ccfindR: single-cell RNA-seq analysis using non-negative matrix factorization

2018-03-12

The `ccfindR` (Cancer Clone findeR) package contains implementations and utilities for analyzing single-cell RNA-sequencing data, including quality control, unsupervised clustering for discovery of cell types, and visualization of the outcomes. It is especially suitable for analysis of transcript-count data utilizing unique molecular identifiers (UMIs), e.g., data derived from 10x Genomics platform. In these data sets, RNA counts are non-negative integers, enabling clustering using non-negative matrix factorization (NMF)[1].

Input data are UMI counts in the form of a matrix with each genetic feature ("genes") in rows and cells (tagged by barcodes) in columns, produced by read alignment and counting pipelines. The count matrix and associated gene and cell annotation files are bundled into a main object of class `scNMFSet`. Quality control for both cells and genes can be performed via filtering steps based on UMI counts and variance of expressions, respectively. The NMF factorization is first performed for multiple values of ranks (the reduced dimension of factorization) to find the most likely value. A production run for the chosen rank then leads to factor matrices, allowing the user to identify and visualize genes representative of clusters and assign cells into clusters.

Algorithm

The NMF approach offers a means to identify cell subtypes and classify individual cells into these clusters based on clustering using expression counts. In contrast to alternatives such as principal component analyses[2], NMF leverages the non-negative nature of count data and factorizes the data matrix X into two matrices W and H[1]:

$$\mathsf{X} \sim \mathsf{WH}.$$

If X is a $p \times n$ matrix (p genes and n cells), the basis matrix W is $p \times r$ and coefficient matrix H is $r \times n$ in dimension, respectively, where the rank r is a relatively small integer. A statistical inference-based interpretation of NMF is to view X_{ij} as a realization of a Poisson distribution with the mean for each matrix elements given by $(\mathsf{WH})_{ij} \equiv \Lambda_{ij}$, or

$$\Pr(x_{ij}) = \frac{e^{-\Lambda_{ij}} {\Lambda_{ij}}^{x_{ij}}}{\Gamma(1 + x_{ij})}.$$

The maximum likelihood inference of the latter is then achieved by maximizing

$$L = \sum_{ij} \left(X_{ij} \ln \frac{\Lambda_{ij}}{X_{ij}} - \Lambda_{ij} + X_{ij} \right).$$

The Kullback-Leibler measure of the distance between X and Λ, which is minimized, is equal to $-L$. Lee and Seung's update rule[1] solves this optimization task iteratively. In addition to this classical iterative update algorithm to find basis and coefficient factors of the count matrix, the `ccfindR` package implements variational Bayesian inference developed by Cemgil[3].

Key features of `ccfindR` distinguishing it from other existing implementations – `NMF` for generic data[4] and `NMFEM` for single-cell analysis[5] – are

- Bayesian inference allowing for a statistically well-controlled procedure to determine the most likely value of rank r.
- Procedure to derive hierarchical relationships among clusters identified under different ranks.

A traditional way (in maximum likelihood inference) to determine the rank is to evaluate the factorization quality measures (and optionally compare with those from randomized data). The Bayesian formulation of NMF algorithm[3] incorporates priors for factored matrix elements W and H modeled by gamma distributions. Inference can be combined with hyperparameter update to optimize the evidence (conditional probability of data under hyperparameters and rank), which provides a statistically well-controlled means to determine the optimal rank describing data.

For large rank values, it can be challenging to interpret clusters identified. To facilitate biological interpretation, we provide a procedure where cluster assignment of cells is repeated for multiple rank values, typically ranging from 2 to the optimal rank, and a phylogenetic tree connecting different clusters at neighboring rank values are constructed. This tree gives an overview of different types of cells present in the system viewed at varying resolution.

Workflow

We illustrate a typical workflow with a single-cell count data set generated from peripheral blood mononuclear cell (PBMC) data available from https://support.10xgenomics.com/single-cell-gene-expression/datasets/1.1.0/. The particular data set used below was created by sampling from 5 purified immune cell subsets.

1. Installation

To install the package, download the source tar-ball and

```
$ R CMD INSTALL ccfindR_0.1-6.tar.gz
```

After installation, load the package by

```
library(ccfindR)
```

```
## Package 'ccfindR' version 0.9.0
```

2. Data input

The input data can be a simple matrix:

```
# A toy matrix for count data
set.seed(1)
mat <- matrix(rpois(n = 80, lambda = 2), nrow = 4, ncol = 20)
ABC <- LETTERS[1:4]
abc <- letters[1:20]
```

The main `S4` object containing data and subsequent analysis outcomes is of class `scNMFSet`, created by

```
# create scNMFSet object
sc <- scNMFSet(count = mat)
```

There are three main data slots in an `scNMFSet` object: `count`, `genes`, and `cells`. The count matrix is stored in `count` and the other two slots can store data frames annotating genes and cells. In the simplest initialization above, the omitted `genes` and `cells` slots are filled from the row and column names of `mat`. If these names are missing in `mat`, we recommend filling them before input as above.

Extra annotation columns in gene and cell-lists are read by

```
# read count and annotations
genes <- data.frame(ABC)
rownames(genes) <- ABC
```

```
cells <- data.frame(abc)
rownames(cells) <- abc
sc <- scNMFSet(count=mat,genes=genes,cells=cells)
sc
```

```
## An object of class scNMFSet
##  4 genes by 20 cells.
```

The number of rows in `count` and `genes`, the number of columns in `count` and rows in `cells` must match.

Alternatively, sparse matrix format can be used. The main count data in an `scNMFSet` object is in fact of class `dgCMatrix`, to which `matrix` or `data.frame` objects are cast internally. One may read a `MatrixMarket` format file directly:

```
# read sparse matrix
mat <- Matrix::readMM('pbmc/matrix.mtx')
rownames(mat) <- 1:nrow(mat)
colnames(mat) <- 1:ncol(mat)
sc <- scNMFSet(count=mat)
sc
```

```
## An object of class scNMFSet
##  11727 genes by 450 cells.
```

The gene and barcode meta-data and count files resulting from 10x Genomics' Cell Ranger pipeline can also be read:

```
# read 10x files
sc <- read_10x(dir = 'pbmc', count = 'matrix.mtx',
               genes = 'genes.tsv', barcodes = 'barcodes.tsv')
sc
```

```
## An object of class scNMFSet
##  11727 genes by 450 cells.
```

The parameter `dir` is the directory containing the files. File names shown above are defaults and can be omitted. The function returns an `scNMFSet` object. By default, any row or column entirely consisting of zeros in `count` and the corresponding elements in `genes` and `cells` slots will be removed. This feature can be turned off by `remove.zeros = FALSE`.

3. Quality control

For quality control, cells and genes can be filtered manually using normal subsetting syntax of `R`: the slots in the object `sc` are accessed and edited, for example, by

```
# slots and subsetting
sc@count[1:7,1:3]
```

```
## 7 x 3 sparse Matrix of class "dgCMatrix"
##   1 2 3
## 1 . . .
## 2 . . .
## 3 . . .
## 4 . . .
## 5 . . .
## 6 . . .
## 7 . 2 .
```

```
count(sc)[7,2] <- 0
count(sc)[1:7,1:3]
```

```
## 7 x 3 sparse Matrix of class "dgCMatrix"
##   1 2 3
## 1 . . .
## 2 . . .
## 3 . . .
## 4 . . .
## 5 . . .
## 6 . . .
## 7 . . .
```

```
count(sc)[7,2] <- 2
head(genes(sc))
```

```
##                                V1            V2
## ENSG00000237683 ENSG00000237683    AL627309.1
## ENSG00000228327 ENSG00000228327 RP11-206L10.2
## ENSG00000225880 ENSG00000225880     LINC00115
## ENSG00000230368 ENSG00000230368        FAM41C
## ENSG00000188976 ENSG00000188976         NOC2L
## ENSG00000188290 ENSG00000188290          HES4
```

```
head(sc@cells)
```

```
##                                  V1
## ATGCAGTGCTTGGA-1 ATGCAGTGCTTGGA-1
## CATGTACTCCATGA-1 CATGTACTCCATGA-1
## GAGAAATGGCAAGG-1 GAGAAATGGCAAGG-1
## TGATATGACGTTAG-1 TGATATGACGTTAG-1
## AGTAGGCTCGGGAA-1 AGTAGGCTCGGGAA-1
## TGACCGCTGTAGCT-1 TGACCGCTGTAGCT-1
```

```
sc2 <- sc[1:20,1:70]  # subsetting of object
```

```
## 10 empty genes removed
## 26 empty cells removed
```

```
sc2@genes
```

```
##                                V1       V2
## ENSG00000188976 ENSG00000188976    NOC2L
## ENSG00000187608 ENSG00000187608    ISG15
## ENSG00000186891 ENSG00000186891 TNFRSF18
## ENSG00000078808 ENSG00000078808     SDF4
## ENSG00000160087 ENSG00000160087   UBE2J2
## ENSG00000131584 ENSG00000131584    ACAP3
## ENSG00000169972 ENSG00000169972    PUSL1
## ENSG00000127054 ENSG00000127054   CPSF3L
## ENSG00000175756 ENSG00000175756 AURKAIP1
## ENSG00000221978 ENSG00000221978    CCNL2
```

We provide two streamlined functions each for cell and gene filtering as shown below:

```
sc <- filter_cells(sc, umi.min = 10^2.8, umi.max = 10^3.6)
```

```
## 410 cells out of 450 selected
```

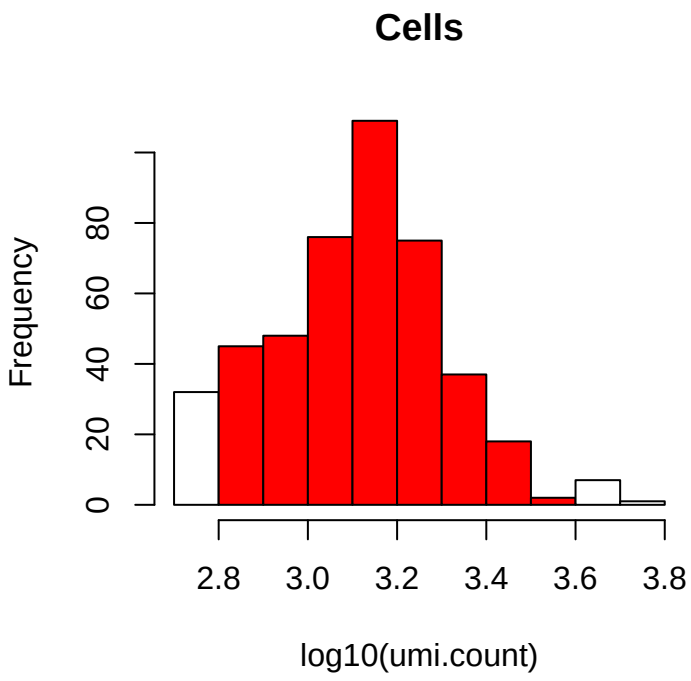

Figure 1: Quality control filtering of cells. Histogram of UMI counts is shown. Cells can be selected (red) by setting lower and upper thresholds of the UMI count.

```
## 255 empty genes removed
```

```
markers <- c('CD4','CD8A','CD8B','CD19','CD3G','CD3D',
             'CD3Z','CD14')
sc2 <- filter_genes(sc, markers = markers, vmr.min = 2, min.cells.expressed = 100,
                    rescue.genes = FALSE)
```

```
## 7 marker genes found
## 125 variable genes out of 11472
## 132 genes selected
```

The function `filter_cells()` plots histogram of UMI counts for all cells when called without threshold parameters (Fig. 1). This plot can be used to set desirable thresholds, `umi.min` and `umi.max`. Cells with UMI counts outside will be filtered out. The function `filter_genes()` displays scatter plot of the total number of cells with nonzero count and VMR (variance-to-mean ratio) for each gene (Fig. 2). In both plots, selected cells and genes are shown in red. Note that the above example has thresholds that are too stringent, which is intended to speed up the subsequent illustrative runs. A list of pre-selected marker genes can be provided to help identify clusters via the `markers` parameter in `filter_genes()`. Here, we use a set of classical PBMC marker genes (shown in orange).

Gene-filtering can also be augmented by scanning for those genes whose count distributions among cells are non-trivial: most have zero count as its maximum; some have one or more distinct peaks at nonzero count values. These may signify the existence of groups of cells in which the genes are expressed in distinguishable fasion. The selection of genes by `filter_genes()` will be set as the union of threshold-based group and those with such nonzero-count modes by setting `rescue.genes = TRUE` (default):

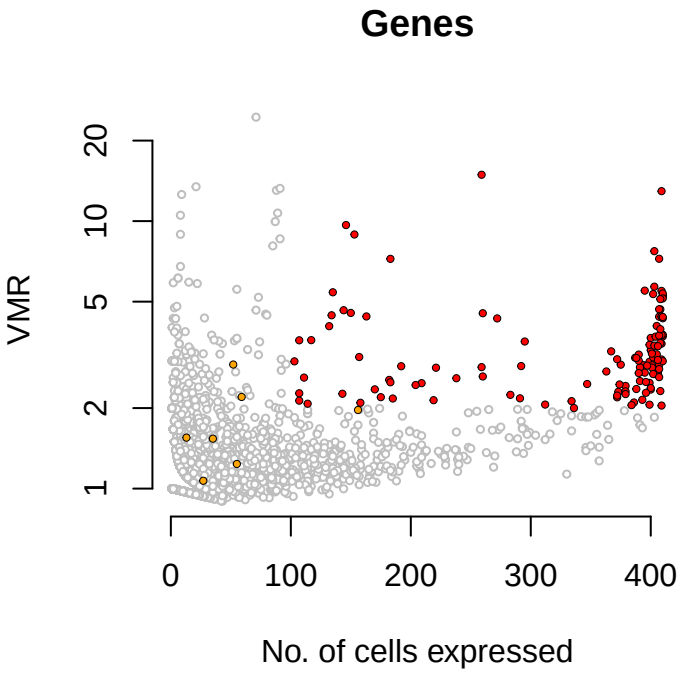

Figure 2: Selection of genes for clustering. The scatter plot shows distributions of expression variance to mean ratio (VMR) and the number of cells expressed. Minimum VMR and a range of cell number can be set to select genes (red). Symbols in orange are marker genes provided as input, selected irrespective of expression variance.

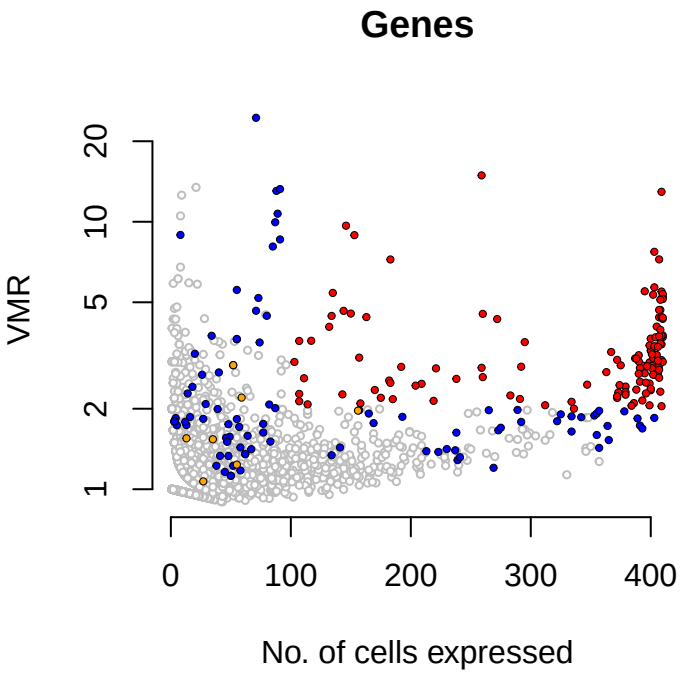

Figure 3: Additional selection of genes with modes at nonzero counts. Symbols in blue represent genes rescued.

```
sc_rescue <- filter_genes(sc, markers = markers, vmr.min = 2, min.cells.expressed = 100,
                          rescue.genes = TRUE, progress.bar = FALSE)
```

```
## Scanning genes for those with modes...
```

```
## 7 marker genes found
## 125 variable genes out of 11472
## 102 additional genes rescued
## 227 genes selected
```

This "gene rescue" scan will take some time and a progress bar is displayed if `progress.bar = TRUE`.

For subsequent analysis, we will use the latter selection and also name rows with gene symbols:

```
rownames(genes(sc_rescue)) <- rownames(count(sc_rescue)) <- genes(sc_rescue)[,2]
sc <- sc_rescue
```

4. Rank determination

The main function for maximum likelihood NMF on a count matrix is `factorize()`. It performs a series of iterative updates to matrices W and H. Since the global optimum of likelihood function is not directly accessible, computational inference relies on local maxima, which depends on initializations. We adopt the randomized initialization scheme, where the factor matrix elements are drawn from uniform distributions. To make the inference reproducible, one can set the random number seed by `set.seed(seed)`, where `seed` is a positive integer, prior to calling `factorize()`. Updates continue until convergence is reached, defined by either the fractional change in likelihood being smaller than `Tol` (`criterion = likelihood`) or a set

number (`ncnn.step`) of steps observed during which the connectivity matrix remains unchanged (`criterion = connectivity`). The connectivity matrix C is a symmetric $n \times n$ matrix with elements $C_{jl} = 1$ if j and l cells belong to the same cluster and 0 otherwise. The cluster membership is dynamically checked by finding the row index k for which the coefficient matrix element H_{kj} is maximum for each cell indexed by j.

During iteration, with `verbose = 3`, step number, log likelihood per elements, and the number of terms in the upper-diagonal part of C that changed from the previous step are printed:

```
set.seed(2)
sc <- factorize(sc, ranks = 3, nrun = 1, ncnn.step = 1,
                criterion='connectivity', verbose = 3)
```

```
## Rank 3
## Run # 1 :
## 1 : likelihood =  -0.7147593 , connectivity change =  83845
## 2 : likelihood =  -0.6870366 , connectivity change =  5335
## 3 : likelihood =  -0.6696675 , connectivity change =  4243
## 4 : likelihood =  -0.6568376 , connectivity change =  7600
## 5 : likelihood =  -0.6455727 , connectivity change =  6998
## 6 : likelihood =  -0.6340417 , connectivity change =  5774
## 7 : likelihood =  -0.6210451 , connectivity change =  7862
## 8 : likelihood =  -0.6060978 , connectivity change =  7020
## 9 : likelihood =  -0.5896878 , connectivity change =  4530
## 10 : likelihood =  -0.5729525 , connectivity change =  4620
## 11 : likelihood =  -0.5567839 , connectivity change =  3990
## 12 : likelihood =  -0.5414567 , connectivity change =  3974
## 13 : likelihood =  -0.5269475 , connectivity change =  2994
## 14 : likelihood =  -0.5132586 , connectivity change =  3498
## 15 : likelihood =  -0.5004836 , connectivity change =  1988
## 16 : likelihood =  -0.4887549 , connectivity change =  2671
## 17 : likelihood =  -0.4782006 , connectivity change =  620
## 18 : likelihood =  -0.4689458 , connectivity change =  2131
## 19 : likelihood =  -0.4611105 , connectivity change =  1854
## 20 : likelihood =  -0.4547479 , connectivity change =  618
## 21 : likelihood =  -0.4497727 , connectivity change =  1244
## 22 : likelihood =  -0.4459773 , connectivity change =  620
## 23 : likelihood =  -0.443105 , connectivity change =  1266
## 24 : likelihood =  -0.4409113 , connectivity change =  313
## 25 : likelihood =  -0.4391978 , connectivity change =  1911
## 26 : likelihood =  -0.4378228 , connectivity change =  654
## 27 : likelihood =  -0.4366946 , connectivity change =  948
## 28 : likelihood =  -0.4357547 , connectivity change =  318
## 29 : likelihood =  -0.4349631 , connectivity change =  330
## 30 : likelihood =  -0.4342888 , connectivity change =  486
## 31 : likelihood =  -0.4337062 , connectivity change =  0
## Nsteps = 31 , likelihood = -0.4337062 , dispersion = 1
##
## Sample# 1 : Max(likelihood) = -0.4337062 , dispersion = 1 , cophenetic = 1
```

The function `factorize()` returns the same object `sc` with extra slots `ranks` (the rank value for which factorization was performed), `basis` (a list containing the basis matrix W), `coeff` (a list containing the coefficient matrix H), and `measure` (a data frame containing the factorization quality measure; see below). The `criterion` used to stop iteration is either `connectivity` (no changes to connectivity matrix for `ncnn.steps`) or `likelihood` (changes to likelihood smaller than `Tol`).

To reduce the dependence of final estimates for W and H on initial guess, inferences need to be repeated for

many different initializations:

```
sc <- factorize(sc, ranks = 3, nrun = 10, verbose = 2)
```

```
## Rank 3
## Run # 1 :
## Nsteps = 71 , likelihood = -0.4264638 , dispersion = 1
##
## Run # 2 :
## Nsteps = 113 , likelihood = -0.4264588 , dispersion = 0.9643546
##
## Run # 3 :
## Nsteps = 94 , likelihood = -0.4285751 , dispersion = 0.9320193
##
## Run # 4 :
## Nsteps = 92 , likelihood = -0.4258037 , dispersion = 0.9262939
##
## Run # 5 :
## Nsteps = 51 , likelihood = -0.4264938 , dispersion = 0.9279477
##
## Run # 6 :
## Nsteps = 89 , likelihood = -0.4260693 , dispersion = 0.9349593
##
## Run # 7 :
## Nsteps = 115 , likelihood = -0.4277983 , dispersion = 0.8780007
##
## Run # 8 :
## Nsteps = 114 , likelihood = -0.4276658 , dispersion = 0.8573163
##
## Run # 9 :
## Nsteps = 91 , likelihood = -0.4262824 , dispersion = 0.8628858
##
## Run # 10 :
## Nsteps = 119 , likelihood = -0.425915 , dispersion = 0.8627117
##
## Sample# 1 : Max(likelihood) = -0.4258037 , dispersion = 0.8627117 , cophenetic = 0.9625778
```

After each run, the residual and dispersion are printed, and the global minimum of residual as well as the corresponding matrices W and H are stored. The dispersion ρ is a scalar measure of how close the consistency matrix $\bar{\mathsf{C}} \equiv \text{Mean}(\mathsf{C})$ elements, where C is the connectivity matrix, are to binary values $0, 1$. The mean is over multiple runs:

$$\rho = \frac{4}{n^2} \sum_{jl} \left(\bar{C}_{jl} - 1/2 \right)^2 .$$

Note in the output above that ρ decays from 1 as the number of runs increases and then stabilizes. This degree of convergence of ρ is a good indication for the adequacy of `nrun`. The cophenetic is the correlation between the distance $1 - \bar{\mathsf{C}}$ and the height matrix of hierarchical clustering[6].

To discover clusters of cells, the reduced dimensionality of factorization, or the rank r, must be estimated. The examples above used a single rank value. If the parameter `ranks` is a vector, the set of inferences will be repeated for each rank value.

```
sc <- factorize(sc, ranks = 3:8, nrun = 5, verbose = 1, progress.bar = FALSE)
```

```
## Rank 3
## Sample# 1 : Max(likelihood) = -0.4257138 , dispersion = 0.7601999 , cophenetic = 0.9367336
## Rank 4
```

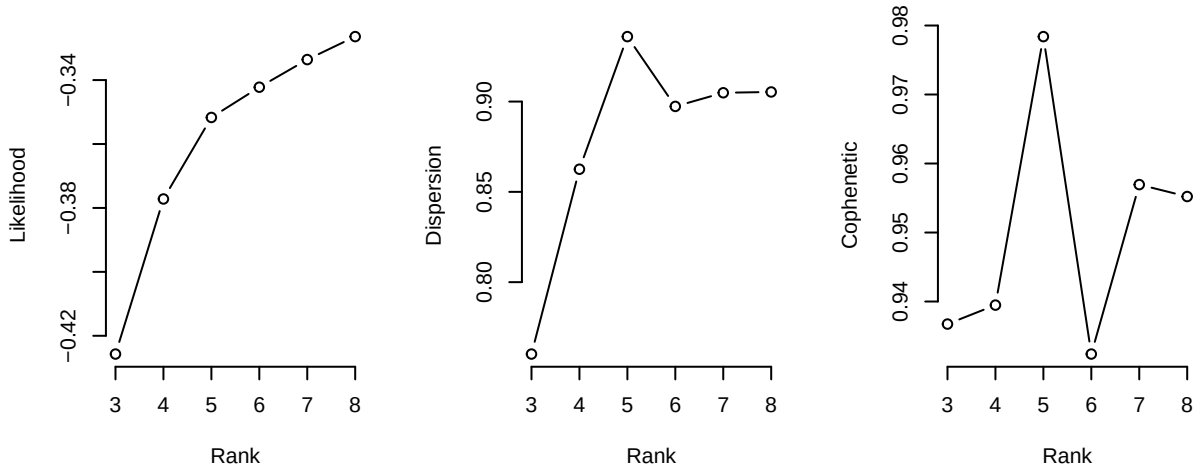

Figure 4: Factorization quality measures as functions of the rank. Residual is (negative) the likelihood function being optimized. Dispersion measures the degree of bimodality in consistency matrix. Cophenetic correlation measures the degree of agreement between consistency matrix and hierarchical clustering.

```
## Sample# 1 : Max(likelihood) = -0.3771986 , dispersion = 0.8625428 , cophenetic = 0.9394812
## Rank 5
## Sample# 1 : Max(likelihood) = -0.3516938 , dispersion = 0.9359733 , cophenetic = 0.9783924
## Rank 6
## Sample# 1 : Max(likelihood) = -0.3422231 , dispersion = 0.8973183 , cophenetic = 0.9323892
## Rank 7
## Sample# 1 : Max(likelihood) = -0.3335847 , dispersion = 0.9048757 , cophenetic = 0.9569438
## Rank 8
## Sample# 1 : Max(likelihood) = -0.3263959 , dispersion = 0.9053135 , cophenetic = 0.955215
```

Note that `nrun` parameter above is set to a small value for illustration. In a real application, typical values of `nrun` would be larger. The progress bar shown by default under `verbose = 1` for overall `nrun` runs is turned off above. It can be set to `TRUE` here (and below) to monitor the progress. After factorization, the `measure` slot has been filled:

```
measure(sc)
```

```
##   rank likelihood dispersion cophenetic
## 1    3 -0.4257138  0.7601999  0.9367336
## 2    4 -0.3771986  0.8625428  0.9394812
## 3    5 -0.3516938  0.9359733  0.9783924
## 4    6 -0.3422231  0.8973183  0.9323892
## 5    7 -0.3335847  0.9048757  0.9569438
## 6    8 -0.3263959  0.9053135  0.9552150
```

These measures can be plotted (Fig. 4):

```
plot(sc)
```

5. Bayesian NMF

The maximum likelihood-based inference must rely on quality measures to choose optimal rank. Bayesian NMF allows for the statistical comparison of different models, namely those with different ranks. The quantity compared is the log probability ("evidence") of data conditional to models (defined by rank and hyperparameters). The main function for Bayesian factorization is `vb_factorize()`:

```
set.seed(1)
sb <- sc_rescue
sb <- vb_factorize(sb, verbose = 3, Tol = 2e-4, hyper.update.n0 = 5)
```

```
## Rank 2
## Run # 1 :
## 1: log(evidence) = -1.768736, aw = 1, bw = 1, ah = 1, bh = 1
## 2: log(evidence) = -1.862852, aw = 1, bw = 1, ah = 1, bh = 1
## 3: log(evidence) = -1.905034, aw = 1, bw = 1, ah = 1, bh = 1
## 4: log(evidence) = -1.927561, aw = 1, bw = 1, ah = 1, bh = 1
## 5: log(evidence) = -1.941132, aw = 1, bw = 1, ah = 1, bh = 1
## 6: log(evidence) = -1.950156, aw = 0.454796, bw = 1.980187, ah = 4.893257, bh = 1.002442
## 7: log(evidence) = -1.949589, aw = 0.5495591, bw = 1.980233, ah = 5.054645, bh = 1.002368
## 8: log(evidence) = -1.953776, aw = 0.544875, bw = 1.98028, ah = 5.020796, bh = 1.002289
## 9: log(evidence) = -1.955178, aw = 0.5361917, bw = 1.980376, ah = 4.85887, bh = 1.002211
## 10: log(evidence) = -1.952889, aw = 0.5238948, bw = 1.980563, ah = 4.596734, bh = 1.002134
## 11: log(evidence) = -1.94713, aw = 0.5082815, bw = 1.980869, ah = 4.277967, bh = 1.00206
## 12: log(evidence) = -1.939088, aw = 0.4902396, bw = 1.981301, ah = 3.945504, bh = 1.001986
## 13: log(evidence) = -1.930012, aw = 0.4711949, bw = 1.981847, ah = 3.628305, bh = 1.001914
## 14: log(evidence) = -1.920718, aw = 0.452933, bw = 1.982488, ah = 3.339913, bh = 1.001842
## 15: log(evidence) = -1.911661, aw = 0.4368788, bw = 1.983204, ah = 3.083905, bh = 1.00177
## 16: log(evidence) = -1.903084, aw = 0.4231766, bw = 1.983981, ah = 2.859185, bh = 1.001698
## 17: log(evidence) = -1.895097, aw = 0.4110275, bw = 1.984808, ah = 2.662934, bh = 1.001623
## 18: log(evidence) = -1.887729, aw = 0.3999381, bw = 1.985673, ah = 2.491811, bh = 1.001547
## 19: log(evidence) = -1.880952, aw = 0.3897963, bw = 1.986567, ah = 2.342357, bh = 1.001468
## 20: log(evidence) = -1.874696, aw = 0.380616, bw = 1.987481, ah = 2.211174, bh = 1.001386
## 21: log(evidence) = -1.868861, aw = 0.3724151, bw = 1.988402, ah = 2.095112, bh = 1.001301
## 22: log(evidence) = -1.86333, aw = 0.3651485, bw = 1.989322, ah = 1.99145, bh = 1.001211
## 23: log(evidence) = -1.857995, aw = 0.3587184, bw = 1.990229, ah = 1.898045, bh = 1.001118
## 24: log(evidence) = -1.85278, aw = 0.3529829, bw = 1.991115, ah = 1.813347, bh = 1.00102
## 25: log(evidence) = -1.847652, aw = 0.3478041, bw = 1.991973, ah = 1.736302, bh = 1.000918
## 26: log(evidence) = -1.842618, aw = 0.3431211, bw = 1.992798, ah = 1.666176, bh = 1.000811
## 27: log(evidence) = -1.837705, aw = 0.3388942, bw = 1.993588, ah = 1.602393, bh = 1.0007
## 28: log(evidence) = -1.832943, aw = 0.3350308, bw = 1.994338, ah = 1.544438, bh = 1.000584
## 29: log(evidence) = -1.828357, aw = 0.3314368, bw = 1.995048, ah = 1.491813, bh = 1.000464
## 30: log(evidence) = -1.823965, aw = 0.3280909, bw = 1.995716, ah = 1.444041, bh = 1.000339
## 31: log(evidence) = -1.819778, aw = 0.3250541, bw = 1.996341, ah = 1.400674, bh = 1.000209
## 32: log(evidence) = -1.815798, aw = 0.3222908, bw = 1.996923, ah = 1.360076, bh = 1.000076
## 33: log(evidence) = -1.812029, aw = 0.3197891, bw = 1.997464, ah = 1.324569, bh = 0.9999379
## 34: log(evidence) = -1.808456, aw = 0.3175515, bw = 1.997962, ah = 1.292267, bh = 0.9997963
## 35: log(evidence) = -1.805083, aw = 0.3157653, bw = 1.99842, ah = 1.262981, bh = 0.9996508
## 36: log(evidence) = -1.801896, aw = 0.3145452, bw = 1.998838, ah = 1.236459, bh = 0.9995017
## 37: log(evidence) = -1.798876, aw = 0.3137732, bw = 1.999219, ah = 1.212485, bh = 0.9993491
## 38: log(evidence) = -1.796006, aw = 0.3132734, bw = 1.999563, ah = 1.190863, bh = 0.999193
## 39: log(evidence) = -1.793275, aw = 0.3129313, bw = 1.999873, ah = 1.171406, bh = 0.9990337
## 40: log(evidence) = -1.790679, aw = 0.3126797, bw = 2.000151, ah = 1.153933, bh = 0.9988714
## 41: log(evidence) = -1.788221, aw = 0.3124775, bw = 2.0004, ah = 1.138253, bh = 0.9987063
## 42: log(evidence) = -1.785914, aw = 0.3123046, bw = 2.000622, ah = 1.124164, bh = 0.9985387
```

```
## 43: log(evidence) = -1.783787, aw = 0.3121653, bw = 2.000822, ah = 1.111443, bh = 0.9983688
## 44: log(evidence) = -1.781878, aw = 0.3120871, bw = 2.001001, ah = 1.09985, bh = 0.9981968
## 45: log(evidence) = -1.780224, aw = 0.3120919, bw = 2.001164, ah = 1.089137, bh = 0.9980228
## 46: log(evidence) = -1.778837, aw = 0.3121684, bw = 2.001314, ah = 1.079078, bh = 0.997847
## 47: log(evidence) = -1.777688, aw = 0.3122895, bw = 2.001452, ah = 1.069499, bh = 0.9976694
## 48: log(evidence) = -1.776703, aw = 0.3124399, bw = 2.00158, ah = 1.0603, bh = 0.9974902
## 49: log(evidence) = -1.77579, aw = 0.3126302, bw = 2.001699, ah = 1.051455, bh = 0.9973094
## 50: log(evidence) = -1.774868, aw = 0.312889, bw = 2.001809, ah = 1.042979, bh = 0.997127
## 51: log(evidence) = -1.773896, aw = 0.3132321, bw = 2.001911, ah = 1.034877, bh = 0.9969433
## 52: log(evidence) = -1.772877, aw = 0.3136463, bw = 2.002005, ah = 1.027095, bh = 0.9967582
## 53: log(evidence) = -1.771857, aw = 0.3141066, bw = 2.002094, ah = 1.019484, bh = 0.9965717
## 54: log(evidence) = -1.770925, aw = 0.3145903, bw = 2.002179, ah = 1.011774, bh = 0.9963839
## 55: log(evidence) = -1.770208, aw = 0.3150789, bw = 2.002264, ah = 1.003568, bh = 0.9961947
## Nsteps =56, log(evidence) =-1.770208, hyper = (0.3155578,2.002351,0.9943697,0.996004), dispersion =
##
## Max(evidence) = -1.770208
```

The iteration maximizes (log) evidence (per matrix elements) and terminates when its fractional change becomes smaller than `Tol`. The option `criterion = connectivity` can also be used. By default, hyperparameters of priors are also updated after `hyper.update.n0` steps. As in maximum likelihood, multiple ranks can be specified:

```
sb <- vb_factorize(sb, ranks = 2:8, verbose = 1, nrun = 5, progress.bar = FALSE)
```

```
## Rank 2
## Max(evidence) = -1.751809
##
## Rank 3
## Max(evidence) = -1.658169
##
## Rank 4
## Max(evidence) = -1.593949
##
## Rank 5
## Max(evidence) = -1.585794
##
## Rank 6
## Max(evidence) = -1.593662
##
## Rank 7
## Max(evidence) = -1.602038
##
## Rank 8
## Max(evidence) = -1.610532
```

With `nrun` larger than 1, multiple inferences will be performed for each rank with different initial conditions and the solution with the highest evidence will be chosen. The object after a `vb_factorize` run will have its `measure` slot filled:

```
head(sb@measure)
```

```
##   rank  evidence        aw        bw         ah        bh
## 1    2 -1.751809 0.3429588 2.0492517 0.43172083 0.9665502
## 2    3 -1.658169 0.3158229 1.2846845 0.14743452 0.9862813
## 3    4 -1.593949 0.2938319 0.9866842 0.09890424 0.9667814
## 4    5 -1.585794 0.2594244 0.7982787 0.13985891 0.9820918
```

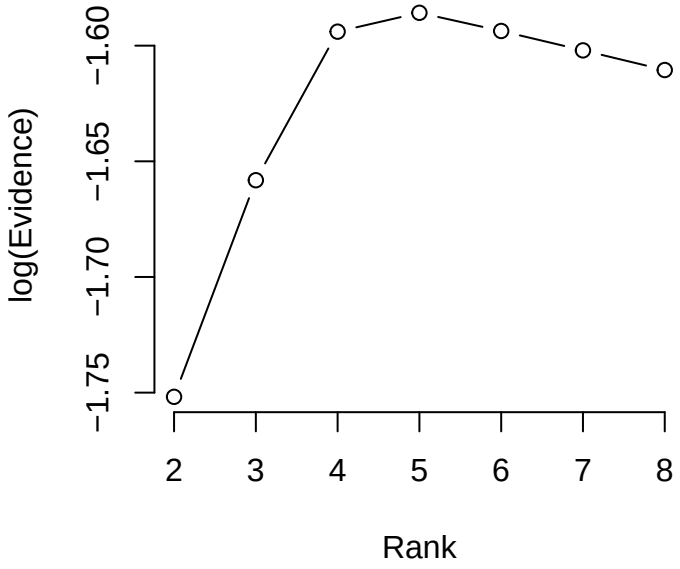

Figure 5: Dependence of log evidence with rank.

```
## 5    6 -1.593662 0.2225920 0.6847888 0.16406499 0.9459589
## 6    7 -1.602038 0.1726058 0.5846701 0.19495146 0.9428658
```

Plotting the object displays the log evidence as a function of rank (Fig. 5):

```
plot(sb)
```

6. Visualization

The rank scan above using Bayesian inference correctly identifies $r = 5$ as the optimal rank. The fit results for each rank – from either maximum likelihood or Bayesian inference – are stored in `sb@basis` and `sb@coeff`. Both are lists of matrices of length equal to the number of rank values scanned. We can access them by, e.g.,

```
sb@ranks
```

```
## [1] 2 3 4 5 6 7 8
```

```
head(sb@basis[[which(sb@ranks==5)]]) # basis matrix W for rank 5
```

```
##                   1          2          3          4           5
## ISG15    0.000711465 0.02671446 0.11843101 0.04393540 0.0906401384
## RPL11    5.025406172 2.31647674 2.58589074 0.81922001 1.0597916473
## SH3BGRL3 0.195047430 0.20599563 1.14866241 0.42579310 0.7326395263
## CD52     0.535672414 0.67590013 0.83460904 0.09443982 0.0364335830
## KHDRBS1  0.034834464 0.02313756 0.06659954 0.01330109 0.0007052528
## YBX1     0.123354063 0.17697456 0.62798282 0.16286968 0.0460645237
```

Heatmaps of W and H matrices are displayed by `gene_map()` and `cell_map()`, respectively (Figs. 6-7):

```
gene_map(sb, markers = markers, rank = 5, max.per.cluster = 4, gene.name=sb@genes[,2],
         cexRow = 0.7)
```

In addition to the marker gene list provided as a parameter, the representative groups of genes for clusters are selected by the "max" scheme[7]: genes are sorted for each cluster with decreasing magnitudes of coefficient matrix elements, and first top members of the list for which the magnitude is the actual maximum over all clusters are chosen. Based on the marker-metagene map in Fig. 8, we rename the clusters as follows:

```
cell_type <- c('CD8+_T','B_cells','CD4+_T','NK','Monocytes')
colnames(sb@basis[[which(sb@ranks == 5)]]) <- cell_type
rownames(sb@coeff[[which(sb@ranks == 5)]]) <- cell_type
```

```
cell_map(sb, rank = 5)
```

In `visualize_clusters()`, each column of H matrix is used to assign cells into clusters, and inter/intra-cluster separations are visualized using tSNE algorithm[8]. It uses the `Rtsne()` function of the `Rtsne` package. A barplot of cluster cell counts are also displayed (Fig. 8):

```
visualize_clusters(sb, rank = 5, cex = 0.7)
```

```
## Read the 410 x 5 data matrix successfully!
## Using no_dims = 2, perplexity = 30.000000, and theta = 0.500000
## Computing input similarities...
## Normalizing input...
## Building tree...
##  - point 0 of 410
## Done in 0.06 seconds (sparsity = 0.274277)!
## Learning embedding...
## Iteration 50: error is 49.680854 (50 iterations in 0.10 seconds)
## Iteration 100: error is 46.131078 (50 iterations in 0.09 seconds)
## Iteration 150: error is 45.669665 (50 iterations in 0.09 seconds)
## Iteration 200: error is 45.389212 (50 iterations in 0.09 seconds)
## Iteration 250: error is 45.309030 (50 iterations in 0.09 seconds)
## Iteration 300: error is 0.276564 (50 iterations in 0.09 seconds)
## Iteration 350: error is 0.219905 (50 iterations in 0.09 seconds)
## Iteration 400: error is 0.206563 (50 iterations in 0.09 seconds)
## Iteration 450: error is 0.196946 (50 iterations in 0.09 seconds)
## Iteration 500: error is 0.191316 (50 iterations in 0.09 seconds)
## Iteration 550: error is 0.184085 (50 iterations in 0.10 seconds)
## Iteration 600: error is 0.182262 (50 iterations in 0.10 seconds)
## Iteration 650: error is 0.179678 (50 iterations in 0.10 seconds)
## Iteration 700: error is 0.177394 (50 iterations in 0.10 seconds)
## Iteration 750: error is 0.175468 (50 iterations in 0.09 seconds)
## Iteration 800: error is 0.174292 (50 iterations in 0.10 seconds)
## Iteration 850: error is 0.173138 (50 iterations in 0.10 seconds)
## Iteration 900: error is 0.172916 (50 iterations in 0.10 seconds)
## Iteration 950: error is 0.172497 (50 iterations in 0.09 seconds)
## Iteration 1000: error is 0.171518 (50 iterations in 0.09 seconds)
## Fitting performed in 1.91 seconds.
```

It is useful to extract hierarchical relationships among the clusters identified. This feature requires a series of inference outcomes for an uninterrupted range of rank values, e.g., from 2 to 7:

```
tree <- build_tree(sb, rmax = 5)
tree <- rename_tips(tree, rank = 5, tip.labels = cell_type)
plot_tree(tree, cex = 0.8, show.node.label = TRUE)
```

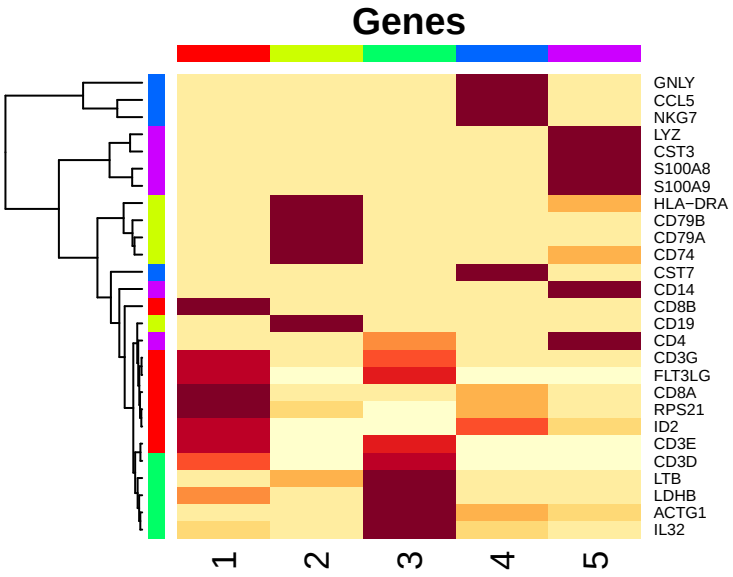

Figure 6: Heatmap of basis matrix elements. Marker genes selected in rows, other than those provided as input, are based on the degree to which each features strongly in a particular cluster only and not in the rest. Columns represent the clusters.

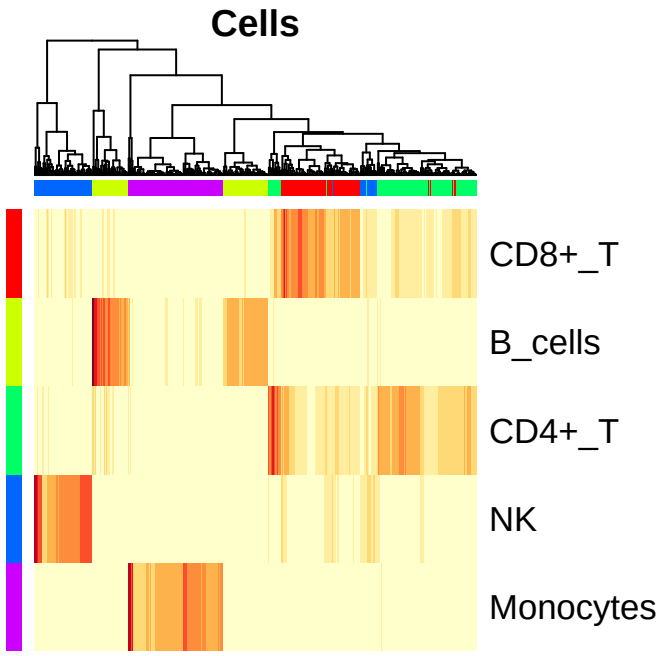

Figure 7: Heatmap of cluster coefficient matrix elements. Rows indicate clusters and columns the cells.

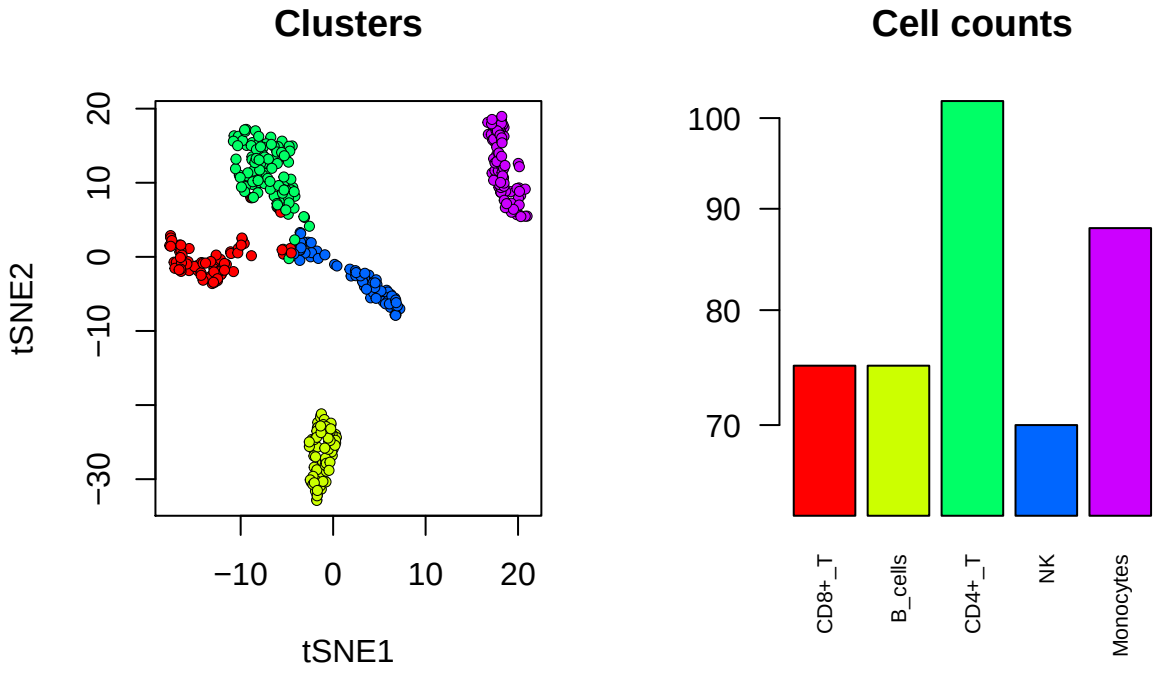

Figure 8: tSNE-based visualization of coefficient matrix elements of cells with colors indicating predicted cluster assignment. The bar plot shows the cell counts of each cluster.

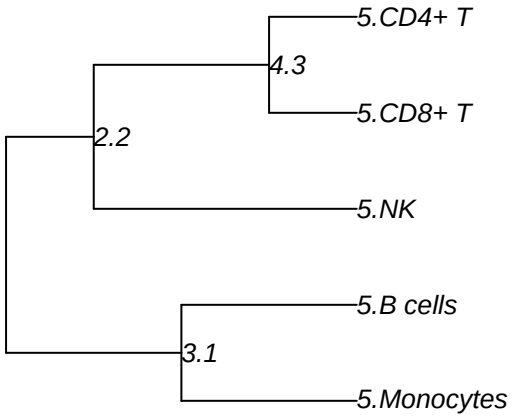

Figure 9: Hierarchical tree of clusters derived from varying ranks. The rank increases from 2 to 5 horizontally and nodes are labeled by cluster IDs which bifurcated in each rank.

The `build_tree` function returns a list containing the tree. The second command above renames the label of terminal nodes by our cell type label. In Fig. 9, the relative distance between clusters can be seed to be consistent with the tSNE plot in Fig. 8.

References

1. Lee, D. D. & Seung, H. S. Learning the parts of objects by non-negative matrix factorization. *Nature* **401,** 788–791 (1999).

2. Hastie, T., Tibshirani, R. & Friedman, J. *The elements of statistical learning: data mining, inference, and prediction.* (2009).

3. Cemgil, A. T. Bayesian inference for nonnegative matrix factorisation models. *Comput. Intell. Neurosci.* 785152 (2009).

4. Gaujoux, R. & Seoighe, C. A flexible R package for nonnegative matrix factorization. *BMC Bioinformatics* **11,** 367 (2010).

5. Zhu, X., Ching, T., Pan, X., Weissman, S. M. & Garmire, L. Detecting heterogeneity in single-cell RNA-Seq data by non-negative matrix factorization. *PeerJ* **5,** e2888 (2017).

6. Brunet, J. P., Tamayo, P., Golub, T. R. & Mesirov, J. P. Metagenes and molecular pattern discovery using matrix factorization. *Proc. Natl. Acad. Sci. U.S.A.* **101,** 4164–4169 (2004).

7. Carmona-Saez, P., Pascual-Marqui, R. D., Tirado, F., Carazo, J. M. & Pascual-Montano, A. Biclustering

of gene expression data by non-smooth non-negative matrix factorization. *BMC Bioinformatics* **7,** 78 (2006).

8. Maaten, L. van der & Hinton, G. Visualizing high-dimensional data using t-SNE. *J. Mach. Learn. Res.* **9,** 2579–2605 (2008).