

CAF Single cell Transcriptomics Analysis

Mickael Coquerelle & Loik Galtier

2025-12-01

Contents

1	Résumé	2
2	Préparation des données.	2
2.1	QC et filtrage des données	3
3	Normalisation avec SC Transform.	4
3.1	Le modèle mathématique expliqué.	4
3.2	Application de la méthode	5
4	Réduction de dimension, clustering et visualisation	5

```
# Setup global chunk options and libraries.
knitr::opts_chunk$set(echo = TRUE, warning = FALSE, message = FALSE,
                      fig.width = 8, fig.height = 6)

# Tidyverse-like utilities
library(dplyr)
library(ggplot2)

# Seurat and single-cell specific tools
library(Seurat)

# Parallelization for Seurat (kept sequential by default in this report for reproducibility)
library(future)

# Annotation via Ensembl
library(AnnotationDbi)
library(EnsDb.Hsapiens.v86)
sessionInfo()

# Parallelisation:
plan(sequential)
options(future.globals.maxSize=10*1024^3) # 10 GB

" data loading : we load the three datasets corresponding to three different patients' CAF single cell transcriptomics data.

p5 <- read.csv("~/Projets_GIT/Single_Cell_Project/Data/GSM4805570_CountsMatrix_20G00953M_TN.txt")
p4a <- read.csv("~/Projets_GIT/Single_Cell_Project/Data/GSM4805568_CountsMatrix_19G02977B_TN.txt")
```

```
p4b <- read.csv("~/Projets_GIT/Single_Cell_Project/Data/GSM4805566_CountsMatrix_19G02977A_TN.t
```

1 Résumé

L'objet de ce travail est de se familiariser avec l'analyse transcriptomique Single-Cell de CAF (Cancer Associated Fibroblasts) à partir d'une matrice de comptages issues des trois fichiers proposé. Notre pipeline d'analyse en se basant sur les éléments donnés en cours couvre le nettoyage (filtrage cellules/gènes), normalisation, réduction de dimension (PCA, t-SNE, UMAP), clustering, identification de(s) signature(s) transcriptomique(s) pour marquer les clusters et enfin nous proposons une visualisation ciblée par heatmap et features plots.

2 Préparation des données.

We combine the three datasets and map Ensembl IDs to gene symbols and types. the objectif is to filter for protein-coding genes with unique symbols and no missing values. ## Pré-processing des données

```
caf.data <- data.matrix(cbind(p5,p4a,p4b))
ens <- read.csv("~/Projets_GIT/Single_Cell_Project/Data/ensembl-38-108-genes.txt",
               sep="\t")
# map Ensembl IDs to genes symbols and types :
ens2symb <- setNames(ens$Gene.name, ens$Gene.stable.ID)
ens2type <- setNames(ens$Gene.type, ens$Gene.stable.ID)
symbols <- ens2symb[rownames(caf.data)] # map ensembl IDs to gene symbols

# filter for protein-coding genes with unique symbols and no missing values :
types <- ens2type[rownames(caf.data)]
# condition filter is defined here
good <- types=="protein_coding" & !is.na(symbols) & !duplicated(symbols)
sum(good) # number of genes passing the filter
symb <- symbols[good]
caf.data <- caf.data[good,]
rownames(caf.data) <- symb
```

On crée un premier objet SeuratObject pour tirer profit des routines intégrées à la librairie (FindVariableFeatures, ScaleData, RunPCA, etc.):

```
caf <- CreateSeuratObject(counts=caf.data, project="cafs",
                        min.cells=0.01*ncol(caf.data), min.features=1000)
#caf
```

2.1 QC et filtrage des données

Puis on va définir des filtres. La détermination de ces derniers se fait sur des règles d'usage en *single-cell*. $min.cells = 0.01$ au premier passage pour exclure les gènes exprimés dans très peu de cellules (ici adapté au `CreateSeuratObject`). On applique ensuite des QC aux cellules : - Les UMIs élevés (> 50000) indiquent des duplicats il faut donc les gérer. - Les cellules avec peu de gènes détectés (< 1000) sont souvent de mauvaise qualité, il y'a donc nécessité de les filtrer. - Ensuite on ajoute une condition pour gérer la proportion de gènes mitochondriaux élevée (> 50), qui est un indicateur de cellules mortes. En combinant ces filtres, on nettoie notre dataset de manière avoir un jeu de cellule de haute qualité.

```
ensdb.genes <- genes(EnsDb.Hsapiens.v86) # extract from ensdb
# mito gene from MT chromosome
MT.names <- ensdb.genes[seqnames(ensdb.genes) == "MT"]$gene_name
counts <- GetAssayData(caf, "RNA") # get the count matrix

# data for cleaning data next
umi.tot <- colSums(counts)
gene.tot <- colSums(counts > 0)
sum(rownames(counts) %in% MT.names)

# percent of mito :
mito.pc <- colSums(counts[rownames(counts) %in% MT.names, ]) / umi.tot * 100
bad.high <- umi.tot > 50000 ; bad.low <- gene.tot < 1000 ; bad.mito <- mito.pc > 50

# Disjonction des filtres :
bad <- bad.high | bad.low | bad.mito
' # to check the quantity of removed element
table(bad.high); table(bad.low); table(bad.mito); table(bad)
'

# Nettoyage effectif :
counts <- counts[, !bad] ; counts
good.genes <- rowSums(counts > 1) >= 0.01 * ncol(counts)
counts <- counts[good.genes, ]
dim(counts) # Pour vérifier les dimensions après nettoyage
```

Nous pouvons visualiser la distribution de la proportion de gènes mitochondriaux, avant nettoyage:

```
hist(mito.pc, breaks=50)
```

Nous mettons à jour notre objet Seurat avec les données nettoyées :

```
caf <- CreateSeuratObject(counts = counts,
                          project = "CAF_clean",
```

```

min.cells = 0,
min.features = 0)
caf

```

3 Normalisation avec SC Transform.

La normalisation étant une étape charnière et pouvant grandement changer les résultats finaux en Single-Cell. Nous avons choisi ici d'exploiter les fonctionnalités de normalisation de la librairie Seurat. Plutôt qu'une normalisation CPM (par millions) ou LogNormalize, nous appliquons ici **SCTransform**, une méthode plus moderne et robuste, qui est construite sur un modèle négatif binomial (Hafemeister & Satija, 2019). De ce que nous en avons compris, une telle approche va agir sur plusieurs aspects :

- La stabilisation de variance,
- La sélection des gènes variables,
- La correction des effets techniques (ici regression des % mitochondriaux). Cette méthode est d'après la littérature **la référence Seurat** pour le pré-traitement.

3.1 Le modèle mathématique expliqué.

La normalisation SCTransform repose sur un modèle de **régression négative binomiale**.

Pour chaque gène g et chaque cellule i , on suppose :

$$y_{ig} \sim \text{NB}(\mu_{ig}, \theta_g)$$

où nous avons :

- y_{ig} : nombre de transcripts observés (comptes brutes)
- μ_{ig} : moyenne attendue de nos comptes
- θ_g : le paramètre de dispersion du gène g

La moyenne μ_{ig} est modélisée via un lien log-linéaire.

$$\log(\mu_{ig}) = \beta_{0g} + \beta_{1g} \cdot \log(\text{UMI}_i) + \sum_k \gamma_{kg} x_{ik}$$

avec :

- β_{0g} : intercept spécifique au gène g
- UMI_i : total de counts de la cellule i (size factor)

- x_{ik} : variables techniques à régresser (ex. % mitochondriaux, batch)
- γ_{kg} : coefficients associés aux covariables

La normalisation finale utilisée pour l'analyse (PCA, clustering) est obtenue via les **résidus Pearson standardisés** :

$$r_{ig} = \frac{y_{ig} - \hat{\mu}_{ig}}{\sqrt{\hat{\mu}_{ig} + \frac{\hat{\mu}_{ig}^2}{\theta_g}}}$$

Ces résidus sont **variance-stabilisés** et permettent d'atténuer l'effet de la profondeur de séquençage et du bruit technique.

3.2 Application de la méthode

```
# Calcul du pourcentage mitochondrial pour régression
caf[["percent.mt"]] <- PercentageFeatureSet(
  caf, pattern = "^MT-"
)

# Normalisation moderne SCTransform
caf <- SCTransform(
  caf, # applique SCTransform
  vars.to.regress = "percent.mt", # régresse l'effet des gènes mitochondriaux
  verbose = TRUE # affiche la progression
)
```

4 Réduction de dimension, clustering et visualisation

Nous procédons à la réduction de dimension via PCA, puis UMAP pour la visualisation. Nous effectuons ensuite le clustering des cellules et identifions les marqueurs de chaque cluster.

```
caf <- RunPCA(caf, verbose = FALSE)
DimPlot(caf, reduction = "pca")
```

