MBASIC: Matrix Based Analysis for State-space Inference and Clustering

Chandler Zuo and Sündüz Keleş
Departments of Statistics and of Biostatistics and Medical Informatics, University of Wisconsin-Madison

Contents

1 Introduction

This document provides an introduction to the power analysis of ChIP-seq data with the *MBASIC* package (**MBASIC** which stands for **M**atrix **B**ased **A**nalysis for **S**tate-space **I**nference and **C**lustering) [1]. *MBASIC* provides a Bayesian framework for clustering units based on their infered states over a set of experimental conditions.

MBASIC is especially useful for integrative analysis for ChIP-seq experiments. In this case, a set of prespecified loci is clustered based on their activities over celltypes and transcription factors. We build a pipeline in the *MBASIC* package and will focus on this pipeline in this vignette. We will introduce some advanced options in building the **MBASIC** model in Section 4.

2 MBASIC Pipeline for Sequencing Data

2.1 Overview

Applying the **MBASIC** framework to analyzing ChIP-seq data sets includes three steps:

1. *Matching each ChIP replicate data set with their input data set:* This step matches ChIP replicate files with their matching input files;
2. *Calculating mapped counts and the covariate on the target loci:* This step calculates the mapped counts from each ChIP and input replicate files on each of the target locus;

3. *Fitting MBASIC model:* This step fits the MBASIC model, using either an E-M algorithm or a MAD-Bayes algorithm, to identify the binding states for each locus and cluster the loci based on their binding states across different conditions.

MBASIC integrates Step 2-3 in a single function `MBASIC.pipeline`. For Step 1 *MBASIC* provides a function `ChIPInputMatch` that assists matching the ChIP files with input files based on the naming convention of the ENCODE datasets. We have found that in practice, more often than not, some violations to the ENCODE file name conventions always occur, and manual adjustments to the results of our automated matching are inevitable. Therefore, we do not integrate this function in `MBASIC.pipeline`.

2.2 Step 1: Match ChIP and Input Datasets

To illustrate Step 1 we first generate a set of synthetic data. *MBASIC* package provides a function `generateSyntheticData` to assist our demo. This function generates synthetic BED data for ChIP and input samples, as well as mappability and GC scores in a directory specified by the 'dir' argument. It also generates a target set of loci for our analysis. By default, the number of loci is 100, each with size 20 bp. All data are generated across 5 chromosomes, each with size 10K bp. ChIP data are from 2 celltypes, and for each celltype there are K=5 TFs. Under each condition randomly 1-3 replicates for the ChIP data are generated. All ChIP data from the same celltype are matched to the same set of 3 input replicates.

```
library(MBASIC)
```

```
target <- generateSyntheticData(dir = "syntheticData")
head(target)

## RangedData with 6 rows and 0 value columns across 5 spaces
##        space        ranges |
##     <factor>     <IRanges> |
## 1       chr1 [9760, 9780] |
## 2       chr1 [1920, 1940] |
## 3       chr1 [7000, 7020] |
## 4       chr1 [3560, 3580] |
## 5       chr1 [8760, 8780] |
## 6       chr1 [7120, 7140] |

system("ls syntheticData/*/*", intern = TRUE)[1:5]

## [1] "syntheticData/chip/wgEncodeLabExpCell1Fac1CtrlAlnRep1.bed"
## [2] "syntheticData/chip/wgEncodeLabExpCell1Fac2CtrlAlnRep1.bed"
## [3] "syntheticData/chip/wgEncodeLabExpCell1Fac3CtrlAlnRep1.bed"
## [4] "syntheticData/chip/wgEncodeLabExpCell1Fac3CtrlAlnRep2.bed"
## [5] "syntheticData/chip/wgEncodeLabExpCell1Fac3CtrlAlnRep3.bed"
```

We have developed the `ChIPInputMatch` function to help match the ChIP and input data files. In the following example, we use this function to read all files with suffices ".bed" and ".bam" in directories specified by the argument "dir", and matches the files assuming ENCODE naming convention. It looks up files up to the number of levels of subdirectories specified by "depth". The output of this function contains multiple columns. The first column contains the file name for each ChIP replicate. The second column is the initial string for the matching input replicates, because for each ChIP replicate there are possibly multiple input replicates. The rest of the columns contains information for lab, experiment identifier, factor and control identifier. This information is parsed from the file names.

We acknowledge that the current function may not parse all data file names correctly. For practical data files, users are suggested to check its result, and manual corrections may be needed.

```
tbl <- ChIPInputMatch(dir = paste("syntheticData/",
    c("chip", "input"), sep = ""), celltypes = c("Cell1",
    "Cell2"), suffices = c(".bam", ".bed"), depth = 5)
head(tbl)

##                                                           chipfile
## 1 syntheticData/chip/wgEncodeLabExpCell1Fac1CtrlAlnRep1.bed
## 2 syntheticData/chip/wgEncodeLabExpCell1Fac2CtrlAlnRep1.bed
## 3 syntheticData/chip/wgEncodeLabExpCell1Fac3CtrlAlnRep1.bed
## 4 syntheticData/chip/wgEncodeLabExpCell1Fac3CtrlAlnRep2.bed
## 5 syntheticData/chip/wgEncodeLabExpCell1Fac3CtrlAlnRep3.bed
## 6 syntheticData/chip/wgEncodeLabExpCell1Fac4CtrlAlnRep1.bed
##                                           inputfile
## 1 syntheticData/input/wgEncodeLabExpCell1InputCtrl
## 2 syntheticData/input/wgEncodeLabExpCell1InputCtrl
## 3 syntheticData/input/wgEncodeLabExpCell1InputCtrl
## 4 syntheticData/input/wgEncodeLabExpCell1InputCtrl
## 5 syntheticData/input/wgEncodeLabExpCell1InputCtrl
## 6 syntheticData/input/wgEncodeLabExpCell1InputCtrl
##   lab experiment  cell factor control chipformat
## 1 Lab        Exp Cell1   Fac1    Ctrl        BED
## 2 Lab        Exp Cell1   Fac2    Ctrl        BED
## 3 Lab        Exp Cell1   Fac3    Ctrl        BED
## 4 Lab        Exp Cell1   Fac3    Ctrl        BED
## 5 Lab        Exp Cell1   Fac3    Ctrl        BED
## 6 Lab        Exp Cell1   Fac4    Ctrl        BED
##   inputformat
## 1         BED
## 2         BED
## 3         BED
## 4         BED
## 5         BED
## 6         BED
```

We also need to specify the experimental condition for each data set by a vector 'conds'.

```
conds <- paste(tbl$cell, tbl$factor, sep = ".")
```

2.3 Executing Step 2 and 3 Altogether

Now we are in a position to continue the next steps in the pipeline. There are two ways to execute these steps: (1) use function `MBASIC.pipeline` to execute the two steps together; or (2) execute each step separately. The following code calls the function `MBASIC.pipeline`:

```
## remove file 'data.Rda' since it will be generated
## try(file.remove('data.Rda'))
fit.mbasic <- MBASIC.pipeline(chipfile = tbl$chipfile,
    inputfile = tbl$inputfile, input.suffix = ".bed",
    target = target, chipformat = tbl$chipformat, inputformat = tbl$inputformat,
    fragLen = 150, pairedEnd = FALSE, unique = TRUE,
    fac = conds, J = 3, S = 2, family = "negbin", datafile = "data.Rda")
class(fit.mbasic)
```

We list the meanings of the arguments of `MBASIC.pipeline` in Table 1. The above example fits a clustering model with 3 clusters. It returns an object of class *MBASICFit*. This class is described in more details in Section 3.

When we specify 'method="mbasic"', we can fit models with varying numbers of clusters simultaneously using the MBASIC algorithm, and pick the one with the minimum BIC value:

```
allfits.mbasic <- MBASIC.pipeline(chipfile = tbl$chipfile,
    inputfile = tbl$inputfile, input.suffix = ".bed",
    target = target, chipformat = tbl$chipformat, inputformat = tbl$inputformat,
    fragLen = 150, pairedEnd = FALSE, unique = TRUE,
    fac = conds, J = 3:10, S = 2, family = "negbin",
    datafile = "data.Rda", method = "mbasic")
names(allfits.mbasic)

## [1] "allFits" "BestFit" "Time"

class(allfits.mbasic$BestFit)

## [1] "MBASICFit"
## attr(,"package")
## [1] "MBASIC"
```

We can also invoke the MAD-Bayes algorithm, a K-means-like algorithm, by specifying 'method="madbayes"'. Notice that in this case, we do not need specify the number of clusters ('J') and the distribution family ('family'):

```
allfits.madbayes <- MBASIC.pipeline(chipfile = tbl$chipfile,
    inputfile = tbl$inputfile, input.suffix = ".bed",
    target = target, chipformat = tbl$chipformat, inputformat = tbl$inputformat,
    fragLen = 150, pairedEnd = FALSE, unique = TRUE,
    fac = conds, J = 3:10, S = 2, datafile = "data.Rda",
    method = "madbayes")
names(allfits.madbayes)

## [1] "allFits"     "BestFit"     "BestFit.bic"
## [4] "Iter"        "Loss"        "lambda"
## [7] "Time"        "InitLoss"

class(allfits.madbayes$BestFit)

## [1] "MBASICFit"
## attr(,"package")
## [1] "MBASIC"
```

Before we move on to describe the step-wise execution, we highlight the usage of the argument 'datafile'. In the above codes, when we compute 'fit.mbasic', the file 'datafile' is not generated yet, so we process ChIP and input data and save the processed data in 'datafile'. When we compute 'allfits.mbasic', `MBASIC.allfit` detects that 'datafile' already exists, so it automatically loads the preprocessed data and skip the step of processing the ChIP and input data. This can save substantially amount of time if the size of our data is large.

2.4 Step 2: Generate the Data Matrices

We can execute Step 2 and 3 separately. In Step 2, we use the function `generateReadMatrices` to calculate the ChIP count at each locus for each ChIP replicate. We also calculate the count at each locus for each matching input.

```
## Step 2: Generate mapped count matrices
dat <- generateReadMatrices(chipfile = tbl$chipfile,
    inputfile = tbl$inputfile, input.suffix = ".bed",
    target = target, chipformat = tbl$chipformat, inputformat = tbl$inputformat,
    fragLen = 150, pairedEnd = FALSE, unique = TRUE)
```

```
conds <- paste(tbl$cell, tbl$factor, sep = ".")
```

We can directly use the matching input counts as the covariate data in our model. Advanced users of our package may want to normalize the input counts according to the mappability and GC scores and use the normalized counts as the covariate. In that case, we need call functions `averageMGC` and `bkng_mean`.

```
## Step 2': calculate the mappability and GC-content
## scores for each locus
target <- averageMGC(target = target, m.prefix = "syntheticData/mgc/",
    m.suffix = "_M.txt", gc.prefix = "syntheticData/mgc/",
    gc.suffix = "_GC.txt")
## Step 2': compute the normalized input counts
dat$input1 <- bkng_mean(inputdat = dat$input, target = target,
    family = "negbin")
```

Notice that such a normalization step is automatically executed if users specify the 'm.prefix', 'm.suffix', 'gc.prefix' and 'gc.suffix' arguments in `MASIC.pipeline`.

2.5 Step 3: Build the MBASIC Model

When using the MBASIC's E-M algorithm, we can either fit one MBASIC model using function `MBASIC`, or simultaneously fit models with different numbers of clusters by using function `MBASIC.full`:

```
## Step 3: Fit an MBASIC model
fit.mbasic <- MBASIC(Y = t(dat$chip), Gamma = t(dat$input),
    S = 2, fac = conds, J = 3, maxitr = 10, family = "negbin")
## Step 3: Fit multiple MBASIC models simultaneously
allfits.mbasic <- MBASIC.full(Y = t(dat$chip), Gamma = t(dat$input),
    S = 2, fac = conds, J = 3:10, maxitr = 10, family = "negbin",
    ncores = 10)
fit.mbasic <- allfits.mbasic$BestFit
```

We can also invoke the MAD-Bayes algorithm in this step by using function `MBASIC.MADBayes.full`:

```
allfits.madbayes <- MBASIC.MADBayes.full(Y = log(t(dat$chip) +
    1), Gamma = log(t(dat$input) + 1), S = 2, fac = conds,
    maxitr = 10, ncores = 10, nlambdas = 10, nfits = 1)
fit.madbayes <- allfits.madbayes$BestFit
```

3 Interpreting the Results

The slot “BestFit” in the return value of `MBASIC.full`, `MBASIC.MADBayes.full`, and `MBASIC.pipeline` are of the S-4 class *MBASICFit*.

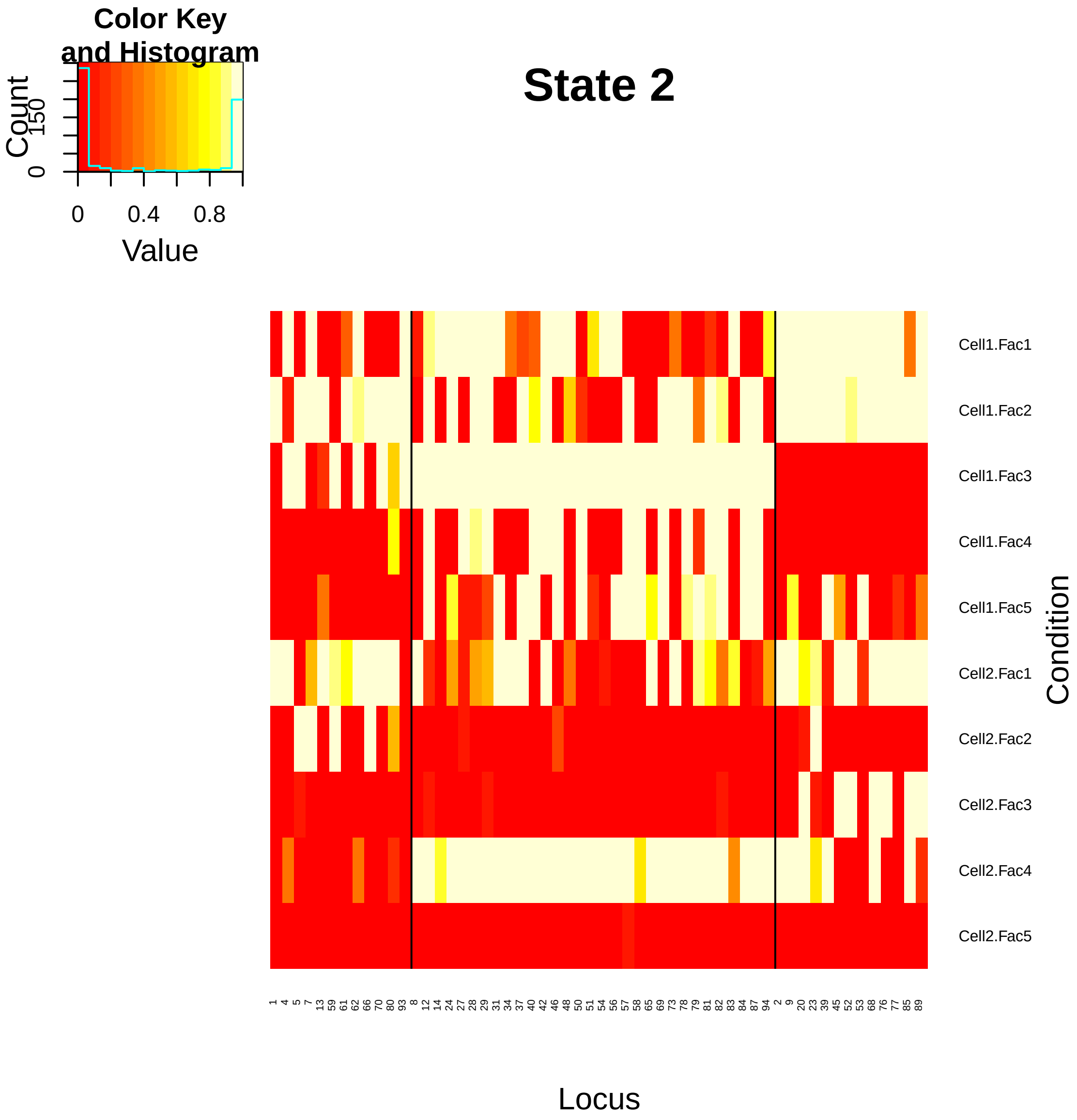
```
showClass("MBASICFit")
```

Slot 'Theta' records the probabilities for each locus to have each state under each condition. For a model with K experiment conditions, S states and I units, slot 'Theta' is a matrix of dimension KS by I, and the (K(s-1)+k, i)-th entry is the probability for the i-th locus to have state s under condition k.

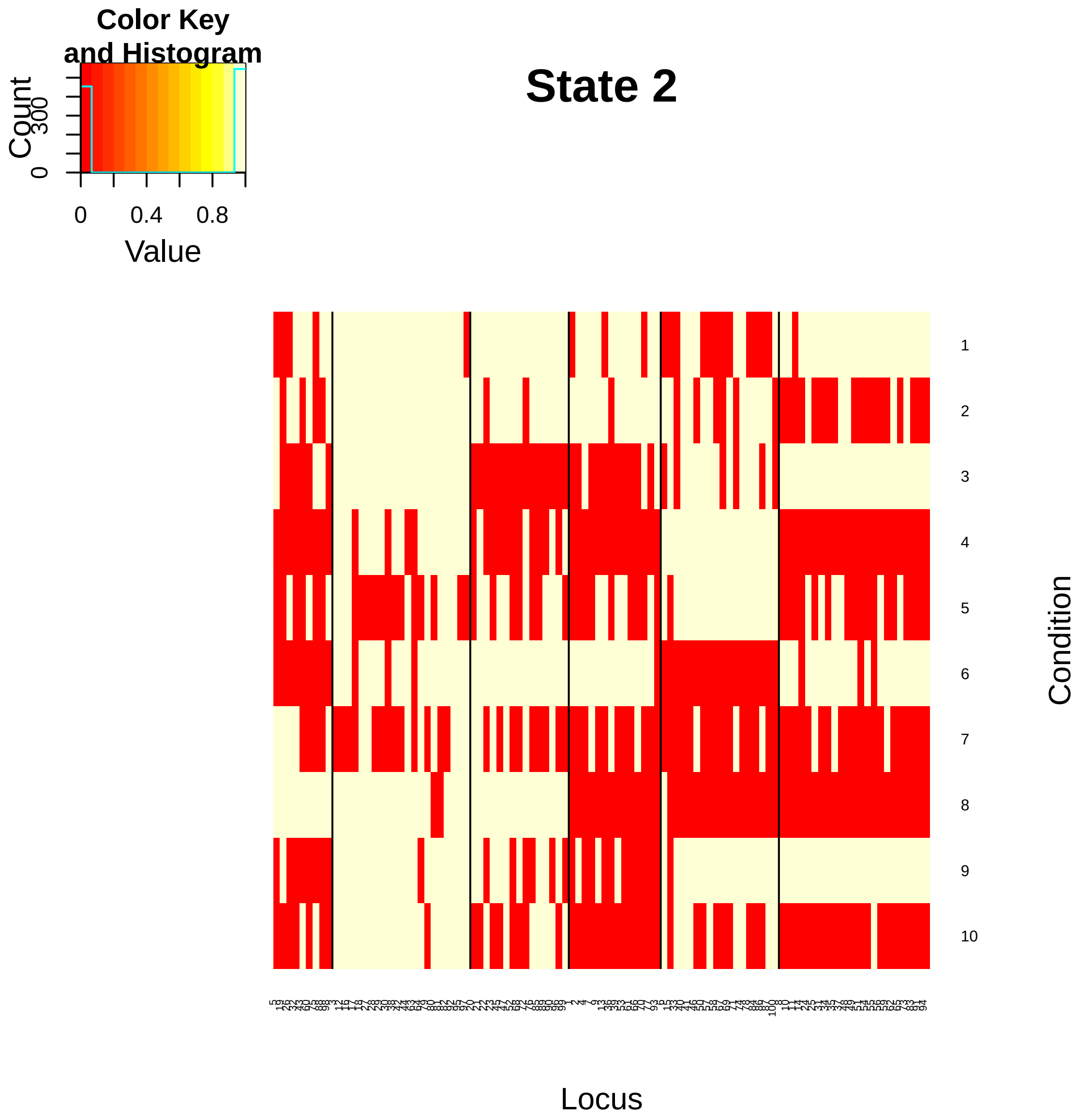
```
dim(fit.mbasic@Theta)

## [1]  20 100

rownames(fit.mbasic@Theta)

##  [1] "Cell1.Fac1" "Cell1.Fac2" "Cell1.Fac3"
##  [4] "Cell1.Fac4" "Cell1.Fac5" "Cell2.Fac1"
##  [7] "Cell2.Fac2" "Cell2.Fac3" "Cell2.Fac4"
## [10] "Cell2.Fac5" "Cell1.Fac1" "Cell1.Fac2"
## [13] "Cell1.Fac3" "Cell1.Fac4" "Cell1.Fac5"
## [16] "Cell2.Fac1" "Cell2.Fac2" "Cell2.Fac3"
## [19] "Cell2.Fac4" "Cell2.Fac5"

head(fit.mbasic@Theta[1, ])

## [1] 0.9991287360 0.0008872948 0.9839761153
## [4] 0.0151795339 0.9994389041 0.9162819515
```

In our example, suppose state 1 corresponds to the un-enriched state, and state 2 the enriched state. We can use the function `plot` to draw a heatmap to visualize the enrichment states across all data.

```
plot(fit.mbasic, slot = "Theta", xlab = "Locus", state = 2,
    cexRow = 0.6, cexCol = 0.4)
```



```
plot(fit.madbayes, slot = "Theta", xlab = "Locus",
    state = 2, cexRow = 0.6, cexCol = 0.4)
```



Slot 'clustProb' is a matrix for the posterior probablity of each locus to belong to each cluster. The first column contains the probability for each locus to be a singleton (i.e. not belong to any clusters). The (j+1)-th column contains the probabilities for each locus to be in cluster j.

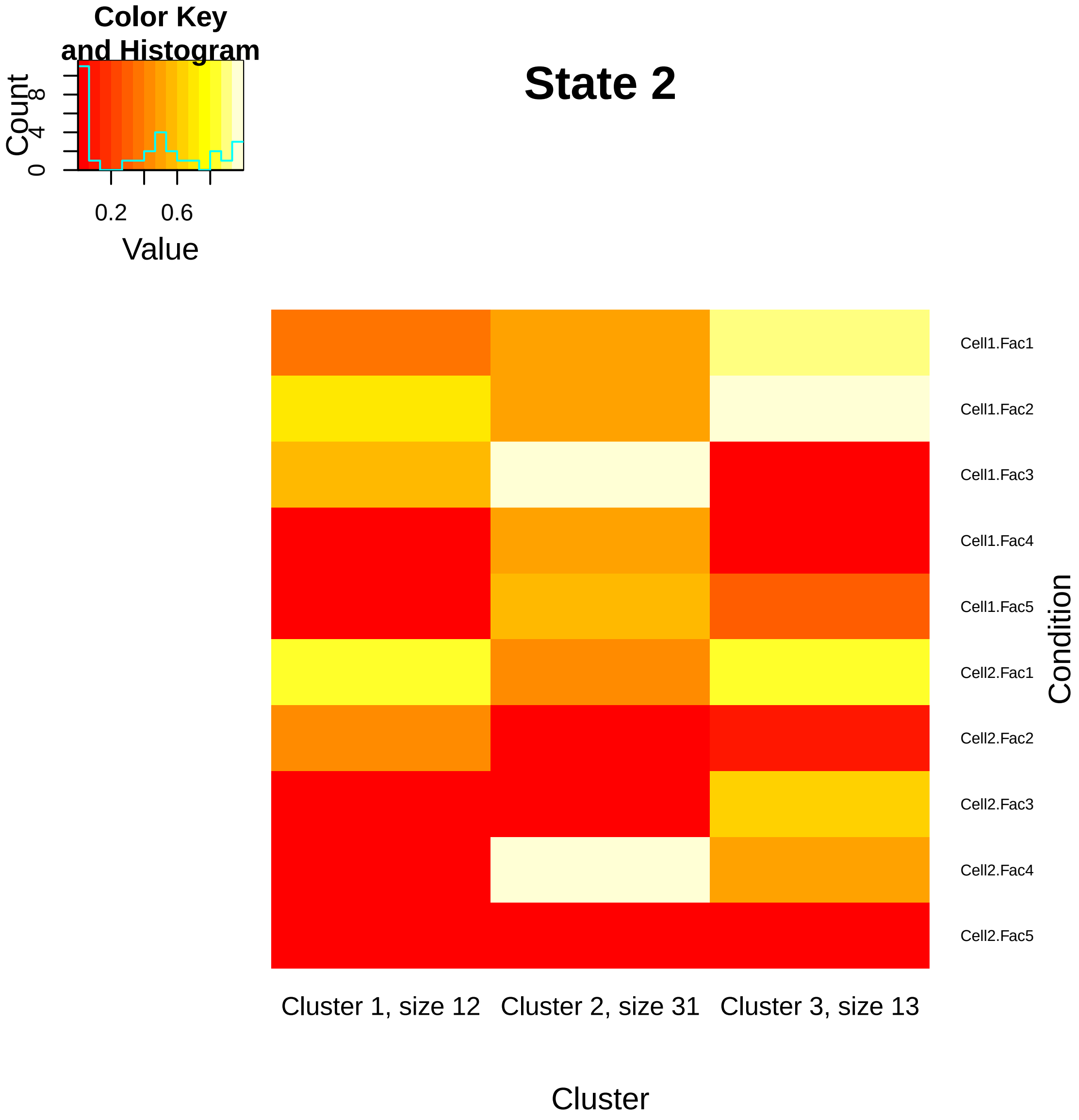
```
dim(fit.mbasic@clustProb)

## [1] 100   4

round(head(fit.mbasic@clustProb), 3)

##      b.prob
## [1,]  0.279 0.636 0.000 0.085
## [2,]  0.041 0.002 0.014 0.943
## [3,]  0.931 0.000 0.067 0.002
## [4,]  0.209 0.443 0.347 0.000
## [5,]  0.407 0.593 0.000 0.000
## [6,]  0.942 0.000 0.001 0.058

clusterLabels <- apply(fit.mbasic@clustProb, 1, which.max) -
    1
table(clusterLabels)

## clusterLabels
##  0  1  2  3
## 44 12 31 13
```

Slot 'W' is a matrix for the state-space profiles for all clusters. For a model with K conditions, S states and J clusters, this matrix has dimension KS by J, where the (k+K(s-1),j)-th entry contains the probability that a unit in cluster j has state s under condition k. For our example, the enriched probability for all clusters is in rows 11-20.

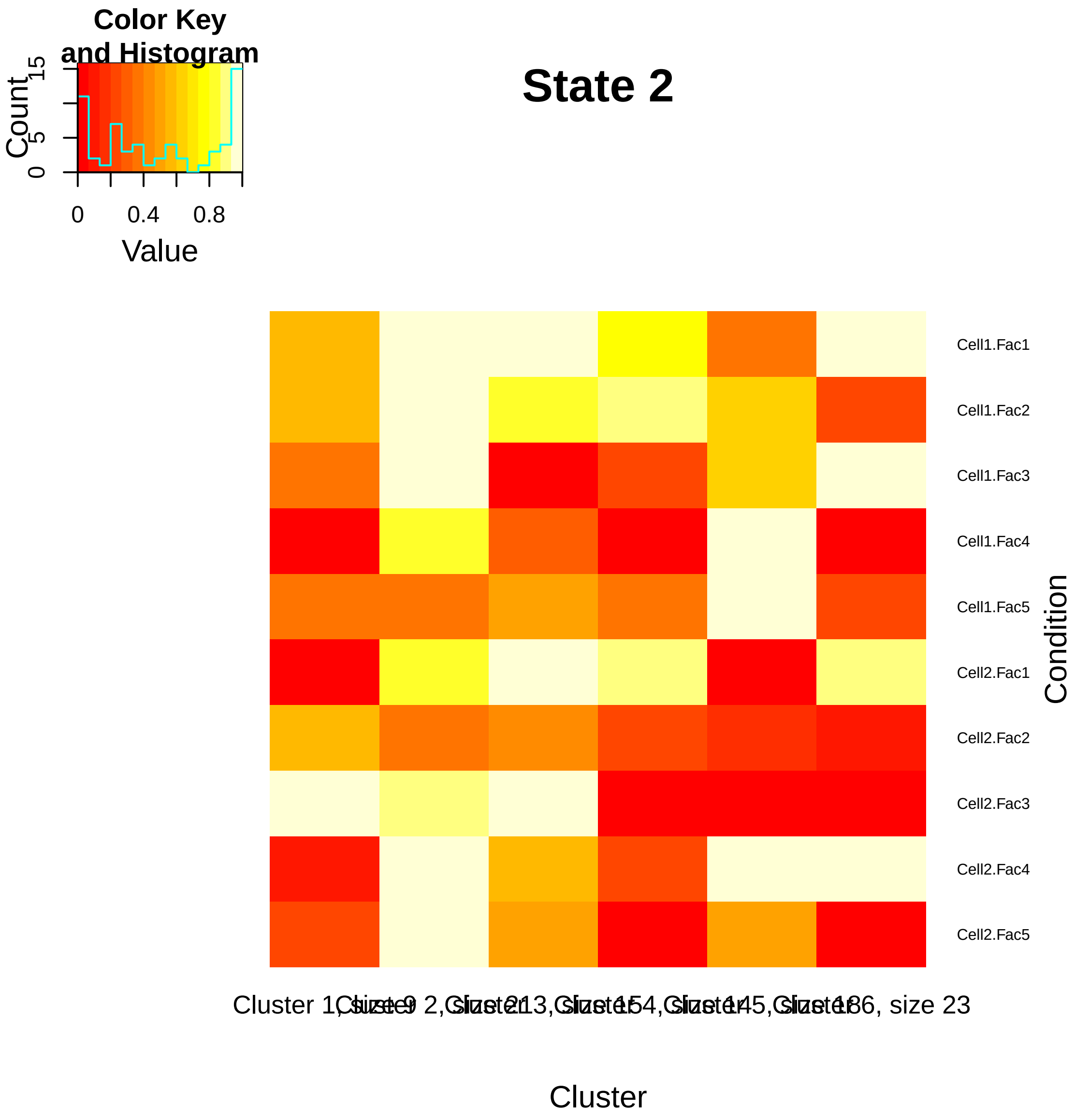
```
rownames(fit.mbasic@W)

##  [1] "Cell1.Fac1" "Cell1.Fac2" "Cell1.Fac3"
##  [4] "Cell1.Fac4" "Cell1.Fac5" "Cell2.Fac1"
##  [7] "Cell2.Fac2" "Cell2.Fac3" "Cell2.Fac4"
## [10] "Cell2.Fac5" "Cell1.Fac1" "Cell1.Fac2"
## [13] "Cell1.Fac3" "Cell1.Fac4" "Cell1.Fac5"
## [16] "Cell2.Fac1" "Cell2.Fac2" "Cell2.Fac3"
## [19] "Cell2.Fac4" "Cell2.Fac5"

dim(fit.mbasic@W)

## [1] 20  3

round(head(fit.mbasic@W[seq(10) + 10, ]), 3)

##             [,1]  [,2]  [,3]
## Cell1.Fac1 0.353 0.517 0.926
## Cell1.Fac2 0.694 0.527 0.979
## Cell1.Fac3 0.599 0.979 0.001
## Cell1.Fac4 0.065 0.517 0.000
## Cell1.Fac5 0.001 0.577 0.294
## Cell2.Fac1 0.824 0.404 0.820
```

We can also use the `plot` function to visualize the probability for each cluster to have a particular state under all conditions.

```
plot(fit.mbasic, slot = "W", state = 2, cexRow = 0.6,
    cexCol = 1, srtCol = 0, adjCol = c(0.5, 1))
```



```
plot(fit.madbayes, slot = "W", state = 2, cexRow = 0.6,
    cexCol = 1, srtCol = 0, adjCol = c(0.5, 1))
```

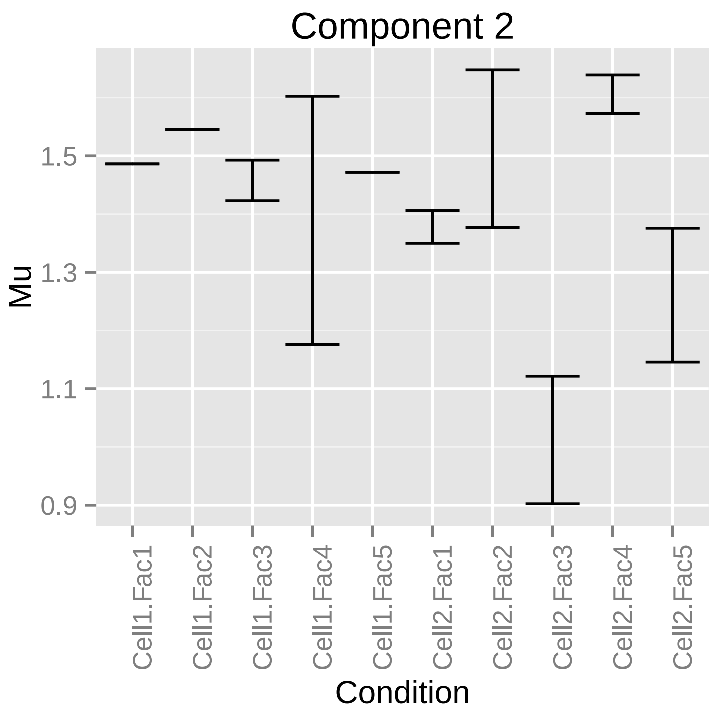


Slot 'Mu' and 'Sigma' are matrices for the distribution parameters for each replicate and each component (usually, one component is one state, see Section 4.1).

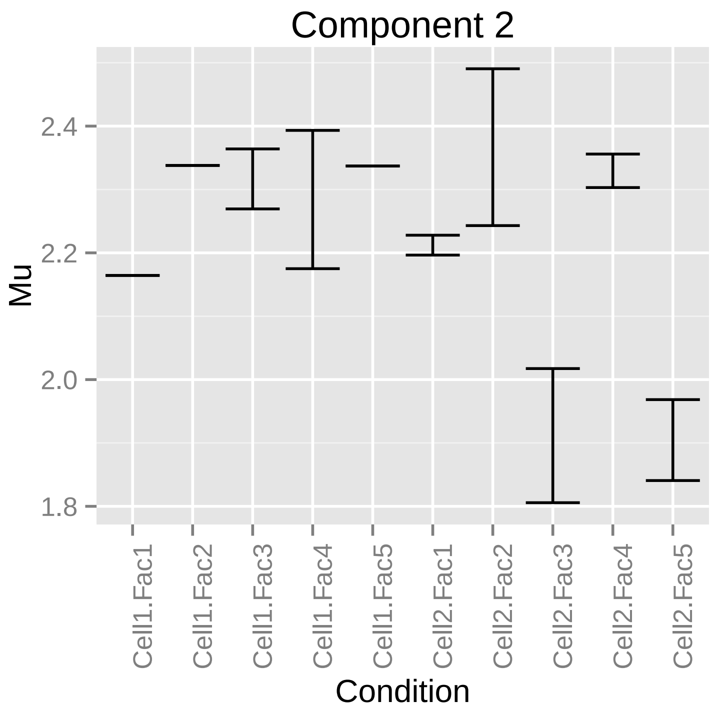
```
dim(fit.mbasic@Mu)

## [1] 21  2

dim(fit.mbasic@Sigma)

## [1] 21  2
```

Function `plot` enables visualizing the range of the fitted parameter values across different replicates for the same condition. Notice that for these slots the function is implemented using *ggplot2*, so additional arguments can be passed by '+':

```
plot(fit.mbasic, slot = "Mu", state = 2) + theme(axis.text.x = element_text(angle = 90,
    hjust = 1))
```



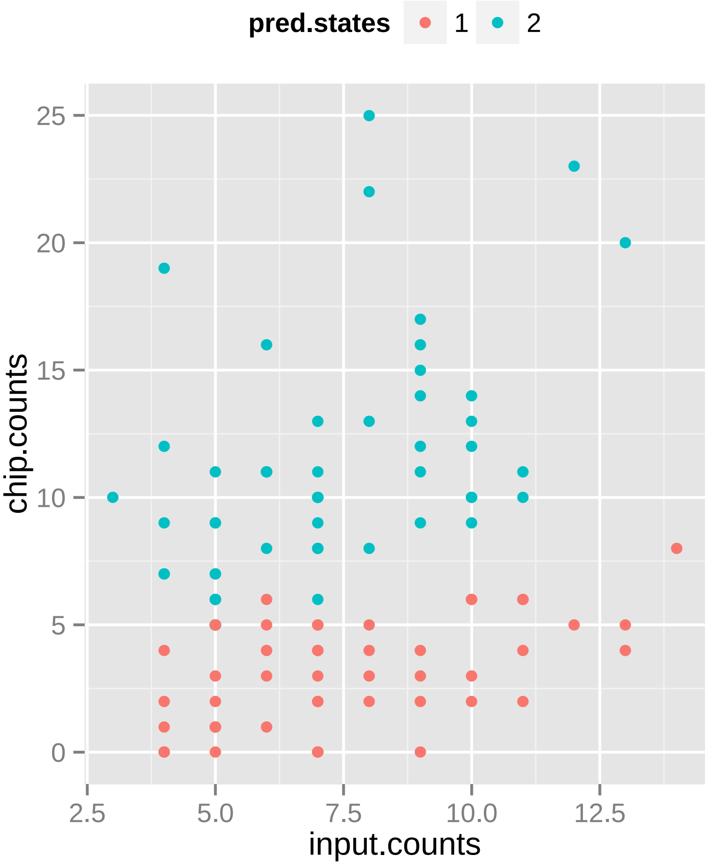
```
plot(fit.madbayes, slot = "Mu", state = 2) + theme(axis.text.x = element_text(angle = 90,
    hjust = 1))
```



Besides calling the `plot` function on our fitted object, we can also use `ggplot` to visualize how the loci within a single replicate are allocated to different states:

```
## which replicate to plot
repid <- 1
chip.counts <- dat$chip[, repid]
```

```
input.counts <- dat$input[, repid]
pred.states <- as.character(apply(fit.mbasic@Theta[rownames(fit.mbasic@Theta) ==
    conds[repid], ], 2, which.max))
ggplot() + geom_point(aes(x = input.counts, y = chip.counts,
    color = pred.states)) + theme(legend.position = "top")
```



4 Advanced Functions

4.1 Multiple Components in One State

MBASIC model assumes that the distribution of each data is a mixture of several components, and the components are mapped to different states. In the simpliest cases, each component corresponds to a distinct state. In some cases, we might want to include multiple components in one state. For example, for ChIP-seq data, we may want to include two components for the enriched state to capture both the weakly enriched and strongly enriched loci. This can be done by specifying the 'statemap' value. In the following example, we assume there are three components for data set, the first component corresponds to the un-enriched state, the second and the third components both belong to the enriched states.

```
fit.mix <- MBASIC(Y = t(dat$chip), Gamma = t(dat$input),
    S = 2, fac = conds, J = 3, maxitr = 10, family = "negbin",
    statemap = c(1, 2, 2))
```

4.2 Minimum Enrichment Setting for Negative Binomial Distributions

When using `MBASIC` to analyze data with the negative binomial distribution, the 'min.count' can be used to specify the minimum data threshold for enriched components. For ChIP-seq data, this can be used as a threshold for the enrichment detection. 'min.count' accepts four formats: a scalar, a vector with the same length as the number of components, a vector with the same length as the number of replicates, and a matrix.

We recommend that users specify different thresholds for individual replicates that account for their sequencing depth difference. In this case, we should let 'min.count' be a vector the same length as the number of replicates, and it specifies the threshold for each replicate for all components other than 1. The following example sets the threshold based on the 25% percentiles for the matching input of each replicate. Notice that the output of `generateReadMatrices` contains the sequencing depth information ("dat$depth") that is used here to scale the ChIP samples against their matching inputs:

```
mincount.thresholds <- apply(dat$input, 2, function(x) quantile(x,
    0.25)) * apply(dat$depth, 1, function(x) x[1]/x[2])
mincount.thresholds <- as.integer(mincount.thresholds)
mincount.thresholds[mincount.thresholds < 5] <- 5
summary(mincount.thresholds)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       5       5       5       5       5       5

fit.threshold1 <- MBASIC(Y = t(dat$chip), Gamma = t(dat$input),
    S = 2, fac = conds, J = 3, maxitr = 10, family = "negbin",
    statemap = c(1, 2, 2), min.count = mincount.thresholds)
```

For the scalar value, 'min.count' specifies the threshold for all units with components other than 1. In the previous example, if we want to add a restriction that all loci with states other than 1 must have values at least 5, we can use:

```
fit.threshold2 <- MBASIC(Y = t(dat$chip), Gamma = t(dat$input),
    S = 2, fac = conds, J = 3, maxitr = 10, family = "negbin",
    statemap = c(1, 2, 2), min.count = 5)
```

If 'min.count' is a vector the same length as the number of components, it specifies the threshold for each component for all the replicates:

```
fit.threshold3 <- MBASIC(Y = t(dat$chip), Gamma = t(dat$input),
    S = 2, fac = conds, J = 3, maxitr = 10, family = "negbin",
    statemap = c(1, 2, 2), min.count = c(0, 5, 10))
```

Finally, we can specify the thresholds for every component and every replicate using a matrix input. We recommend setting the first column of this matrix as all 0s unless there are specific reasons to do otherwise:

```
mincount.mat <- cbind(0, mincount.thresholds, 2 * mincount.thresholds)
fit.threshold4 <- MBASIC(Y = t(dat$chip), Gamma = t(dat$input),
    S = 2, fac = conds, J = 3, maxitr = 10, family = "negbin",
    statemap = c(1, 2, 2), min.count = mincount.mat)
```

4.3 Model Initialization

Function `MBASIC` accepts initial values for a number of parameters. This is especially useful when an initial run of `MBASIC` reaches the maximum number of iterations, but the algorithm is not converged. In this case, we can rerun the `MBASIC` using the previously fitted model parameters to initialize the model.

```
fit.update <- MBASIC(Y = t(dat$chip), Gamma = t(dat$input),
    S = 2, fac = conds, J = 3, maxitr = 10, family = "negbin",
    initial = fit.mbasic)
```

4.4 Analyze the Allele Specific Binding Data

The *MBASIC* can be used to analyze allele-specific binding data. For such data, we need prepare two data matrices for the read counts for all loci across all replicates. One matrix should be the counts from either the paternal or the maternal allele; the other matrix should be the sum of maternal and paternal allele counts. Calculating allele-specific counts requires either remapping the sequencing reads to the non-reference genome or first matching to the reference genome then analyzing the mismatch information. *MBASIC* does not implement such procedures, and require the users to provide the processed data matrices.

We can use the `MBASIC.sim` function (see Section 4.5) to generate a synthetic data set for allele-specific counts:

```
dat.asb <- MBASIC.sim(xi = 10, family = "binom", I = 1000,
    fac = rep(seq(10), each = 2), J = 3, S = 3, zeta = 0.1)
## The counts from either the maternal or the
## paternal allele
dim(dat.asb$Y)

## [1]   20 1000

## The total number of counts from both alleles
dim(dat.asb$X)

## [1]   20 1000
```

We can fit MBASIC models on such data with either mixtures of binomial distributions or gamma-binomial distributions (i.e. binomial distributions with Gamma priors). Notice that the total count matrix is passed through the 'Gamma' parameter:

```
## Using binomial distributions
fit.asb.bin <- MBASIC(Y = dat.asb$Y, Gamma = dat.asb$X,
    S = 3, fac = dat.asb$fac, J = 3, maxitr = 5, para = dat.asb,
    family = "binom")
## Using gamma-binomial distributions
fit.asb.gb <- MBASIC(Y = dat.asb$Y, Gamma = dat.asb$X,
    S = 3, fac = dat.asb$fac, J = 3, maxitr = 5, para = dat.asb,
    family = "gamma-binom")
```

4.5 Simulation

The *MBASIC* package also provides functions to simulate and fit general MBASIC models. Function `MBASIC.sim` simulates data with 'I' units and 'J' clusters. The 'S' argument specifies the number of different states, and 'zeta' is the proportion of singleton. 'fac' specifies the condition for each experiment. The 'xi' argument relates to the magnitude of the simulated data. For detailed description users are recommended to read our manual.

```
## Simulate data across I=1000 units with J=3
## clusters There are S=3 states
dat.sim <- MBASIC.sim(xi = 2, family = "lognormal",
    I = 1000, fac = rep(1:10, each = 2), J = 3, S = 3,
    zeta = 0.1)
```

`MBASIC.sim` returns a list object. The 'Y' field contains the simulated data matrix at each unit (column) for each experiment (row). The 'Theta' field is the matrix for the states for each unit (column) and each experimental condition (column). The 'W' field is a matrix with dimensions KS × J, where the (S(k-1)+s,j)-th entry is the probability that units in the j-th cluster have state s under the k-th condition.

```
names(dat.sim)

##  [1] "Theta"       "Y"           "X"
##  [4] "fac"         "W"           "Z"
##  [7] "V"           "delta"       "zeta"
## [10] "prior.mean" "prior.sd"   "stdev"
## [13] "Mu"         "bkng"       "snr"
## [16] "non.id"

dim(dat.sim$Y)

## [1]   20 1000

dim(dat.sim$W)

## [1] 30  3

dim(dat.sim$Theta)

## [1]   10 1000
```

We can apply MBASIC to this simulated data. If we pass the simulated data to the function through the 'para' argument, we can get the following slots on the estimation error:

- *ARI*: Adjusted Rand Index;
- *W.err*: The mean squared error in matrix W;
- *Theta.err*: The mean squared error in state estimation;
- *MisClassRate*: The mis-classification rate.

```
dat.sim.fit <- MBASIC(Y = dat.sim$Y, S = 3, fac = dat.sim$fac,
    J = 3, maxitr = 3, para = dat.sim, family = "lognormal")
```

```
dat.sim.fit@ARI

## [1] 0.9379704

dat.sim.fit@W.err

## [1] 0.07783956

dat.sim.fit@Theta.err

## [1] 0.1968652

dat.sim.fit@MisClassRate

## [1] 0.04869996
```

4.6 Degenerate MBASIC Models

In a degenerate MBASIC model, the states for each unit under each condition are directly observed. `MBASIC.sim.state` and `MBASIC.state` functions allows users to simulate and fit such models. The usage of these functions are similar to functions `MBASIC.sim` and `MBASIC`.

`MBASIC.sim.state` simulates data from a degenerate MBASIC model. Different from `MBASIC.sim`, `MBASIC.sim.state` does not need arguments 'fac' and 'family', but it needs the 'K' argument, specifying the number of experimental conditions.

```
state.sim <- MBASIC.sim.state(I = 1000, K = 10, J = 4,
    S = 3, zeta = 0.1)
```

`MBASIC.state` fits a degenerate MBASIC model. Different to function `MBASIC`, it does not need arguments 'Y' and 'family'. Instead, it needs the argument 'Theta' to pass the observed states.

```
state.sim.fit <- MBASIC.state(Theta = state.sim$Theta,
    J = 4, zeta = 0.1)
```

5 Session Information

```
## R version 3.1.1 (2014-07-10)
## Platform: x86_64-redhat-linux-gnu (64-bit)
##
## locale:
##  [1] LC_CTYPE=zh_TW.UTF-8
##  [2] LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8
##  [4] LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8
##  [6] LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8
##  [8] LC_NAME=C
##  [9] LC_ADDRESS=C
## [10] LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8
## [12] LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel  stats     graphics  grDevices
## [5] utils     datasets  methods   base
##
## other attached packages:
##  [1] MBASIC_0.99.0        Rcpp_0.11.4
##  [3] msm_1.5              mclust_4.4
##  [5] MASS_7.3-33          gtools_3.4.1
##  [7] gplots_2.16.0        ggplot2_1.0.0
##  [9] GenomicRanges_1.14.4 XVector_0.2.0
## [11] IRanges_1.20.7       BiocGenerics_0.8.0
## [13] doParallel_1.0.8     iterators_1.0.7
## [15] foreach_1.4.2        cluster_1.15.2
##
## loaded via a namespace (and not attached):
##  [1] BiocStyle_1.0.0    bitops_1.0-6
##  [3] caTools_1.17.1     codetools_0.2-8
##  [5] colorspace_1.2-4   compiler_3.1.1
##  [7] digest_0.6.8       evaluate_0.5.5
##  [9] expm_0.99-1.1      formatR_1.0
## [11] gdata_2.13.3       grid_3.1.1
## [13] gtable_0.1.2       highr_0.4
## [15] KernSmooth_2.23-12 knitr_1.9
## [17] labeling_0.3       lattice_0.20-29
## [19] Matrix_1.1-5       munsell_0.4.2
## [21] mvtnorm_1.0-2      plyr_1.8.1
## [23] proto_0.3-10       reshape2_1.4.1
## [25] scales_0.2.4       splines_3.1.1
## [27] stats4_3.1.1       stringr_0.6.2
## [29] survival_2.37-7    tools_3.1.1
```

References

[1] C. Zuo, Kyle Hewitt, Emery Bresnick, and S. Keleş. A hierarchical framework for state-space inference and clustering. Submitted, May 2015.

Table 1: Arguments for the `MBASIC.pipeline` function. For a comprehensive list of arguments and their details, users are recommended to read our manual.

Data Sources	
chipfile	A string vector for the ChIP files.
inputfile	A string vector for the matching input files. The length must be the same as "chipfile".
input.suffix	A string for the suffix of input files. If NULL, "inputfile" will be treated as the full names of the input files. Otherwise, all inputfiles with the initial "inputfile" and this suffix will be merged.
chipformat (inputformat)	A string specifying the type of all ChIP (input) files, or a vector of string specifying the types of each ChIP (input) file. Currently three file types are allowed: "BAM", "BED" or "TAGALIGN" ("TAGALIGN" and "BED" files are treated as same). Default: "BAM".
m.prefix (optional)	A string for the prefix of the mappability files. Default: NULL.
m.suffix (optional)	A string for the suffix of the mappability files. See our man files for more details. Default: NULL.
gc.prefix (optional)	A string for the prefix of the GC files. Default: NULL.
gc.suffix (optional)	A string for the suffix of the GC files. See our man files for more details. Default: NULL.
Genomic Information	
target	A RangedData object for the target intervals where the reads are mapped.
fragLen	Either a single value or a 2-column matrix of the fragment lengths, the first column for the different fragment lengths for each file in 'chipfile', and the second column for the fragment lengths for each file in 'inputfile'. Default: 150.
pairedEnd	Either a boolean value or a 2-column boolean matrix for whether each file is a paired-end data set. Currently this function only allows "BAM" files for paired-end data. Default: FALSE.
unique	A boolean value for whether only reads with distinct genomic coordinates or strands are mapped. Default: TRUE.
Model Parameters	
S	The number of states.
fac	A vector of length N for the experimental condition of each ChIP replicate.
J	A single number or a numeric vector of the numbers of clusters to be included in the model.
family	The distribution of family to be used. *MBASIC* currently support five distribution types: 'lognormal', 'negbin', 'binom', 'scaled-t', 'gamma-binom'. See our man files for more information.
Tuning Parameters	
maxitr	The maximum number of iterations in the E-M algorithm. Default: 100.
tol	Tolerance for the relative increment in the log-likelihood function to check the E-M algorithm's convergence. Default: 1e-10.
tol.par	Tolerance for the maximum relative change in parameters to check the algorithm's convergence. Default: 1e-5.
datafile	The location to save the count matrices, or load pre-computed count matrices.