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'
}
```

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 auim of this document is not to explain all options for the functions, but to give one working example.

But not too much here lets start.

```
## Creating a generic function for 't' from package 'base' in package
'StefansExpressionSet'
```

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 (24062, 17) and one data.frame containing the sample information (15, 20):

```
        head(PMID25158935exp)

##   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
## 6  Tcea1   2847     60108     40703     76248     42001     47513
##   ERR420376 ERR420377 ERR420378 ERR420379 ERR420380 ERR420381 ERR420382
## 1         0         2         0         0         8         0         0
## 2         0         0         0         0         0         0         0
```

```
## 3         0         3         0         0         0         0         0
## 4     24629     31956     30394     19962     26656     31353     30320
## 5     31099     64446     52233     33589     54067     38740     44750
## 6     37063     81829     59751     47000     74187     57080     57637
##   ERR420383 ERR420384 ERR420385
## 1         0         0         0
## 2         0         0         0
## 3         0         0        10
## 4     29746     11910     28940
## 5     53342     10846     42312
## 6     72262     27752     60033
```

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 = ''
        )
```

Here you already see, that the object is tailored to my needs: 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. Please use the R help system to get more information on all functions. And 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
```

This should be quite straight forward.

4 Subsetting the object

This is the main purpose why I created the class in the first place. An easy way to consitantly subsett 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.

```
        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 depth.

I have created a 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
```

And this dataset is a very good one. And as I am lasy there is another function, that directly gives the names for the bad samples back: check.depth()

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

## [1] "ERR420376" "ERR420372"
```

You see I have lowered the cutoff so that I find bad samples in this extremely good dataset.

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 grouping part of this object is under development at the moment and likely to gain new functions. The underlying logics in the future grouping will be, that whichever grouping process you ask for, the results will be added to the respective description table slot (samples or annotation) and will be accessible for potting later on.
I am planning to implement most of the Rscexv grouping functions here too. But that will take time.

7.1 rfCluster_col

The most interesting grouping function is rfCluster_col(). It utilizes a unsupervised random foret 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. But you can also use that on you local computer.

7.1.1 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 (100,2): 'GeneID', 'Length'
## Sample annotation (15,20): '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   HSC/gr.4  MPP1/gr.3  MPP1/gr.5
##          1          1          1          1          1          1
##  MPP1/gr.6  MPP2/gr.6  MPP2/gr.7  MPP3/gr.7  MPP3/gr.8  MPP3/gr.9
##          1          2          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.1
##         3         1         2         1         1         2         1
## MPP3/gr.2 MPP4/gr.2
##         2         2
```

7.1.2 TODO

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

7.2 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.

```
        subs
```

```
## An object of class NGSexpressionSet
## named  max_sd_genes
## with 100 genes and 15  samples.
## Annotation datasets (100,2): 'GeneID', 'Length'
## Sample annotation (15,20): '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(50)
```

```
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 10  2  3  4  5  6  7  8  9
##  4 13 29 43  4  1  1  1  2  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_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
## 41 59
```

8 Plotting

This is the second most important part of the object.

8.1 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 10; the function gg.heatmap.list is described in figure 8.1 on page 11.

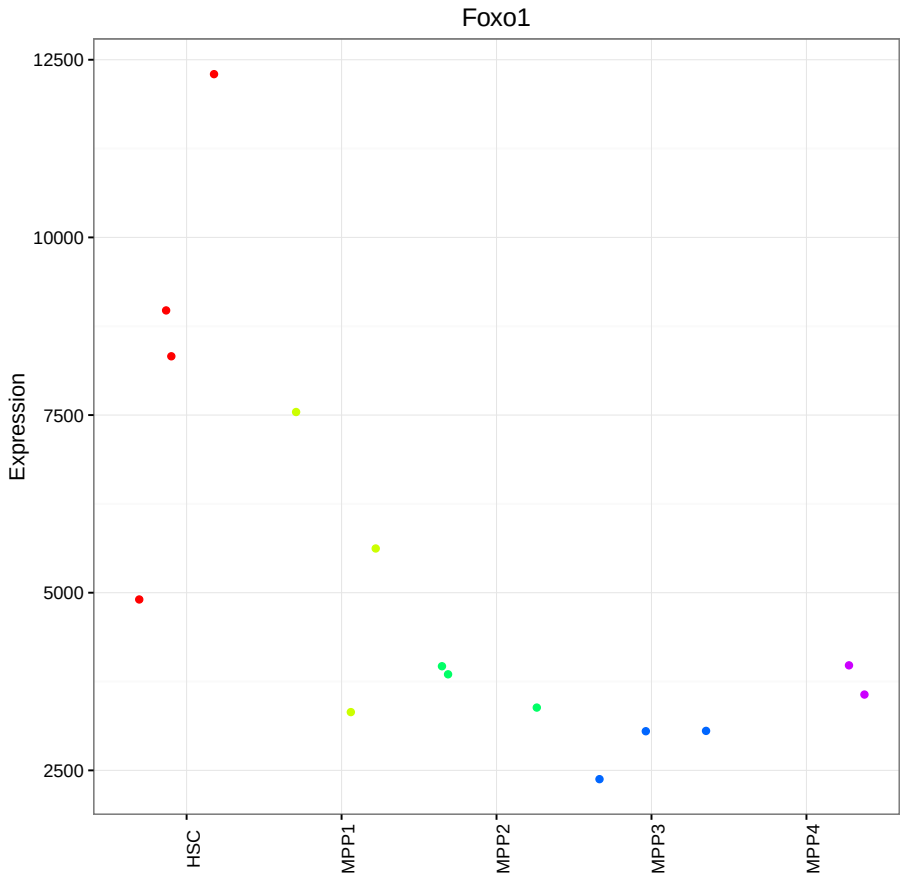
I have created the gg.heatmap.list function in a way, that you can also add column - and row -grouping information to the plot. The first level you have already seen in the groupCol='GroupName' option, but you can enhance that by adding more variables to the groupCol (the first will be used for the facets).

But as you see in figure 8.1 on page 12 the ggplot2 based heatmap.3 is far from perfect at the moment. Instead of putting a lot of time into this function I have implemented a call to heatmap.3 into the object.

8.2 heatmap.3

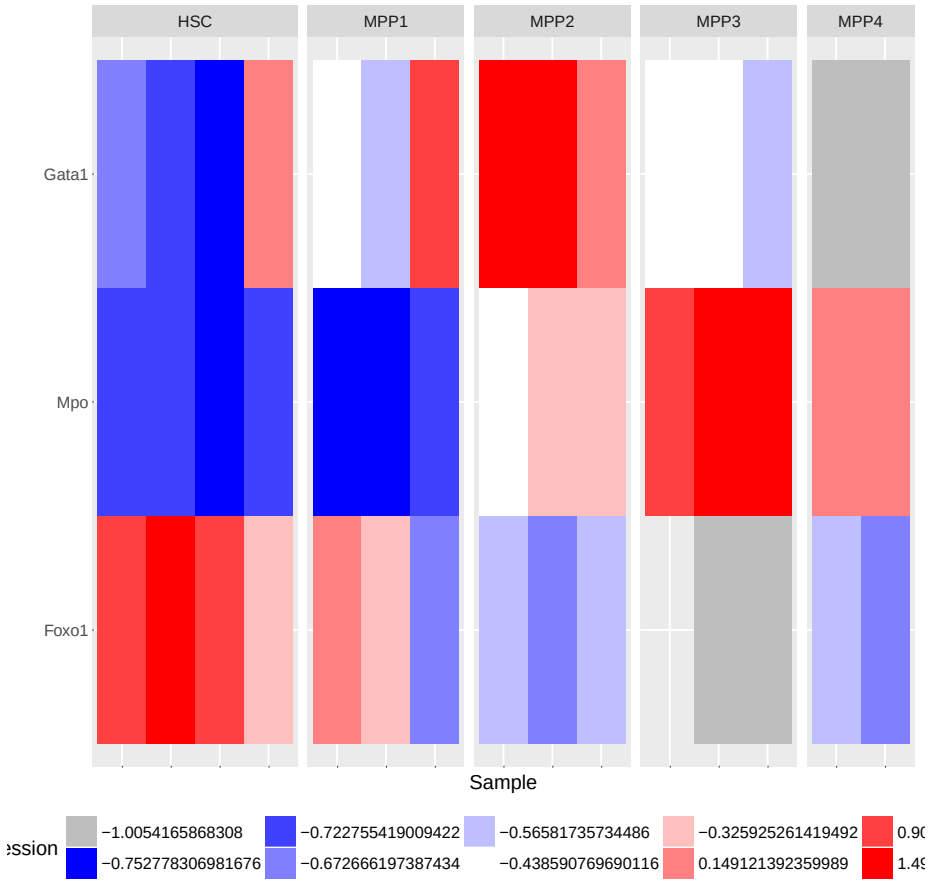
The heatmap.3 function is called internally by the complexHeatmap() function. As the name suggests - this function is far from simple and I recommend reading of the internal R documentation (?complexHeatmap at the promt).

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

## Using rownames(ma) as id variables

## $plot
```



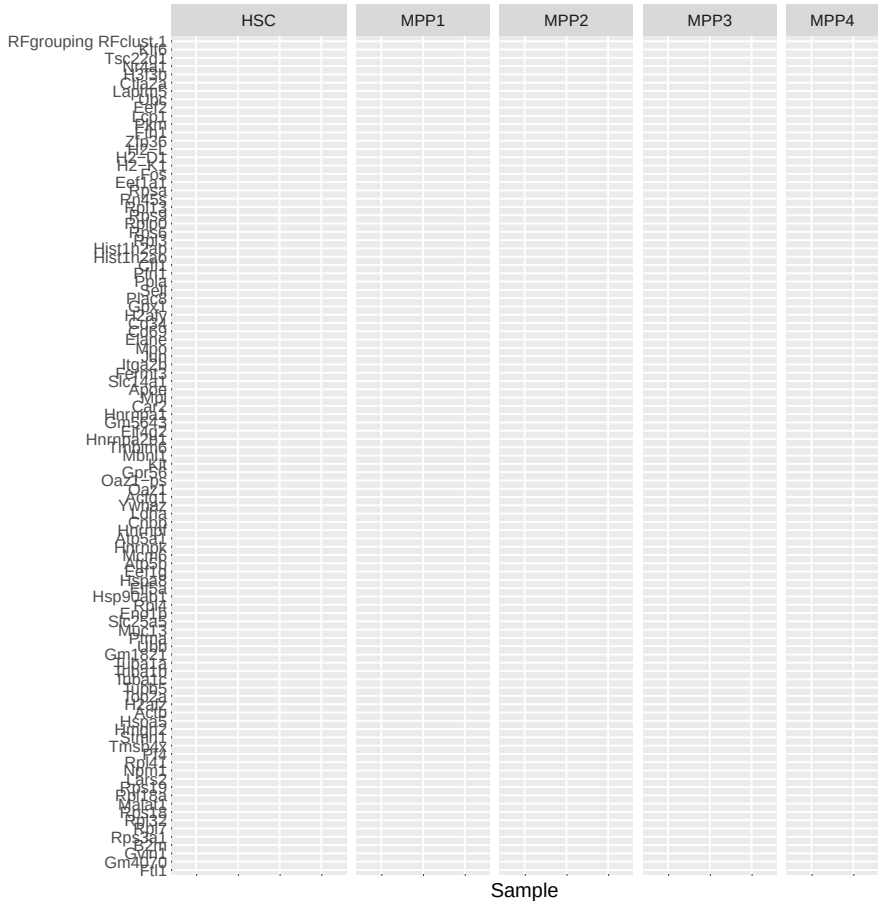

```
##
## $not.in
## [1] "NUKL"
```

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

## Using rownames(ma) as id variables

## $plot
```




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

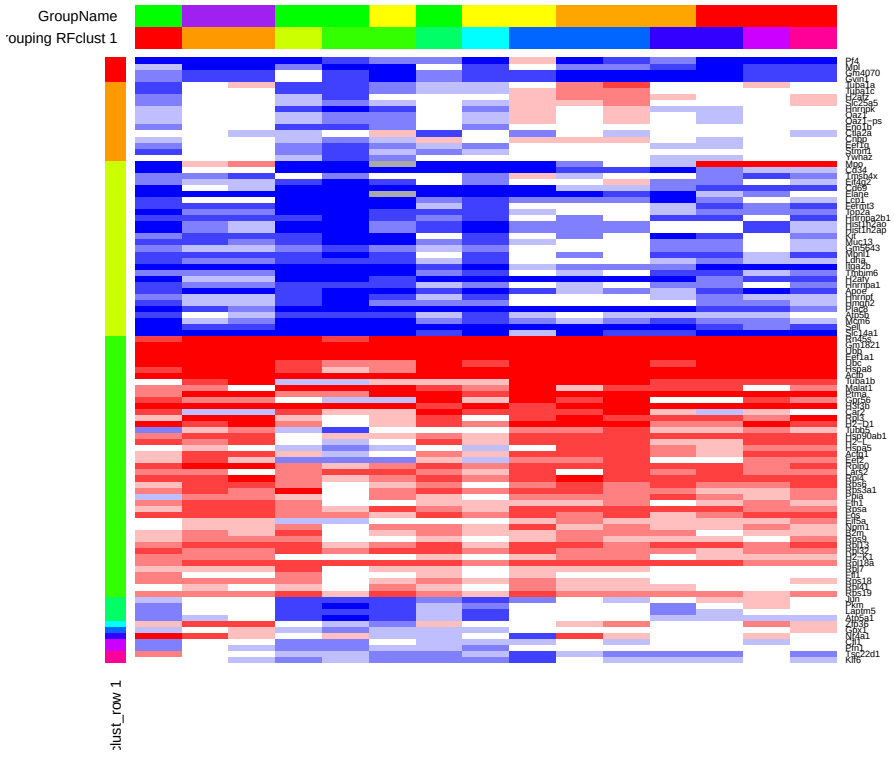
```
        gg.heatmap.list (subs.C,
                         groupCol=c( 'GroupName' ),
                         colCol= c( 'RFgrouping RFclust 1' )
        )

## Using rownames(ma) as id variables

## $plot
```




```
##
## $not.in
## NULL
```

```
# 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 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