from github. More information can be found at my GitHub page.

2 Introduction

The NGSexpressionSet R S4 class has been produced to make my live easier. It is not developed for a larger audience and therefore some functions might not be flexible enought for all workflows.

The overall aim of this package is to (1) keep all analysis results in one object and (2) simplify the plotting by using the previousely stored objects.

The aim of this document is not to explain all options for the functions, but to give one working example of my workflow. Please use the R help system to get more information on all functions - if necessary.

But not too much here lets start.

3 Get your data into the object

The package comes with example data; A data.frame containing count data published in PMID25158935 re-mapped and re-quantified using DEseq (dim:24062, 17) and one data.frame containing the sample information (dim:15, 20):

```
        PMID25158935exp[1:5,1:7]
```

```
##   GeneID Length ERR420371 ERR420372 ERR420373 ERR420374 ERR420375
## 1   Xkr4   3634         0         0         0         0         0
## 2    Rp1   9747         0         0         0         0         0
## 3  Sox17   4095         0         0         0         0         0
## 4 Mrpl15   4201     24888     26974     30814     26962     19968
## 5 Lypla1   2433     46203     21090     68584     28491     26469
```

These two tables are loaded into a NGSexpressionSet object using the command:

```
PMID25158935 <- NGSexpressionSet( PMID25158935exp,
        PMID25158935samples, Analysis = NULL, name='PMID25158935',
        namecol='Sample', namerow= 'GeneID', usecol=NULL ,
        outpath = ''
)
```

This function call is already very workflow focused as the options Analysis and usecol can be used to subselect samples in the samples table and create a smaller than possible object from the count data.

The object does print like that:

```
        PMID25158935

## An object of class NGSexpressionSet
## named  PMID25158935
## with 24062 genes and 15  samples.
## Annotation datasets (24062,2): 'GeneID', 'Length'
## Sample annotation (15,20): 'Source.Name', 'Comment.ENA_SAMPLE', 'Provider', 'Characterist
```

4 Subsetting the object

This is the main purpose why I created the class in the first place. An easy way to consitently subset multiple tables at the same time.

I have implemented two functions: "reduce.Obj" which subsets the object to a list of genes and "drop.samples" which does - guess what - drop samples.

```
        reduced <- reduce.Obj(PMID25158935,
                        sample(rownames(PMID25158935@data), 100),
                        name="100 genes" )
        reduced

## An object of class NGSexpressionSet
## named  100 genes
## with 100 genes and 15  samples.
## Annotation datasets (100,2): 'GeneID', 'Length'
## Sample annotation (15,20): 'Source.Name', 'Comment.ENA_SAMPLE', 'Provider', 'Characterist
```

```
        dropped <- drop.samples(
                        PMID25158935,
                        colnames(PMID25158935@data)[1:3],
                        name='3 samples dropped'
        )
        dropped
```

```
## An object of class NGSexpressionSet
## named  3 samples dropped
## with 24062 genes and 12  samples.
## Annotation datasets (24062,2): 'GeneID', 'Length'
## Sample annotation (12,20): 'Source.Name', 'Comment.ENA_SAMPLE', 'Provider', 'Characterist
```

```
        subs <- reduce.Obj ( PMID25158935,
                        rownames(PMID25158935@data)[
                                        order( apply( PMID25158935@data, 1, sd), decreasing=
                        ],
                        'max_sd_genes'
        )
```

An additional function 'restrictSamples' removes samples based on a match on a variable in the samples table.

```
levels( PMID25158935@samples[,'Characteristics.cell.type'])
```

```
## [1] "hematopoietic stem cells"
## [2] "multipotent progenitor cell, fraction 1"
## [3] "multipotent progenitor cell, fraction 2"
## [4] "multipotent progenitor cell, fraction 3"
## [5] "multipotent progenitor cell, fraction 4"
```

```
dropped <- restrictSamples(
        PMID25158935,
        column ='Characteristics.cell.type',
        value= 'multipotent progenitor',
        mode='grep',
        name='only HSC left'
)
dropped
```

```
## An object of class NGSexpressionSet
## named  only HSC left
## with 24062 genes and 4  samples.
## Annotation datasets (24062,2): 'GeneID', 'Length'
## Sample annotation (4,20): 'Source.Name', 'Comment.ENA_SAMPLE', 'Provider', 'Characteristi
```

5 Unconventional checks

I have implemented a rather unconventional check for the NGS data objects: reads.taken(). This function checks the percentage of reads consumed by the top 5 percent of genes and thereby creates a measurement of the library complexity. I have seen very complex NGS libraries with about 60-70% reads consumed by

the top 5% genes and very shallow libraries where all 100% of reads were specific to the top 5% genes.

My rule of thub: a good (mouse) expression dataset should not use more than 77% of the reads in the top 5% of the genes.

```
        reads.taken(PMID25158935)$reads.taken

## ERR420375 ERR420376 ERR420384 ERR420380 ERR420379 ERR420372 ERR420377
## 0.6037290 0.6224907 0.6136839 0.5901804 0.6105994 0.6235713 0.6033397
## ERR420383 ERR420373 ERR420381 ERR420371 ERR420382 ERR420374 ERR420378
## 0.6164420 0.6189190 0.6131443 0.6052369 0.6086387 0.6098405 0.6159916
## ERR420385
## 0.6115526
```

Here comes an example of my workflow combination: I have implemented a function, that gives back the 'failed' samples for the reads.taken function: check.depth. In the default version the function checks whether the 5% reads taken value is higher than the default cutoff of 77%. But this test is uninformative in this extremely good dataset. Therefore I had to lower the cutoff value to 0.62 here. The sample names can be directly fead into the drop.samples function to subset the data.

```
check.depth(PMID25158935,cutoff=0.62 )

## [1] "ERR420376" "ERR420372"

drop.samples(PMID25158935, check.depth(PMID25158935,cutoff=0.62))

## An object of class NGSexpressionSet
## named  dropped_samples
## with 24062 genes and 13  samples.
## Annotation datasets (24062,2): 'GeneID', 'Length'
## Sample annotation (13,20): 'Source.Name', 'Comment.ENA_SAMPLE', 'Provider', 'Characterist
```

6 Statistics

The statistic analysis is also keeping my workload low: One call runs them all.

But unfortunaltely this is broken! Need to fix that :-( )

```
        ## from these values I can choose to create statistics:
        colnames(PMID25158935@samples)

##  [1] "Source.Name"               "Comment.ENA_SAMPLE"
##  [3] "Provider"                  "Characteristics.organism"
##  [5] "Characteristics.strain"    "Characteristics.cell.type"
```

```
##  [7] "Material.Type.1"           "Comment.LIBRARY_LAYOUT"
##  [9] "Comment.LIBRARY_SOURCE"    "Comment.LIBRARY_STRATEGY"
## [11] "Comment.LIBRARY_SELECTION" "Performer"
## [13] "GroupName"                 "Technology.Type"
## [15] "Comment.ENA_EXPERIMENT"    "Scan.Name"
## [17] "Sample"                    "Comment.FASTQ_URI"
## [19] "Factor.Value.cell.type"    "bam filename"
```

```
        ## and the GroupName might be the best to start from
        table(PMID25158935@samples$GroupName)
```

```
##
##  HSC MPP1 MPP2 MPP3 MPP4
##    4    3    3    3    2
```

```
        #withStats <- createStats( subs, 'GroupName' )
```

7 Grouping

The underlying logics in the grouping is that all grouping functions add the grouoing results directly into either the samples or annotation tables, add a color for this grouping to the StefansExpressionSet object and return the object, not the grouping.

The aim for the final object is to have a wide set of grouping functionallity in the object ranging from simple hclust calls to more complex PCA based approaches to my favorite grouping, the random forest unsupervised grouping. Not all is implemented.

7.1 group.hclust

This function uses the hclust algorithm to cluster the data and add the grouping information into the samples resp. annotation tables. Please try to keep the grouing names unique as new groups with the same name will replace old ones and you are not able to change colors for a single groups if they share one name. Please read the documentation on the R command line for all the different options.

```
subs <- group.hclust ( subs, groups=4, name='hclust genes', type='gene' )

table(subs@annotation[,c('hclust genes 4 groups')])
```

```
##
##  1  2  3  4
##  1  4 45 50
```

```
subs <- group.hclust ( subs, groups=4, name='hclust samples', type='sample' )

table(
  apply(
        subs@samples[,c('GroupName', 'hclust samples 4 groups')],
        1,      paste,  collapse= "/gr.")
)

##
##  HSC/gr.1  HSC/gr.2 MPP1/gr.1 MPP1/gr.2 MPP2/gr.2 MPP2/gr.3 MPP2/gr.4
##         3         1         2         1         1         1         1
## MPP3/gr.2 MPP3/gr.4 MPP4/gr.2 MPP4/gr.3
##         2         1         1         1
```

7.2 rfCluster_col

The most interesting grouping function is rfCluster_col(). It utilizes an unsupervised random forest to calculate the distance matrix for the data. As this process is very computer intensive the function allows the calculation to be run on a sun grid engine cluster (SGE), but you can also use the clustering method on you local computer.

7.2.1 Usage

This function has been developed to cluster single cell data with hundreads or thousands of samples.

The rfCluster_col run will create a lot of outout data that you can delete after the grouping process is finished. The files are in the objects outpath/RFclust.mp/ folder. The files starting with runRFclust are all connected with the spawned calculation threads; the objects name'_RFclust_*ID*.RData' files are the subset of the original data for one run and the other *.RData files are the saved random forest distributions.

The random forest output is read into the object after a second run of the same function call. Make sure you use the right StefansExpressionSet object for that!

```
        subs

## An object of class NGSexpressionSet
## named  max_sd_genes
## with 100 genes and 15  samples.
## Annotation datasets (100,4): 'GeneID', 'Length', 'hclust genes order', 'hclust genes 4 gr
## Sample annotation (15,22): 'Source.Name', 'Comment.ENA_SAMPLE', 'Provider', 'Characterist

        subs.C <- rfCluster_col(subs,
```

```
                        rep=1, # one analysis only
                        SGE=F, # Do not use the SGE extension
                        email='not@important.without.SGE', # necessary
                        k=3, #how many clusters to find
                        slice=4, # how many processes to span per run
                        subset=nrow(subs@samples), # use the whole dataset
                        nforest=5, # how many forets per rep - set that to 500
                        ntree=100, # how many trees per forest - set that to 1000
                        name='RFclust' # the name of this analysis (rename if re-run)
        )
```

```
## [1] "max_sd_genes_RFclust_1 : The data is going to be analyszed now - re-run this functio
## [1] "You should wait some time now to let the calculation finish! -> re-run the function'
## [1] "check: system( 'ps -Af | grep Rcmd | grep -v grep')"
```

```
Sys.sleep(50)
```

```
subs.C <- rfCluster_col(subs.C, ## <- this change is important!!
                rep=1, # one analysis only
                SGE=F, # Do not use the SGE extension
                email='not@important.without.SGE', # necessary
                k=3, #how many clusters to find
                slice=4, # how many processes to span per run
                subset=nrow(subs@samples), # use the whole dataset
                nforest=5, # how many forets per rep - set that to 500
                ntree=100, # how many trees per forest - set that to 1000
                name='RFclust' # the name of this analysis (rename if re-run)
)
```

```
## [1] "Done with cluster 1"
```

```
table(apply(
subs.C@samples[,c('GroupName', 'RFgrouping RFclust 1')],1, paste, collapse= "/gr.")
)
```

```
##
##   HSC/gr.1   HSC/gr.2   HSC/gr.3  MPP1/gr.2  MPP1/gr.4  MPP1/gr.5
##          1          2          1          1          1          1
##  MPP2/gr.5  MPP2/gr.6  MPP2/gr.7  MPP3/gr.7  MPP3/gr.8  MPP3/gr.9
##          1          1          1          1          1          1
## MPP4/gr.10
##          2
```

Once this grouping has been run and the object keeps unchanged, you can create a different grouping based on the same random forest distribution. In order to do that you need the createRFgrouping_col() function.

```
subs.C <- createRFgrouping_col ( subs.C,
        'max_sd_genes_RFclust_1' ,
        k=2,
        single_res_col = 'Our new grouing'
)
table(apply(
subs.C@samples[,c('GroupName', 'Our new grouing')], 1, paste, collapse= "/gr.")
)

##
##  HSC/gr.1  HSC/gr.2 MPP1/gr.1 MPP1/gr.2 MPP2/gr.1 MPP2/gr.2 MPP3/gr.2
##         3         1         2         1         1         2         3
## MPP4/gr.2
##         2
```

7.2.2 TODO

Reduce the memory requirement for the final distance matrix reading process.

7.3 rfCluster_row

The rfCluster_col does cluster samples and the rfCluster_row clusters genes in this object. Otherwise the handling is exactly the same. Apart from the fact, that we have way less samples than genes. Therefore it is extremely important to first select a group of interesting genes from the dataset and run the clustering from there.

All in all this function is not tested enough to be called stable in the way that you get good results back! The results from this run are shown in figure 8.2 on page 13.

```
subs

## An object of class NGSexpressionSet
## named  max_sd_genes
## with 100 genes and 15  samples.
## Annotation datasets (100,4): 'GeneID', 'Length', 'hclust genes order', 'hclust genes 4 gr
## Sample annotation (15,22): 'Source.Name', 'Comment.ENA_SAMPLE', 'Provider', 'Characterist

subs.C <- rfCluster_row(subs.C,
        rep=1, # one analysis only
        SGE=F, # Do not use the SGE extension
        email='not@important.without.SGE', # necessary
        k=3, #how many clusters to find
        slice=4, # how many processes to span per run
        subset=nrow(subs@samples)+1, # use the whole dataset
```

```
        nforest=5, # how many forets per rep - set that to 500
        ntree=100, # how many trees per forest - set that to 1000
        name='RFclust_row' # the name of this analysis (rename if re-run)
)

## [1] "max_sd_genes_RFclust_row_1 : The data is going to be analyszed now - re-run this fu
## [1] "You should wait some time now to let the calculation finish! -> re-run the function'
## [1] "check: system( 'ps -Af | grep Rcmd | grep -v grep')"

Sys.sleep(51)

subs.C <- rfCluster_row(subs.C, ## <- this change is important!!
        rep=1, # one analysis only
        SGE=F, # Do not use the SGE extension
        email='not@important.without.SGE', # necessary
        k=3, #how many clusters to find
        slice=4, # how many processes to span per run
        subset=nrow(subs@samples)+1, # use the whole dataset
        nforest=5, # how many forets per rep - set that to 500
        ntree=100, # how many trees per forest - set that to 1000
        name='RFclust_row' # the name of this analysis (rename if re-run)
)

## [1] "Done with cluster 1"

table(subs.C@annotation[,'RFgrouping RFclust_row 1'])

##
##  1  2  3  4  5  6  7  8  9 10
##  2  3  5 48  2  3  9 12 13  3
```

Once this grouping has been run and the object keeps unchanged, you can create a different grouping based on the same random forest distribution. In order to do that you need the createRFgrouping_row() function.

```
subs.C <- createRFgrouping_row (
subs.C, 'max_sd_genes_RFclust_row_1' , k=2, single_res_row = 'Our new grouing'
)

table(subs.C@annotation[,c( 'Our new grouing')])

##
##  1  2
## 55 45
```

8 Plotting

This is the second most important part of the object.

8.1 Heatmap using ggplot2

I will first explain how to create the ggplot2 plots also used for our shiny server. The function ggplot.gene is described in figure 8.1 on page 11; the function gg.heatmap.list is described in figure 8.1 on page 12.

8.2 Heatmap using heatmap.3

The heatmap.3 function is called internally by the complexHeatmap() function. As the name suggests - this function is far from simple and I recommend reading the internal R documentation (?complexHeatmap at the promt).

In short the function is selecting a number of colujmn (sample) and row (gene) grouping variables and creates the colour bars from this information. The data is going to be sorted after the first variable in both the rowGroups and colGroups variables. If you want to enforce additional ordering you need to order the data yourself using the reorder.samples or reorder.genes functions.

Here I show two calls to the function: one plotting with both column and row groups (figure 8.2 on page 13) and one with only column groups (figure 8.2 on page 14). Two plots mainly to visualize the effect of the random forest gene grouping.

8.3 Grouping colours

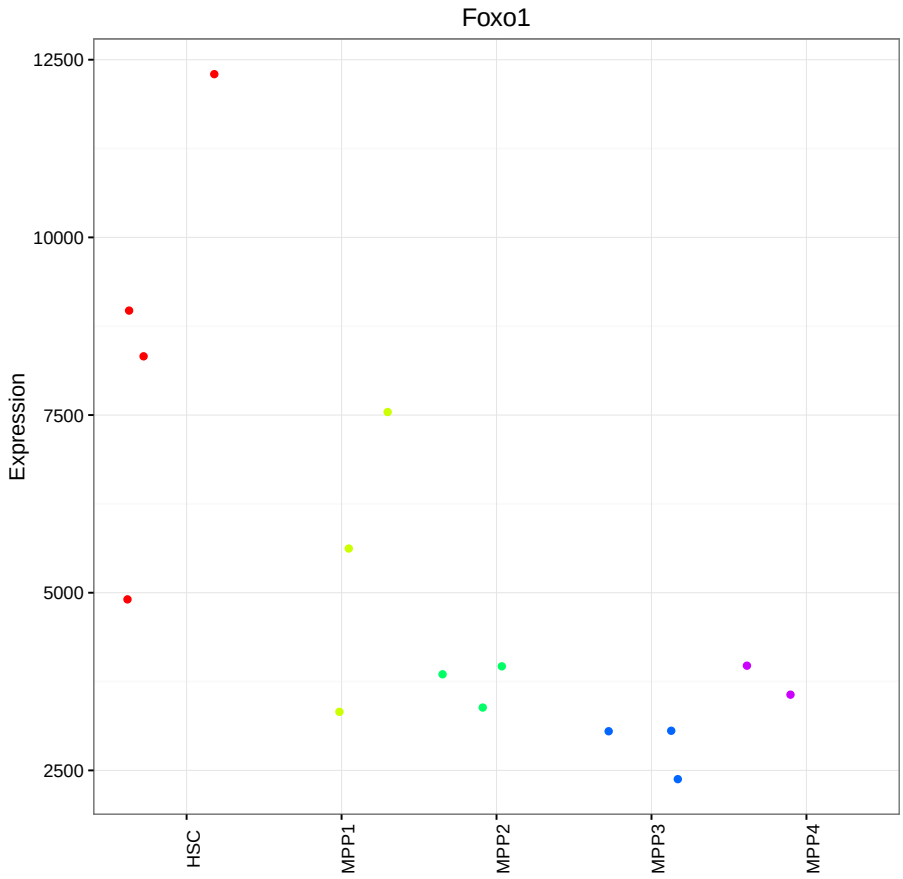
The grouping colours are stored in the usedObj slot of the StefansExpressionSet object in the list entry 'colorRange' using the column name to store a color vector. This colour vector can be changed at any time. The only prerequisite is, that the stored colours match the amout of groups described in the respective grouping.

```
subs.C@usedObj[['colorRange']][['RFgrouping RFclust_row 1']]

##  [1] "#FF0000FF" "#FF9900FF" "#CCFF00FF" "#33FF00FF" "#00FF66FF"
##  [6] "#00FFFFFF" "#0066FFFF" "#3300FFFF" "#CC00FFFF" "#FF0099FF"
```

The most simple way to change the colors is to change these vectors by hand; the most secure way is to use the inbuit color_4 function:

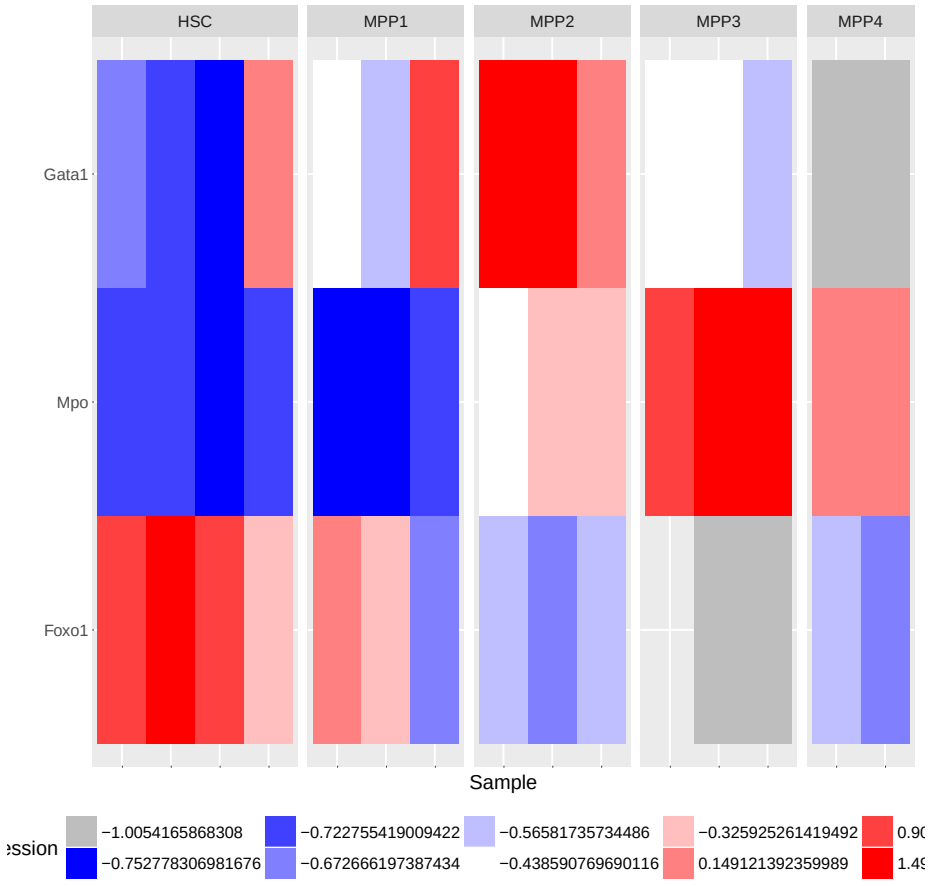
```
subs.C <- colors_4(subs.C, 'RFgrouping RFclust_row 1', function(x){ bluered(x) } )
subs.C@usedObj[['colorRange']][['RFgrouping RFclust_row 1']]

##  [1] "#0000FF" "#4040FF" "#8080FF" "#BFBFFF" "#FFFFFF" "#FFFFFF" "#FFBFBF"
##  [8] "#FF8080" "#FF4040" "#FF0000"
```

```
        ggplot.gene (PMID25158935, 'Foxo1', groupCol='GroupName' )

## Using rownames(ma) as id variables

## $plot
```




```
##
## $not.in
## [1] "NUKL"
```

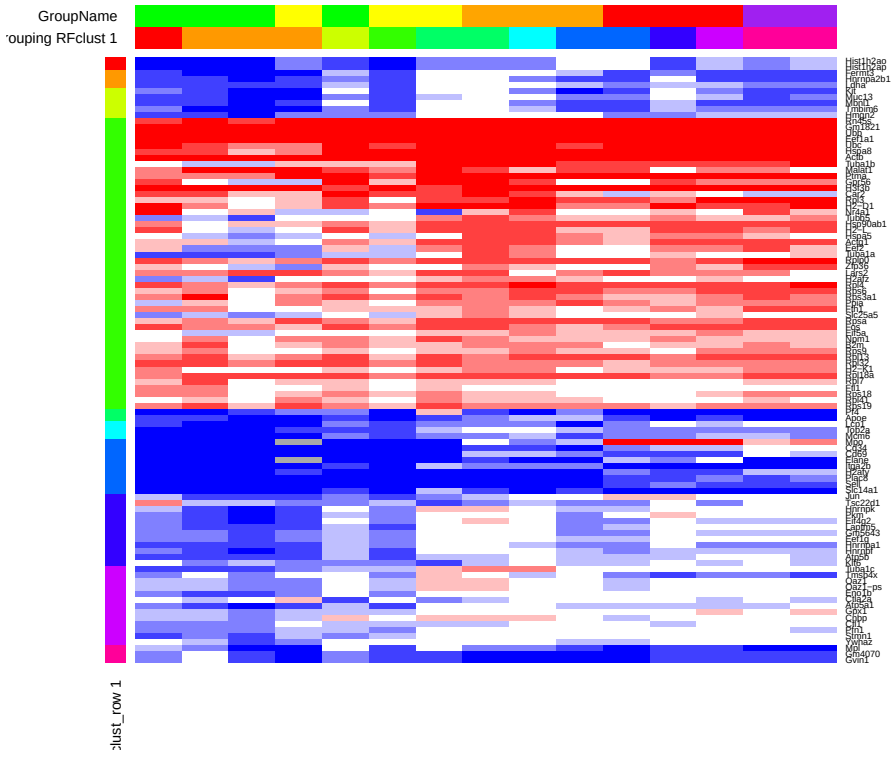
```
        gg.heatmap.list (PMID25158935, c('Mpo', 'Gata1', 'Foxo1'),
                         groupCol='GroupName' )

## Using rownames(ma) as id variables

## $plot
```




```
##
## $not.in
## character(0)
```

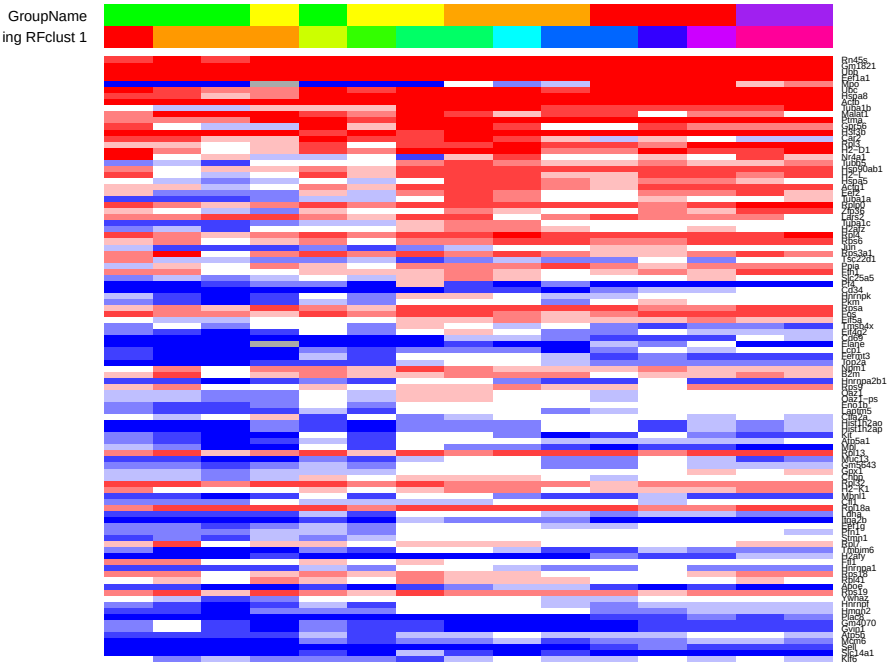
```
# the color information is stored in the subs.C@usedObj[['colorRange']] list
# and we miss the GroupName colours ...
subs.C <- colors_4( subs.C, 'GroupName',
                colFunc=function(x) { c( 'green','yellow', 'orange', 'red', 'purple') }
)
#subs.C <- colors_4( subs.C, 'RFgrouping RFclust 1' )
complexHeatmap (subs.C,
        ofile=NULL,
        colGroups=c('RFgrouping RFclust 1','GroupName'),
        rowGroups='RFgrouping RFclust_row 1',
        pdf=FALSE,
        subpath='',
        main = paste('complexHeatmap in action + RF gene grouping based on', nrow(subs@sampl
        heapmapCols= function(x){ c("darkgrey",bluered(x))}
)
```

plexHeatmap in action + RF gene grouping based on 15 genes

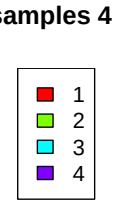


```
# the color information is stored in the subs.C@usedObj[['colorRange']] list
# and we miss the GroupName colours ...
subs.C <- colors_4( subs.C, 'GroupName',
                colFunc=function(x) { c( 'green','yellow', 'orange', 'red', 'purple') }
)
#subs.C <- colors_4( subs.C, 'RFgrouping RFclust_col 1' )
complexHeatmap (subs.C,
        ofile=NULL,
        colGroups=c('RFgrouping RFclust 1','GroupName'),
        pdf=FALSE,
        subpath='',
        main = 'complexHeatmap in action no gene grouping',
        heapmapCols= function(x){ c("darkgrey",bluered(x))}
)
```

complexHeatmap in action no gene grouping



```
## this will create a file 'x@outpath/PlotSomething_hist_4_groups.pdf'
plot.legend( subs, 'hclust samples 4 groups' )
```



The groupings can also be plotted to a separate legend using the plot.legend function:

```
## this will create a file 'x@outpath/PlotSomething_hist_4_groups.pdf'
plot.legend( subs, 'hclust samples 4 groups', file="PlotSomething", pdf=T )
```

The output figure can be seen in figure 8.3 on page 15.