

Analysis of single-cell genomic data with celda

***Sean Corbett¹, Yusuke Koga¹, Shiyi Yang¹, Zhe Wang¹,
and Joshua Campbell^{*1}***

¹Boston University School of Medicine

*camp@bu.edu

2021-05-19

Contents

1	Introduction	2
2	Installation	2
3	Reproducibility note	2
4	Generation of a simulated single cell dataset	3
5	Feature selection	3
6	Performing bi-clustering with celda	4
7	Matrix factorization	4
8	Visualization	6
8.1	Creating an expression heatmap using the celda <i>SCE</i> object	6
8.2	Plotting cell populations with tSNE	6
8.3	Displaying relationships between modules and cell populations	9
8.4	Examining co-expression with module heatmaps	10
9	Identifying the optimal number of cell subpopulations and feature modules	12
9.1	recursiveSplitModule/recursiveSplitCell	12
9.2	celdaGridSearch	14
10	Miscellaneous utility functions	16
10.1	featureModuleLookup	16
10.2	recodeClusterZ, recodeClusterY	16
11	Session Information	17

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.

Starting from Bioconductor release 3.12 (*celda* version 1.6.0), *celda* makes use of *SingleCellExperiment* (SCE) objects for passing inputs and outputs of most built-in functions. The old *celdaModel* (*celda_C*, *celda_G*, *celda_CG*) and *celdaList* objects are deprecated. These objects can be converted to SCE objects using the *celdatosce* function in *celda*.

```
library(celda)
library(SingleCellExperiment)
data(celdaCGSim, celdaCGMod, celdaCGGridSearchRes)
scegrid <- celdatosce(celdaCGGridSearchRes, celdaCGSim$counts)
scecg <- celdatosce(celdaCGMod, celdaCGSim$counts)
```

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")
```

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.

To simulate a dataset of 5 samples with 5 cell populations, 10 feature modules, 200 features, and between 30 to 50 cells per sample using `celda_CG` model:

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

Starting from `celda` version 1.5.6, `celda` makes use of the `SingleCellExperiment` package. The `counts` assay slot in `simsce` contains the counts matrix. The dimensions of counts matrix:

```
dim(assay(simsce, i = "counts"))
## [1] 200 207
```

Columns `celda_sample_label` and `celda_cell_cluster` in `colData(simsce)` contain sample labels and celda cell population cluster labels. Here are the numbers of cells in each subpopulation and in each sample:

```
table(colData(simsce)$celda_cell_cluster)
##
##  1  2  3  4  5
## 42 44 40 47 34
table(colData(simsce)$celda_sample_label)
##
## Sample_1 Sample_2 Sample_3 Sample_4 Sample_5
##      43      48      45      40      31
```

Column `celda_feature_module` in `rowData(simsce)` contains feature module labels. Here is the number of features in each feature module:

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

5 Feature selection

A simple heuristic feature selection is performed to reduce the size of features used for clustering. To speed up the process, only features with at least 3 counts in at least 3 cells are included in downstream clustering for this data. A subset `SingleCellExperiment` object with filtered features is stored in `altExp(simsce, "featureSubset")` slot by default.

```
simsce <- selectFeatures(simsce)
```

6 Performing bi-clustering with celda

There are currently three models within *celda* package: `celda_C` will cluster cells, `celda_G` will cluster features, and `celda_CG` will simultaneously cluster cells and features. Within the functions 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.

```
sce <- celda_CG(x = simsce, K = 5, L = 10, verbose = FALSE, nchains = 1)
```

Here is a comparison between the true cluster labels and the estimated cluster labels.

```
table(celdaClusters(sce), celdaClusters(simsce))
##
##      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
table(celdaModules(sce), celdaModules(simsce))
##
##      1  2  3  4  5  6  7  8  9 10
##  1  0 32  0  0  0  0  0  0  0
##  2 19  0  0  0  0  0  0  0  0
##  3  0  0 15  0  0  0  0  0  0
##  4  0  0  0 13  0  0  0  0  0
##  5  0  0  0  0 21  0  0  0  0
##  6  0  0  0  0  0 19  0  0  0
##  7  0  0  0  0  0  0 12  0  0
##  8  0  0  0  0  0  0 17  0  0
##  9  0  0  0  0  0  0  0  3  0
## 10  0  0  0  0  0  0  0  0 20
```

7 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(x = sce)
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.

Analysis of single-cell genomic data with celda

```
dim(factorized$proportions$cell)
## [1] 10 207
head(factorized$proportions$cell[, seq(3)], 5)
##      Cell_1      Cell_2      Cell_3
## L1 0.01736111 0.010544815 0.01125704
## L2 0.04861111 0.019332162 0.06378987
## L3 0.03125000 0.195079086 0.04878049
## L4 0.04861111 0.182776801 0.08255159
## L5 0.01388889 0.003514938 0.01688555
```

The `cellPopulation` contains each feature module's contribution to each of the cell population. 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 0.020187905 0.16188882 0.10562714 0.01449752 0.03615111
## L2 0.018136410 0.11215736 0.05490132 0.04719914 0.03653816
## L3 0.201403342 0.38323917 0.01944218 0.04011478 0.07853383
## L4 0.171820182 0.17977959 0.02550970 0.06612064 0.02388141
## L5 0.008770887 0.03145707 0.21115642 0.02250852 0.22871187
```

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

```
dim(factorized$proportions$module)
## [1] 171 10
head(factorized$proportions$module, 5)
##      L1 L2 L3      L4 L5      L6      L7 L8 L9      L10
## Gene_1 0.00000000 0 0 0.1539361 0 0.00000000 0.00000000 0 0 0.00000000
## Gene_2 0.00000000 0 0 0.0000000 0 0.00000000 0.00000000 0 0 0.0347646
## Gene_3 0.00000000 0 0 0.0000000 0 0.04701052 0.00000000 0 0 0.00000000
## Gene_5 0.00000000 0 0 0.0000000 0 0.00000000 0.05690496 0 0 0.00000000
## Gene_7 0.03878533 0 0 0.0000000 0 0.00000000 0.00000000 0 0 0.00000000
```

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)
```

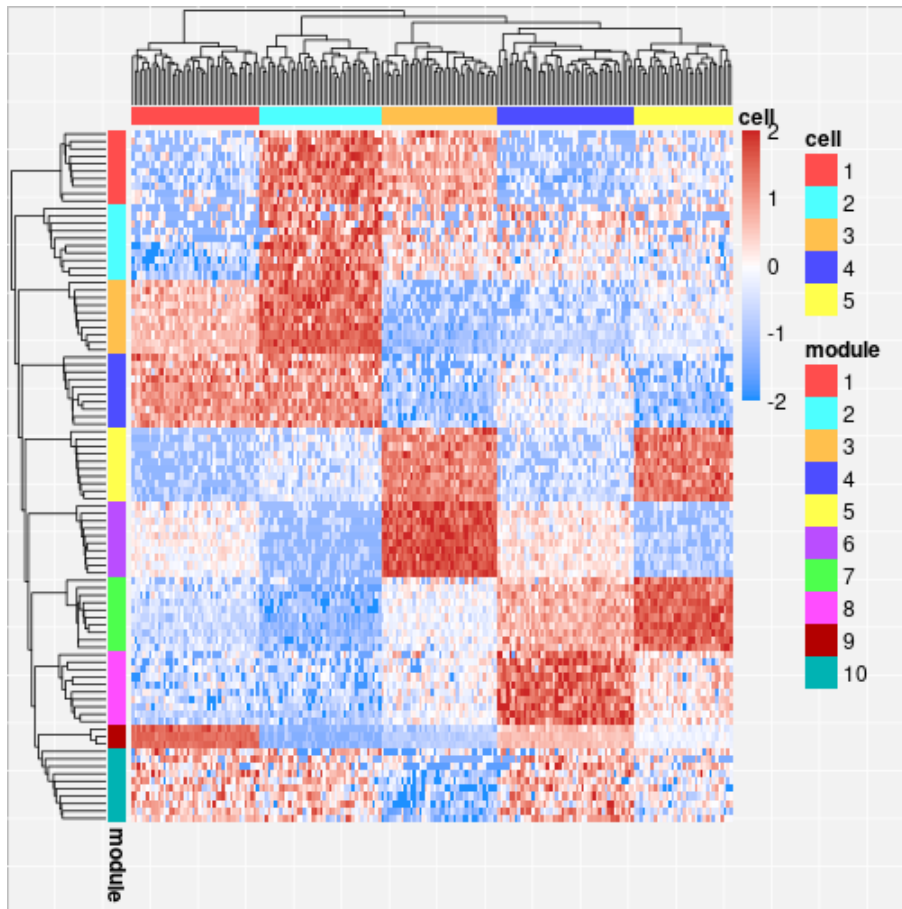
```
topGenes$names$L1
## [1] "Gene_175" "Gene_20" "Gene_180" "Gene_132" "Gene_112" "Gene_134"
## [7] "Gene_10" "Gene_191" "Gene_32" "Gene_90"
```

8 Visualization

8.1 Creating an expression heatmap using the celda *SCE* object

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.

```
plot(celdaHeatmap(sce = sce, nfeatures = 10))
```



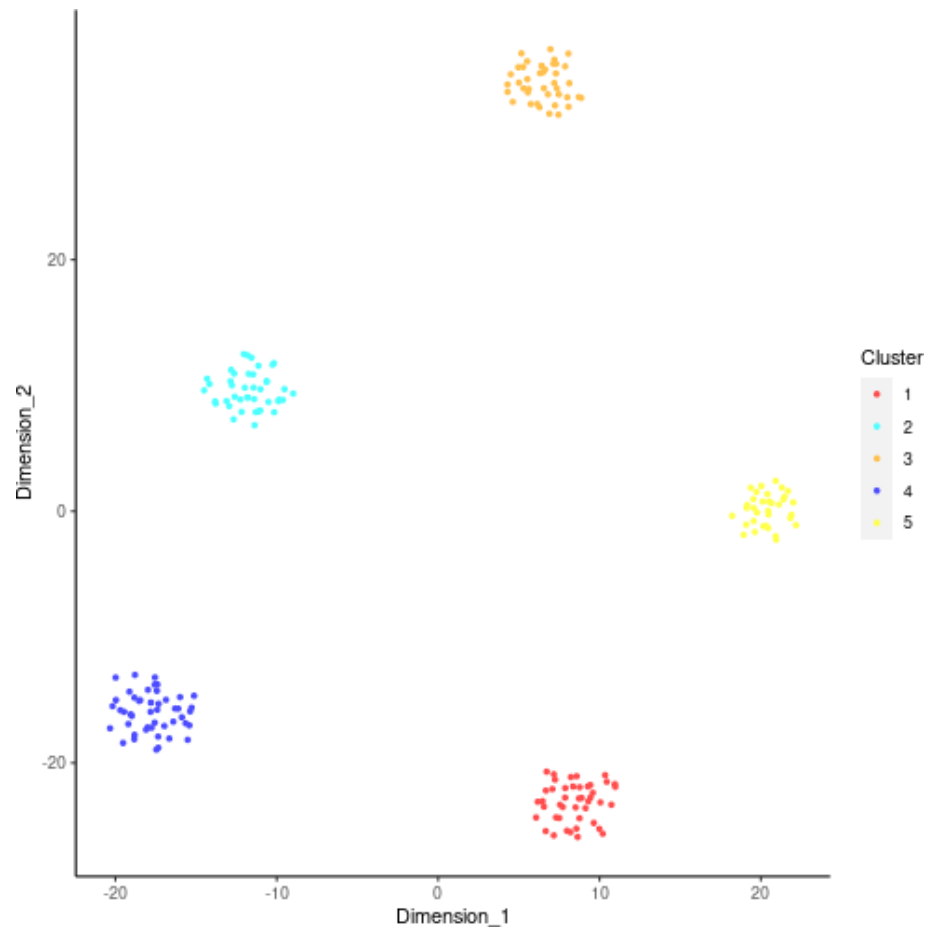
8.2 Plotting cell populations with tSNE

`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 `plotDimReduceCluster`, `plotDimReduceModule`, and `plotDimReduceFeature` to show cell population clusters, module probabilities, and expression of individual features, respectively.

```
sce <- celdaTsne(sce)
```

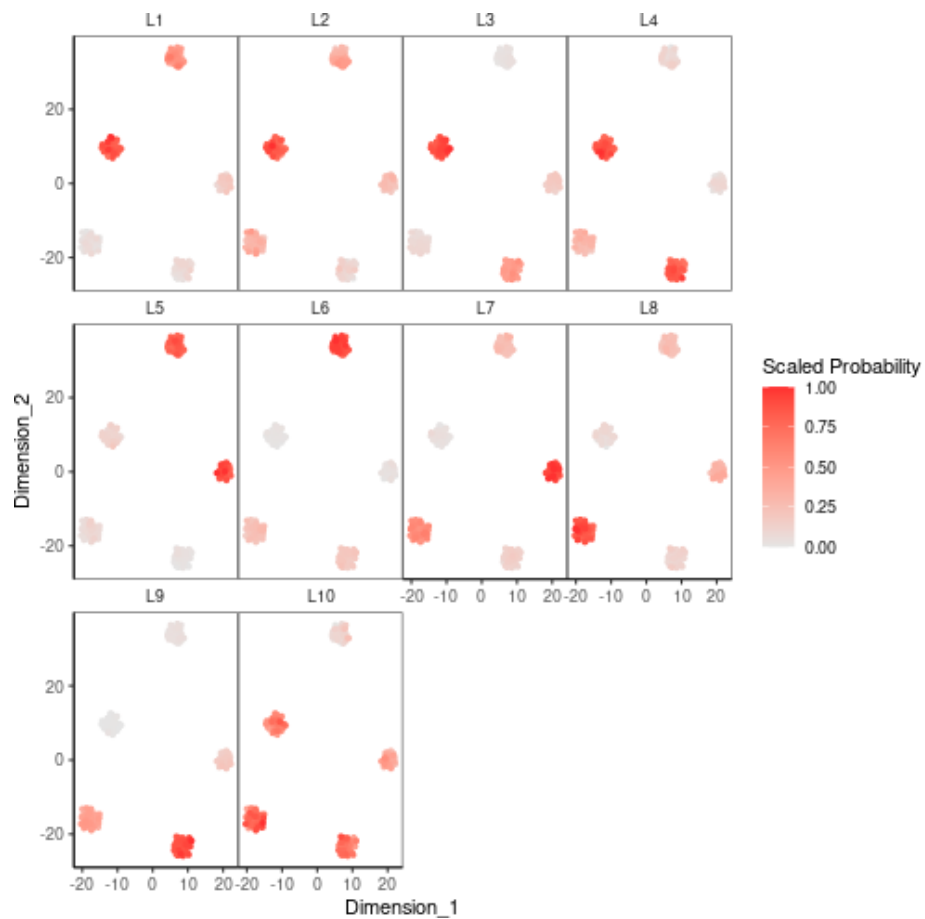
```
plotDimReduceCluster(x = sce, reducedDimName = "celda_tSNE")
```

Analysis of single-cell genomic data with celda

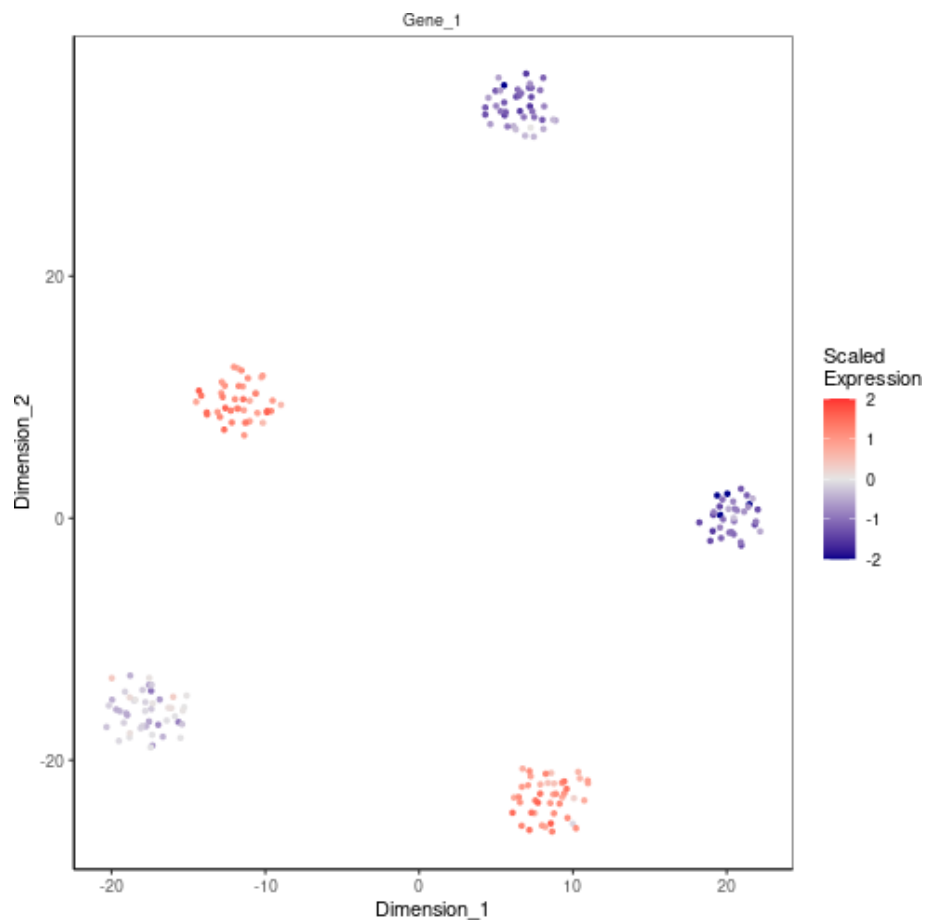


```
plotDimReduceModule(x = sce, reducedDimName = "celda_tSNE", rescale = TRUE)
```

Analysis of single-cell genomic data with celda



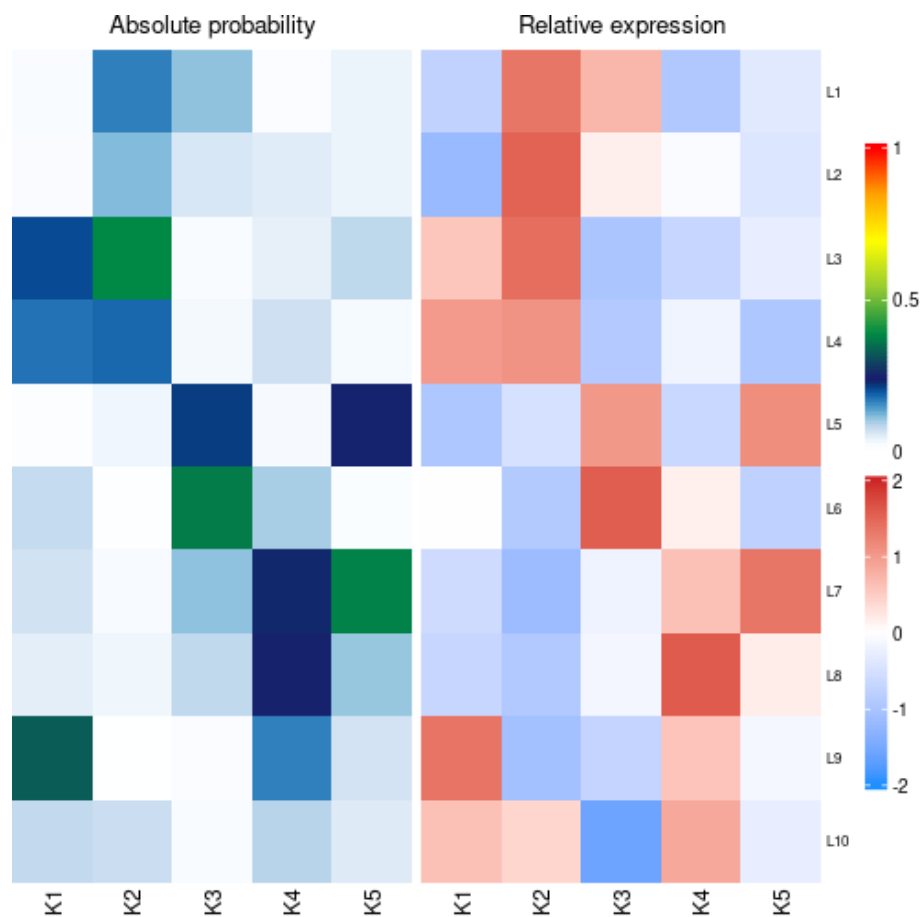
```
plotDimReduceFeature(x = sce, reducedDimName = "celda_tSNE",  
  normalize = TRUE, features = "Gene_1")
```

8.3 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(sce)
```

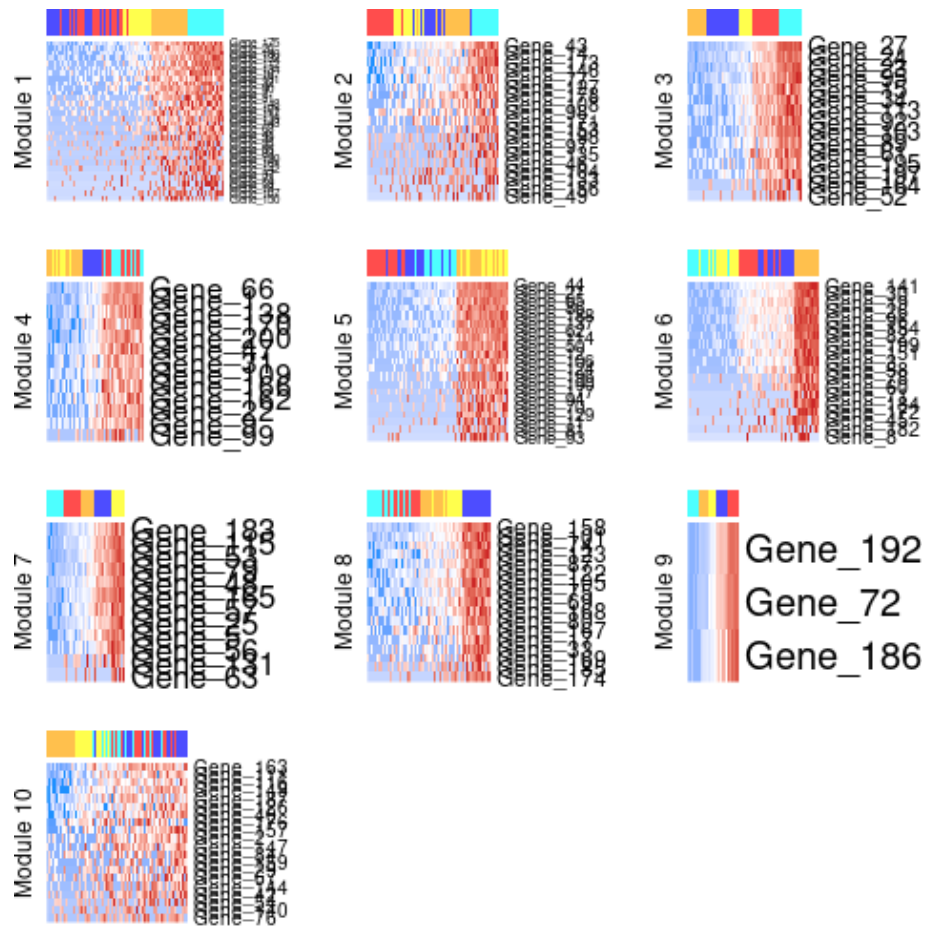


8.4 Examining co-expression with module heatmaps

`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(sce, featureModule = seq(10), topCells = 100)
```

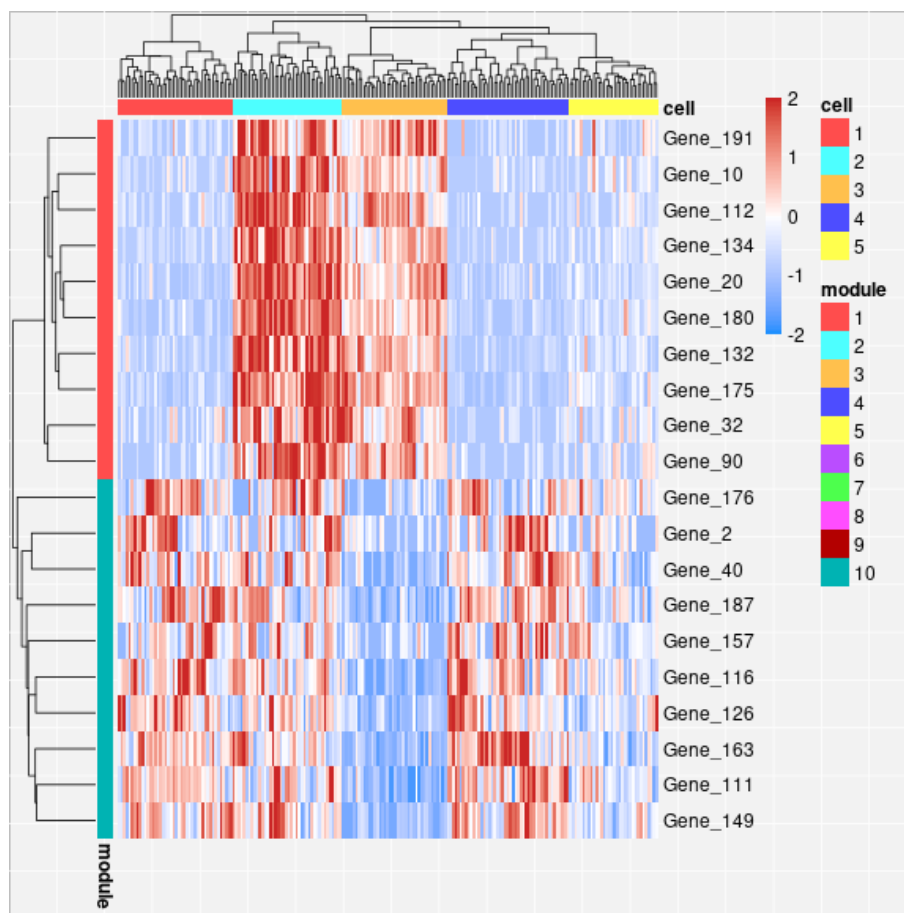
Analysis of single-cell genomic data with celda



While `celdaHeatmap` will plot a heatmap directly with a `celda SCE` 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 `cellIx` parameters can be used to select rows and columns out of the matrix.

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

```
plot(plotHeatmap(counts = normCounts,
  z = celdaClusters(sce),
  y = celdaModules(sce),
  featureIx = geneIx,
  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 .

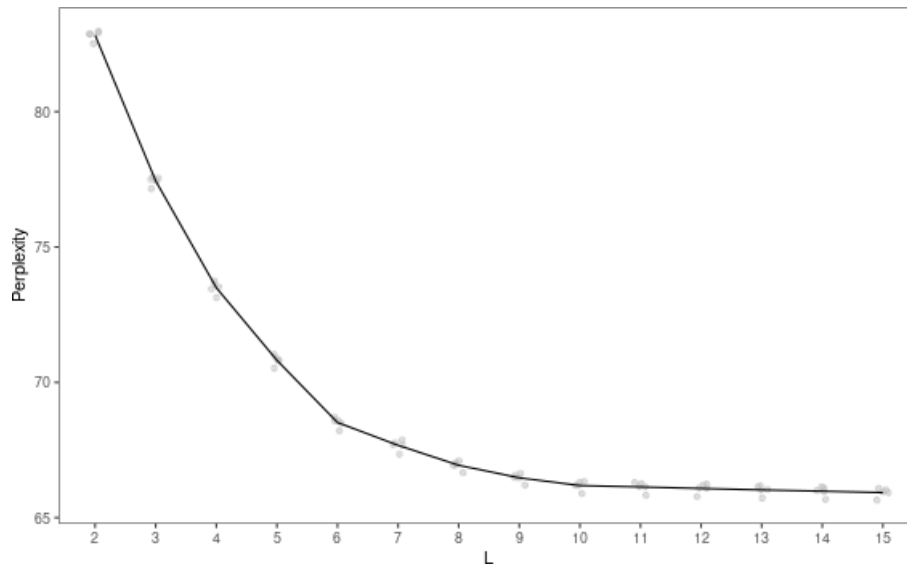
First, `recursiveSplitModule` is used to determine the optimal L . `recursiveSplitModule` first splits features into however many modules are specified in `initialL`. The module labels are then recursively split in a way that would generate the highest log-likelihood, all the way up to `maxL`.

Analysis of single-cell genomic data with celda

```
moduleSplit <- recursiveSplitModule(simsce, 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(moduleSplit)
```



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

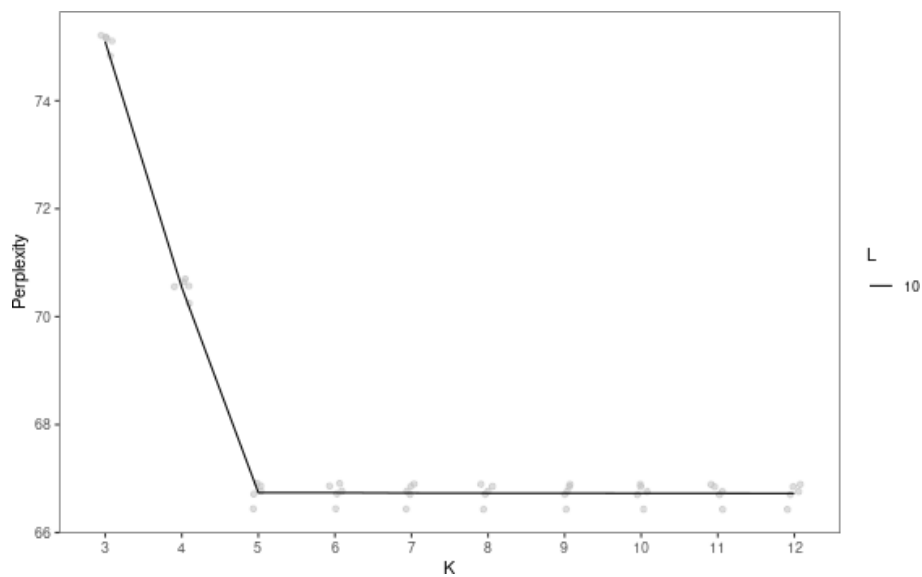
Once you have identified the optimal L (in this case, L is selected to be 10), the module labels are used for initialization in `recursiveSplitCell`. Similarly to `recursiveSplitModule`, cells are initially split into a small number of subpopulations, and the subpopulations are recursively split up by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

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

```
cellSplit <- recursiveSplitCell(moduleSplitSelect,  
  initialK = 3,  
  maxK = 12,  
  yInit = celdaModules(moduleSplitSelect))
```

```
plotGridSearchPerplexity(cellSplit)
```

Analysis of single-cell genomic data with celda



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 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 *SCE* object to a single model *SCE* object.

```
sce <- subsetCeldaList(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(simsce,
  paramsTest = list(K = seq(4, 6), L = seq(9, 11)),
  cores = 1,
  model = "celda_CG",
  nchains = 2,
  maxIter = 100,
  verbose = FALSE,
  bestOnly = TRUE)
```

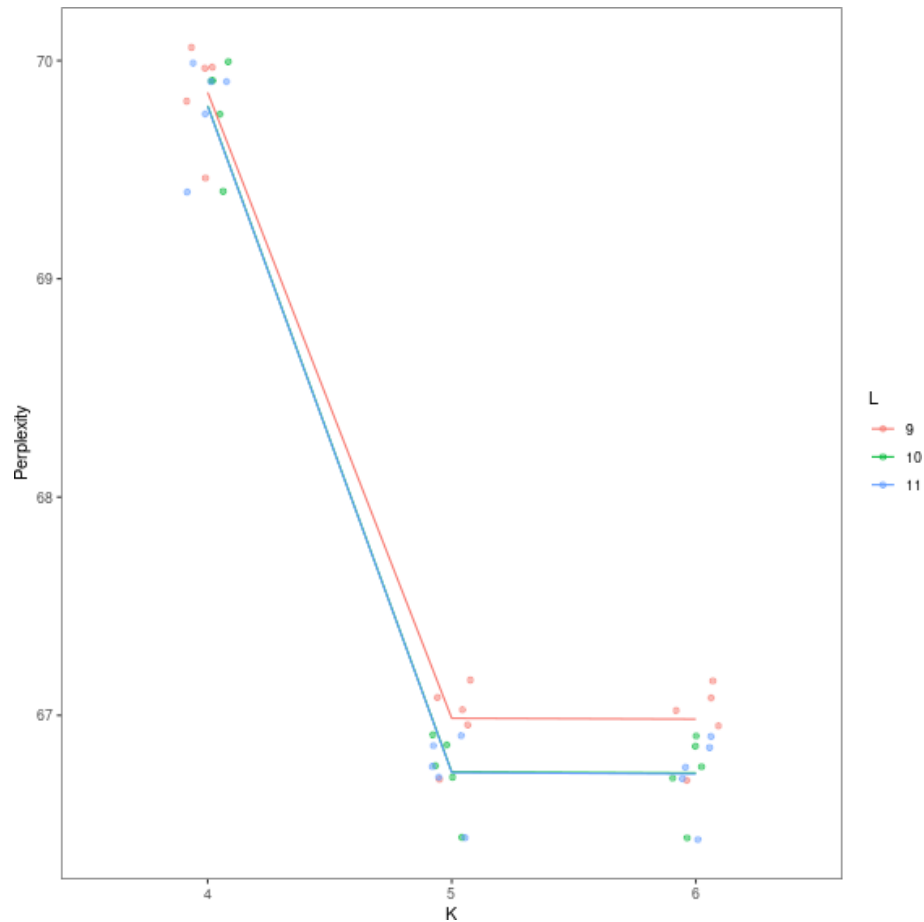
Analysis of single-cell genomic data with celda

`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(cgs, resample = 5)
```

```
plotGridSearchPerplexity(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 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 `SCE` object to a single model `SCE` object.

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

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

```
cgs <- celdaGridSearch(simsce,
  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(cgs, celdaList = cgs, resample = 2)

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

sce <- selectBestModel(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(sce, feature = c("Gene_99"))
## $Gene_99
## [1] 4
## Levels: 1 2 3 4 5 6 7 8 9 10
```

10.2 recodeClusterZ, recodeClusterY

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

```
sceZRecoded <- recodeClusterZ(sce,
  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(celdaClusters(sce), celdaClusters(sceZRecoded))
##
##      2  1  3  4  5
## 1 44  0  0  0  0
## 2  0 42  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 4.1.0 (2021-05-18)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 20.04.2 LTS
##
## Matrix products: default
## BLAS: /home/biocbuild/bbs-3.13-bioc/R/lib/libRblas.so
## LAPACK: /home/biocbuild/bbs-3.13-bioc/R/lib/libRlapack.so
##
## locale:
## [1] LC_CTYPE=en_US.UTF-8 LC_NUMERIC=C
## [3] LC_TIME=en_GB 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] parallel stats4 stats graphics grDevices utils datasets
## [8] methods base
##
## other attached packages:
## [1] SingleCellExperiment_1.14.0 SummarizedExperiment_1.22.0
## [3] Biobase_2.52.0 GenomicRanges_1.44.0
## [5] GenomeInfoDb_1.28.0 IRanges_2.26.0
## [7] S4Vectors_0.30.0 BiocGenerics_0.38.0
## [9] MatrixGenerics_1.4.0 matrixStats_0.58.0
## [11] celda_1.8.0 BiocStyle_2.20.0
##
## loaded via a namespace (and not attached):
## [1] bitops_1.0-7 doParallel_1.0.16
## [3] RColorBrewer_1.1-2 httr_1.4.2
## [5] tools_4.1.0 utf8_1.2.1
## [7] R6_2.5.0 uwot_0.1.10
## [9] DBI_1.1.1 colorspace_2.0-1
## [11] GetoptLong_1.0.5 withr_2.4.2
## [13] tidyselect_1.1.1 compiler_4.1.0
## [15] Cairo_1.5-12.2 assertive.properties_0.0-4
## [17] enrichR_3.0 DelayedArray_0.18.0
## [19] labeling_0.4.2 assertive.files_0.0-2
## [21] bookdown_0.22 scales_1.1.1
## [23] stringr_1.4.0 digest_0.6.27
## [25] rmarkdown_2.8 XVector_0.32.0
```

Analysis of single-cell genomic data with celda

```
## [27] assertive.numbers_0.0-2      pkgconfig_2.0.3
## [29] htmltools_0.5.1.1           rlang_0.4.11
## [31] GlobalOptions_0.1.2         shape_1.4.6
## [33] gridGraphics_0.5-1          generics_0.1.0
## [35] farver_2.1.0                 combinat_0.0-8
## [37] dplyr_1.0.6                  RCurl_1.98-1.3
## [39] magrittr_2.0.1              GenomeInfoDbData_1.2.6
## [41] multipanelfigure_2.1.2      Matrix_1.3-3
## [43] Rcpp_1.0.6                   munsell_0.5.0
## [45] fansi_0.4.2                  lifecycle_1.0.0
## [47] stringi_1.6.2               assertive.base_0.0-9
## [49] yaml_2.2.1                   MCMCprecision_0.4.0
## [51] zlibbioc_1.38.0             Rtsne_0.15
## [53] plyr_1.8.6                   grid_4.1.0
## [55] ggrepel_0.9.1               crayon_1.4.1
## [57] lattice_0.20-44             circlize_0.4.12
## [59] magick_2.7.2                knitr_1.33
## [61] ComplexHeatmap_2.8.0        pillar_1.6.1
## [63] rjson_0.2.20                reshape2_1.4.4
## [65] codetools_0.2-18           glue_1.4.2
## [67] evaluate_0.14               data.table_1.14.0
## [69] BiocManager_1.30.15         vctrs_0.3.8
## [71] png_0.1-7                   foreach_1.5.1
## [73] gtable_0.3.0                purrr_0.3.4
## [75] clue_0.3-59                 assertthat_0.2.1
## [77] ggplot2_3.3.3               xfun_0.23
## [79] assertive.types_0.0-3       RcppEigen_0.3.3.9.1
## [81] tibble_3.1.2                iterators_1.0.13
## [83] cluster_2.1.2               ellipsis_0.3.2
```