BEARscc: Using spike-ins to assess single cell cluster robustness

David T. Severson

19 May 2021

Contents

1 Introduction

1.1 Scope

Single-cell transcriptome sequencing data are subject to substantial technical variation and batch effects that can confound the classification of cellular sub-types. Unfortunately, current clustering algorithms do not account for this uncertainty. To address this shortcoming, we have developed a noise perturbation algorithm called **BEARscc** that is designed to determine the extent to which classifications by existing clustering algorithms are robust to observed technical variation.

BEARscc makes use of spike-in measurements to model technical variance as a function of gene expression and technical dropout effects on lowly expressed genes. In our benchmarks, we found that BEARscc accurately models read count fluctuations and drop-out effects across transcripts with diverse expression levels. Applying our approach to publicly available single-cell transcriptome data of mouse brain and intestine, we have demonstrated that BEARscc identified cells that cluster consistently, irrespective of technical variation. For more details, see the manuscript on bioRxiv.

Importantly, **BEARscc** should not be considered another clustering algorithm. Specifically, this package is designed to supply users with an organic tool to evaluate and explore the impact of noise on their single cell cluster interpretations. The package provides users with a way to clarify the precision of a single cell experiment with respect to grouping cells into clusters that are biologically meaningful. In this way, **BEARscc** allows users to achieve confidence in clusters and relevant cells that consistently cluster together invariant to the noise inherent to a single cell experiment. Conversely, the algorithm provides a mechanism to identify cells and clusters which cannot be resolved given the precision of the experiment in conjuction with the clustering algorithm of choice. It is our hope that **BEARscc** will enable users to proceed with clusters, or biological groups, in which they are confident are robust to noise and in which they have an intimate understanding of those cells and clusters that may be less precisely assigned to the putative biological role.

1.2 Installation

BEARscc is now available on Bioconductor and can be installed using the syntax below.

```
if (!requireNamespace("BiocManager", quietly=TRUE))
    install.packages("BiocManager")
BiocManager::install("BEARscc")
```

1.3 Citation

BEARscc and its associated manuscript are currently under review for publication at a peer-reviewed journal. For now, please cite the bioRxiv pre-print:

```
Severson, DT. Owen, RP. White, MJ. Lu, X. Schuster-Boeckler, B.
BEARscc determines robustness of single-cell clusters using simulated
technical replicates. doi: https://doi.org/10.1101/118919
```

2 Tutorial

2.1 Overview

BEARscc relies upon spike-in count measurements in single-cell transcriptome experiments to estimate experimental noise and produce simulated technical replicates to provide a quantitative understanding of the robustness of proposed single cell cluster labels to experimental noise. In principal, the algorithm is compatible with any clustering algorithm. The following should provide users with a comprehensive tutorial of the use and utlity of BEARscc as a tool for vetting single cell clusters with respect to experimental noise.

Before getting started, we need to load some example single cell data.BEARscc is equipped with a set of sample data for the purpose of testing functions, examples in help files, and this nifty tutorial. The data may be loaded as follows:

```
library("BEARscc")
data("BEARscc_examples")
```

The loaded file `BEARscc_examples` is equipped with separate `data.frame` objects including ERCC spike-in observations (`ERCC.counts.df`), endogenous count observations (`data.counts.df`), and the expected or actual spike-in concentrations (`ERCC.meta.df`) as well as a `SingleCellExperiment` object that contains all of the above seperate components (`BEAR_examples.sce`) as shown below:

```
head(ERCC.counts.df[,1:2])
#>            WTCHG_217386_229230 WTCHG_217386_249229
#> ERCC-00002                 629                 803
#> ERCC-00003                  13                  27
#> ERCC-00004                  61                  49
#> ERCC-00009                 183                 202
#> ERCC-00013                   0                   0
#> ERCC-00019                   0                   0

head(data.counts.df[,1:2])
#>                 WTCHG_217386_229230 WTCHG_217386_249229
#> ENSG00000000003                   0                   0
#> ENSG00000000419                   0                   0
#> ENSG00000000457                   0                   1
#> ENSG00000000460                   1                  37
#> ENSG00000000938                   0                   0
#> ENSG00000000971                   0                   0

head(ERCC.meta.df)
#>             Transcripts
#> ERCC-00002 3.011070e+02
#> ERCC-00003 1.881919e+01
#> ERCC-00004 1.505535e+02
#> ERCC-00009 1.881919e+01
#> ERCC-00012 2.297264e-03
#> ERCC-00013 1.837812e-02

BEAR_examples.sce
```

```
#> class: SingleCellExperiment
#> dim: 174 50
#> metadata(1): spikeConcentrations
#> assays(2): counts observed_expression
#> rownames(174): ENSG00000000003 ENSG00000000419 ... ERCC-001701
#>   ERCC-001711
#> rowData names(0):
#> colnames(50): WTCHG_217386_229230 WTCHG_217386_249229 ...
#>   WTCHG_230414_256254 WTCHG_230414_256278
#> colData names(0):
#> reducedDimNames(0):
#> mainExpName: NULL
#> altExpNames(1): ERCC_spikes
```

In the event we were working with a new set of data, the spike-in concentrations `data.frame` can be computed from industry reported concentrations and the relevant dilution protocol utilized in the experiment. Count tables would need to be mapped and counted with preferred software, and the spike-in control counts (ERCC or otherwise) would need to be identified from the endogenous counts.

Below is how one would create a `SingleCellExperiment` object from spike-in count, endogenous count, and spike-in concentration `data.frame` objects. Note how we create the `observed_expression` assay object in the following code. This object is essential in that all estimation and simulation of replicates occurs assuming these are the observed counts (normalized or otherwise). Without them BEARscc will throw an error indicating that `"observed_expression"` not in `names(assays(<SingleCellExperiment>))`. Also, `data.counts.df` in this example includes both spike-in genes and endogenous genes. In general, we recommend spike-in genes be simulated with endogenous genes as a control, and this will be done by default when the `SingleCellExperiment` object is used.

```
BEAR.se <- SummarizedExperiment(list(counts=as.matrix(data.counts.df)))
BEAR_examples.sce<-as(BEAR.se, "SingleCellExperiment")
metadata(BEAR_examples.sce)<-list(spikeConcentrations=ERCC.meta.df)
assay(BEAR_examples.sce, "observed_expression")<-counts(BEAR_examples.sce)
altExp(BEAR_examples.sce, "ERCC_spikes")<-BEAR_examples.sce[grepl("^ERCC-",
    rownames(BEAR_examples.sce)),]
```

Running the above code should yield a `SingleCellExperiment` object identical to the one that loads with `data("BEARscc_examples")`.

2.2 Building the noise model

We will now estimate the single-cell noise present in the experiment using spike-in controls. In this tutorial, we rely upon a subsample of artificial control data found in `BEARscc_examples`; however, users are encouraged to work through the tutorial with their own single cell data provided some form of spike-ins were included in the experiment. Building the noise models with BEARscc is relatively straightforward with `estimate_noiseparameters()`. We simply provide the function with the now adequately annotated `SingleCellExperiment` object, `BEAR_examples.sce`. Here, the parameter 'alpha_resolution is set to 0.1 to speed things along, but we suggest values between 0.001 and 0.01 be used in real applications of BEARscc.

```
BEAR_examples.sce <- estimate_noiseparameters(BEAR_examples.sce,
    max_cumprob=0.9999, alpha_resolution = 0.1, bins=10,
    write.noise.model=FALSE, file="BEAR_examples")
#> [1] "Fitting parameter alpha to establish spike-in derived noise model."
#> [1] "Estimating error for spike-ins with alpha = 0"
#> [1] "Estimating error for spike-ins with alpha = 0.5"
#> [1] "Estimating error for spike-ins with alpha = 1"
#> [1] "There are adequate spike-in drop-outs to build the drop-out model. Estimating the drop-out model now.
```

Several options exist for `estimate_noiseparameters()`. These are fully documented in the help page `?estimate_noiseparameters`.

2.3 Simulating technical replicates

Following estimation of noise, the parameters computed are then used to generate a simulate a technical replicate. Here we will simulate replicates on our local computer, but frequently users will want to utilize the methods described in Section 2.4. Notably the necessary parameters are conveniently stored in the `metadata` of our `SingleCellExpression` object, `BEAR_examples.sce` following noise estimation, and so we simply run:

```
    BEAR_examples.sce <- simulate_replicates(BEAR_examples.sce,
        max_cumprob=0.9999, n=10)
#> [1] "Creating a simulated replicated counts matrix: 1."
#> [1] "Creating a simulated replicated counts matrix: 2."
#> [1] "Creating a simulated replicated counts matrix: 3."
#> [1] "Creating a simulated replicated counts matrix: 4."
#> [1] "Creating a simulated replicated counts matrix: 5."
#> [1] "Creating a simulated replicated counts matrix: 6."
#> [1] "Creating a simulated replicated counts matrix: 7."
#> [1] "Creating a simulated replicated counts matrix: 8."
#> [1] "Creating a simulated replicated counts matrix: 9."
#> [1] "Creating a simulated replicated counts matrix: 10."
```

Recall that `BEAR_examples.sce` is our `SingleCellExperiment` object that we recently annoated with model parameters describing experimental noise using the function `estimate_noiseparameters()` and note that the variable `n` is the desired number of simulated technical replicates (e.g. 10). Finally, the `maxcum_prob` is identical to its use in the noise estimation. If the user deviated from the default parameter, it is highly recommended that this value be identical to the value utilized in `estimate_noiseparmaters()` The resulting object is a list, where each element is a simulated technical replicate, and one element is the original counts matrix.

2.4 Simulation of replicates for larger datasets

For larger datasets, we set `write.noise.model=TRUE` when running `estimate_noiseparameters()`and copy the written bayesian drop-out and noise estimate files with the observed counts table to a high performance computing environment. The following code provides an example:

```
BEAR_examples.sce <- estimate_noiseparameters(BEAR_examples.sce,
                                    write.noise.model=TRUE,
```

```
                                        file="tutorial_example",
                                        model_view=c("Observed","Optimized"))
```

After running the above code, then within the current working directory (if unsure use `getwd()`), we should find the two tab-delimited files that together completely describe the BEARscc noise model. These are the parameters describing the mixed model of technical variation (`tutorial_example_parameters4randomize.xls`, see Section 3.1.1) and the parameters describing the drop-out model (`tutorial_example_bayesianestimates.xls`, see Section 3.1.2). In addition, we should find our `"observed_expression"`matrix in the form of `tutorial_example_counts4clusterperturbation.xls`. Note that the `xls` subscript just allows users to quickly open these tab-delimited files in Microsoft Excel if desired, but these can be readily viewed on the terminal or in simple text editors as well.

With the original counts file and noise model prepared, we then copy these files to our high performance compute cluster. The following code provides a sense of how to proceed once these files have been copied to a high performance cluster; however, the job submission structure of each user's environment will dictate the precise syntax for the following procedure.

The script `HPC_generate_noise_matrices` contains an analogous `simulate_replicates()` for a high performance computational node. To utilize these functions for simulating technical replicates on a cluster, please install BEARscc on the relevant cluster. The user should write an R script to load the BEARscc library and run the clustering. The following code provides a suggested format for both calling the R script with a bash job script and the relevant R invocation of BEARscc and may also be found as a stand alone script in `inst/example/HPC_run_example.R`.

Our cluster utilizes a job submission format that interacts seamlessly with bash code; therefore, the `$SGE_TASKID` represents an array id for
jobs to conveniently generate 100 simulated technical replicates in a single job array. In any case, this variable should be treated as the index for the simulated technical replicate as we recommend from experience that users generate 50 to 100 such simulated technical replicates to reach a stable noise consensus matrix solution.

The following bash code could be included in one such job script:

```
Rscript --vanilla HPC_run_example.R $SGE_TASK_ID
```

Noting that the file `HPC_run_example.R` contains the following suggested code to run BEARscc:

```
library("BEARscc")

#### Load data ####
ITERATION<-commandArgs(trailingOnly=TRUE)[1]
counts.df<-read.delim("tutorial_example_counts4clusterperturbation.xls")
#filter out zero counts to speed up algorithm
counts.df<-counts.df[rowSums(counts.df)>0,]
probs4detection<-fread("tutorial_example_bayesianestimates.xls")
parameters<-fread("tutorial_example_parameters4randomize.xls")

#### Simulate replicates ####
counts.error<-HPC_simulate_replicates(counts_matrix=counts.df,
    dropout_parameters=dropout_parameters,
    spikein_parameters=spikein_parameters)
```

```
    write.table(counts.error, file=paste("simulated_replicates/",
        paste(ITERATION,"sim_replicate_counts.txt",sep="_"),
        sep=""), quote =FALSE, row.names=TRUE)
    #########
```

The script generates seperate simulated technical replicate files, which can be loaded into R as a list for clustering or, in the case of more computationally intense clustering algorithms, re-clustered individually in a high performance compute environment.

2.5 Forming a noise consensus

After generating simulated technical replicates, these should be re-clustered using the clustering method applied to the original dataset. For simplicity, here we use hierarchical clustering on a euclidean distance metric to identify two clusters. In our experience, some published clustering algorithms are sensitive to cell order, so we suggest scrambling the order of cells for each noise iteration as we do below in the function, `recluster()`. The function below will serve our purposes for this tutorial, but BEARscc may be used with any clustering algorithm (e.g. SC3, RaceID2, or BackSPIN).

To quickly recluster a list of simulated technical replicates, we define a reclustering function:

```
    recluster <- function(x) {
        x <- data.frame(x)
        scramble <- sample(colnames(x), size=length(colnames(x)), replace=FALSE)
        x <- x[,scramble]
        clust <- hclust(dist(t(x), method="euclidean"),method="complete")
        clust <- cutree(clust,2)
        data.frame(clust)
    }
```

First, we need to determine how the observed data clusters under our algorithm of choice (e.g. `recluster()`, which is a simple hiearchical clustering for illustrative purposes). These are the clusters that can be evaluated by `BEARscc` and compared to `BEARscc` meta-clustering:

```
    OG_clusters<-recluster(data.frame(assay(BEAR_examples.sce,
        "observed_expression")))
    colnames(OG_clusters)<-"Original"
```

We then apply the function `recluster()` to all simulated technical replicates and the original counts matrix and manipulate the list into a `data.frame`.

```
    cluster.list<-lapply(metadata(BEAR_examples.sce)$simulated_replicates,
        `recluster`)
    clusters.df<-do.call("cbind", cluster.list)
    colnames(clusters.df)<-names(cluster.list)
```

Note: If running clustering algorithms on a seperate high performance cluster, the user should retrieve labels and format as a `data.frame` of cluster labels, where the last column must be the original cluster labels derived from the observed count data. As an example, examine the file, inst/example/example_clusters.tsv.

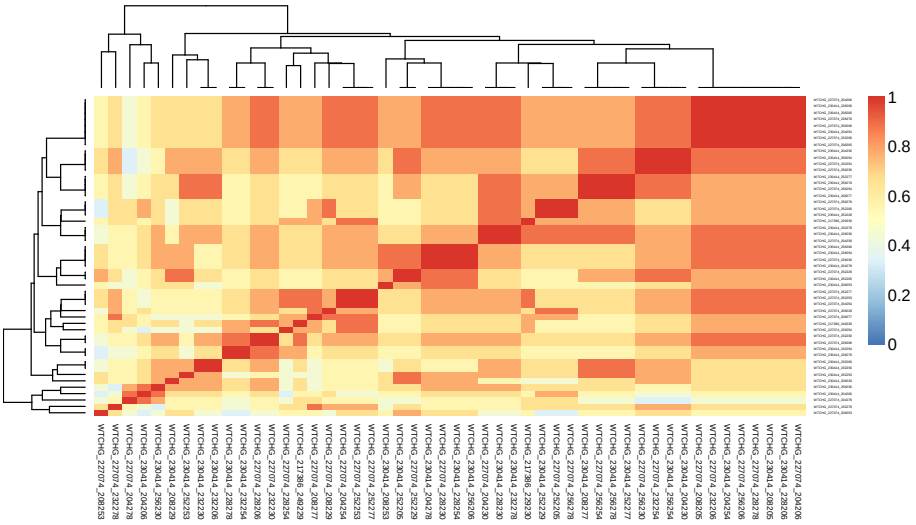
Using the cluster labels file generated by clustering simulated technical replicates on a high performance compute environment or with `recluster()` as described above , we can generate a noise consensus matrix as shown below. An illustrative three rows and columns of an example noise consensus matrix are displayed:

```
noise_consensus <- compute_consensus(clusters.df)
head(noise_consensus[,1:3], n=3)
#>                     WTCHG_230414_256254 WTCHG_217386_229230 WTCHG_230414_204206
#> WTCHG_230414_256254           1.0000000           0.6666667           0.4444444
#> WTCHG_217386_229230           0.6666667           1.0000000           0.5555556
#> WTCHG_230414_204206           0.4444444           0.5555556           1.0000000
```

Using the `aheatmap()` function in the `NMF` library, the consensus matrix result of 10 simulated replicates by BEARscc:

To reproduce the plot run:

```
library("NMF")
#> Loading required package: pkgmaker
#> Loading required package: registry
#>
#> Attaching package: 'pkgmaker'
#> The following object is masked from 'package:S4Vectors':
#>
#>     new2
#> Loading required package: rngtools
#> Loading required package: cluster
#> NMF - BioConductor layer [OK] | Shared memory capabilities [NO: synchronicity] | Cores 23/24
#>   To enable shared memory capabilities, try: install.extras('
#> NMF
#> ')
#>
#> Attaching package: 'NMF'
#> The following object is masked from 'package:S4Vectors':
#>
#>     nrun
aheatmap(noise_consensus, breaks=0.5)
```



Although, 10 simulated replicates is sparse (we recommend 50 to 100), we can already see that these samples likely belong to a single cluster. Indeed, these samples were prepared from a single ground truth of dilute whole tissue brain RNA-seq data from two batches of experimental data. If desired, we could annotate the above heatmap with relevant metadata concerning sample batch, origin, etc.

2.6 Evaluating the noise consensus

In order to further interpret the noise consensus, we have defined three cluster (and analagous cell) metrics. Stability indicates the propensity for a putative cluster to contain the same cells across noise-injected counts matrices. Promiscuity indicates a tendency for cells in a putative cluster to associate with other clusters across noise-injected counts matrices. Score represents the promiscuity substracted from the stability.

We have found it useful to inform the optimal number of clusters in terms of resiliance to noise by examining these metrics by cutting hierarchical clustering dendograms of the noise consensus and comparing the results to the original clustering labels. To do this create a vector containing each number of clusters one wishes to examine (the function automatically determines the results for the dataset as a single cluster) and then cluster the consensus with various cluster numbers using `cluster_consensus()`:

```
vector <- seq(from=2, to=5, by=1)
BEARscc_clusts.df <- cluster_consensus(noise_consensus,vector)
```

We add the original clustering to the `data.frame` to evaluate its robustness as well as the suggested BEARscc clusters:

```
BEARscc_clusts.df <- cbind(BEARscc_clusts.df,
    Original=OG_clusters)
```

2.6.1 Understanding robustness at the cluster level

Now we can compute cluster metrics for each of the BEARscc cluster number scenarious and the original clustering; indeed, any cluster labels of the users choosing could be supplied to vet with the information provided by the noise consensus. We accomplish this by running the command and displaying the first 5 rows of the resulting melted `data.frame`:

```
cluster_scores.df <- report_cluster_metrics(BEARscc_clusts.df,
    noise_consensus, plot=FALSE)
#> Warning in melt(metric_list, id.vars = c("size")): The melt generic in
#> data.table has been passed a list and will attempt to redirect to the relevant
#> reshape2 method; please note that reshape2 is deprecated, and this redirection
#> is now deprecated as well. To continue using melt methods from reshape2 while
#> both libraries are attached, e.g. melt.list, you can prepend the namespace like
#> reshape2::melt(metric_list). In the next version, this warning will become an
#> error.

#> Warning in melt(metric_list, id.vars = c("size")): The melt generic in
#> data.table has been passed a list and will attempt to redirect to the relevant
#> reshape2 method; please note that reshape2 is deprecated, and this redirection
#> is now deprecated as well. To continue using melt methods from reshape2 while
#> both libraries are attached, e.g. melt.list, you can prepend the namespace like
#> reshape2::melt(metric_list). In the next version, this warning will become an
#> error.

#> Warning in melt(metric_list, id.vars = c("size")): The melt generic in
#> data.table has been passed a list and will attempt to redirect to the relevant
#> reshape2 method; please note that reshape2 is deprecated, and this redirection
#> is now deprecated as well. To continue using melt methods from reshape2 while
#> both libraries are attached, e.g. melt.list, you can prepend the namespace like
#> reshape2::melt(metric_list). In the next version, this warning will become an
#> error.

#> Warning in melt(metric_list, id.vars = c("size")): The melt generic in
#> data.table has been passed a list and will attempt to redirect to the relevant
#> reshape2 method; please note that reshape2 is deprecated, and this redirection
#> is now deprecated as well. To continue using melt methods from reshape2 while
#> both libraries are attached, e.g. melt.list, you can prepend the namespace like
#> reshape2::melt(metric_list). In the next version, this warning will become an
#> error.

#> Warning in melt(metric_list, id.vars = c("size")): The melt generic in
#> data.table has been passed a list and will attempt to redirect to the relevant
#> reshape2 method; please note that reshape2 is deprecated, and this redirection
#> is now deprecated as well. To continue using melt methods from reshape2 while
#> both libraries are attached, e.g. melt.list, you can prepend the namespace like
#> reshape2::melt(metric_list). In the next version, this warning will become an
#> error.
#> Warning in melt(cluster.scores, id.var = c("rn", "size", "metric")): The melt
#> generic in data.table has been passed a list and will attempt to redirect to
#> the relevant reshape2 method; please note that reshape2 is deprecated, and
#> this redirection is now deprecated as well. To continue using melt methods from
```

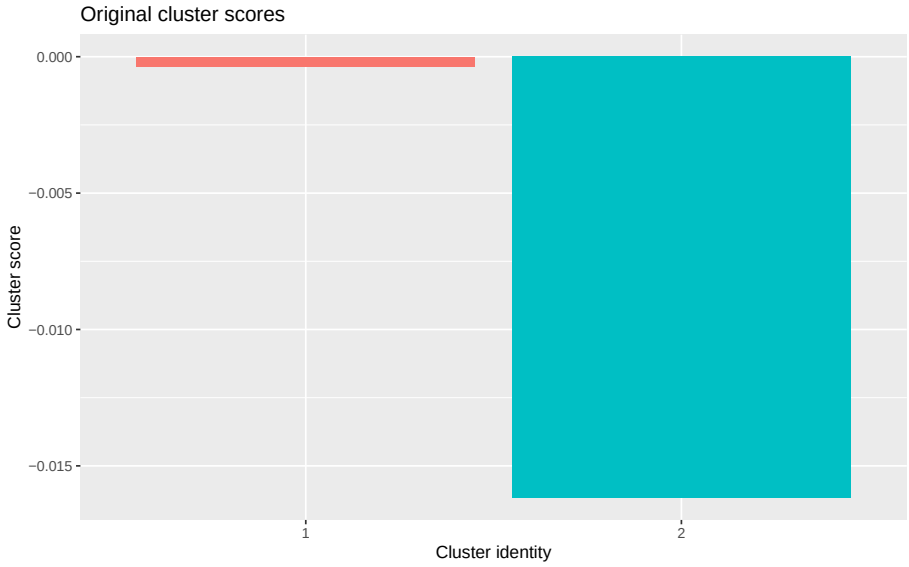
```
#> reshape2 while both libraries are attached, e.g. melt.list, you can prepend the
#> namespace like reshape2::melt(cluster.scores). In the next version, this warning
#> will become an error.
head(cluster_scores.df, n=5)
#>   Cluster.identity Cluster.size      Metric     Value Clustering
#> 1                1           50       Score 0.7142222          1
#> 2                1           50 Promiscuity 0.0000000          1
#> 3                1           50   Stability 0.7142222          1
#> 4                1           49       Score 0.1899093          2
#> 5                2            1       Score 0.2222222          2
#>           Singlet Clustering.Mean
#> 1 Cell number > 1       0.7142222
#> 2 Cell number > 1       0.0000000
#> 3 Cell number > 1       0.7142222
#> 4 Cell number > 1       0.2060658
#> 5 Cell number = 1       0.2060658
```

Above is a melted `data.frame` that displays the name of each cluster, the size of each cluster, the metric (Score, Promiscuity, Stability), the value of each metric for the respective cluster and clustering, the clustering in question (1,2,. . . ,Original), whether the cluster consists of only one cell, and finally the mean of each metric across all clusters in a clustering.

In the previous example, we display all metrics for generating 1 clusters from the data given the previously computed noise consensus from 10 simulated technical replicates and the same for the score metric generating 2 clusters from the data. Importantly, by definition, 1 cluster scenarios have a promiscuity value of 0. Observe that the score for the cluster in the 1 cluster scenario is much larger than either cluster of the 2 cluster scenario, and that this is reflected in the average clustering column.

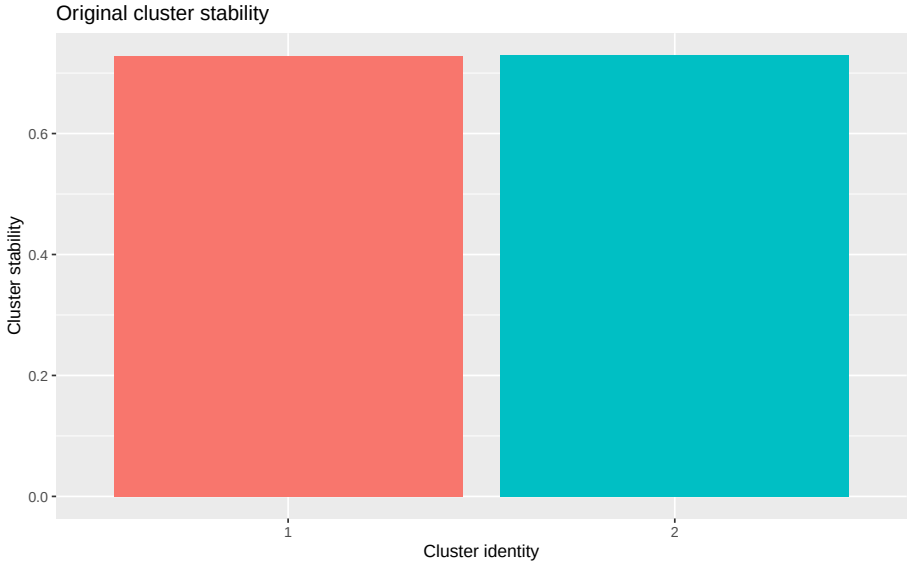
We can examine the BEARcc metrics for the original cluster using `ggplot2` and by subsetting the `data.frame` to the original cluster and the score metric:

```
library("ggplot2")
library("cowplot")
original_scores.df<-cluster_scores.df[
    cluster_scores.df$Clustering=="Original",]
ggplot(original_scores.df[original_scores.df$Metric=="Score",],
    aes(x=`Cluster.identity`, y=Value) )+
    geom_bar(aes(fill=`Cluster.identity`), stat="identity")+
    xlab("Cluster identity")+ylab("Cluster score")+
    ggtitle("Original cluster scores")+guides(fill=FALSE)
```



We can see that this initial clustering is terrible, which makes sense given the ground truth consists of a single biological entity. The stability and promiscuity metrics bear this out:

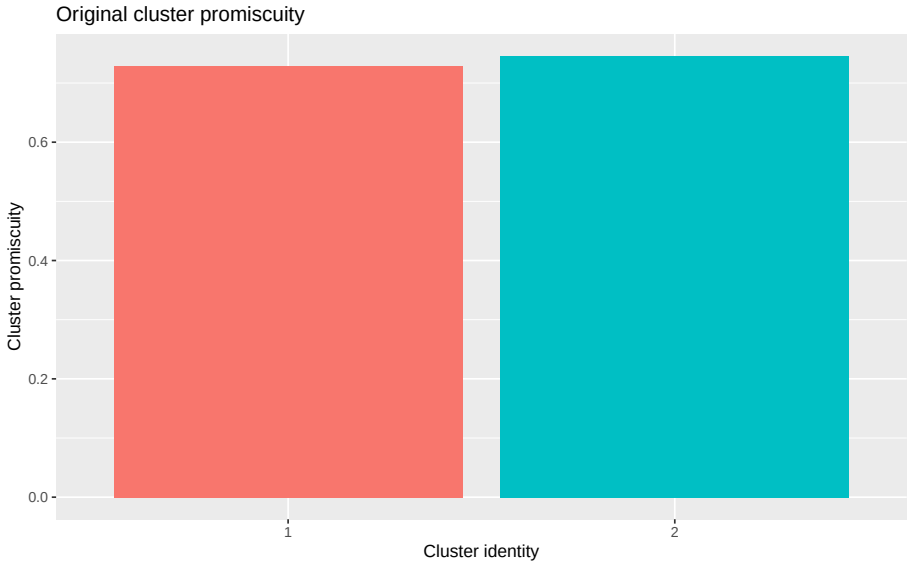
```
ggplot(original_scores.df[original_scores.df$Metric=="Stability",],
    aes(x=`Cluster.identity`, y=Value) )+
    geom_bar(aes(fill=`Cluster.identity`), stat="identity")+
    xlab("Cluster identity")+ylab("Cluster stability")+
    ggtitle("Original cluster stability")+guides(fill=FALSE)
```



The high stability exhibited in the example above is not suprising as samples in this example should have strong association with one another. However, the promuscuity below reveals the reason for the low score:

```
ggplot(original_scores.df[original_scores.df$Metric=="Promiscuity",],
    aes(x=`Cluster.identity`, y=Value) )+
    geom_bar(aes(fill=`Cluster.identity`), stat="identity")+
```

```
        xlab("Cluster identity")+ylab("Cluster promiscuity")+
        ggtitle("Original cluster promiscuity")+guides(fill=FALSE)
```



Despite the high stability, the samples within each cluster have high association with cells in the other cluster, which results in a high promiscuity reported from the noise consensus. As a net result, the scores in the original 2 cluster case for each cluster are subpar. Again, this is consistent with the ground truth of the example data.

2.6.2 Understanding robustness at the sample level

Completely analagous to cluster metrics, the extent to which cells belong within a given cluster may be evaluated with respect to the noise consensus. Below we demonstrate how to compute the cell metrics and display 4 cells for illustrative purposes.

```
cell_scores.df <- report_cell_metrics(BEARscc_clusts.df, noise_consensus)
#> Warning in melt(lapply(cluster_names, calculate_cell_metrics_by_cluster, :
#> The melt generic in data.table has been passed a list and will attempt
#> to redirect to the relevant reshape2 method; please note that reshape2 is
#> deprecated, and this redirection is now deprecated as well. To continue using
#> melt methods from reshape2 while both libraries are attached, e.g. melt.list,
#> you can prepend the namespace like reshape2::melt(lapply(cluster_names,
#> calculate_cell_metrics_by_cluster, consensus_matrix = consensus_matrix,
#> cluster_labels = cluster_labels)). In the next version, this warning will become
#> an error.

#> Warning in melt(lapply(cluster_names, calculate_cell_metrics_by_cluster, :
#> The melt generic in data.table has been passed a list and will attempt
#> to redirect to the relevant reshape2 method; please note that reshape2 is
#> deprecated, and this redirection is now deprecated as well. To continue using
#> melt methods from reshape2 while both libraries are attached, e.g. melt.list,
#> you can prepend the namespace like reshape2::melt(lapply(cluster_names,
#> calculate_cell_metrics_by_cluster, consensus_matrix = consensus_matrix,
#> cluster_labels = cluster_labels)). In the next version, this warning will become
```

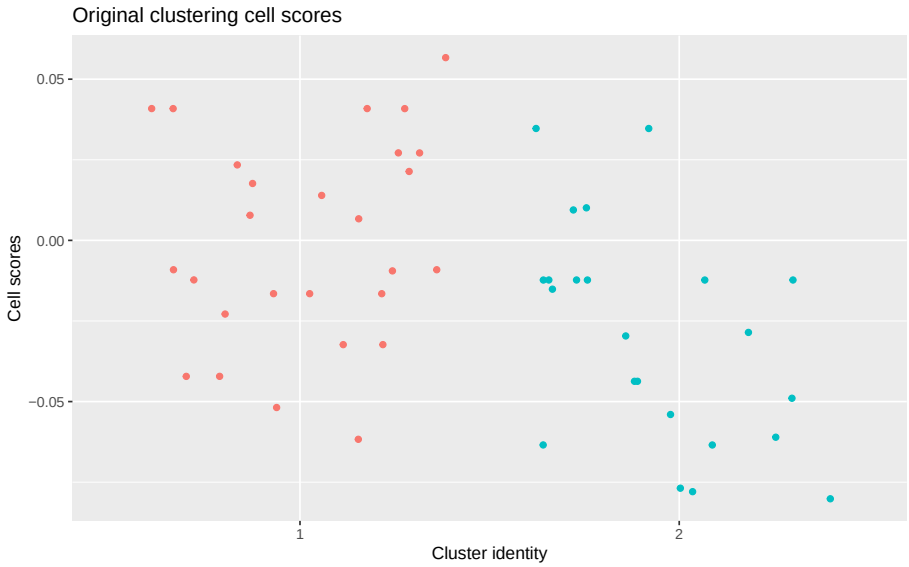
```
#> an error.

#> Warning in melt(lapply(cluster_names, calculate_cell_metrics_by_cluster, :
#> The melt generic in data.table has been passed a list and will attempt
#> to redirect to the relevant reshape2 method; please note that reshape2 is
#> deprecated, and this redirection is now deprecated as well. To continue using
#> melt methods from reshape2 while both libraries are attached, e.g. melt.list,
#> you can prepend the namespace like reshape2::melt(lapply(cluster_names,
#> calculate_cell_metrics_by_cluster, consensus_matrix = consensus_matrix,
#> cluster_labels = cluster_labels)). In the next version, this warning will become
#> an error.

#> Warning in melt(lapply(cluster_names, calculate_cell_metrics_by_cluster, :
#> The melt generic in data.table has been passed a list and will attempt
#> to redirect to the relevant reshape2 method; please note that reshape2 is
#> deprecated, and this redirection is now deprecated as well. To continue using
#> melt methods from reshape2 while both libraries are attached, e.g. melt.list,
#> you can prepend the namespace like reshape2::melt(lapply(cluster_names,
#> calculate_cell_metrics_by_cluster, consensus_matrix = consensus_matrix,
#> cluster_labels = cluster_labels)). In the next version, this warning will become
#> an error.

#> Warning in melt(lapply(cluster_names, calculate_cell_metrics_by_cluster, :
#> The melt generic in data.table has been passed a list and will attempt
#> to redirect to the relevant reshape2 method; please note that reshape2 is
#> deprecated, and this redirection is now deprecated as well. To continue using
#> melt methods from reshape2 while both libraries are attached, e.g. melt.list,
#> you can prepend the namespace like reshape2::melt(lapply(cluster_names,
#> calculate_cell_metrics_by_cluster, consensus_matrix = consensus_matrix,
#> cluster_labels = cluster_labels)). In the next version, this warning will become
#> an error.

#> Warning in melt(lapply(cluster_names, calculate_cell_metrics_by_cluster, :
#> The melt generic in data.table has been passed a list and will attempt
#> to redirect to the relevant reshape2 method; please note that reshape2 is
#> deprecated, and this redirection is now deprecated as well. To continue using
#> melt methods from reshape2 while both libraries are attached, e.g. melt.list,
#> you can prepend the namespace like reshape2::melt(lapply(cluster_names,
#> calculate_cell_metrics_by_cluster, consensus_matrix = consensus_matrix,
#> cluster_labels = cluster_labels)). In the next version, this warning will become
#> an error.
#> Warning in melt(cell.scores, id.var = c("rn", "cell", "cluster.size",
#> "metric")): The melt generic in data.table has been passed a list and will
#> attempt to redirect to the relevant reshape2 method; please note that reshape2
#> is deprecated, and this redirection is now deprecated as well. To continue using
#> melt methods from reshape2 while both libraries are attached, e.g. melt.list,
#> you can prepend the namespace like reshape2::melt(cell.scores). In the next
#> version, this warning will become an error.
head(cell_scores.df, n=4)
#>   Cluster.identity                Cell Cluster.size    Metric     Value
#> 1                1 WTCHG_217386_229230           49 Stability 0.7175926
```

```
#> 2               1 WTCHG_217386_249229          49 Stability 0.6898148
#> 3               1 WTCHG_227074_204206          49 Stability 0.8240741
#> 4               1 WTCHG_227074_204230          49 Stability 0.7893519
#>   Clustering
#> 1          2
#> 2          2
#> 3          2
#> 4          2
```

The output is a melted `data.frame` that displays the name of each cluster to which the cell belongs, the cell label, the size of each cluster, the metric (Score, Promiscuity, Stability), the value for each metric, and finally the clustering in question (1,2,. . . ,Original).

As with cluster metrics, these results can be plotted to visualize cells in the context of the original clusters using `ggplot2`. The score metric is plotted below to illustrate this:

```
original_cell_scores.df<-cell_scores.df[cell_scores.df$Clustering=="Original",]
ggplot(original_cell_scores.df[original_cell_scores.df$Metric=="Score",],
    aes(x=factor(`Cluster.identity`), y=Value) )+
    geom_jitter(aes(color=factor(`Cluster.identity`)), stat="identity")+
    xlab("Cluster identity")+ylab("Cell scores")+
    ggtitle("Original clustering cell scores")+guides(color=FALSE)
```

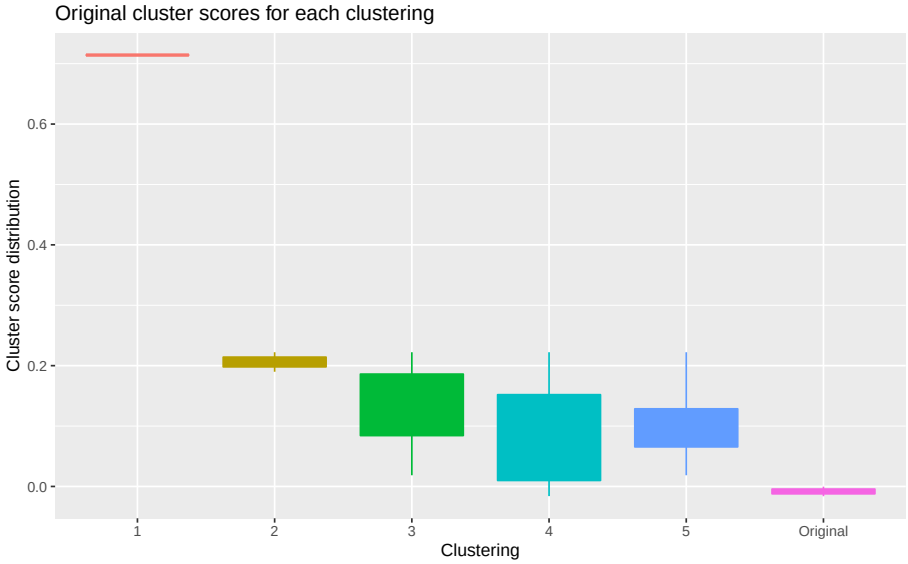


2.6.3 Using BEARscc to inform cluster number, k, choice

While BEARscc certainly does not claim to provide a definitive solution to the question concerning the number of clusters by any means, we provide what we believe to be a useful perspective on the matter. Specifically, we have found that by examining the cluster metrics across various hiearchical clusterings of the BEARscc noise consensus, the cluster number k with the highest score tends to provide a number of clusters that resembles ground truth more closely than simple gene sampling or relevant algorithms along as evidenced by experiments in control samples, c. elegans , and murine brain data (see our manuscript on bioRxiv. Importantly, this only provides a heuristic for determining cluster number k that takes into count the inherent noise of the single cell experiment, and the heuristic is dependent upon

the clustering algorithm of choosing (the better the utilized clustering algorithm, the better the BEARscc k heuristic) and represents a form of "meta-clustering" rather than a new way to determine the number of clusters, k, per se. As an illustration of utilizing this heuristic, we plot the average cluster score values for various cluster number scenarios below:

```
ggplot(cluster_scores.df[cluster_scores.df$Metric=="Score",],
    aes(x=`Clustering`, y=Value) )+ geom_boxplot(aes(fill=Clustering,
    color=`Clustering`))+ xlab("Clustering")+
    ylab("Cluster score distribution")+
    ggtitle("Original cluster scores for each clustering")+
    guides(fill=FALSE, color=FALSE)
```



As we can see, the 1 cluster scenario provides the best cluster score as determined by the noise consensus. As mentioned previously, the single cluster solution resembles the biological ground truth.

3 Algorithm and theory

In order to simulate technical replicates, BEARscc first builds a statistical model of expression variance and drop-out frequency, which is dependent only on observed gene expression. The parameters of this model are estimated from spike-in counts. Expression-dependent variance is approximated by fitting read counts of each spike-in transcript across cells to a mixture model comprised of a Poisson and negative binomial distribution (Section 3.1.1). The drop-out model (Section 3.1.2) in BEARscc has two distinct parts: the *drop-out injection distribution* models the likelihood that a given transcript concentration will result in a drop-out, and the *drop-out recovery distribution* models the likelihood that an observed drop-out resulted from a given transcript concentration. The drop-out injection distribution is taken to be the observed drop-out rate in spike-in controls as a function of actual spike-in transcript concentration. This distribution is then used to estimate the drop-out recovery distribution density via Bayes' theorem and a an empirically informed set of priors and assumptions. Briefly, BEARscc utilizes the drop-out injection distribution and the number of observed zeroes for each endogenous gene to infer a gene-specific probability distribution describing the likelihood that an observed drop-out should in fact have been some non-zero value, given the drop-out rate of the endogenous gene. This entire process is facilitated by the function, `estimate_noiseparameters()`.

In the second step, BEARscc generates simulated technical replicates by applying the models described in the first step (Section 3.2). For every observed count in the range of values where drop-outs occurred amongst the spike-in transcripts, BEARscc uses the drop-out injection distribution from Step 1 to determine whether to convert the count to zero. For observations where the count is zero, the drop-out recovery distribution is used to estimate a new value, given the overall drop-out frequency for the gene (Section 3.2). Next, BEARscc substitutes all values larger than zero with a value generated from the derived model of expression variability, parameterized to the observed count for that gene. This procedure can then be repeated any number of times to generate a collection of simulated technical replicates. This step is carried out by `create_noiseinjected_counts()`.

In the third step, the simulated technical replicates are then re-clustered, using exactly the same method as for the observed data; this re-clustering for each simulated technical replicate is described as an *association matrix* where each element indicates whether two cells share a cluster identity (1) or cluster apart from each other (0). The association matrices for each simulated technical replicate are averaged to form a noise consensus matrix that can be easily interpreted (Section 3.3). This is accomplished with the function `compute_consensus()`. Each element of the noise consensus matrix represents the fraction of simulated technical replicates that, upon applying the clustering method of choice, resulted in two cells clustering together (the *association frequency*). Then, the functions `report_cell_metrics()` and `report_cluster_metrics()` may be used to explore and quantitate the noise consensus matrix at the cell sample and cluster levels, respectively.

3.1 Noise estimation

As mentioned previously, BEARscc uses spike-ins to estimate the noise of the experiment for the purpose of producing simulated technical replicates. BEARscc models overall technical variation with a mixture-model (Section 3.1.1) and inferred drop-out effects (Section 3.1.2) independently using the spike-in observations. However, a single function in BEARscc `estimate_noiseparameters()` accomplishes this task.

3.1.1 Estimating transcript variation

Technical variance is modeled in BEARsccby fitting a single parameter mixture model, $Z(c)$, to the spike-ins' observed count distributions. The noise model is fit independently for each spike-in transcript and subsequently regressed onto spike-in mean expression to define a generalized noise model. This is accomplished in three steps:

1. Define a mixture model composed of *poisson* and *negative binomial* random variables:

$$Z \sim (1-\alpha) * Pois(\mu) + \alpha * NBin(\mu, \sigma) \quad \boxed{1}$$

2. Empirically fit the parameter, α_i, in a spike-in specific mixture-model, Z_i, to the observed distribution of counts for each ERCC spike-in transcript, i, where μ_i and σ_i are the observed mean and variance of the given spike-in. The parameter, α_i, is chosen such that the error between the observed and mixture-model is minimized.

3. Generalize the mixture-model by regressing α_i parameters and the observed variance, σ_i, onto the observed spike-in mean expression, μ_i. Thus the mixture model describing the noise observed in ERCC transcripts is defined solely by μ, which is treated as the count transformation parameter, c, in the generation of simulated technical replicates.

In step 2, a mixture model distribution is defined for each spike-in, i:

$$Z_i(\alpha_i, \mu_i, \sigma_i) \sim (1-\alpha_i) * Pois(\mu_i) + \alpha_i * NBin(\mu_i, \sigma_i). \quad \boxed{2}$$

The distribution, Z_i, is fit to the observed counts of the respective spike-in, where α_i is an empirically fitted parameter, such that the α_i minimizes the difference between the observed count distribution of the spike-in and the respective fitted model, Z_i. Specifically, for each spike-in transcript, μ_i and σ_i are taken to be the mean and standard deviation, respectively, of the observed counts for spike-in transcript, i. Then, α_i is computed by empirical parameter optimization; α_i is taken to be the $\alpha_{i,j}$ in the mixture-model,

$$Z_{i,j}(\alpha_{i,j}, \mu_i, \sigma_i) \sim (1-\alpha_{i,j}) * Pois(\mu_i) + \alpha_{i,j} * NBin(\mu_i, \sigma_i), \quad \boxed{3}$$

found to have the least absolute total difference between the observed count density and the density of the fitted model, Z_i. In the case of ties, the minimum $\alpha_{i,j}$ is chosen.

In step 3, $\alpha(c)$ is then defined with a linear fit, $\alpha_i = \alpha * log2(\mu_i) + b$. $\sigma(c)$ was similarly defined, $log2(\sigma_i) = \alpha * log2(\mu_i) + b$. In this way, the observed distribution of counts in spike-in transcripts defines the single parameter mixture-model, $Z(c)$, used to transform counts during generation of simulated technical replicates:

$$Z(c) \sim (1-\alpha(c)) * Pois(c) + \alpha(c) * NBin(c, \sigma(c)) \quad \boxed{4}$$

During technical replicate simulation, the parameter c is set to the observed count value, a, and the transformed count in the simulated replicate was determined by sampling a single value from $Z(c = a)$.

3.1.2 Defining the drop-out models

A model of the drop-outs is developed by BEARscc in order to inform the permutation of zeros during noise injection. The observed zeros in spike-in transcripts as a function of actual transcript concentration and Bayes' theorem are used to define two models: the *drop-out injection distribution* and the *drop-out recovery distribution*.

The drop-out injection distribution is described by $Prob(X = 0|Y = y)$, where X is the distribution of observed counts and Y is the distribution of actual transcript counts; the density is computed by regressing the fraction of zeros observed in each sample, D_i, for a given spike-in, i, onto the expected number spike-in molecules in the sample, y_i, e.g. $D = a * y + b$. Then, D describes the density of zero-observations conditioned on actual transcript number, y, or $Prob(X = 0|Y = y)$. Notably, each gene was treated with an identical density distribution for drop-out injection.

In contrast, the density of the drop-out recovery distribution, $Prob(Y_j = y|X_j = 0)$, is specific to each gene, j, where X_j is the distribution of the observed counts and Y_j is the distribution of actual transcript counts for a given gene. The gene-specific drop-out recovery distribution is inferred from drop-out injection distribution using Bayes' theorem and a prior. This is accomplished in 3 steps:

1. For the purpose of applying Bayes' theorem, the gene-specific distribution, $Prob(X_j = 0|Y_j = y)$, is taken to be the the drop-out injection density for all genes, j.
2. The probability that a specific transcript count is present in the sample, $Prob(Y_j = y)$, is a necessary, but empirically unknowable prior. Therefore, the prior was defined using the law of total probability, an assumption of uniformity, and the probability that a zero was observed in a given gene, $Prob(X_j = 0)$. The probability, $Prob(X_j = 0)$, is taken to be the fraction of observations that are zero for a given gene. BEARscc does this in order to better inform the density estimation of the gene-specific drop-out recovery distribution.
3. The drop-out recovery distribution density is then computed by applying Bayes' theorem:

$$Prob(Y_j = y|X_j = 0) = \frac{Prob(X_j = 0|Y_j = y) * Prob(Y_j = y)}{Prob(X_j = 0)}, \quad \boxed{5}$$

In the second step, the law of total probability, an assumption of uniformity, and the fraction of zero observations in a given gene are leveraged to define the prior, $Prob(Y_j = y)$. First, a threshold of expected number of transcripts, k in Y, is chosen such that k was the maximum value for which the drop-out injection density was non-zero. Next, uniformity is assumed for all expected number of transcript values, y greater than zero and less than or equal to k; that is $Prob(Y_j = y)$ is defined to be some constant probability, n. Furthermore, $Prob(Y_j = y)$ is defined to be 0 for all $y > k$. In order to inform $Prob(Y_j = y)$ empirically, $Prob(Y_j = 0)$ and n are derived by imposing the law of total probability (Equation 6) and unity (Equation 7) yielding a system of equations:

$$Prob(X_j = 0) = \Sigma_{y=0}^{k-1}(Prob(X_j = 0|Y_j = y) * Prob(Y_j = y)) \quad \boxed{6}$$

$$\Sigma_{y=0}^{k-1} Prob(Y_j = y) = Prob(Y_j = 0) + (k - 1) * n = 1 \quad \boxed{7}$$

The probability that a zero is observed given there are no transcripts in the sample, $Prob(X_j = 0|Y_j = 0)$, is assumed to be 1. With the preceding assumption, solving for $Prob(Y_j = 0)$ and n give:

$$n = \frac{1 - Prob(Y_j = 0)}{k - 1} \quad \boxed{8}$$

$$Prob(Y_j = 0) = \frac{Prob(X_j = 0) - \frac{1}{k-1} * \Sigma_{y=1}^{k-1}(Prob(X_j = 0|Y_j = y))}{(1 - \frac{1}{k-1} * \Sigma_{y=1}^{k-1}(Prob(X_j = 0|Y_j = y))} \quad 9$$

\

In this way, $Prob(Y_j = 0)$ is defined by (Equation 8) for y in Y_j less than or equal to k and greater than zero, and defined by (Equation 9) for y in Y_j equal to zero. For y in Y_j greater than k, the prior $Prob(Y_j = y)$ is defined to be equal to zero.

In the third step, the previously computed prior, $Prob(Y_j = y)$, the fraction of zero observations in a given gene, $Prob(X_j = 0)$, and the drop-out injection distribution, $Prob(X_j = 0|Y_j = y)$, are utilized to estimate, with Bayes's theorem, the density of the drop-out recovery distribution, $Prob(Y_j = y|X_j = 0)$. During the generation of simulated technical replicates for zero observations and count observations less than or equal to k, values are sampled from the drop-out recovery and injection distributions as described in the pseudocode of the BEARscc algorithm for simulating technical replicates.

3.2 Simulating technical replicates

Simulated technical replicates were generated from the noise mixture-model and two drop-out models. For each gene, the count value of each sample is systematically transformed using the mixture-model, $Z(c)$, and the drop-out injection, $Prob(X = 0|Y = y)$, and recovery, $Prob(Y_j = y|X_j = 0)$, distributions in order to generate simulated technical replicates as indicated by the following pseudocode:

```
FOR EACH gene, $j$
    FOR EACH count, $c$
        IF $c=0$
            $n \leftarrow SAMPLE$ one count, y, from $Prob(Y_j=y | X_j=0)$
            IF $n=0$
                $c \leftarrow 0$
            ELSE
                $c \leftarrow SAMPLE$ one count from $Z(n)$
            ENDIF
        ELSE
            IF $c\leq k$
                $dropout \leftarrow TRUE$ with probability, $Prob(X=0 | Y=k)$
                IF $dropout=TRUE$
                    $c \leftarrow 0$
                ELSE
                    $c \leftarrow SAMPLE$ one count from $Z(c)$
                ENDIF
            ELSE
                $c \leftarrow SAMPLE$ one count from $Z(c)$
            ENDIF
        ENDIF
        RETURN $c$
    DONE
DONE
```

4 List of functions

4.1 `estimate_noiseparameters()`

4.1.1 Description

Estimates the drop-out model and technical variance from spike-ins present in the sample.

For greater detail, please see help file `?estimate_noiseparameters()`.

4.1.2 Usage

```
data(BEARscc_examples)
BEAR_examples.sce <- estimate_noiseparameters(BEAR_examples.sce,
alpha_resolution=0.25, write.noise.model=FALSE)

BEAR_examples.sce
```

4.1.3 Output

The resulting output of `estimate_noiseparameters()` is a long list, which is enumerated in the function's package help page.

4.1.4 Note

The above usage is for execution of `simulate_replicates` on a local machine. To save results as files for us of `prepare_probabilities` on high performance computing environment, then use:

```
estimate_noiseparameters(BEAR_examples.sce,
    write.noise.model=TRUE, alpha_resolution=0.25,
    file="noise_estimation", model_view=c("Observed","Optimized"))
```

4.2 `simulate_replicates()`

4.2.1 Description

Computes BEARscc simulated technical replicates from the previously estimated noise parameters computed with the function `estimate_noise_parameters()`.

For greater detail, please see help file `?simulate_replicates()`.

4.2.2 Usage

```
data(analysis_examples)
BEAR_examples.sce<-simulate_replicates(BEAR_examples.sce, n=3)

BEAR_examples.sce
```

4.2.3 Output

The resulting object is a list of counts data, where each element of the list is a `data.frame` of the counts representing a BEARscc simulated technical replicate. For further details refer to the function help page.

4.2.4 Note

This function is the in-package analog of the high-performance computing function,`prepare_probabilities`.

4.3 `HPC_simulate_replicates()`

4.3.1 Description

The high-performance computing function analog to `simulate_replicates()`.

4.3.2 Usage

Please refere to section 2.4.

4.3.3 Output

The resulting objects would normally be output to a tab-delimited file, where each file results from a `data.frame` of the counts representing a BEARscc simulated technical replicate.

4.3.4 Note

This function has no help file, but is referred to in the section 2.4 of this document on simulating for larger datasets.

4.4 `compute_consensus()`

4.4.1 Description

Computes the consensus matrix using a `data.frame` of cluster labels across different BEARscc simulated technical replicates.The consensus matrix is is a visual and quantitaive representation of the clustering variation on a cell-by-cell level created by using cluster labels to compute the number of times any given pair of cells associates in the same cluster; this forms the 'noise consensus matrix'. Each element of this matrix represents the fraction of simulated technical replicates in which two cells cluster together (the 'association frequency'), after using a clustering method of the user's choice to generate a data.frame of clustering labels. This consensus matrix may be used to compute BEARscc metrics at both the cluster and cell level.

For greater detail, please see help file `?compute_consensus()`.

4.4.2 Usage

```
data("analysis_examples")
noise_consensus <- compute_consensus(clusters.df)
noise_consensus
```

4.4.3 Output

When the number of samples are n, then the noise consensus resulting from this function is an $n \times n$ matrix describing the fraction of simulated technical replicates in which each cell of the experiment associates with another cell.

4.5 `cluster_consensus()`

4.5.1 Description

This function will perform hierarchical clustering on the noise consensus matrix allowing the user to investigate the appropriate number of clusters, k, considering the noise within the experiment. Frequently one will want to assess multiple possible cluster number situations at once. In this case it is recommended that one use a `lapply` in conjunction with a vector of all biologically reasonable cluster numbers to fulfill the task of attempting to identify the optimal cluster number.

For greater detail, please see help file `?cluster_consensus()`.

4.5.2 Usage

```
data(analysis_examples)
vector <- seq(from=2, to=5, by=1)
BEARscc_clusts.df <- cluster_consensus(consensus_matrix=noise_consensus,
    vector)
BEARscc_clusts.df
```

4.5.3 Output

The output is a vector of cluster labels based on hierarchical clustering of the noise consensus. In the event that a vector is supplied for number of clusters in conjunction with `lapply`, then the output is a data.frame of the cluster labels for each of the various number of clusters deemed biologically reasonable by the user.

4.6 `report_cell_metrics()`

4.6.1 Description

To quantitatively evaluate the results, three metrics are calculated from the noise consensus matrix: 'stability' is the average frequency with which a cell within a cluster associates with other cells within the same cluster across simulated replicates; 'promiscuity' measures the average association frequency of a cell within a cluster with the n cells outside of the cluster with the strongest association with the cell in question; and 'score' is the difference between 'stability' and 'promiscuity'. Importantly, 'score' reflects the overall "robustness" of a given cell's assignment to a user-provided cluster label with respect to technical variance. Together these metrics provide a quantitative measure of the extent to which cluster labels provided by the user are invariant across simulated technical replicates.

For greater detail, please see help file `?report_cell_metrics()`.

4.6.2 Usage

```
data(analysis_examples)
cell_scores.dt <- report_cell_metrics(BEARscc_clusts.df, noise_consensus)
cell_scores.dt
```

4.6.3 Output

A melted `data.frame` describing the BEARscc metrics for each cell, where the columns are enumerated in the help file.

4.7 `report_cluster_metrics()`

4.7.1 Description

To quantitatively evaluate the results, three metrics are calculated from the noise consensus matrix: 'stability' is the average frequency with which cells within a cluster associate with each other across simulated replicates; 'promiscuity' measures the association frequency between cells within a cluster and those outside of it; and 'score' is the difference between 'stability' and 'promiscuity'. Importantly, 'score' reflects the overall "robustness" of a cluster to technical variance. Together these metrics provide a quantitative measure of the extent to which cluster labels provided by the user are invariant across simulated technical replicates.

For greater detail, please see help file `?report_cluster_metrics()`.

4.7.2 Usage

```
data(analysis_examples)
cluster_scores.dt <- report_cluster_metrics(BEARscc_clusts.df,noise_consensus,
    plot=TRUE, file="example")
cluster_scores.dt
```

4.7.3 Output

A melted `data.frame` describing the BEARscc metrics for each cluster, where the columns are enumerated in the help file.

#Example data Within the package there are data subsampled from single cell sequencing protocol applied to water samples containing ERCC spike-ins (blanks) and dilute RNA from brain whole tissue (brain) discussed at length in in a manuscript on bioRxiv

4.8 BEARscc_examples

4.8.1 Description

A toy dataset for applying BEARscc functions as described in the README on https://bitbucket.org/bsblabludwig/bearscc.git. These data are a subset of observations made by Drs. Michael White and Richard Owen in the Xin Lu Lab. Samples were sequenced by the Wellcome Trust Center for Genomics, Oxford, UK. These data are available in full with GEO accession number, GSE95155.

For greater detail, please see help file `?BEARscc_examples`.

4.8.2 Usage

```
data("BEARscc_examples")
```

4.9 analysis_examples

4.9.1 Description

BEARscc downstream example objects: The `analysis_examples` Rdata object contains downstream data objects for use in various help pages for dynamic execution resulting from running tutorial in README and vignette on `BEARscc_examples`. The objects are a result of applying BEARscc functions as described in the README found at https://bitbucket.org/bsblabludwig/bearscc.git.

For greater detail, please see help file `?analysis_examples`.

4.9.2 Usage

```
data("analysis_examples")
```

5 License