Analysis of single-cell genomic data with celda

Sean Corbett[1], **Yusuke Koga**[1], **Shiyi Yang**[1], **Zhe Wang**[1], and **Joshua Campbell**[*1]

[1]Boston University School of Medicine

[*]camp@bu.edu

2020-10-08

Contents

1   Introduction

CEllular **L**atent **D**irichlet **A**llocation (celda) is a collection of Bayesian hierarchical models to perform feature and cell bi-clustering for count data generated by single-cell platforms. This algorithm is an extension of the Latent Dirichlet Allocation (LDA) topic modeling framework that has been popular in text mining applications and has shown good performance with sparse data. celda simultaneously clusters features (i.e. gene expression) into modules based on co-expression patterns across cells and cells into subpopulations based on the probabilities of the feature modules within each cell.

In this vignette we will demonstrate how to use celda to perform cell and feature clustering with a simple simulated dataset.

2   Installation

celda can be installed from Bioconductor:

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

The package can be loaded using the `library` command.

```
library(celda)
```

Complete list of help files are accessible using the help command with a `package` option.

```
help(package = celda)
```

To see the latest updates and releases or to post a bug, see our GitHub page at https://github.com/campbio/celda. To ask questions about running celda, post a thread on Bioconductor support site at https://support.bioconductor.org/.

3   Reproducibility note

Many functions in *celda* make use of stochastic algorithms or procedures which require the use of random number generator (RNG) for simulation or sampling. To maintain reproducibility, all these functions use a **default seed of 12345** to make sure same results are generated each time one of these functions is called. Explicitly setting the `seed` arguments is needed for greater control and randomness.

4   Generation of a simulated single cell dataset

celda will take a matrix of counts where each row is a feature, each column is a cell, and each entry in the matrix is the number of counts of each feature in each cell. To illustrate the utility of celda, we will apply it to a simulated dataset.

In the function `simulateCells`, the *K parameter designates the number of cell clusters*, the *L parameter determines the number of feature modules*, the *S parameter determines the number of samples in the simulated dataset*, the *G parameter determines the number of features to be simulated*, and *CRange specifies the lower and upper bounds of the number of cells to be generated in each sample*.

```
simCounts <- simulateCells("celda_CG",
    K = 5, L = 10, S = 5, G = 200, CRange = c(30, 50))
```

The `counts` variable contains the counts matrix. The dimensions of counts matrix:

```
dim(simCounts$counts)
## [1] 200 207
```

The `z` variable contains the cluster labels for each cell. Here is the number of cells in each subpopulation:

```
table(simCounts$z)
##
##  1  2  3  4  5
## 42 44 40 47 34
```

The `y` variable contains the feature module assignment for each feature. Here is the number of features in each feature module:

```
table(simCounts$y)
##
##  1  2  3  4  5  6  7  8  9 10
## 23 39 17 15 21 22 19 12  4 28
```

The `sampleLabel` is used to denote the sample from which each cell was derived. In this simulated dataset, we have 5 samples:

```
table(simCounts$sampleLabel)
##
## Sample_1 Sample_2 Sample_3 Sample_4 Sample_5
##       43       48       45       40       31
```

5 Performing bi-clustering with celda

There are currently three models within this package: `celda_C` will cluster cells, `celda_G` will cluster features, and `celda_CG` will simultaneously cluster cells and features. Within the function the K parameter will be the number of cell populations to be estimated, while the L parameter will be the number of feature modules to be estimated in the output model.

```
celdaModel <- celda_CG(counts = simCounts$counts,
    K = 5, L = 10, verbose = FALSE)
```

Here is a comparison between the true cluster labels and the estimated cluster labels, which can be found within the `z` and `y` objects.

```
table(clusters(celdaModel)$z, simCounts$z)
##
##     1  2  3  4  5
##   1 42  0  0  0  0
##   2  0 44  0  0  0
##   3  0  0 40  0  0
##   4  0  0  0 47  0
##   5  0  0  0  0 34
table(clusters(celdaModel)$y, simCounts$y)
##
##       1  2  3  4  5  6  7  8  9 10
##   1   1 39  0  0  0  0  0  0  0  0
##   2  22  0  0  0  0  0  0  0  0  0
##   3   0  0 17  0  0  0  0  0  0  0
##   4   0  0  0 15  0  0  0  0  0  0
##   5   0  0  0  0 21  0  0  0  0  0
##   6   0  0  0  0  0 22  0  0  0  0
##   7   0  0  0  0  0  0 18  0  0  2
##   8   0  0  0  0  0  0  0 12  0  0
##   9   0  0  0  0  0  0  0  0  4  0
##   10  0  0  0  0  0  0  1  0  0 26
```

6   Matrix factorization

celda can also be thought of as performing matrix factorization of the original counts matrix into smaller matrices that describe the contribution of each feature in each module, each module in each cell population, or each cell population in each sample. Each of these following matrices can be viewed as raw counts, proportions, or posterior probabilities.

```
factorized <- factorizeMatrix(counts = simCounts$counts, celdaMod = celdaModel)
names(factorized)
## [1] "counts"     "proportions" "posterior"
```

The `cell` object contains each feature module's contribution to each cell subpopulation. Here, there are 10 feature modules to 207 cells.

```
dim(factorized$proportions$cell)
## [1]  10 207
head(factorized$proportions$cell[, seq(3)], 5)
##        Cell_1      Cell_2      Cell_3
## L1 0.01709402 0.010452962 0.01117318
## L2 0.04957265 0.019163763 0.06331471
## L3 0.03076923 0.193379791 0.04841713
## L4 0.04957265 0.182926829 0.08193669
## L5 0.01367521 0.003484321 0.01675978
```

The `cellPopulation` contains each feature module's contribution to each of the cell states. Since K and L were set to be 5 and 10, there are 5 cell subpopulations to 10 feature modules.

```
cellPop <- factorized$proportions$cellPopulation
dim(cellPop)
## [1] 10  5
head(cellPop, 5)
##           K1         K2         K3         K4         K5
## L1 0.16530605 0.020828416 0.10820256 0.01485501 0.03701279
## L2 0.11236265 0.018261742 0.05533085 0.04767835 0.03708982
## L3 0.37986979 0.200820156 0.01931726 0.03993951 0.07837775
## L4 0.17846440 0.171554166 0.02556221 0.06609144 0.02391773
## L5 0.03107024 0.008703092 0.20944831 0.02232699 0.22758435
```

The `module` object contains each feature's contribution to the module it belongs.

```
dim(factorized$proportions$module)
## [1] 200  10
head(factorized$proportions$module, 5)
##        L1 L2 L3       L4 L5          L6 L7          L8 L9          L10
## Gene_1  0  0  0 0.152979  0 0.00000000  0 0.00000000  0 0.000000000
## Gene_2  0  0  0 0.000000  0 0.00000000  0 0.00000000  0 0.033413638
## Gene_3  0  0  0 0.000000  0 0.04679137  0 0.00000000  0 0.000000000
## Gene_4  0  0  0 0.000000  0 0.00000000  0 0.00000000  0 0.009427045
## Gene_5  0  0  0 0.000000  0 0.00000000  0 0.05690496  0 0.000000000
```

The top features in a feature module can be selected using the `topRank` function on the `module` matrix:

```
topGenes <- topRank(matrix = factorized$proportions$module,
    n = 10, threshold = NULL)
```

```
topGenes$names$L1
## [1] "Gene_175" "Gene_20"  "Gene_180" "Gene_132" "Gene_112" "Gene_134"
## [7] "Gene_10"  "Gene_191" "Gene_32"  "Gene_90"
```

7 Visualization

7.1 Creating an expression heatmap using the celda model

The clustering results can be viewed with a heatmap of the normalized counts using the function `celdaHeatmap`. The top `nfeatures` in each module will be selected according to the factorized module probability matrix.

```
celdaHeatmap(counts = simCounts$counts, celdaMod = celdaModel, nfeatures = 10)
```
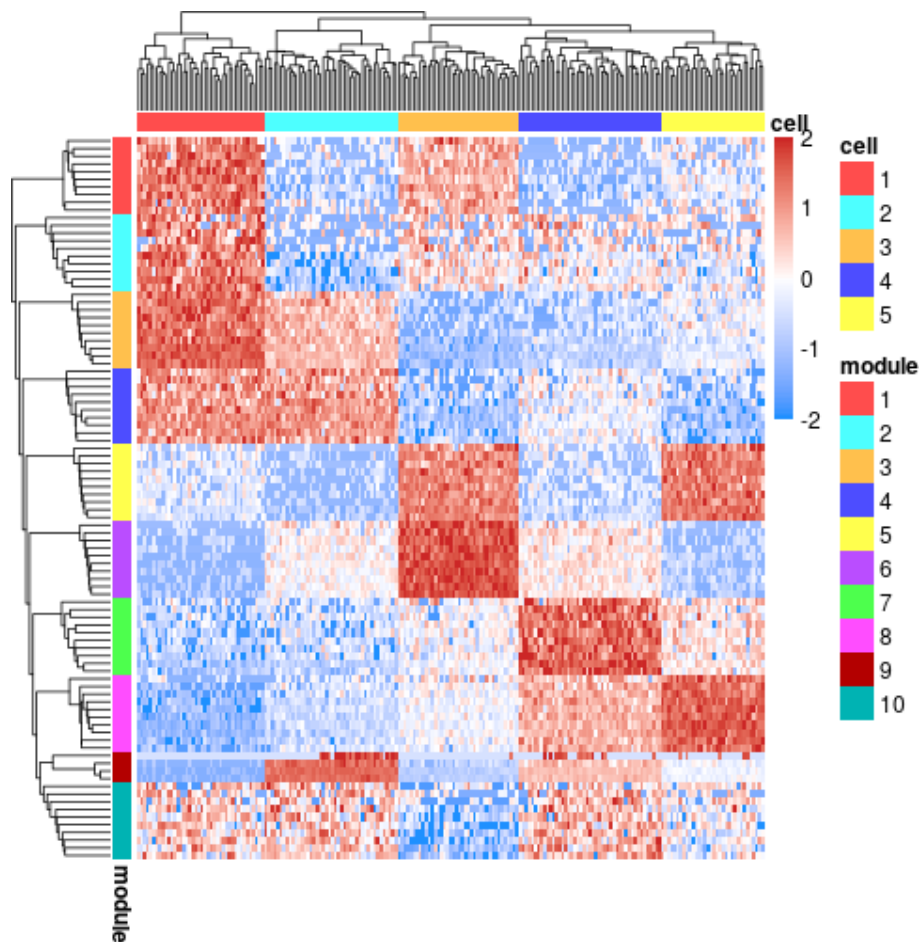
7.2   Plotting cell populations with tSNE

celda contains its own wrapper function for tSNE, `celdaTsne`, which can be used to embed cells into 2-dimensions. The output can be used in the downstream plotting functions `plotDim ReduceCluster`, `plotDimReduceModule`, and `plotDimReduceFeature` to show cell population clusters, module probabilities, and expression of a individual features, respectively.

```
tsne <- celdaTsne(counts = simCounts$counts, celdaMod = celdaModel)
```

```
plotDimReduceCluster(dim1 = tsne[, 1],
    dim2 = tsne[, 2],
    cluster = clusters(celdaModel)$z)
```
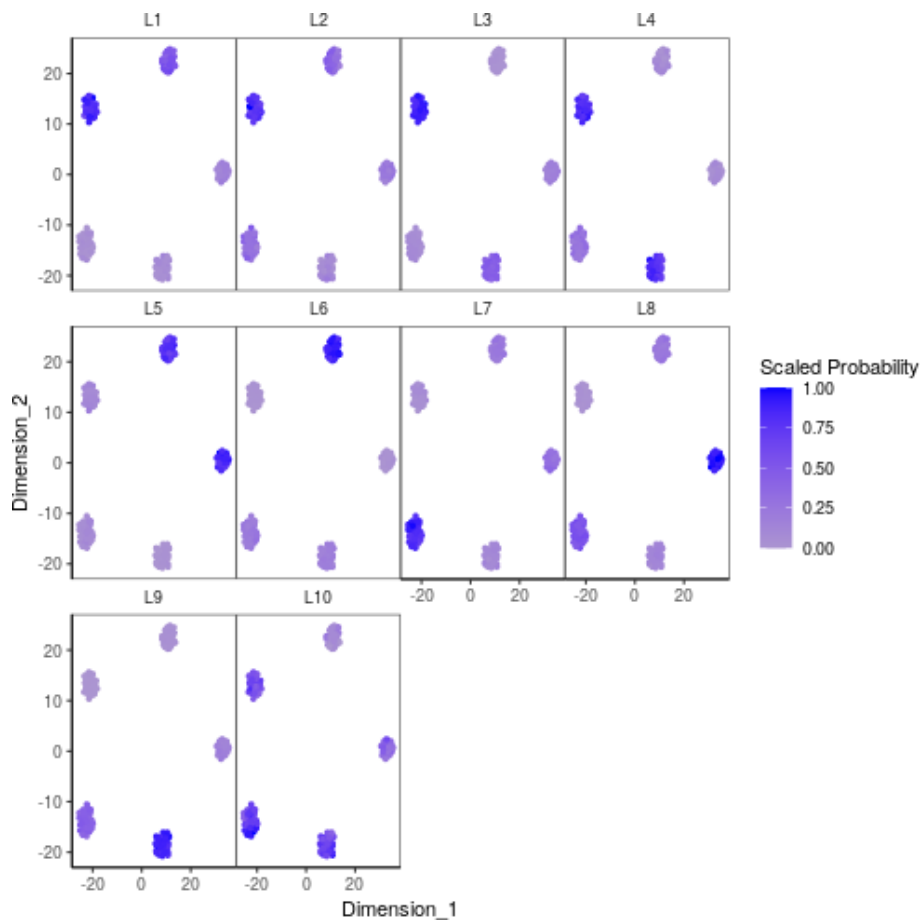
Analysis of single-cell genomic data with celda
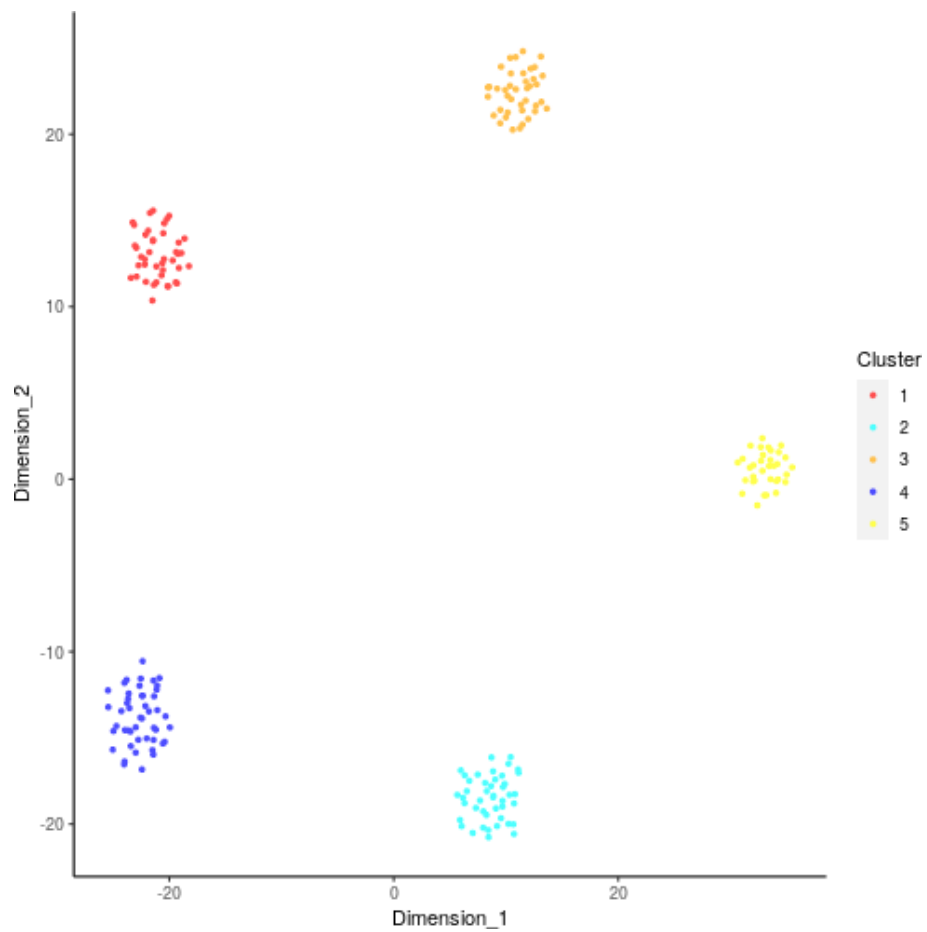

```
plotDimReduceModule(dim1 = tsne[, 1],
    dim2 = tsne[, 2],
    celdaMod = celdaModel,
    counts = simCounts$counts,
    rescale = TRUE)
```
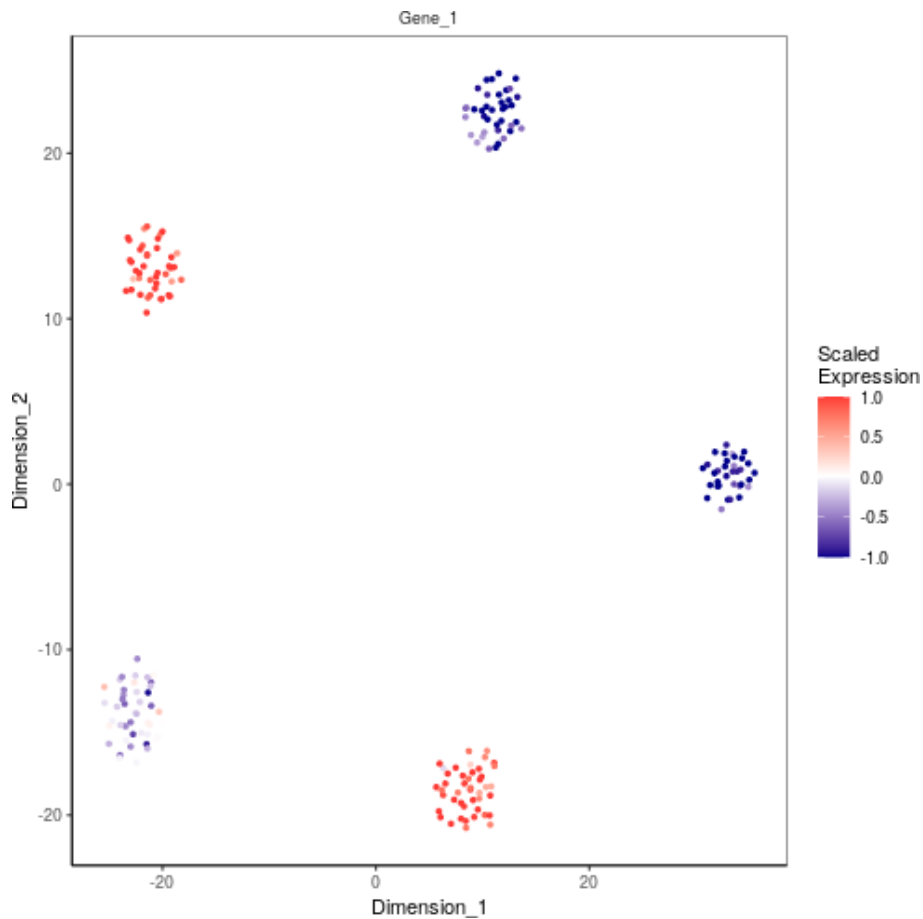
Analysis of single-cell genomic data with celda


```
plotDimReduceFeature(dim1 = tsne[, 1],
    dim2 = tsne[, 2],
    counts = simCounts$counts,
    normalize = TRUE,
    features = "Gene_1")
```

7.3 Displaying relationships between modules and cell populations

The relationships between feature modules and cell populations can be visualized with `celdaProbabilityMap`. The absolute probabilities of each feature module in each cellular subpopulation is shown on the left. The normalized and z-scored expression of each module in each cell population is shown on the right.

```
celdaProbabilityMap(counts = simCounts$counts, celdaMod = celdaModel)
```
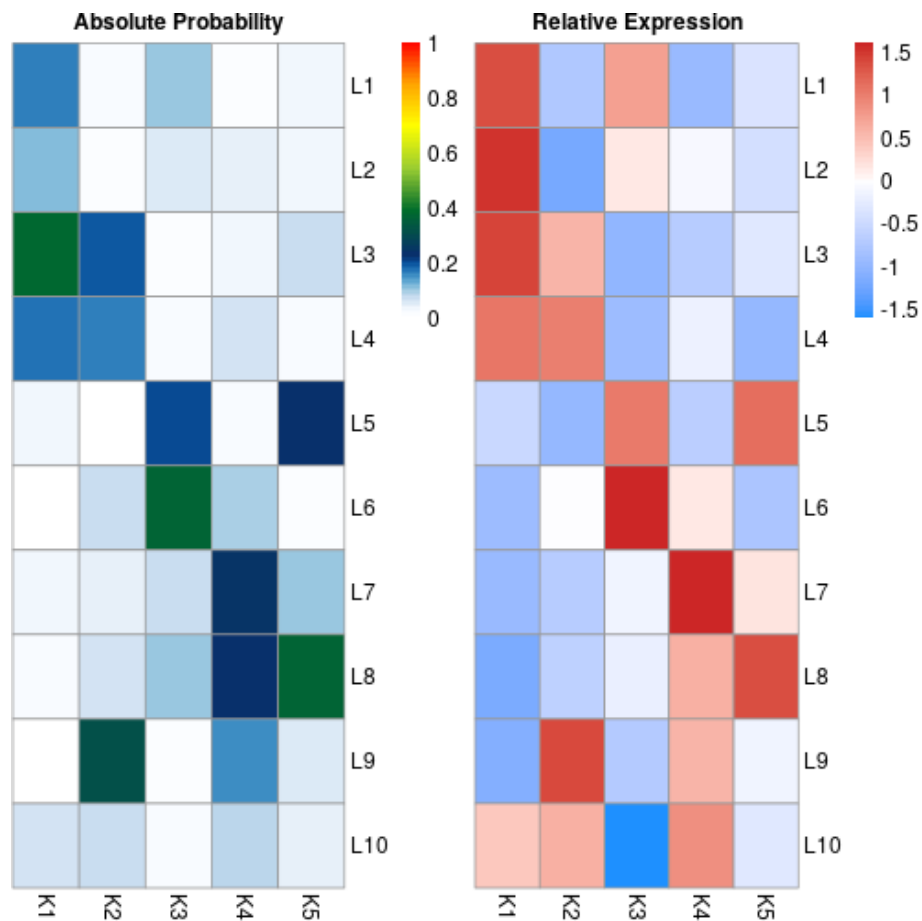
7.4   Examining co-expression with module heatmaps

`moduleHeatmap` creates a heatmap using only the features from a specific feature module. Cells are ordered from those with the lowest probability of the module to the highest. If more than one module is used, then cells will be ordered by the probabilities of the first module.

```
moduleHeatmap(counts = simCounts$counts, celdaMod = celdaModel,
    featureModule = 1, topCells = 100)
```

Analysis of single-cell genomic data with celda
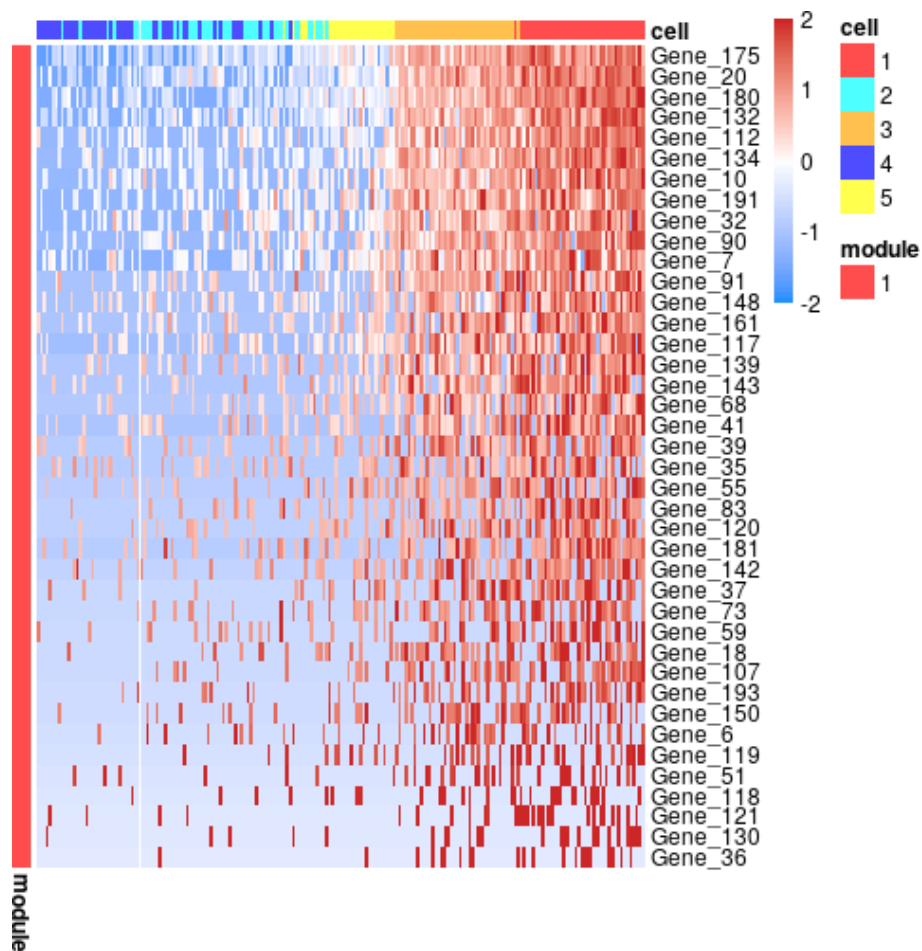
While `celdaHeatmap` will plot a heatmap directly with a celda object, the `plotHeatmap` function is a more general heatmap function which takes a normalized expression matrix as the input. Simple normalization of the counts matrix can be performed with the `normalizeCounts` function. For instance, if users want to display specific modules and cell populations, the `featureIx` and `cells.ix` parameters can be used to select rows and columns out of the matrix.

```
genes <- c(topGenes$names$L1, topGenes$names$L10)
geneIx <- which(rownames(simCounts$counts) %in% genes)
normCounts <- normalizeCounts(counts = simCounts$counts, scaleFun = scale)
```

```
plotHeatmap(counts = normCounts,
    z = clusters(celdaModel)$z,
    y = clusters(celdaModel)$y,
    featureIx = geneIx,
    showNamesFeature = TRUE)
```
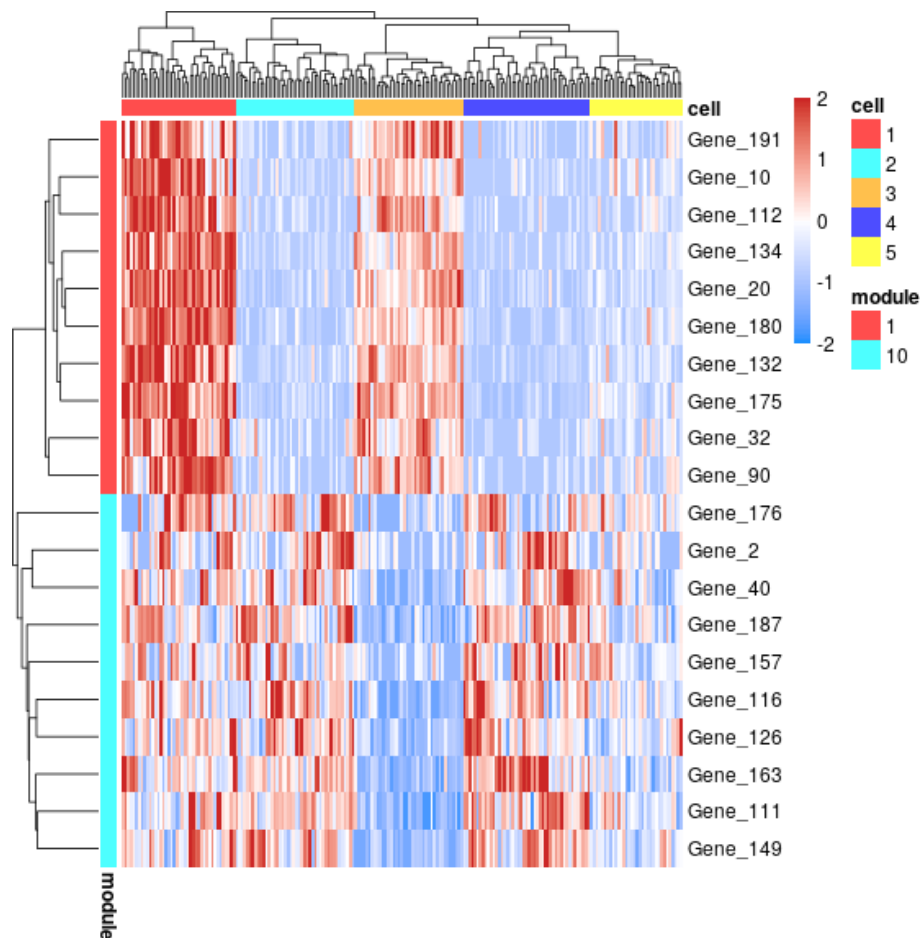
8 Differential Expression Analysis

celda employs the MAST package (McDavid A, 2018) for differential expression analysis of single-cell data. The `differentialExpression` function can determine features that are differentially expressed in one cell subpopulation versus all other subpopulations, between two individual cell subpopulations, or between different groups of cell populations.

If you wish to compare one cell subpopulation compared to all others, leave c2 as NULL.

```
diffexpClust1 <- differentialExpression(counts = simCounts$counts,
    celdaMod = celdaModel,
    c1 = 1,
    c2 = NULL)

head(diffexpClust1, 5)
##       Gene      Pvalue    Log2_FC          FDR
## 1: Gene_192 3.205425e-41 -7.800409 3.452950e-39
## 2:  Gene_72 3.452950e-41 -7.001386 3.452950e-39
## 3: Gene_113 6.275161e-38  3.084507 4.183441e-36
## 4:  Gene_92 1.313045e-36  3.271233 6.565227e-35
```

```
## 5:  Gene_27 6.584171e-36  2.111428 2.633668e-34
```

If you wish to compare two cell subpopulations, use both c1 and c2 parameters.

```
diffexpClust1vs2 <- differentialExpression(
    counts = simCounts$counts,
    celdaMod = celdaModel,
    c1 = 1,
    c2 = 2)

diffexpClust1vs2 <- diffexpClust1vs2[diffexpClust1vs2$FDR < 0.05 &
    abs(diffexpClust1vs2$Log2_FC) > 2, ]
head(diffexpClust1vs2, 5)
##        Gene      Pvalue    Log2_FC          FDR
## 1:  Gene_72 4.658824e-56 -8.431850 9.317649e-54
## 2: Gene_192 8.689595e-50 -8.965264 8.689595e-48
## 3: Gene_186 1.219911e-45 -9.016481 8.132743e-44
## 4:  Gene_20 2.396962e-26  3.283106 1.198481e-24
## 5: Gene_180 2.422467e-25  4.249271 9.689866e-24
```

The differentially expressed genes can be visualized further with a heatmap:

```
diffexpGeneIx <- which(rownames(simCounts$counts) %in% diffexpClust1vs2$Gene)

normCounts <- normalizeCounts(counts = simCounts$counts, scaleFun = scale)
```

```
plotHeatmap(counts = normCounts[, clusters(celdaModel)$z %in% c(1, 2)],
    clusterCell = TRUE,
    z = clusters(celdaModel)$z[clusters(celdaModel)$z %in% c(1, 2)],
    y = clusters(celdaModel)$y,
    featureIx = diffexpGeneIx,
    showNamesFeature = TRUE)
```
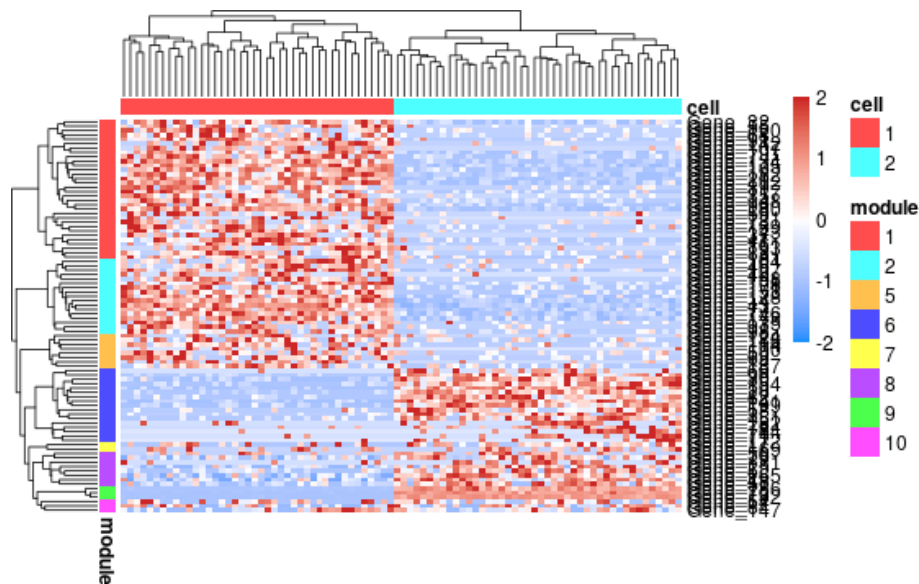
9 Identifying the optimal number of cell subpopulations and feature modules

In the previous example, the best K (the number of cell clusters) and L (the number of feature modules) was already known. However, the optimal K and L for each new dataset will likely not be known beforehand and multiple choices of K and L may need to be tried and compared. celda offers two sets of functions to determine the optimum K and L, `recursiveSplitModule`/`recursiveSplitCell`, and `celdaGridSearch`.

9.1 recursiveSplitModule/recursiveSplitCell

Functions `recursiveSplitModule` and `recursiveSplitCell` offer a fast method to generate a celda model with optimum K and L.

First, `recursiveSplitModule` is used to determine the optimal L. `recursiveSplitModule` first splits features into however many modules are specified in `initialL`. The module labels are then recursively split in a way that would generate the highest log likelihood, all the way up to `maxL`.

```
moduleSplit <- recursiveSplitModule(counts = simCounts$counts,
    initialL = 2, maxL = 15)
```

Perplexity is a statistical measure of how well a probability model can predict new data. Lower perplexity indicates a better model. The perplexity of each model can be visualized with `plotGridSearchPerplexity`. In general, visual inspection of the plot can be used to select the optimal number of modules (L) or cell populations (K) by identifying the "elbow" - where the rate of decrease in the perplexity starts to drop off.

```
plotGridSearchPerplexity(celdaList = moduleSplit)
```
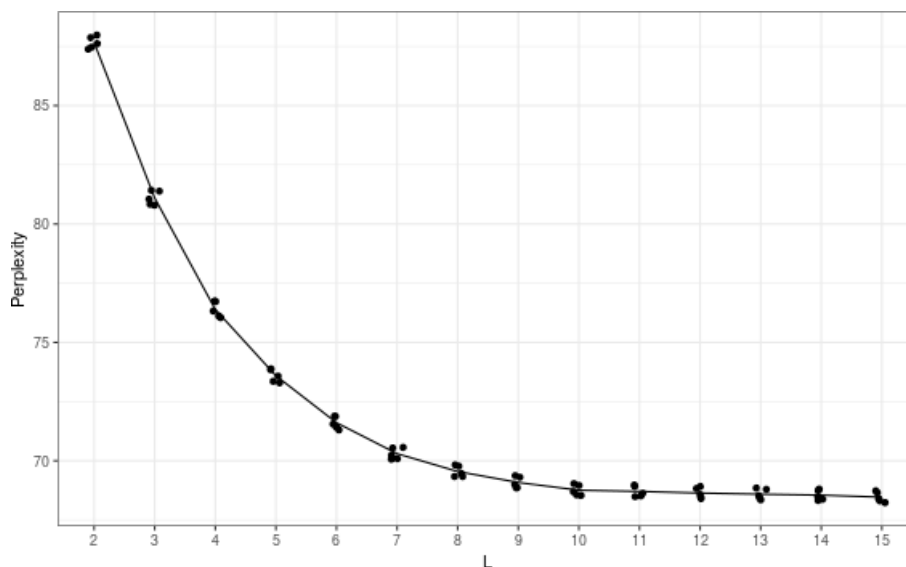

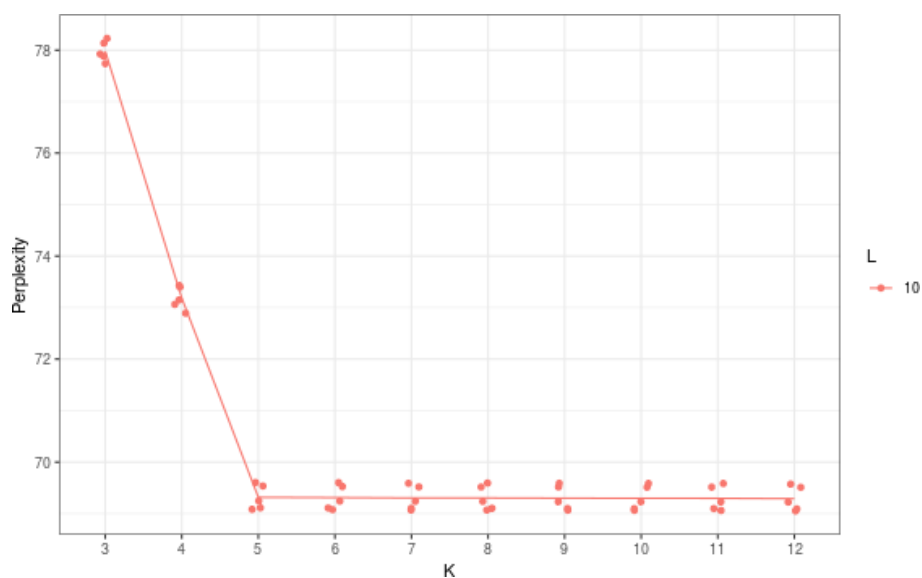In this example, the perplexity for L stops decreasing at L = 10, thus L = 10 would be a good choice.

Once you have identified the optimal L (in this case, L is selected to be 10), the module labels are used for intialization in `recursiveSplitCell`. Similarly to `recursiveSplitModule`, cells are initially split into a small number of subpopulations, and the subpopulations are recursively split up by log-likelihood.

```
moduleSplitSelect <- subsetCeldaList(moduleSplit, params = list(L = 10))

cellSplit <- recursiveSplitCell(counts = simCounts$counts,
    initialK = 3,
    maxK = 12,
    yInit = clusters(moduleSplitSelect)$y)
```

```
plotGridSearchPerplexity(celdaList = cellSplit)
```


In this plot, the perplexity for K stops decreasing at K = 5, with a final K/L combination of K = 5, L = 10. Generally, this method can be used to pick a reasonable L and a potential range of K. However, manual review of specific selections of K is often be required to ensure results are biologically coherent.

Once users have chosen the K/L parameters for further analysis, the `subsetCeldaList` function can be used to subset the celda_list object to a single model.

```
celdaModel <- subsetCeldaList(celdaList = cellSplit,
    params = list(K = 5, L = 10))
```

9.2   celdaGridSearch

celda is able to run multiple combinations of K and L with multiple chains in parallel via the `celdaGridSearch` function. Setting `verbose` to `TRUE` will print the output of each model to a text file.

The `resamplePerplexity` function "perturbs" the original counts matrix by resampling the counts of each cell according to its normalized probability distribution. Perplexity is calculated on the resampled matrix and the procedure is repeated `resample` times. These results can be visualized with `plotGridSearchPerplexity`. The major goal is to pick the lowest K and L combination with relatively good perplexity. In general, visual inspection of the plot can be used to select the number of modules (L) or cell populations (K) where the rate of decrease in the perplexity starts to drop off.

```
cgs <- celdaGridSearch(simCounts$counts,
    paramsTest = list(K = seq(4, 6), L = seq(9, 11)),
    cores = 1,
    model = "celda_CG",
    nchains = 2,
    maxIter = 100,
    verbose = FALSE,
    bestOnly = TRUE)
```

`bestOnly = TRUE` indicates that only the chain with the best log likelihood will be returned for each K/L combination.

`resamplePerplexity` calculates the perplexity of each model's cluster assignments, as well as resamplings of that count matrix. The result of this function can be visualized with `plotGridSearchPerplexity` for determination of the optimal K/L values.

```
cgs <- resamplePerplexity(counts = simCounts$counts,
    celdaList = cgs, resample = 5)
```

```
plotGridSearchPerplexity(celdaList = cgs)
```
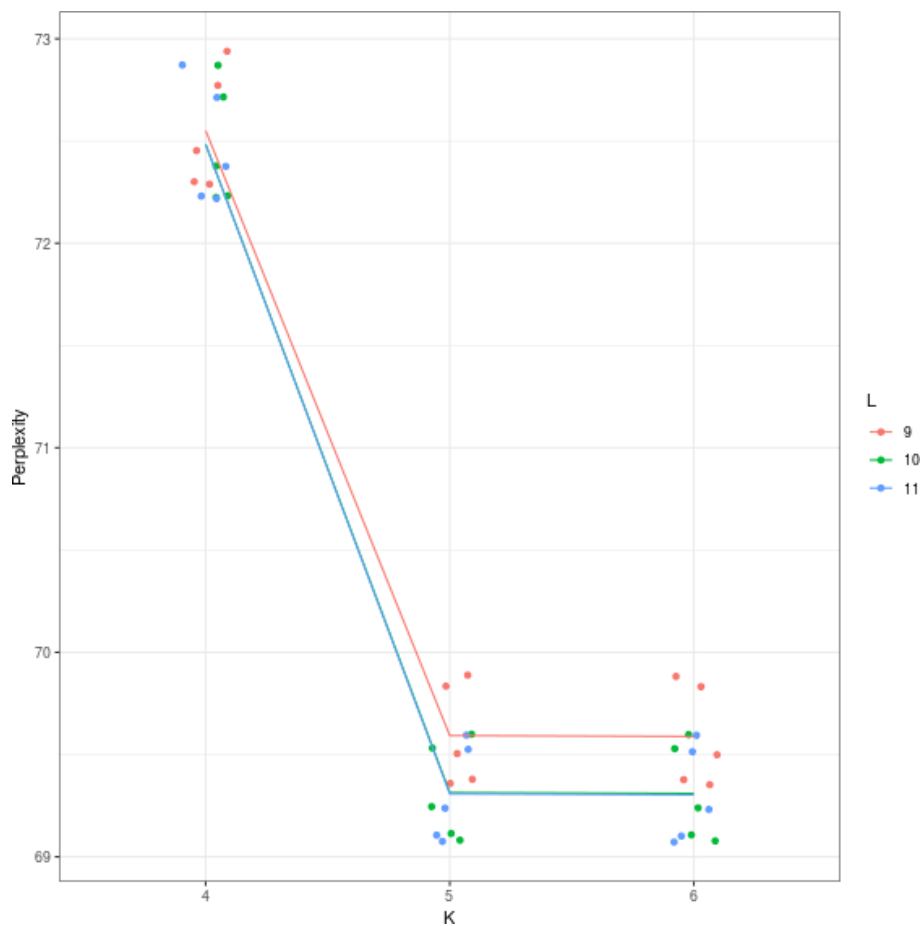
In this example, the perplexity for L stops decreasing at $L = 10$ for the majority of K values. For the line corresponding to $L = 10$, the perplexity stops decreasing at $K = 5$. Thus $L = 10$ and $K = 5$ would be a good choice. Again, manual review of specific selections of K is often be required to ensure results are biologically coherent.

Once users have chosen the K/L parameters for further analysis, the `subsetCeldaList` function can be used to subset the celda_list object to a single model.

```
celdaModel <- subsetCeldaList(celdaList = cgs, params = list(K = 5, L = 10))
```

If the "bestOnly" parameter is set to FALSE in the `celdaGridSearch`, then the `selectBest Model` function can be used to select the chains with the lowest log likelihoods within each combination of parameters. Alternatively, users can use select a specific chain by specifying the index within the `subsetCeldaList` function.

```
cgs <- celdaGridSearch(simCounts$counts,
    paramsTest = list(K = seq(4, 6), L = seq(9, 11)),
    cores = 1,
    model = "celda_CG",
    nchains = 2,
    maxIter = 100,
    verbose = FALSE,
    bestOnly = FALSE)
```

```
cgs <- resamplePerplexity(counts = simCounts$counts,
    celdaList = cgs,
    resample = 2)

cgsK5L10 <- subsetCeldaList(celdaList = cgs, params = list(K = 5, L = 10))

celdaModel1 <- selectBestModel(celdaList = cgsK5L10)
```

10   Miscellaneous utility functions

celda also contains several utility functions for the users' convenience.

10.1   featureModuleLookup

`featureModuleLookup` can be used to look up the module a specific feature was clustered to.

```
featureModuleLookup(counts = simCounts$counts, celdaMod = celdaModel,
    feature = c("Gene_99"))
## $Gene_99
## [1] 3
```

10.2   recodeClusterZ, recodeClusterY

`recodeClusterZ` and `recodeClusterY` allows the user to recode the cell and feature cluster labels, respectively.

```
celdaModelZRecoded <- recodeClusterZ(celdaMod = celdaModel,
    from = c(1, 2, 3, 4, 5), to = c(2, 1, 3, 4, 5))
```

The model prior to reordering cell labels compared to after reordering cell labels:

```
table(clusters(celdaModel)$z, clusters(celdaModelZRecoded)$z)
##
##      1  2  3  4  5
##   1  0 44  0  0  0
##   2 42  0  0  0  0
##   3  0  0 40  0  0
##   4  0  0  0 47  0
##   5  0  0  0  0 34
```

11 Session Information

```
sessionInfo()
## R version 4.0.2 (2020-06-22)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 18.04.4 LTS
##
## Matrix products: default
## BLAS:   /home/biocbuild/bbs-3.11-bioc/R/lib/libRblas.so
## LAPACK: /home/biocbuild/bbs-3.11-bioc/R/lib/libRlapack.so
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] celda_1.4.7     BiocStyle_2.16.1
##
## loaded via a namespace (and not attached):
##  [1] ggrepel_0.8.2             Rcpp_1.0.5
##  [3] lattice_0.20-41           prettyunits_1.1.1
##  [5] digest_0.6.25             foreach_1.5.0
##  [7] SingleCellExperiment_1.10.1 R6_2.4.1
##  [9] GenomeInfoDb_1.24.2       plyr_1.8.6
## [11] stats4_4.0.2              evaluate_0.14
## [13] httr_1.4.2                ggplot2_3.3.2
## [15] pillar_1.4.6              progress_1.2.2
## [17] zlibbioc_1.34.0           rlang_0.4.8
## [19] data.table_1.13.0         magick_2.4.0
## [21] S4Vectors_0.26.1          combinat_0.0-8
## [23] Matrix_1.2-18             rmarkdown_2.4
## [25] labeling_0.3              Rtsne_0.15
## [27] stringr_1.4.0             RcppEigen_0.3.3.7.0
## [29] RCurl_1.98-1.2            munsell_0.5.0
## [31] uwot_0.1.8                DelayedArray_0.14.1
## [33] compiler_4.0.2            xfun_0.18
## [35] pkgconfig_2.0.3           BiocGenerics_0.34.0
## [37] htmltools_0.5.0           tidyselect_1.1.0
## [39] SummarizedExperiment_1.18.2 tibble_3.0.3
## [41] gridExtra_2.3             GenomeInfoDbData_1.2.3
## [43] enrichR_2.1               bookdown_0.20
## [45] IRanges_2.22.2            codetools_0.2-16
## [47] matrixStats_0.57.0        withr_2.3.0
```

```
## [49] MCMCprecision_0.4.0    crayon_1.3.4
## [51] dplyr_1.0.2            bitops_1.0-6
## [53] MAST_1.14.0            grid_4.0.2
## [55] gtable_0.3.0           lifecycle_0.2.0
## [57] magrittr_1.5           scales_1.1.1
## [59] stringi_1.5.3          farver_2.0.3
## [61] XVector_0.28.0         reshape2_1.4.4
## [63] doParallel_1.0.15      ellipsis_0.3.1
## [65] vctrs_0.3.4            generics_0.0.2
## [67] rjson_0.2.20           RColorBrewer_1.1-2
## [69] iterators_1.0.12       tools_4.0.2
## [71] Biobase_2.48.0         glue_1.4.2
## [73] purrr_0.3.4            hms_0.5.3
## [75] abind_1.4-5            parallel_4.0.2
## [77] yaml_2.2.1             colorspace_1.4-1
## [79] BiocManager_1.30.10    GenomicRanges_1.40.0
## [81] knitr_1.30
```