Computer Lab, Introduction to Bioinformatics (ARCAID course): scRNA-seq data analysis

Perry Moerland

Wednesday, March 9, 2022

1 Preliminaries

In these computer exercises you will mainly use the statistical software package R. Since we will focus on just running the code and interpretation of the results, no previous exposure to R is required. If you want to learn more about R, see our Amsterdam UMC Doctoral School course Computing in R.

The goal of this computer lab is to give you an overview of some of the steps typically applied when analyzing scRNA-seq data:

- Quality control and filtering
- Normalization and feature selection
- Dimensionality reduction and visualization
- Clustering
- Cell annotation

1.1 Setting up R

First download the Rmd (Rmarkdown) file and open it in RStudio (Alle programma's - R - RStudio, and 'Ignore Update'). **If you didn't do so yet**, first install the different R packages that you'll need. In order to execute R code from within RStudio, just click the green arrow head in the chunk of code shown below or put the cursor somewhere in the chunk and select Run - Run Current Chunk from the menu. You can also execute code line-by-line using Ctrl-Enter:

```
# If you didn't do so yet, first install the required packages and their dependencies.
# Installation of packages might take a few minutes
# If in the console you are asked "Update all/some/none? [a/s/n]:". Just reply "n"
install.packages("BiocManager")
```

```
# package 'BiocManager' successfully unpacked and MD5 sums checked
#
# The downloaded binary packages are in
#   C:\Users\pdmoerland\AppData\Local\Temp\Rtmp8cKOcr\downloaded_packages
```

```
BiocManager::install(c("Seurat","hdf5r","dplyr","ggplot2","alluvial","bioDist"))
```

Now load the libraries so that you can use the functions defined in them:

```
library(Seurat)
library(dplyr)
library(ggplot2)
library(alluvial)
library(bioDist)
```

1.2 Datasets

For this tutorial we will use three different PBMC datasets from the 10x Genomics website.

- 1k PBMCs using 10X v2 chemistry
- 1k PBMCs using 10X v3 chemistry
- 1k PBMCs using 10X v3 chemistry in combination with cell surface proteins, but disregarding the protein data and only looking at gene expression.

1.3 Downloading and importing the data and creating a Seurat object

Here, we download the data and then use the function `Read10X_h5` of the **Seurat** package to read in the expression matrices. R stores these matrices as sparse matrix objects, which are essentially memory-efficient tables of values. In this case the values represent the RNA counts in each cell.

```
urlroot <-
  "https://github.com/LeidenCBC/MGC-BioSB-SingleCellAnalysis2021/raw/main/session-qc-normalization/"
dir.create("./data/")
file_list <- c("pbmc_1k_v3_filtered_feature_bc_matrix.h5",
               "pbmc_1k_v2_filtered_feature_bc_matrix.h5",
               "pbmc_1k_protein_v3_filtered_feature_bc_matrix.h5")
for (i in file_list) {
    download.file(url = paste0(urlroot, i), destfile = paste0("./data/", i), mode = "wb")
}
setwd("data")
v3.1k <- Read10X_h5("pbmc_1k_v3_filtered_feature_bc_matrix.h5")
v2.1k <- Read10X_h5("pbmc_1k_v2_filtered_feature_bc_matrix.h5")
p3.1k <- Read10X_h5("pbmc_1k_protein_v3_filtered_feature_bc_matrix.h5")
```

```
# Genome matrix has multiple modalities, returning a list of matrices for this genome
```

```
# select only gene expression data and exclude the cell surface protein (CITE-seq) data.
p3.1k <- p3.1k$`Gene Expression`
setwd("..")
```

Note that in R you can obtain a detailed explanation of a function by typing `?` followed by the name of the function in the Console window, for example `?Read10X_h5`.

Rather than working directly with matrices, Seurat works with custom objects that wrap around them. These Seurat objects also conveniently contain tables of metadata for the cells and features, which avoids the clutter of managing them as separate objects. As we will see later, normalized expression values are stored in a separate matrix within the Seurat object, which allows us to play around with different normalization strategies without manually keeping a backup of the original values. In addition to RNA counts, we are able to store additional data types (termed assays) within the Seurat object, such as protein measurements measured by CITE-seq, though we will stick to the default RNA assay here.

First, create Seurat objects for each of the datasets, and then merge into one large Seurat object. We will use the cell metadata to keep track of which dataset the cell originated from.

```
sdata.v2.1k <- CreateSeuratObject(v2.1k, project = "v2.1k")
sdata.v3.1k <- CreateSeuratObject(v3.1k, project = "v3.1k")
sdata.p3.1k <- CreateSeuratObject(p3.1k, project = "p3.1k")
# Merge into one single Seurat object. +
# Prefix cell ids with dataset name (`all.cell.ids`) just in case you have
# overlapping barcodes between the datasets.
alldata <- merge(sdata.v2.1k, c(sdata.v3.1k, sdata.p3.1k), add.cell.ids=c("v2.1k","v3.1k","p3.1k"))
# Also add in a metadata column that indicates v2 vs v3 chemistry.
chemistry <- rep("v3", ncol(alldata))
```

```
chemistry[Idents(alldata) == "v2.1k"] <- "v2"
alldata <- AddMetaData(alldata, chemistry, col.name = "Chemistry")
alldata
```

```
# An object of class Seurat
# 33538 features across 2931 samples within 1 assay
# Active assay: RNA (33538 features, 0 variable features)
```

The metadata of the Seurat object, which itself is a data frame, can be accessed using the slot operator (`@`) like so `alldata@meta.data`. Alternatively one can call the object with double empty square brackets: `alldata[[]]`. Another slot to be aware of is `alldata@active.ident`, or alternatively `Idents(alldata)`, which stores a column of the metadata that should be used to identify groups of cells. The value of the identities are by default chosen to be whatever is passed to the `project` parameter in the `CreateSeuratObject` call, and is stored in the `orig.ident` column of the metadata object. We are free to change the column that represents the cell identities but for this tutorial (and in the general case) we keep it as is.

Let's check number of cells from each dataset using the `Idents`.

```
table(Idents(alldata))
```

```
#
# p3.1k v2.1k v3.1k
#   713   996  1222
```

Let's also have a look at part of the count data.

```
as.matrix(alldata[["RNA"]]@counts[1:10, 1:5])
```

```
#             v2.1k_AAACCTGAGCGCTCCA-1 v2.1k_AAACCTGGTGATAAAC-1
# MIR1302-2HG                        0                        0
# FAM138A                            0                        0
# OR4F5                              0                        0
# AL627309.1                         0                        0
# AL627309.3                         0                        0
# AL627309.2                         0                        0
# AL627309.4                         0                        0
# AL732372.1                         0                        0
# OR4F29                             0                        0
# AC114498.1                         0                        0
#             v2.1k_AAACGGGGTTTGTGTG-1 v2.1k_AAAGATGAGTACTTGC-1
# MIR1302-2HG                        0                        0
# FAM138A                            0                        0
# OR4F5                              0                        0
# AL627309.1                         0                        0
# AL627309.3                         0                        0
# AL627309.2                         0                        0
# AL627309.4                         0                        0
# AL732372.1                         0                        0
# OR4F29                             0                        0
# AC114498.1                         0                        0
#             v2.1k_AAAGCAAGTCTCTTAT-1
# MIR1302-2HG                        0
# FAM138A                            0
# OR4F5                              0
# AL627309.1                         0
# AL627309.3                         0
# AL627309.2                         0
```

```
# AL627309.4                        0
# AL732372.1                        0
# OR4F29                            0
# AC114498.1                        0
```

Question 1 What do the columns and rows correspond to? How can you see that this is probably data corresponding to an scRNA-seq experiment?

Let's now show the 10 genes that are on average most highly expressed.

```
indx <- order(rowSums(alldata[["RNA"]]@counts),decreasing = TRUE)[1:10]
as.matrix(sdata.v3.1k[["RNA"]]@counts[indx, 1:5])
```

```
#         AAACCCAAGGAGAGTA-1 AAACGCTTCAGCCCAG-1 AAAGAACAGACGACTG-1
# MALAT1                 182                243                271
# MT-CO1                 186                 76                 56
# MT-CO3                 157                 73                 47
# MT-CO2                  90                 50                 38
# EEF1A1                  67                 85                 53
# MT-ATP6                 88                 56                 30
# B2M                     84                 29                 81
# RPS27                   19                 69                 70
# TMSB4X                  74                 16                 51
# MT-ND3                  82                 54                 14
#         AAAGAACCAATGGCAG-1 AAAGAACGTCTGCAAT-1
# MALAT1                 159                275
# MT-CO1                  25                 52
# MT-CO3                  22                107
# MT-CO2                  20                 37
# EEF1A1                  24                116
# MT-ATP6                 24                 53
# B2M                     53                 99
# RPS27                   32                103
# TMSB4X                  23                 69
# MT-ND3                  15                 48
```

Question 2 Is there a particular class of genes that seems consistently highly expressed? Is this something to worry about? If so, why?

2 Quality control (QC)

Before analysing the single-cell gene expression data, we must ensure that all cellular barcode data correspond to viable cells. Cell QC is commonly performed based on three QC covariates:

1. The number of UMIs per barcode (count depth);
2. The number of expressed genes per barcode;
3. The fraction of counts from mitochondrial genes per barcode.

The distributions of these QC covariates are examined for outlier values that are then filtered out by thresholding. These outlier barcodes can correspond to dying cells, cells whose membranes are broken, or doublets.

On object creation, Seurat automatically calculates the first two of these QC statistics namely the number of UMIs and the number of expressed genes (features) per cell. This information is stored in the columns `nCount_RNA` and `nFeature_RNA` of the metadata, respectively.

```
head(alldata@meta.data)
```

```
#                          orig.ident nCount_RNA nFeature_RNA Chemistry
# v2.1k_AAACCTGAGCGCTCCA-1      v2.1k       6631         2029        v2
# v2.1k_AAACCTGGTGATAAAC-1      v2.1k       2196          881        v2
# v2.1k_AAACGGGGTTTGTGTG-1      v2.1k       2700          791        v2
# v2.1k_AAAGATGAGTACTTGC-1      v2.1k       3551         1183        v2
# v2.1k_AAAGCAAGTCTCTTAT-1      v2.1k       3080         1333        v2
# v2.1k_AAAGCAATCCACGAAT-1      v2.1k       5769         1556        v2
```

Note that the `_RNA` suffix is due to the aforementioned potential to hold multiple assays. The default assay is named `RNA`, accessible by `alldata[["RNA"]]` or using the assays slot `alldata@assays$RNA`, which is by default set to be the standard active assay (see `alldata@active.assay`). Effectively this means that any calls that are done on the Seurat object are applied on the `RNA` assay data.

2.1 Calculate mitochondrial proportion

We manually calculate the proportion of mitochondrial reads and add it to the metadata table. Mitochondrial genes start with a `MT-` prefix.

```
percent.mito <- PercentageFeatureSet(alldata, pattern = "^MT-")
alldata <- AddMetaData(alldata, percent.mito, col.name = "percent.mito")
```

2.2 Calculate ribosomal proportion

In the same manner we calculate the proportion of the counts that come from genes coding for ribosomal proteins, identified by the `RPS` and `RPL` prefixes.

```
percent.ribo <- PercentageFeatureSet(alldata, pattern = "^RP[SL]")
alldata <- AddMetaData(alldata, percent.ribo, col.name = "percent.ribo")
```

Now have another look at the metadata table.

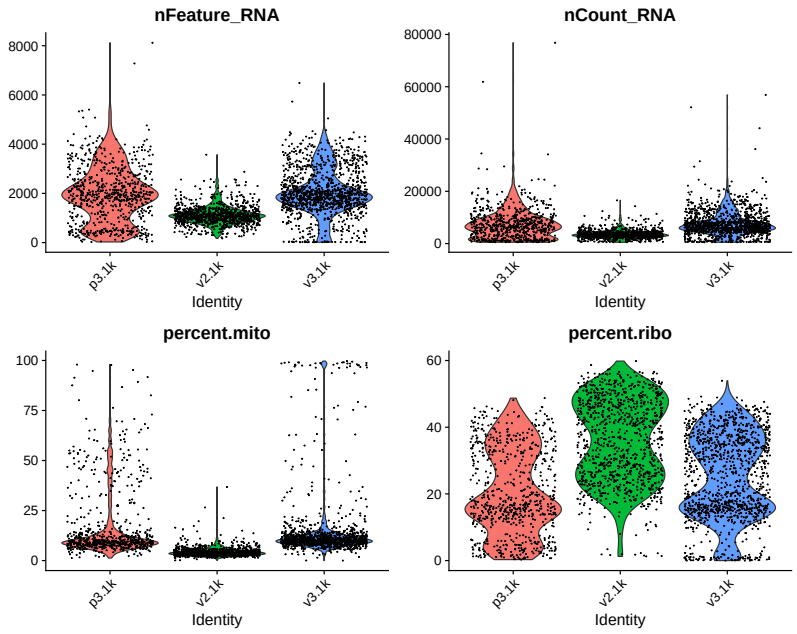
```
head(alldata@meta.data)
```

```
#                          orig.ident nCount_RNA nFeature_RNA Chemistry
# v2.1k_AAACCTGAGCGCTCCA-1      v2.1k       6631         2029        v2
# v2.1k_AAACCTGGTGATAAAC-1      v2.1k       2196          881        v2
# v2.1k_AAACGGGGTTTGTGTG-1      v2.1k       2700          791        v2
# v2.1k_AAAGATGAGTACTTGC-1      v2.1k       3551         1183        v2
# v2.1k_AAAGCAAGTCTCTTAT-1      v2.1k       3080         1333        v2
# v2.1k_AAAGCAATCCACGAAT-1      v2.1k       5769         1556        v2
#                          percent.mito percent.ribo
# v2.1k_AAACCTGAGCGCTCCA-1     5.172674     25.84829
# v2.1k_AAACCTGGTGATAAAC-1     4.143898     20.81056
# v2.1k_AAACGGGGTTTGTGTG-1     3.296296     51.55556
# v2.1k_AAAGATGAGTACTTGC-1     5.885666     29.25936
# v2.1k_AAAGCAAGTCTCTTAT-1     2.987013     17.53247
# v2.1k_AAAGCAATCCACGAAT-1     2.010747     45.69249
```

2.3 QC plots

Now we can plot some of the QC measures as violin plots. Note that Seurat by default will generate a violin plot per identity class.

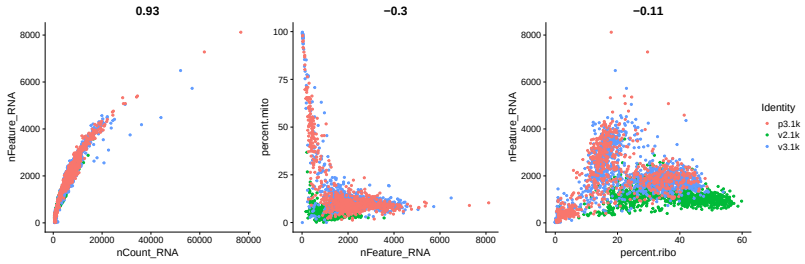
```
VlnPlot(alldata, features = c("nFeature_RNA", "nCount_RNA", "percent.mito", "percent.ribo"),
        ncol = 2, pt.size = 0.1) + NoLegend()
```



As you can see, the v2 chemistry gives lower gene detection, but higher detection of ribosomal genes. As the ribosomal genes are highly expressed they will make up a larger proportion of the transcriptional landscape when fewer of the lowly expressed genes are detected.

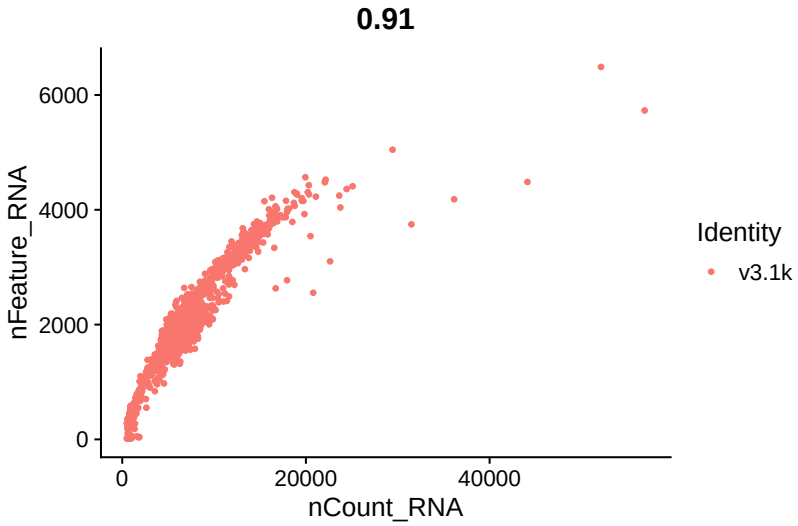
We can also plot the different QC measures as scatter plots.

```
p1 <- FeatureScatter(alldata, feature1 = "nCount_RNA", feature2 = "nFeature_RNA") + NoLegend()
p2 <- FeatureScatter(alldata, feature1 = "nFeature_RNA", feature2 = "percent.mito") + NoLegend()
p3 <- FeatureScatter(alldata, feature1="percent.ribo", feature2="nFeature_RNA")
p1 + p2 + p3
```



We can also subset the data to only plot one sample.

```
FeatureScatter(alldata, feature1 = "nCount_RNA", feature2 = "nFeature_RNA",
               cells = WhichCells(alldata, expression = orig.ident == "v3.1k") )
```



2.4 Filtering

2.4.1 Mitochondrial filtering

We have quite a lot of cells with a high percentage of mitochondrial reads. As indicated above, it could be wise to remove those cells, if we have enough cells left after filtering. Another option would be to either remove all mitochondrial reads from the dataset and hope that the remaining genes still have enough biological signal.

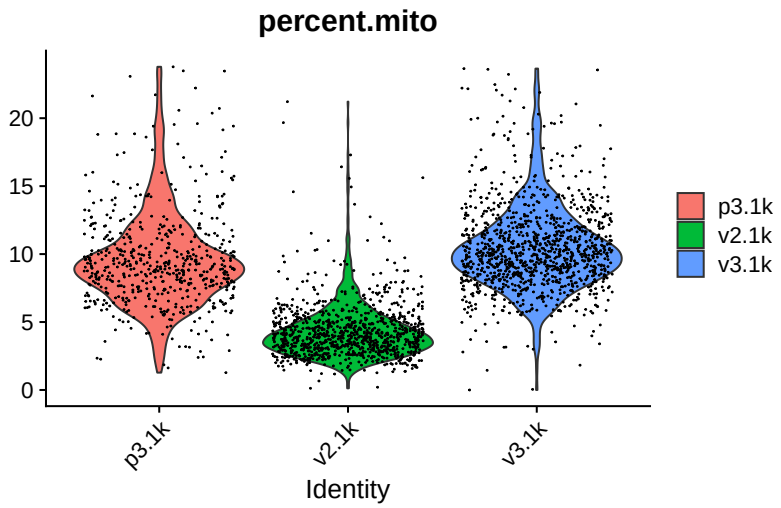
In this case we have as much as 99.7% mitochondrial reads in some of the cells, so it is quite unlikely that there is much cell type signature left in those.

By eyeballing the plots we can make reasonable decisions on where to draw the cutoff. In this case, the bulk of the cells are below 25% mitochondrial reads and that will be used as a cutoff.

```
# Select cells with percent.mito < 25
idx <- which(alldata$percent.mito < 25)
selected <- WhichCells(alldata, cells = idx)
length(selected)
```

```
# [1] 2703
```

```
# and subset the object to only keep those cells.
data.filt <- subset(alldata, cells = selected)
# plot violins for new data
VlnPlot(data.filt, features = "percent.mito")
```



As you can see, there is still quite a lot of variation in `percent.mito`, so it will have to be dealt with in later data analysis steps (not shown in this computer lab).

Question 3 What is the consequence of choosing a too stringent cutoff for the percentage of mitochondrial reads? And of choosing a too relaxed cutoff?

2.4.2 Gene detection filtering

An extremely high number of detected genes could indicate a doublet, that is, a barcode tagging multiple cells. However, depending on the cell type composition in your sample, you may have cells with higher number of genes (and also higher counts) for some cell types.

In our datasets, we observe a clear difference between the v2 vs v3 10x chemistry with regards to gene detection, so it may not be fair to apply the same cutoffs to all of them.

Also, in the p3.1k data there are a lot of cells with few detected genes giving a bimodal distribution. This type of distribution is not seen in the other two datasets. Considering that they are all PBMC datasets, it makes sense to regard this distribution as low quality libraries.

Filter the cells with high gene detection (putative doublets) with cutoffs 4100 for v3 chemistry and 2000 for v2.

```
# Start with cells with many genes detected.
high.det.v3 <- WhichCells(data.filt, expression = nFeature_RNA > 4100)
high.det.v2 <- WhichCells(data.filt, expression = nFeature_RNA > 2000 & orig.ident == "v2.1k")
# Remove these cells.
data.filt <- subset(data.filt, cells=setdiff(WhichCells(data.filt),c(high.det.v2,high.det.v3)))
# Check number of cells.
ncol(data.filt)
```

```
# [1] 2631
```

Filter the cells with low gene detection (low quality libraries) with less than 1000 genes for v2 and less than 500 for v2.

```
# Start with cells with few genes detected.
low.det.v3 <- WhichCells(data.filt, expression = nFeature_RNA < 1000 & orig.ident != "v2.1k")
low.det.v2 <- WhichCells(data.filt, expression = nFeature_RNA < 500 & orig.ident == "v2.1k")
# remove these cells
data.filt <- subset(data.filt, cells=setdiff(WhichCells(data.filt),c(low.det.v2,low.det.v3)))
```

```
# check number of cells
ncol(data.filt)
```

```
# [1] 2531
```

2.5 Plot QC statistics again

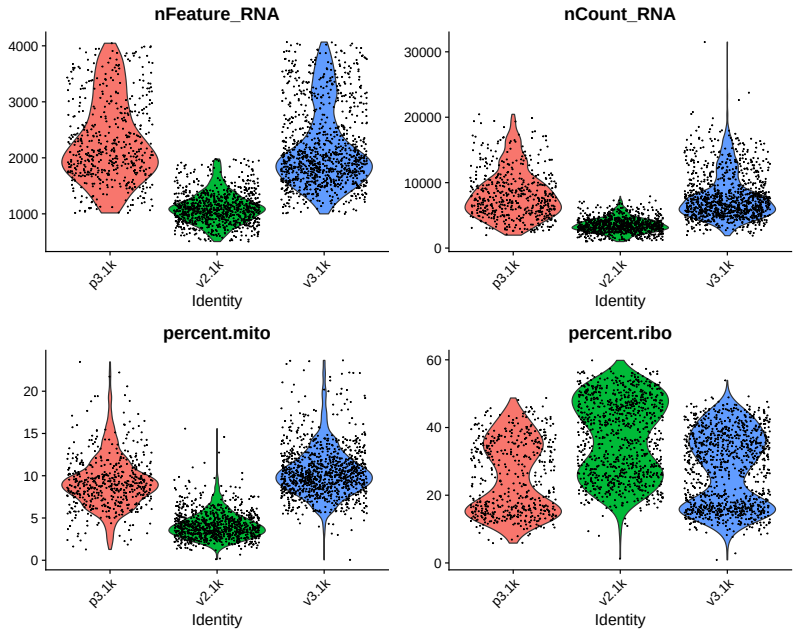
Let's plot the same QC statistics once more after filtering.

```
VlnPlot(data.filt, features = c("nFeature_RNA", "nCount_RNA", "percent.mito", "percent.ribo"),
        ncol = 2, pt.size = 0.1) + NoLegend()
# and check the number of cells per sample before and after filtering
table(Idents(alldata))
```

```
#
# p3.1k v2.1k v3.1k
#   713   996  1222
```

```
table(Idents(data.filt))
```

```
#
# p3.1k v2.1k v3.1k
#   526   933  1072
```



3 Normalization and feature selection

Each count in a count matrix represents the successful capture, reverse transcription and sequencing of a molecule of cellular mRNA. Count depths for identical cells can differ due to the variability inherent in each of these steps. Thus, when gene expression is compared between cells based on count data, any difference may have arisen solely due to sampling effects. Normalization addresses this issue by e.g. scaling count data to obtain correct relative gene expression abundances between cells.

To speed things up, we will continue working with the v3.1k dataset only.

3.1 Normalization: Log

In the default normalization method in Seurat, counts for each cell are divided by the total counts for that cell and multiplied by the scale factor 10,000. The result is then log transformed.

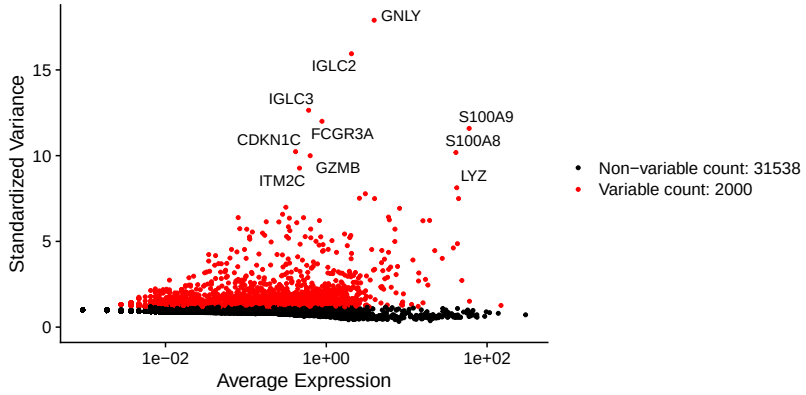
Here we first select the filtered data from just the v3.1k dataset and then normalize it.

```
pbmc.seu <- subset(x = data.filt, idents = "v3.1k")
pbmc.seu <- NormalizeData(pbmc.seu)
```

3.2 Feature selection

The default method in Seurat is variance-stabilizing transformation. A trend is fitted to to predict the variance of each gene as a function of its mean. For each gene, the variance of standardized values is computed across all cells and used to rank the features. By default, 2000 top genes are returned.

```
pbmc.seu <- FindVariableFeatures(pbmc.seu, selection.method = "vst")
top10 <- head(VariableFeatures(pbmc.seu), 10)
vplot <- VariableFeaturePlot(pbmc.seu)
LabelPoints(plot = vplot, points = top10, repel = TRUE, xnudge = 0, ynudge = 0)
```



Question 4 Can you explain why most of the genes indicated by their symbol in the figure are detected as highly variable genes?

Seurat automatically stores the feature metrics in the metadata of the assay.

```
head(pbmc.seu[["RNA"]][[]])
```

```
#                vst.mean vst.variance vst.variance.expected
# MIR1302-2HG 0.0000000000 0.0000000000          0.0000000000
# FAM138A     0.0000000000 0.0000000000          0.0000000000
# OR4F5       0.0000000000 0.0000000000          0.0000000000
# AL627309.1  0.0055970149 0.0055708851          0.0060999579
# AL627309.3  0.0009328358 0.0009328358          0.0009326186
# AL627309.2  0.0000000000 0.0000000000          0.0000000000
#             vst.variance.standardized vst.variable
# MIR1302-2HG                 0.0000000        FALSE
# FAM138A                     0.0000000        FALSE
# OR4F5                       0.0000000        FALSE
# AL627309.1                  0.9132661        FALSE
# AL627309.3                  1.0002329        FALSE
# AL627309.2                  0.0000000        FALSE
```

3.3 Saving the data

We save the Seurat object for future analysis downstream.

```
saveRDS(pbmc.seu, file = "pbmc3k_featsel.rds")
```

4 Dimensionality reduction

A human single-cell RNA-seq dataset can contain expression values for up to 25,000 genes. Many of these genes will not be informative for a given scRNA-seq dataset, and many genes will mostly contain zero counts. Even after filtering out these zero count genes in the QC step, the feature space for a single-cell dataset can have over 15,000 dimensions. To ease the computational burden on downstream analysis tools, reduce the noise in the data, and to visualize the data, one can use several approaches to reduce the dimensionality of the dataset.

In this section we will look at different ways to visualize single cell RNA-seq datasets using dimensionality reduction. We will apply Principal Component Analysis (PCA), t-distributed Stochastic Neighbor Embedding (t-SNE) and Uniform Manifold Approximation and Projection (UMAP) algorithms. Further, we will look at different ways to plot the dimensionality reduced data and augment them with additional information, such as gene expression or meta-information.

4.1 Data loading and preprocessing

We continue with the v3.1k dataset, which you have preprocessed in the previous section. We will also add cell type labels for visualization purposes. Later, you will see an example of how to annotate the cells yourself.

First, we load the Seurat object from the previous section and attach the cell type labels.

```
pbmc <- readRDS('pbmc3k_featsel.rds')
download.file(
  "https://raw.githubusercontent.com/LeidenCBC/MGC-BioSB-SingleCellAnalysis2021/main/session-dimensional
  destfile = "data/celltype_labels.tsv")
labels <- read.delim("data/celltype_labels.tsv", row.names = 1, stringsAsFactors = FALSE)
rownames(labels) <- paste0("v3.1k_", rownames(labels))
pbmc <- AddMetaData(
    object = pbmc,
    metadata = labels)
# Leave out cells without annotation
selected <- WhichCells(pbmc, cells = which(!is.na(pbmc$celltype)))
pbmc <- subset(pbmc, cells = selected)
```

Since the data has already been normalized in the previous section, we can skip these steps here. We only need to scale the data.

```
# Run the standard workflow for visualization and clustering
pbmc <- ScaleData(pbmc, verbose = FALSE)
```

4.2 Principal component analysis

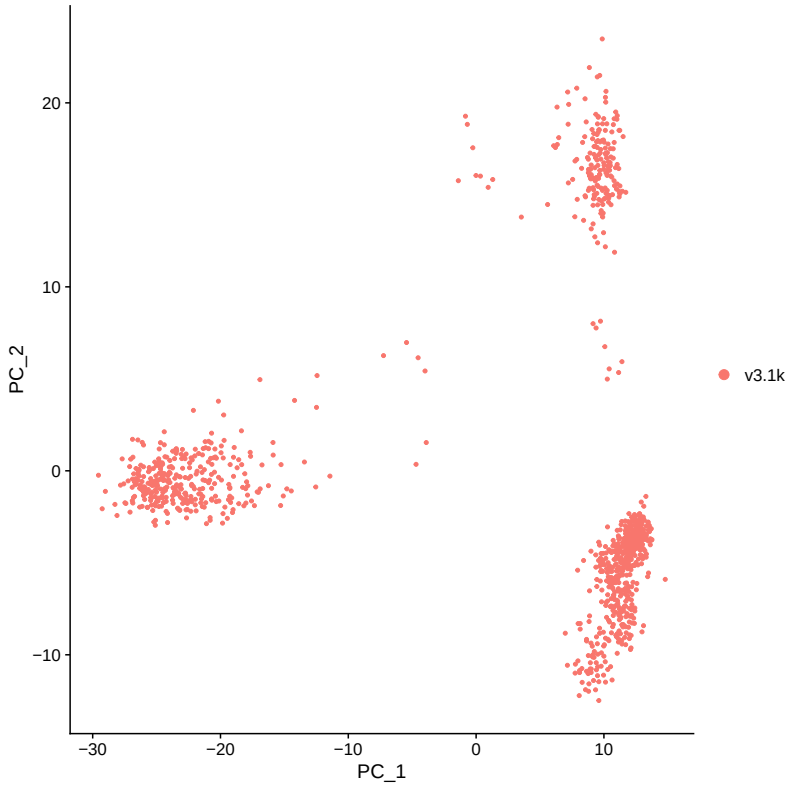
From here on, we will have a look at the different dimensionality reduction methods and their parameterizations.

We start with PCA. Seurat provides the `RunPCA` function, we use `pbmc` as input data. `npcs` refers to the number of principal components to compute. We set it to `100`. This will take a bit longer to compute but will allow us to explore the effect of using different numbers of PCs below. By assigning the result to our `pbmc` data object it will be available in the object with the default name `pca`.

```
pbmc <- RunPCA(pbmc, npcs = 100, verbose = FALSE)
```

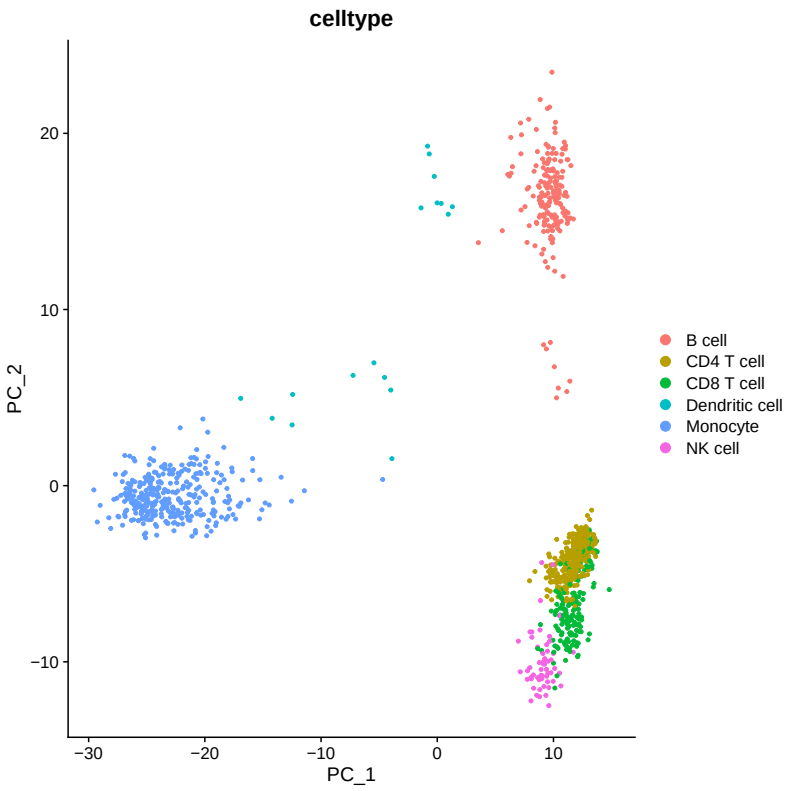
We can now plot the first two components using the `DimPlot` function. The first argument here is the Seurat data object `pbmc`. By providing the `Rreduction = "pca"` argument, `DimPlot` looks in the object for the PCA we created and assigned above.

```
DimPlot(pbmc, reduction = "pca")
```



This plot shows some structure of the data. Every dot is a cell, but we do not know which cells belong to which dot, etc. We can add meta information by the `group.by` parameter. We use the `celltype` that we have included in the metadata.

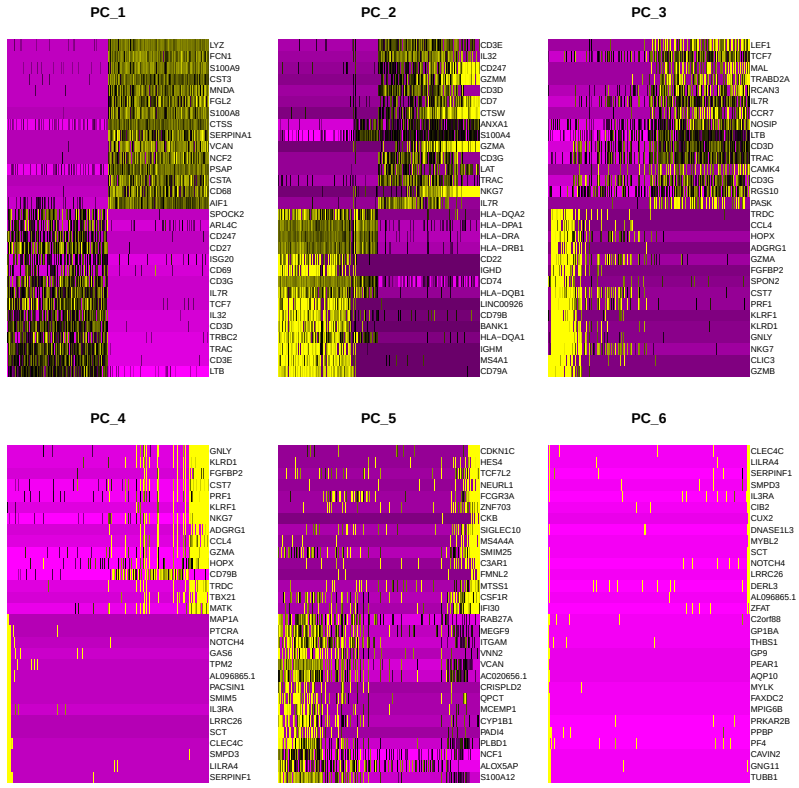
```
DimPlot(pbmc, reduction = "pca", group.by = "celltype")
```



We can see that only a few of the labeled cell types separate well but many are clumped together on the bottom of the plot.

Let's have a look at the PCs to understand a bit better how PCA separates the data. Using the `DimHeatmap` function, we can plot the expression of the top genes for each PC for a number of cells. We plot the first six components `dims = 1:6` for 500 top cells `cells = 500`. Each component produces one heatmap, the cells are the columns in the heatmap and the top genes for each component the rows.

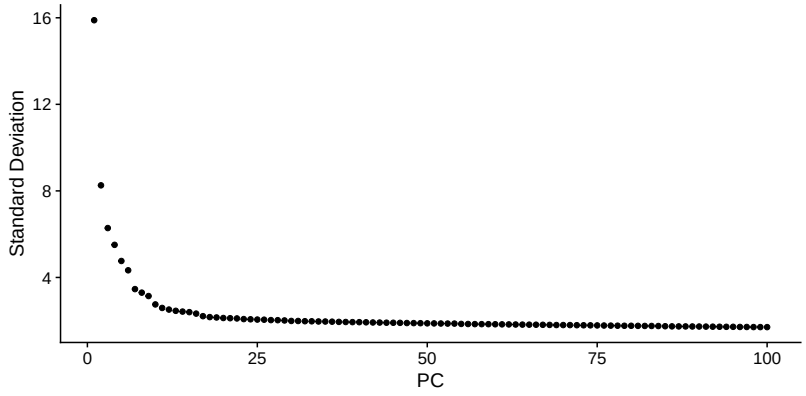
```
DimHeatmap(pbmc, dims = 1:6, cells = 500, balanced = TRUE)
```



In these heatmaps it is easy to see thatt the first few PCs have clear-cut expression differences for the genes most affecting the principal components. The distinction becomes less clear for more distant principal components.

As discussed in the lecture, and indicated above, PCA is not optimal for visualization, but can be very helpful in reducing the complexity before applying non-linear dimensionality reduction methods. For that, let's have a look how many PCs actually cover the main variation. A very simple, fast-to-compute way is simply looking at the standard deviation per PC. We use the `ElbowPlot`.

```
ElbowPlot(pbmc, ndims = 100)
```



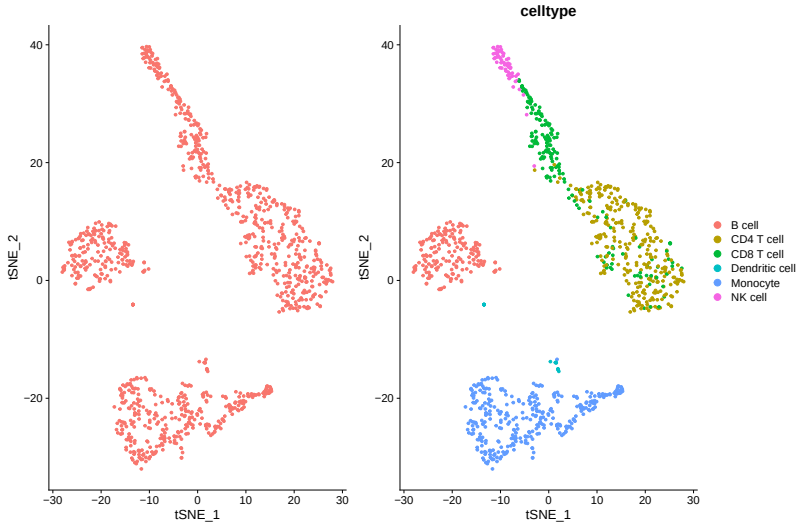
As we can see there is a very steep drop in standard deviation within the first 20 or so PCs indicating that we will likely be able to use roughly that number of PCs as input to following computations with little impact on the results.

4.3 t-SNE

Let's try out t-SNE. Seurat by default uses the Barnes Hut (BH) SNE implementation.

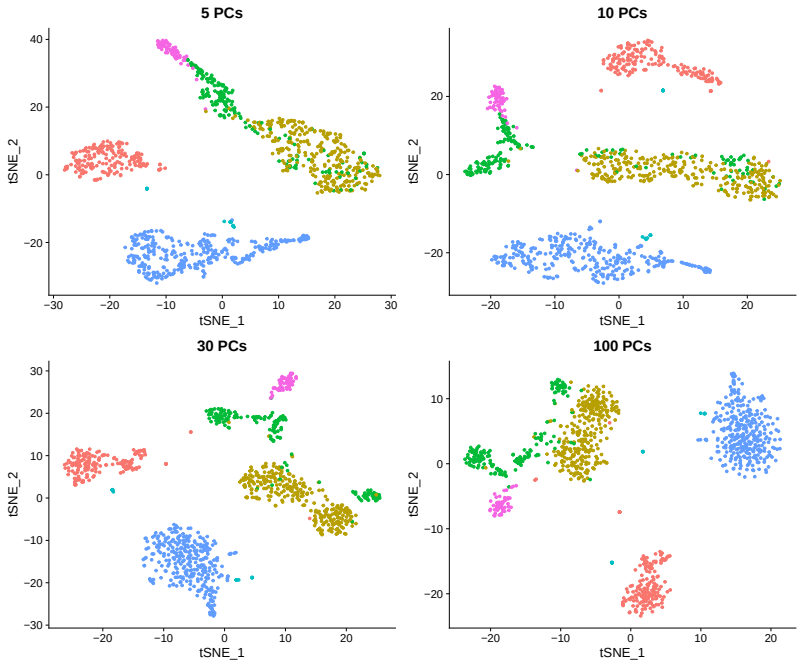
Similar to the PCA, Seurat provides a convenient function to run t-SNE called `RunTSNE`. We provide the `pbmc` data object as parameter. By default `RunTSNE` will look for and use the PCA we created above as input, we can also force it with `reduction = "pca"`. Again we use `DimPlot` to plot the result, this time using `reduction = "tsne"` to indicate that we want to plot the t-SNE computation. We create two plots, the first without and the second with the cell types used for grouping. Already without the coloring, we can see much more structure in the plot than in the PCA plot. With the color overlay we see that most cell types are nicely separated in the plot.

```
pbmc <- RunTSNE(pbmc, reduction = "pca")
p1 <- DimPlot(pbmc, reduction = "tsne") + NoLegend()
p2 <- DimPlot(pbmc, reduction = "tsne", group.by = "celltype")
p1 + p2
```



Above we did not specify the number of PCs to use as input. Let's have a look what happens with different numbers of PCs as input. We simply run `RunTSNE` multiple times with `dims` defining a range of PCs. *Note* every run overwrites the `tsne` object nested in the `pbmc` object. Therefore we plot the tsne directly after each run and store all plots in a object. We add `+ NoLegend() + ggtitle("n PCs")` to remove the list of cell types for compactness and add a title.

```
# PC_1 to PC_5
pbmc <- RunTSNE(pbmc, reduction = "pca", dims = 1:5)
p1 <- DimPlot(pbmc, reduction = "tsne", group.by = "celltype") + NoLegend() + ggtitle("5 PCs")
# PC_1 to PC_10
pbmc <- RunTSNE(pbmc, reduction = "pca", dims = 1:10)
p2 <- DimPlot(pbmc, reduction = "tsne", group.by = "celltype") + NoLegend() + ggtitle("10 PCs")
# PC_1 to PC_30
pbmc <- RunTSNE(pbmc, reduction = "pca", dims = 1:30)
p3 <- DimPlot(pbmc, reduction = "tsne", group.by = "celltype") + NoLegend() + ggtitle("30 PCs")
# PC_1 to PC_100
pbmc <- RunTSNE(pbmc, reduction = "pca", dims = 1:100)
p4 <- DimPlot(pbmc, reduction = "tsne", group.by = "celltype") + NoLegend() + ggtitle("100 PCs")
p1 + p2 + p3 + p4
```



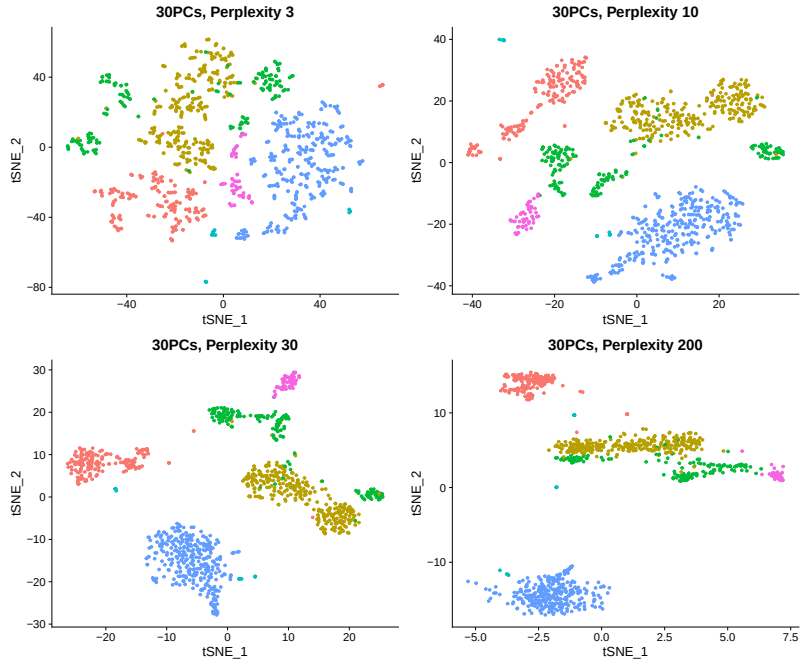
Looking at these plots, it seems `RunTSNE` by default only uses 5 PCs (the plot is identical to the plot with default parameters above), but we get much clearer separation of clusters using 10 or even 30 PCs. This is not surprising considering the plot above of the standard deviation per PC above. Therefore we will use 30 PCs in the following, by explicitly setting `dims = 1:30`. *Note* that t-SNE is slower with more input dimensions (here the PCs), so it is good to find a middle ground between capturing as much variation as possible with as few PCs as possible. However, using the default 5 clearly does not produce optimal results for this dataset. When using t-SNE with PCA preprocessing with your own data, always check how many PCs you need to cover the variance, as done above. t-SNE has a few hyper-parameters that can be tuned for better visualization. There is an excellent tutorial. We will now investigate the influence of two of these hyperparameters.

4.3.1 [OPTIONAL] t-SNE: perplexity

The main parameter is the perplexity, basically indicating how many neighbors to look at. We will run different perplexities to see the effect. As we will see, a perplexity of 30 is the default. This value often works well, again it might be advisable to test different values with other data. *Note*, higher perplexity values make t-SNE slower to compute.

```
# Perplexity 3
pbmc <- RunTSNE(pbmc, reduction = "pca", dims = 1:30, perplexity = 3)
p1 <- DimPlot(pbmc, reduction = "tsne", group.by = "celltype") + NoLegend() +
  ggtitle("30PCs, Perplexity 3")
# Perplexity 10
pbmc <- RunTSNE(pbmc, reduction = "pca", dims = 1:30, perplexity = 10)
p2 <- DimPlot(pbmc, reduction = "tsne", group.by = "celltype") + NoLegend() +
  ggtitle("30PCs, Perplexity 10")
# Perplexity 30
pbmc <- RunTSNE(pbmc, reduction = "pca", dims = 1:30, perplexity = 30)
p3 <- DimPlot(pbmc, reduction = "tsne", group.by = "celltype") + NoLegend() +
  ggtitle("30PCs, Perplexity 30")
# Perplexity 200
pbmc <- RunTSNE(pbmc, reduction = "pca", dims = 1:30, perplexity = 200)
p4 <- DimPlot(pbmc, reduction = "tsne", group.by = "celltype") + NoLegend() +
  ggtitle("30PCs, Perplexity 200")
```

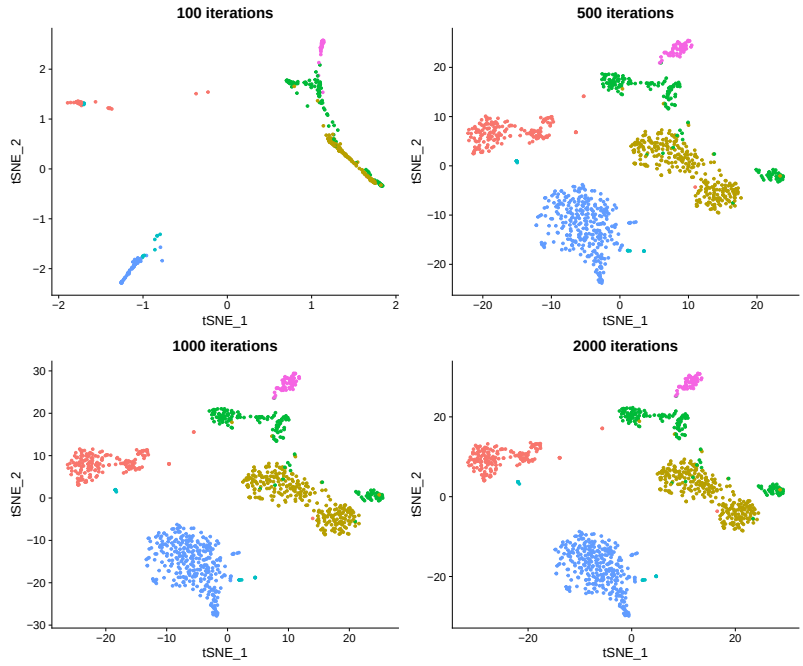
```
p1 + p2 + p3 + p4
```



4.3.2 [OPTIONAL] t-SNE: number of iterations

Another important parameter is the number of iterations. t-SNE gradually optimizes the embedding in low-dimensional space. The more iterations the more there is time to optimize. We will run different numbers of iterations to see the effect. As we will see, 1000 iterations is the default. This value often works well, again it might be advisable to test different values with other data. Especially for larger datasets you will need more iterations. *Note*, more iterations make t-SNE slower to compute.

```
# 100 iterations
pbmc <- RunTSNE(pbmc, reduction = "pca", dims = 1:30, max_iter = 100)
p1 <- DimPlot(pbmc, reduction = "tsne", group.by = "celltype") + NoLegend() +
  ggtitle("100 iterations")
# 500 iterations
pbmc <- RunTSNE(pbmc, reduction = "pca", dims = 1:30, max_iter = 500)
p2 <- DimPlot(pbmc, reduction = "tsne", group.by = "celltype") + NoLegend() +
  ggtitle("500 iterations")
# 1000 iterations
pbmc <- RunTSNE(pbmc, reduction = "pca", dims = 1:30, max_iter = 1000)
p3 <- DimPlot(pbmc, reduction = "tsne", group.by = "celltype") + NoLegend() +
  ggtitle("1000 iterations")
# 2000 iterations
pbmc <- RunTSNE(pbmc, reduction = "pca", dims = 1:30, max_iter = 2000)
p4 <- DimPlot(pbmc, reduction = "tsne", group.by = "celltype") + NoLegend() +
  ggtitle("2000 iterations")
p1 + p2 + p3 + p4
```



As we see after 100 iterations the main structure becomes apparent, but there is very little detail. 500 to 2000 iterations all look very similar, with 500 still a bit more loose than 1000, indicating that the optimization converges somewhere between 500 and 1000 iterations. In this case running 2000 would definitely not be necessary.

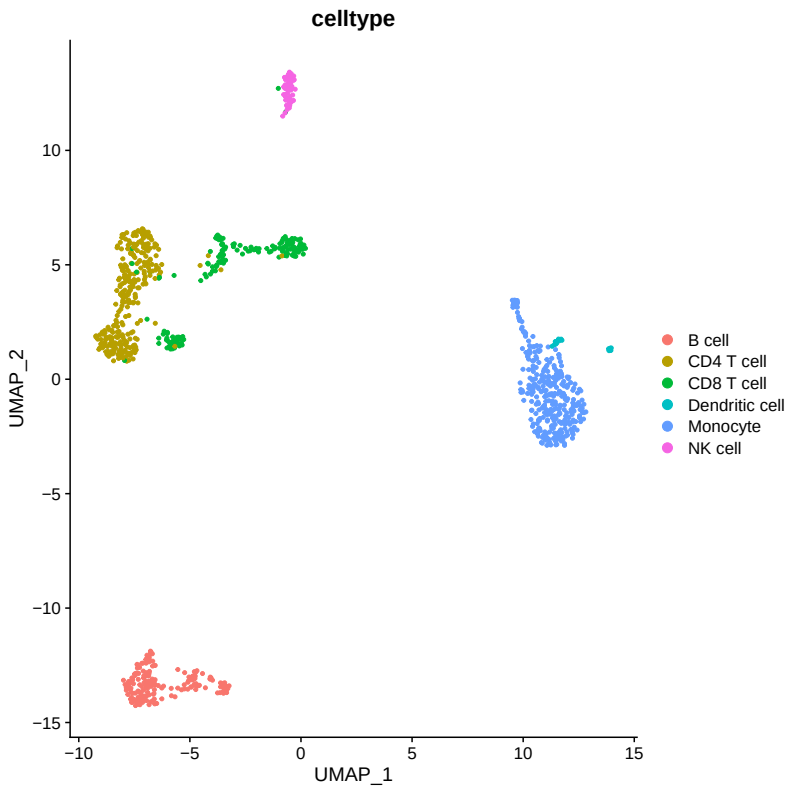
4.4 UMAP

We have seen t-SNE and its main parameters. Let's have a look at UMAP. Its main function call is very similar to t-SNE and PCA and called `RunUMAP`. Again, by default it looks for the PCA in the `pbmc` data object, but we have to provide it with the number of PCs (or `dims`) to use. Here, we use `30`. As expected, the plot looks rather similar to the t-SNE plot, with more compact clusters.

```
pbmc <- RunUMAP(pbmc, dims = 1:30, verbose = FALSE)
```

```
# 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
```

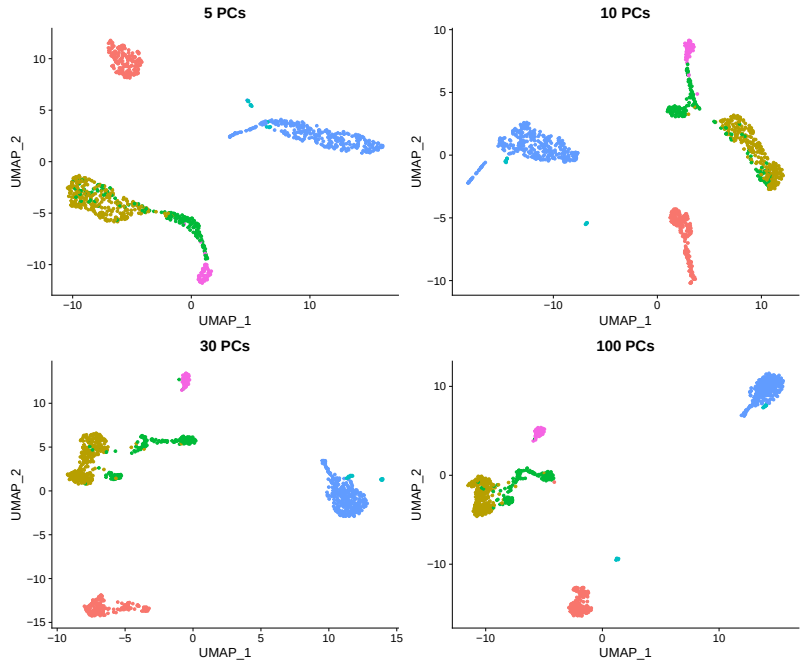
```
DimPlot(pbmc, reduction = "umap", group.by = "celltype")
```



Just like t-SNE, UMAP has a bunch of parameters. In fact, Seurat exposes quite a few more than for t-SNE. We will look at the most important in the following. An interactive tutorial can be found here, and here is a comparison with the same datasets between UMAP and t-SNE.

Again, we start with varying the number of PCs. Similar to t-SNE, 5 is clearly not enough, 30 provides decent separation and detail. Just like for t-SNE, test this parameter to match your own data in real-world experiments.

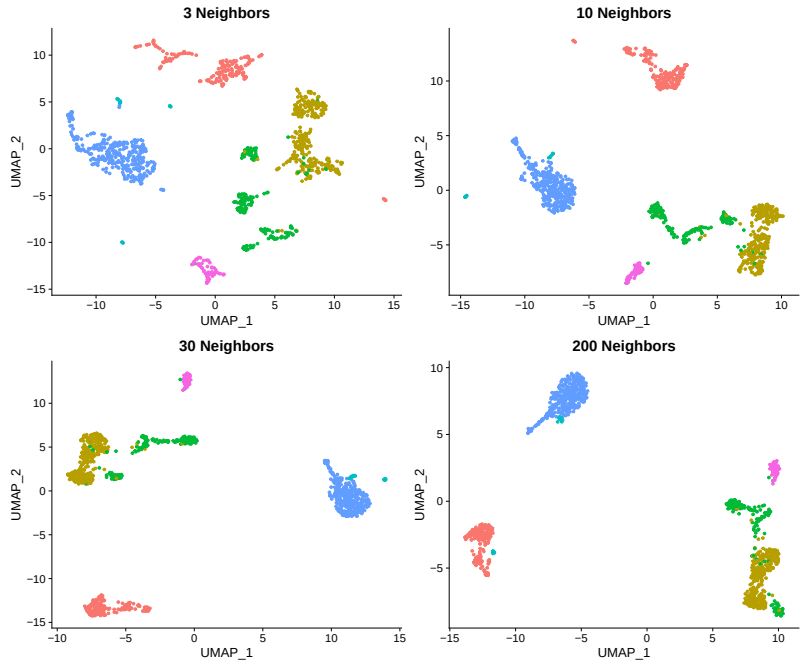
```
# PC_1 to PC_5
pbmc <- RunUMAP(pbmc, dims = 1:5, verbose = FALSE)
p1 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() + ggtitle("5 PCs")
# PC_1 to PC_10
pbmc <- RunUMAP(pbmc, dims = 1:10, verbose = FALSE)
p2 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() + ggtitle("10 PCs")
# PC_1 to PC_30
pbmc <- RunUMAP(pbmc, dims = 1:30, verbose = FALSE)
p3 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() + ggtitle("30 PCs")
# PC_1 to PC_100
pbmc <- RunUMAP(pbmc, dims = 1:100, verbose = FALSE)
p4 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() + ggtitle("100 PCs")
p1 + p2 + p3 + p4
```



4.4.1 [OPTIONAL] UMAP: number of neighbours

The `n.neighbors` parameter sets the number of neighbors to consider for UMAP. Larger values will result in more global structure being preserved at the loss of detailed local structure. This parameter is similar to the perplexity in t-SNE. We try a similar range of values for comparison. The results are quite similar to t-SNE. With low values, clearly the structures are too spread out, but quickly the embeddings become quite stable. The default vaule for RunUMAP is 30. Similar to t-SNE this value generally gives reasonable results. Again, it's always a good idea to run some tests with new data to find a good value.

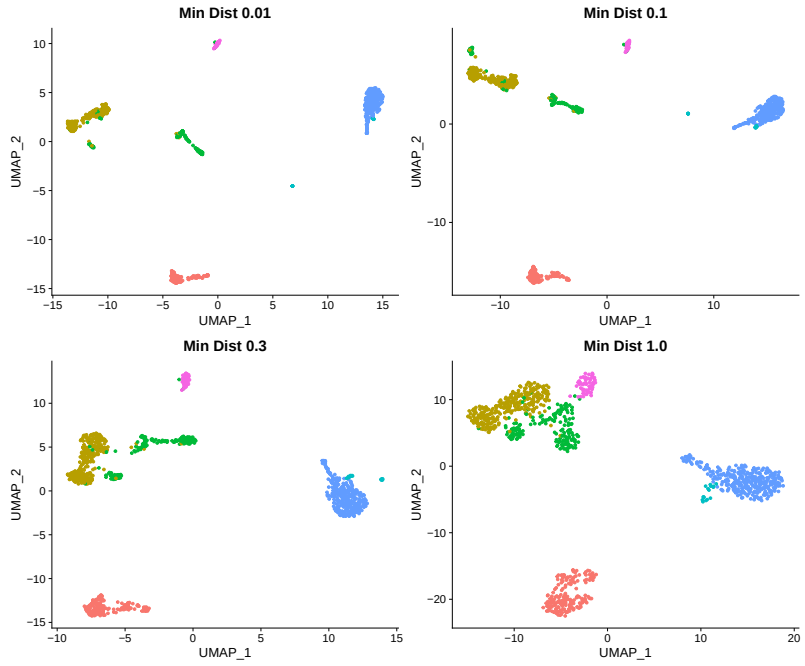
```
# 3 Neighbors
pbmc <- RunUMAP(pbmc, dims = 1:30, n.neighbors = 3, verbose = FALSE)
p1 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() +
  ggtitle("3 Neighbors")
# 10 Neighbors
pbmc <- RunUMAP(pbmc, dims = 1:30, n.neighbors = 10, verbose = FALSE)
p2 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() +
  ggtitle("10 Neighbors")
# 30 Neighbors
pbmc <- RunUMAP(pbmc, dims = 1:30, n.neighbors = 30, verbose = FALSE)
p3 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() +
  ggtitle("30 Neighbors")
# 200 Neighbors
pbmc <- RunUMAP(pbmc, dims = 1:30, n.neighbors = 200, verbose = FALSE)
p4 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() +
  ggtitle("200 Neighbors")
p1 + p2 + p3 + p4
```



4.4.2 [OPTIONAL] UMAP: min.dist

Another parameter is `min.dist`, which defines the compactness of the final embedding. Larger values ensure embedded points are more evenly distributed, while smaller values allow the algorithm to optimise more accurately with regard to local structure. The default value is 0.3. There is no directly comparable parameter in t-SNE.

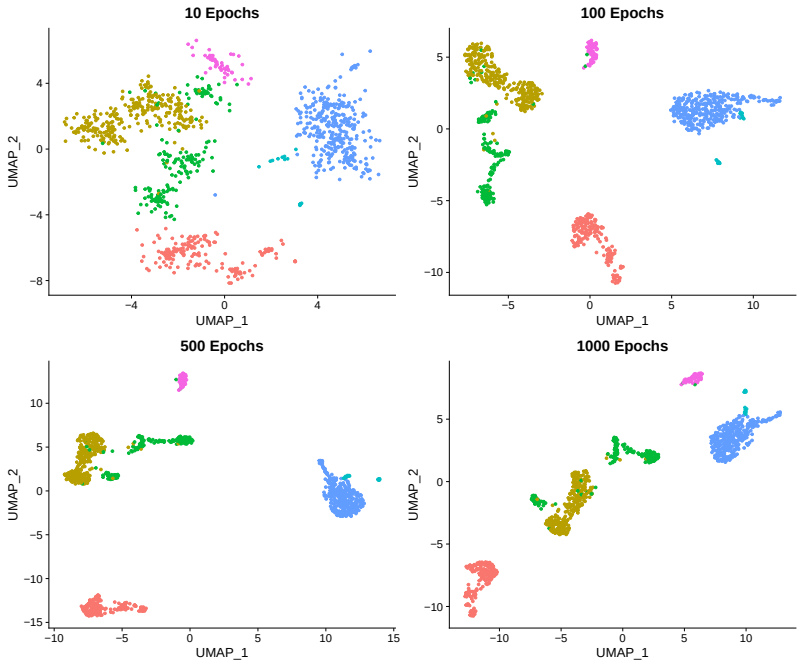
```
# Min Distance 0.01
pbmc <- RunUMAP(pbmc, dims = 1:30, min.dist = 0.01, verbose = FALSE)
p1 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() +
  ggtitle("Min Dist 0.01")
# Min Distance 0.1
pbmc <- RunUMAP(pbmc, dims = 1:30, min.dist = 0.1, verbose = FALSE)
p2 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() +
  ggtitle("Min Dist 0.1")
# Min Distance 0.3
pbmc <- RunUMAP(pbmc, dims = 1:30, min.dist = 0.3, verbose = FALSE)
p3 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() +
  ggtitle("Min Dist 0.3")
# Min Distance 1.0
pbmc <- RunUMAP(pbmc, dims = 1:30, min.dist = 1.0, verbose = FALSE)
p4 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() +
  ggtitle("Min Dist 1.0")
p1 + p2 + p3 + p4
```



4.4.3 [OPTIONAL] UMAP: number of epochs

`n.epochs` is comparable to the number of iterations in t-SNE. Typically, UMAP needs fewer of iterations to converge than t-SNE, but also changes more when it is run longer. The default is 500. Again, this should be adjusted to your data.

```
# 10 epochs
pbmc <- RunUMAP(pbmc, dims = 1:30, n.epochs = 10, verbose = FALSE)
p1 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() +
  ggtitle("10 Epochs")
# 100 epochs
pbmc <- RunUMAP(pbmc, dims = 1:30, n.epochs = 100, verbose = FALSE)
p2 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() +
  ggtitle("100 Epochs")
# 500 epochs
pbmc <- RunUMAP(pbmc, dims = 1:30, n.epochs = 500, verbose = FALSE)
p3 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() +
  ggtitle("500 Epochs")
# 1000 epochs
pbmc <- RunUMAP(pbmc, dims = 1:30, n.epochs = 1000, verbose = FALSE)
p4 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() +
  ggtitle("1000 Epochs")
p1 + p2 + p3 + p4
```

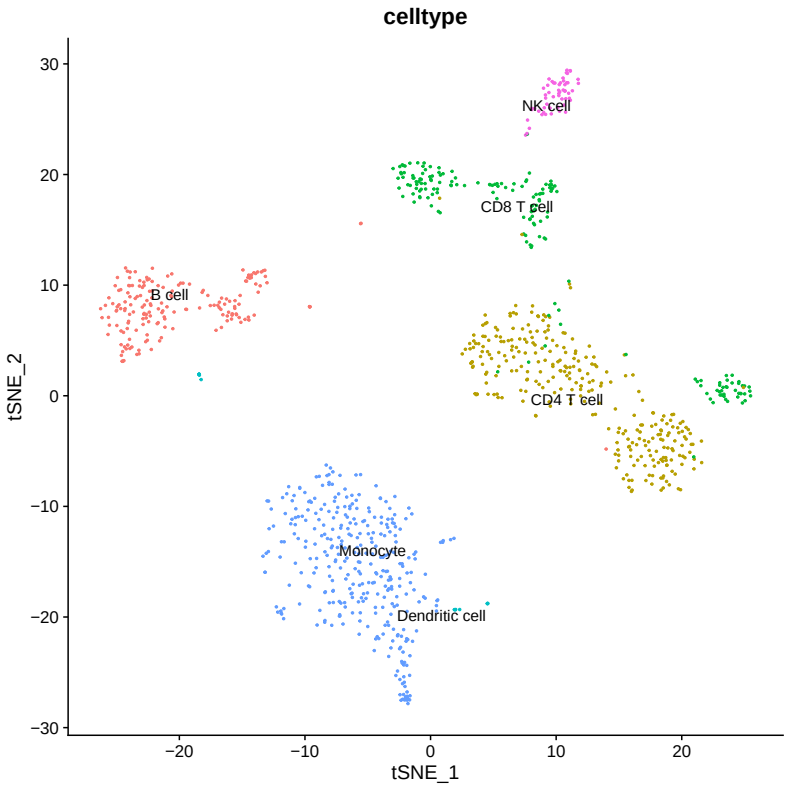


See this presentation of Dmitry KObak for all detailed ins and outs on similarities and differences between t-SNE and UMAP.

4.5 Visualization

Finally, let's have a brief look at some visualization options. We have already used color for grouping. With `label = TRUE` we can add a text-label to each group and with `repel = TRUE` we can make sure those labels don't clump together. Finally, `pt.size = 0.5` changes the size of the dots used in the plot.

```
# Re-run a t-SNE so we do not rely on changes above
pbmc <- RunTSNE(pbmc, dims = 1:30)
DimPlot(pbmc, reduction = "tsne", group.by = "celltype", label = TRUE, repel = TRUE,
        pt.size = 0.5) + NoLegend()
```



Another property we might want to look at in our dimensionality reduction plot is the expression of individual genes. We can overlay gene expression as color using the `FeaturePlot` function. Here, we first find the *top two* features correlated to the *first and second PCs* and combine them into a single vector which will be the parameter for the `FeaturePlot`.

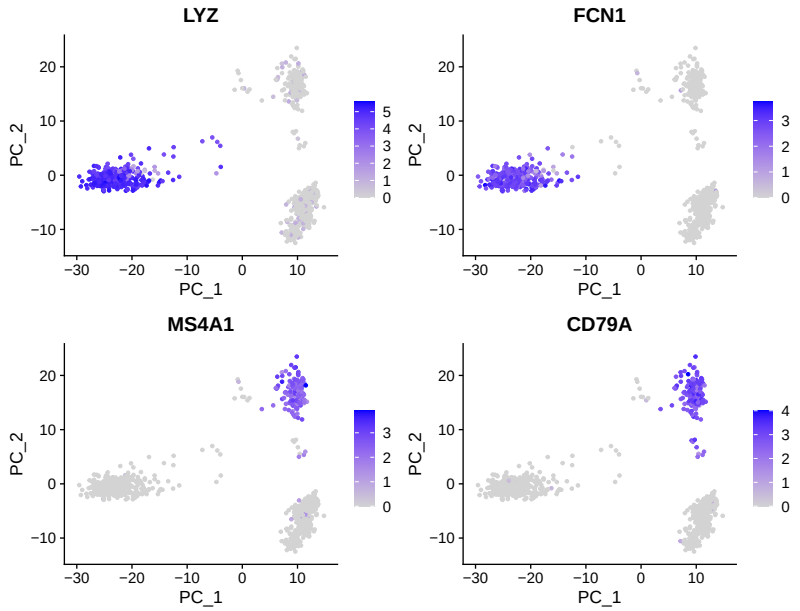
Finally, we call `FeaturePlot` with the `pbmc` data object, `features = topFeaturesPC` uses the extracted feature vector to create one plot for each feature in the list and lastly, `reduction = "pca"` will create PCA plots.

Not surprisingly, the top two features of the first PC form a smooth gradient on the PC_1 axis and the top two features of the second PC a smooth gradient on the PC_2 axis.

```
# find top genes for PCs 1 and 2
topFeaturesPC1 <- TopFeatures(object = pbmc[["pca"]], nfeatures = 2, dim = 1)
topFeaturesPC2 <- TopFeatures(object = pbmc[["pca"]], nfeatures = 2, dim = 2)
# combine the genes into a single vector
topFeaturesPC <- c(topFeaturesPC1, topFeaturesPC2)
print(topFeaturesPC)
```

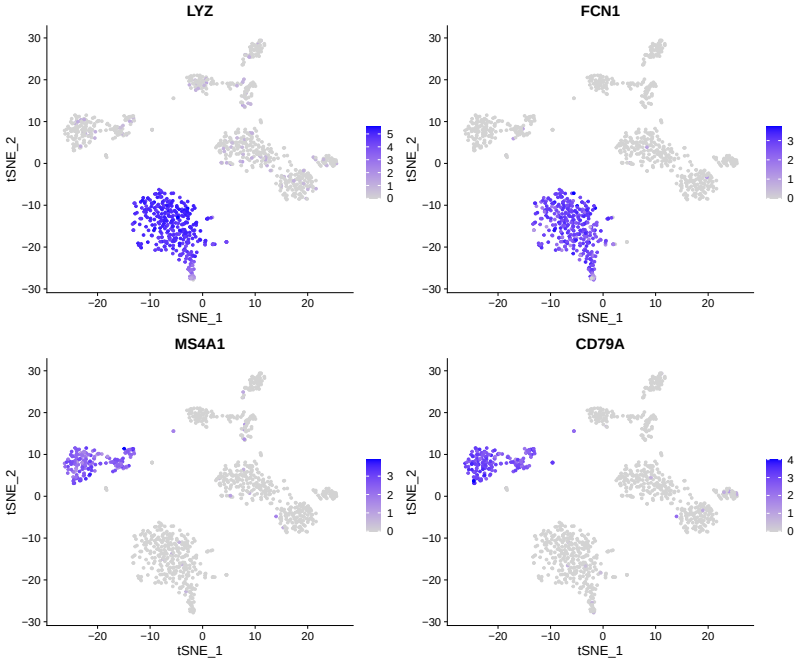
```
# [1] "LYZ"   "FCN1"  "MS4A1" "CD79A"
```

```
# feature plot with the defined genes
FeaturePlot(pbmc, features = topFeaturesPC, reduction = "pca")
```



Let's have a look at the same features on a t-SNE plot. The behavior here is quite different, with the high expression being very localized to specific clusters in the maps.

```
FeaturePlot(pbmc, features = topFeaturesPC, reduction = "tsne")
```

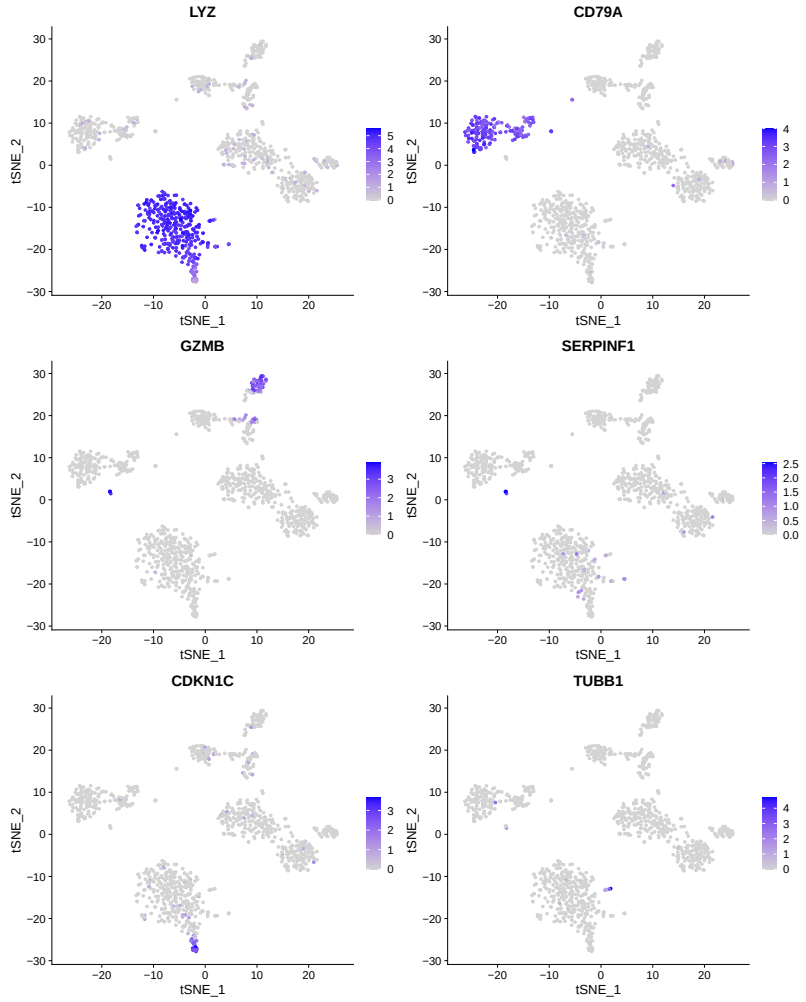


It is clear that the top PCs are fundamental to forming the clusters, so let's have a look at more PCs and pick the top gene per PC for a few more PCs.

```
for(i in 1:6) {
  topFeaturesPC[[i]] <- TopFeatures(object = pbmc[["pca"]], nfeatures = 1, dim = i)
}
print(topFeaturesPC)
```

```
# [1] "LYZ"      "CD79A"    "GZMB"     "SERPINF1" "CDKN1C"   "TUBB1"
```

```
FeaturePlot(pbmc, features = topFeaturesPC, reduction = "tsne")
```



Finally, let's create a more interactive plot. First we create a regular `FeaturePlot`, here with just one gene `features = "CD3D"`, a marker of T cells. Instead of plotting it directly we save the plot in the `interactivePlot` variable.

With `HoverLocator` we can then embed this plot into an interactive version. The `information = FetchData(pbmc, vars = c("celltype", topFeaturesPC))` creates a set of properties that will be shown on hover over each point. Im this case, we show the cell type from the meta information.

The result is a plot that allows us to inspect single cells in detail.

```
interactivePlot <- FeaturePlot(pbmc, reduction = "tsne", features = "CD3D")
HoverLocator(plot = interactivePlot,
             information = FetchData(pbmc, vars = c("celltype", topFeaturesPC)))
```

4.6 Saving the data

Save the Seurat object with the new embeddings for future use downstream.

```
saveRDS(pbmc, file = "pbmc3k_emb.rds")
```

If you're interested in more ways to visualize your data, this vignette might be useful.

5 Clustering

In this section we will look at different approaches to cluster scRNA-seq datasets in order to characterize the different subgroups of cells. Using unsupervised clustering, we will try to identify groups of cells based on the similarities of their transcriptomes without any prior knowledge of the labels.

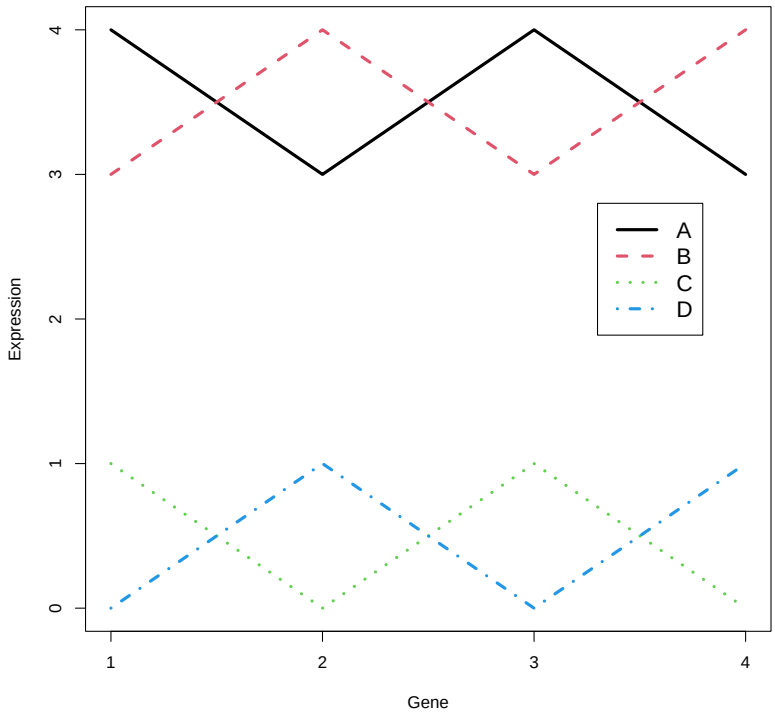
5.1 Distances and hierarchical clustering of a toy example

Cluster algorithms group similar data together. What is meant by the word *similar* is formally defined by the notion of a *distance*. We will first have a look at two commonly used distance measures in more detail.

For this purpose we will use a synthetic dataset corresponding to a mini-experiment with just four cells (A,B,C,D) and four genes per cell given in file hcexample.txt. Download this file and save it in the same folder `data` as the single-cell experiment data.

The following piece of R code reads in `hcexample.txt` and then plots the cell profiles:

```
E <- read.table("data/hcexample.txt")
matplot(E,type="l",col=1:4,lty=1:4,lwd=3,xlab="Gene",ylab="Expression",xaxt="n")
axis(1,1:4)
legend(3.3,2.8,c("A","B","C","D"),lty=1:4,col=1:4,lwd=3,y.intersp=1,cex=1.3)
```



Calculate the Euclidean (`euc`) and the Pearson sample correlation (`cor.dist`) distance between the cell profiles. These are functions from the **bioDist** package that we loaded at the beginning of the computer lab.

```
# Note that you have to transpose (t) the data matrix E since pairwise distances
# are calculated for rows of a matrix
```

```
d.euc <- euc(t(E))
d.euc
```

```
#          A        B        C
# B 2.000000
# C 6.000000 6.324555
# D 6.324555 6.000000 2.000000
```

```
d.cor <- cor.dist(t(E),abs=FALSE)
d.cor
```

```
#   A B C
# B 2
# C 0 2
# D 2 0 2
```

As always you can obtain a detailed explanation of a function by typing `?` followed by the name of the function in the Console window, for example `?euc` or `?cor.dist`.

Question 5 Can you explain the resulting distance matrix when using the Pearson correlation distance?

Such a distance or *dissimilarity* matrix forms the basis for most clustering algorithms. In omics literature, an agglomerative (*i.e.*, bottum-up) hierarchical approach such as implemented in the `hclust` function is popular. As explained in the lecture, hierarchical clustering tries to find a tree-like representation of the data in which clusters have subclusters, which have subclusters, and so on. The number of clusters depends on in how much detail one looks at the tree. Hierarchical clustering uses two types of distances:

- Distance between two individual data points (Euclidean, correlation etc.)
- Distance between two clusters of data points, also called *linkage* (single, average, etc.).

Question 6 Draw (just with pencil on paper) the dendrograms for both the Euclidean and the correlation distance matrix generated above. Explain your results.

One property of most clustering algorithms is that they always produce clusters. This happens regardless of whether there is actually any meaningful clustering structure present in the data. Let us now simulate some unstructured data (`rnorm` randomly generates data from a normal distribution) and see what happens.

```
# 1000 genes
n.genes <- 1000
# 50 samples
n.samples <- 50
# Generate labels for the cells
descr <- paste("S", rep(c("0",""),times=c(9,41)), 1:50, sep="")
# Fix the random seed to make the exercise reproducible
set.seed(13)
# Matrix of expression data for 1000 genes and 50 cells
dataMatrix <- matrix(rnorm(n.genes*n.samples),nrow=n.genes)
```

Question 7 Use `hclust` with Euclidean distance and average, single, and complete linkage to cluster the samples and plot the resulting dendrograms. First have a look at ?hclust and especially the examples at the bottom of the help page to see how to use this function. If this goes beyond your current R skills, have a look at the answers and just run the code given there.

Question 8 Which phenomenon do you see with single linkage? How might this make interpretation difficult?

Question 9 The complete linkage dendrogram seems to show some structure in the data and you might decide that four clusters can be discerned. Is this structure real?

5.2 Hierarchical clustering of scRNA-seq data

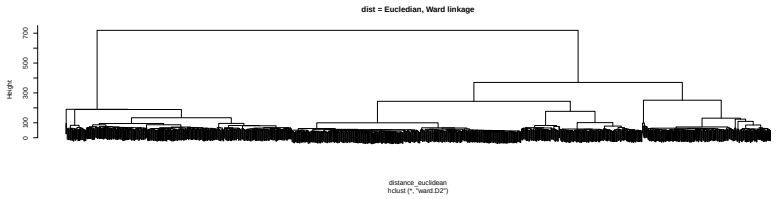
Now, we will continue with the v3.1k dataset that you have preprocessed and visualized in the previous sections. Let's start by loading the data again.

In one of the previous sections, we have already selected highly variable genes. This step is also to decide which genes to use when clustering the cells. Single cell RNA-seq can profile a huge number of genes in a lot of cells. But most of the genes are not expressed enough to provide a meaningful signal and are often driven by technical noise. Including them could potentially add some unwanted signal that would blur the biological variation. Moreover gene filtering can also speed up the computational time for downstream analysis.

```
pbmc <- readRDS('pbmc3k_emb.rds')
```

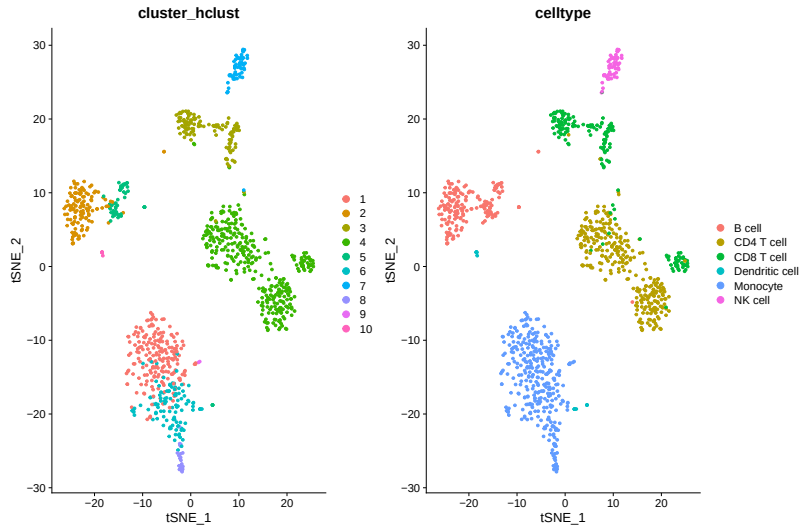
We start by performing hierarchical clustering with Euclidean distance and yet another linkage method, called Ward linkage.

```
# Get scaled counts from the Seurat object
scaled_pbmc <- pbmc@assays$RNA@scale.data
# Calculate Distances (default: Euclidean distance)
distance_euclidean <- dist(t(scaled_pbmc))
# Perform hierarchical clustering using Ward linkage
ward_hclust_euclidean <- hclust(distance_euclidean,method = "ward.D2")
plot(ward_hclust_euclidean, main = "dist = Eucledian, Ward linkage", labels=FALSE)
```



Now cut the dendrogram to generate 10 clusters and plot the cluster labels and the previously given `celltype` labels on the t-SNE plot. For now, we just pick 10, but you can of course vary this number to see how it influences your results.

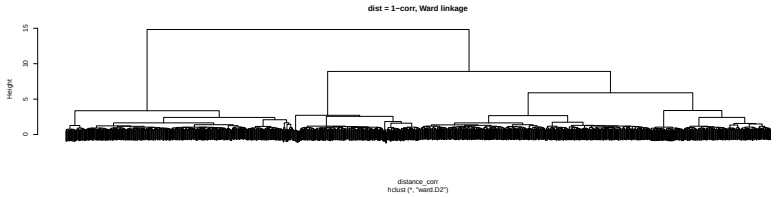
```
# Cutting the cluster tree to make 10 groups
cluster_hclust <- cutree(ward_hclust_euclidean,k = 10)
pbmc@meta.data$cluster_hclust <- factor(cluster_hclust)
p1 <- DimPlot(pbmc, reduction="tsne", group.by = "cluster_hclust")
p2 <- DimPlot(pbmc, reduction="tsne", group.by = "celltype")
p1+p2
```



Now let's try a different distance measure. A commonly used distance measure is 1 - correlation.

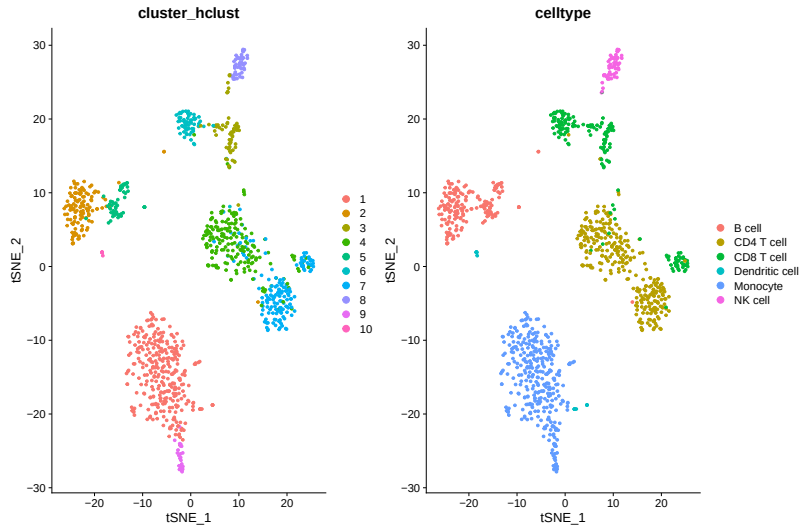
```
# Calculate Distances (1 - correlation)
C <- cor(scaled_pbmc)
# Run clustering based on the correlations, where the distance will
# be 1-correlation, e.g. higher distance with lower correlation.
distance_corr <- as.dist(1-C)

#Perform hierarchical clustering using ward linkage
ward_hclust_corr <- hclust(distance_corr,method="ward.D2")
plot(ward_hclust_corr, main = "dist = 1-corr, Ward linkage", labels=FALSE)
```



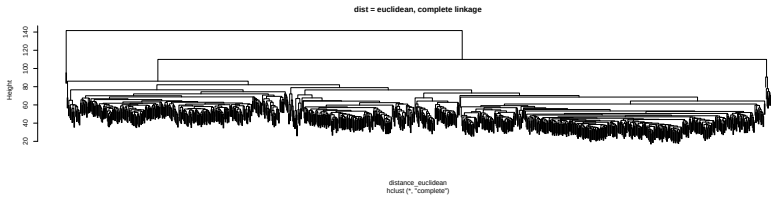
Again, let's cut the dendrogram to generate 10 clusters and plot the cluster labels on the t-SNE plot.

```
#Cutting the cluster tree to make 10 groups
cluster_hclust <- cutree(ward_hclust_corr,k = 10)
pbmc@meta.data$cluster_hclust <- factor(cluster_hclust)
p1 <- DimPlot(pbmc, reduction="tsne", group.by = "cluster_hclust")
p2 <- DimPlot(pbmc, reduction="tsne", group.by = "celltype")
p1+p2
```



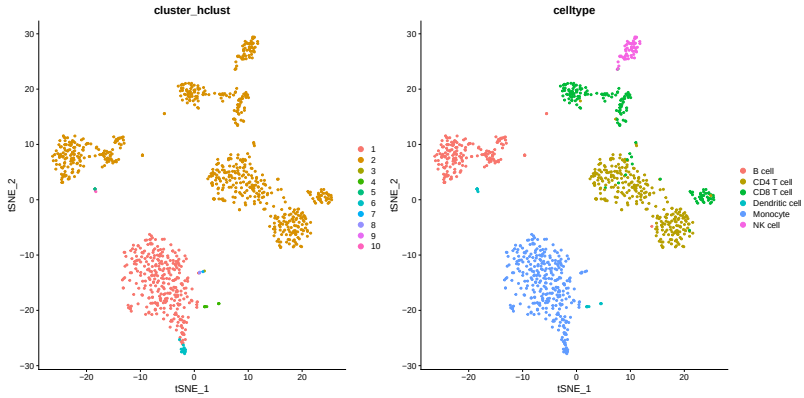
Instead of changing the distance metric, we can change the linkage method. Instead of using Ward's method, let's use complete linkage.

```
#Perform hierarchical clustering using complete linkage and Euclidean distance
comp_hclust_eucledian <- hclust(distance_euclidean,method = "complete")
plot(comp_hclust_eucledian, main = "dist = euclidean, complete linkage", labels=FALSE)
```



Once more, let's cut the dendrogram to generate 10 clusters and plot the cluster labels on the t-SNE plot.

```
#Cutting the cluster tree to make 10 groups
cluster_hclust <- cutree(comp_hclust_eucledian,k = 10)
pbmc@meta.data$cluster_hclust <- factor(cluster_hclust)
p1 <- DimPlot(pbmc, reduction="tsne", group.by = "cluster_hclust")
p2 <- DimPlot(pbmc, reduction="tsne", group.by = "celltype")
p1+p2
```



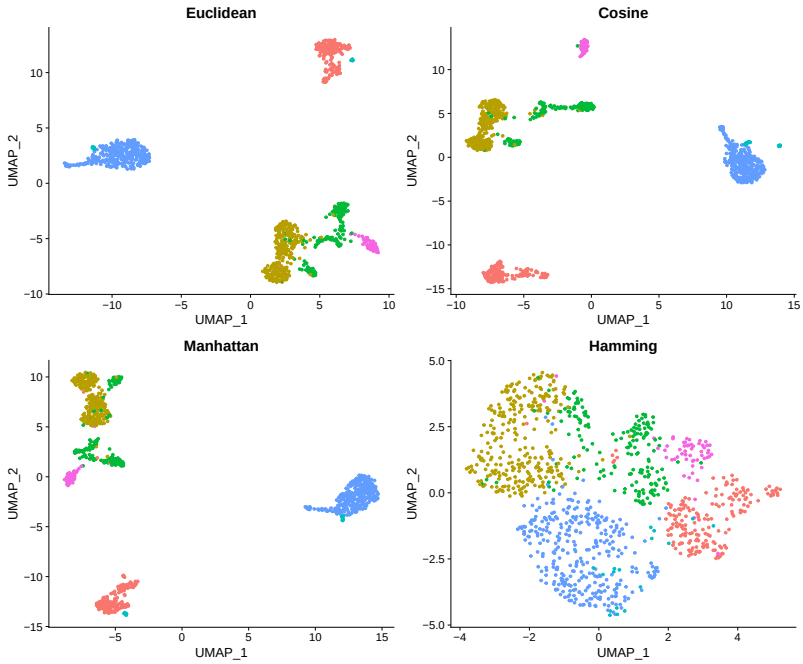
As you can see, these linkage methods and distances cluster the data differently. If you want, there are even more distance measures and linkage methods to play around with.

5.2.1 [OPTIONAL] UMAP: influence of distance measure

This was not mentioned beforem but also `RunUMAP` allows to set the distance metric used in high-dimensional space. While in principle this is also possible with t-SNE, `RunTSNE`, and most other implementations, do not provide this option. The four possibilities, `euclidean`, `cosine`, `manhattan`, and `hamming` distance are shown below. `Cosine` distance is the default (RunTSNE uses Euclidean distances).

There is not necessarily a clear winner. Hamming distances perform worse but they are usually used for different data, such as text as they ignore the numerical difference for a given comparison. Going with the default cosine is definitely not a bad choice in most applications.

```
# Euclidean distance
pbmc <- RunUMAP(pbmc, dims = 1:30, metric = "euclidean", verbose = FALSE)
p1 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() +
  ggtitle("Euclidean")
# Cosine distance
pbmc <- RunUMAP(pbmc, dims = 1:30, metric = "cosine", verbose = FALSE)
p2 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() +
  ggtitle("Cosine")
# Manhattan distance
pbmc <- RunUMAP(pbmc, dims = 1:30, metric = "manhattan", verbose = FALSE)
p3 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() +
  ggtitle("Manhattan")
# Hamming distance
pbmc <- RunUMAP(pbmc, dims = 1:30, metric = "hamming", verbose = FALSE)
p4 <- DimPlot(pbmc, reduction = "umap", group.by = "celltype") + NoLegend() +
  ggtitle("Hamming")
p1 + p2 + p3 + p4
```

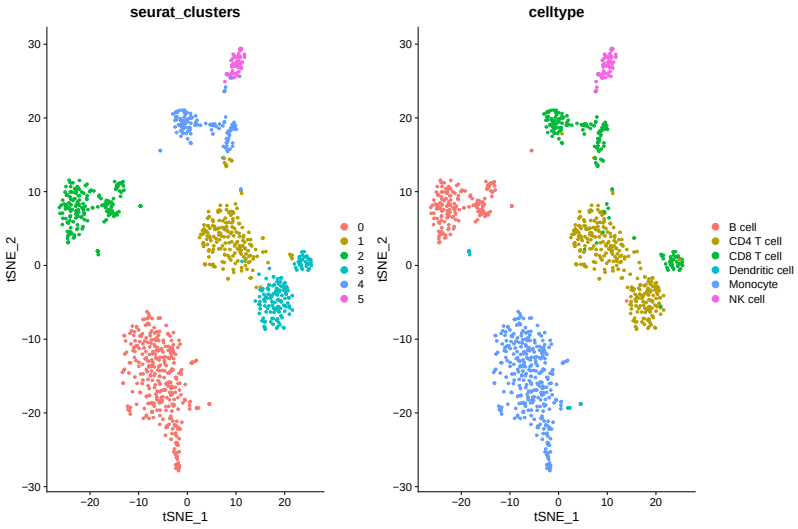


5.3 Graph based clustering

The clustering algorithm of Seurat itself is based on so-called graph based clustering (not explained in the lecture). The output of the clustering, will be saved automatically in the metadata as 'seurat_clusters'. This method includes a resolution parameter related to the number of clusters. You can play around with this

parameters to see how it influences the results.

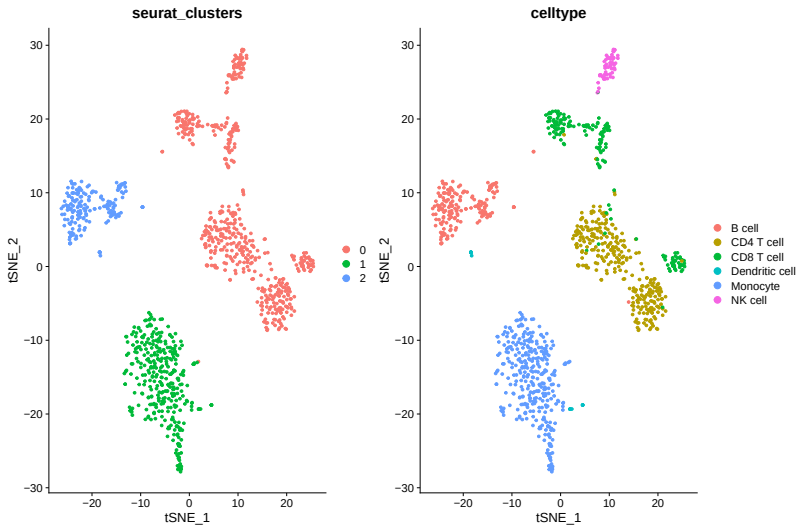
```
pbmc <- FindNeighbors(pbmc, dims = 1:10, verbose = FALSE)
pbmc <- FindClusters(pbmc, resolution = 0.25, verbose = FALSE)
p1 <- DimPlot(pbmc, reduction="tsne", group.by = "seurat_clusters")
p2 <- DimPlot(pbmc, reduction="tsne", group.by = "celltype")
p1+p2
```



5.4 Visualizing marker genes and annotating the cells

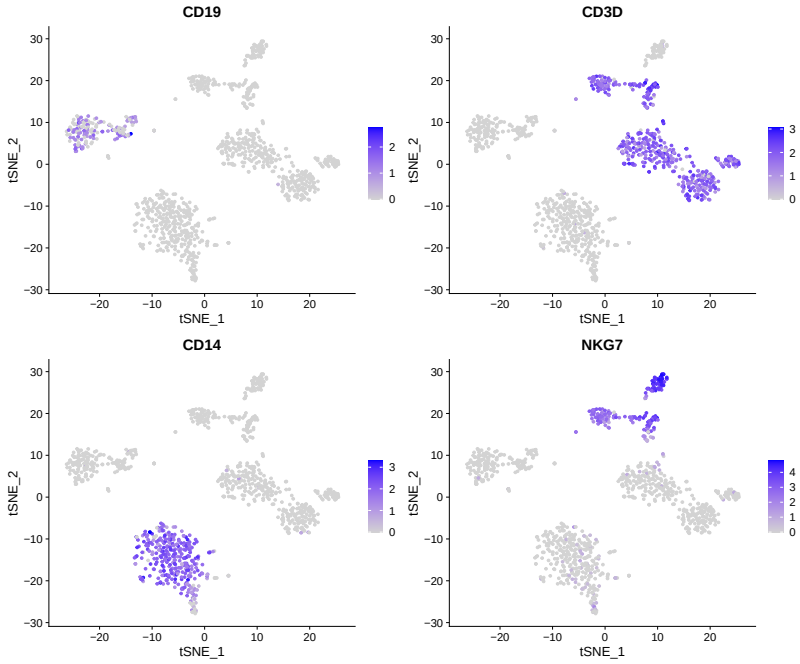
Once, you are satisfied with the clusters, these can be annotated by visualizing known marker genes or by looking at differentially expressed genes. Here, we just focus on known marker genes. A commonly used approach is that the data is annotated in a hierarchical fashion. First the data is annotated at a low resolution (e.g. only 2-3 cell types) and afterwards each cluster is subsetted from the data, clustered and annotated again. This process can continue until you're satisfied with the resolution.

```
pbmc <- FindNeighbors(pbmc, dims = 1:10, verbose = FALSE)
pbmc <- FindClusters(pbmc, resolution = 0.01, verbose = FALSE)
p1 <- DimPlot(pbmc, reduction="tsne", group.by = "seurat_clusters")
p2 <- DimPlot(pbmc, reduction="tsne", group.by = "celltype")
p1+p2
```



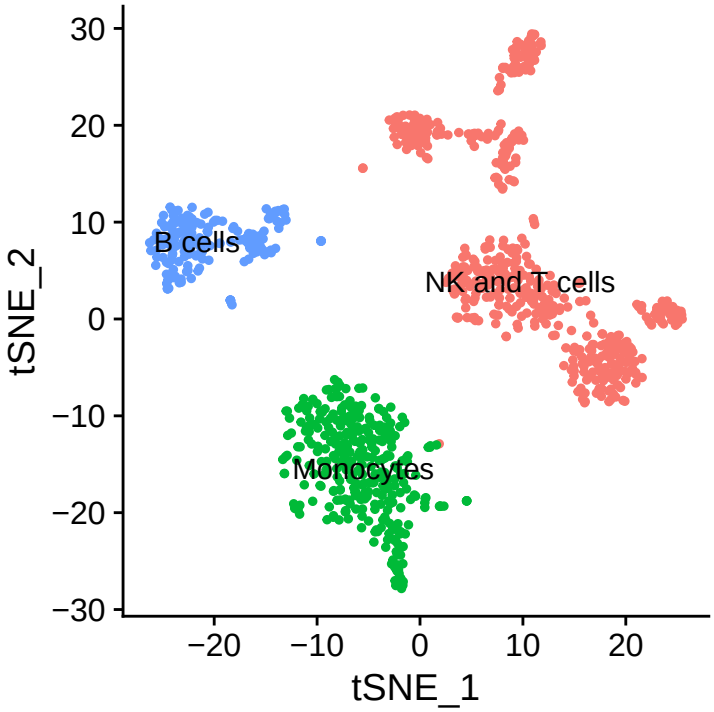
So now that we have clustered the data at a low resolution, we can visualize some marker genes: CD19 (B cells), CD3D (T cells), CD14 (Monocytes), NKG7 (NK cells).

```
FeaturePlot(pbmc, reduction='tsne', features=c('CD19', 'CD3D', 'CD14', 'NKG7'))
```



For a new, more complex dataset, you will probably need to visualize more genes before you can label a cluster. For now, we will assume that cluster 0 are NK and T cells, cluster 1 are Monocytes and cluster 2 are B cells. In the code below, you will assign these labels to your cluster.

```
new.cluster.ids <- c("NK and T cells", "Monocytes", "B cells")
names(new.cluster.ids) <- levels(pbmc)
pbmc <- RenameIdents(pbmc, new.cluster.ids)
DimPlot(pbmc, reduction = "tsne", label = TRUE) + NoLegend()
```



If you want to cluster the cells at a higher resolution, you could for instance subset the data now and repeat these steps. For now, we will just save the object.

```
saveRDS(pbmc, file = "pbmc3k_clust.rds")
```

5.5 Annotation using a reference atlas

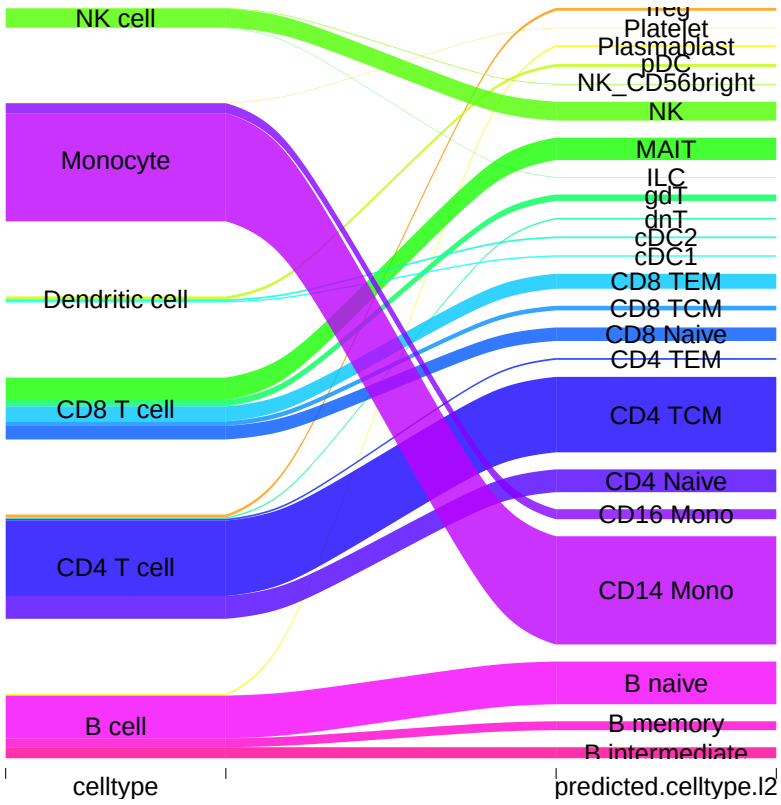
For some well studied tissues, there exists already a reference atlas. Cell type labels from this reference atlas can then be easily propagated to your own new dataset. As discussed in the lecture, clustering can be quite subjective and time-consuming. With these automatic approaches, you can overcome these issues.

An example of an automatic method is Azimuth. When using Azimuth for small datasets, it is easiest to use the web interface. Here, you can choose which reference atlas you want to use, upload your own dataset, choose the normalization procedure, and transfer the labels. You can try this yourself with the v3.1k PBMC dataset using the PBMC reference. The easiest way to annotate thee v3.1k dataset, is to upload the file `pbmc3k_clust.rds` that we saved at the end of the previous subsection. Then accept the default QC filters and click "Map cells to reference". Next click on "Download Results" and download the "Predicted cell types and scores (TSV)". Put the file `azimuth_pred.tsv` in the same folder `data` as the single-cell experiment data. Then run the code below to generate an alluvial plot, comparing the cell types predicted using Azimuth and the ones used throughout the computer lab.

```
labels <- read.delim("data/celltype_labels.tsv", stringsAsFactors = FALSE)
labels$cell <- paste0("v3.1k_", labels$cell)
labels.azimuth <- read.delim("data/azimuth_pred.tsv", stringsAsFactors = FALSE)
annotation <- merge(labels, labels.azimuth)[,c("celltype", "predicted.celltype.l2")]
# Column 'Freq' to store the total count of all combinations
annotation$Freq <- 1
annotation2D <- aggregate(Freq ~ ., data = annotation, sum)

cols <- rev(rainbow(nrow(annotation2D), start = 0.1, end = 0.9))
```

```
alluvial(
    select(annotation2D, celltype, predicted.celltype.l2),
    freq = annotation2D$Freq,
    col = cols,
    alpha = 0.8,
    gap.width = 0.5,
    cw = 0.2,
    blocks = FALSE,
    axis_labels = colnames(annotation2D)[1:2],
    cex = 1.5,
    cex.axis = 1.5)
```



Not bad at all!

6 Acknowledgements

Thanks to the developers of the MGC/BioSB single-cell analysis course for making their teaching material publicly available. This computer lab is largely a mash up of some of the practicals of the MGC/BioSB course. I also thank the authors of Current best practices in single-cell RNA-seq analysis: a tutorial from which I quoted several sentences (and for having written a very good review article of course).