Analyzing multiple single-cell gene expression samples from biologically heterogeneous sources using scINSIGHT

Kun Qian, Wei Vivian Li

2022-02-04

This vignette demonstrates how to use the scINSIGHT package to analyze multiple single-cell gene expression matrices and how to obtain the factorized results from its returned objects. All datasets and obtained objects can be downloaded from here.

Load data and create an scINSIGHT object

The major input required by the scINSIGHT algorithm is a list of multiple (at least two) gene expression matrices. Genes should be in rows and cells in columns for each matrix. The row names represent gene names, and should match and be in the same order in all matrices.

We consider two scenarios. (1) If users have used their preferred methods to perform normalization and gene feature selection, scINSIGHT can directly take a list of normalized and filtered matrices as the input (we also recommend performing log2-transformation after normalization); (2) If users have raw count matrices, we recommend using the Seurat package to perform normalization and gene feature selection before running scINSIGHT. Scenario (1) is demonstrated with a simulated dataset, and Scenario (2) is demonstrated with a real dataset.

Create an scINSIGHT object with processed data

We first introduce how to create an scINSIGHT object with processed data, using a list of simulated gene expression matrices as an example. The simulation process is described in our manuscript. Suppose we have the normalized, filtered, and log2-transformed gene expression matrices saved as a list.

```
library(scINSIGHT)
# Load the simulated data
sim.counts = readRDS("sim_count.rds")
```

`sim.counts` is a list of six gene expression samples with rows representing 4977 genes.

To create the scINSIGHT object for running the algorithm, we need to provide the biological condition each sample belongs to through the `condition` parameter in `create_scINSIGHT` function. In this simulation example, the six samples are from three time points (T1, T2, and T3). Therefore, we can initialize the scINSIGHT object using the code below:

```
sim.sample = c("S1", "S2", "S3", "S4", "S5", "S6")
# Name of count list should represent sample names
names(sim.counts) = sim.sample
sim.condition = c("T1", "T1", "T2", "T2", "T3", "T3")
# Name of condition vector should match sample names in count list
names(sim.condition) = sim.sample

# Create an scINSIGHT object
sim.scobj = create_scINSIGHT(norm.data = sim.counts, condition = sim.condition)
```

Create an scINSIGHT object with raw data

Now, we introduce how to initialize an scINSIGHT object with raw count data, using a list of two count matrices as an example. If users do not have a preferred method, we recommend using Seurat to perform normalization and gene feature selection. Please see the example code below:

```
library(Seurat)
```

```
## Attaching SeuratObject
```

```
# Load a list of raw count matrices
data.list = readRDS("real_count.rds")

data.list = lapply(1:length(data.list), function(i){
  # Initialize the Seurat object with the raw data
  x = CreateSeuratObject(counts = data.list[[i]], assay = "RNA")
  # Normalize the data
  x = NormalizeData(x)
  # Identify 2000 highly variable genes
  x = FindVariableFeatures(x, nfeatures = 2000)
  return(x)
})

# Select the 2000 gene features for all samples
features = SelectIntegrationFeatures(object.list = data.list, nfeatures = 2000)

# Obtain a list of processed matrices for scINSIGHT
real.counts = lapply(data.list, function(x){
  as.matrix(x@assays$RNA@data[features, ])
})
```

`real.counts` is a list of two gene expression samples with rows representing 2000 genes.

In this example, the two samples are from the control or activation conditions, respectively. Therefore, we can initialize the scINSIGHT object as follows:

```
real.sample = c("sample1", "sample2")
# Name of count list should represent sample names
names(real.counts) = real.sample
# Name of condition vector should match sample names in count list
real.condition = c("sample1" = "control", "sample2" = "activation")

# Create a scINSIGHT object for real data
real.scobj = create_scINSIGHT(norm.data = real.counts, condition = real.condition)
```

Run the scINSIGHT algorithm

After creating scINSIGHT object, we can use the `run_scINSIGHT` function to run the scINSIGHT algorithm, the key parameters include the following:

- *K*. An integer or vector specifying the candidate number(s) of common gene modules. The default value is `c(5, 7, 9, 11, 13, 15)`. If input a vector, the function will select the optimal value as described in the manuscript.The selected K will be stored in `parameters[["K"]]`.
- *K_j*. Integer value specifying the number of condition-specific gene modules. We recommend a relative small number (defaults to 2) to facilitate the interpretation of factorization results.
- *out.dir*. A character specifying the full path to a directory where intermediate results will be saved.
- *num.cores*. An integer specifying the number of cores used for parallel computation.

- *B*. An integer specifying the number of initializations for running scINSIGHT with each `K` value. The default value is 5.

For simulated dataset:

```
sim.scobj = run_scINSIGHT(sim.scobj, K = seq(5,15,2), out.dir = './test_save_sim/',
                          num.cores = 5)
```

For real dataset:

```
real.scobj = run_scINSIGHT(real.scobj, K = seq(5,15,2), out.dir = './test_save_real/',
                           num.cores = 5)
```

In addition, users can tune the following parameters given the number and size of gene expression matrices:

- *thre.niter*. Maximum number of iterations in scINSIGHT algorithm. The default value is 500.
- *thre.delta*. Convergence threshold. The iteration will stop when the decrease of objective function between adjacent iterations is less than the threshold. The default value is 0.01.

Interpret scINSIGHT results

After running the algorithm, the results are stored in the scINSIGHT object. We explain the results using the object obtained for the simulated data.

The selected parameters and scores are saved in the `parameters` slot. In this example, the selected `K` (number of common gene modules) is 13 and the selected `lda` (regularization parameters) is 0.01.

```
sim.scobj = readRDS("sim_scobj.rds")
print(sim.scobj@parameters)
```

```
## $K_j
## [1] 2
##
## $lda
## [1] 0.01
##
## $stability
##       K_5       K_7       K_9      K_11      K_13      K_15
## 0.9719409 0.9909814 0.9945204 0.9951915 0.9949469 0.9956020
##
## $K
## [1] 13
##
## $specificity
## lda_0.001  lda_0.01   lda_0.1     lda_1    lda_10
## 1.3273222 1.5671731 1.0059919 0.9871796 0.9892358
```

Also, we can check the length or dimension of slots `V` and `H`.

`V` is the membership matrix of `K` common gene modules, and it's shared by all samples.

`H` is a list containing the membership matrix of `K_j` condition-specific gene modules of each condition.

```
# V is a matrix whose dimension is K by gene number
print(dim(sim.scobj@V))
# H is a list whose length equals the condition number
print(length(sim.scobj@H))
```

```
## [1]   13 4977
## [1] 3
```

The `norm.W_2` slot, a list containing the normalized expression levels of common gene modules in each sample, could be used for visualization and downstream analysis. For example,

```
# Sample name of each single cell
sample = unlist(lapply(1:length(sim.counts), function(i){
  rep(sim.sample[i], ncol(sim.counts[[i]]))
}))
# Condition of each single cell
condition = unlist(lapply(1:length(sim.counts), function(i){
  rep(sim.condition[i], ncol(sim.counts[[i]]))
}))
# True cell type of each single cell
celltype = unlist(lapply(1:length(sim.counts), function(i){
  lapply(1:ncol(sim.counts[[i]]), function(j){
    toupper(unlist(strsplit(colnames(sim.counts[[i]])[[j]], "[.]"))[[1]])})}))

library(Rtsne)
# W2 is the normalized expression levels of common gene modules
W2 = Reduce(rbind, sim.scobj@norm.W_2)
set.seed(1)
res_tsne = Rtsne(W2, dims = 2, check_duplicates = FALSE)

da_tsne = data.frame(tSNE1 = res_tsne$Y[,1], tSNE2 = res_tsne$Y[,2],
                     sample = sample, condition = condition,
                     celltype = celltype)
```

Then, we can obtain the tSNE plots colored by sample, condition or cell type.

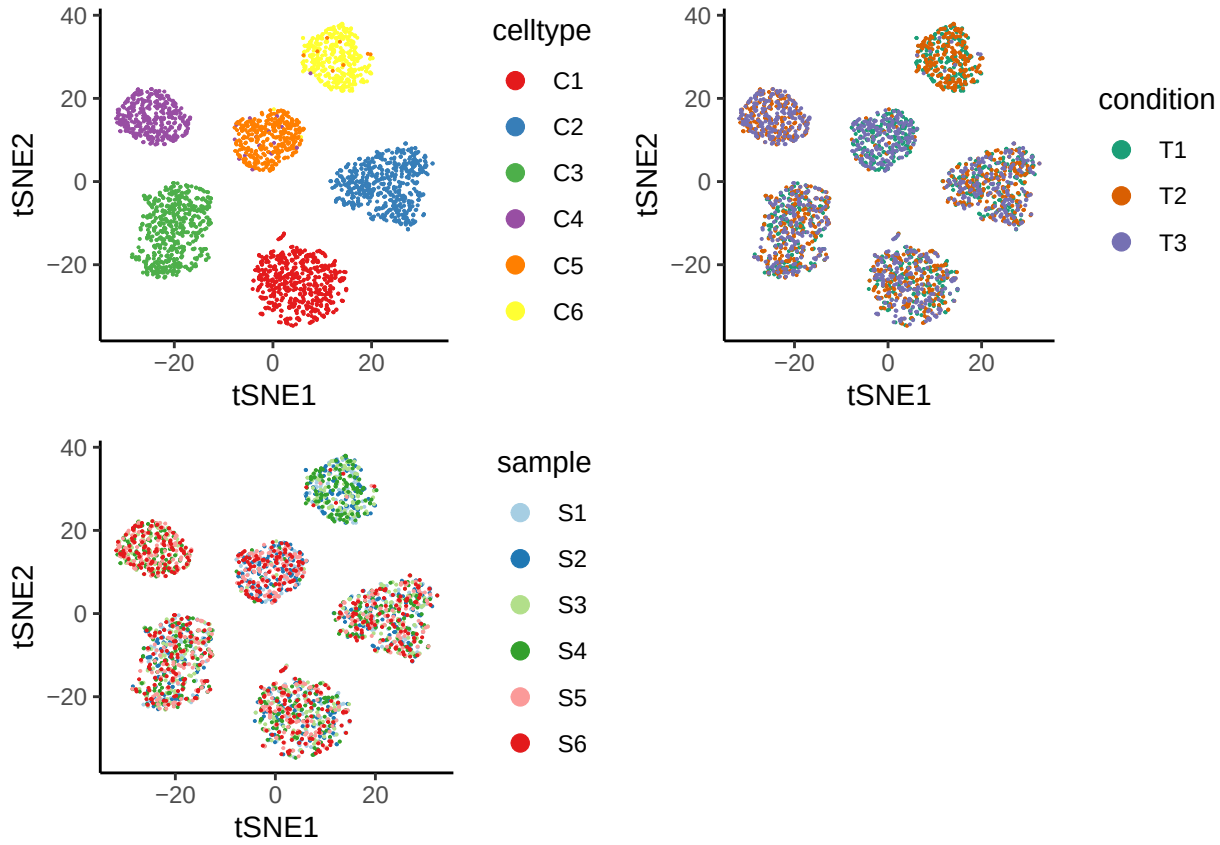
```
library(ggplot2); theme_set(theme_classic())
library(RColorBrewer)

p1 <- ggplot(da_tsne, aes(x = tSNE1, y = tSNE2, color = celltype)) +
    geom_point(cex = .3, stroke = .3) + scale_color_brewer(palette="Set1") +
    guides(colour = guide_legend(override.aes = list(size=3)))

p2 <- ggplot(da_tsne, aes(x = tSNE1, y = tSNE2, color = condition)) +
    geom_point(cex = .3, stroke = .3) + scale_color_brewer(palette="Dark2") +
    guides(colour = guide_legend(override.aes = list(size=3)))

p3 <- ggplot(da_tsne, aes(x = tSNE1, y = tSNE2, color = sample)) +
    geom_point(cex = .3, stroke = .3) + scale_color_brewer(palette="Paired") +
    guides(colour = guide_legend(override.aes = list(size=3)))

cowplot::plot_grid(p1, p2, p3, nrow = 2, ncol=2)
```



For real data, the selected the selected `K` (number of common gene modules) is 9 and the selected `lda` (regularization parameters) is 0.1.

```
real.scobj = readRDS("real_scobj.rds")
print(real.scobj@parameters)
```

```
## $K_j
## [1] 2
##
## $lda
## [1] 0.1
##
## $stability
##       K_5       K_7       K_9      K_11      K_13      K_15
## 0.9432459 0.9750923 0.9809241 0.9824284 0.9584675 0.9474269
##
## $K
## [1] 9
##
## $specificity
## lda_0.001  lda_0.01   lda_0.1     lda_1    lda_10
##  1.058958  1.056734  1.136457  1.116313  1.133409
```

We can also obtain the tSNE plots colored by sample or condition.

```
# Sample name of each single cell

sample = unlist(lapply(1:length(real.counts), function(i){
```

```
  rep(real.sample[i], ncol(real.counts[[i]]))
}))
# Condition of each single cell
condition = unlist(lapply(1:length(real.counts), function(i){
  rep(real.condition[i], ncol(real.counts[[i]]))
}))

# W2 is the normalized expression levels of common gene modules
W2 = Reduce(rbind, real.scobj@norm.W_2)
set.seed(1)
res_tsne = Rtsne(W2, dims = 2, check_duplicates = FALSE)
da_tsne = data.frame(tSNE1 = res_tsne$Y[,1], tSNE2 = res_tsne$Y[,2],
                     sample = sample, condition = condition)


p1 <- ggplot(da_tsne, aes(x = tSNE1, y = tSNE2, color = condition)) +
  geom_point(cex = .3, stroke = .3) + scale_color_brewer(palette="Dark2") +
  guides(colour = guide_legend(override.aes = list(size=3)))

p2 <- ggplot(da_tsne, aes(x = tSNE1, y = tSNE2, color = sample)) +
  geom_point(cex = .3, stroke = .3) + scale_color_brewer(palette="Set1") +
  guides(colour = guide_legend(override.aes = list(size=3)))

cowplot::plot_grid(p1, p2, nrow = 2, ncol=2)
```