MCbiclust 0.1

Robert Bentham[*1]

[1]Cell and Developmental Biology, University College London

[*]robert.bentham.11@ucl.ac.uk

2016-08-18

Abstract

MCbiclust is a R package for running massively correlating biclustering analysis. MCbiclust aims to find large scale biclusters with selected features being highly correlated with each other over a subset of samples. MCbiclust was particularly designed for the application of studying gene expression data, finding and understanding biclusters that are related to large scale co-regulation of genes.

Package

MCbiclust 0.1.0

Report issues on https://github.com/rbentham/MCbiclust/issues

Contents

1 Getting started

Once installed MCbiclust can be loaded with the following command:

```
library(MCbiclust)
```

MCbiclust also makes sure that the packages BiocParallel, cluster, stats, GGally, ggplot2 and scales are all installed. It is also advised that the packages ggplot2 and gplots are separately installed and loaded.

```
library(ggplot2)
library(gplots)
```

2 Example of a single run

2.1 Loading Cancer Cell Line Encyclopedia (CCLE) and MitoCarta data sets

For this example analysis we will be seeking to find biclusters related to mitochondrial function in the cancer cell line encyclopedia. For this two datasets are needed, both of which are available on the MCbiclust package. The first in `CCLE_data` that contains the gene expression values found in the CCLE data set, the second, `Mitochondrial_genes`, is a list of mitochondrial genes that can be found from MitoCarta1.0.

```
data(CCLE_data)
data(Mitochondrial_genes)
```

It is a simple procedure to create a new matrix `CCLE.mito` only containing the mitochondrial genes. While there are 1023 known mitochondrial genes, not all of these are measured in `CCLE_data`.

```
mito.loc <- which(as.character(CCLE_data[,2]) %in% Mitochondrial_genes)
CCLE.mito <- CCLE_data[mito.loc,-c(1,2)]
row.names(CCLE.mito) <- CCLE_data[mito.loc,2]
```

2.2 Finding a bicluster seed

The first step in using MCbiclust is to find a subset of samples that have the most highly correlating genes in the chosen gene expression matrix. This is done by, calculating the associated correlation matrix and then calculating the absolute mean of the correlations, as a correlation score.

This is achieved with function `FindSeed`, the argument `gem` stands for gene expression matrix, `seed.size` indicates the size of the subset of samples that is sought. `iterations` indicates how many iterations of the algorithm to carry out before stopping, in general the higher the iterations the more optimal the solution in terms of maximising the strength of the correlation.

For reproducibility `set.seed` has been used to set R's pseudo-random number generator. It should also be noted that the for `gem` the data matrix can not contain all the genes, since `FindSeed` involves the calculation of correlation matrices which are not computationally efficient to compute if they involve greater than ~1000 genes.

```
set.seed(102)
CCLE.seed <- FindSeed(gem = CCLE.mito,
                      seed.size = 10,
                      iterations = 10000)
```

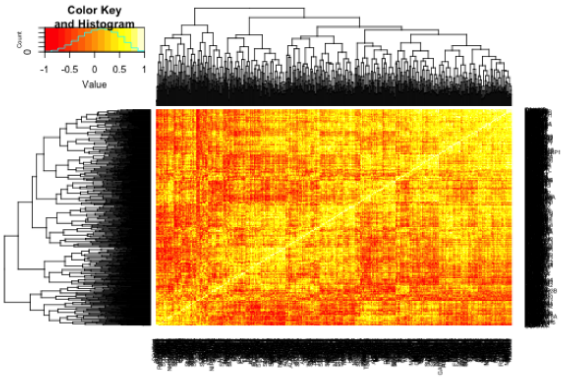
`FindSeed` has one more additional options, `initial.seed` allows the user to specify the initial subsample to be tested, by default the initial sample subset is randomly chosen.

There is a function `CorScoreCalc` that can calculate the correlation score used in FindSeed, though in general you should not need to use it, unless you wish to manually check the chosen seed is an improvement on one that is randomly generated.

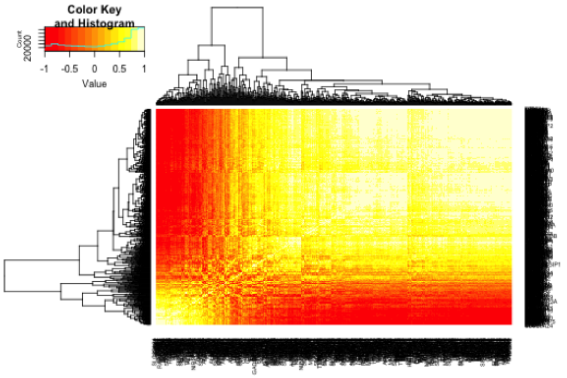
```
set.seed(103)
random.seed <- sample(seq(length = dim(CCLE.mito)[2]), 10)
CorScoreCalc(CCLE.mito,random.seed)
## [1] 0.3058819
CorScoreCalc(CCLE.mito,CCLE.seed)
## [1] 0.6190576
```

The results of `FindSeed` can also be visualised by examining the associated correlation matrix, and viewing the result as a heatmap. Again it is easy to see the difference between the random subsample and the one outputted from `FindSeed`.

```
CCLE.random.cor <- cor(t(CCLE.mito[,random.seed]))
heatmap.2(CCLE.random.cor,trace = "none")
```




```
CCLE.mito.cor <- cor(t(CCLE.mito[,CCLE.seed]))
heatmap.2(CCLE.mito.cor,trace = "none")
```



Note that when the genes are represented as the rows in a matrix, that matrix needs to be transposed before the calculation of the correlation matrix.

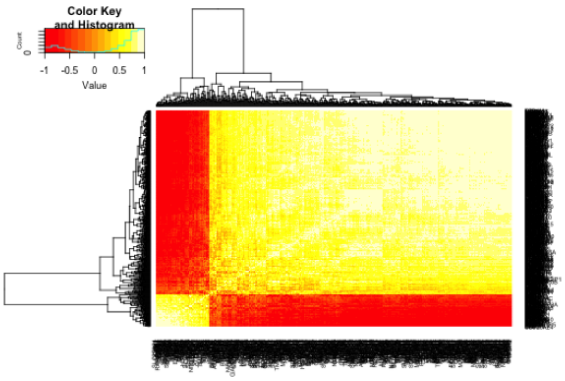
`heatmap.2` is a function from the `gplots` R package.

2.3 Selecting highly correlated genes

As can be clearly seen from the heat map, not all the mitochondrial genes are equally strongly correlated to each other. There is a function in `MCbiclust` which automatically selects those genes that are most strongly associated with the pattern. This function is `HclustGenesHiCor` and it works by using hierarchical clustering to select the genes into n different groups, and then discarding any of these groups

that fails to have a correlation score greater than the correlation score from all the genes together.

```
CCLE.hicor.genes <- as.numeric(HclustGenesHiCor(CCLE.mito,
                                                CCLE.seed,
                                                cuts = 8))
CCLE.mito.cor2 <- cor(t(CCLE.mito[CCLE.hicor.genes, CCLE.seed]))
CCLE.heat <- heatmap.2(CCLE.mito.cor2,trace = "none")
```



There are two groups of genes, strongly correlated to themselves and anti-correlated to each other.

```
CCLE.groups <- list(labels(CCLE.heat$rowDendrogram[[1]]),
                    labels(CCLE.heat$rowDendrogram[[2]]))
```

2.4 Calculate the correlation vector

As was seen in the last section, a distinct correlation pattern was found. However this was only examined for genes involved in mitochondrial function. Non-mitochondrial genes are likely also involved in this pattern and it is important to identify them.

All genes can be measured by how they match to this pattern in two steps. The first step is to summarise this pattern. This is done by finding a subset of genes which all strongly correlate with each other, and calculating their average expression value. The function `GeneVecFun` achieves this step. Similarly to `HclustGenesHiCor` the genes are clustered into groups using hierarchical clustering, but the best group is judged by the correlation score multiplied by the square root of the number of genes. This is done to bias against selecting a group of very small genes.

The second function `CalcCorVector` calculates the correlation vector by calculating the correlation of the average expression value found in the first step to every gene measured in the data set. This value is called the correlation vector.

```
CCLE.cor.vec <- CVEval(gem.part = CCLE.mito,
                       gem.all = CCLE_data[,-c(1,2)],
                       seed = CCLE.seed, splits = 10)
```

2.5 Gene Set Enrichment

Using the calculated correlation vector, it is a relatively simple task to perform gene set enrichment. This can be done on any platform (e.g. DAVID, gprofiler, etc.) but MCbiclust comes with an inbuilt function for calculating GO enrichment values using the Mann-Whitney non-parametric test.

```
GSE.MW <- GOEnrichmentAnalysis(gene.names = as.character(CCLE_data[,2]),
                               gene.values = CCLE.cor.vec,
                               sig.rate = 0.05)
```

There are 1082 significant terms and the top 10 most significant can be viewed below:

```
pander::pandoc.table(GSE.MW[1:10,],row.names = FALSE)
```

Table 1: Table continues below

	GOID	TERM	num.genes
1404	GO:0006396	RNA processing	993
14993	GO:0003723	RNA binding	1808
13638	GO:0012505	endomembrane system	4489
13788	GO:0030529	ribonucleoprotein complex	744
2489	GO:0009653	anatomical structure morphogenesis	3188
14696	GO:0098588	bounding membrane of organelle	2558
17948	GO:0044822	poly(A) RNA binding	1170
3308	GO:0016071	mRNA metabolic process	885
13965	GO:0031981	nuclear lumen	2785
13496	GO:0005794	Golgi apparatus	1552

	g.in.genelist	adj.p.value	g.av.value
1404	629	6.343e-84	0.7072
14993	1346	3.887e-70	0.5299
13638	3002	2.523e-64	0.02952
13788	528	5.976e-63	0.6651
2489	2184	4.844e-60	0.01128
14696	1891	1.365e-58	-0.005724
17948	1019	1.763e-55	0.5275
3308	537	1.814e-54	0.6566
13965	1990	2.273e-53	0.4681
13496	1173	1.978e-52	-0.07116

2.6 Sample ordering

Already all the genes in the data set have had the correlation calculated to the pattern found. One more task that can be readily done is to order the samples according to the strength of correlation. Function `FindSeed` found the initial n samples that had a very strong correlation with the gene set of interest, the $n+1$ sample is to be selected as that sample which best maintains the correlation strength, this process can be simply repeated until all or the desired number of samples are ordered.

`SampleSort` is the function in `MCbiclust` that completes this procedure, it has 4 main inputs, the first is the gene expression matrix with all the samples and the gene set of interest. `seed` is the initial subsample found with `FindSeed`. For increasing what can be a very slow computation, the code can be run on multiple cores, with the number of cores selected from the argument `num.cores` and instead of sorting the entire length, only the first `sort.length` samples need to be ordered.

```
CCLE.samp.sort <- SampleSort(CCLE.mito[as.numeric(CCLE.hicor.genes),],
                             seed = CCLE.seed, num.cores = 4)
```

2.7 PCA and ggplot2

Once the samples have been sorted it is possible to summarise the correlation pattern found using principal component analysis (PCA).

PCA is a method of dimensional reduction, and converts a data set to a new set of variables known as the principal components. These are designed to be completely uncorrelated or orthogonal to each other. In this way the principal components are new variables that capture the correlations between the old variables, and are in fact a linear combination of the old variables. The first principal component (PC1) is calculated as the one that explains the highest variance within the data, the second than is that which has the highest variance but is completely uncorrelated or orthogonal to the previous principal component. In this way additional principal components are calculated until all the variance in the data set is explained.

PC1 captures the highest variance within the data, so if PCA is run on the found bicluster with very strong correlations between the genes, PC1 will be a variable that summarises this correlation.

`PC1VecFun` is a function that calculates the PC1 values for all sorted samples. It takes three inputs: 1.`top.gem` is the gene expression matrix with only the most highly correlated genes but with all the sample data. 1. `seed.sort` is the sorting of the data samples found with function `SampleSort` 1. `n` is the number of samples used for initially calculating the weighting of PC1. If set to 10, the first 10 samples are used to calculate the weighting of PC1 and then the value of PC1 is calculated for all samples in the ordering.

```
top.mat <- CCLE.mito[as.numeric(CCLE.hicor.genes),]
pc1.vec <- PC1VecFun(top.gem = top.mat,
                     seed.sort = CCLE.samp.sort, n = 10)

CCLE.bic <- ThresholdBic(cor.vec = CCLE.cor.vec,
                         sort.order = CCLE.samp.sort,
                         pc1 = pc1.vec,samp.sig = 0.05)

pc1.vec <- PC1Align(gem = CCLE.data[,-c(1,2)], PC1 = pc1.vec,
                    CV = CCLE.cor.vec, bic = CCLE.bic)
```

As an alternative to calculating PC1, the user may want to calculate the average expression value of certain gene sets. This gives a better idea of the type of regulation occurring in the correlation pattern, as an abstract notion of a principal component does not have to be understood.

Either the two groups of mitochondrial genes strongly correlated to themselves and anti-correlated to each other could be used, or the top n correlating and anti-correlating genes in the correlation vector, or those genes in the correlation vector above a particular threshold.

```
av.genes.group1 <- colMeans(CCLE.mito[CCLE.groups[[1]],
                                      CCLE.samp.sort])
av.genes.group2 <- colMeans(CCLE.mito[CCLE.groups[[2]],
                                      CCLE.samp.sort])

top.1000 <- order(CCLE.cor.vec, decreasing = T)[seq(1000)]
bottom.1000 <- order(CCLE.cor.vec, decreasing = F)[seq(1000)]

greater.0.9 <- which(CCLE.cor.vec > 0.9)
less.m0.9 <- which(CCLE.cor.vec < -0.9)

av.genes.top1 <- colMeans(CCLE_data[top.1000, CCLE.samp.sort])
av.genes.top2 <- colMeans(CCLE_data[bottom.1000, CCLE.samp.sort])

av.genes.thrhld1 <- colMeans(CCLE_data[greater.0.9, CCLE.samp.sort])
av.genes.thrhld2 <- colMeans(CCLE_data[less.m0.9, CCLE.samp.sort])
```

Once the samples have been ordered and PC1 and the average gene sets calculated it is a simple procedure to produce plots of these against the ordered samples.

This is done here using the `ggplot2` package, a plotting package for R by Hadley Wickam.

```
CCLE.names <- colnames(CCLE_data)[-c(1,2)][CCLE.samp.sort]
CCLE.df <- data.frame(CCLE.name = CCLE.names,
                      PC1 = pc1.vec,
                      Average.Group1 = av.genes.group1,
                      Average.Group2 = av.genes.group2,
                      Order = seq(length = length(pc1.vec)))

ggplot(CCLE.df, aes(Order,PC1)) +
  geom_point() + ylab("PC1")

ggplot(CCLE.df, aes(Order,Average.Group1)) +
  geom_point() + ylab("Average Group 1")

ggplot(CCLE.df, aes(Order,Average.Group2)) +
  geom_point() + ylab("Average Group 2")
```

2.8 Thresholding the bicluster

It is natural to classify the samples into different groups based on the value of PC1 or the average expression of the gene sets above. With a strong correlation pattern bicluster plotting PC1 against the sorted pattern results in a 'fork' like pattern as seen above. Those samples on the upper fork hence receive the label "Upper Fork" samples while those on the lower fork are known as "Lower Fork" samples.

Given a numeric vector of PC1 values, the samples can be classified as "Upper" or "Lower" using the `ForkClassifier` function. This function has two inputs, the PC1 numeric vector and how many of the first samples do you wish to classify.

```
fork.class <- ForkClassifier(pc1.vec,100)
fork.status <- rep("Normal",length(pc1.vec))
fork.status[fork.class$Upper] <- "Upper"
fork.status[fork.class$Lower] <- "Lower"
```

This fork information can now be added to the plot.

```
CCLE.df <- data.frame(CCLE.name = CCLE.names,
                      PC1 = pc1.vec,
                      Order = seq(length = length(pc1.vec)),
                      Fork = fork.status)

ggplot(CCLE.df, aes(Order,PC1)) +
```

```
  geom_point(aes(colour=factor(Fork))) + ylab("PC1")
```

This is still however not very enlightening, and in the next sections additional information will be added to these plots.

2.9 Comparing results with sample data

This section will deal with two addition data sets both of which are available in the MCbiclust package.

1. CCLE sample information, a data set containing information for every sample in the data set, including gender of the patient the cell line was derived from, as well as the primary site it came from.

This section is meant as an example of the type of analysis that can be done with additional data set. Each new data set may have different additional data available with it and may be in formats that need some extra work to become compatible with the results from the `MCbiclust` analysis.

2.9.1 CCLE sample data

This data set is available within the `MCbiclust` package.

```
data(CCLE_samples)
```

The first step is to compare the column names of both data sets and to make sure we are dealing with the same correctly labeled samples.

```
CCLE.samples.names <- as.character(CCLE_samples[,1])
CCLE.data.names <- colnames(CCLE_data)[-c(1,2)]
CCLE.samples.names[seq(10)]
CCLE.data.names[seq(10)]
```

In this case some samples have an additional "X" not present in some CCLE_samples data so it is necessary to add it for consistency.

```
CCLE.samples.names[c(1:15)] <- paste("X",CCLE.samples.names[c(1:15)],
                                     sep="")
CCLE_samples$CCLE.name <- CCLE.samples.names
```

Using the `dplyr` library, it is possible to join this new data set to the one we made for plotting the values of PC1 in the previous section. This can be easily done as both datasets share a column - the name of the samples. Once this is done, it is again simple to produce additional plots.

```
library(dplyr)
CCLE.df.samples <- inner_join(CCLE.df,CCLE_samples,by="CCLE.name")

ggplot(CCLE.df.samples, aes(Order,PC1)) +
  geom_point(aes(colour=factor(Site.Primary))) + ylab("PC1")

ggplot(CCLE.df.samples, aes(Order,PC1)) +
  geom_point(aes(colour=factor(Gender))) + ylab("PC1")
```

Since in this case the data is categorical, it can be tested for significance using Pearson's chi squared test.

```
library(MASS)

# create contingency tables
ctable.site <- table(CCLE.df.samples$Fork,
                     CCLE.df.samples$Site.Primary,
                     exclude = c("autonomic_ganglia",
                                 "biliary_tract",
                                 "liver",
                                 "oesophagus",
                                 "pancreas",
                                 "prostate",
                                 "salivary_gland",
                                 "small_intestine",
                                 "stomach",
                                 "thyroid"))

ctable.gender <- table(CCLE.df.samples$Fork,
                       CCLE.df.samples$Gender,
                       exclude = "U")

chisq.test(ctable.site)
chisq.test(ctable.gender)
```

As was easily apparent from examining the plots, the primary site the cell line is derived from is highly significant, while gender is not.

Next instead of categorical data, numerical data will be examined.

3 Dealing with multiple runs

`MCbiclust` is a stochastic method so for best results it needs to be run multiple times, in practice this means using high-performance computing the run the algorithm on a computer cluster which will be dealt with in a later section. Here however the task of dealing with the results will be looked at. The algorithm will be run 100

times with only 500 iterations each. Typically more iterations are required, but for this demonstration it will be sufficient.

```
CCLE.multi.seed <- list()
initial.seed1 <- list()

for(i in seq(100)){
  set.seed(i)
  initial.seed1[[i]] <- sample(seq(length = dim(CCLE.mito)[2]),10)
  CCLE.multi.seed[[i]] <- FindSeed(gem = CCLE.mito,
                                   seed.size = 10,
                                   iterations = 500,
                                   initial.seed = initial.seed1[[i]])
}
```

The associated correlation vector must also be calculated for each run and these correlation vectors can be put into a matrix.

```
CCLE.cor.vec.multi <- list()

for(i in seq(100)){
  CCLE.gene.vec <- GeneVecFun(CCLE.mito, CCLE.multi.seed[[i]], 10)
  CCLE.gem1 <- CCLE_data[,-c(1,2)][,CCLE.multi.seed[[i]]]
  CCLE.cor.vec.multi[[i]] <- CalcCorVector(gene.vec = CCLE.gene.vec,
                                           gem = CCLE.gem1)
}

len.a <- length(CCLE.cor.vec.multi[[1]])
len.b <- length(CCLE.cor.vec.multi)
multi.run.cor.vec.mat <- matrix(0,len.a,len.b)
for(i in 1:100){
  multi.run.cor.vec.mat[,i] <- CCLE.cor.vec.multi[[i]]
}
rm(CCLE.cor.vec.multi)
```

A correlation matrix can be formed from the correlation vectors, and in this way they can be viewed as a heatmap.

```
CV.cor.mat1 <- abs(cor((multi.run.cor.vec.mat)))
cor.dist <- function(c){as.dist(1 - abs(c))}

routput.corvec.matrix.cor.heat <- heatmap.2(CV.cor.mat1,
                                            trace="none",
                                            distfun = cor.dist)
```

It needs to be known how many distinct patterns have been found, this is done with clustering and particular silhouette coefficients to judge what number of clusters is optimum within the data. Function `SilhouetteClustGroups` achieves this and uses hierarchical clustering to split the patterns into clusters, for comparison a

randomly generated correlation vector is also added to allow for the possibility that all patterns found are best grouped into a single cluster.

```
multi.clust.groups <- SilhouetteClustGroups(multi.run.cor.vec.mat,
                                            max.clusters = 20,
                                            plots = T)
```

Here only one cluster was found, and we can visualise this pattern (and any additional others found) with the function `CVPlot`, which highlights a chosen gene set, in this case the mitochondrial genes.

```
microarray.genes <- as.character(CCLE_data[,2])
av.corvec.fun <- function(x) rowMeans(multi.run.cor.vec.mat[,x]
average.corvec <- lapply(X = multi.clust.groups,
                         FUN = av.corvec.fun)

CVPlot(cv.df = as.data.frame(average.corvec),
        geneset.loc = mito.loc,
        geneset.name = "Mitochondrial",
        alpha1 = 0.1)
```

As before can also calculate the gene set enrichment.

```
GOfun <- function(x) GOEnrichmentAnalysis(gene.names = microarray.genes,
                                          gene.values = x,
                                          sig.rate = 0.05))


corvec.gsea <- lapply(X = average.corvec,
                      FUN = GOfun)
```

Instead of using `SampleSort` a special sorting function for this case is used, `Multi SampleSort`. This works very similarly to `SampleSort` but selects the top genes in the average correlation vector and selects the initial seed with the highest correlation score corresponding to those genes.

```
CCLE.samp.multi.sort <-  MultiSampleSort(CCLE_data[,-c(1,2)],
                                         average.corvec, 750,
                                         multi.clust.groups,
                                         CCLE.multi.seed,2,50)
```

Session Info

```
devtools::session_info()
## Session info -------------------------------------------------------
##  setting  value
##  version  R version 3.3.1 (2016-06-21)
##  system   x86_64, darwin13.4.0
##  ui       X11
##  language (EN)
##  collate  en_GB.UTF-8
##  tz       Europe/London
##  date     2016-08-18
## Packages -----------------------------------------------------------
##  package      * version date       source
##  BiocParallel * 1.6.6   2016-08-15 Bioconductor
##  BiocStyle    * 2.0.3   2016-08-04 Bioconductor
##  bitops         1.0-6   2013-08-17 CRAN (R 3.3.0)
##  caTools        1.17.1  2014-09-10 CRAN (R 3.3.0)
##  cluster      * 2.0.4   2016-04-18 CRAN (R 3.3.1)
##  codetools      0.2-14  2015-07-15 CRAN (R 3.3.1)
##  colorspace     1.2-6   2015-03-11 CRAN (R 3.3.0)
##  devtools       1.12.0  2016-06-24 CRAN (R 3.3.0)
##  digest         0.6.10  2016-08-02 CRAN (R 3.3.0)
##  evaluate       0.9     2016-04-29 CRAN (R 3.3.0)
##  formatR        1.4     2016-05-09 CRAN (R 3.3.0)
##  gdata          2.17.0  2015-07-04 CRAN (R 3.3.0)
##  GGally       * 1.2.0   2016-07-01 CRAN (R 3.3.0)
##  ggplot2      * 2.1.0   2016-03-01 CRAN (R 3.3.0)
##  gplots       * 3.0.1   2016-03-30 CRAN (R 3.3.0)
##  gtable         0.2.0   2016-02-26 CRAN (R 3.3.0)
##  gtools         3.5.0   2015-05-29 CRAN (R 3.3.0)
##  htmltools      0.3.5   2016-03-21 CRAN (R 3.3.0)
##  KernSmooth     2.23-15 2015-06-29 CRAN (R 3.3.1)
##  knitr          1.14    2016-08-13 CRAN (R 3.3.0)
##  magrittr       1.5     2014-11-22 CRAN (R 3.3.0)
##  MCbiclust    * 0.1.0   2016-08-18 local
##  memoise        1.0.0   2016-01-29 CRAN (R 3.3.0)
##  munsell        0.4.3   2016-02-13 CRAN (R 3.3.0)
##  pander         0.6.0   2015-11-23 CRAN (R 3.3.0)
##  plyr           1.8.4   2016-06-08 CRAN (R 3.3.0)
##  Rcpp           0.12.6  2016-07-19 CRAN (R 3.3.0)
##  reshape        0.8.5   2014-04-23 CRAN (R 3.3.0)
##  rmarkdown      1.0     2016-07-08 CRAN (R 3.3.0)
##  scales       * 0.4.0   2016-02-26 CRAN (R 3.3.0)
##  stringi        1.1.1   2016-05-27 CRAN (R 3.3.0)
##  stringr        1.0.0   2015-04-30 CRAN (R 3.3.0)
##  withr          1.0.2   2016-06-20 CRAN (R 3.3.0)
```

```
##  yaml          2.1.13  2014-06-12 CRAN (R 3.3.0)
```