Data integration

Maksim Kholmatov

Contents

Introduction

This vignette will demonstrate how to combine separate datasets containing single-cell chromatin data. This is a common task that arises in cases when:

- You have too many cells to fit into a single lane of the 10x chip.
- You produce libraries on multiple days.
- You want to integrate your data with public datasets (to increase statistical power or improve cell type annotations).

References

This vignette is a based on this and this vignettes for the Signac package with some explanations borrowed from the ArchR documentation.

Load required R libraries

Before we start with any analysis, it is good practice to load all required libraries. This will also immediately identify libraries that may be missing. Note that for this course, we pre-installed all libraries for you. When you run your own analysis, you have to check which libraries are already available, and which are not. We use `suppressPackageStartupMessages` here to suppress the output messages of the various packages for reasons of brevity.

In order to control the computing resource allocation we will explicitly load the `future` package and set the required parameters.

When using functions that sample pseudorandom numbers, each time you execute them you will obtain a different result. For the purpose of this vignette, this is not what we want. We therefore set a particular seed value(here: 1990) so that the result will always be the same. For more information, check out this webpage that explains this general concept in more detail.

```
suppressPackageStartupMessages({
library(Seurat)
library(Signac)
library(tidyverse)
library(GenomicRanges)
```

```
library(future)
})

# parallelization parameters
plan("multisession", workers = 8) # number of parallel workers for functions that support it
options(future.globals.maxSize = 40000 * 1024^2) # RAM limit of 40 Gb

set.seed(1990)
```

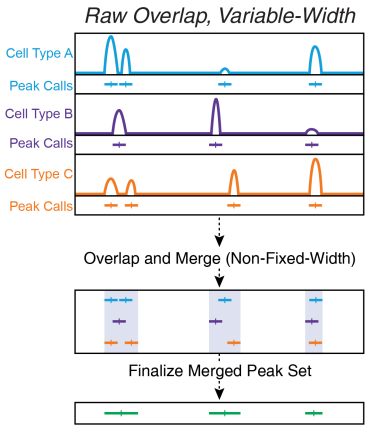
Create Seurat object from multiple datasets

Create consensus peak set

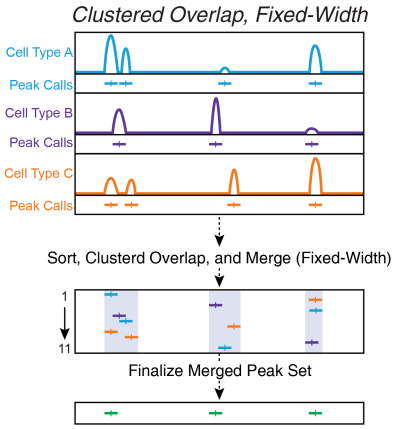
When merging multiple single-cell chromatin datasets, it's important to be aware that if peak calling was performed on each dataset independently, the peaks are unlikely to be exactly the same. We therefore need to create a common set of peaks across all the datasets to be merged.

There are multiple ways to combine different peak sets with different pros and cons. Here is a couple of options:

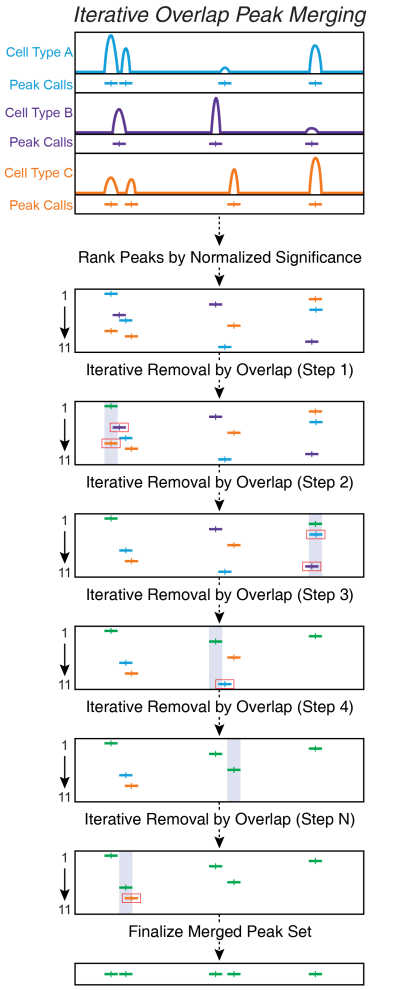
- **Raw peak overlap**: Raw peak overlap involves taking any peaks that overlap each other and merging these into a single larger peak. In this scheme, daisy-chaining can become a problem because peaks that don't directly overlap each other get included in the same larger peak because they are bridged by a shared internal peak. Another problem with this type of approach is that, if you want to keep track of peak summits, you are forced to either pick a single new summit for each new merged peak or keep track of all of the summits that apply to each new merged peak.



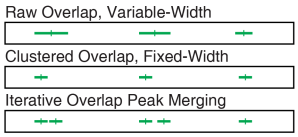
- **Clustered Overlap**: Clustered overlap takes peaks that cluster together and picks a single winner. This is done by keeping the most significant peak in each cluster. This may end up under-calling peaks and missing smaller peaks located nearby.



- **Iterative Overlap**: Iterative overlap removal is the method used in the ArchR package. It avoids the issues mentioned above. Peaks are first ranked by their significance. The most significant peak is retained and any peak that directly overlaps with the most significant peak is removed from further analysis. Then, of the remaining peaks, this process is repeated until no more peaks exist. This avoids daisy-chaining and still allows for use of fixed-width peaks.



Here is the comparisons of results of all 3 methods for the peaks from above.



They all produce distinct sets of peaks, and the **Iterative Overlap** is argued by the authors of ArchR to be the most optimal. However in our analysis we will stick with the **Raw Overlap** method for the sake of simplicity and because it is still widely used.

To create a unified set of peaks we can use functions from the `GenomicRanges` package. The `reduce` function from `GenomicRanges` will merge all intersecting peaks.

```
# Make sure to have a trailing slash here
outFolder="/home/max/Work/EMBL/PROJECTS/atac-seq_course_2022/data/"
sample1Folder = paste0(outFolder, "samples/Subsample01Seed1/")
sample2Folder = paste0(outFolder, "samples/Subsample01Seed2/")
sample3Folder = paste0(outFolder, "samples/Subsample01Seed3/")


# read BED files containing sample-specific peaks as a data.frame
peaks.1 = read.table(
  file = paste0(sample1Folder, "peaks.bed"),
  skip = 52,
  col.names = c("chr", "start", "end")
)
peaks.2 = read.table(
  file = paste0(sample2Folder, "peaks.bed"),
  skip = 52,
  col.names = c("chr", "start", "end")
)
peaks.3 = read.table(
  file = paste0(sample3Folder, "peaks.bed"),
  skip = 52,
  col.names = c("chr", "start", "end")
)

head(peaks.1)
```

```
##    chr   start     end
## 1 chr1 3012183 3013102
## 2 chr1 3035401 3036308
## 3 chr1 3062439 3063334
## 4 chr1 3147316 3148260
## 5 chr1 3191280 3192176
## 6 chr1 3263546 3264432
```

Now we can use these data.frames to create `GenomicRanges` objects for each peak set.

```
# convert data frames to GenomicRanges
gr.1 = makeGRangesFromDataFrame(peaks.1)
gr.2 = makeGRangesFromDataFrame(peaks.2)
gr.3 = makeGRangesFromDataFrame(peaks.3)

rm(peaks.1, peaks.2, peaks.3)
```

```
gr.1
```

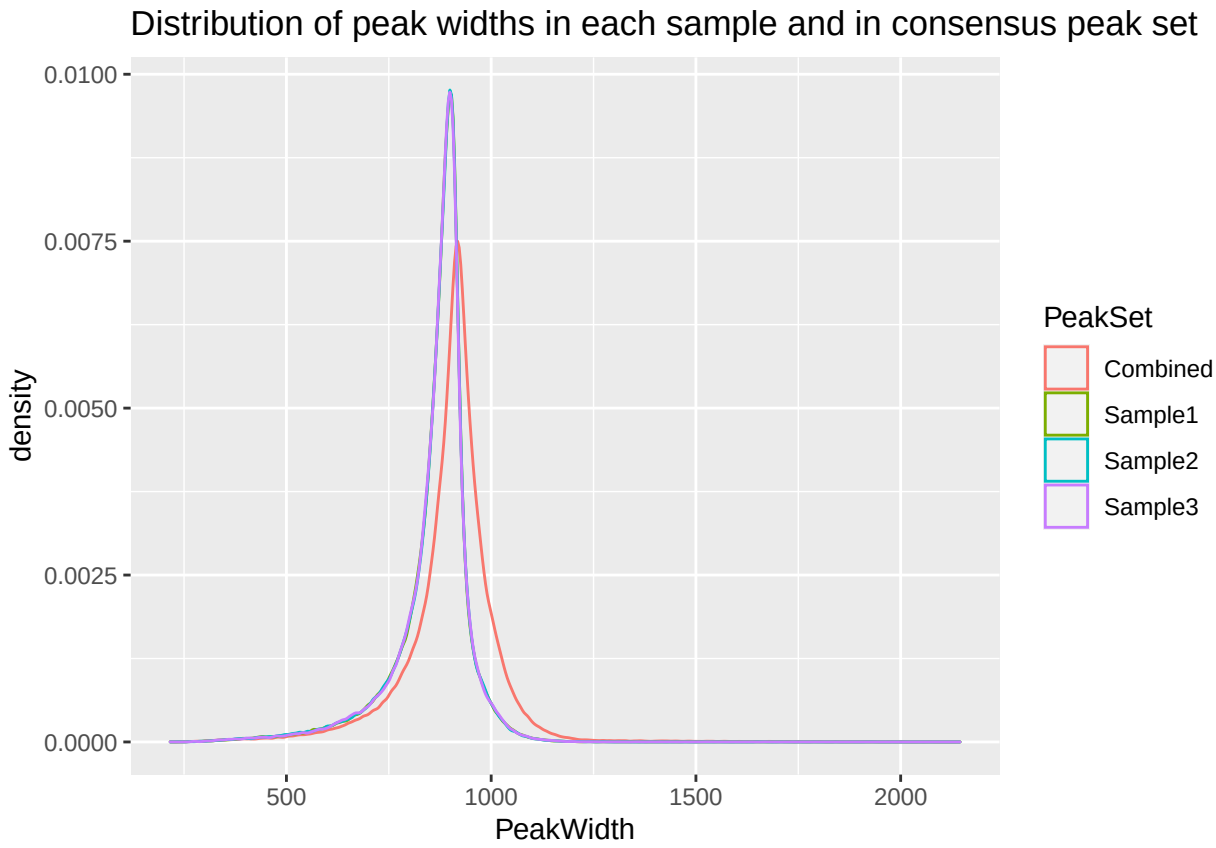
```
## GRanges object with 159254 ranges and 0 metadata columns:
##               seqnames          ranges strand
##                  <Rle>       <IRanges>  <Rle>
##         [1]       chr1 3012183-3013102      *
##         [2]       chr1 3035401-3036308      *
##         [3]       chr1 3062439-3063334      *
##         [4]       chr1 3147316-3148260      *
##         [5]       chr1 3191280-3192176      *
##         ...        ...             ...    ...
##    [159250] GL456216.1     40223-41109      *
##    [159251] GL456216.1     43813-44701      *
##    [159252] GL456216.1     47298-48078      *
##    [159253] GL456216.1     48813-49682      *
##    [159254] JH584295.1       1259-1967      *
##    -------
##    seqinfo: 26 sequences from an unspecified genome; no seqlengths
```

Now we can combine the ranges into a single peak set.

```
# Create a unified set of peaks to quantify in each dataset
combined.peaks = reduce(x = c(gr.1, gr.2, gr.3))
```

If you are concerned about potential drawbacks of using **Raw Overlap** peak merging strategy, we can check how it affects the distribution of peak widths.

```
peak_widths = rbind(
  tibble(PeakWidth = width(combined.peaks), PeakSet="Combined"),
  tibble(PeakWidth = width(gr.1), PeakSet="Sample1"),
  tibble(PeakWidth = width(gr.2), PeakSet="Sample2"),
  tibble(PeakWidth = width(gr.3), PeakSet="Sample3")
  )

ggplot(data = peak_widths, aes(x = PeakWidth, col = PeakSet))+
  geom_density()+
  ggtitle("Distribution of peak widths in each sample and in consensus peak set")
```



We can see that Combined peak set has consistently broader peaks, which is expected from the way smaller sample-specific peaks are merged into one bigger peak. The density distribution may not be indicative of the rare extreme events (narrowest and widest peaks), so we may want to check manually the top and bottom of the distribution.

```
peak_widths %>%
  group_by(PeakSet) %>%
  slice_max(n=10, order_by=PeakWidth) %>%
  mutate(rank = 1:10) %>%
  pivot_wider(names_from = PeakSet, values_from = PeakWidth, id_cols = rank)
```

```
## # A tibble: 10 x 5
##     rank Combined Sample1 Sample2 Sample3
##    <int>    <int>   <int>   <int>   <int>
##  1     1     2145    1848    1944    1963
##  2     2     2138    1840    1932    1943
##  3     3     2072    1711    1927    1868
##  4     4     2072    1707    1925    1812
##  5     5     2070    1706    1916    1791
##  6     6     2067    1703    1879    1761
##  7     7     2050    1693    1841    1736
##  8     8     2043    1666    1789    1678
##  9     9     2018    1657    1788    1672
## 10    10     2017    1653    1756    1670
```

```
peak_widths %>%
  group_by(PeakSet) %>%
  slice_min(n=10, order_by=PeakWidth) %>%
```

```
  mutate(rank = 1:10) %>%
  pivot_wider(names_from = PeakSet, values_from = PeakWidth, id_cols = rank)
```

```
## # A tibble: 10 x 5
##     rank Combined Sample1 Sample2 Sample3
##    <int>    <int>   <int>   <int>   <int>
##  1     1      216     219     216     217
##  2     2      217     243     221     217
##  3     3      217     244     229     228
##  4     4      219     247     234     245
##  5     5      221     250     235     250
##  6     6      228     253     236     254
##  7     7      229     259     240     257
##  8     8      235     261     246     265
##  9     9      236     261     246     266
## 10    10      240     261     260     269
```

If there are any anomalous peaks that arise from the merging we can filter them out. Additionally we can remove all non-standard chromosomes.

```
# Filter out bad peaks based on length
peakwidths = width(combined.peaks)
combined.peaks = combined.peaks[peakwidths  < 10000 & peakwidths > 20]

# might as well filter out non-standard chromosomes
seqlevels(combined.peaks, pruning.mode="coarse") = standardChromosomes(combined.peaks)

combined.peaks
```

```
## GRanges object with 196433 ranges and 0 metadata columns:
##             seqnames            ranges strand
##                <Rle>         <IRanges>  <Rle>
##        [1]     chr1   3003196-3004129      *
##        [2]     chr1   3012183-3013102      *
##        [3]     chr1   3035398-3036342      *
##        [4]     chr1   3062436-3063366      *
##        [5]     chr1   3147316-3148260      *
##        ...      ...               ...    ...
##   [196429]     chrY 90808622-90809350      *
##   [196430]     chrY 90810521-90811352      *
##   [196431]     chrY 90812345-90813082      *
##   [196432]     chrY 90813459-90814185      *
##   [196433]     chrY 90824962-90825857      *
##   -------
##   seqinfo: 21 sequences from an unspecified genome; no seqlengths
```

Create Fragment objects

To quantify our combined set of peaks we'll need to create a `Fragment` object for each experiment. The `Fragment` class is a specialized class defined in Signac to hold all the information related to a single fragment file.

Reading and filtering the cell/barcode metadata First we'll load the cell metadata for each experiment so that we know what cell barcodes are contained in each file

```
# load metadata
md.1 = read.table(
  file = paste0(sample1Folder, "singlecell.csv"),
  stringsAsFactors = FALSE,
  sep = ",",
  header = TRUE,
  row.names = 1
)[-1, ] # remove the first row

md.2 = read.table(
  file = paste0(sample2Folder, "singlecell.csv"),
  stringsAsFactors = FALSE,
  sep = ",",
  header = TRUE,
  row.names = 1
)[-1, ] # remove the first row

md.3 = read.table(
  file = paste0(sample3Folder, "singlecell.csv"),
  stringsAsFactors = FALSE,
  sep = ",",
  header = TRUE,
  row.names = 1
)[-1, ] # remove the first row
```

Now we can filter out non-cells and low quality cells. `singlecell.csv` metadata file already has quite a lot of QC information similar to what you produced in the `QClustering` vignette. You can check all of the options available here, but for simplicity we will just use 2 of the available parameters:

- **is__cell__barcode**: binary indicator of whether barcode is associated with a cell according to cellranger's automated cell calling algorithm.
- **passed__filters**: number of non-duplicate, usable read-pairs i.e. "fragments".

```
# perform an initial filtering of low count cells and non-cells
md.1 = md.1[as.logical(md.1$is__cell_barcode) & (md.1$passed_filters > 1000), ]
md.2 = md.2[as.logical(md.2$is__cell_barcode) & (md.2$passed_filters > 1000), ]
md.3 = md.3[as.logical(md.3$is__cell_barcode) & (md.3$passed_filters > 1000), ]
# As a side note. The cellranger-atac version 2.0 that we ran during testing
# produced a singlecell.csv file with the `is__cell_barcode` column spelled with
# a double underscore. This is probably an innocent bug, that would be fixed
# in later versions.
```

Alternatively to the last step we could have used the Seurat objects that you have created already in the `QClustering` vignette to extract high quality cells since it involved more thorough quality controls:

```
# read the cell names from the filtered seurat objects
cells_from_seurat_sample1 = readRDS(file = paste0(sample1Folder, "obj.filt.rds")) %>%  colnames()
cells_from_seurat_sample2 = readRDS(file = paste0(sample2Folder, "obj.filt.rds")) %>%  colnames()
cells_from_seurat_sample3 = readRDS(file = paste0(sample3Folder, "obj.filt.rds")) %>%  colnames()

md.1 = md.1[cells_from_seurat_sample1, ]
md.2 = md.2[cells_from_seurat_sample2, ]
md.3 = md.3[cells_from_seurat_sample3, ]
```

Creating the fragment objects Now we can create `Fragment` objects using the `CreateFragmentObject` function. The `CreateFragmentObject` function performs some checks to ensure that the file is present on disk and that it is compressed and indexed, computes the MD5 sum for the file and the tabix index so that we can tell if the file is modified at any point, and checks that the expected cells are present in the file.

```
frags.1 = CreateFragmentObject(
  path = paste0(sample1Folder, "fragments.tsv.gz"),
  cells = rownames(md.1)
)
```

```
## Computing hash
```

```
frags.2 = CreateFragmentObject(
  path = paste0(sample2Folder, "fragments.tsv.gz"),
  cells = rownames(md.2)
)
```

```
## Computing hash
```

```
frags.3 = CreateFragmentObject(
  path = paste0(sample3Folder, "fragments.tsv.gz"),
  cells = rownames(md.3)
)
```

```
## Computing hash
```

Quantify peaks in each dataset

We can now create a matrix of peaks x cell for each sample using the `FeatureMatrix` function. This function is parallelized using the `future` package.

```
sample1.counts = FeatureMatrix(
  fragments = frags.1,
  features = combined.peaks,
  cells = rownames(md.1)
)
```

```
## Extracting reads overlapping genomic regions
```

```
sample2.counts = FeatureMatrix(
  fragments = frags.2,
  features = combined.peaks,
  cells = rownames(md.2)
)
```

```
## Extracting reads overlapping genomic regions
```

```
sample3.counts = FeatureMatrix(
  fragments = frags.3,
  features = combined.peaks,
  cells = rownames(md.3)
)
```

```
## Extracting reads overlapping genomic regions
```

Create Seurat objects

We will now use the quantified matrices to create a Seurat object for each dataset, storing the Fragment object for each dataset in the assay.

```
# First we create a ChromatinAssay object
sample1_assay = CreateChromatinAssay(
  sample1.counts,
  fragments = frags.1
)
# Then we create a Seurat object based on this ChromatinAssay
sample1 = CreateSeuratObject(
  sample1_assay,
  assay = "ATAC",
  project = "mouse_ES_LIF",
  meta.data=md.1
)
```

```
## Warning: Keys should be one or more alphanumeric characters followed by an
## underscore, setting key from atac to atac_
```

```
# First we create a ChromatinAssay object
sample2_assay = CreateChromatinAssay(
  sample2.counts,
  fragments = frags.2
)
# Then we create a Seurat object based on this ChromatinAssay
sample2 = CreateSeuratObject(
  sample2_assay,
  assay = "ATAC",
  project = "mouse_ES_LIF",
  meta.data=md.2
)
```

```
## Warning: Keys should be one or more alphanumeric characters followed by an
## underscore, setting key from atac to atac_
```

```
# First we create a ChromatinAssay object
sample3_assay = CreateChromatinAssay(
  sample3.counts,
  fragments = frags.3
)
# Then we create a Seurat object based on this ChromatinAssay
sample3 = CreateSeuratObject(
  sample3_assay,
  assay = "ATAC",
  project = "mouse_ES_LIF",
  meta.data=md.3
)
```

```
## Warning: Keys should be one or more alphanumeric characters followed by an
## underscore, setting key from atac to atac_
```

Merge Seurat objects

Now that the objects each contain an assay with the same set of features, we can use the standard `merge` function to merge the objects. This will also merge all the fragment objects so that we retain the fragment information for each cell in the final merged object.

```
# add information to identify dataset of origin
sample1$dataset = 'sample1'
sample2$dataset = 'sample2'
```

```
sample3$dataset = 'sample3'


# merge all datasets, adding a cell ID to make sure cell names are unique
combined = merge(
  x = sample1,
  y = list(sample2, sample3),
  add.cell.ids = c("1", "2", "3")
)

#Since we renamed corresponding cells in the combined object we should also rename them in separate one
sample1 = RenameCells(sample1, add.cell.id = "1")
sample2 = RenameCells(sample2, add.cell.id = "2")
sample3 = RenameCells(sample3, add.cell.id = "3")


combined
```

```
## An object of class Seurat
## 196433 features across 10883 samples within 1 assay
## Active assay: ATAC (196433 features, 0 variable features)
```

Validate our integration

We finally have our combined object. Now we can perform standard pre-processing, keeping track of each cell's dataset of origin. We can check for any batch effects visually on a UMAP plot.

```
# Perform variable feature selection, data normalization, PCA and
# UMAP embedding into a 2D space.
combined = FindTopFeatures(combined, min.cutoff = 20)
combined = RunTFIDF(combined)
```

```
## Performing TF-IDF normalization
```

```
combined = RunSVD(combined)
```

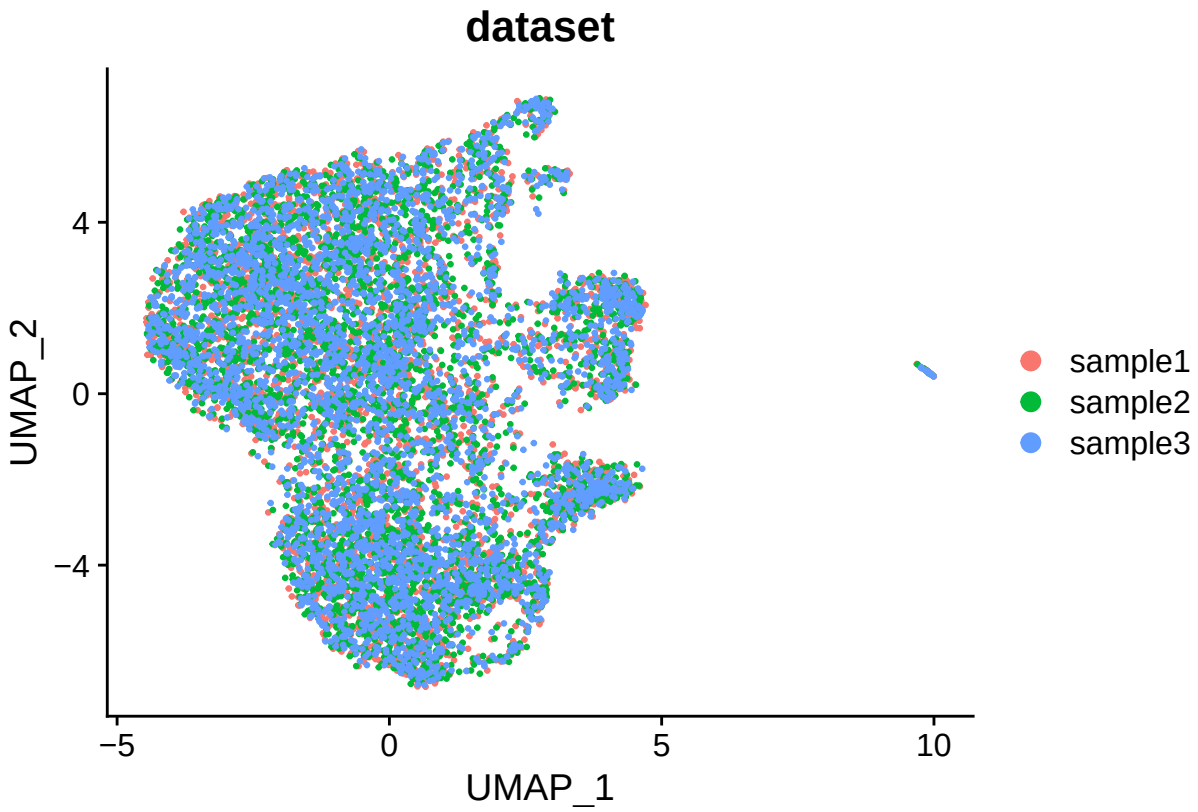
```
## Running SVD
```

```
## Scaling cell embeddings
```

```
combined = RunUMAP(combined, dims = 2:50, reduction = 'lsi', verbose = F)
```

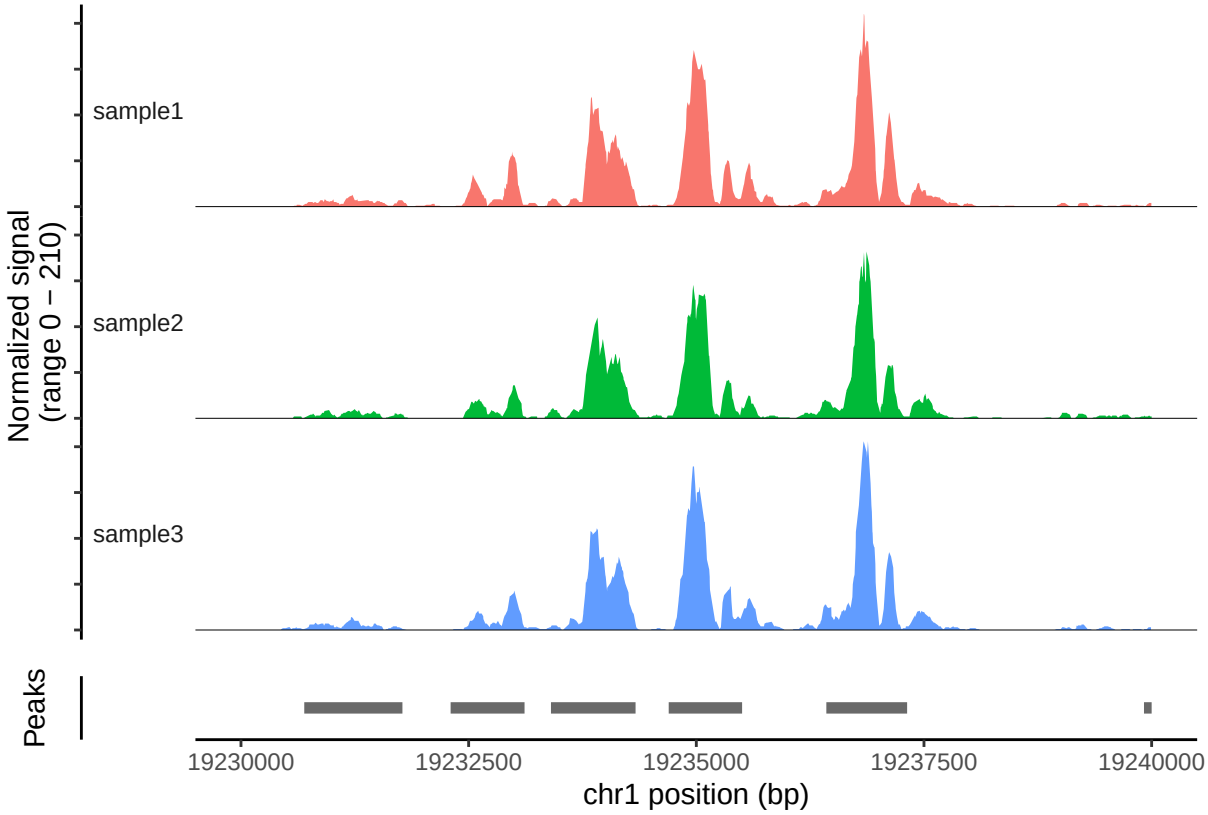
```
## Warning: The default method for RunUMAP has changed from calling Python UMAP via reticulate to the R
## To use Python UMAP via reticulate, set umap.method to 'umap-learn' and metric to 'correlation'
## This message will be shown once per session
```

```
# Plot the resulting UMAP colored by the original dataset of each cell.
DimPlot(combined, group.by = 'dataset', pt.size = 0.5)
```



The merged object contains all fragment objects, and contains an internal mapping of cell names in the object to the cell names in each fragment file so that we can retrieve information from the files without having to change the cell names in each fragment file. We can check that functions that pull data from the fragment files work as expected on the merged object by plotting a region of the genome. Here we can can also check for any significant differences between datasets/batches, although it may depend heavily on the genomice region in question.

```
CoveragePlot(
  object = combined,
  group.by = 'dataset',
  region = "chr1-19230000-19240000"
)
```



Depending on the results of this section we may or may not need to perform further steps in order to mitigate any systematic batch effects between your samples. Whether or not this is needed you can gauge by how much sample dependent clustering is present in your merged object. If the cells from different datasets are more or less evenly distributed across your UMAP plot you can assume that there is no major batch effect that needs correcting. If the cells cluster primarily based on the dataset or within each cell type-specific cluster they further subcluster based on the initial dataset you probably need to correct this dataset-specific variation in order to avoid any downstream artefacts stemming from this dataset-specific variation.

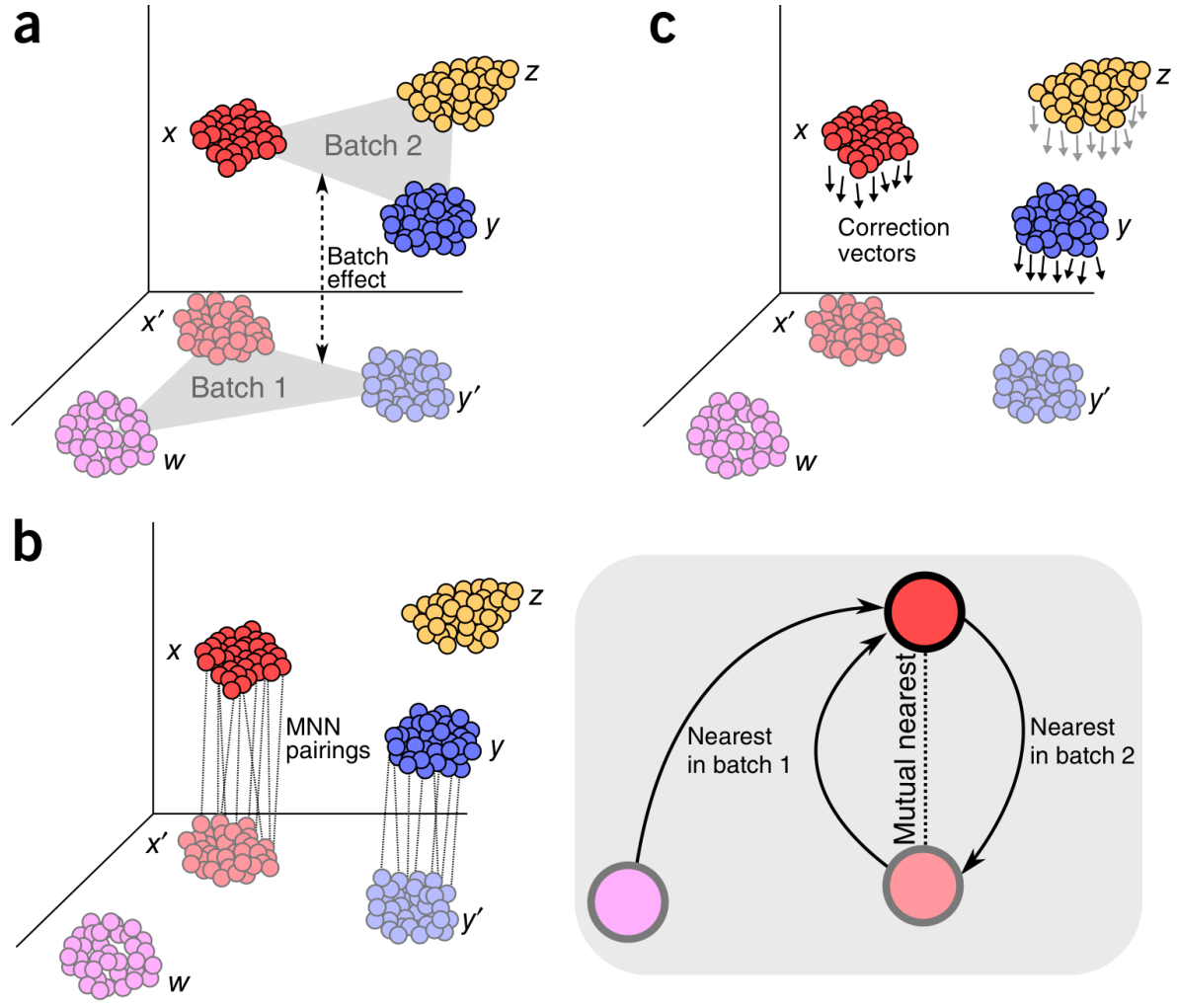
P.S.: One special case you need to keep in mind in your analyses is when you observe a cluster that fully or mostly corresponds to one dataset, but you don't observe similar imbalance in other clusters. This may correspond to a batch effect, but in many cases it can represent genuine heterogeneity in cell type composition between your datasets.

Integration

If you are following this part of the vignette it means that you found significant evidence for a batch effect between the datasets you were integrating in the previous steps.

We will integrate your datasets, using tools available in the Seurat package. This method is based on identifying so-called **mutual nearest neighbors** (Haghverdi et al., 2018) or **anchors** (Stuart, Butler, et al., 2019). These are pairs of cells from different datasets that correspond to the same or similar cell state. Once you have this set of **anchors** you can find a transformation that would project these anchors from dataset 1 into the space of the dataset 2 while minimising the distance between the projection it's corresponding anchor in dataset 2.

Following figure from Haghverdi et al., 2018 illustrates the general idea behind the approach.



Perform LSI embedding on separate datasets

Rather than integrating the normalized data matrix, as is typically done for scRNA-seq data, we'll integrate the low-dimensional cell embeddings (the LSI coordinates) across the datasets using the `IntegrateEmbeddings` function. This is much better suited to scATAC-seq data, as we typically have a very sparse matrix with a large number of features. Note that this requires that we first compute an uncorrected LSI embedding using the merged dataset (as we did above).

```
sample1 = FindTopFeatures(sample1, min.cutoff = 20)
sample1 = RunTFIDF(sample1)
```

```
## Performing TF-IDF normalization
```

```
## Warning in RunTFIDF.default(object = GetAssayData(object = object, slot =
## "counts"), : Some features contain 0 total counts
```

```
sample1 = RunSVD(sample1)
```

```
## Running SVD
```

```
## Scaling cell embeddings
```

```
sample2 = FindTopFeatures(sample2, min.cutoff = 20)
sample2 = RunTFIDF(sample2)
```

```
## Performing TF-IDF normalization
```

```
sample2 = RunSVD(sample2)
```

```
## Running SVD
## Scaling cell embeddings
```

```
sample3 = FindTopFeatures(sample3, min.cutoff = 20)
sample3 = RunTFIDF(sample3)
```

```
## Performing TF-IDF normalization

## Warning in RunTFIDF.default(object = GetAssayData(object = object, slot =
## "counts"), : Some features contain 0 total counts
```

```
sample3 = RunSVD(sample3)
```

```
## Running SVD
## Scaling cell embeddings
```

Identify integration anchors

To find integration anchors between the two datasets, we need to project them into a shared low-dimensional space. To do this, we'll use reciprocal LSI projection (projecting each dataset into the others LSI space) by setting `reduction="rlsi"`.

```
# find integration anchors
integration.anchors = FindIntegrationAnchors(
  object.list = list(sample1, sample2, sample3),
  anchor.features = rownames(sample1),
  reduction = "rlsi",
  dims = 2:30
)
```

```
## Computing within dataset neighborhoods

## Finding all pairwise anchors

## Warning: No filtering performed if passing to data rather than counts

## Projecting new data onto SVD

## Projecting new data onto SVD

## Finding neighborhoods

## Finding anchors

##  Found 6389 anchors

## Warning: No filtering performed if passing to data rather than counts

## Projecting new data onto SVD

## Projecting new data onto SVD

## Finding neighborhoods

## Finding anchors

##  Found 6410 anchors

## Warning: No filtering performed if passing to data rather than counts

## Projecting new data onto SVD
```

```
## Projecting new data onto SVD

## Finding neighborhoods

## Finding anchors

##  Found 6399 anchors
```

Integrate datasest

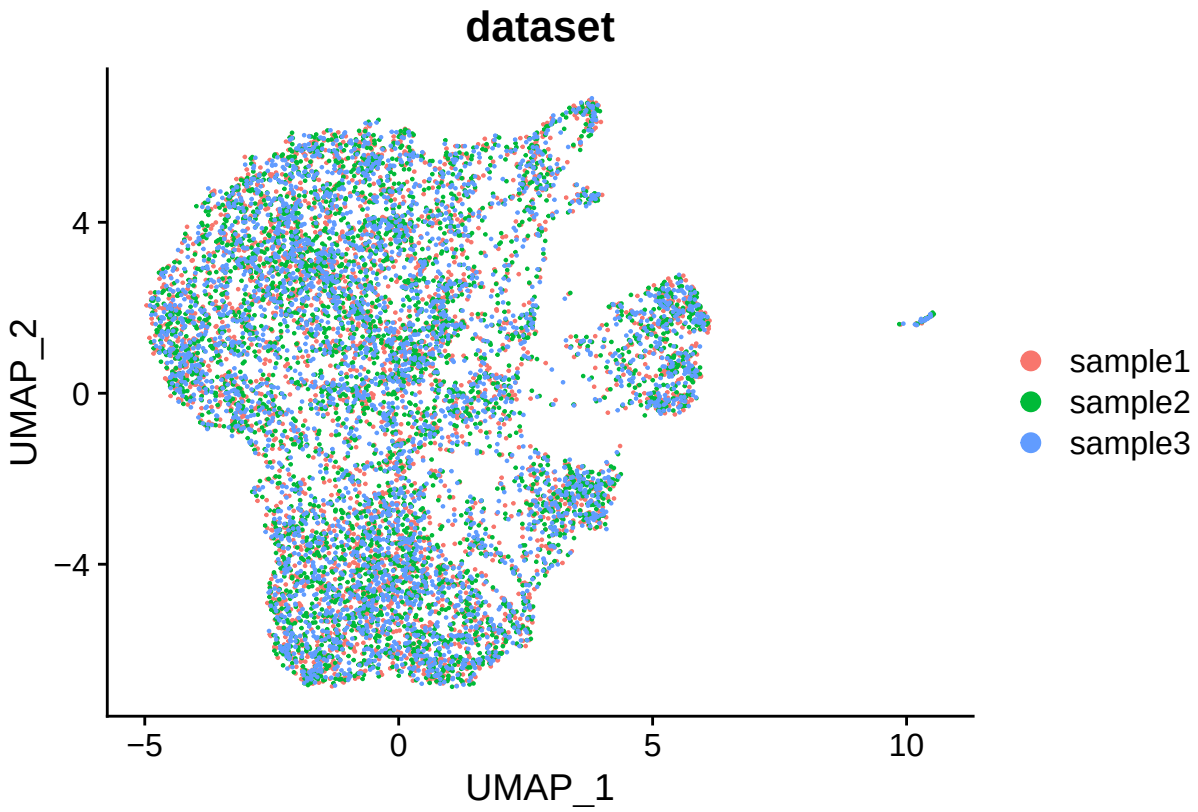
```
# integrate LSI embeddings
integrated = IntegrateEmbeddings(
  anchorset = integration.anchors,
  reductions = combined[["lsi"]],
  new.reduction.name = "integrated_lsi",
  dims.to.integrate = 1:30
)
```

```
## Merging dataset 1 into 3

## Extracting anchors for merged samples

## Finding integration vectors

## Finding integration vector weights

## Integrating data

## Merging dataset 2 into 3 1

## Extracting anchors for merged samples

## Finding integration vectors

## Finding integration vector weights

## Integrating data
```

Check integration results

Now that we have our final integration in the LSI space we can make another UMAP plot to see whether is was successful.

```
# create a new UMAP using the integrated embeddings
integrated = RunUMAP(integrated, reduction = "integrated_lsi", dims = 2:30, verbose = F)
DimPlot(integrated, group.by = "dataset")
```



Hopefully now all the samples are well integrated.

P.S.: The method we presented here is not the only one that can correct batch effects in single cell data. But one needs to pay attention when applying these tools to whether they were designed to work specifically with scRNA data. In case of the **anchor** based integration the method can be easily extended for use with scATAC data, but even in this case some adjustments were required. As was mentioned above partly due to extreme sparsity of the data we had to use reciprocal LSI for anchor identification instead of Canonical Correlation Analysis which is the original approach described for scRNA data. Another popular tool for data integration is Harmony, which is the default approach for dealing with batch effects when using ArchR, but it can also be used with a Seurat object.

Session Info

It is good practice to print the so-called session info at the end of an R script, which prints all loaded libraries, their versions etc. This can be helpful for reproducibility and recapitulating which package versions have been used to produce the results obtained above.

```
sessionInfo()
```

```
## R version 4.2.0 (2022-04-22)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Arch Linux
##
## Matrix products: default
## BLAS:   /usr/lib/libblas.so.3.10.1
## LAPACK: /usr/lib/liblapack.so.3.10.1
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
```

```
##  [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] stats4    stats     graphics  grDevices utils     datasets  methods
## [8] base
##
## other attached packages:
##  [1] future_1.25.0        GenomicRanges_1.48.0 GenomeInfoDb_1.32.1
##  [4] IRanges_2.30.0       S4Vectors_0.34.0     BiocGenerics_0.42.0
##  [7] forcats_0.5.1        stringr_1.4.0        dplyr_1.0.9
## [10] purrr_0.3.4          readr_2.1.2          tidyr_1.2.0
## [13] tibble_3.1.6         ggplot2_3.3.5        tidyverse_1.3.1
## [16] Signac_1.6.0         sp_1.4-7             SeuratObject_4.1.0
## [19] Seurat_4.1.1
##
## loaded via a namespace (and not attached):
##   [1] readxl_1.4.0           backports_1.4.1        fastmatch_1.1-3
##   [4] plyr_1.8.7             igraph_1.3.1           lazyeval_0.2.2
##   [7] splines_4.2.0          BiocParallel_1.30.0    listenv_0.8.0
##  [10] SnowballC_0.7.0        scattermore_0.8        digest_0.6.29
##  [13] htmltools_0.5.2        fansi_1.0.3            magrittr_2.0.3
##  [16] tensor_1.5             cluster_2.1.3          ROCR_1.0-11
##  [19] tzdb_0.3.0             globals_0.14.0         Biostrings_2.64.0
##  [22] modelr_0.1.8           matrixStats_0.62.0     docopt_0.7.1
##  [25] spatstat.sparse_2.1-1  colorspace_2.0-3       rvest_1.0.2
##  [28] ggrepel_0.9.1          haven_2.5.0            xfun_0.30
##  [31] sparsesvd_0.2          crayon_1.5.1           RCurl_1.98-1.6
##  [34] jsonlite_1.8.0         progressr_0.10.0       spatstat.data_2.2-0
##  [37] survival_3.3-1         zoo_1.8-10             glue_1.6.2
##  [40] polyclip_1.10-0        gtable_0.3.0           zlibbioc_1.42.0
##  [43] XVector_0.36.0         leiden_0.3.10          future.apply_1.9.0
##  [46] abind_1.4-5            scales_1.2.0           DBI_1.1.2
##  [49] spatstat.random_2.2-0  miniUI_0.1.1.1         Rcpp_1.0.8.3
##  [52] viridisLite_0.4.0      xtable_1.8-4           reticulate_1.24
##  [55] spatstat.core_2.4-2    htmlwidgets_1.5.4      httr_1.4.2
##  [58] RColorBrewer_1.1-3     ellipsis_0.3.2         ica_1.0-2
##  [61] pkgconfig_2.0.3        farver_2.1.0           dbplyr_2.1.1
##  [64] ggseqlogo_0.1          uwot_0.1.11            deldir_1.0-6
##  [67] utf8_1.2.2             labeling_0.4.2         tidyselect_1.1.2
##  [70] rlang_1.0.2            reshape2_1.4.4         later_1.3.0
##  [73] cellranger_1.1.0       munsell_0.5.0          tools_4.2.0
##  [76] cli_3.3.0              generics_0.1.2         broom_0.8.0
##  [79] ggridges_0.5.3         evaluate_0.15          fastmap_1.1.0
##  [82] yaml_2.3.5             goftest_1.2-3          fs_1.5.2
##  [85] knitr_1.39             fitdistrplus_1.1-8     RANN_2.6.1
##  [88] pbapply_1.5-0          nlme_3.1-157           mime_0.12
##  [91] slam_0.1-50            RcppRoll_0.3.0         xml2_1.3.3
##  [94] compiler_4.2.0         rstudioapi_0.13        plotly_4.10.0
##  [97] png_0.1-7              spatstat.utils_2.3-0   reprex_2.0.1
## [100] tweenr_1.0.2           stringi_1.7.6          highr_0.9
## [103] RSpectra_0.16-1        rgeos_0.5-9            lattice_0.20-45
```

```
## [106] Matrix_1.4-1           vctrs_0.4.1            pillar_1.7.0
## [109] lifecycle_1.0.1        spatstat.geom_2.4-0    lmtest_0.9-40
## [112] RcppAnnoy_0.0.19       data.table_1.14.2      cowplot_1.1.1
## [115] bitops_1.0-7           irlba_2.3.5            httpuv_1.6.5
## [118] patchwork_1.1.1        R6_2.5.1               promises_1.2.0.1
## [121] KernSmooth_2.23-20     gridExtra_2.3          lsa_0.73.2
## [124] parallelly_1.31.1      codetools_0.2-18       MASS_7.3-56
## [127] assertthat_0.2.1       withr_2.5.0            qlcMatrix_0.9.7
## [130] sctransform_0.3.3      Rsamtools_2.12.0       GenomeInfoDbData_1.2.8
## [133] hms_1.1.1              mgcv_1.8-40            parallel_4.2.0
## [136] grid_4.2.0             rpart_4.1.16           rmarkdown_2.14
## [139] Rtsne_0.16             ggforce_0.3.3          lubridate_1.8.0
## [142] shiny_1.7.1
```