Chromosome5_analysis

Introduction

This is an R markdown file for running analysis scripts to cluster data from Sawh et al 2020, specially the wild type analysis of the structure of chromosome 5.

We first need to load the required libraries using calls to library(). If this is the first time running this script, you will need to install each package using install_packages().

```r
library(Seurat)
library(Matrix)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
library(R.matlab)
```

```
## R.matlab v3.6.2 (2018-09-26) successfully loaded. See ?R.matlab for help.

##
## Attaching package: 'R.matlab'

## The following objects are masked from 'package:base':
##
##     getOption, isOpen
```

```r
library(stringr)
library(ggplot2)
library(patchwork)
```

Then we set the current working directory, where the raw data and associated files are located:

```r
setwd("/Volumes/BZ/Home/gizevo30/R_Projects/clustering_traces") # Change this to your personal path
```

Load in the PairNames and datasets that come from the Matlab scripts from Ahilya Sawh. We will use just the wild type chromosome 5 data.

The dataset file needs to be modified so that Seurat will accept it.

```
PairNames <- read.csv("raw.data/PairNamesV2.txt", header = FALSE)
PairNames <- PairNames$V1

AllStructures <- readMat("raw.data/wtAllStructures2to40.mat")[[1]]
AllStructures[1:5,1:5]
```

```
##           [,1] [,2] [,3] [,4]     [,5]
## [1,] 0.0000000    0    0    0 0.000000
## [2,] 0.0000000    0    0    0 0.000000
## [3,] 0.0000000    0    0    0 0.000000
## [4,] 0.0000000    0    0    0 0.000000
## [5,] 0.1522289    0    0    0 1.165012
```

The matlab file doesn't have rownames or colnames, which should correspond to the PairNames and trace IDs, respectively.

We can easily add these by making the rownames equal to the PairNames and by making column names with 'paste'. This is now our input for Seurat (trace by pairname matrix).

```
row.names(AllStructures) <- PairNames
colnames(AllStructures) <- paste("wt_Chromosome", 1:ncol(AllStructures), sep = "_") # labels the column

AllStructures[1:5,1:5]
```

```
##      wt_Chromosome_1 wt_Chromosome_2 wt_Chromosome_3 wt_Chromosome_4
## 1to2       0.0000000               0               0               0
## 1to3       0.0000000               0               0               0
## 2to3       0.0000000               0               0               0
## 1to4       0.0000000               0               0               0
## 2to4       0.1522289               0               0               0
##      wt_Chromosome_5
## 1to2        0.000000
## 1to3        0.000000
## 2to3        0.000000
## 1to4        0.000000
## 2to4        1.165012
```

Though single-cell RNA-seq and clustering can deal with the sparse nature of the data (lots of zeros where transcripts aren't detected), we don't think this makes sense for tracing data. First of all, non-detection ~= to no expression makes sense for sequencing data, but isn't logical for tracing. The default value should be the average distance instead. Therefore, we need to replace all 0 values in the matrix with the 'non-zero average'.

```
non.z.avg <- apply(AllStructures, 1, function(x) mean(x[x>0])) # this generates the non-zero average fo
head(non.z.avg)
```

```
##      1to2      1to3      2to3      1to4      2to4      3to4
## 0.9221019 1.0788077 0.9478193 1.0741884 0.9118615 0.8344760
```

```r
non.z.avg[is.nan(non.z.avg)] <- 0 # in case NANs are introduced by the above

# for loop to replace zero values with the non-zero average for each row
for (j in 1:length(non.z.avg)) {
     AllStructures[j, AllStructures[j,] == 0] <- non.z.avg[j]
}

AllStructures[1:5,1:5]
```

```
##      wt_Chromosome_1 wt_Chromosome_2 wt_Chromosome_3 wt_Chromosome_4
## 1to2       0.9221019       0.9221019       0.9221019       0.9221019
## 1to3       1.0788077       1.0788077       1.0788077       1.0788077
## 2to3       0.9478193       0.9478193       0.9478193       0.9478193
## 1to4       1.0741884       1.0741884       1.0741884       1.0741884
## 2to4       0.1522289       0.9118615       0.9118615       0.9118615
##      wt_Chromosome_5
## 1to2       0.9221019
## 1to3       1.0788077
## 2to3       0.9478193
## 1to4       1.0741884
## 2to4       1.1650125
```

Now the matrix is ready for Seurat.

Creating the Seurat object

The first step is to make a Seurat object, which allows us to use all of the built-in functions for clustering and plotting. In the future it might be beneficial to write functions for doing all of the Seurat steps without Seurat. The 'min.genes' variable is the minimum number of datapoints (pairwise distances) for a trace to be included in the analysis. Too few pairs means that trace doesn't have enough information to be reliably clustered. Removes poor quality traces.

```r
chromo.wt <- CreateSeuratObject(AllStructures, min.genes = 6, project = "Sawh_chromosomes", add.cell.id
```

Next we can add meta data, such as the embryonic age of the trace, experimental batch, etc. The chromosome ages file from Matlab can be read in the same way as the Pairnames and AllStructures data. The 'names' of the vector will also have to be renamed so they match the trace names in the Seurat object.

```r
# Embryo age information
ChrAges.wt <- readMat("raw.data/wtChrAges2to40.mat")
ChrAges.wt <- unlist(ChrAges.wt)
names(ChrAges.wt) <- paste("wt_Chromosome", 1:length(ChrAges.wt), sep = "_")

# Add Batch information
batch <- read.csv("raw.data/ChrExpt.txt", head = F)
colnames(batch) <- c("batch")
batch$batch <- str_sub(batch$batch, start = 2, end = 7)
rownames(batch) <- paste("wt_Chromosome_", rownames(batch), sep = "")
```

You can directly access the meta data of the object through the 'meta.data' slot, which is a data.frame that can be modified easily to add or remove information.

We use the ‘WhichCells’ function from Seurat to identify the traces that were used to create the Seurat object - remember, we may have removed some traces due to too few measured Pairs (min.genes variable)

```
head(chromo.wt@meta.data)
```

```
##                 orig.ident nCount_RNA nFeature_RNA
## wt_Chromosome_1         wt   331.7081          231
## wt_Chromosome_2         wt   325.9114          231
## wt_Chromosome_3         wt   356.4475          231
## wt_Chromosome_4         wt   358.2306          231
## wt_Chromosome_5         wt   341.8351          231
## wt_Chromosome_6         wt   357.5248          231
```

```
chromo.wt@meta.data$chromosome_age <- ChrAges.wt[WhichCells(chromo.wt)]

chromo.wt@meta.data$batch <- batch[WhichCells(chromo.wt), 1]
```

Next we normalize, find Variable Features and scale the distance values before running PCA for clustering. Seurat can find features that display a lot of variability, normally to reduce the number of ‘genes’ or ‘features’ used in PCA and clustering (since many features are not informative when measuring thousands of features). However, for traces we can also just use all pairwise distances. Either way, for Seurat to work, we have to ‘FindVariableFeatures’.

```
chromo.wt <- NormalizeData(object = chromo.wt, normalization.method = "LogNormalize", scale.factor = 10
chromo.wt <- FindVariableFeatures(chromo.wt)
chromo.wt <- ScaleData(object = chromo.wt, genes.use = VariableFeatures, do.par = TRUE, num.cores = 6)
```

```
## Warning: The following arguments are not used: genes.use, do.par, num.cores
```

```
## Centering and scaling data matrix
```

Next we run PCA. This generates however many principal components that you want. The basic idea is to reduce the ~hundreds of PairNames down to fewer dimensions for clustering. For this dataset we used 20, but these should be determined empirically through trial and testing.

For this, we also use all PairNames as variable features, minus those from Probe 11 (which had poor detection in the dataset). Including 11 leads to weird clustering.

```
VariableFeatures <- as.vector(PairNames[!(PairNames %in% PairNames[grep("11", PairNames)])])

chromo.wt <- RunPCA(object = chromo.wt, features = VariableFeatures, do.print = TRUE, pcs.print = 1:5,
```

```
## PC_ 1
## Positive:  4to20, 5to20, 3to20, 4to19, 5to19, 5to18, 4to18, 2to19, 2to20, 4to21
##     5to21, 6to20, 4to16, 8to20, 2to18, 5to16, 1to20, 2to21, 2to16, 1to21
##     3to18, 3to19, 5to17, 4to17, 3to21, 4to22, 5to15, 3to16, 5to22, 6to19
## Negative:  13to17, 9to13, 3to9, 12to13, 1to9, 13to14, 3to10, 9to17, 5to9, 13to18
##     6to9, 14to15, 9to10, 13to19, 10to13, 7to12, 2to7, 5to7, 2to9, 12to17
##     7to17, 7to13, 7to10, 4to10, 7to22, 13to21, 4to9, 6to13, 10to17, 10to15
## PC_ 2
## Positive:  15to17, 15to18, 20to21, 15to19, 14to19, 16to19, 19to20, 15to20, 17to18, 18to20
##     16to21, 16to17, 14to21, 16to20, 14to15, 15to21, 13to15, 14to17, 18to21, 18to19
##     13to20, 12to15, 15to22, 17to20, 14to20, 14to18, 16to18, 19to22, 21to22, 12to19
## Negative:  7to21, 7to20, 7to19, 4to7, 6to7, 3to7, 7to16, 7to10, 2to7, 7to22
##     5to7, 7to18, 7to8, 7to13, 7to17, 7to14, 3to19, 7to12, 7to15, 1to7
##     3to21, 7to9, 6to10, 5to21, 8to10, 3to20, 3to6, 3to8, 3to10, 6to9
## PC_ 3
## Positive:  7to21, 9to17, 7to17, 1to17, 17to19, 7to20, 8to17, 7to19, 6to17, 2to17
##     5to17, 15to17, 1to21, 13to17, 17to22, 3to17, 17to20, 9to19, 4to17, 9to21
##     17to18, 7to16, 7to14, 12to17, 7to18, 16to17, 15to21, 14to17, 17to21, 7to13
## Negative:  2to10, 2to4, 4to8, 1to4, 3to10, 5to8, 4to10, 3to8, 5to10, 2to3
##     2to6, 3to12, 2to5, 3to6, 2to8, 4to5, 8to10, 3to4, 5to12, 4to6
##     1to10, 1to6, 1to3, 1to8, 5to6, 8to12, 2to12, 3to5, 6to12, 10to12
## PC_ 4
## Positive:  7to22, 1to22, 12to22, 8to22, 3to22, 1to21, 2to22, 12to21, 18to22, 6to22
##     9to22, 4to22, 10to22, 8to21, 15to22, 1to20, 1to6, 20to22, 19to22, 13to22
##     17to22, 14to22, 1to12, 17to21, 6to12, 16to22, 5to22, 2to21, 13to17, 21to22
## Negative:  3to16, 4to15, 3to15, 5to15, 5to16, 4to16, 3to14, 7to15, 3to13, 9to16
##     4to14, 9to14, 9to15, 9to13, 5to14, 7to16, 4to13, 2to15, 8to15, 7to14
##     8to16, 2to14, 10to15, 6to15, 5to13, 13to15, 9to19, 8to13, 10to13, 2to16
## PC_ 5
## Positive:  6to18, 10to18, 7to9, 6to19, 7to18, 6to20, 7to19, 6to7, 12to18, 8to18
##     16to17, 6to14, 10to17, 5to18, 6to16, 14to18, 6to12, 12to19, 16to18, 10to20
##     18to20, 10to19, 10to16, 10to12, 4to18, 9to18, 6to10, 3to12, 6to9, 5to16
## Negative:  1to13, 1to15, 1to21, 1to20, 1to16, 2to13, 1to14, 1to22, 1to19, 1to9
##     1to18, 1to4, 1to5, 1to17, 1to12, 5to13, 1to3, 9to13, 1to8, 1to7
##     4to13, 1to10, 3to13, 16to22, 2to15, 1to6, 9to22, 2to22, 3to15, 1to2
```

Next we can find clusters.

Seurat provides statistical tests to inform you how many principal components capture the majority of variance in the data – this told us that many more than 30 PCs were significant. We used 20 principal components for the final analysis, but similar results were obtained using as low as 8 or more than 20 PCs.

The Louvain algorithm, and Seurat, supports the use of a resolution parameter, which sets the 'granularity' of the clustering (the number of clusters in the dataset). Clustering using too low a resolution fails to identify the heterogeneity in the dataset. Clustering using too a high resolution can lead to 'over-clustering', where bona fide clusters are split into multiple highly similar clusters. Optimal resolution was determined through trial and error (as in scRNA-seq analyses), using low to high resolution, until all clusters represented visually distinct structures. We performed Seurat clustering using 4 resolutions (0.6, 0.8, 1.0, 1.2) and analyzed the chromosome structures for all clusters for all resolutions. Resolution 1.0 was used for all datasets, and consistently outputted clusters with the most distinct structures, with minimal over-clustering (which can be corrected by merging two highly similar sub-populations).

Other variables are used to reproduce the original result (older default settings in Seurat). These can be changed. The goal is not to use specific numbers, but to find the numbers which best separate your data into clusters.

We can cluster using a couple of resolutions:

```r
chromo.wt <- FindNeighbors(object = chromo.wt, dims = 1:20, k.param = 30, nn.eps = 0.5)
```

```
## Computing nearest neighbor graph
```

```
## Computing SNN
```

```r
chromo.wt <- FindClusters(object = chromo.wt, resolution = 0.25, random.seed = 0)
```

```
## Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
##
## Number of nodes: 1629
## Number of edges: 94377
##
## Running Louvain algorithm...
## Maximum modularity in 10 random starts: 0.7926
## Number of communities: 2
## Elapsed time: 0 seconds
```

```r
chromo.wt <- FindClusters(object = chromo.wt, resolution = 0.5, random.seed = 0)
```

```
## Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
##
## Number of nodes: 1629
## Number of edges: 94377
##
## Running Louvain algorithm...
## Maximum modularity in 10 random starts: 0.6998
## Number of communities: 4
## Elapsed time: 0 seconds
```

```r
chromo.wt <- FindClusters(object = chromo.wt, resolution = 1.0, random.seed = 0)
```

```
## Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
##
## Number of nodes: 1629
## Number of edges: 94377
##
## Running Louvain algorithm...
## Maximum modularity in 10 random starts: 0.5728
## Number of communities: 6
## Elapsed time: 0 seconds
```

```r
chromo.wt <- FindClusters(object = chromo.wt, resolution = 1.5, random.seed = 0)
```

```
## Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
##
## Number of nodes: 1629
## Number of edges: 94377
##
## Running Louvain algorithm...
## Maximum modularity in 10 random starts: 0.4797
## Number of communities: 9
## Elapsed time: 0 seconds
```

```r
chromo.wt <- FindClusters(object = chromo.wt, resolution = 2.0, random.seed = 0)
```

```
## Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
##
## Number of nodes: 1629
## Number of edges: 94377
##
## Running Louvain algorithm...
## Maximum modularity in 10 random starts: 0.4060
## Number of communities: 12
## Elapsed time: 0 seconds
```

```
chromo.wt <- FindClusters(object = chromo.wt, resolution = 2.5, random.seed = 0)
```

```
## Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
##
## Number of nodes: 1629
## Number of edges: 94377
##
## Running Louvain algorithm...
## Maximum modularity in 10 random starts: 0.3425
## Number of communities: 18
## Elapsed time: 0 seconds
```

Next we calculate tSNE and UMAP embeddings for plotting. These are alternative dimensionality reduction techniques (a la PCA), and are only used for visualization, not clustering. Clustering is done in high dimensional PCA space.

To run UMAP, you must first install the umap-learn python package (e.g. via pip install umap-learn). Details on this package can be found here: https://github.com/lmcinnes/umap. For a more in depth discussion of the mathematics underlying UMAP, see the ArXiv paper here: https://arxiv.org/abs/1802.03426.

```
chromo.wt <- RunTSNE(object = chromo.wt, reduction = "pca", dims = 1:20, tsne.method = "Rtsne", reducti
chromo.wt <- RunUMAP(object = chromo.wt, reduction = "pca", dims = 1:20, reduction.name = "umap", reduc
```

```
## Warning: The following arguments are not used: check_duplicates
```

```
## Warning: The default method for RunUMAP has changed from calling Python UMAP via reticulate to the R
## To use Python UMAP via reticulate, set umap.method to 'umap-learn' and metric to 'correlation'
## This message will be shown once per session
```

```
## 13:03:14 UMAP embedding parameters a = 0.583 b = 1.334
```

```
## 13:03:14 Read 1629 rows and found 20 numeric columns
```

```
## 13:03:14 Using Annoy for neighbor search, n_neighbors = 30
```

```
## 13:03:14 Building Annoy index with metric = cosine, n_trees = 50
```

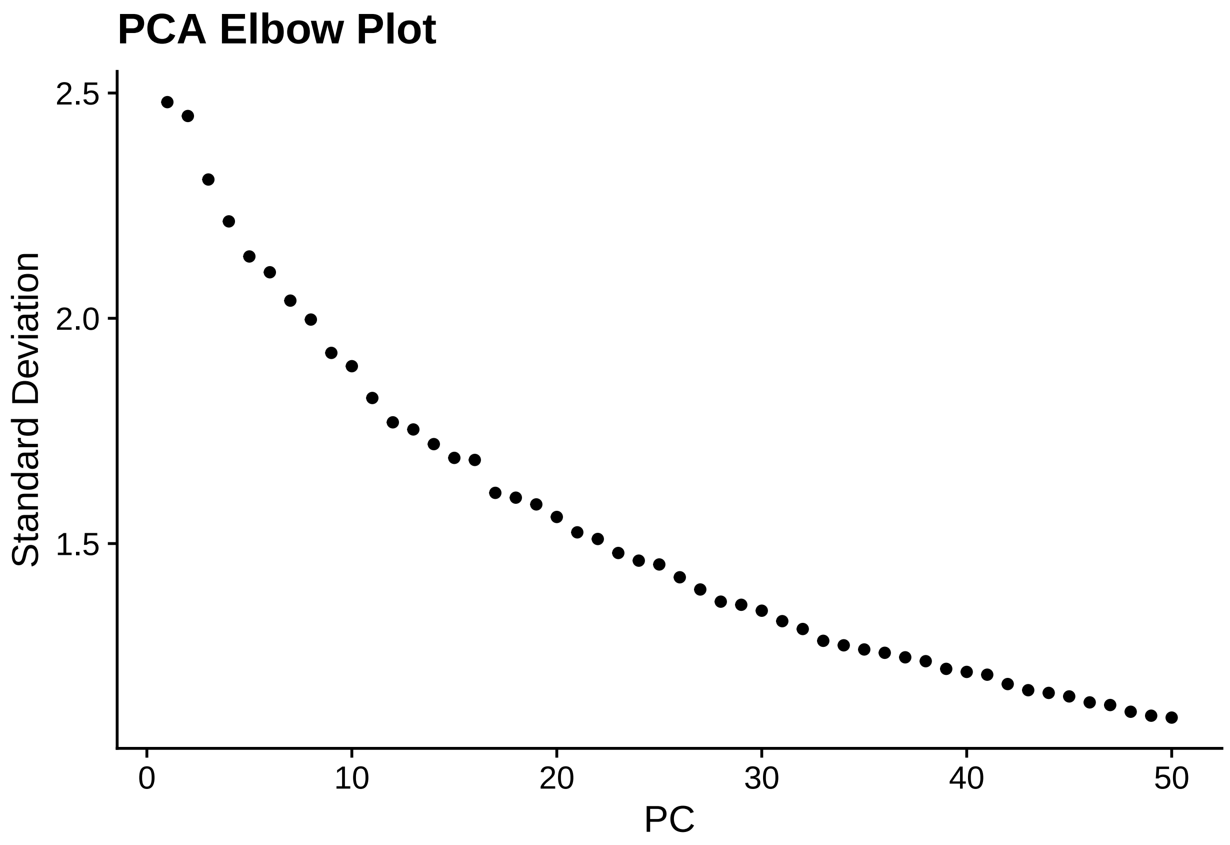
```
## 0%   10   20   30   40   50   60   70   80   90   100%
```

```
## [----|----|----|----|----|----|----|----|----|----|
## **************************************************|
## 13:03:14 Writing NN index file to temp file /var/folders/p4/zxrf946n5b12zyw8hfmn5ksc0000gn/T//RtmpNT
## 13:03:14 Searching Annoy index using 1 thread, search_k = 3000
## 13:03:15 Annoy recall = 100%
## 13:03:15 Commencing smooth kNN distance calibration using 1 thread
## 13:03:15 Initializing from normalized Laplacian + noise
## 13:03:15 Commencing optimization for 500 epochs, with 66224 positive edges
## 13:03:18 Optimization finished
```

Plotting data and looking at clustering results

Clustering results can be examined in multiple ways. We can look at the PCs that contribute the most to clustering:

```r
pca_plot <- ElbowPlot(object = chromo.wt, ndims = 50, reduction = "pca") + ggtitle("PCA Elbow Plot")
pca_plot
```



Typically you are looking for the 'elbow', where there is a clear transition between clusters with high standard deviation (across traces), and those with very little, which aren't informative for the differences between traces.

We can make DimPlots next. Both to show the clusters, as well as the ages.

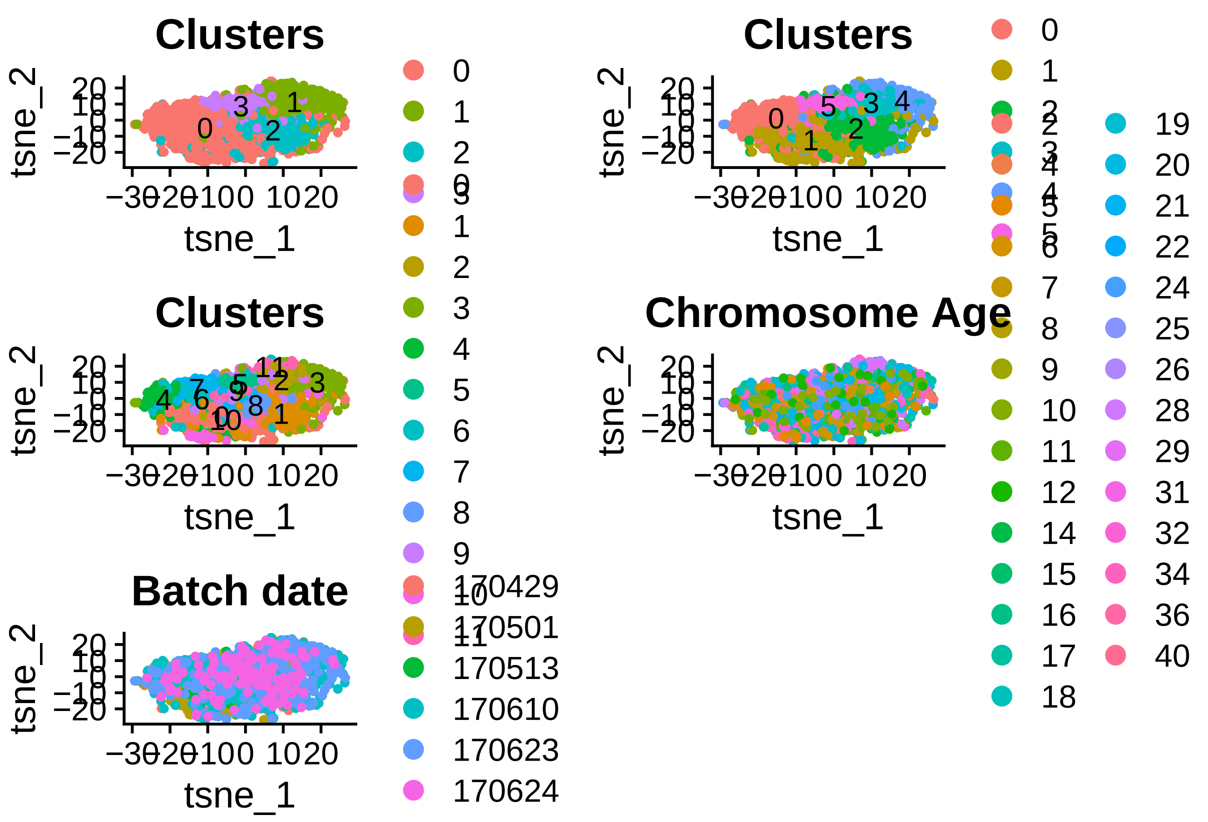
```r
chromo_res.0.5 <- DimPlot(object = chromo.wt, group.by = "RNA_snn_res.0.5", reduction = "tsne", label =
## Warning: Using ‘as.character()‘ on a quosure is deprecated as of rlang 0.3.0.
## Please use ‘as_label()‘ or ‘as_name()‘ instead.
## This warning is displayed once per session.
```

```
chromo_res.1.0 <- DimPlot(object = chromo.wt, group.by = "RNA_snn_res.1", reduction = "tsne", label = T
chromo_res.2.0 <- DimPlot(object = chromo.wt, group.by = "RNA_snn_res.2", reduction = "tsne", label = T

chromo_age <- DimPlot(object = chromo.wt, group.by = "chromosome_age", reduction = "tsne", label = F, pt

chromo_batch <- DimPlot(object = chromo.wt, group.by = "batch", reduction = "tsne", label = F, pt.size =

# use patchwork package to plot them together
chromo_res.0.5 + chromo_res.1.0 + chromo_res.2.0 + chromo_age + chromo_batch + plot_layout(ncol = 2, wid
```



From this, you can see a bunch of clusters, and that they won’t really group by age or batch (traces from embryos of all ages and batches are included in each cluster).

By increasing the resolution, you get more clusters, however, most are probably real clusters that have been fragmented by the higher resolution, and either represent minor differences, or noise due to technical effects (poor trace quality)

We can examine the relationships between cluster IDs from different resolutions to get a sense of this using the clustree package:

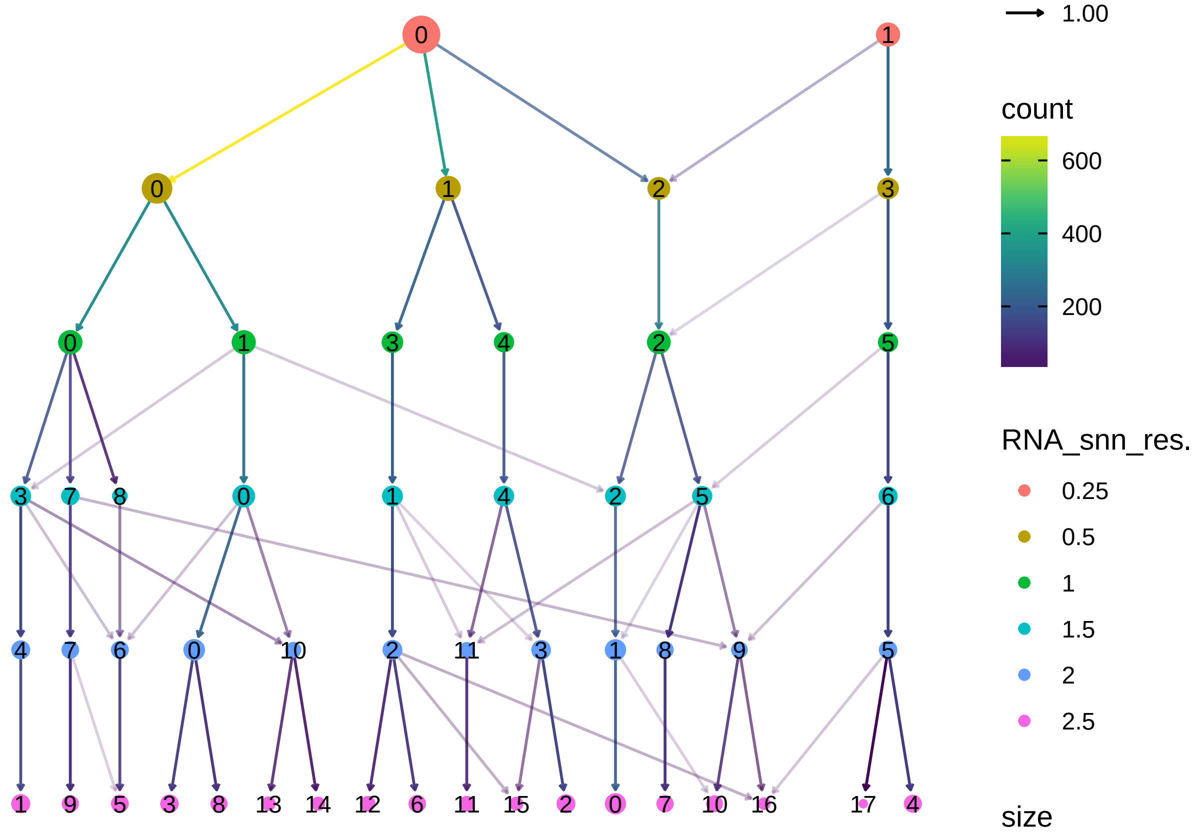
```
library(clustree)
```

```
## Loading required package: ggraph
```

```
meta.data <- chromo.wt@meta.data[,grep("res.", colnames(chromo.wt@meta.data))]

clustree(meta.data, prefix = "RNA_snn_res.", edge_width = 0.5, node_size_range = c(1,6), alt_colour = "
```

```
## Warning: The 'add' argument of 'group_by()' is deprecated as of dplyr 1.0.0.
## Please use the '.add' argument instead.
## This warning is displayed once every 8 hours.
## Call 'lifecycle::last_warnings()' to see where this warning was generated.
```



The criss-crossing lines above resolution 1.0 indicate unstable clustering assignments, and above 1.0 you get too few clusters (for example, cluster 0 and 1 from res 1.0).

The next step is to take the cluster assignments from R and import them back into Matlab for making matrices, to see if the clusters actually represent anything.

We have to modify the files a bit so that they work with your Matlab scripts:

```
meta.data <- chromo.wt@meta.data[, grep("res.", colnames(chromo.wt@meta.data))]
rownames(meta.data) <- str_sub(rownames(meta.data), start = 15) # remove excess info from rownames so t
write.csv(meta.data, file = "Chromosome_5_wt_clustering-results.csv") # saves in current directory
```