CSSP: ChIP-Seq Statistical Power

Chandler Zuo*
Sündüz Keleş†

Contents

1 Introduction

This document provides an introduction to the power analysis of ChIP-seq data with the `CSSP` package. This `R` package implements **CSSP** which stands for **C**hIP-**s**eq **S**tatistical **P**ower and is developed in [**?**]. **CSSP** provides a Bayesian framework for modeling bin-level read counts in a ChIP-Seq experiment as Poisson processes. These bin-level local Poisson models utilize genome sequence features such as mappability and GC-content.

In this vignette, we utilize an artificially constructed ChIP-seq data-set with 5 chromosomes contained in the package. Another example using a real data-set from a mouse ChIP-seq experiment of GATA1 in the G1E-ER4+E2 cell line ([**?**]) is shown on the package website `http://code.google.com/p/chip-seq/power`.

2 Workflow

The power analysis by `CSSP` consist of two major steps:

1. *Fitting the CSSP model:* This step estimates parameters of the local Poisson models for the ChIP and input samples.

2. *Calculating power:* This step implements power calculations based on estimated parameters for user specified FDR (q), minimum fold change (r), and minimum ChIP read count thresholds (τ).

2.1 Constructing the Bin-level Data in `R` Environment

`CSSP` requires four input files for model fitting: bin-level ChIP data, bin-level input data, bin-level mappability score, and bin-level GC content score. For human genome (hg18) and mouse genome (mm9), the mappability and GC content scores are provided by the `mosaics` package ([**?**]). The corresponding files for other genomes can be requested via `http://code.google.com/p/chip-seq-power`. We will be uploading these files for other genomes, read lengths, and bin sizes as the demand arises.

There are two ways to prepare the bin-level data-set. The first method is to use the "constructBins" function from the `mosaics` package. This function accommodates commonly used read alignment formats including ELAND, BED, BOWTIE and SAM and computes bin-level data for user specified bin size and average fragment length. It outputs the bin-level counting data in seperate text files, which can be loaded by either the "readBins" function in `mosaics` or the "readBinFile" function in our package. Further details on the input arguments for the "constructBins" and "readBins" functions are available through `mosaics` vignette at Bioconductor. Notice that "readBins" returns an object of S-4 class "BinData", while "readBinFile" returns a "data.frame". Both data types are accepted for the down-stream analysis.

Here we show an example of importing text files through the "readBinFile" function, which uses a data.frame object in this package "bindata.chr1" containing bin-level data for chip, input, M and GC data. For illustration purposes, we break the data.frame into different text files and read them back. In the "readBinFile" function, argument "type" specifies the data types of the files to be imported. The actual file names are specified by the vector "fileName". The vectors in both arguments should have the same length and ordering. The "readBinFile" function constructs a "data.frame" object with columns named "chip", "input", "M" and "GC".

```
> library( CSSP )
> data( bindata.chr1 )
> #break the data.frame into different files
```

*Department of Statistics, 1300 University Avenue, Madison, WI, 53706, USA.
†Departments of Statistics and of Biostatistics and Medical Informatics, 1300 University Avenue, Madison, WI, 53706, USA.

```
> write.table( bindata.chr1[,c(1,4)], file = "chr1_map.txt", sep = "\t",
+             row.names = FALSE, col.names = FALSE )
> write.table( bindata.chr1[,c(1,5)], file = "chr1_gc.txt", sep = "\t",
+              row.names = FALSE, col.names = FALSE )
> write.table( bindata.chr1[,c(1,2)], file = "chr1_chip.txt", sep = "\t",
+              row.names = FALSE, col.names = FALSE )
> write.table( bindata.chr1[,c(1,3)], file = "chr1_input.txt", sep = "\t",
+              row.names = FALSE, col.names = FALSE )
> #read the text files and construct a unified data.frame
> dat <- readBinFile ( type = c("chip", "input","M","GC"),
+                     fileName = c("chr1_chip.txt", "chr1_input.txt",
+                       "chr1_map.txt", "chr1_gc.txt" ) )
> file.remove( paste( "chr1_", c("chip", "input", "map", "gc" ), ".txt", sep = "" ) )

[1] TRUE TRUE TRUE TRUE

> head( dat )

  pos input chip         M        GC
1   0    76  110 0.9464916 0.9464916
2 100    89  144 0.9372329 0.9372329
3 200    85  123 0.9406308 0.9406308
4 300   104  115 0.9154102 0.9154102
5 400    96  117 0.9152835 0.9152835
6 500    81  118 0.9418648 0.9418648
```

The second method to prepare the bin-level data-set is to use functions provided by the SPP package ([**?**]), followed by the "tag2bin" and "createBinData" in our package. SPP provides a list of functions "read.xxx.tags" that accomodates eland, bowtie, bam, tagalign formats and convert them to a list of signed genome coordinates for each chromosome. The sign of each coordinate represent the strand information for each read, with positive numbers representing the 5' strand and negative numbers representing the 3' strand. The output of "read.xxx.tags" can be imported by "tag2bin" function through the "tagdat" argument. By accomodating the output format of the SPP package, we thus enable using the functionalities in SPP for preprocessing and filtering raw alignment reads before generating bin-level data.

The output of the "tag2bin" function is a list of bin-level counts for each chromosome. After the outputs for both ChIP and input samples are generated, they can be merged with the external text files containing M, GC and N scores by the "createBinData" function. The external files can be either genome-wide files containing data from all chromosomes, or lists of files for single chromosomes. If the external files are seperate for each chromosome, they should have the common prefix specified by arguments "mfile", "gcfile" and "nfile", as well as the common suffix specified by arguments "m.suffic", "gc.suffic" and "n.suffic". The name of each chromosome file should be in format "xfile[chr]x.suffix", where "x" stands for "m", "gc" or "n", and "chr" is the name of the chromosome. The names of the chromosomes must be consistent with "names(dat.chip)". Alternatively, if "x.suffix" is NULL, then a genome-wide file containing all chromosomes named "xfile" is imported. The output of "createBinData" function is an object of class "BinData".

The following example utilizes an artificially constructed data-set containing starting positions for both ChIP and input samples, following the format of the "tag2bin" input. It then constructs both genome-wide and chromosome-level M, GC and N files, and shows how to use the "createBinData" with both seperate files for each chromosome and genome-wide files.

```
> #convert read alignments to bin-level counts
> data( tagdat_chip )
> data( tagdat_input )
> dat_chip <- tag2bin( tagdat_chip, binS = 100, fragL = 100 )
> dat_input <- tag2bin( tagdat_input, binS = 100, fragL = 100 )
> #construct M, GC and N text files
> numBins <- as.integer( runif(5, 190, 220 ))
> mapdat <- gcdat <- ndat <- list(1:5)
> allmapdat <- allgcdat <- allndat <- NULL
> for(i in 1:5){
+   mapdat[[i]] <- data.frame( pos = ( 0:(numBins[i]-1 ) ) * 100,
+                              M = runif( numBins[i], 0.9, 1 ) )
+   gcdat[[i]] <- data.frame( pos = ( 0:( numBins[i]-1 ) ) * 100,
```

```
+                            GC = runif( numBins[i], 0.5, 1 ) )
+   ndat[[i]] <- data.frame( pos = ( 0:( numBins[i]-1 ) ) * 100,
+                           N = rbinom( numBins[i], 1, 0.01 ) )
+   allmapdat <- rbind( allmapdat,
+                       cbind( paste("chr", i, sep="" ), mapdat[[i]] ))
+   allgcdat <- rbind( allgcdat,
+                      cbind( paste( "chr", i, sep=""), gcdat[[i]] ))
+   allndat <- rbind( allndat,
+                     cbind( paste("chr", i, sep=""), ndat[[i]]))
+
+   write.table(mapdat[[i]], file = paste( "map_chr", i, ".txt", sep = "" ),
+               sep = "\t",  row.names = FALSE, col.names = FALSE)
+   write.table(gcdat[[i]], file = paste( "gc_chr", i, ".txt", sep = "" ),
+               sep = "\t",  row.names = FALSE, col.names = FALSE)
+   write.table(ndat[[i]], file = paste( "n_chr", i, ".txt", sep = "" ),
+               sep = "\t",  row.names = FALSE, col.names = FALSE)
+ }
> write.table( allmapdat, file = "allmap.txt", sep = "\t",
+             row.names = FALSE, col.names = FALSE )
> write.table( allgcdat, file = "allgc.txt", sep = "\t",
+             row.names = FALSE, col.names = FALSE )
> write.table( allndat, file = "alln.txt", sep = "\t",
+             row.names = FALSE, col.names = FALSE )
> #construct BinData object
> #all M, GC, N files are seperate for each chromosome
> bindata1 <- createBinData( dat_chip, dat_input, mfile = "map_", gcfile = "gc_",
+                           nfile = "n_", m.suffix = ".txt", gc.suffix = ".txt",
+                           n.suffix = ".txt", chrlist = NULL,
+                           dataType = "unique" )

> #GC, N files seperate for each chromosome, M file is genome-wide
> bindata2 <- createBinData( dat_chip, dat_input, mfile = "allmap.txt",
+                           gcfile = "gc_", nfile = "n_", m.suffix = NULL,
+                           gc.suffix = ".txt", n.suffix = ".txt",
+                           chrlist = NULL, dataType = "unique" )
> #M, N files seperate for each chromosome, GC file is genome-wide
> bindata3 <- createBinData( dat_chip, dat_input, mfile = "map_",
+                           gcfile = "allgc.txt", nfile = "n_", m.suffix = ".txt",
+                           gc.suffix = NULL, n.suffix = ".txt",
+                           chrlist = NULL, dataType = "unique" )
> #GC, M files seperate for each chromosome, N file is genome-wide
> bindata4 <- createBinData( dat_chip, dat_input, mfile = "map_", gcfile = "gc_",
+                           nfile = "alln.txt", m.suffix = ".txt",
+                           gc.suffix = ".txt", n.suffix = NULL,
+                           chrlist = NULL, dataType = "unique" )
> #only imports data for chr1 and chr2
> bindata5 <- createBinData( dat_chip, dat_input, mfile = "map_", gcfile = "gc_",
+                           nfile = "n_", m.suffix = ".txt", gc.suffix = ".txt",
+                           n.suffix = ".txt", chrlist = c("chr1", "chr2"),
+                           dataType = "unique" )
```

2.2 Fitting the CSSP model

Now we are in a position to fit the CSSP model using the above bin-level data in either a "data.frame" or a "BinData" object. We provide two methods for fitting the CSSP model: *Minimum Distance Estimation* and *generalized EM method*. The default is the *Minimum Distance Estimation*, which can be applied by:

```
> sampleFit <- cssp.fit( bindata1, ngc=0, beta.init=NULL, e0.init=0.9, p2=0.99,
+                       p1=0.5, tol=0.01)
```

Here, "beta.init" and "e0.init" specify the initial values of b and e_0. "p1" and "p2" specify a range of p-values where the expected distribution is uniform. When testing the ChIP read counts against the background read

distribution, the p-values in this region should be uniformly distributed. "tol" specifies the tolerance level of iterations. "ngc" is the number of knots to use when fitting a spline model for the GC-content scores. For deeply sequenced data, "ngc=2" provides reasonably good fits. For data with low sequencing depths, "ngc=0" is recommended since larger values might result in ill defined model matrices. Because our fitting algorithm can be time consuming for large datasets, during the fitting process we print out the information for each step of our algorithm.

The sample data-set in our package is in no way comaprable with the real data-sets which contain millions of reads. For computational purposes we provide additional options to accomodate large genome size. The related two arguments are "useGrid" (default: FALSE) and "nsize" (default: NULL). If "useGrid=TRUE" is specified, the gridding method is used for fitting the input model. In this case, we adaptively grid the covariate space spanned by mappability and GC-content scores into a certain number of bins, and choose one representative observation within each bin to fit the regression model. The "nsize" argument specifies the number of randomly sampled observations used for fitting the normalization parameter. By default, "nsize=NULL", and all bins are used for fitting the normalization parameter. For genome-wide data, we suggest using "useGrid=TRUE" and "nsize=5000". This usually enables us to fit the model within one hour. The following code shows how to fit the model for genome-wide data:

```
> sampleFit.grid <- cssp.fit( bindata1, useGrid = TRUE, nsize = 500 )
```

In the current version of `CSSP`, two additional arguments are provided to support different practical settings. First, "nonpa" (default: FALSE) specifies whether a nonparametric model is used instead of a GLM to fit the input data. Second, "zeroinfl" (default: FALSE) specifies whether a zero-inflated model is fitted for the ChIP sample. A zero-inflated model is particularly useful for data with low sequencing depth and thus having lots of zero-count bins.

```
> sampleFit.np <- cssp.fit( bindata1, nonpa = TRUE )
> sampleFit.zi <- cssp.fit( bindata1, nonpa = TRUE, zeroinfl = TRUE )
```

An alternative method is to use the *generalized EM method* described in [**?**] by specifying the "method='gem'" argument. In [**?**], we notice that although this method is less time consuming than MDE, its model estimate is not robust in many practical settings. Therefore, in our package, we provide this method for exploratory purposes only.

```
> sampleFit.gem <- cssp.fit( bindata1, ngc = 0, beta.init = NULL, e0.init = 0.9,
+                            tol = 0.01, method = "gem" )
```

The function returns an object of S4 class "CSSPFit" which includes estimates of the model parameters. For example, "e0" is the proportion of ChIP read counts that are generated by the background model. "b" and "mu.chip" are the shape parameter and the mean for the Negative Binomial distribution of the background reads in the ChIP sample, respectively.

```
> slotNames( sampleFit )

 [1] "lambdax"         "lambday"         "e0"              "pi0"
 [5] "a"               "b"               "mu.chip"         "mu.input"
 [9] "mean.sig"        "size.sig"        "p.sig"           "prob.zero"
[13] "post.p.sig"      "post.p.bind"     "post.p.zero"     "post.shape.sig"
[17] "post.shape.back" "post.scale.sig"  "post.scale.back" "n"
[21] "k"               "map.id"          "pvalue"          "cum.pval"

> sampleFit@b

[1] 95.64778

> sampleFit@e0

[1] 0.9196875

> head( sampleFit@mu.chip )

       1        2        3        4        5        6
117.3366 117.5800 117.0066 117.4130 117.2459 117.8487
```

To check the model fit, we suggest plotting the empirical distribution of continuously-corrected p-values obtained by testing the ChIP counts against the fitted background (Fig. **??**):

```
> slotNames( sampleFit )

 [1] "lambdax"         "lambday"         "e0"              "pi0"
 [5] "a"               "b"               "mu.chip"         "mu.input"
 [9] "mean.sig"        "size.sig"        "p.sig"           "prob.zero"
[13] "post.p.sig"      "post.p.bind"     "post.p.zero"     "post.shape.sig"
[17] "post.shape.back" "post.scale.sig"  "post.scale.back" "n"
[21] "k"               "map.id"          "pvalue"          "cum.pval"

> ## code for Fig 1
> plot( sampleFit@pvalue, seq( 0, 1, length = length( sampleFit@pvalue ) ),
+       xlab = "P-value", ylab = "Cumulative Probability", cex = 0.3 )

null device
          1
```

In addition, we may compute the frequency of ChIP counts using the estimated posterior distribution, and compare that to the empirical frequency of the ChIP counts, as shown in Fig. **??**.

```
> freqFit <- fit.freq(sampleFit, bindata1@tagCount)

> ## code for Fig 2
> plot( freqFit$freq ~ freqFit$count, ylab = "Frequency",
+       xlab = "Count", lty = 2, type = "l" )
> lines( freqFit$freq.est ~ freqFit$count, lty = 1 )
> legend( "topright", c( "Empirical", "Fitted" ), lty = 2:1 )

null device
          1
```

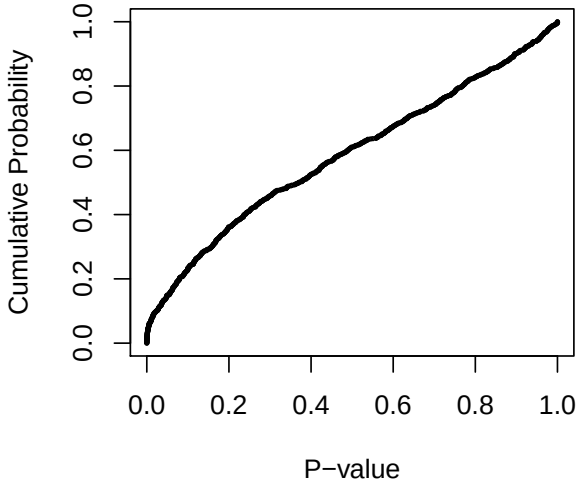

Figure 1: Empirical distribution of the p-values obtained by testing the ChIP bin-level counts against the fitted background model.

2.3 Power Computation

After the CSSP model is fitted, function "cssp.power" enables computing power at a given sequencing depth and FDR level specified by "qval" (default 0.05). In the below script, "x=sampleFit@lambday/10" specifies the sequencing depth at which the power is evaluated, "fold=2" and "min.count=0" specify the fold change (r) and minimum count thresholds (τN_y), respectively:

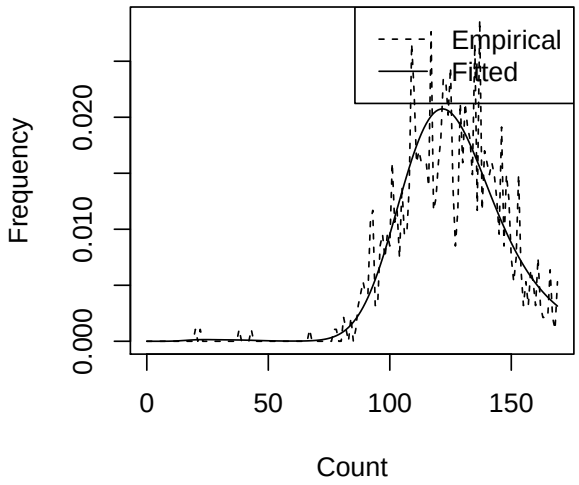

Figure 2: The frequency plots for ChIP bin-level counts.

```
> cssp.power( fit = sampleFit, x = sampleFit@lambday, ite = 5, fold = 2,
+            min.count = 0, useC = TRUE )

[1] 0.5755513
```

We can compute the power at various sequencing depths, ranging from 20% to 100% of the full data sequencing depth, as shown in Fig. **??**, as follows:

```
> power.fit<-sapply( (1:5)/5, function(x)
+                     {
+                       cssp.power( fit = sampleFit, x = sampleFit@lambday*x,
+                                   ite = 5, fold = 0, min.count = 0,useC = TRUE )
+                     })

null device
          1


> power.fit

[1] 0.02894652 0.08415801 0.12802785 0.17729067 0.20554501

> plot(power.fit~seq(20,100,length=5), ylab="Power", type="l",
+      xlab="Sequencing Depth")
```

In the above power computation, the use of minimum count threshold "min.count" should adapt to the underlying sequencing depth of the data. In practice, we can use the values of intensity percentiles of the bin-level Poisson processes. Such values can be estimated by function "qBBT". This function estimates the percentiles of bin-level Poisson distributions based on their estimated posterior distributions.

```
> fit.tail1 <- qBBT( fit = sampleFit, prob = 0.99, depth = sampleFit@lambday )
> fit.tail2 <- qBBT( fit = sampleFit, prob = 0.99, depth = sampleFit@lambday/10 )

> fit.tail1

[1] 187.6083

> fit.tail2

[1] 18.76083
```

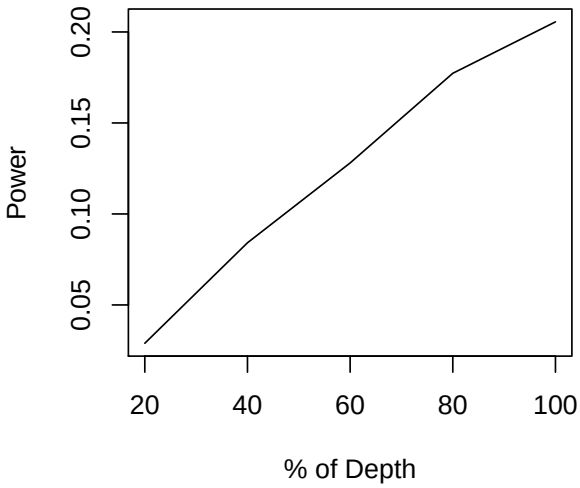

Figure 3: Estimated power for the sample ChIP-seq data at different percentiles of the full data sequencing depth.

For example, we may compute the power for our example data at 10% of its current sequencing depth, with fold change threshold being 2 and minimum count threshold being the 99% of the ChIP sample bin-level intensities by the code below. Notice that the sequencing depth of the "x" argument in the "cssp.power" function must be the same as the "depth" argument in the "qBBT" function before.

```
> cssp.power( fit = sampleFit, x = sampleFit@lambday*0.1, ite = 5,
+             fold = 2, min.count = fit.tail2, useC = TRUE )
```

2.4 Supplementary Functions

CSSP package can be used to estimate an upper bound on the power of other peak callers. We notice that many peak callers do not explicitly report the underlying fold change and minimum thresholds. In this situation, it is important to estimate these two thresholds required by our model. It is desirable to summarize the total number of reads overlapping these binding positions. Here, we introduce the additional functionalities provided by CSSP for mapping tag counts to binding positions.

Binding positions can be reported at either the nucleotide level or as peak intervals. Function "bindcount" maps aligned reads to binding positions reported at the nucleotide level. The argument "bindpos" is a list of genome coordinates of binding positions for each chromosome. The "bindcount" function calculates the number of reads mapping to regions within a distance specified by "whs" of these coordinates. The following example uses an artificially constructed dataset "bindpos", which contains binding positions on 5 chromosomes:

```
> data( bindpos )
> bind.count <- bindcount( tagdat_chip, tagdat_input, bindpos = bindpos, fragL = 100,
+                          whs = 200 )
> summary( bind.count[["chr1"]] )

      Length Class  Mode
chip  93     -none- numeric
input 93     -none- numeric
```

Alternatively, for binding positions reported as peak intervals, the number of reads mapping to each interval can be computed by function "peakcount". The argument "peakpos" is a list of 2-column matrices representing the genome coordinates of peak intervals for each chromosome. The following example uses an artificially constructed dataset "peakpos", which contains peak intervals on 5 chromosomes:

```
> data( peakpos )
> peak.count <- peakcount( tagdat_chip, tagdat_input, peakpos = peakpos, fragL = 100 )
> summary( peak.count[["chr1"]] )
```

```
      Length Class  Mode
chip  93     -none- numeric
input 93     -none- numeric
```

After we obtain the number of reads mapping to all binding sites, the underlying fold change and minimum count thresholds can be computed as follows.

```
> thr.fold <- min( c( bind.count[[1]]$chip / bind.count[[1]]$input,
+                     bind.count[[2]]$chip / bind.count[[2]]$input,
+                     bind.count[[3]]$chip / bind.count[[3]]$input,
+                     bind.count[[4]]$chip / bind.count[[4]]$input,
+                     bind.count[[5]]$chip / bind.count[[5]]$input ) )*
+   sampleFit@lambdax / sampleFit@lambday / sampleFit@e0
> thr.count <- min( c( bind.count[[1]]$chip, bind.count[[2]]$chip,
+                     bind.count[[3]]$chip, bind.count[[4]]$chip, bind.count[[5]]$chip ) )
```

Using these values in the arguments of "cssp.power" function, an estimate of the power for the binding positions in data "bindpos" can be obtained by the following code. Here "qval" specifies the desired FDR level for peak calling:

```
> cssp.power( fit = sampleFit, x = sampleFit@lambday, ite = 5, fold = thr.fold,
+            min.count = thr.count, qval = 0.05, useC = TRUE )
```

3 Session Information

```
R version 3.3.0 RC (2016-04-26 r70550)
Platform: x86_64-apple-darwin13.4.0 (64-bit)
Running under: OS X 10.9.5 (Mavericks)

locale:
[1] C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] CSSP_1.10.0

loaded via a namespace (and not attached):
[1] tools_3.3.0   splines_3.3.0
```