BitSeq User Guide

Peter Glaus, Antti Honkela and Magnus Rattray

July 11, 2013

1 Abstract

The *BitSeq* package is targeted for transcript expression analysis and differential expression analysis of RNA-seq data in two stage process. In the first stage it uses Bayesian inference methodology to infer expression of individual transcripts from individual RNA-seq experiments. The second stage of *BitSeq* embraces the differential expression analysis of transcript expression. Providing expression estimates from replicates of multiple conditions, Log-Normal model of the estimates is used for inferring the condition mean transcript expression and ranking the transcripts based on the likelihood of differential expression.

2 Citing *BitSeq*

The *BitSeq* package is based on probabilistic models and inference methods described in the manuscript [1]. For citing *BitSeq* in publications please refer to the manuscript above and to the source of the software.

3 Installing the *BitSeq* package

The recommended way to install *BitSeq* is to use the `biocLite` function available from the bioconductor website.

```
> source("http://www.bioconductor.org/biocLite.R")
> biocLite("BitSeq")
```

To load the package start R and run

```
> library(BitSeq)
```

4 Preparing data

The size of data normally analysed by *BitSeq* and results represented by samples from posterior distribution usually ranges in Gigabytes. Keeping this kind of datasets in memory within R environment would be rather inefficient and in most cases unnecessary, thus most of the data used within *BitSeq* is loaded and saved directly to the local hard drive.

The input for *BitSeq* package can be either SAM or BAM file containing aligned reads, as well as reference Fasta file. These files do not need to be loaded

into the environment as *BitSeq* will read the data from disk space. The transcriptome Fasta file can be either downloaded from Ensembl website [1] or constructed by UCSC genome browser at `http://genome.ucsc.edu/cgi-bin/hgTables`. In this example we will use file *ensSelect1.fasta* which contains sequence of all transcripts for five genes from the Ensembl homo sapiens, release 65 annotation.

For the alignment of reads to transcriptome we recommend using software `bowtie`, which is able to report all valid alignments for all the reads. The following `bash` commands can be used to create the SAM file with alignments:

```
# create bowtie reference index for the annotation
$ bowtie-build -f --ntoa ensSelect1.fasta ensSelect1-index
# align reads in data-c0b0.fastq against index
$ bowtie -q -v 3 -3 0 -p 4 -a -m 100 --sam ensSelect1-index \
  data-c0b0.fastq data-c0b0.sam
```

In the following examples we will be using the *data-c0b0.sam* file provided with the package. To make the life easier we set our current directory to the *extdata* directory.

```
> # save the current directory
> # (we move back to old_directory at the end of vignette)
> old_directory = getwd();
> on.exit(setwd(old_directory))
> # move to directory with the data
> setwd(system.file("extdata",package="BitSeq"));
```

5 Basic use

5.1 Estimating expression

To estimate expression we use the function `getExpression`, which takes as an input the SAM file with alignments as well as reference Fasta file that was used for the alignment. The function returns a `list` in which the first item `exp` is a `DataFrame` with expression mean and standard deviation of each transcript. The second item `fn` is a file name of a file containing all the expression samples, which are used in the later DE analysis. The last two items are `counts`, vector containing estimated read counts per transcript, and `trInfo`, `DataFrame` with information about transcripts.

The log option tells the function to return mean and standard deviation of logged samples and the last three options, which are parameters for the sampling algorithm, are passed to the `estimateExpression` function used for expression inference.

```
> res1 <- getExpression("data-c0b0.sam",
+     "ensSelect1.fasta",
+     log = TRUE,
+     MCMC_burnIn=200,
+     MCMC_samplesN=200,
+     MCMC_samplesSave=50,
+     seed=47)
```

[1] link for homo sapiens, release 66: `ftp://ftp.ensembl.org/pub/release-66/fasta/homo_sapiens/cdna/Homo_sapiens.GRCh37.66.cdna.all.fa.gz`

```
[time: +0.000000 m]
Reads: all(Ntotal): 4817  mapped(Nmap): 4663
[time: +0.000000 m]
[time: +0.000000 m]
Alignments: 21259.
[time: +0.000000 m]
Mappings: 4663
Ntotal: 4817
Finished Reading!
Total hits = 25910
Isoforms: 56
Burn in: 200 DONE. [time: +0.016667 m]

Sampling DONE. [time: +0.000000 m]
rHat (for 200 samples)
   rHat (rHat from subset |    tid | mean theta)
    1.2936 ( 1.2588 |     14 |  0.0299)
    1.2527 ( 1.2370 |     15 |  0.0837)
    1.2243 ( 1.2424 |      2 |  0.0188)
  Mean rHat of worst 10 transcripts: 1.145211
  Mean C0: (0 0 0 0 ). Nunmap: 154

Producing 951 final samples from each chain.

Sampling DONE. [time: +0.016667 m]
rHat (for 951 samples)
   rHat (rHat from subset |    tid | mean theta)
    1.0228 ( 1.0199 |     16 |  0.0052)
    1.0211 ( 1.0948 |      2 |  0.0181)
    1.0208 ( 1.0240 |     32 |  0.0076)
  Mean rHat of worst 10 transcripts: 1.012858
  Mean C0: (0 0 0 0 ). Nunmap: 154

Total samples: 4604
```

The data being processed in this vignette is a small, example dataset, thus it is safe to use lower values for `MCMC_burnIn`, `MCMC_samplesN`, `MCMC_samplesSave`. For normal sized dataset, you can use default values of these parameters (all three parameters have default value 1000).

To view histogram of log RPKM expression use:

```
> hist(res1$exp$mean)
```

We can load the expression samples using function `loadSamples`, which returns `DataFrame` containing all expression samples.

```
> samples1 <- loadSamples(res1$fn)
```

Following command produces plot showing correlation between two transcript expression estimates:

```
> plot( unlist(s2["ENST00000436661",]),
+    unlist(s2["ENST00000373501",]))
```

The `getExpression` function first computes probabilities for every alignment and then uses this data in Markov chain Monte Carlo algorithm which samples from the posterior distribution of transcript expression. Both these steps are computationally very expensive and can take several hours to finish. In case of MCMC sampling, which has to converge to the correct posterior distribution this can take more than day for extensive data.

5.2 Identifying differentially expressed transcripts

In the differential expression analysis we will use the expression samples produced in first step as well as expression samples from other experiments provided with the package. It is essential to use biological replicates in both conditions in order to account for biological variation which could otherwise cause false positive DE calls.

```
> cond1Files = c(res1$fn,"data-c0b1.rpkm")
> cond2Files = c("data-c1b1.rpkm","data-c1b1.rpkm")
```

We use the function `getDE` to infer the Probability of Positive Log Ratio for each transcript. The function again returns `list` with first item `pplr` containing `DataFrame` with PPLR and other information. The second item `fn` contains list of filenames with the names of produced files. Using the option`samples=TRUE`, the function creates also files containing the condition mean expression samples.

```
> de1 <- getDE(list(cond1Files, cond2Files),
+    samples=TRUE)

[time: +0.000000 s]
# 5 done. [time: +0.000000 s]
# 10 done. [time: +0.000000 s]
# 15 done. [time: +0.000000 s]
# 20 done. [time: +0.000000 s]
# 25 done. [time: +0.000000 s]
# 30 done. [time: +0.000000 s]
# 35 done. [time: +0.000000 s]
# 40 done. [time: +0.000000 s]
# 45 done. [time: +0.000000 s]
# 50 done. [time: +0.000000 s]

> print(de1$fn)

$pplr
[1] "/tmp/RtmpOfhNzN/dataBS-DE-135e6c7adc55.pplr"

$samplesFiles
[1] "/tmp/RtmpOfhNzN/dataBS-DE-135e6c7adc55-C0.est"
[2] "/tmp/RtmpOfhNzN/dataBS-DE-135e6c7adc55-C1.est"
```

Now we can rank the transcripts based on the PPLR value to identify the ones with the highest probability of being differentially expressed:

```
> head( de1$pplr[ order(abs(0.5-de1$pplr$pplr), decreasing=TRUE ), ], 5)
```

```
DataFrame with 5 rows and 6 columns
                 pplr 1~2 log2FC 1~2
                <numeric>  <numeric>
ENST00000370994  0.000000  -0.967742
ENST00000465588  0.942308   0.868840
ENST00000479075  0.134615  -0.548985
ENST00000489538  0.846154   0.465076
ENST00000373501  0.826923   0.601043
                    ciLow    ciHigh
                <numeric> <numeric>
ENST00000370994 -1.606570 -0.363337
ENST00000465588 -0.123376  2.209360
ENST00000479075 -1.331340  0.158981
ENST00000489538 -0.456433  1.303560
ENST00000373501 -0.481008  1.981010
                   mean 1    mean 2
                <numeric> <numeric>
ENST00000370994   9.48083   8.81004
ENST00000465588   8.79790   9.40013
ENST00000479075   9.02961   8.64908
ENST00000489538   9.09444   9.41681
ENST00000373501   9.74622  10.16280
```

6 Advanced use

Both expression estimation and identification of differentially expressed transcripts involves multiple steps which are independent. Computing these steps independently might be useful for keeping intermediate results in case of crash or error. As BitSeq makes extensive use of local files, it is essential to set path to working directory containing alignment files and which will be used for storing results of individual steps. In this example we use the *extdata* directory provided with the package:

```
> setwd(system.file("extdata",package="BitSeq"))
```

6.1 Stage 1 - Transcript expression analysis

6.1.1 Pre-processing alignments

In the pre-processing step, the `parseAlignment` function reads the SAM file and assigns a probability to every valid alignment. These probabilities are saved into *.prob* file and are the direct input for the expression estimation. We have to specify the reference file which is used for identifying base mismatches and we use uniform model for the read distribution along transcript:

```
> parseAlignment( "data-c0b0.sam",
+     outFile = "data-c0b0.prob",
+     trSeqFile = "ensSelect1.fasta",
+     trInfoFile = "data.tr",
+     verbose = TRUE )
```

```
Assuming alignment file in 'sam' format.
Using alignments' header for transcript information.
Initializing fasta sequence reader.
Found gene names in sequence file, updating transcript information.
[time: +0.000000 m]
Using uniform read distribution.
Reads: all(Ntotal): 4817  mapped(Nmap): 4663
  491 reads were used to estimate non-uniform distribution.
[time: +0.000000 m]
Writing alignment probabilities.
# 481 done. [time: +0.000000 m]
# 962 done. [time: +0.000000 m]
# 1443 done. [time: +0.000000 m]
# 1924 done. [time: +0.000000 m]
# 2405 done. [time: +0.000000 m]
# 2886 done. [time: +0.000000 m]
# 3367 done. [time: +0.016667 m]
# 3848 done. [time: +0.000000 m]
# 4329 done. [time: +0.000000 m]
# 4810 done. [time: +0.000000 m]
[time: +0.016667 m]
Analyzed 4817 reads:
 154 had no alignments
The rest had 21259 alignments:
 21259 single-read alignments
Computing effective lengths.
Transcript information saved into data.tr.
[time: +0.000000 m]
DONE. [time: +0.016667 m]
```

The program passes the SAM file twice and produces the *data-c0b0.prob* file with the alignment probabilities as well as transcript information file *data.tr* which contains transcript names and lengths extracted from the SAM file.

6.1.2 Estimating transcript expression

The `estimateExpression` function implements a generative model of RNA-seq data and infers the transcript expression using Markov chain Monte Carlo algorithm. The default MCMC algorithm is the Collapsed Gibbs sampling which converges faster than regular Gibbs sampling (selectable by option `gibbs=TRUE`. It is the most time consuming part of the *BitSeq* analysis process as it uses multiple independent chains to sample the expression values and it iterates until the chains converge to the same distribution.

The following example runs the sampler using the *.prob* file from previous step, produces expression in RPKM measure and produces files with the prefix *data-c0b0*. It will produce two files, file *data-c0b0.rpkm* will contain a row for each transcript with `MCMC_samplesSave` RPKM expression samples. The second file *data-c0b0.thetaMeans* will contain the mean relative expression values for every transcript.

```
> estimateExpression("data-c0b0.prob",
```

```
+     outFile = "data-c0b0", outputType = "RPKM",
+     trInfoFile = "data.tr", MCMC_burnIn = 200,
+     MCMC_samplesN = 200, MCMC_samplesSave = 100,
+     MCMC_chainsN = 2)

Mappings: 4663
Ntotal: 4817
Finished Reading!
Total hits = 25910
Isoforms: 56
Burn in: 200 DONE. [time: +0.000000 m]

Sampling DONE. [time: +0.000000 m]
rHat (for 200 samples)
   rHat (rHat from subset |    tid | mean theta)
    1.2368 ( 1.2506 |     54 |  0.0081)
    1.2116 ( 1.2341 |     43 |  0.0114)
    1.0893 ( 1.0824 |      2 |  0.0185)
  Mean rHat of worst 10 transcripts: 1.080797
  Mean C0: (0 0 ). Nunmap: 154

Producing 3222 final samples from each chain.

Sampling DONE. [time: +0.050000 m]
rHat (for 3222 samples)
   rHat (rHat from subset |    tid | mean theta)
    1.0074 ( 1.0025 |     43 |  0.0118)
    1.0074 ( 1.0068 |     54 |  0.0078)
    1.0039 ( 1.0058 |     31 |  0.0087)
  Mean rHat of worst 10 transcripts: 1.002842
  Mean C0: (0 0 ). Nunmap: 154

Total samples: 6844
```

The behavior of the sampling algorithm can be adjusted by optional arguments, such as `MCMC_chainsN` which selects the number of chains. After producing `MCMC_burnIn` burn-in samples, the algorithm produces first `MCMC_samplesN` samples from each chain in the first iteration. These are used to estimate the number of samples needed for recording `MCMC_samplesSave` effective samples, in the second, final, iteration.

6.1.3 Convergence checking via possible scale reduction estimation

The `estimateExpressionLegacy` uses different convergence checking mechanism which mostly results in multiple iterations, producing more samples in total. After each iteration, the possible scale reduction of marginal posterior variance $\widehat{R}$ is computed for each transcript expression and the ten highest values are reported. If the average of ten highest possible scale reductions is less than the `MCMC_scaleReduction` parameter, then the sampler produces one last iteration during which subset of `MCMC_samplesSave` samples is recorded. Otherwise the program continues with next iteration in which it produces twice as

many samples. The program terminates either after reaching the target scale reduction or after iteration which produces `MCMC_samplesNmax` samples. All these parameters can be set also within a parameters file specified by the option `parFile` with the advantage that the parameters such as `MCMC_scaleReduction` or `MCMC_samplesNmax` can be adjusted while the sampler is running, example of the parameters file *parameters1.txt* is provided in the *extdata* directory.

```
> estimateExpressionLegacy("data-c0b0.prob",
+     outFile = "data-c0b0", outputType = "RPKM",
+     trInfoFile = "data.tr", MCMC_burnIn = 200,
+     MCMC_samplesN = 200, MCMC_samplesSave = 100,
+     MCMC_scaleReduction = 1.1,
+     MCMC_chainsN = 2)

Mappings: 4663
Ntotal: 4817
Finished Reading!
Total hits = 25910
Isoforms: 56
Burn in: 200 DONE. [time: +0.000000 m]

Sampling DONE. [time: +0.000000 m]
rHat (for 200 samples)
   rHat (rHat from subset |    tid | mean theta)
    1.2282 ( 1.2399 |     43 |  0.0111)
    1.2216 ( 1.2721 |     54 |  0.0087)
    1.1249 ( 1.1188 |     15 |  0.0927)
  Mean rHat of worst 10 transcripts: 1.084929
   (target: 1.100)
  Mean C0: (0 0 ). Nunmap: 154

Producing 200 final samples from each chain.

Sampling DONE. [time: +0.000000 m]
rHat (for 200 samples)
   rHat (rHat from subset |    tid | mean theta)
    1.0755 ( 1.0814 |     16 |  0.0053)
    1.0386 ( 1.0362 |     31 |  0.0086)
    1.0354 ( 1.0322 |     10 |  0.0012)
  Mean rHat of worst 10 transcripts: 1.025933
   (target: 1.100)
  Mean C0: (0 0 ). Nunmap: 154

Total samples: 800
```

6.1.4 Examining the samples

Again, we can view the resulting file (*data-c0b0.rpkm* in this case) using the function `loadSamples`.

```
> loadSamples("data-c0b0.rpkm")
```

6.2 Stage 2 - Differential expression analysis

6.2.1 Preparing for Differential Expression analysis

In the differential expression analysis we are comparing samples from two different conditions. Also in order to estimate biological variance of transcript expression, we have to use data from at least one extra biological replicates. We first specify the files containing expression samples from the Stage 1, using file *data-c0b0.rpkm* computed in previous example and three other files provided with the package:

```
> allConditions = list(c("data-c0b0.rpkm","data-c0b1.rpkm"),
+                      c("data-c1b1.rpkm","data-c1b1.rpkm"))
```

The estimation of expression specific hyperparameters for the DE model requires pre computing joint mean expression over all experiments using the `getMeanVariance` function. As the DE model uses logged expression samples, we have to compute the mean and variance of logged expression samples:

```
> getMeanVariance(allConditions,
+    outFile = "data.means",
+    log = TRUE )
```

6.2.2 Estimating model hyperparameters

The hyperparameters for the model are estimated from the entire data using Metropolis-Hastings MCMC algorithm. The values are smoothed afterwards using the non-parametric Lowess smoothing algorithm:

```
> estimateHyperPar( outFile = "data.par",
+    conditions = allConditions,
+    meanFile = "data.means",
+    verbose = TRUE )

Transcripts in expression file: 55
Samples are not logged. (will log for you)
Using -100 as minimum instead of log(0).
Files with samples loaded.
Number of all replicates: 4
seed: 1373601739
Expression step: 0.0435576
[time: +0.000000 s]
Running sampler.
.....
Sampling done.
Have 10 parameters to smooth.
# alphaSmooth f: 0.2 nSteps: 5
# betaSmooth f: 0.2 nSteps: 5
DONE.
[time: +0.150000 m]
```

6.2.3 Inferring condition mean expression and calculating Probability of Positive Log Ratio

The model for Differential Expression analysis uses the posterior samples from expression analysis to infer samples of the mean expression for each transcript in every condition. Function `estimateDE` computes the samples and uses them to compute the Probability of Positive Log Ratio, which is the probability of a transcript being up-regulated in the first condition as well as inverse probability of transcript being down-regulated in the first condition. The PPLR, mean $\log_2$ fold change with confidence intervals and mean condition mean expression are saved into the final output file with extension *.pplr* and prefix specified by the option `outFile`:

```
> estimateDE(allConditions,
+    outFile = "data",
+    parFile = "data.par" )

# 5 done. [time: +0.000000 s]
# 10 done. [time: +0.000000 s]
# 15 done. [time: +0.000000 s]
# 20 done. [time: +0.000000 s]
# 25 done. [time: +0.000000 s]
# 30 done. [time: +0.000000 s]
# 35 done. [time: +0.000000 s]
# 40 done. [time: +0.000000 s]
# 45 done. [time: +0.000000 s]
# 50 done. [time: +0.000000 s]

> ##
> ## pretend run with three conditions and normalization constants
> ##
> cond3Files = c("data-c2b0.rpkm","data-c2b1.rpkm", "data-c2b2.rpkm")
> estimateDE(list( allConditions[[1]], allConditions[[2]], cond3Files),
+            outFile="mydata",
+            parFile="mydata.par",
+            norm=c(1.0, 0.999, 1.0017, 0.9998, 1.0, 0.99, 0.97),
+            pretend=TRUE)

estimateDE data-c0b0.rpkm data-c0b1.rpkm C data-c1b1.rpkm data-c1b1.rpkm C\
 data-c2b0.rpkm data-c2b1.rpkm data-c2b2.rpkm --outPrefix mydata\
 --parameters mydata.par --norm 1,0.999,1.0017,0.9998,1,0.99,0.97
```

In case one is interested in the condition mean expression samples as well, they can be obtained by using the `samples` flag:

```
> estimateDE(allConditions,
+    outFile = "data",
+    parFile = "data.par",
+    samples = TRUE )

# 5 done. [time: +0.000000 s]
# 10 done. [time: +0.000000 s]
```

```
# 15 done. [time: +0.000000 s]
# 20 done. [time: +0.000000 s]
# 25 done. [time: +0.000000 s]
# 30 done. [time: +0.000000 s]
# 35 done. [time: +0.000000 s]
# 40 done. [time: +0.000000 s]
# 45 done. [time: +0.000000 s]
# 50 done. [time: +0.000000 s]
```

This produces three extra files, the first two *data-C0.est*, *data-C1.est* containing the condition means for each condition and the third file *data.estVar* containing samples of inferred variance for the first condition.

7 External use of *BitSeq*

All major computation in *BitSeq* is executed by running `C++` libraries, and there is also a `C++` only implementation of *BitSeq* package available at `http://code.google.com/p/bitseq/`. The standalone package can be particularly useful for clusters without support for R or Bioconductor.

In order to facilitate the use of `C++` version of the package, the relevant functions in R interface provide `pretend` option. Using this option, the computation will not be executed, instead the function will print out one or more command line commands which can be directly used with the `C++` implementation.

```
> res1 <- getExpression("data-c0b0.sam",
+     "ensSelect1.fasta",
+     outPrefix="localDir/data-c0b0",
+     log = TRUE,
+     MCMC_burnIn=200,
+     MCMC_samplesN=200,
+     pretend=TRUE)
```

```
parseAlignment data-c0b0.sam --outFile localDir/data-c0b0.prob --trSeqFile\
 ensSelect1.fasta --trInfoFile localDir/data-c0b0.tr --uniform
estimateExpression  localDir/data-c0b0.prob --outPrefix localDir/data-c0b0\
 --outType rpkm --trInfoFile localDir/data-c0b0.tr --MCMC_burnIn 200\
 --MCMC_samplesN 200
getVariance localDir/data-c0b0.rpkm --outFile localDir/data-c0b0.mean --log
```

References

[1] Glaus, P., Honkela, A. and Rattray, M. (2012). Identifying differentially expressed transcripts from RNA-seq data with biological variation. *Bioinformatics*, **28**(13), 1721-1728.