subSeq Package Vignette (Version 1.4.0)

David G. Robinson and John D. Storey

October 17, 2016

Contents

1 Introduction

This is a vignette for the `subSeq` package, which performs subsampling of sequencing data to test the effect of depth on your conclusions. When you use a RNA-Seq differential expression method, such as edgeR or DESeq2, you can answer a couple of biological questions:

1. What genes are differentially expressed?
2. What classes of genes show differential expression?

However, what if we're interested in questions of experimental design:

1. Do I have enough reads to detect most of the biologically relevant differences?
2. If I run a similar experiment, should I run additional lanes, or can I multiplex even more samples and work with fewer reads?

One way to help answer these questions is to pretend you have *fewer* reads than you do, and to see how your results (the number of significant genes, your estimates of their effects, and so on) change. If you can achieve the same results with just 10% of your reads, it indicates that (when using your particular analysis method to answer your particular question) the remaining 90% of the reads added very little. In turn, if your conclusions changed considerably between 80% and 100% of your reads, it is likely they would change more if you added additional reads.

`subSeq` is also designed to work with *any analysis method you want*, as long as it takes as input a matrix of count data per gene. `edgeR`, `DESeq2`, `voom` and `DEXSeq` are provided as default examples, but if you prefer to use a different package, to use these packages with a different set of options, or even to use your own method, it is very easy to do so and still take advantage of this package's subsampling methods. See **??** for more.

2 Quick Start Guide

We demonstrate the use of this method on a subset of the Hammer et al 2010 dataset **?**. This dataset (as provided by the ReCount database **?**), comes built in to the `subSeq` package:

```
library(subSeq)
data(hammer)
```

Alternatively we could have downloaded the RNA-Seq counts directly from the ReCount database:

```
load(url("http://bowtie-bio.sourceforge.net/recount/ExpressionSets/hammer_eset.RData"))
hammer = hammer.eset
```

We then filter such that no genes had fewer than 5 reads:

```
hammer.counts = Biobase::exprs(hammer)[, 1:4]
hammer.design = Biobase::pData(hammer)[1:4, ]
hammer.counts = hammer.counts[rowSums(hammer.counts) >=
    5, ]
```

First we should decide which proportions to use. In general, it's a good idea to pick subsamplings on a logarithmic scale, since there tends to be a rapid increase in power for low read depths that then slows down for higher read depths.

```
proportions = 10^seq(-2, 0, 0.5)
proportions

## [1] 0.01000000 0.03162278 0.10000000 0.31622777
## [5] 1.00000000
```

The more proportions you use for subsampling, the greater your resolution and the better your understanding of the trend. However, it will also take longer to run. We give these proportions, along with the matrix and the design, to the `subsample` function, which performs the subsampling:

```
subsamples = subsample(hammer.counts, proportions,
    method = c("edgeR", "voomLimma"), treatment = hammer.design$protocol)
```

In this command, we are giving it a count matrix (`hammer.counts`), a vector of proportions to sample, and a vector describing the treatment design (`hammer.design$protocol`), and telling it to try two methods (edgeR and voom/limma) and report the results of each. (If there are a large number of proportions, this step may take a few minutes. If it is being run interactively, it will show a progress bar with an estimated time remaining).

2.1 The `subsamples` object

The `subsample` function returns a `subsamples` object. This also inherits the `data.table` class, as it is a table with one row *for each gene, in each subsample, with each method*:

```
options(width = 40)
subsamples

##            method proportion
##      1:     edgeR       0.01
##      2:     edgeR       0.01
##      3:     edgeR       0.01
##      4:     edgeR       0.01
##      5:     edgeR       0.01
##     ---
## 150206: voomLimma       1.00
## 150207: voomLimma       1.00
## 150208: voomLimma       1.00
## 150209: voomLimma       1.00
## 150210: voomLimma       1.00
##         replication coefficient
##      1:           1   0.0000000
##      2:           1   0.0000000
##      3:           1   0.0000000
##      4:           1   2.3005438
##      5:           1   0.0000000
##     ---
## 150206:           1  -0.4907982
## 150207:           1   0.9883726
## 150208:           1   2.7152540
## 150209:           1  -1.0073941
## 150210:           1  -0.5339148
##              pvalue                 ID
##      1: 1.000000000 ENSRNOG00000000001
##      2: 1.000000000 ENSRNOG00000000007
##      3: 1.000000000 ENSRNOG00000000008
##      4: 1.000000000 ENSRNOG00000000010
##      5: 1.000000000 ENSRNOG00000000012
##     ---
## 150206: 0.010546926 ENSRNOG00000043503
## 150207: 0.126636226 ENSRNOG00000043504
## 150208: 0.002538513 ENSRNOG00000043507
## 150209: 0.103295778 ENSRNOG00000043512
## 150210: 0.012900077 ENSRNOG00000043513
##         count    depth      qvalue
```

```
##      1:     0   198714 1.000000000
##      2:     0   198714 1.000000000
##      3:     0   198714 1.000000000
##      4:     1   198714 1.000000000
##      5:     0   198714 1.000000000
##     ---
## 150206:  1404 19862441 0.009464488
## 150207:    17 19862441 0.065719919
## 150208:     6 19862441 0.003240602
## 150209:    12 19862441 0.056127854
## 150210:   430 19862441 0.011053328
```

The fields it contains are:

- `coefficient`: The estimated effect size at each gene. For differential expression between two conditions, this is the log2 fold-change.
- `pvalue`: The p-value estimated by the method
- `ID`: The ID of the gene, normally provided by the rownames of the count matrix
- `count`: The number of reads for *this particular gene* at this depth
- `depth`: The total depth (across all reads) at this level of sampling
- `replication`: Which subsampling replication (in this subsampling run, only one replicate was performed; use the `replications` argument to `subsample` to perform multiple replications)
- `method`: The name of the differential expression method used
- `qvalue`: A q-value estimated from the p-value distribution using the `qvalue` package. If you consider all genes with a q-value below q as candidates, you expect to achieve a false discovery rate of q.

2.2 Summarizing Output by Depth

This object gives the differential analysis results per gene, but we are probably more interested in the results per sampling depth: to see how the overall conclusions change at lower read depths. Performing `summary` on the subsamplings object shows this:

```
subsamples.summary = summary(subsamples)
subsamples.summary
```

```
##        depth proportion    method
##  1:   198714 0.01000000     edgeR
```

```
##  2:   198714 0.01000000 voomLimma
##  3:   628221 0.03162278     edgeR
##  4:   628221 0.03162278 voomLimma
##  5:  1985775 0.10000000     edgeR
##  6:  1985775 0.10000000 voomLimma
##  7:  6283463 0.31622777     edgeR
##  8:  6283463 0.31622777 voomLimma
##  9: 19862441 1.00000000     edgeR
## 10: 19862441 1.00000000 voomLimma
##     replication significant   pearson
##  1:           1         193 0.4600407
##  2:           1           0 0.5078410
##  3:           1         797 0.5799243
##  4:           1           0 0.6209718
##  5:           1        2397 0.6974153
##  6:           1        1440 0.7448685
##  7:           1        5170 0.8370498
##  8:           1        5036 0.8737625
##  9:           1        8273 1.0000000
## 10:           1        8614 1.0000000
##      spearman concordance       MSE
##  1: 0.4766101   0.3642695 2.4171128
##  2: 0.5046149   0.4968303 0.9057424
##  3: 0.6093349   0.5148022 1.5551137
##  4: 0.6275447   0.6164827 0.7055174
##  5: 0.7396629   0.6663734 0.9491185
##  6: 0.7494845   0.7443483 0.4748395
##  7: 0.8750287   0.8288772 0.4352593
##  8: 0.8775711   0.8737163 0.2379659
##  9: 1.0000000   1.0000000 0.0000000
## 10: 1.0000000   1.0000000 0.0000000
##          estFDP        rFDP    percent
##  1: 0.006055901 0.010362694 0.02687491
##  2: 0.000000000 0.000000000 0.00000000
##  3: 0.005464025 0.006273526 0.10121406
##  4: 0.000000000 0.000000000 0.00000000
##  5: 0.012841959 0.013350021 0.28993503
##  6: 0.020521426 0.024305556 0.16744131
##  7: 0.026275300 0.024177950 0.61062697
##  8: 0.040583880 0.042494043 0.56154652
##  9: 0.051009112 0.000000000 1.00000000
## 10: 0.050390966 0.000000000 1.00000000
```

As one example: the first row of this summary table indicates that at a read depth of 198714, edgeR found 193 genes significant (at an FDR of .05), and that its fold change estimates had a 0.4766101 Spearman correlation and a 0.4600407

Pearson correlation with the full experiment.

Note that different methods are compared *on the same subsampled matrix* for each depth. That is, at depth 198714, results from `edgeR` are being compared to results from `voomLimma` on the same matrix.

2.2.1 Estimating False Discovery Proportion

One question that `subSeq` can answer is whether decreasing your read depth ever *introduces* false positives. This would be a worrisome sign, as it suggests that *increasing* your read depth even more might prove that you are finding many false positives. `subSeq` has two ways of estimating the false discovery proportion at each depth, each of which requires choosing an FDR threshold to control for (default 5%):

- `eFDP`: The estimated false discovery proportion at each depth when controlling for FDR. This is found by calculating the local false discovery rate of each gene in the oracle, (which we will call the "oracle lFDR"), then finding the mean oracle lFDR of the genes found significant at each depth. This is effectively using the best information available (the full, oracle experiment) to estimate how successful your FDR control is at each depth. This estimate will converge to the desired FDR threshold (e.g. 5%) at the full depth.

- `rFDP`: the "relative" false discovery proportion at each depth, where a relative false discovery is defined as a gene found significant at a subsampled depth that was not found significant at the full depth. This estimate will converge to 0% at the full depth.

Generally, we recommend the `eFDP` over the `rFDP` (and the `eFDP` is the default metric plotted), as a) it is a less noisy estimate, b) It converges to 5% rather than dropping back down to 0, and c) it takes into account *how* unlikely each hypothesis appeared to be in the oracle, rather than simply the binary question of whether it fell above a threshold. We report the `rFDP` only because it a simpler metric that does not require the estimation of local FDR.

In either case, it is important to note that the full experiment is not perfect and has false positives and negatives of its own, so these metrics should not be viewed as an absolute false discovery rate (i.e., "proof" that an experiment is correct). Instead, they examine whether decreasing read depth tends to introduce false results.

3 Plotting

The best way to understand your subsampling results intuitively is to plot them. You can do this by plotting the summary object (not the original subsamples). `plot(subsamples.summary)` creates Figure **??**, which plots some useful metrics of quality by read depth.

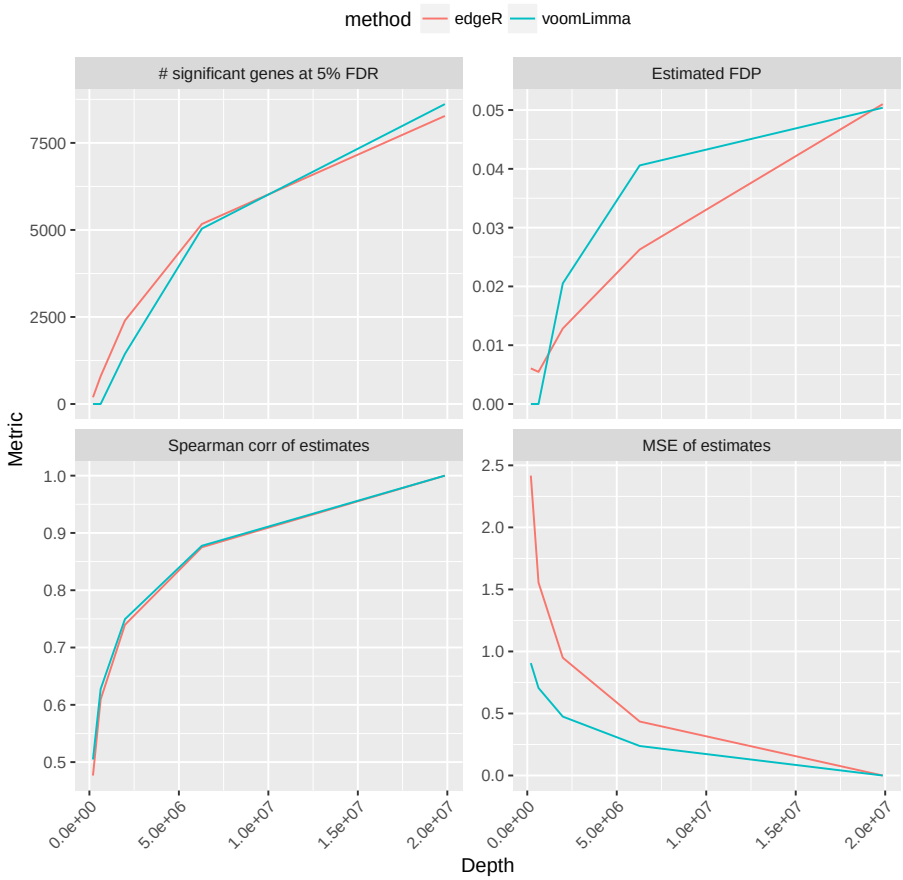
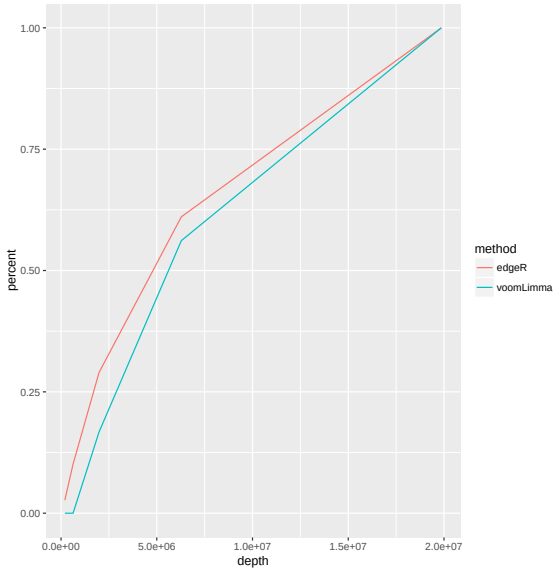


Figure 1: The default output of `plot(subsamples.summary)`. This shows four plots: (i) The number of significant genes found significant at each depth; (ii) the estimated false discovery at this depth, calculated as the average oracle local FDR of the genes found significant at this depth; (iii) The Spearman correlation of estimates at each depth with the estimates at the full experiment; (iv) the mean-squared error of the estimates at each depth with the estimates at the full experiment.

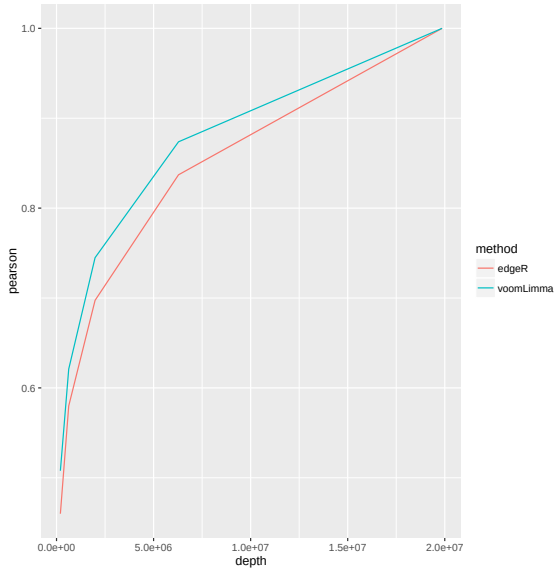
Creating your own plot by read depth from the summary object is easy using ggplot2. For instance, if you would like to plot the percentage of oracle (full experiment) genes that were found significant at each depth, you could do:

```
library(ggplot2)
ggplot(subsamples.summary, aes(x = depth,
    y = percent, col = method)) + geom_line()
```



If you'd like to focus on the Pearson correlations of the estimates, you could do:

```
ggplot(subsamples.summary, aes(x = depth,
    y = pearson, col = method)) + geom_line()
```



4 Writing Your Own Analysis Method

There are five methods of RNA-Seq differential expression analysis built in to `subSeq`:

- `edgeR` Uses edgeR's exact negative binomial test to compare two conditions **?**
- `edgeR.glm` Uses edgeR's generalized linear model, which allows fitting a continuous variable and including covariates.
- `voomLimma` Uses `voom` from the `edgeR` package to normalize the data, then apply's limma's linear model with empirical Bayesian shrinkage **?**.
- `DESeq2` Applies the negative binomial Wald test from the `DESeq2` package **?**. DESeq2 is notable for including shrinkage on the parameters.
- `DEXSeq` Uses a negative binomial generalized linear model to look for differential exon usage in mapped exon data (see **??**) **?**.

These handlers are provided so you can perform subSeq analyses "out-of-the-box". However, it is very likely that your RNA-Seq analysis does not fit into one of these methods! It may use different published software, include options you need to set manually, or it could be a method you wrote yourself. No problem: all you need to do is write a function, called a *handler*, that takes in a count matrix and any additional options, then performs your analysis and returns the result as a data.frame.

Start by writing up the code you would use to analyze a count matrix to produce (i) coefficients (such as the log fold-change) and (ii) p-values. But put your code inside a function, and have the function take as its first argument the (possibly subsampled) count matrix, and return the results as a two-column data.frame with one row per gene (in the same order as the matrix):

```
myMethod = function(count.matrix, treatment) {
    # calculate your coefficients based on the input count matrix
    coefficients = <some code>
    # calculate your p-values
    pvalues = <some more code>

    # return them as a data.frame
    return(data.frame(coefficient=coefficients, pvalue=pvalues))
}
```

Your function can be as long or complicated as you want, as long as its first argument is a `count.matrix` and it returns a data frame with `coefficient` and `pvalue` columns.

Now that you've defined this function, you can pass its name to `subsample` just like you would one of the built-in ones:

```
subsamples = subsample(hammer.counts, proportions,
    method = c("edgeR", "DESeq2", "myMethod"),
    treatment = hammer.design$protocol)
```

4.1 Advanced Uses of Handlers

Your handler can do more than take these two arguments and calculate a p-value:

- Your handler can return more columns than a coefficient and p-value (for example, you can return a column of dispersion estimates, confidence interval boundaries, or anything else on which you would like to examine the effect of depth). These columns will be included in the output.

  If you provide multiple methods, including one that returns an extra columns and one that doesn't, the column will be filled with NA for the method that doesn't provide the column.

- Your handler can take more arguments than just a count matrix and treatment vector. Any arguments passed at the end of the 'subsample' function will be passed on to it. *However*, you cannot use multiple methods where some take extra arguments and some do not. Instead, perform them separately (using the same seed; see **??**) and then use `combineSubsamples`.

- Your handler does not necessarily have to return one row per gene. For example, you could have a handler that performs gene set enrichment analysis (GSEA) on your data, which then returns one gene set per row. If so, it must return an additional column: `ID` (those were otherwise acquired from the rownames of the count matrix). Note that `count` column (noting the read depth per gene) will become NA unless it is provided by the handler function.

5 Reproducing or Adding to a Previous Run

If you have a subsamples or subsample summary object that either you or someone else ran, you might be interested in working with the original data further. For example:

- You may want to perform additional methods, and be able to compare them directly to the original
- You may want to sample additional depths, and combine them together
- You may have noticed a strange discrepancy in one of the results, and want to further examine the count matrix at that specific depth

All of these are possible, as **subSeq** stores the random seed used for the run in the results.

5.1 Adding Methods or Depths

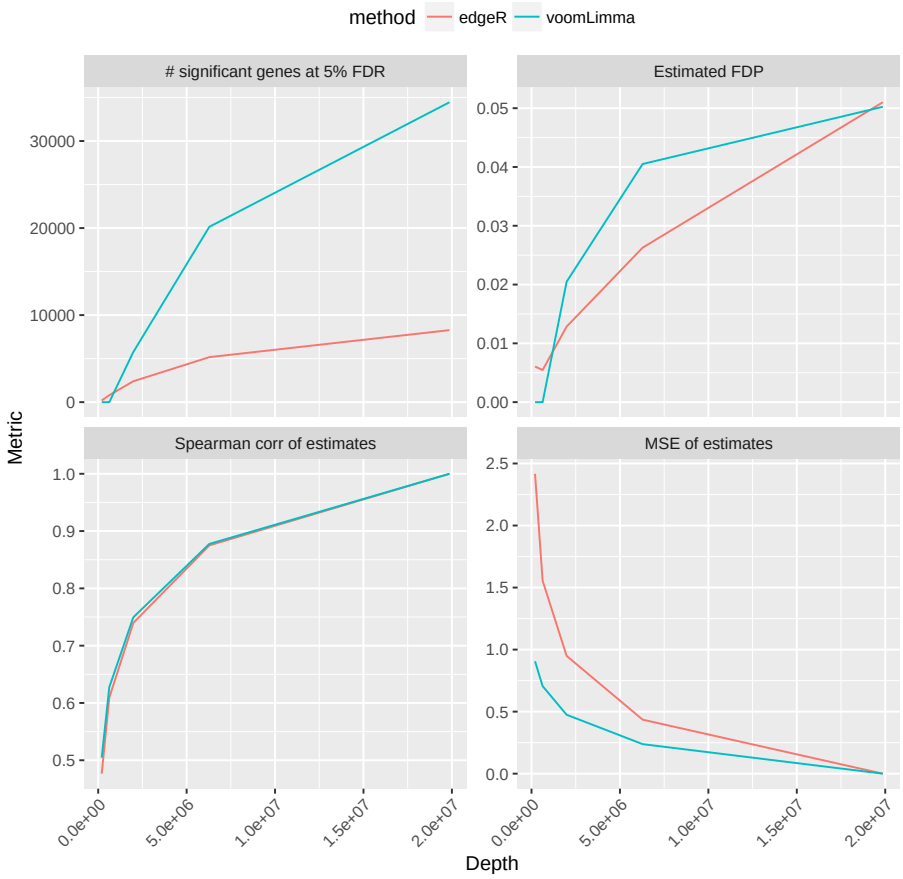
If you already have some methods performed (as we do in `subsamples`), we may want to add additional methods. For example you might want to analyze the same depths with the voom method. To do this, use the `getSeed` function to retrieve the random seed from the subsamples results, and then provide that seed to the `subsample` function:

```
seed = getSeed(subsamples)

subsamples.more = subsample(hammer.counts,
    proportions, method = c("voomLimma"),
    treatment = hammer.design$protocol, seed = seed)
```

After that, you can combine the two objects using `combineSubsamples`:

```
subsamples.combined = combineSubsamples(subsamples,
    subsamples.more)
plot(summary(subsamples.combined))
```



5.2 Examining a Matrix More Closely

Say that after your analysis, you are surprised that a particular method performed how it did at a particular depth, and you wish to examine that depth further.

You can also use the seed to retrieve the precise matrix used. For instance, if we want to find the matrix used for the .1 proportion subsampling:

```
submatrix = generateSubsampledMatrix(hammer.counts,
    0.1, seed = seed)
dim(submatrix)

## [1] 15021     4

sum(submatrix)

## [1] 1985775
```

6 Note on Subsampling

subSeq performs read subsampling using a random draw from a binomial distribution. Specifically, for a subsampling proportion p_k, each value m, n in the subsampled matrix $Y^{(k)}$ is generated as:

$$Y^{(k)}_{m,n} \sim \text{Binom}(X_{m,n}, p_k)$$

This is equivalent to allowing each original mapped read to have probability p_k of being included in the new counts, as done, for example, by the Picard DownsampleSam function (**?**).

A computationally intensive alternative is to sample the reads *before* they are mapped, then to perform mapping on the sampled data. When the mapping is done independently and deterministically for each read, as it is e.g. in Bowtie, this is mathematically identical to subsampling the aligned reads or counts, since the inclusion of one read does not affect the mapping of any other. This applies to methods such as Bar-Seq and Tn-Seq, or for RNA-Seq in organisms with no or very few introns (e.g. bacteria or yeast). Note that some spliced read mappers, such as TopHat, do not perform all their mappings entirely independently, since unspliced mappings are first used to determine exonic regions before searching for spliced reads. However, those methods are (not coincidentally) among the most computationally intensive mappers in terms of running time and memory usage, giving subSeq an especially large advantage.

Several other papers that perform subsampling have made the assumption of independence implicitly, but if you are unsure, it may be useful to subsample the original fastq file, then perform the read mapping on each sample individually, and compare the subsampled counts to those from `subSeq` to check that this approximation is reasonable for your data.