Supplementary material

Alexey Uvarovskii, Christoph Dieterich

2017-03-09

Contents

Introduction

Here we validate performance of the pulseR package on the simulated data. We consider a pulse-experiment with no spike-ins added. The data include the total, the pull-out and the flow-through fractions for four time points in two replicates.

In the pulse-experiment, the unlabelled RNA is being degraded, starting from the steady-state amount at time $t = 0$hr. The labelled fraction is being synthesised with the rate s, and it degrades with the same rate d as well. A system of ordinary differential equations, which describes this processes, is

$$\begin{aligned} \frac{d[\text{labelled RNA}]}{dt} &= s - d[\text{labelled RNA}] \\ \frac{d[\text{unlabelled RNA}]}{dt} &= -d[\text{unlabelled RNA}] \\ [\text{total}] &= [\text{labelled}] + [\text{unlabelled}] \end{aligned}$$

For constant rates of degradation d and synthesis s, the amounts of RNA at the time t are

$$\begin{aligned}
[\text{total RNA}] &= \frac{s}{d} \equiv \mu \\
[\text{labelled RNA}] &= \mu(1 - e^{-dt}) \\
[\text{unlabelled RNA}] &= \mu e^{-dt},
\end{aligned}$$

where μ is the expression level of a gene in a steady state.

Definitions

Formulas

We can simulate count data using internal pulseR function `pulseR:::generateTestDataFrom`. For this, one needs to define the formulas for the mean read number of all three fractions:

```
library(pulseR)

set.seed(259)

nGenes <- 530
nReplicates <- 2
nTime <- 4


formulas <- MeanFormulas(
  total         = mu,
  labelled      = mu * (1 - exp(-d*time)),
  unlabelled    = mu * exp(-d*time)
)
```

Fractions

In reality, there are no pure labelled or unlabelled fractions in samples. That is why it may be important to model fraction cross-contamination. Here we assume that the labelled fraction consists of certain amount of the unlabelled one, and, vice versa, the unlabelled one is contaminated with the labelled molecules. Hence, we distinguish the number of labelled reads and the number of reads in the "labelled" pull-out fraction.

This definition allows us to calculate mean read number in the fractions as a linear combination of the formulas for the labelled and unlabelled RNA amounts:

```
# important: the labels must be the same as in formulas,
# or they must be integer list indexes
```

```r
formulaIndexes <- list(
  total_sample   = 'total',
  label_sample   = c('labelled', 'unlabelled'),
  unlabel_sample = c('unlabelled', 'labelled')
)
```

Conditions

We specify the condition `data.frame`, which includes information about the type of sample and its time point:

```r
# create nReplicates * nTime * 3 (labelled, unlabelled, total) data points
conditions <- data.frame(
  condition = rep(names(formulaIndexes), each = nTime),
  time      = rep(1:nTime, length(formulaIndexes) * nReplicates))
rownames(conditions) <- paste0("sample_", seq_along(conditions$condition))
knitr::kable(conditions)
```

	condition	time
sample_1	total_sample	1
sample_2	total_sample	2
sample_3	total_sample	3
sample_4	total_sample	4
sample_5	label_sample	1
sample_6	label_sample	2
sample_7	label_sample	3
sample_8	label_sample	4
sample_9	unlabel_sample	1
sample_10	unlabel_sample	2
sample_11	unlabel_sample	3
sample_12	unlabel_sample	4
sample_13	total_sample	1
sample_14	total_sample	2
sample_15	total_sample	3
sample_16	total_sample	4
sample_17	label_sample	1
sample_18	label_sample	2
sample_19	label_sample	3
sample_20	label_sample	4
sample_21	unlabel_sample	1
sample_22	unlabel_sample	2
sample_23	unlabel_sample	3
sample_24	unlabel_sample	4

Normalisation factors

Since we are going to simulate an experiment without spike-ins, one needs to define the relation between different samples via *normalisation factors*. We divide samples into groups depending on their time point and fraction. Except the total fraction, other samples are considered to belong to different group if the time points are different:

```
fractions <- as.character(interaction(conditions))
# assume that the total RNA amount does not change with the time
# so all total fractions can be treated together
fractions[grep("total", fractions)] <- "total_sample"
knitr::kable(cbind(rownames(conditions),fractions))
```

	fractions
sample_1	total_sample
sample_2	total_sample
sample_3	total_sample
sample_4	total_sample
sample_5	label_sample.1
sample_6	label_sample.2
sample_7	label_sample.3
sample_8	label_sample.4
sample_9	unlabel_sample.1
sample_10	unlabel_sample.2
sample_11	unlabel_sample.3
sample_12	unlabel_sample.4
sample_13	total_sample
sample_14	total_sample
sample_15	total_sample
sample_16	total_sample
sample_17	label_sample.1
sample_18	label_sample.2
sample_19	label_sample.3
sample_20	label_sample.4
sample_21	unlabel_sample.1
sample_22	unlabel_sample.2
sample_23	unlabel_sample.3
sample_24	unlabel_sample.4

We use the internal package function `pulseR:::addKnownToFormulas` in order to get the correct form of the list with the normalisation factors.

```
known <- pulseR:::addKnownToFormulas(formulas, formulaIndexes, conditions)
normFactors <- known$formulaIndexes[unique(names(known$formulaIndexes))]
```

```r
# the total fractions have the reference factor = 1
normFactors <- normFactors[-grep("total", names(normFactors))]
normFactors <- c(list(total_sample = 1), normFactors)
# linear combinations with the coefficient 0.1 for contaminating fraction
# and the coefficient 3 for the main fraction of the sample
normFactors <- relist(c(1, rep(3, length(unlist(normFactors)) - 1)),
                      normFactors)
normFactors[-1] <- lapply(normFactors[-1], '[[<-',2,.10)
str(normFactors)
```

```
## List of 9
##  $ total_sample    : num 1
##  $ label_sample.1  : num [1:2] 3 0.1
##  $ label_sample.2  : num [1:2] 3 0.1
##  $ label_sample.3  : num [1:2] 3 0.1
##  $ label_sample.4  : num [1:2] 3 0.1
##  $ unlabel_sample.1: num [1:2] 3 0.1
##  $ unlabel_sample.2: num [1:2] 3 0.1
##  $ unlabel_sample.3: num [1:2] 3 0.1
##  $ unlabel_sample.4: num [1:2] 3 0.1
```

Simulation

We sample the mean expression level and the degradation rates from finite intervals:

```r
# set size factor for the rnbinom function
par <- list(size = 1e2)
# set mean read number for every gene
# by sampling from the log-uniform distribution
par$mu <- exp(runif(nGenes, 0, log(1e5)))
# set the degradation rates from log-uniform distribution
par$d <- exp(runif(nGenes,log(0.01), log(2)))
```

And, finally, we simulate the data from the negative binomial distribution using the `generateTestDataFrom` function:

```r
# the generateTestDataFrom needs for an input normFactors
# provided for every individual sample (i.e. we need to utilise
# the group information ourselves)
allNormFactors <- pulseR:::multiplyList(normFactors, fractions)

counts <- generateTestDataFrom(
  formulas, formulaIndexes, allNormFactors, par, conditions)
```

Fitting

This section describes a usual workflow for the analysis of the count data originating from the spike-ins-free experiment defined above. This is an entry point of the package application to the real data.

Create the PulseData object

We define a PulseData object on the basis of the generated read counts, the condition matrix, relations between formulas and samples. In addition, we provide the information how to split the samples during the normalisation.

```
pd <- PulseData(
  counts = counts,
  conditions = conditions,
  formulas = formulas,
  formulaIndexes = formulaIndexes,
  groups = fractions
)
```

Set fitting options

Before the fitting, one needs to provide the lower and upper boundaries for the parameters values. Also we set here the relative tolerance thresholds.

```
# lower and upper boundaries for the normalisation factors.
# The total one will be ignored, because the first coefficient
# is always the reference one with the value 1.
# The reason is that we can identify the means of read numbers only
# up to an unknown multiplier.
lbNormFactors <- list(
  total_sample   = 1,
  label_sample   = c(.1,.010),
  unlabel_sample = c(.1,.010))
ubNormFactors <- list(
  total_sample   = 1,
  label_sample   = c(10, 2),
  unlabel_sample = c(10, 2))
# set lower and upper boundaries for the rest of the parameters
opts <- setBoundaries(
  list(mu = c(1, 1e6), d = range(par$d) * c(1 / 5, 5)),
  normFactors = list(lbNormFactors, ubNormFactors))
# set the tolerance for the gene-specific parameters and
```

```r
# the normalisation factors
opts <- setTolerance(params = .01, normFactors = .001, options = opts)
```

Fit

One needs to provide the first guess of the parameters values to the fitting function. The outcome of the fitting procedure may depend on the choice of the initial values.

```r
# initialise mu and d as gene-specific parameters, i.e. they must
# be sampled for every single gene individually
initPars <- initParameters(par = NULL, geneParams = c("mu", "d"), pd, opts)
str(initPars)
```

```
## List of 4
##  $ mu         : num [1:530] 224008 871258 876756 302339 915588 ...
##  $ d          : num [1:530] 8.6 3.77 2.83 4.78 9.04 ...
##  $ size       : num 7.31e+09
##  $ normFactors:List of 9
##   ..$ : num 1
##   ..$ : num [1:2] 1 0.01
##   ..$ : num [1:2] 1 0.01
##   ..$ : num [1:2] 1 0.01
##   ..$ : num [1:2] 1 0.01
##   ..$ : num [1:2] 1 0.01
##   ..$ : num [1:2] 1 0.01
##   ..$ : num [1:2] 1 0.01
##   ..$ : num [1:2] 1 0.01
```

Now we are ready to start the fitting:

```r
res <- fitModel(pd, initPars, opts)
```

Results

Normalisation factors

The normalisation of the main fraction in the samples is recovered well (i.e. the coefficient 3).

```r
str(res$normFactors)
```

```
## List of 9
##  $ : num 1
##  $ : num [1:2] 2.997 0.079
##  $ : num [1:2] 3.0179 0.0581
##  $ : num [1:2] 3.0185 0.0399
```

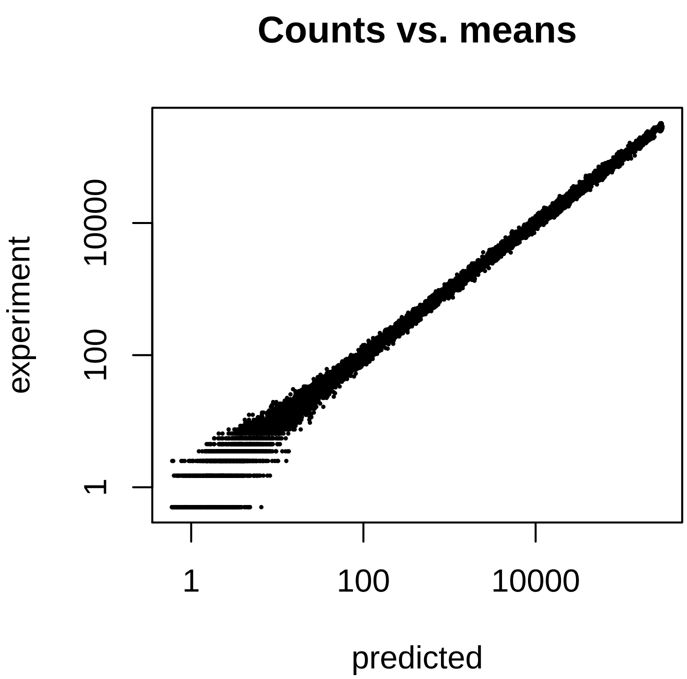
```
##  $ : num [1:2] 3.05 0.01
##  $ : num [1:2] 3.0621 0.0827
##  $ : num [1:2] 3.0837 0.0885
##  $ : num [1:2] 3.0865 0.0972
##  $ : num [1:2] 3.0995 0.0964
```

Since the normalisation factors are affect the mean read number in a sample, they affect the identifiability of the expression levels and the deagradation rates.

Gene-specific parameters

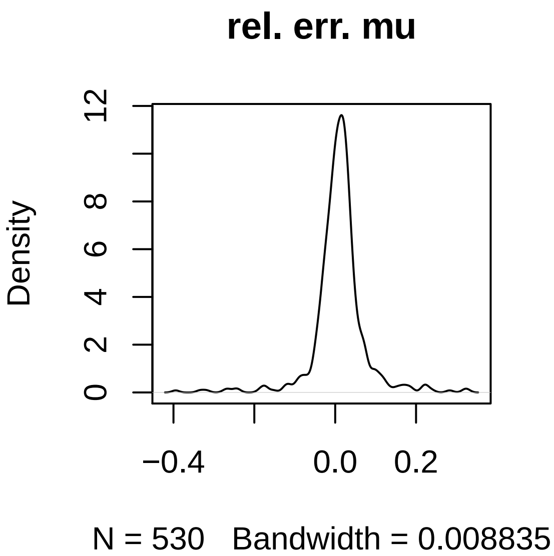
The estimated read mean numbers are well consistent with the raw simulated data:

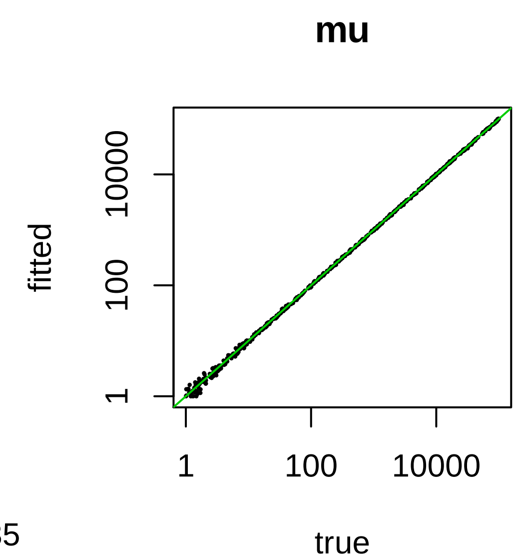
```r
# get the expected read number for every data point in counts
pr <- predictExpression(pd, res)
plot(x = as.vector(pr$predictions)+.5, y = as.vector(pd$counts)+.5,
     pch = 16, cex = .3, log = 'xy',
     xlab = "predicted", ylab = "experiment",
     main = "Counts vs. means")
```



Due to high number of total fraction samples, the expression level μ is also well recovered from the data:

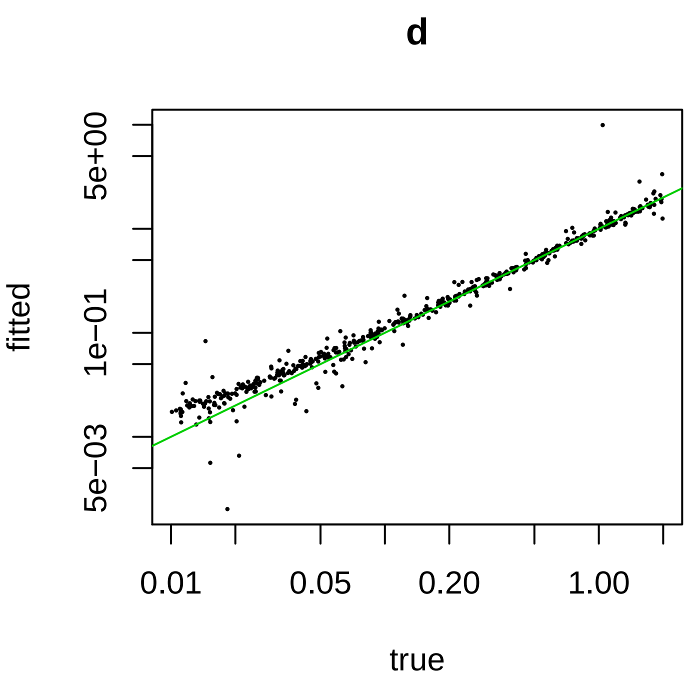
```r
plot(density(1 - res$mu/par$mu), main="rel. err. mu")
plot(x = par$mu, y = res$mu,
     pch = 16, cex = .3, log = 'xy',
     xlab = "true", ylab = "fitted", main = "mu")
abline(0, 1, col = 3)
```





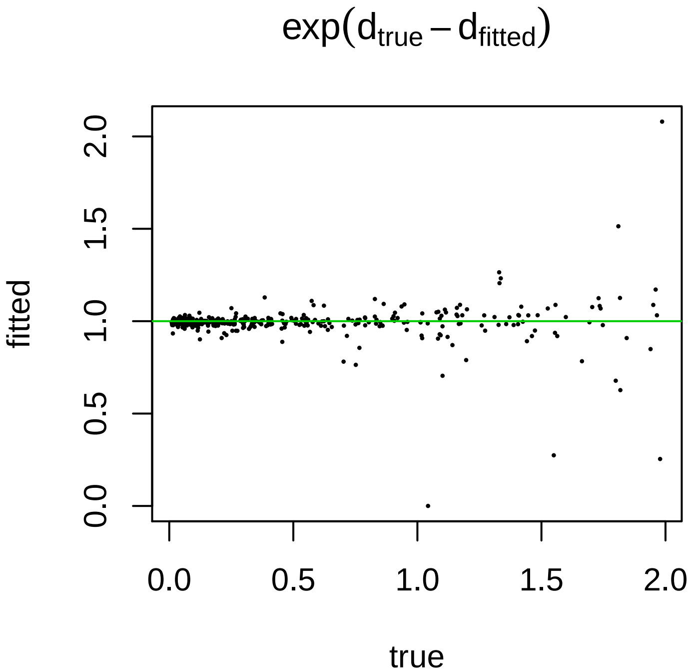
The estimation of the degradation rate is based on the less number of samples (the total fractions provide the information only about the expression level). Hence, the quality of the fit is worse, than for the μ parameter. We see that the error is higher for less degrading genes:

```
plot(x = par$d, y = res$d,
     pch = 16, cex = .3, log = 'xy',
     xlab = "true", ylab = "fitted", main = "d")
abline(0, 1, col = 3)
```



However, the misfit in d for lowly degrading genes has a minor effect on the error in predictions of the read number in comparison to the misfit in the higher ones, because it has an exponential impact:

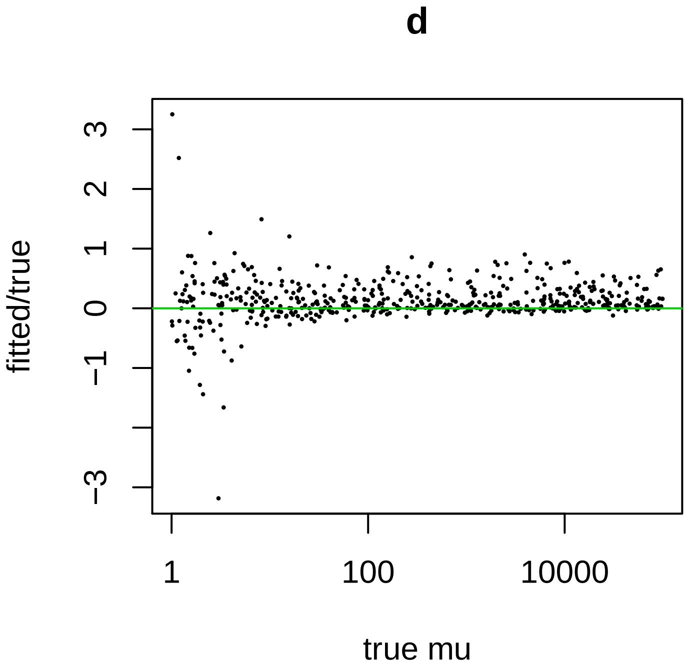
```r
plot(x =par$d, y = exp(par$d-res$d),
     pch = 16, cex = .3,
     xlab = "true", ylab = "fitted",
     main = expression(exp(d[true]-d[fitted])))
abline(1, 0, col = 3)
```



Hence, the genes with the higher turn-over get a higher weight during optimisation of the degradation rate for the case of this simulated data.

As mentioned before, the normalisation of the samples has an impact on the gene-specific parameters. In our experiment, the degradation rates are over-estimated for the majority of the genes. This is counter-balanced by a positive bias in the normalisation factors estimation The estimation of d for the less expressed genes are less precise, because the dispersion of the negative binomial random variable increases for lower means:

```r
plot(x = par$mu, y = log2(res$d / par$d),
     pch = 16, cex = .3, log = 'x',
     xlab = "true mu", ylab = "fitted/true", main = "d")
abline(a = 0, b = 0, col = 3)
```




```r
summary(res$d/par$d)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.1100  0.9939  1.0620  1.1440  1.2080  9.5230
```

Conclusion

Here we presented a validation of the pulseR package performance on the simulated data. As a result, we show that the values of the gene-specific parameters can be recovered by the maximum likelihood estimation implemented in the package. The quality of the estimations depends directly on the data, i.e. on the amount of information and noise in the read numbers.

The pulseR package enables to estimate parameters even
in the absence of spike-ins. However, this approach is limited to the systems, which has structurally and practically identifiable parameters. The identifiability is defined by the number of replicates, the type of the fractions provided, the cross-contamination rates and the sequence depth (read numbers per feature).