

Normalization example using internal CTCF peaks as a standard

Andrew Holding

8/7/2017

Introduction

This is a worked example of our method to normalize ER ChIP-seq data using CTCF binding to provide an internal control.

Preprocessing.

To prepare the data for the differential binding analysis the data had to be aligned and CTCF and ER peaks called. As our CTCF control doesn't require any special alignment, data was aligned as for a standard ChIP-seq experiment. Peak calling was done with macs2 using the script shown below.

```
#Calls peaks for all files in a directory against a single control. Change
# <filename> to the filename of your input sample.

control=<filename>

for bam in *bam
do
  root=`basename $bam .bam`
  macs2 callpeak -t $bam -c $control -f BAM -n $root -g hs
done
```

As macs2 cannot differentiate between a peak arising from CTCF binding or ER binding we used two samples were sequenced to establish a CTCF binding consensus. One treated and one untreated. These peaks were then merged as follows.

```
cat SLX-14438.D701_D504.HKN27BBXX.s_8.r_1.fq.gz.bam_peaks.xls \
      SLX-14438.D702_D503.HKN27BBXX.s_8.r_1.fq.gz.bam_peaks.xls > CTCF_merged.bed

grep -v '\# CTCF_merged.bed' | grep -v start > CTCF_merged_filtered.bed
sort -k1,1 -k2,2n CTCF_merged_filtered.bed > CTCF_merged_sorted_filtered.bed

bedtools merge -i CTCF_merged_sorted_filtered.bed > CTCF_union.bed
```

We then used this CTCF consensus to establish specific CTCF or ER binding in the experimental samples. Either taking the intersection of the CTCF consensus and the peak file for a particular sample to provide a list of CTCF binding sites in the sample, or subtracting the consensus CTCF peaks from the macs2 output to provide a list ER binding sites for that sample.

```
mkdir CTCF
for f in *.narrowPeak
do
  echo $f
  bedtools intersect -a $f -b CTCF_union.bed > CTCF/$f
done
```

```

mkdir ER
for f in *.narrowPeak
do
    echo $f
    bedtools subtract -b CTCF_union.bed -a $f -A > ER/$f
done

```

These were then used to write two DiffBind samplesheets as described in the DiffBind vignette. One using the CTCF peak list generated from the above scripts, this is the control sample sheet. The second uses the ER peaks, this is the experimental sample sheet. These two sample sheets are then provided as the input in the analysis below.

Load convience functions

Brundle provides a series of functions we developed for this pipeline that are used to normalizing ChIP-seq data. The usage of these functions are explained in the example that follows.

```
source('../package/brundle.R')
```

Apply settings

The initial settings for the analysis are minimal, `jg.controlMinOverlap` is set to 5 as this is 1 less than the total sample count and we want a strong consensus of CTCF peaks to normalise against. The control and experiment sample sheets are as described above. The conditions reflect the treated and untreated condition we have used when writing our sample sheet.

```

jg.controlMinOverlap      <- 5
jg.controlSampleSheet    <- "samplesheet/samplesheet_SLX14438_hs_CTCF_DBA.csv"
jg.experimentSampleSheet <- "samplesheet/samplesheet_SLX14438_hs_ER_DBA.csv"
jg.treatedCondition      = "Fulvestrant"
jg.untreatedCondition    = "none"

```

Load control and experimental DiffBind object

Typically we would load the raw data using Diffbind as we are plan to return our normalized data as a Diffbind object.

In this example, as the BAM files which are greater in size than the github filesize limit, we have included the data as the Diffbind object these function would output and instead load that directly. If the code doesn't find the R data object it will try to regenerate if from the raw data.

```

filename<- "Rdata/example_001_SLX-14438_dba_human_ER_CTCF.rda"
if(!file.exists(filename)){
  dbaExperiment <- jg.getDb(a(jg.experimentSampleSheet, bRemoveDuplicates=TRUE)
  dbaControl    <- jg.getDb(a(jg.controlSampleSheet, bRemoveDuplicates=TRUE)
  save(dbaExperiment, dbaControl, file=filename)
} else {
  load(filename)
}

```

Diffbind allows us to extract the counts in each peak, we use this function to obtain the data in a form that is easier to normalize.

```
# Extract Peak set from DiffBind

jg.experimentPeakset <- jg.dbaGetPeakset(dbaExperiment)
jg.controlPeakset    <- jg.dbaGetPeakset(dbaControl)
```

To normalize we need to separate samples out by condition, this is done using the following function. The peakset contains the reads in peaks, the samplesheet provides the metadata needed and the jg.treatedCondition/jg.untreatedCondition specify which condition we want to get the counts for.

```
#Get counts for each condition
jg.controlCountsTreated<-jg.getControlCounts(jg.controlPeakset,
                                              jg.controlSampleSheet,
                                              jg.treatedCondition)
jg.controlCountsUntreated<-jg.getControlCounts(jg.controlPeakset,
                                                jg.controlSampleSheet,
                                                jg.untreatedCondition)
```

At the same time we can extract the samples names for each condition to use later.

```
#Get sample names for conditions
jg.untreatedNames <- names(jg.controlCountsUntreated)
jg.treatedNames   <- names(jg.controlCountsTreated)
```

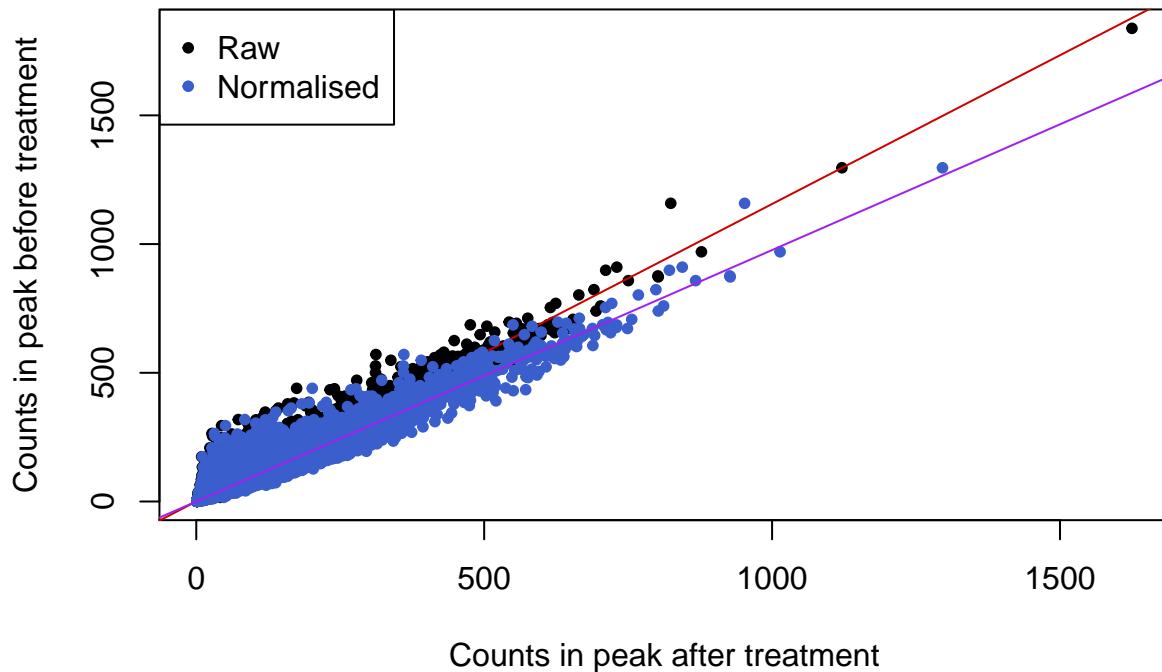
The normalization factor is generated by a linear regression generated from the mean number of counts for each peak before and after treatment. The gradient of the linear fit should be equal to 1 given the assumption that before and after treatment the majority of the control peaks do not change.

This plot is not needed to calculate the normalization factor but it does provide a visual interpretation of the process.

```
# Plot showing normalization calculation (Optional)

jg.plotNormalization(jg.controlCountsTreated,
                     jg.controlCountsUntreated)
```

Comparision of Counts in peaks



```
## rowMeans(jg.controlCountsTreated)
## 1.155732
```

There are two parts to the normalization coefficient we use in the final correction. The first is the the coefficient to make the levels of CTCF binding before and after treatment equal. The second is because DiffBind normalises against the total read count for each sample, we therefore generated a correction factor to control for this.

```
# Get Normalization Coefficient

jg.coefficient<-jg.getNormalizationCoefficient(jg.controlCountsTreated,
                                                jg.controlCountsUntreated)
jg.correctionFactor<-jg.getCorrectionFactor(jg.experimentSampleSheet,
                                              jg.treatedNames,
                                              jg.untreatedNames)
```

Once we have these two pieces of information we apply both the normalisation coefficient and the correction factor to our data.

```
#Apply coefficient and control factor
jg.experimentPeaksetNormalised<-jg.applyNormalisation(jg.experimentPeakset,
                                                       jg.coefficient,
                                                       jg.correctionFactor,
                                                       jg.treatedNames)
```

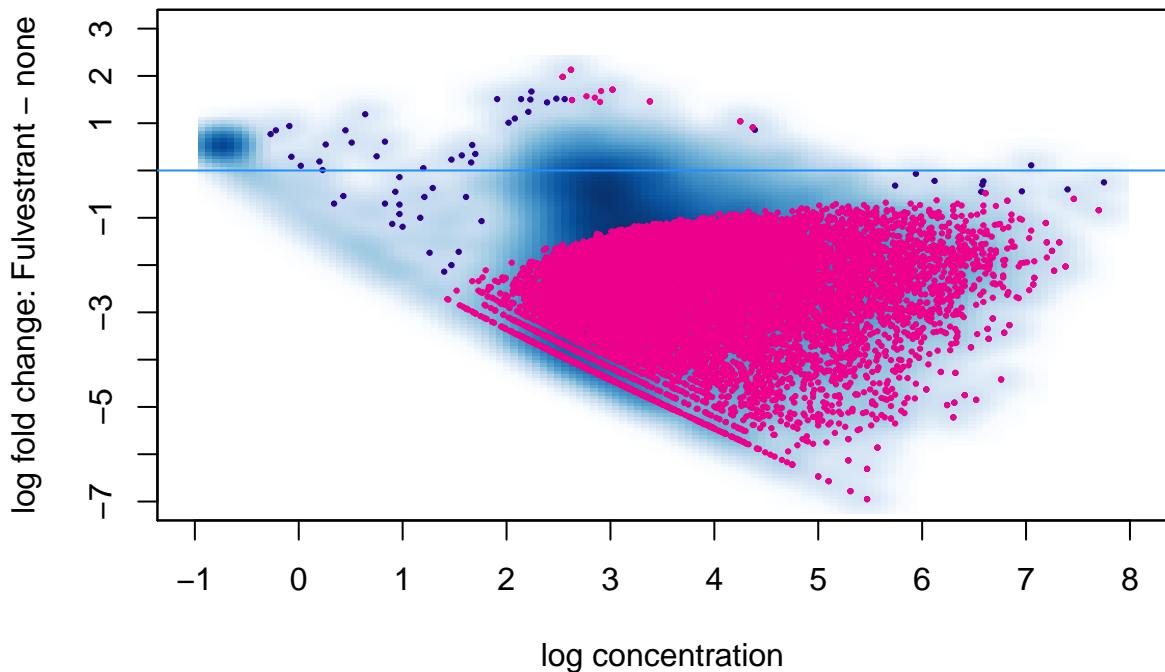
Finally we return the correct values into the DiffBind object. This requires a modified version Diffbind as included in the original manuscript. It is expected that this will be supported natively by future versions of Diffbind.

```
#Return values to Diffbind and plot normalised result.
jg.dba <- DiffBind:::pv.resetCounts(dbaExperiment,
                                      jg.experimentPeaksetNormalised)
```

We can then visualise the data as normal using DiffBind.

```
jg.dba_analysis<-dba.analyze(jg.dba)
dba.plotMA(jg.dba_analysis,bFlip=TRUE)
```

Binding Affinity: Fulvestrant vs. none (10649 FDR < 0.050)



Save results

```
write.csv(dba.report(jg.dba_analysis),file="results/Example_001.csv")
```