scRNAseq Analysis in R with Seurat

2023-09-25

Contents

Chapter 1

About

Material for scRNAseq analysis in R with Seurat workshop.

This workshop follows the introductory Guided Clustering Tutorial tutorial from Seurat.

It is also drawing from a similar workshop held by Monash Bioinformatics Platform Single-Cell-Workshop, with material here

Chapter 2

Schedule

We will take 2 short breaks between topics each day.

Day 1

Time	Content
1:00 AEST	Welcome and Intro
1:30 AEST	Seurat Intro
	QC and filtering
	PCA and UMAP
4:30 AEST	Finish

Day 2

Time	Content
1:00 AEST	PCA and UMAP continued
	Clustering + Markers
	Assigning Celltypes
	Differential Expression
4:30 AEST	Finish

Part I

Content

Chapter 3

The Seurat object

Most of todays workshop will be following the Seurat PBMC tutorial (reproduced in the next section). We'll load raw counts data, do some QC and setup various useful information in a Seurat object.

But before that - what does a Seurat object look like, and what can we do with it once we've made one?

Lets have a look at a Seurat object that's already setup.

3.1 Load an existing Seurat object

The data we're working with today is a small dataset of about 3000 PBMCs (peripheral blood mononuclear cells) from a healthy donor. Just one sample.

This is an early demo dataset from 10X genomics (called pbmc3k) - you can find more information like qc reports here.

First, load Seurat package.

```
library(Seurat)
```

And here's the one we prepared earlier. Seurat objects are usually saved as '.rds' files, which is an R format for storing binary data (not-text or human-readable). The functions `readRDS()` can load it.

```
pbmc_processed <- readRDS("data/pbmc_tutorial.rds")
pbmc_processed
#> An object of class Seurat
#> 13714 features across 2700 samples within 1 assay
#> Active assay: RNA (13714 features, 2000 variable features)
#>  2 dimensional reductions calculated: pca, umap
```

3.2 What's in there?

Some of the most important information for working with Seurat objects is in the metadata. This is cell level information - each row is one cell, identified by its barcode. Extra information gets added to this table as analysis progresses.

```
head(pbmc_processed@meta.data)
#>                  orig.ident nCount_RNA nFeature_RNA
#> AAACATACAACCAC-1     pbmc3k       2419          779
#> AAACATTGAGCTAC-1     pbmc3k       4903         1352
#> AAACATTGATCAGC-1     pbmc3k       3147         1129
#> AAACCGTGCTTCCG-1     pbmc3k       2639          960
#> AAACCGTGTATGCG-1     pbmc3k        980          521
#> AAACGCACTGGTAC-1     pbmc3k       2163          781
#>                  percent.mt RNA_snn_res.0.5 seurat_clusters
#> AAACATACAACCAC-1  3.0177759               0               0
#> AAACATTGAGCTAC-1  3.7935958               3               3
#> AAACATTGATCAGC-1  0.8897363               2               2
#> AAACCGTGCTTCCG-1  1.7430845               5               5
#> AAACCGTGTATGCG-1  1.2244898               6               6
#> AAACGCACTGGTAC-1  1.6643551               2               2
```

That doesn't have any gene expression though, that's stored in an 'Assay'. The Assay structure has some nuances (see discussion below), but there are functions that get the assay data out for you.

By default this object will return the normalised data (from the only assay in this object, called RNA). Every '.' is a zero.

```
GetAssayData(pbmc_processed)[1:15,1:2]
#> 15 x 2 sparse Matrix of class "dgCMatrix"
#>               AAACATACAACCAC-1 AAACATTGAGCTAC-1
#> AL627309.1                    .                .
#> AP006222.2                    .                .
#> RP11-206L10.2                 .                .
#> RP11-206L10.9                 .                .
#> LINC00115                     .                .
#> NOC2L                         .                .
#> KLHL17                        .                .
#> PLEKHN1                       .                .
#> RP11-5407.17                  .                .
#> HES4                          .                .
#> RP11-5407.11                  .                .
#> ISG15                         .                .
#> AGRN                          .                .
#> C1orf159                      .                .
#> TNFRSF18                      .         1.625141
```

But the raw counts data is accessible too.

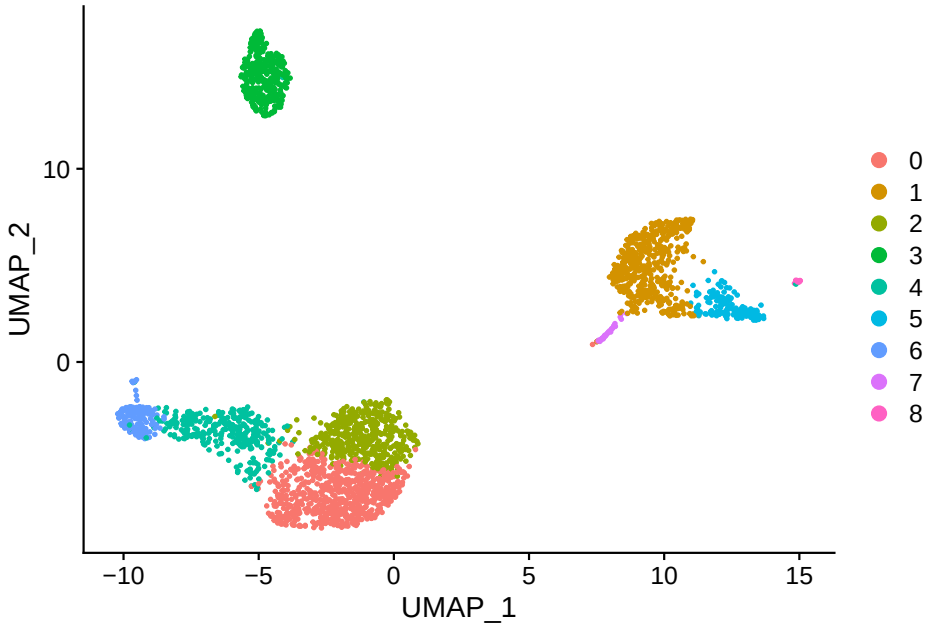
```
GetAssayData(pbmc_processed, slot='counts')[1:15,1:2]
#> 15 x 2 sparse Matrix of class "dgCMatrix"
#>               AAACATACAACCAC-1 AAACATTGAGCTAC-1
#> AL627309.1                    .                .
#> AP006222.2                    .                .
#> RP11-206L10.2                 .                .
#> RP11-206L10.9                 .                .
#> LINC00115                     .                .
#> NOC2L                         .                .
#> KLHL17                        .                .
#> PLEKHN1                       .                .
#> RP11-54O7.17                  .                .
#> HES4                          .                .
#> RP11-54O7.11                  .                .
#> ISG15                         .                .
#> AGRN                          .                .
#> C1orf159                      .                .
#> TNFRSF18                      .                2
```

Seurat generally hides alot of this data complexity, and provides functions for typical tasks. Like plotting.

3.3 Plotting

Lets plot a classic UMAP with `DimPlot`- defaults to the clusters on a 'umap' view.

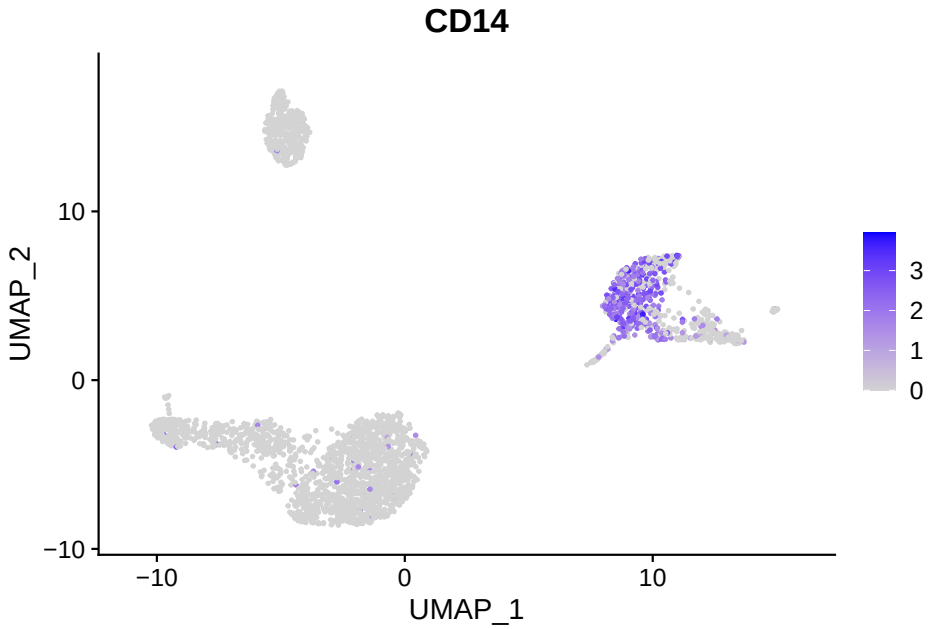
```
DimPlot(pbmc_processed)
```




```
# equivalent to
#DimPlot(pbmc_processed, reduction = 'umap')
```

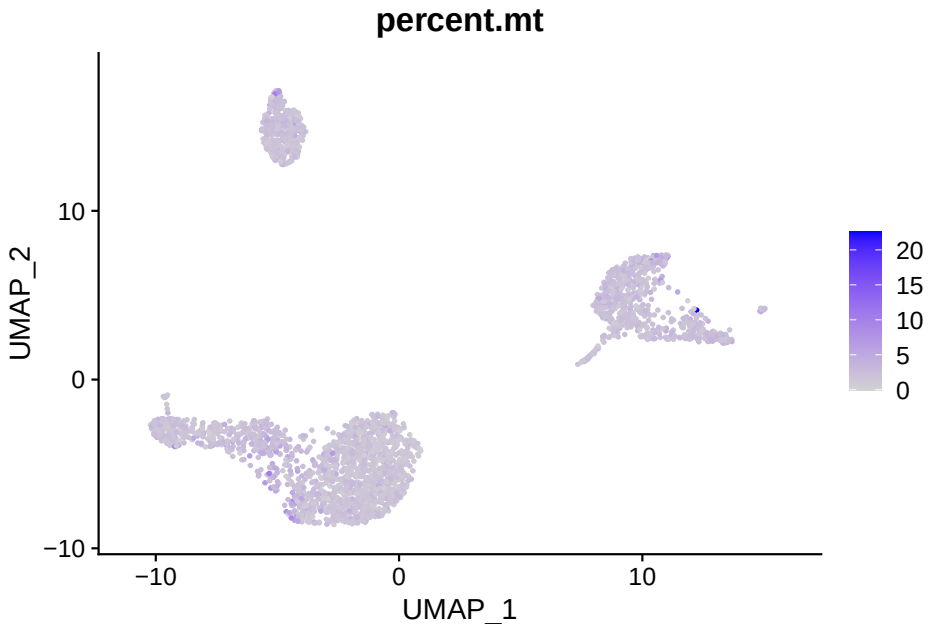
What about checking some gene expression? Genes are called 'features' in a Seurat object (because if it was CITE-seq they'd be proteins!). So the `FeaturePlot()` function will show gene expression.

```
FeaturePlot(pbmc_processed, features =  "CD14")
```



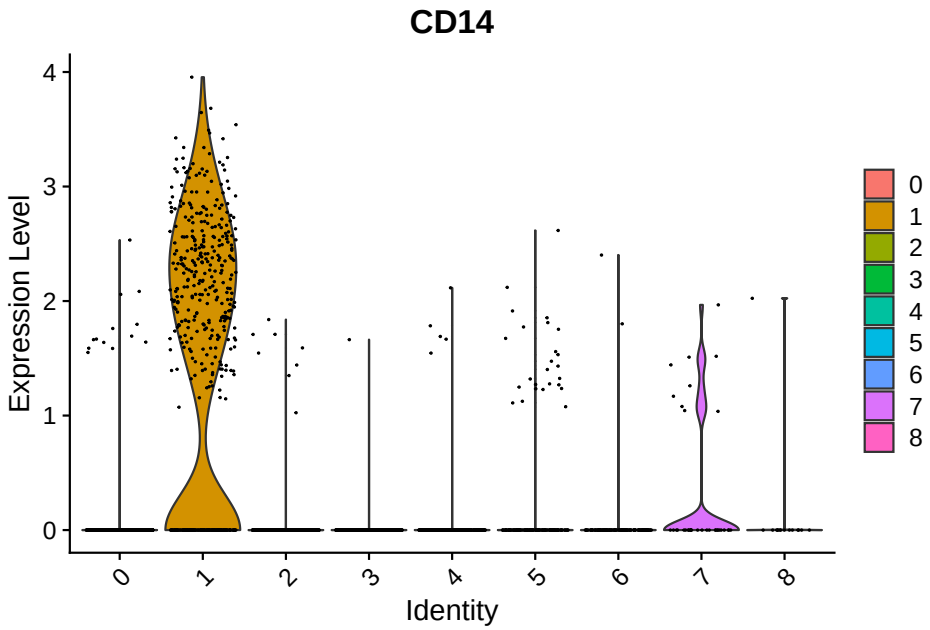
It also works for continuous cell-level data - any column in the metadata table.

```
FeaturePlot(pbmc_processed, features =  "percent.mt")
```



And what about showing CD14 expression across which clusters (or any other categorical information in the metadata)

```
VlnPlot(pbmc_processed, features = 'CD14')
```



Challenge: Plotting

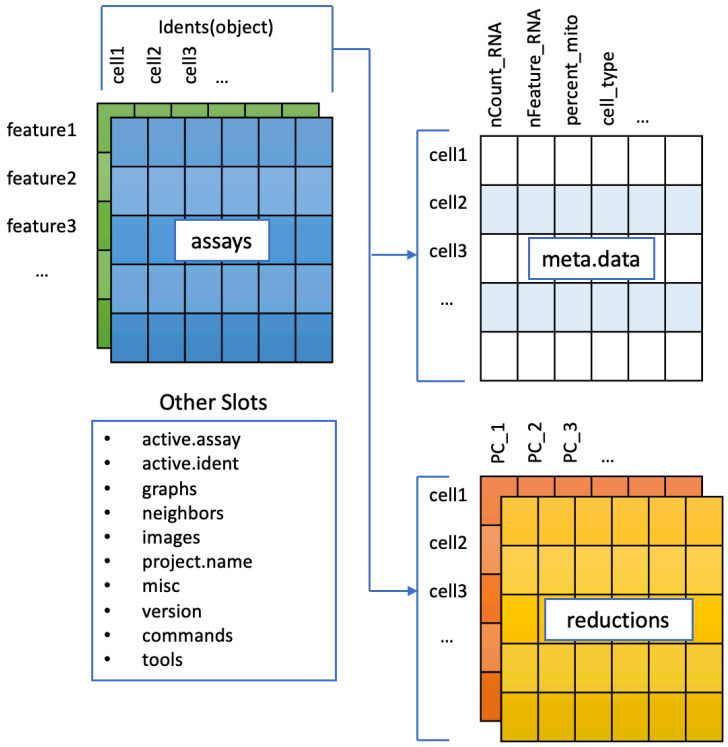
Plot your favourite gene. What cluster is it found in?

Tip: You can check if the gene is in the dataset by looking for it in the rownames of the seurat object `"CCT3" %in% rownames(pbmc_processed)`

Discussion: The Seurat Object in R

Lets take a look at the seurat object we have just created in R, `pbmc_processed`

To accomodate the complexity of data arising from a single cell RNA seq experiment, the seurat object keeps this as a container of multiple data tables that are linked.



The functions in seurat can access parts of the data object for analysis and visualisation, we will cover this later on.

There are a couple of concepts to discuss here.

Class

These are essentially data containers in R as a class, and can accessed as a variable in the R environment.

Classes are pre-defined and can contain multiple data tables and metadata. For Seurat, there are three types.

- Seurat - the main data class, contains all the data.
- Assay - found within the Seurat object. Depending on the experiment a cell could have data on RNA, ATAC etc measured
- DimReduc - for PCA and UMAP

Slots

Slots are parts within a class that contain specific data. These can be lists, data tables and vectors and can be accessed with conventional R methods.

Data Access

Many of the functions in Seurat operate on the data class and slots within them

seamlessly. There maybe occasion to access these separately to `hack` them, however this is an advanced analysis method.

The ways to access the slots can be through methods for the class (functions) or with standard R accessor nomenclature.

Examples of accessing a Seurat object

The `assays` slot in `pbmc_processed` can be accessed with `pbmc_processed@assays`.

The `RNA` assay can be accessed from this with `pbmc_processed@assays$RNA`.

We often want to access assays, so Seurat nicely gives us a shortcut `pbmc_processed$RNA`. You may sometimes see an alternative notation `pbmc_processed[["RNA"]]`.

In general, slots that are always in an object are accessed with `@` and things that may be different in different data sets are accessed with `$`.

Have a go

Use `str` to look at the structure of the Seurat object `pbmc_processed`.

What is in the `meta.data` slot within your Seurat object currently? What type of data is contained here?

Where is our count data within the Seurat object?

Part II

Seurat PBMC3k Tutorial

Chapter 4

Load data

This workshop follows the introductory Guided Clustering Tutorial tutorial from Seurat.

The bulk of this workshop may be found in its original format there.

4.1 Setup the Seurat Object

For this tutorial, we will be analyzing the a dataset of Peripheral Blood Mononuclear Cells (PBMC) freely available from 10X Genomics. There are 2,700 single cells that were sequenced on the Illumina NextSeq 500. The raw data can be found here.

We start by reading in the data. The `Read10X()` function reads in the output of the cellranger pipeline from 10X, returning a unique molecular identified (UMI) count matrix. The values in this matrix represent the number of molecules for each feature (i.e. gene; row) that are detected in each cell (column).

Note: What does the data look like?

What do the input files look like? It varies, but this is the output of the CellRanger pipleine, described here

```
analysis
    clustering
    diffexp
    pca
    tsne
    umap
cloupe.cloupe
filtered_feature_bc_matrix
    barcodes.tsv.gz
```

```
    features.tsv.gz
    matrix.mtx.gz
filtered_feature_bc_matrix.h5
metrics_summary.csv
molecule_info.h5
possorted_genome_bam.bam
possorted_genome_bam.bam.bai
raw_feature_bc_matrix
    barcodes.tsv.gz
    features.tsv.gz
    matrix.mtx.gz
raw_feature_bc_matrix.h5
web_summary.html
```

We next use the count matrix to create a `Seurat` object. The object serves as a container that contains both data (like the count matrix) and analysis (like PCA, or clustering results) for a single-cell dataset. For a technical discussion of the `Seurat` object structure, check out the GitHub Wiki. For example, the count matrix is stored in `pbmc@assays$RNA@counts`.

```
library(dplyr)
library(ggplot2)
library(Seurat)
library(patchwork)
```

```
# Load the PBMC dataset
pbmc.data <- Read10X(data.dir = "data/pbmc3k/filtered_gene_bc_matrices/hg19/")
# Initialize the Seurat object with the raw (non-normalized data).
pbmc <- CreateSeuratObject(counts = pbmc.data, project = "pbmc3k", min.cells = 3, min.
pbmc
#> An object of class Seurat
#> 13714 features across 2700 samples within 1 assay
#> Active assay: RNA (13714 features, 0 variable features)
```

What does data in a count matrix look like?

```
# Lets examine a few genes in the first thirty cells
pbmc.data[c("CD3D","TCL1A","MS4A1"), 1:30]
#> 3 x 30 sparse Matrix of class "dgCMatrix"
#>   [[ suppressing 30 column names 'AAACATACAACCAC-1', 'AAACATTGAGCTAC-1', 'AAACATTGA
#>
#> CD3D  4 . 10 . . 1 2 3 1 . . 2 7 1 . . 1 3 . 2  3 . . . . .
#> TCL1A . .  . . . . . 1 . . . . . . . . . . . . . 1 . . . .
#> MS4A1 . 6  . . . . . 1 1 1 . . . . . . . . . 36 1 2 . . 2
#>
```

```
#> CD3D  3 4 1 5
#> TCL1A . . . .
#> MS4A1 . . . .
```

The . values in the matrix represent 0s (no molecules detected). Since most values in an scRNA-seq matrix are 0, Seurat uses a sparse-matrix representation whenever possible. This results in significant memory and speed savings for Drop-seq/inDrop/10x data.

```
dense.size <- object.size(as.matrix(pbmc.data))
dense.size
#> 709591472 bytes
sparse.size <- object.size(pbmc.data)
sparse.size
#> 29905192 bytes
dense.size / sparse.size
#> 23.7 bytes
```

Chapter 5

QC Filtering

The steps below encompass the standard pre-processing workflow for scRNA-seq data in Seurat. These represent the selection and filtration of cells based on QC metrics, data normalization and scaling, and the detection of highly variable features.

5.1 QC and selecting cells for further analysis

Why do we need to do this?

Low quality cells can add noise to your results leading you to the wrong biological conclusions. Using only good quality cells helps you to avoid this. Reduce noise in the data by filtering out low quality cells such as dying or stressed cells (high mitochondrial expression) and cells with few features that can reflect empty droplets.

Seurat allows you to easily explore QC metrics and filter cells based on any user-defined criteria. A few QC metrics commonly used by the community include

- The number of unique genes detected in each cell.
  - Low-quality cells or empty droplets will often have very few genes
  - Cell doublets or multiplets may exhibit an aberrantly high gene count
- Similarly, the total number of molecules detected within a cell (correlates strongly with unique genes)
- The percentage of reads that map to the mitochondrial genome
  - Low-quality / dying cells often exhibit extensive mitochondrial contamination
  - We calculate mitochondrial QC metrics with the `PercentageFeatureSet()` function, which calculates the percentage of counts originating from

a set of features
  - We use the set of all genes starting with `MT-` as a set of mitochondrial genes

```
# The $ operator can add columns to object metadata.
# This is a great place to stash QC stats
pbmc$percent.mt <- PercentageFeatureSet(pbmc, pattern = "^MT-")
```

Challenge: The meta.data slot in the Seurat object

Where are QC metrics stored in Seurat?

- The number of unique genes and total molecules are automatically calculated during `CreateSeuratObject()`
  - You can find them stored in the object meta data

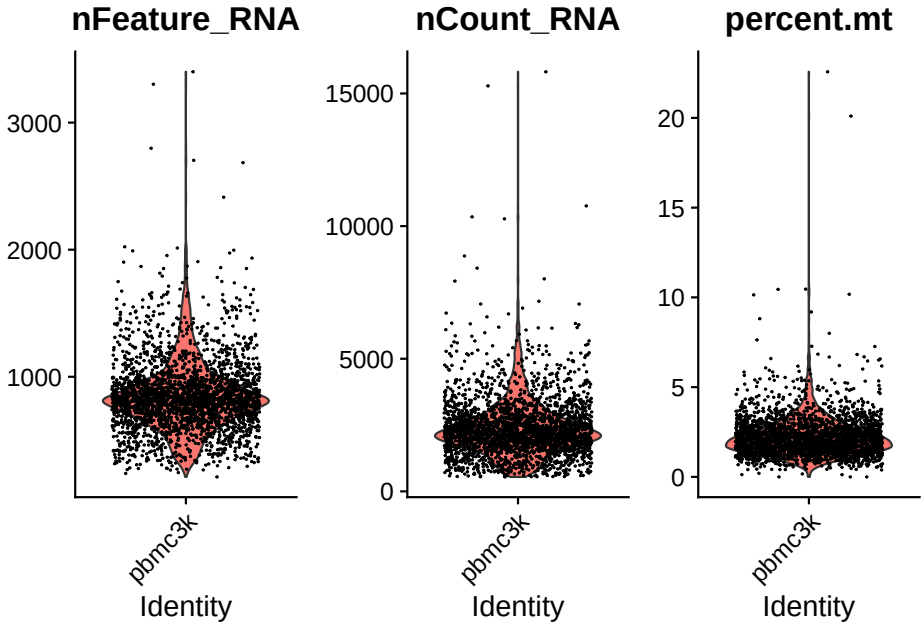
1. What do you notice has changed within the `meta.data` table now that we have calculated mitochondrial gene proportion?

2. Imagine that this is the first of several samples in our experiment. Add a `samplename` column to to the `meta.data` table.

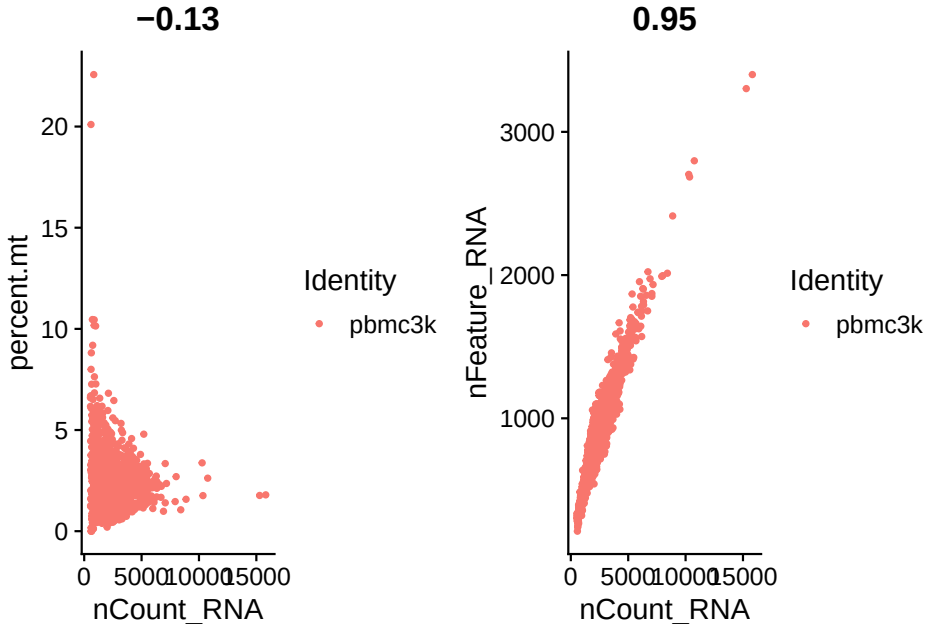
In the example below, we visualize QC metrics, and use these to filter cells.

- We filter cells that have unique feature counts over 2,500 or less than 200
- We filter cells that have >5% mitochondrial counts

```
#Visualize QC metrics as a violin plot
VlnPlot(pbmc, features = c("nFeature_RNA", "nCount_RNA", "percent.mt"), ncol = 3)
```

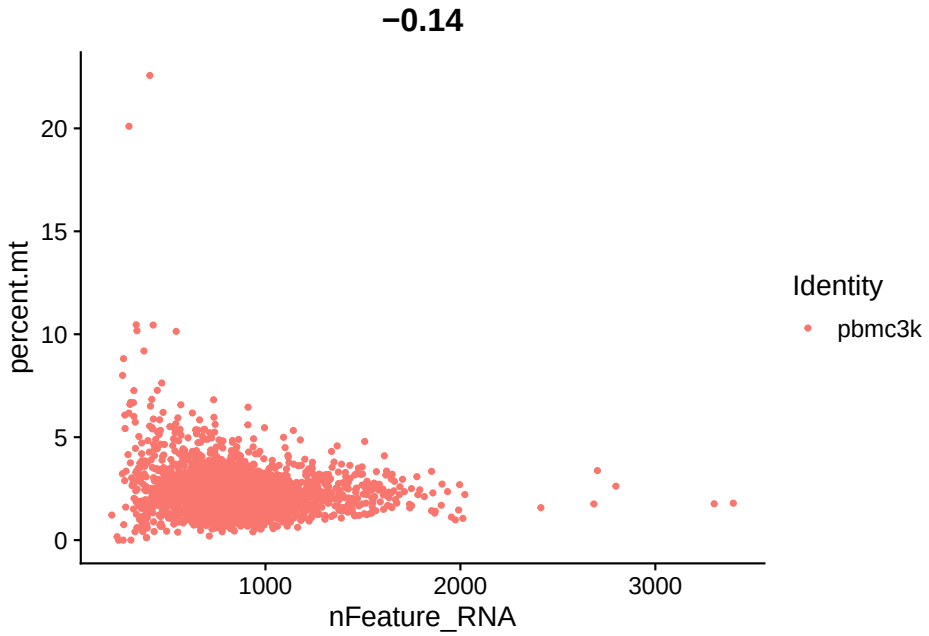



```
# FeatureScatter is typically used to visualize feature-feature relationships,
# but can be used for anything calculated by the object,
# i.e. columns in object metadata, PC scores etc.
plot1 <- FeatureScatter(pbmc, feature1 = "nCount_RNA", feature2 = "percent.mt")
plot2 <- FeatureScatter(pbmc, feature1 = "nCount_RNA", feature2 = "nFeature_RNA")
plot1 + plot2
```



Lets look at the number of features (genes) to the percent mitochondrial genes plot.

```
plot3 <- FeatureScatter(pbmc, feature1 = "nFeature_RNA", feature2 = "percent.mt")
plot3
```

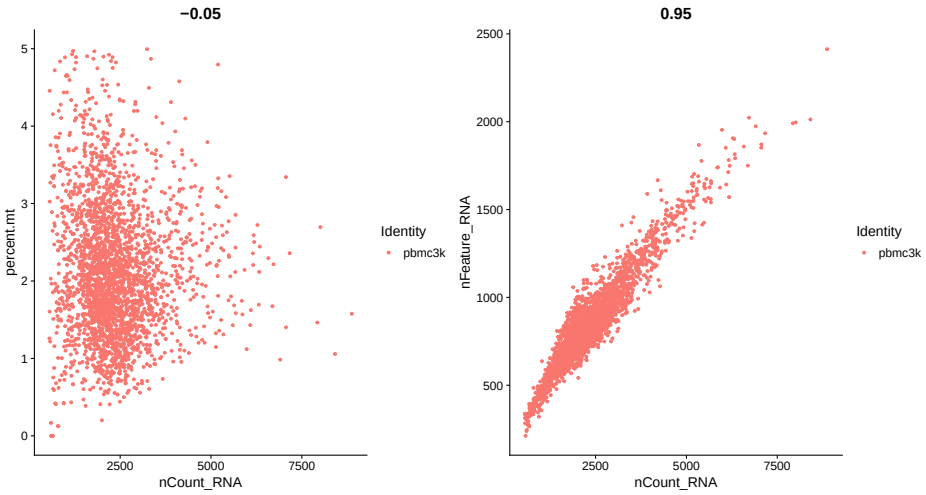


Okay we are happy with our thresholds for mitochondrial percentage in cells, lets apply them and subset our data. This will remove the cells we think are of poor quality.

```
pbmc <- subset(pbmc, subset = nFeature_RNA > 200 & nFeature_RNA < 2500 & percent.mt < 5)
```

Lets replot the feature scatters and see what they look like.

```
plot5 <- FeatureScatter(pbmc, feature1 = "nCount_RNA", feature2 = "percent.mt")
plot6 <- FeatureScatter(pbmc, feature1 = "nCount_RNA", feature2 = "nFeature_RNA")
plot5 + plot6
```



Challenge: Filter the cells

Apply the filtering thresolds defined above.

- How many cells survived filtering?

The PBMC3k dataset we're working with in this tutorial is quite old. There are a number of other example datasets available from the 10X website, including this one - published in 2022, sequencing 10k PBMCs with a newer chemistry and counting method.

- What thresholds would you chose to apply to this modern dataset?

```
pbmc10k_unfiltered <- readRDS("data/10k_PBMC_v3.1ChromiumX_Intronic.rds")
VlnPlot(pbmc10k_unfiltered, features = c("nFeature_RNA", "nCount_RNA"), ncol = 2)
```

Chapter 6

Normalisation

Why do we need to do this?

The sequencing depth can be different per cell. This can bias the counts of expression showing higher numbers for more sequenced cells leading to the wrong biological conclusions. To correct this the feature counts are normalized.

After removing unwanted cells from the dataset, the next step is to normalize the data. By default, we employ a global-scaling normalization method "Log-Normalize" that normalizes the feature expression measurements for each cell by the total expression, multiplies this by a scale factor (10,000 by default), and log-transforms the result. Normalized values are stored in `pbmc$RNA@data`.

```
pbmc <- NormalizeData(pbmc, normalization.method = "LogNormalize", scale.factor = 1e4)
```

For clarity, in this previous line of code (and in future commands), we provide the default values for certain parameters in the function call. However, this isn't required and the same behavior can be achieved with:

```
pbmc <- NormalizeData(pbmc)
```

Chapter 7

PCAs and UMAPs

7.1 Identification of highly variable features (feature selection)

Why do we need to do this?

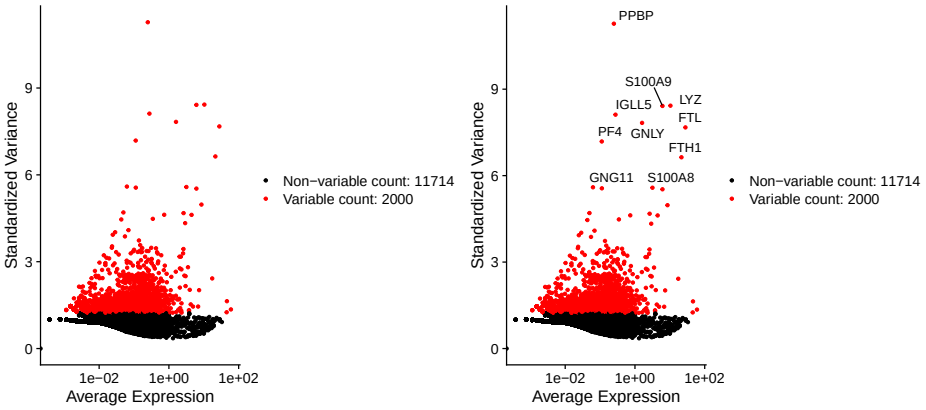
Identifying the most variable features allows retaining the real biological variability of the data and reduce noise in the data.

We next calculate a subset of features that exhibit high cell-to-cell variation in the dataset (i.e, they are highly expressed in some cells, and lowly expressed in others). We and others have found that focusing on these genes in downstream analysis helps to highlight biological signal in single-cell datasets.

Our procedure in Seurat is described in detail here, and improves on previous versions by directly modeling the mean-variance relationship inherent in single-cell data, and is implemented in the `FindVariableFeatures()` function. By default, we return 2,000 features per dataset. These will be used in downstream analysis, like PCA.

```
pbmc <- FindVariableFeatures(pbmc, selection.method = 'vst', nfeatures = 2000)
# Identify the 10 most highly variable genes
top10 <- head(VariableFeatures(pbmc), 10)
# plot variable features with and without labels
plot1 <- VariableFeaturePlot(pbmc)
plot2 <- LabelPoints(plot = plot1, points = top10, repel = TRUE)
#> When using repel, set xnudge and ynudge to 0 for optimal results
plot1 + plot2
#> Warning: Transformation introduced infinite values in continuous
```

```
#> x-axis
#> Transformation introduced infinite values in continuous
#> x-axis
```



7.2 Scaling the data

Why do we need to do this?

Highly expresed genes can overpower the signal of other less expresed genes with equal importance. Within the same cell the assumption is that the underlying RNA content is constant. Aditionally, If variables are provided in vars.to.regress, they are individually regressed against each feature, and the resulting residuals are then scaled and centered. This step allows controling for cell cycle and other factors that may bias your clustering.

Next, we apply a linear transformation ('scaling') that is a standard pre-processing step prior to dimensional reduction techniques like PCA. The `ScaleData()` function:

- Shifts the expression of each gene, so that the mean expression across cells is 0
- Scales the expression of each gene, so that the variance across cells is 1
  - This step gives equal weight in downstream analyses, so that highly-expressed genes do not dominate
- The results of this are stored in `pbmc$RNA@scale.data`

```
all.genes <- rownames(pbmc)
pbmc <- ScaleData(pbmc, features = all.genes)
#> Centering and scaling data matrix
```

This step takes too long! Can I make it faster?

Scaling is an essential step in the Seurat workflow, but only on genes that will be used as input to PCA. Therefore, the default in `ScaleData()` is only to perform scaling on the previously identified variable features (2,000 by default). To do this, omit the `features` argument in the previous function call, i.e.

```
# pbmc <- ScaleData(pbmc)
```

Your PCA and clustering results will be unaffected. However, Seurat heatmaps (produced as shown below with `DoHeatmap()`) require genes in the heatmap to be scaled, to make sure highly-expressed genes don't dominate the heatmap. To make sure we don't leave any genes out of the heatmap later, we are scaling all genes in this tutorial.

How can I remove unwanted sources of variation, as in Seurat v2?

In `Seurat v2` we also use the `ScaleData()` function to remove unwanted sources of variation from a single-cell dataset. For example, we could 'regress out' heterogeneity associated with (for example) cell cycle stage, or mitochondrial contamination. These features are still supported in `ScaleData()` in `Seurat v3`, i.e.:

```
# pbmc <- ScaleData(pbmc, vars.to.regress = 'percent.mt')
```

However, particularly for advanced users who would like to use this functionality, we strongly recommend the use of our new normalization workflow, `SCTransform()`. The method is described in our paper, with a separate vignette using Seurat v3 here. As with `ScaleData()`, the function `SCTransform()` also includes a `vars.to.regress` parameter.

Chapter 8

Dimensionality reduction

Why do we need to do this?

Imagine each gene represents a dimension - or an axis on a plot. We could plot the expression of two genes with a simple scatterplot. But a genome has thousands of genes - how do you collate all the information from each of those genes in a way that allows you to visualise it in a 2 dimensional image. This is where dimensionality reduction comes in, we calculate meta-features that contains combinations of the variation of different genes. From thousands of genes, we end up with 10s of meta-features

8.1 Perform linear dimensional reduction

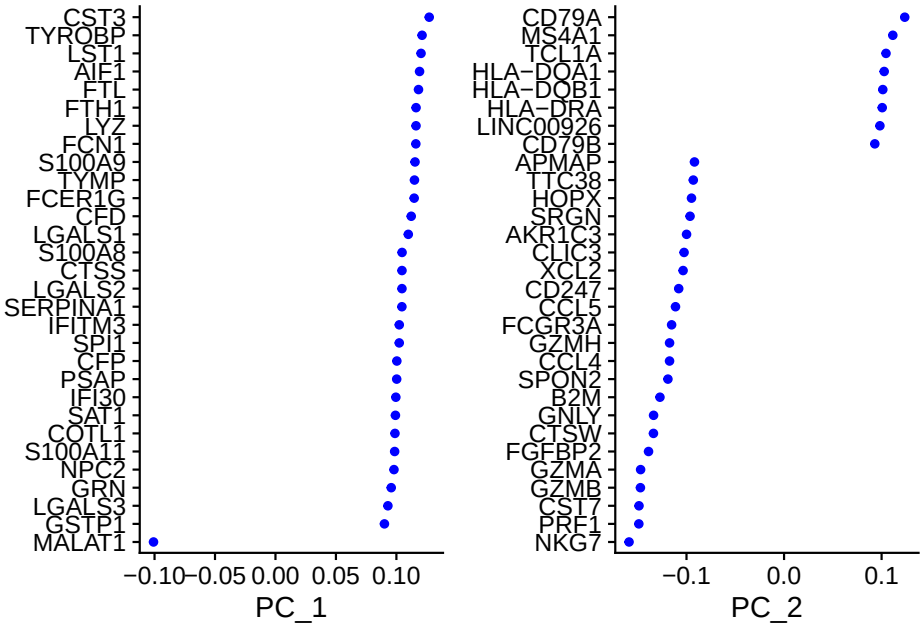
Next we perform PCA on the scaled data. By default, only the previously determined variable features are used as input, but can be defined using `features` argument if you wish to choose a different subset.

```
pbmc <- RunPCA(pbmc, features = VariableFeatures(object = pbmc))
#> PC_ 1
#> Positive:  CST3, TYROBP, LST1, AIF1, FTL, FTH1, LYZ, FCN1, S100A9, TYMP
#>     FCER1G, CFD, LGALS1, S100A8, CTSS, LGALS2, SERPINA1, IFITM3, SPI1, CFP
#>     PSAP, IFI30, SAT1, COTL1, S100A11, NPC2, GRN, LGALS3, GSTP1, PYCARD
#> Negative:  MALAT1, LTB, IL32, IL7R, CD2, B2M, ACAP1, CD27, STK17A, CTSW
#>     CD247, GIMAP5, AQP3, CCL5, SELL, TRAF3IP3, GZMA, MAL, CST7, ITM2A
#>     MYC, GIMAP7, HOPX, BEX2, LDLRAP1, GZMK, ETS1, ZAP70, TNFAIP8, RIC3
#> PC_ 2
#> Positive:  CD79A, MS4A1, TCL1A, HLA-DQA1, HLA-DQB1, HLA-DRA, LINC00926, CD79B, HLA-DRB1, CD74
#>     HLA-DMA, HLA-DPB1, HLA-DQA2, CD37, HLA-DRB5, HLA-DMB, HLA-DPA1, FCRLA, HVCN1, LTB
#>     BLNK, P2RX5, IGLL5, IRF8, SWAP70, ARHGAP24, FCGR2B, SMIM14, PPP1R14A, C16orf74
```

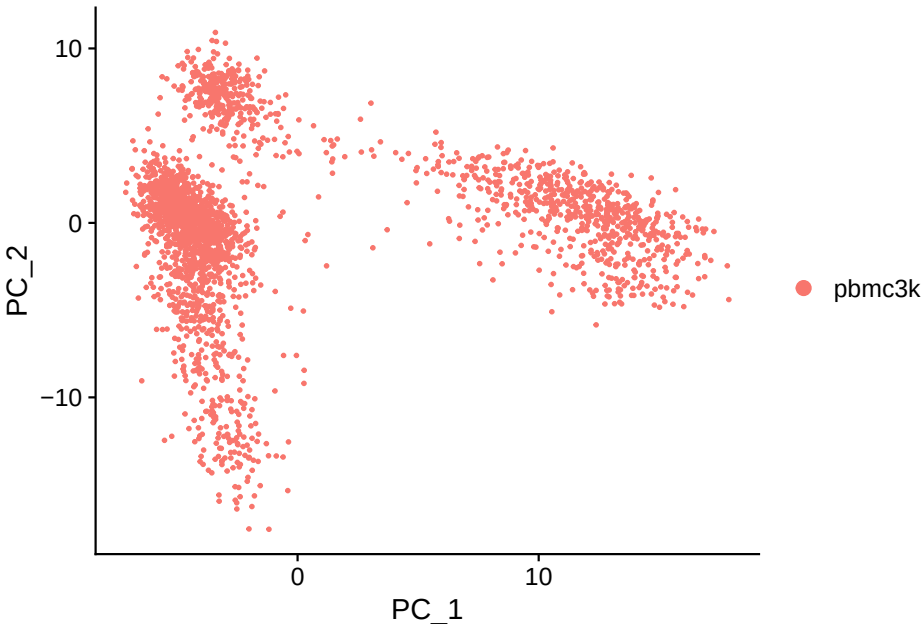
```
#> Negative:  NKG7, PRF1, CST7, GZMB, GZMA, FGFBP2, CTSW, GNLY, B2M, SPON2
#>     CCL4, GZMH, FCGR3A, CCL5, CD247, XCL2, CLIC3, AKR1C3, SRGN, HOPX
#>     TTC38, APMAP, CTSC, S100A4, IGFBP7, ANXA1, ID2, IL32, XCL1, RHOC
#> PC_ 3
#> Positive:  HLA-DQA1, CD79A, CD79B, HLA-DQB1, HLA-DPB1, HLA-DPA1, CD74, MS4A1, HLA-D
#>     HLA-DRB5, HLA-DQA2, TCL1A, LINC00926, HLA-DMB, HLA-DMA, CD37, HVCN1, FCRLA, IRF
#>     PLAC8, BLNK, MALAT1, SMIM14, PLD4, P2RX5, IGLL5, LAT2, SWAP70, FCGR2B
#> Negative:  PPBP, PF4, SDPR, SPARC, GNG11, NRGN, GP9, RGS18, TUBB1, CLU
#>     HIST1H2AC, AP001189.4, ITGA2B, CD9, TMEM40, PTCRA, CA2, ACRBP, MMD, TREML1
#>     NGFRAP1, F13A1, SEPT5, RUFY1, TSC22D1, MPP1, CMTM5, RP11-367G6.3, MYL9, GP1BA
#> PC_ 4
#> Positive:  HLA-DQA1, CD79B, CD79A, MS4A1, HLA-DQB1, CD74, HIST1H2AC, HLA-DPB1, PF4,
#>     TCL1A, HLA-DRB1, HLA-DPA1, HLA-DQA2, PPBP, HLA-DRA, LINC00926, GNG11, SPARC, HL
#>     GP9, AP001189.4, CA2, PTCRA, CD9, NRGN, RGS18, CLU, TUBB1, GZMB
#> Negative:  VIM, IL7R, S100A6, IL32, S100A8, S100A4, GIMAP7, S100A10, S100A9, MAL
#>     AQP3, CD2, CD14, FYB, LGALS2, GIMAP4, ANXA1, CD27, FCN1, RBP7
#>     LYZ, S100A11, GIMAP5, MS4A6A, S100A12, FOLR3, TRABD2A, AIF1, IL8, IFI6
#> PC_ 5
#> Positive:  GZMB, NKG7, S100A8, FGFBP2, GNLY, CCL4, CST7, PRF1, GZMA, SPON2
#>     GZMH, S100A9, LGALS2, CCL3, CTSW, XCL2, CD14, CLIC3, S100A12, RBP7
#>     CCL5, MS4A6A, GSTP1, FOLR3, IGFBP7, TYROBP, TTC38, AKR1C3, XCL1, HOPX
#> Negative:  LTB, IL7R, CKB, VIM, MS4A7, AQP3, CYTIP, RP11-290F20.3, SIGLEC10, HMOX1
#>     LILRB2, PTGES3, MAL, CD27, HN1, CD2, GDI2, CORO1B, ANXA5, TUBA1B
#>     FAM110A, ATP1A1, TRADD, PPA1, CCDC109B, ABRACL, CTD-2006K23.1, WARS, VMO1, FYB
```

Seurat provides several useful ways of visualizing both cells and features that define the PCA, including `VizDimReduction()`, `DimPlot()`, and `DimHeatmap()`

```
# Examine and visualize PCA results a few different ways
print(pbmc$pca, dims = 1:5, nfeatures = 5)
#> PC_ 1
#> Positive:  CST3, TYROBP, LST1, AIF1, FTL
#> Negative:  MALAT1, LTB, IL32, IL7R, CD2
#> PC_ 2
#> Positive:  CD79A, MS4A1, TCL1A, HLA-DQA1, HLA-DQB1
#> Negative:  NKG7, PRF1, CST7, GZMB, GZMA
#> PC_ 3
#> Positive:  HLA-DQA1, CD79A, CD79B, HLA-DQB1, HLA-DPB1
#> Negative:  PPBP, PF4, SDPR, SPARC, GNG11
#> PC_ 4
#> Positive:  HLA-DQA1, CD79B, CD79A, MS4A1, HLA-DQB1
#> Negative:  VIM, IL7R, S100A6, IL32, S100A8
#> PC_ 5
#> Positive:  GZMB, NKG7, S100A8, FGFBP2, GNLY
#> Negative:  LTB, IL7R, CKB, VIM, MS4A7
VizDimLoadings(pbmc, dims = 1:2, reduction = 'pca')
```



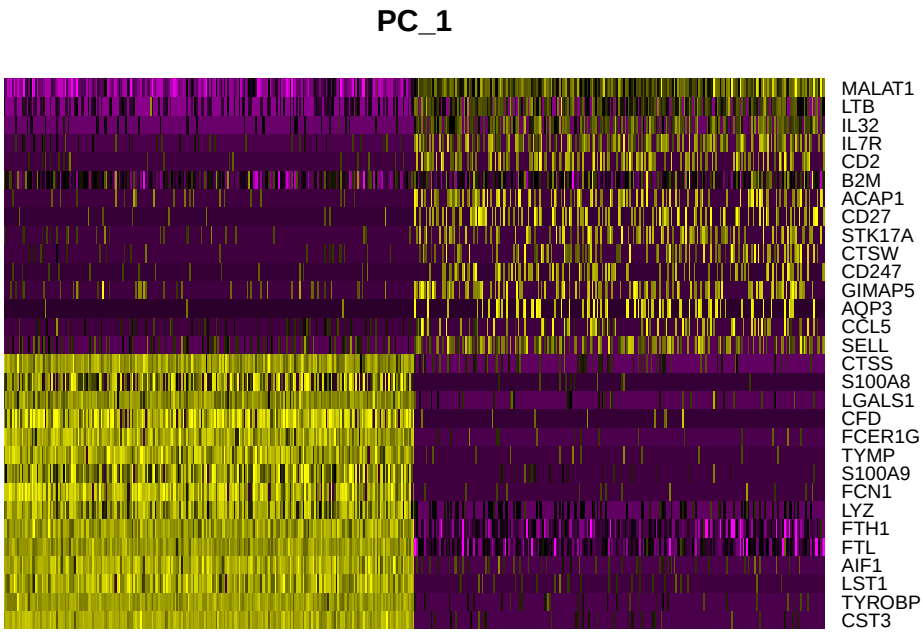

```
DimPlot(pbmc, reduction = 'pca')
```



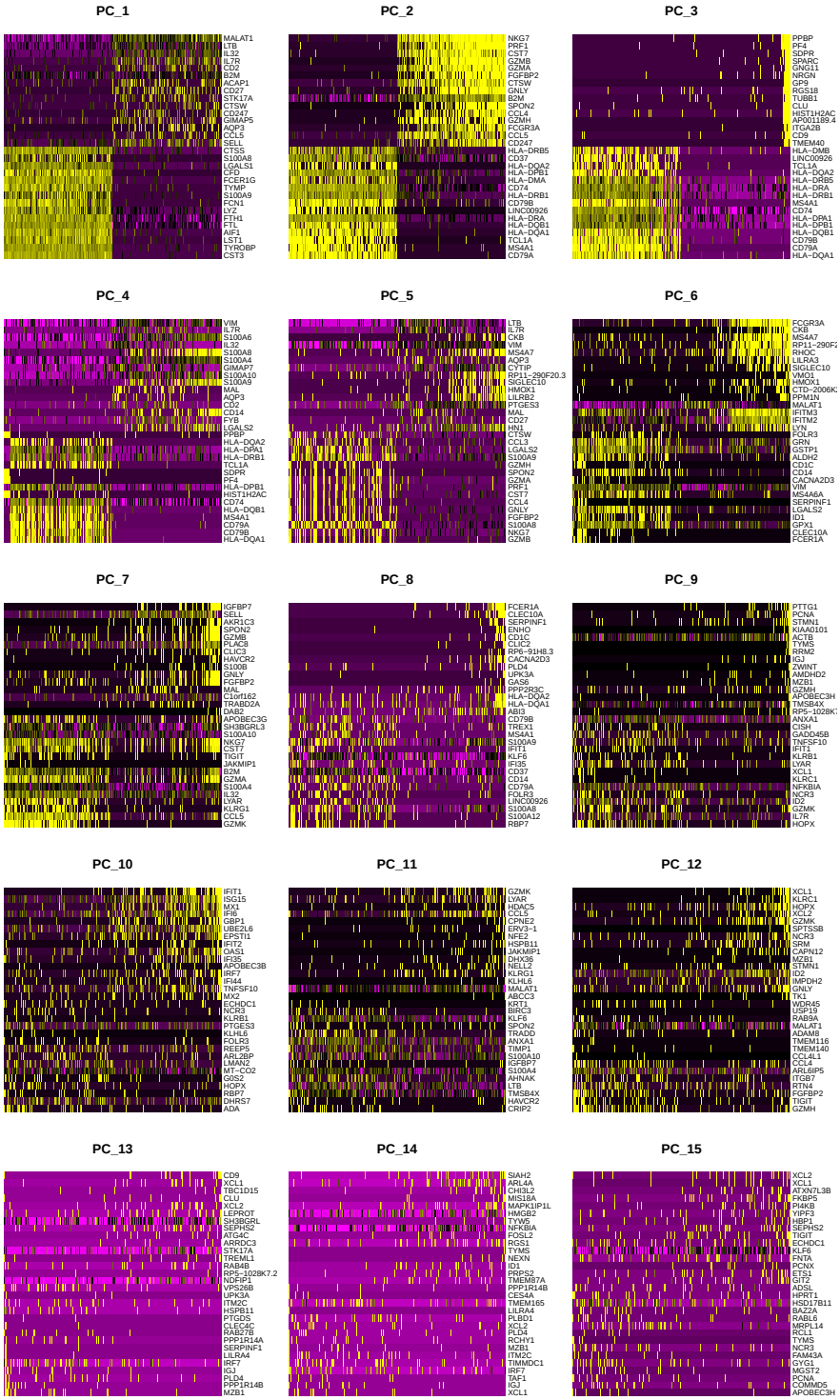
In particular `DimHeatmap()` allows for easy exploration of the primary sources of heterogeneity in a dataset, and can be useful when trying to decide which PCs to include for further downstream analyses. Both cells and features are ordered according to their PCA scores. Setting `cells` to a number plots the

'extreme' cells on both ends of the spectrum, which dramatically speeds plotting for large datasets. Though clearly a supervised analysis, we find this to be a valuable tool for exploring correlated feature sets.

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




```
DimHeatmap(pbmc, dims = 1:15, cells = 500, balanced = TRUE)
```


PC_1
MALAT1
LTB
IL32
IL7R
CD2
B2M
ACAP1
CD27
STK17A
CTSW
CD247
GIMAP5
AQP3
CCL5
SELL
CTSS
S100A8
LGALS1
CFD
FCER1G
TYMP
S100A9
FCN1
LYZ
FTH1
FTL
AIF1
LST1
TYROBP
CST3
PC_2
NKG7
PRF1
CST7
GZMB
GZMA
FGFBP2
CTSW
GNLY
B2M
SPON2
CCL4
GZMH
FCGR3A
CCL5
CD247
HLA−DRB5
CD37
HLA−DQA2
HLA−DPB1
HLA−DMA
CD74
HLA−DRB1
CD79B
LINC00926
HLA−DRA
HLA−DQB1
HLA−DQA1
TCL1A
MS4A1
CD79A
PC_3
PPBP
PF4
SDPR
SPARC
GNG11
NRGN
GP9
RGS18
TUBB1
CLU
HIST1H2AC
AP001189.4
ITGA2B
CD9
TMEM40
HLA−DMB
LINC00926
TCL1A
HLA−DQA2
HLA−DRB5
HLA−DRA
HLA−DRB1
MS4A1
CD74
HLA−DPA1
HLA−DPB1
HLA−DQB1
CD79B
CD79A
HLA−DQA1
PC_4
VIM
IL7R
S100A6
IL32
S100A8
S100A4
GIMAP7
S100A10
S100A9
MAL
AQP3
CD2
CD14
FYB
LGALS2
PPBP
HLA−DQA2
HLA−DPA1
HLA−DRB1
TCL1A
SDPR
PF4
HLA−DPB1
HIST1H2AC
CD74
HLA−DQB1
MS4A1
CD79A
CD79B
HLA−DQA1
PC_5
LTB
IL7R
CKB
VIM
MS4A7
AQP3
CYTIP
RP11−290F20.3
SIGLEC10
HMOX1
LILRB2
PTGES3
MAL
CD27
HN1
CTSW
CCL3
LGALS2
S100A9
GZMH
SPON2
GZMA
PRF1
CST7
CCL4
GNLY
FGFBP2
S100A8
NKG7
GZMB
PC_6
FCGR3A
CKB
MS4A7
RP11−290F2
RHOC
LILRA3
SIGLEC10
VMO1
HMOX1
CTD−2006K
IPMIN
MALAT1
IFITM3
IFITM2
LYN
TOLR3
GRN
GSTP1
ALDH2
CD1C
CD14
CACNA2D3
VIM
MS4A6A
SERPINF1
LGALS2
ID1
GPX1
CLEC10A
FCER1A
PC_7
IGFBP7
SELL
AKR1C3
SPON2
GZMB
PLAC8
CLIC3
HAVCR2
S100B
GNLY
FGFBP2
MAL
C1orf162
TRABD2A
DAB2
APOBEC3G
SH3BGRL3
S100A10
NKG7
CST7
TIGIT
JAKMIP1
B2M
GZMA
S100A4
IL32
LYAR
KLRG1
CCL5
GZMK
PC_8
FCER1A
CLEC10A
SERPINF1
ENHO
CD1C
CLIC2
RP6−91H8.3
CACNA2D3
PLD4
UPK3A
GAS6
PPP2R3C
HLA−DQA2
HLA−DQA1
ABI3
CD79B
TREX1
MS4A1
S100A9
IFIT1
KLF6
IFI35
CD37
CD14
CD79A
FOLR3
LINC00926
S100A8
S100A12
RBP7
PC_9
PTTG1
PCNA
STMN1
KIAA0101
CTB
TYMS
RRM2
IGJ
ZWINT
AMDHD2
MZB1
GZMH
APOBEC3H
TMSB4X
RPS−1028K
ANXA1
CISH
GADD45B
TNFSF10
IFIT1
KLRB1
LYAR
XCL1
KLRC1
NFKBIA
NCR3
ID2
GZMK
IL7R
HOPX
PC_10
IFIT1
ISG15
MX1
IFI6
GBP1
UBE2L6
EPSTI1
HIT2
OAS1
IFI35
APOBEC3B
IRF7
IFI44
TNFSF10
MX2
ECHDC1
NCR3
KLRB1
PTGES3
KLHL6
FOLR3
REEP5
ARL2BP
LMAN2
MT−CO2
GOS2
HOPX
RBP7
DHRS7
ADA
PC_11
GZMK
LYAR
HDAC5
CCL5
CPNE2
ERV3−1
NFE2
FGFBP11
JAKMIP1
DHX36
NELL2
KLRG1
KLHL6
MALAT1
ABCC3
KRT1
BIRC3
KLF6
SPON2
TRADD
ANXA1
TIMP1
S100A10
FGFBP7
S100A4
AHNAK
LTB
TMSB4X
HAVCR2
CRIP2
PC_12
XCL1
KLRC1
HOPX
XCL2
GZMK
SPTSSB
NCR3
SOX4
CAPN12
MZB1
STMN1
ID2
IMPDH2
GNLY
TK1
WDR45
HSP19
RAB9A
MALAT1
ADAM8
TMEM116
TMEM140
CCL4L1
CCL4
ARL6IP5
ITGB7
RTN4
FGFBP2
TIGIT
GZMH
PC_13
CD9
XCL1
TBC1D15
CLU
XCL2
LEPROT
SH3BGRL
SEPHS2
ATG4C
ARRDC3
STK17A
TREML1
RAB4B
RPS−1028K7.2
NDFIP1
VPS26B
UPK3A
ITM2C
HSPB11
PTGDS
CLEC4C
RAB27B
PPP1R14A
SERPINF1
LILRA4
IRF7
IGJ
PLD4
PPP1R14B
MZB1
PC_14
SIAH2
ARL4A
CHI3L2
MIS18A
MAPK1IP1L
HMGB2
TYW5
NFKBIA
FOSL2
RGS1
TYMS
NEXN
ID3
PRPS2
TMEM87A
PPP1R14B
CES4A
TMEM165
LILRA4
PLBD1
XCL2
PLD4
RCHY1
MZB1
ITM2C
TIMMDC1
IRF7
TAF1
IGJ
XCL1
PC_15
XCL2
XCL1
ATXN7L3B
FKBP5
PI4KB
YIPF3
HBP1
SEPHS2
TIGIT
ECHDC1
KLF6
PNTA
PCNX
ETS1
GIT2
ADSL
HPRT1
HSD17B11
BAZ2A
RABL6
MRPL14
RCL1
TYMS
NCR3
FAM43A
GYG1
MGST2
PCNA
COMMD5
APOBEC3H

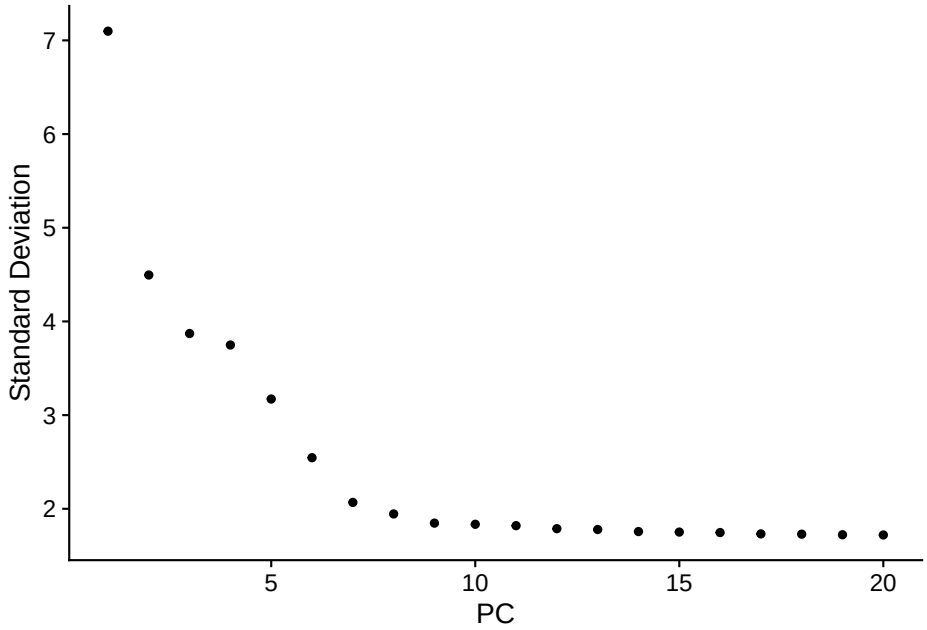
8.2 Determine the 'dimensionality' of the dataset

To overcome the extensive technical noise in any single feature for scRNA-seq data, Seurat clusters cells based on their PCA scores, with each PC essentially representing a 'metafeature' that combines information across a correlated feature set. The top principal components therefore represent a robust compression of the dataset. However, how many components should we choose to include? 10? 20? 100?

Note: The Seurat developers suggest using a JackStraw resampling test to determine this. See Macosko *et al*, and the original pbmc3 vignette. We're going to use an Elbow Plot instead here, because its much quicker.

An alternative heuristic method generates an 'Elbow plot': a ranking of principle components based on the percentage of variance explained by each one (`ElbowPlot()` function). In this example, we can observe an 'elbow' around PC9-10, suggesting that the majority of true signal is captured in the first 10 PCs.

```
ElbowPlot(pbmc)
```



Identifying the true dimensionality of a dataset – can be challenging/uncertain for the user. We therefore suggest these three approaches to consider. The first

is more supervised, exploring PCs to determine relevant sources of heterogeneity, and could be used in conjunction with GSEA for example. The second implements a statistical test based on a random null model, but is time-consuming for large datasets, and may not return a clear PC cutoff. The third is a heuristic that is commonly used, and can be calculated instantly. In this example, all three approaches yielded similar results, but we might have been justified in choosing anything between PC 7-12 as a cutoff.

We chose 10 here, but encourage users to consider the following:

- Dendritic cell and NK aficionados may recognize that genes strongly associated with PCs 12 and 13 define rare immune subsets (i.e. MZB1 is a marker for plasmacytoid DCs). However, these groups are so rare, they are difficult to distinguish from background noise for a dataset of this size without prior knowledge.
- We encourage users to repeat downstream analyses with a different number of PCs (10, 15, or even 50!). As you will observe, the results often do not differ dramatically.
- We advise users to err on the higher side when choosing this parameter. For example, performing downstream analyses with only 5 PCs does significantly and adversely affect results.

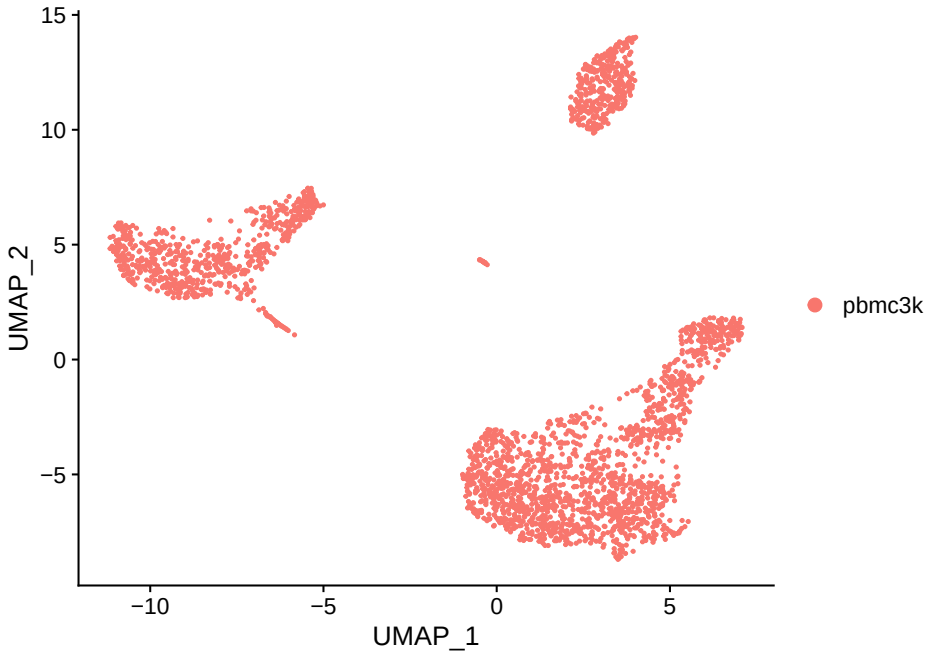
8.3 Run non-linear dimensional reduction (UMAP/tSNE)

Seurat offers several non-linear dimensional reduction techniques, such as tSNE and UMAP, to visualize and explore these datasets. The goal of these algorithms is to learn the underlying manifold of the data in order to place similar cells together in low-dimensional space. Cells within the graph-based clusters determined above should co-localize on these dimension reduction plots. As input to the UMAP and tSNE, we suggest using the same PCs as input to the clustering analysis.

```
# If you haven't installed UMAP, you can do so via reticulate::py_install(packages = "umap-learn"
pbmc <- RunUMAP(pbmc, dims = 1:10)
#> Warning: The default method for RunUMAP has changed from calling Python UMAP via reticulate to
#> To use Python UMAP via reticulate, set umap.method to 'umap-learn' and metric to 'correlation
#> This message will be shown once per session
#> 05:58:11 UMAP embedding parameters a = 0.9922 b = 1.112
#> 05:58:11 Read 2638 rows and found 10 numeric columns
#> 05:58:11 Using Annoy for neighbor search, n_neighbors = 30
#> 05:58:11 Building Annoy index with metric = cosine, n_trees = 50
#> 0%   10   20   30   40   50   60   70   80   90   100%
#> [----|----|----|----|----|----|----|----|----|----|
```

```
#> ************************************************|
#> 05:58:11 Writing NN index file to temp file /var/folders/my/cwpwt_rd1znflqb8gm8py41
#> 05:58:11 Searching Annoy index using 1 thread, search_k = 3000
#> 05:58:11 Annoy recall = 100%
#> 05:58:12 Commencing smooth kNN distance calibration using 1 thread with target n_ne
#> 05:58:12 Initializing from normalized Laplacian + noise (using irlba)
#> 05:58:12 Commencing optimization for 500 epochs, with 105140 positive edges
#> 05:58:14 Optimization finished
```

```
# note that you can set `label = TRUE` or use the LabelClusters function to help label
DimPlot(pbmc, reduction = 'umap')
```



Challenge: PC genes

You can plot gene expression on the UMAP with the `FeaturePlot()` function.

Try out some genes that were highly weighted in the principal component analysis. How do they look?

8.4 Save

You can save the object at this point so that it can easily be loaded back in without having to rerun the computationally intensive steps performed above,

or easily shared with collaborators.

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

Chapter 9

Clustering

Why do we need to do this?

Clustering the cells will allow you to visualise the variability of your data, can help to segregate cells into cell types.

9.1 Cluster cells

Seurat v3 applies a graph-based clustering approach, building upon initial strategies in (Macosko *et al*). Importantly, the *distance metric* which drives the clustering analysis (based on previously identified PCs) remains the same. However, our approach to partitioning the cellular distance matrix into clusters has dramatically improved. Our approach was heavily inspired by recent manuscripts which applied graph-based clustering approaches to scRNA-seq data [SNN-Cliq, Xu and Su, Bioinformatics, 2015] and CyTOF data [PhenoGraph, Levine *et al.*, Cell, 2015]. Briefly, these methods embed cells in a graph structure - for example a K-nearest neighbor (KNN) graph, with edges drawn between cells with similar feature expression patterns, and then attempt to partition this graph into highly interconnected 'quasi-cliques' or 'communities'.

As in PhenoGraph, we first construct a KNN graph based on the euclidean distance in PCA space, and refine the edge weights between any two cells based on the shared overlap in their local neighborhoods (Jaccard similarity). This step is performed using the `FindNeighbors()` function, and takes as input the previously defined dimensionality of the dataset (first 10 PCs).

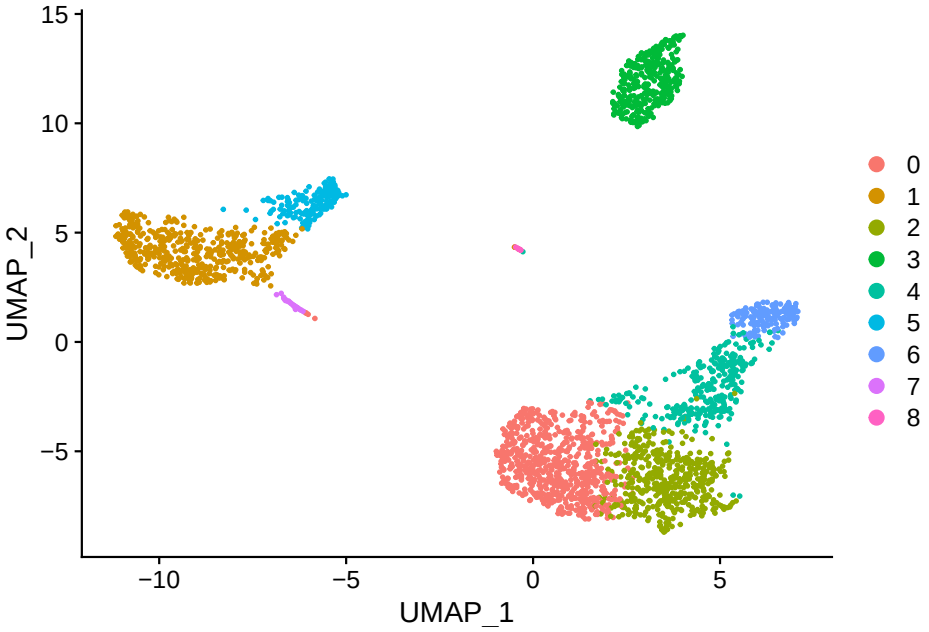
To cluster the cells, we next apply modularity optimization techniques such as the Louvain algorithm (default) or SLM [SLM, Blondel *et al.*, Journal of Statistical Mechanics], to iteratively group cells together, with the goal of optimizing the standard modularity function. The `FindClusters()` function implements

this procedure, and contains a resolution parameter that sets the 'granularity' of the downstream clustering, with increased values leading to a greater number of clusters. We find that setting this parameter between 0.4-1.2 typically returns good results for single-cell datasets of around 3K cells. Optimal resolution often increases for larger datasets. The clusters can be found using the `Idents()` function.

```
pbmc <- FindNeighbors(pbmc, dims = 1:10)
#> Computing nearest neighbor graph
#> Computing SNN
pbmc <- FindClusters(pbmc, resolution = 0.5)
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.8728
#> Number of communities: 9
#> Elapsed time: 0 seconds
# Look at cluster IDs of the first 5 cells
head(Idents(pbmc), 5)
#> AAACATACAACCAC-1 AAACATTGAGCTAC-1 AAACATTGATCAGC-1
#>                2                3                2
#> AAACCGTGCTTCCG-1 AAACCGTGTATGCG-1
#>                1                6
#> Levels: 0 1 2 3 4 5 6 7 8
```

Check out the clusters.

```
DimPlot(pbmc)
```




```
# Equivalent to
# DimPlot(pbmc,reduction="umap", group.by="seurat_clusters")
# DimPlot(pbmc,reduction="umap", group.by="RNA_snn_res.0.5")
```

Challenge: Try different cluster settings

Run `FindNeighbours` and `FindClusters` again, with a different number of dimensions or with a different resolution. Examine the resulting clusters using `DimPlot`.

To maintain the flow of this tutorial, please put the output of this exploration in a different variable, such as `pbmc2`!

9.2 Choosing a cluster resolution

Its a good idea to try different resolutions when clustering to identify the variability of your data.

```
resolution = 2
pbmc <- FindClusters(object = pbmc, reduction = "umap", resolution = seq(0.1, resolution, 0.1),
    dims = 1:10)
#> Warning: The following arguments are not used: reduction,
#> dims
```

```
#> Warning: The following arguments are not used: reduction,
#> dims
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.9623
#> Number of communities: 4
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.9346
#> Number of communities: 7
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.9091
#> Number of communities: 7
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.8890
#> Number of communities: 9
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.8728
```

```
#> Number of communities: 9
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.8564
#> Number of communities: 10
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.8411
#> Number of communities: 10
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.8281
#> Number of communities: 11
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.8159
#> Number of communities: 11
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.8036
```

```
#> Number of communities: 11
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.7918
#> Number of communities: 11
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.7798
#> Number of communities: 12
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.7678
#> Number of communities: 13
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.7575
#> Number of communities: 13
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.7473
```

```
#> Number of communities: 13
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.7370
#> Number of communities: 15
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.7280
#> Number of communities: 15
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.7185
#> Number of communities: 15
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.7093
#> Number of communities: 15
#> Elapsed time: 0 seconds
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 2638
#> Number of edges: 95927
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.7002
```

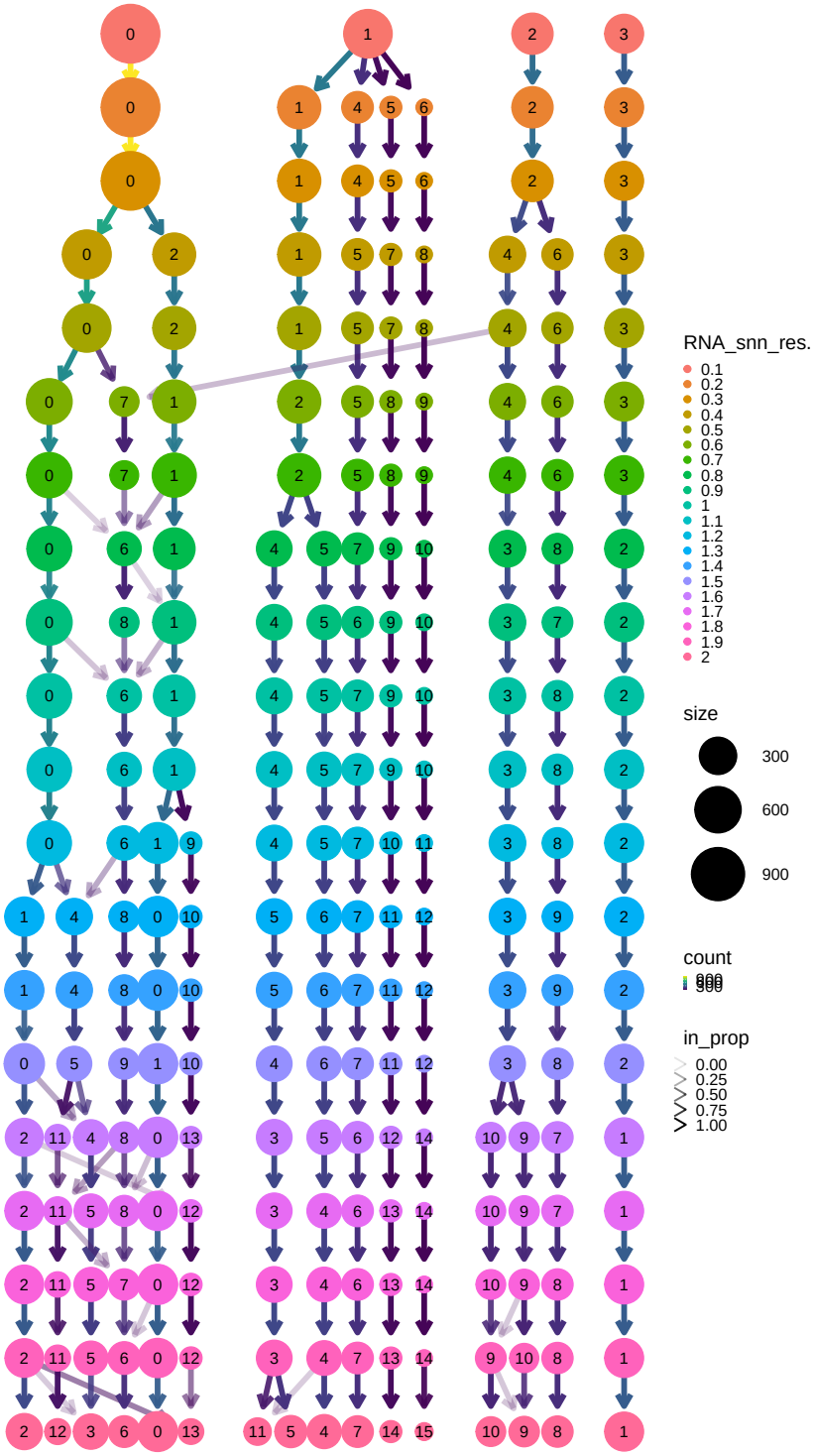
```
#> Number of communities: 16
#> Elapsed time: 0 seconds

# the different clustering created
names(pbmc@meta.data)
#>  [1] "orig.ident"      "nCount_RNA"      "nFeature_RNA"
#>  [4] "percent.mt"      "RNA_snn_res.0.5" "seurat_clusters"
#>  [7] "RNA_snn_res.0.1" "RNA_snn_res.0.2" "RNA_snn_res.0.3"
#> [10] "RNA_snn_res.0.4" "RNA_snn_res.0.6" "RNA_snn_res.0.7"
#> [13] "RNA_snn_res.0.8" "RNA_snn_res.0.9" "RNA_snn_res.1"
#> [16] "RNA_snn_res.1.1" "RNA_snn_res.1.2" "RNA_snn_res.1.3"
#> [19] "RNA_snn_res.1.4" "RNA_snn_res.1.5" "RNA_snn_res.1.6"
#> [22] "RNA_snn_res.1.7" "RNA_snn_res.1.8" "RNA_snn_res.1.9"
#> [25] "RNA_snn_res.2"

# Look at cluster IDs of the first 5 cells
head(Idents(pbmc), 5)
#> AAACATACAACCAC-1 AAACATTGAGCTAC-1 AAACATTGATCAGC-1
#>                6                1                0
#> AAACCGTGCTTCCG-1 AAACCGTGTATGCG-1
#>                5                8
#> Levels: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Plot a clustree to decide how many clusters you have and what resolution capture them.

```
library(clustree)
#> Loading required package: ggraph
clustree(pbmc, prefix = "RNA_snn_res.") + theme(legend.key.size = unit(0.05, "cm"))
```

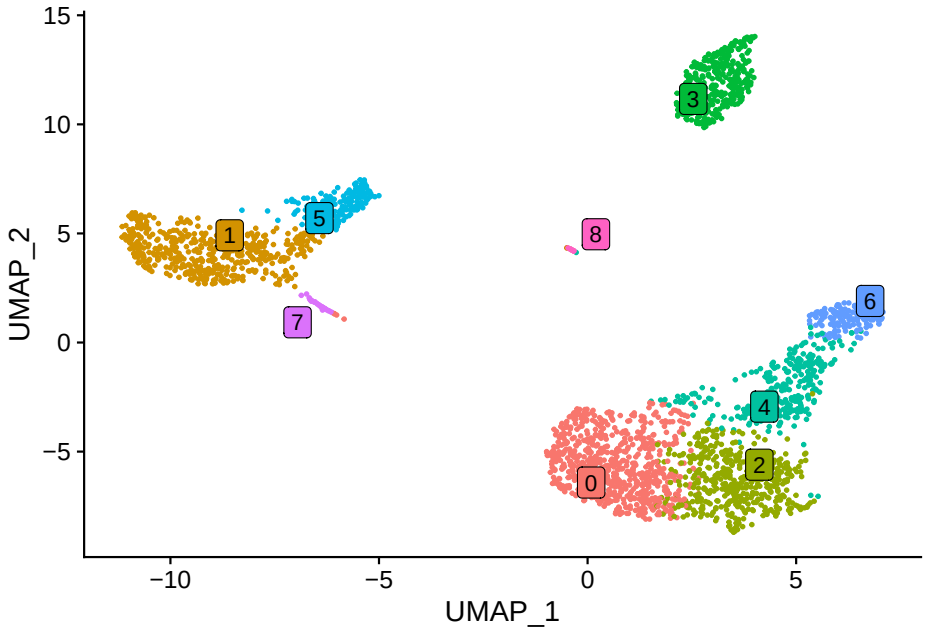

RNA_snn_res.
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
2
size
300
600
900
count
in_prop
0.00
0.25
0.50
0.75
1.00

Name cells with the corresponding cluster name at the resolution you pick. This case we are happy with 0.5.

```
# The name of the cluster is prefixed with 'RNA_snn_res' and the number of the resolut
Idents(pbmc) <- pbmc$RNA_snn_res.0.5
```

Plot the UMAP with colored clusters with Dimplot

```
DimPlot(pbmc, label = TRUE, repel = TRUE, label.box = TRUE) + NoLegend()
```



Chapter 10

Cluster Markers

Why do we need to do this?

Single cell data helps to segragate cell types. Use markers to identify cell types. warning: In this example the cell types/markers are well known and making this step easy, but in reality this step needs the experts curation.

10.1 Finding differentially expressed features (cluster biomarkers)

Seurat can help you find markers that define clusters via differential expression. By default, it identifies positive and negative markers of a single cluster (specified in `ident.1`), compared to all other cells. `FindAllMarkers()` automates this process for all clusters, but you can also test groups of clusters vs. each other, or against all cells.

The `min.pct` argument requires a feature to be detected at a minimum percentage in either of the two groups of cells, and the thresh.test argument requires a feature to be differentially expressed (on average) by some amount between the two groups. You can set both of these to 0, but with a dramatic increase in time - since this will test a large number of features that are unlikely to be highly discriminatory. As another option to speed up these computations, `max.cells.per.ident` can be set. This will downsample each identity class to have no more cells than whatever this is set to. While there is generally going to be a loss in power, the speed increases can be significant and the most highly differentially expressed features will likely still rise to the top.

```
# find all markers of cluster 2
cluster2.markers <- FindMarkers(pbmc, ident.1 = 2, min.pct = 0.25)
```

```
head(cluster2.markers, n = 5)
#>             p_val avg_log2FC pct.1 pct.2    p_val_adj
#> IL32 2.892340e-90  1.2013522 0.947 0.465 3.966555e-86
#> LTB  1.060121e-86  1.2695776 0.981 0.643 1.453850e-82
#> CD3D 8.794641e-71  0.9389621 0.922 0.432 1.206097e-66
#> IL7R 3.516098e-68  1.1873213 0.750 0.326 4.821977e-64
#> LDHB 1.642480e-67  0.8969774 0.954 0.614 2.252497e-63
# find all markers distinguishing cluster 5 from clusters 0 and 3
cluster5.markers <- FindMarkers(pbmc, ident.1 = 5, ident.2 = c(0, 3), min.pct = 0.25)
head(cluster5.markers, n = 5)
#>                       p_val avg_log2FC pct.1 pct.2
#> FCGR3A        8.246578e-205   4.261495 0.975 0.040
#> IFITM3        1.677613e-195   3.879339 0.975 0.049
#> CFD           2.401156e-193   3.405492 0.938 0.038
#> CD68          2.900384e-191   3.020484 0.926 0.035
#> RP11-290F20.3 2.513244e-186   2.720057 0.840 0.017
#>                     p_val_adj
#> FCGR3A        1.130936e-200
#> IFITM3        2.300678e-191
#> CFD           3.292945e-189
#> CD68          3.977587e-187
#> RP11-290F20.3 3.446663e-182
# find markers for every cluster compared to all remaining cells, report only the posi
pbmc.markers <- FindAllMarkers(pbmc, only.pos = TRUE, min.pct = 0.25, logfc.threshold =
#> Calculating cluster 0
#> Calculating cluster 1
#> Calculating cluster 2
#> Calculating cluster 3
#> Calculating cluster 4
#> Calculating cluster 5
#> Calculating cluster 6
#> Calculating cluster 7
#> Calculating cluster 8
pbmc.markers %>% group_by(cluster) %>% slice_max(n = 2, order_by = avg_log2FC)
#> # A tibble: 18 x 7
#> # Groups:   cluster [9]
#>        p_val avg_log2FC pct.1 pct.2 p_val_adj cluster gene
#>        <dbl>      <dbl> <dbl> <dbl>     <dbl> <fct>   <chr>
#>  1 9.57e- 88       1.36 0.447 0.108 1.31e- 83 0       CCR7
#>  2 3.75e-112       1.09 0.912 0.592 5.14e-108 0       LDHB
#>  3 0               5.57 0.996 0.215 0         1       S100A9
#>  4 0               5.48 0.975 0.121 0         1       S100A8
#>  5 1.06e- 86       1.27 0.981 0.643 1.45e- 82 2       LTB
#>  6 2.97e- 58       1.23 0.42  0.111 4.07e- 54 2       AQP3
#>  7 0               4.31 0.936 0.041 0         3       CD79A
```

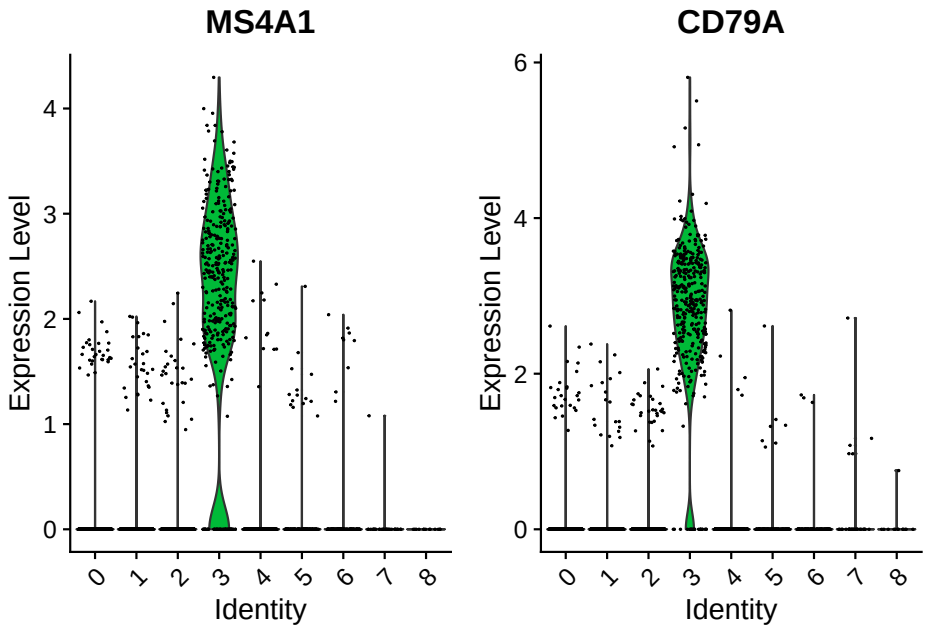
```
#>  8 9.48e-271       3.59 0.622 0.022 1.30e-266 3       TCL1A
#>  9 5.61e-202       3.10 0.983 0.234 7.70e-198 4       CCL5
#> 10 7.25e-165       3.00 0.577 0.055 9.95e-161 4       GZMK
#> 11 3.51e-184       3.31 0.975 0.134 4.82e-180 5       FCGR3A
#> 12 2.03e-125       3.09 1     0.315 2.78e-121 5       LST1
#> 13 3.13e-191       5.32 0.961 0.131 4.30e-187 6       GNLY
#> 14 7.95e-269       4.83 0.961 0.068 1.09e-264 6       GZMB
#> 15 1.48e-220       3.87 0.812 0.011 2.03e-216 7       FCER1A
#> 16 1.67e- 21       2.87 1     0.513 2.28e- 17 7       HLA-D~
#> 17 1.92e-102       8.59 1     0.024 2.63e- 98 8       PPBP
#> 18 9.25e-186       7.29 1     0.011 1.27e-181 8       PF4
```

Seurat has several tests for differential expression which can be set with the test.use parameter (see our DE vignette for details). For example, the ROC test returns the 'classification power' `abs(AUC-0.5)*2` for any individual marker, ranging from 0 = random to 1 = perfect.

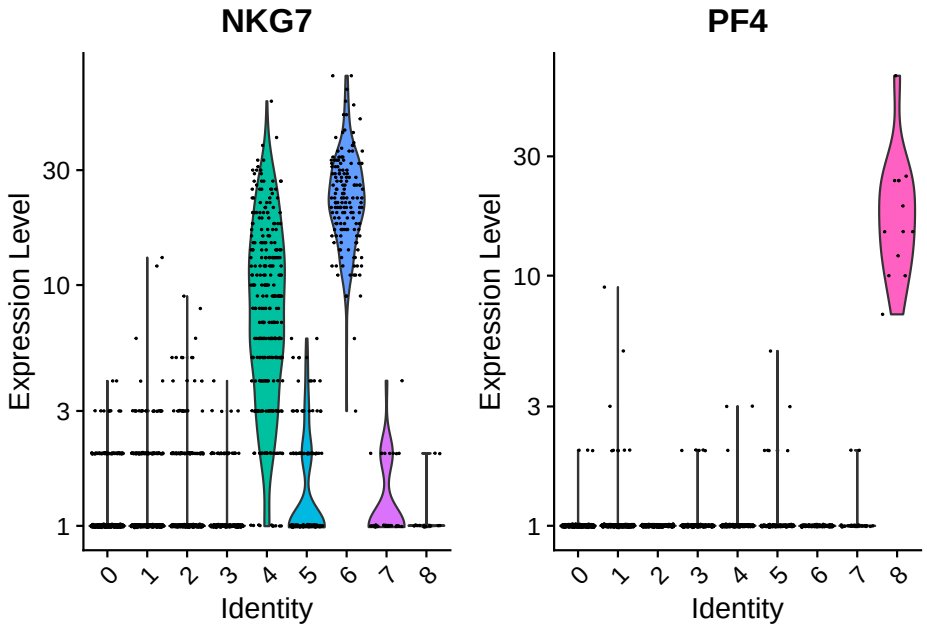
```
cluster0.markers <- FindMarkers(pbmc, ident.1 = 0, logfc.threshold = 0.25, test.use = "roc", only
```

We include several tools for visualizing marker expression. `VlnPlot()` (shows expression probability distributions across clusters), and `FeaturePlot()` (visualizes feature expression on a tSNE or PCA plot) are our most commonly used visualizations. We also suggest exploring `RidgePlot()`, `CellScatter()`, and `DotPlot()` as additional methods to view your dataset.

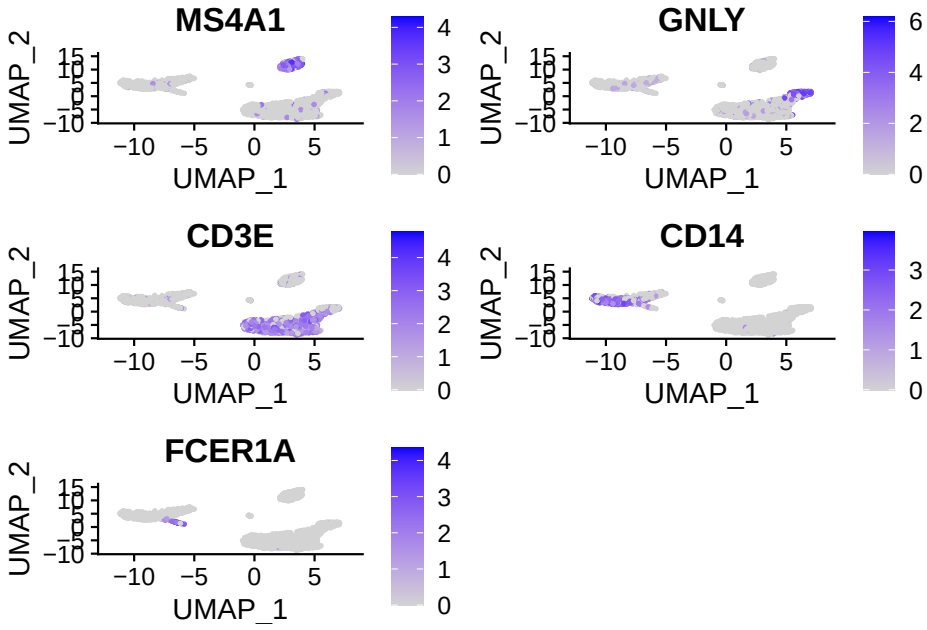
```
VlnPlot(pbmc, features = c("MS4A1", "CD79A"))
```



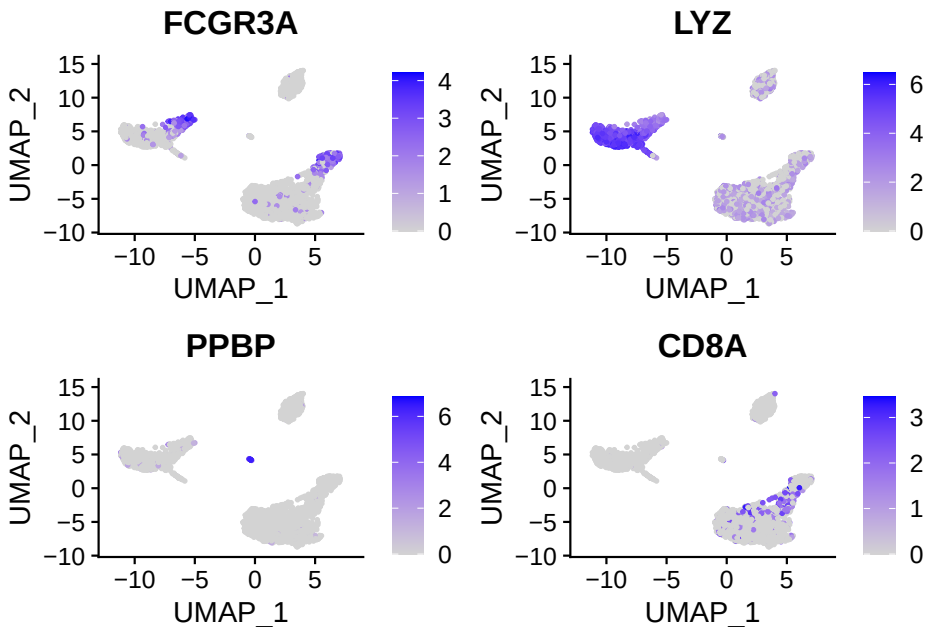
```
# you can plot raw counts as well
VlnPlot(pbmc, features = c("NKG7", "PF4"), slot = 'counts', log = TRUE)
```




```
FeaturePlot(pbmc, features = c("MS4A1", "GNLY", "CD3E", "CD14", "FCER1A"))
```



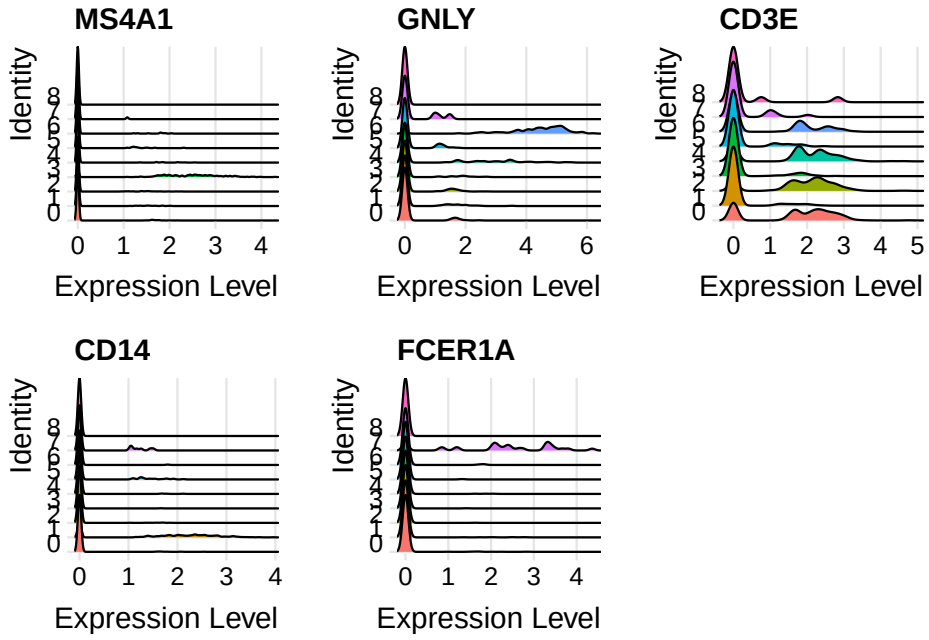
```
FeaturePlot(pbmc, features = c("FCGR3A", "LYZ", "PPBP", "CD8A"))
```



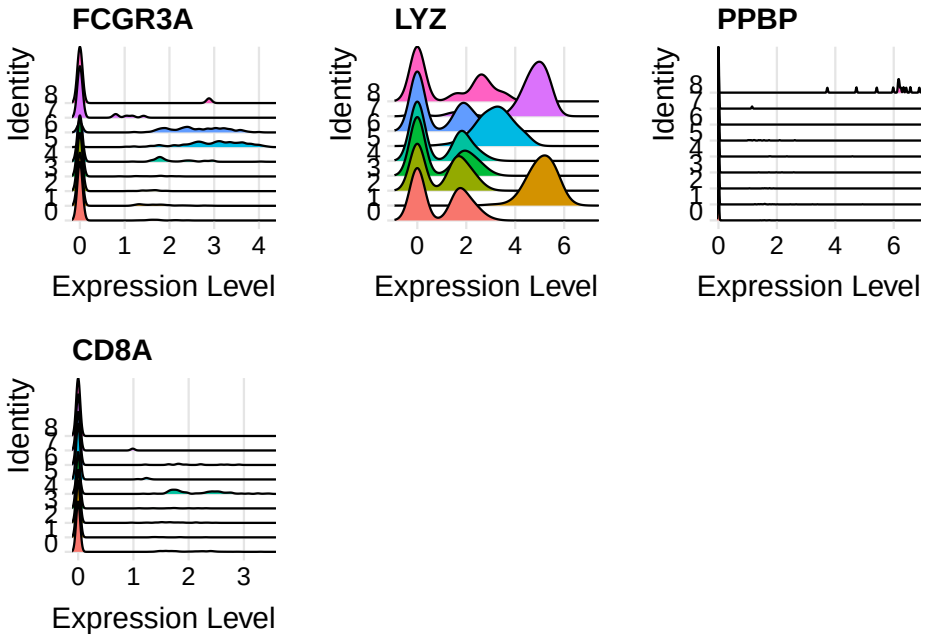
Other useful plots

These are ridgeplots, cell scatter plots and dotplots. Replace `FeaturePlot` with the other functions.

```
RidgePlot(pbmc, features = c("MS4A1", "GNLY", "CD3E", "CD14", "FCER1A"))
#> Picking joint bandwidth of 0.0236
#> Picking joint bandwidth of 0.0859
#> Picking joint bandwidth of 0.126
#> Picking joint bandwidth of 0.0337
#> Picking joint bandwidth of 0.0659
```




```
RidgePlot(pbmc, features = c("FCGR3A", "LYZ", "PPBP", "CD8A"))
#> Picking joint bandwidth of 0.0582
#> Picking joint bandwidth of 0.309
#> Picking joint bandwidth of 0.0156
#> Picking joint bandwidth of 0.0366
```

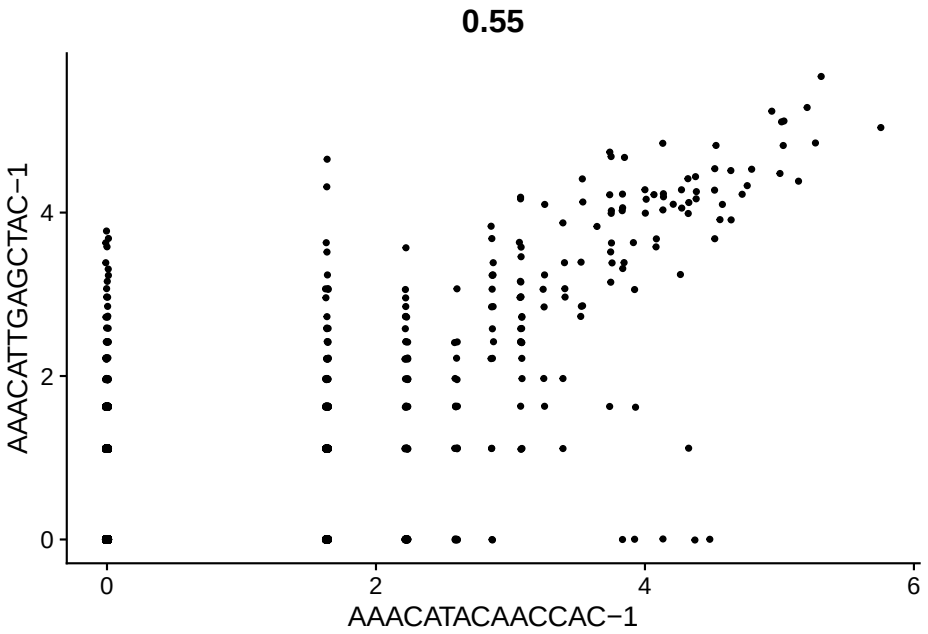


For CellScatter plots, will need the cell id of the cells you want to look at. You can get this from the cell metadata (`pbmc@meta.data`).

```
head( pbmc@meta.data )
#>                  orig.ident nCount_RNA nFeature_RNA
#> AAACATACAACCAC-1     pbmc3k       2419          779
#> AAACATTGAGCTAC-1     pbmc3k       4903         1352
#> AAACATTGATCAGC-1     pbmc3k       3147         1129
#> AAACCGTGCTTCCG-1     pbmc3k       2639          960
#> AAACCGTGTATGCG-1     pbmc3k        980          521
#> AAACGCACTGGTAC-1     pbmc3k       2163          781
#>                  percent.mt RNA_snn_res.0.5 seurat_clusters
#> AAACATACAACCAC-1  3.0177759               2               6
#> AAACATTGAGCTAC-1  3.7935958               3               1
#> AAACATTGATCAGC-1  0.8897363               2               0
#> AAACCGTGCTTCCG-1  1.7430845               1               5
#> AAACCGTGTATGCG-1  1.2244898               6               8
#> AAACGCACTGGTAC-1  1.6643551               2               0
#>                  RNA_snn_res.0.1 RNA_snn_res.0.2
#> AAACATACAACCAC-1               0               0
#> AAACATTGAGCTAC-1               3               3
#> AAACATTGATCAGC-1               0               0
#> AAACCGTGCTTCCG-1               1               1
#> AAACCGTGTATGCG-1               2               2
#> AAACGCACTGGTAC-1               0               0
#>                  RNA_snn_res.0.3 RNA_snn_res.0.4
```

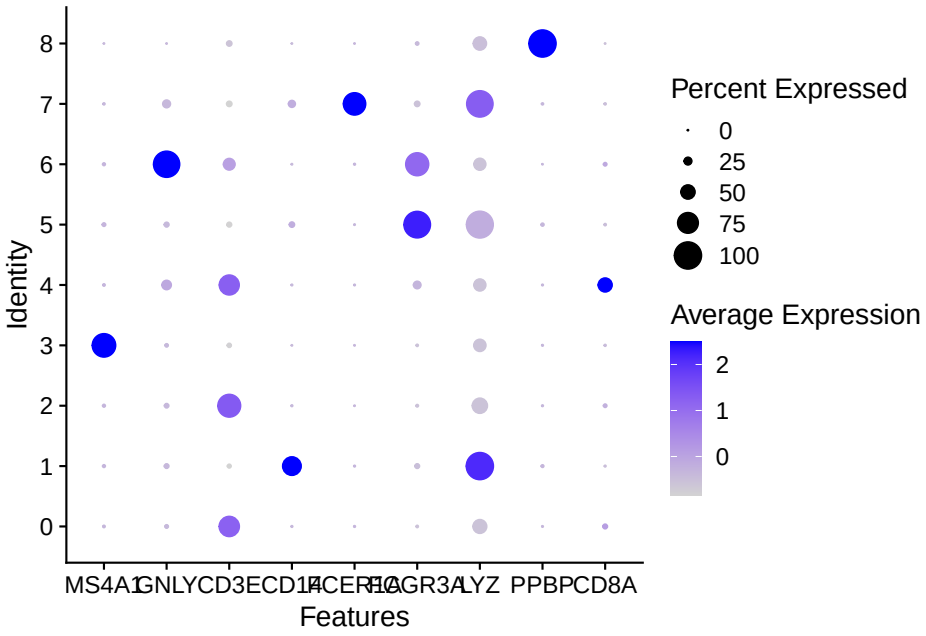
```
#> AAACATACAACCAC-1               0               2
#> AAACATTGAGCTAC-1               3               3
#> AAACATTGATCAGC-1               0               2
#> AAACCGTGCTTCCG-1               1               1
#> AAACCGTGTATGCG-1               2               6
#> AAACGCACTGGTAC-1               0               2
#>                  RNA_snn_res.0.6 RNA_snn_res.0.7
#> AAACATACAACCAC-1               1               1
#> AAACATTGAGCTAC-1               3               3
#> AAACATTGATCAGC-1               1               1
#> AAACCGTGCTTCCG-1               2               2
#> AAACCGTGTATGCG-1               6               6
#> AAACGCACTGGTAC-1               1               1
#>                  RNA_snn_res.0.8 RNA_snn_res.0.9
#> AAACATACAACCAC-1               6               1
#> AAACATTGAGCTAC-1               2               2
#> AAACATTGATCAGC-1               1               1
#> AAACCGTGCTTCCG-1               4               4
#> AAACCGTGTATGCG-1               8               7
#> AAACGCACTGGTAC-1               1               1
#>                  RNA_snn_res.1 RNA_snn_res.1.1
#> AAACATACAACCAC-1             6               6
#> AAACATTGAGCTAC-1             2               2
#> AAACATTGATCAGC-1             1               1
#> AAACCGTGCTTCCG-1             4               4
#> AAACCGTGTATGCG-1             8               8
#> AAACGCACTGGTAC-1             1               1
#>                  RNA_snn_res.1.2 RNA_snn_res.1.3
#> AAACATACAACCAC-1               6               8
#> AAACATTGAGCTAC-1               2               2
#> AAACATTGATCAGC-1               1               0
#> AAACCGTGCTTCCG-1               4               5
#> AAACCGTGTATGCG-1               8               9
#> AAACGCACTGGTAC-1               1               0
#>                  RNA_snn_res.1.4 RNA_snn_res.1.5
#> AAACATACAACCAC-1               8               9
#> AAACATTGAGCTAC-1               2               2
#> AAACATTGATCAGC-1               0               1
#> AAACCGTGCTTCCG-1               5               4
#> AAACCGTGTATGCG-1               9               8
#> AAACGCACTGGTAC-1               0               1
#>                  RNA_snn_res.1.6 RNA_snn_res.1.7
#> AAACATACAACCAC-1               8               8
#> AAACATTGAGCTAC-1               1               1
#> AAACATTGATCAGC-1               0               0
```

```
#> AAACCGTGCTTCCG-1               3               3
#> AAACCGTGTATGCG-1               7               7
#> AAACGCACTGGTAC-1               0               0
#>                  RNA_snn_res.1.8 RNA_snn_res.1.9
#> AAACATACAACCAC-1               7               6
#> AAACATTGAGCTAC-1               1               1
#> AAACATTGATCAGC-1               0               0
#> AAACCGTGCTTCCG-1               3               3
#> AAACCGTGTATGCG-1               8               8
#> AAACGCACTGGTAC-1               0               0
#>                  RNA_snn_res.2
#> AAACATACAACCAC-1             6
#> AAACATTGAGCTAC-1             1
#> AAACATTGATCAGC-1             0
#> AAACCGTGCTTCCG-1             5
#> AAACCGTGTATGCG-1             8
#> AAACGCACTGGTAC-1             0
CellScatter(pbmc, cell1 = "AAACATACAACCAC-1", cell2 = "AAACATTGAGCTAC-1")
```



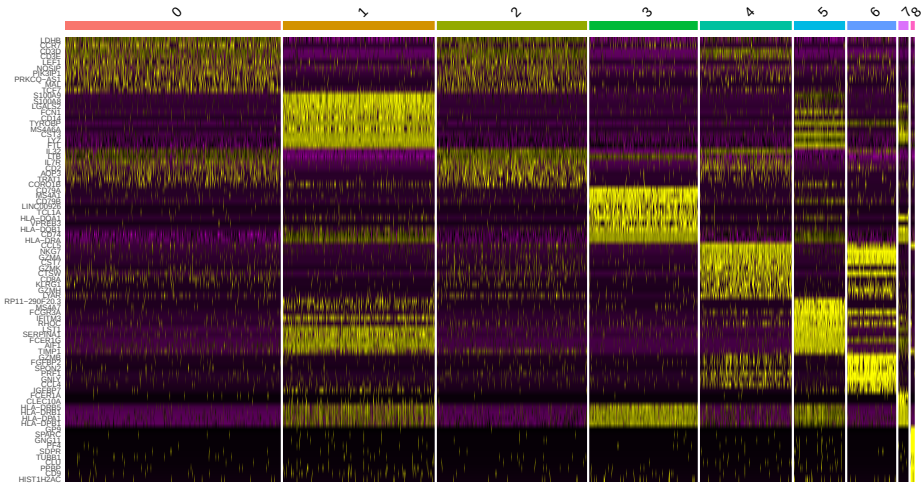
DotPlots

```
DotPlot(pbmc, features = c("MS4A1", "GNLY", "CD3E", "CD14", "FCER1A", "FCGR3A", "LYZ", "PPBP", "C
```



`DoHeatmap()` generates an expression heatmap for given cells and features. In this case, we are plotting the top 10 markers (or all markers if less than 10) for each cluster.

```
top10 <- pbmc.markers %>% group_by(cluster) %>% top_n(n = 10, wt = avg_log2FC)
DoHeatmap(pbmc, features = top10$gene) + NoLegend()
```



10.2 Use makers to label or find a cluster

If you know markers for your cell types, use AddModuleScore to label them.

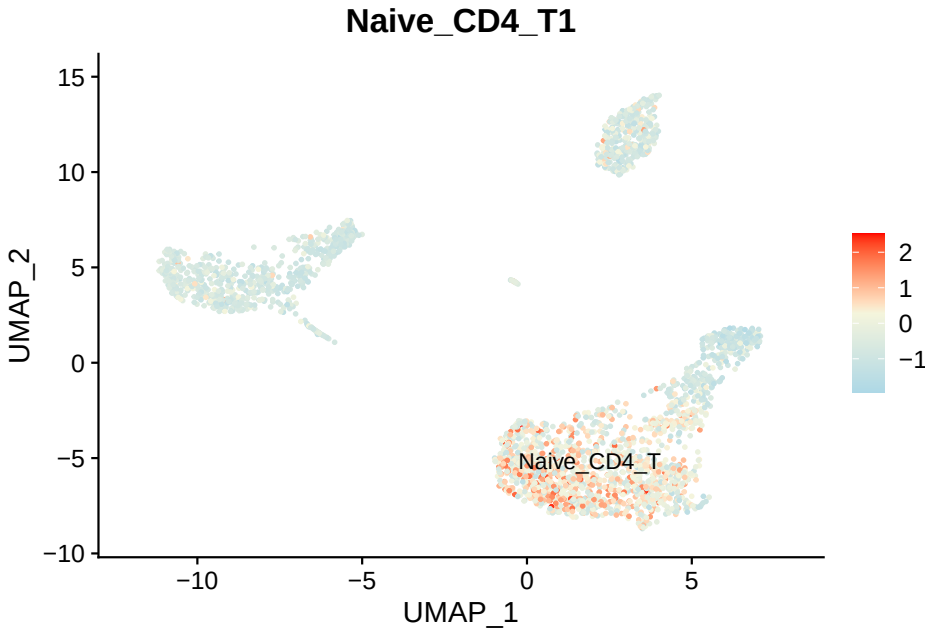
```
genes_markers <- list(Naive_CD4_T = c("IL7R", "CCR7"))

pbmc <- AddModuleScore(object = pbmc, features = genes_markers, ctrl = 5, name = "Naive_CD4_T",
    search = TRUE)


# notice the name of the cluster has a 1 at the end
names(pbmc@meta.data)
#>  [1] "orig.ident"      "nCount_RNA"      "nFeature_RNA"
#>  [4] "percent.mt"      "RNA_snn_res.0.5" "seurat_clusters"
#>  [7] "RNA_snn_res.0.1" "RNA_snn_res.0.2" "RNA_snn_res.0.3"
#> [10] "RNA_snn_res.0.4" "RNA_snn_res.0.6" "RNA_snn_res.0.7"
#> [13] "RNA_snn_res.0.8" "RNA_snn_res.0.9" "RNA_snn_res.1"
#> [16] "RNA_snn_res.1.1" "RNA_snn_res.1.2" "RNA_snn_res.1.3"
#> [19] "RNA_snn_res.1.4" "RNA_snn_res.1.5" "RNA_snn_res.1.6"
#> [22] "RNA_snn_res.1.7" "RNA_snn_res.1.8" "RNA_snn_res.1.9"
#> [25] "RNA_snn_res.2"   "Naive_CD4_T1"

# label that cell type
pbmc$cell_label = NA
pbmc$cell_label[pbmc$Naive_CD4_T1 > 1] = "Naive_CD4_T"
Idents(pbmc) = pbmc$cell_label

# plot
# Using a custom colour scale
FeaturePlot(pbmc, features = "Naive_CD4_T1", label = TRUE, repel = TRUE, ) + scale_colour_gradien
#> Scale for colour is already present.
#> Adding another scale for colour, which will replace the
#> existing scale.
```

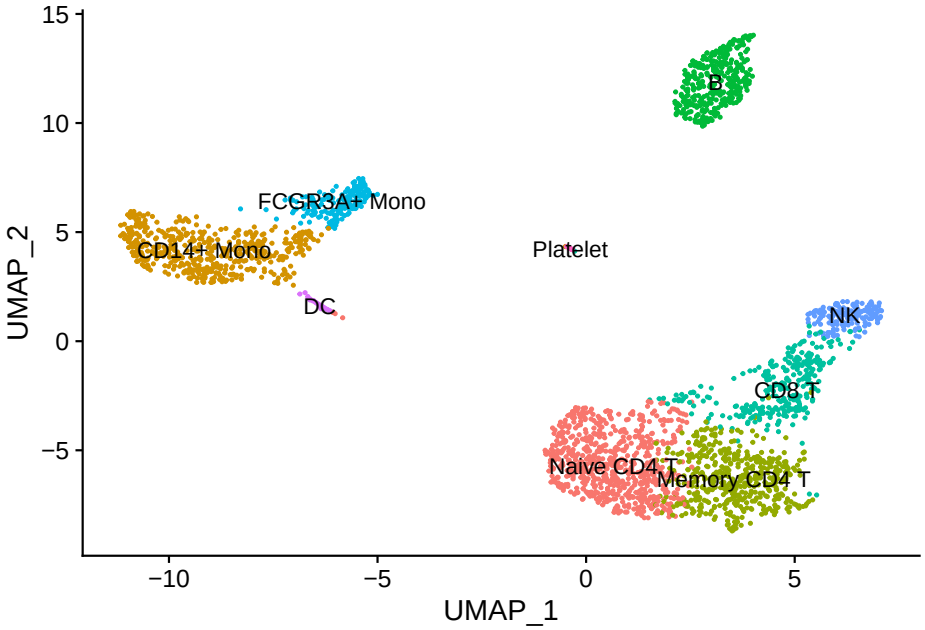


10.3 Assigning cell type identity to clusters

Fortunately in the case of this dataset, we can use canonical markers to easily match the unbiased clustering to known cell types:

Cluster ID	Markers	Cell Type
0	IL7R, CCR7	Naive CD4+ T
1	CD14, LYZ	CD14+ Mono
2	IL7R, S100A4	Memory CD4+
3	MS4A1	B
4	CD8A	CD8+ T
5	FCGR3A, MS4A7	FCGR3A+ Mono
6	GNLY, NKG7	NK
7	FCER1A, CST3	DC
8	PPBP	Platelet

```
Idents(pbmc) <- pbmc$RNA_snn_res.0.5
new.cluster.ids <- c("Naive CD4 T", "CD14+ Mono", "Memory CD4 T", "B", "CD8 T", "FCGR3A
names(new.cluster.ids) <- levels(pbmc)
pbmc <- RenameIdents(pbmc, new.cluster.ids)
DimPlot(pbmc, reduction = 'umap', label = TRUE, pt.size = 0.5) + NoLegend()
```




```
saveRDS(pbmc, file = "pbmc3k_final.rds")
```

Part III

Futher Analysis

Chapter 11

SingleR

```
#install.packages("BiocManager")
#BiocManager::install(c("SingleCellExperiment","SingleR","celldex"),ask=F)
library(SingleCellExperiment)
library(SingleR)
library(celldex)
```

In this workshop we have focused on the Seurat package. However, there is another whole ecosystem of R packages for single cell analysis within Bioconductor. We won't go into any detail on these packages in this workshop, but there is good material describing the object type online : OSCA.

For now, we'll just convert our Seurat object into an object called SingleCellExperiment. Some popular packages from Bioconductor that work with this type are Slingshot, Scran, Scater.

```
sce <- as.SingleCellExperiment(pbmc)
sce
#> class: SingleCellExperiment
#> dim: 13714 2638
#> metadata(0):
#> assays(3): counts logcounts scaledata
#> rownames(13714): AL627309.1 AP006222.2 ... PNRC2.1
#>   SRSF10.1
#> rowData names(0):
#> colnames(2638): AAACATACAACCAC-1 AAACATTGAGCTAC-1 ...
#>   TTTGCATGAGAGGC-1 TTTGCATGCCTCAC-1
#> colData names(28): orig.ident nCount_RNA ...
#>   cell_label ident
#> reducedDimNames(2): PCA UMAP
#> mainExpName: RNA
```

```
#> altExpNames(0):
```

We will now use a package called SingleR to label each cell. SingleR uses a reference data set of cell types with expression data to infer the best label for each cell. A convenient collection of cell type reference is in the `celldex` package which currently contains the follow sets:

```
ls('package:celldex')
#> [1] "BlueprintEncodeData"
#> [2] "DatabaseImmuneCellExpressionData"
#> [3] "HumanPrimaryCellAtlasData"
#> [4] "ImmGenData"
#> [5] "MonacoImmuneData"
#> [6] "MouseRNAseqData"
#> [7] "NovershternHematopoieticData"
```

In this example, we'll use the `HumanPrimaryCellAtlasData` set, which contains high-level, and fine-grained label types. Lets download the reference dataset

```
# This too is a sce object,
# colData is equivalent to seurat's metadata
ref.set <- celldex::HumanPrimaryCellAtlasData()
#> see ?celldex and browseVignettes('celldex') for documentation
#> loading from cache
#> see ?celldex and browseVignettes('celldex') for documentation
#> loading from cache
```

The "main" labels.

```
unique(ref.set$label.main)
#>  [1] "DC"                   "Smooth_muscle_cells"
#>  [3] "Epithelial_cells"     "B_cell"
#>  [5] "Neutrophils"          "T_cells"
#>  [7] "Monocyte"             "Erythroblast"
#>  [9] "BM & Prog."           "Endothelial_cells"
#> [11] "Gametocytes"          "Neurons"
#> [13] "Keratinocytes"        "HSC_-G-CSF"
#> [15] "Macrophage"           "NK_cell"
#> [17] "Embryonic_stem_cells" "Tissue_stem_cells"
#> [19] "Chondrocytes"         "Osteoblasts"
#> [21] "BM"                   "Platelets"
#> [23] "Fibroblasts"          "iPS_cells"
#> [25] "Hepatocytes"          "MSC"
#> [27] "Neuroepithelial_cell" "Astrocyte"
#> [29] "HSC_CD34+"            "CMP"
#> [31] "GMP"                  "MEP"
#> [33] "Myelocyte"            "Pre-B_cell_CD34-"
```

```
#> [35] "Pro-B_cell_CD34+"     "Pro-Myelocyte"
```

An example of the types of "fine" labels.

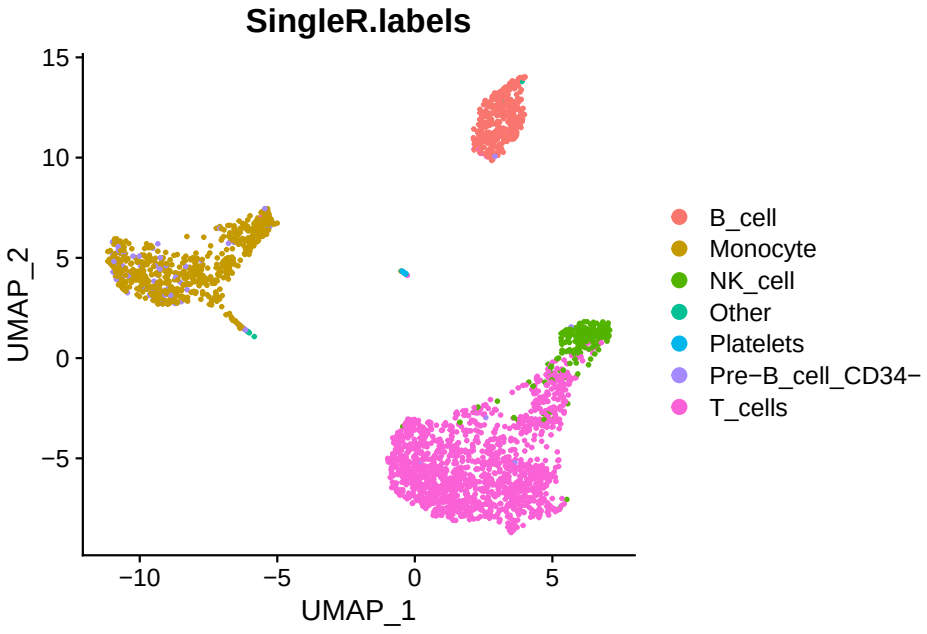
```
head(unique(ref.set$label.fine))
#> [1] "DC:monocyte-derived:immature"
#> [2] "DC:monocyte-derived:Galectin-1"
#> [3] "DC:monocyte-derived:LPS"
#> [4] "DC:monocyte-derived"
#> [5] "Smooth_muscle_cells:bronchial:vit_D"
#> [6] "Smooth_muscle_cells:bronchial"
```

Now we'll label our cells using the SingleCellExperiment object, with the above reference set.

```
pred.cnts <- SingleR::SingleR(test = sce, ref = ref.set, labels = ref.set$label.main)
```

Keep any types that have more than 10 cells to the label, and put those labels back on our Seurat object and plot our on our umap.

```
lbls.keep <- table(pred.cnts$labels)>10
pbmc$SingleR.labels <- ifelse(lbls.keep[pred.cnts$labels], pred.cnts$labels, 'Other')
DimPlot(pbmc, reduction='umap', group.by='SingleR.labels')
```



It is nice to see that even though SingleR does not use the clusters we computed earlier, the labels do seem to match those clusters reasonably well.

Chapter 12

Differential Expression

There are many different methods for calculating differential expression between groups in scRNAseq data. There are a number of review papers worth consulting on this topic.

There is the Seurat differential expression Vignette which walks through the variety implemented in Seurat.

There is also a good discussion of useing pseudobulk approaches which is worth checking out if youre planning differential expression analyses.

We will now look at GSE96583, another PBMC dataset. For speed, we will be looking at a subset of 5000 cells from this data. The cells in this dataset were pooled from eight individual donors. This data contains two batches of single cell sequencing. One of the batches was stimulated with IFN-beta.

The data has already been processed as we have done with the first PBMC dataset, and can be loaded from the `kang2018.rds` file in the data folder.

```
kang <- readRDS("data/kang2018.rds")
head(kang@meta.data)
#>                     orig.ident nCount_RNA nFeature_RNA  ind
#> AGGGCGCTATTTCC-1 SeuratProject       2020          523 1256
#> GGAGACGATTCGTT-1 SeuratProject        840          381 1256
#> CACCGTTGTCGTAG-1 SeuratProject       3097          995 1016
#> TATCGTACACGCAT-1 SeuratProject       1011          540 1488
#> TGACGCCTTGCTTT-1 SeuratProject        570          367  101
#> TACGAGACCTATTC-1 SeuratProject       2399          770 1244
#>                   stim              cell multiplets
#> AGGGCGCTATTTCC-1 stim   CD14+ Monocytes    singlet
#> GGAGACGATTCGTT-1 stim       CD4 T cells    singlet
```

```
#> CACCGTTGTCGTAG-1 ctrl FCGR3A+ Monocytes    singlet
#> TATCGTACACGCAT-1 stim           B cells    singlet
#> TGACGCCTTGCTTT-1 ctrl       CD4 T cells       ambs
#> TACGAGACCTATTC-1 stim       CD4 T cells    singlet
```

How cells from each condition do we have?

```
table(kang$stim)
#>
#> ctrl stim
#> 2449 2551
```

How many cells per individuals per group?

```
table(kang$ind, kang$stim)
#>
#>        ctrl stim
#>   101   178  229
#>   107   117  107
#>   1015  514  496
#>   1016  356  356
#>   1039  100  118
#>   1244  380  313
#>   1256  394  390
#>   1488  410  542
```

And for each sample, how many of each cell type has been classified?

```
table(paste(kang$ind,kang$stim), kang$cell)
#>
#>             B cells CD14+ Monocytes CD4 T cells CD8 T cells
#>   101 ctrl       24              48          61          15
#>   101 stim       30              52          84          17
#>   1015 ctrl      81             149         145          46
#>   1015 stim      68             151         150          22
#>   1016 ctrl      22              72          89         112
#>   1016 stim      29              65          66         115
#>   1039 ctrl       7              35          40           6
#>   1039 stim       7              28          51           6
#>   107 ctrl        9              51          32           6
#>   107 stim        9              35          43           1
#>   1244 ctrl      23              86         206           8
#>   1244 stim      18              58         191           4
#>   1256 ctrl      32              81         180          29
#>   1256 stim      42              70         198          25
#>   1488 ctrl      36              59         247          13
#>   1488 stim      59              59         325          15
```

```
#>
#>             Dendritic cells FCGR3A+ Monocytes
#>   101 ctrl                4                11
#>   101 stim                6                23
#>   1015 ctrl               4                50
#>   1015 stim              17                44
#>   1016 ctrl               4                22
#>   1016 stim               2                32
#>   1039 ctrl               1                 3
#>   1039 stim               1                 8
#>   107 ctrl                3                12
#>   107 stim                2                 5
#>   1244 ctrl               8                19
#>   1244 stim               6                 4
#>   1256 ctrl               6                20
#>   1256 stim               3                11
#>   1488 ctrl               8                25
#>   1488 stim              12                28
#>
#>             Megakaryocytes NK cells
#>   101 ctrl                4        11
#>   101 stim                1        16
#>   1015 ctrl               5        34
#>   1015 stim               5        39
#>   1016 ctrl               4        31
#>   1016 stim               1        46
#>   1039 ctrl               6         1
#>   1039 stim              10         5
#>   107 ctrl                1         3
#>   107 stim                0        12
#>   1244 ctrl               5        25
#>   1244 stim               4        28
#>   1256 ctrl               1        45
#>   1256 stim               8        33
#>   1488 ctrl               4        18
#>   1488 stim               6        38
```

12.1 Prefiltering

Why do we need to do this?

If expression is below a certain level, it will be almost impossible to see any differential expression.

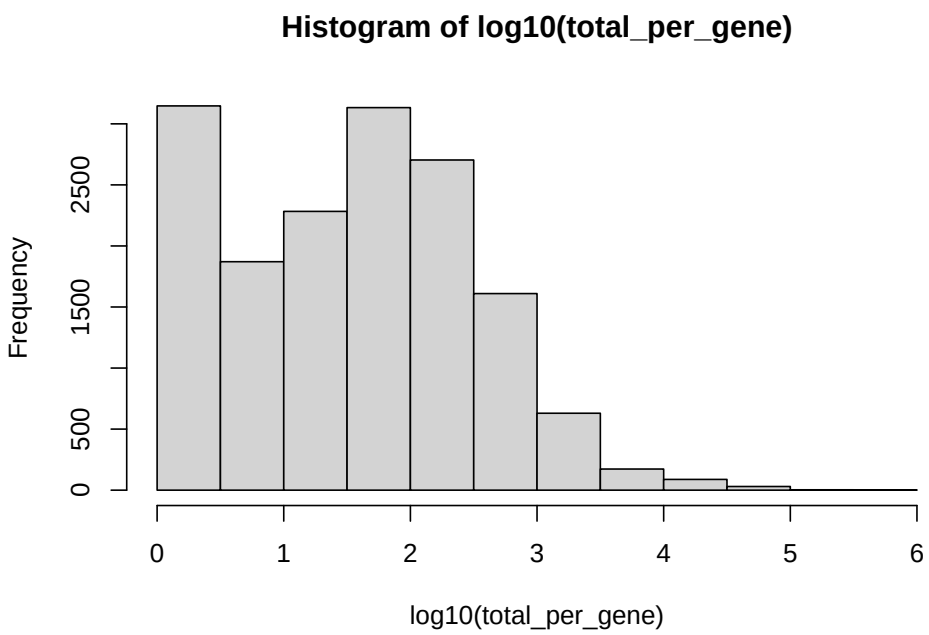
When doing differential expression, you generally ignore genes with low expression. In single cell datasets, there are many genes like this. Filtering here to make our dataset smaller so it runs quicker, and there is less aggressive correction for multiple hypotheses.

How many genes before filtering?

```
kang
#> An object of class Seurat
#> 35635 features across 5000 samples within 1 assay
#> Active assay: RNA (35635 features, 2000 variable features)
#>  2 dimensional reductions calculated: pca, umap
```

How many copies of each gene are there?

```
total_per_gene <- rowSums(GetAssayData(kang, 'counts'))
hist(log10(total_per_gene))
```



Lets keep only those genes with at least 50 copies across the entire experiment.

```
kang <- kang[total_per_gene >= 50, ]
```

How many genes after filtering?

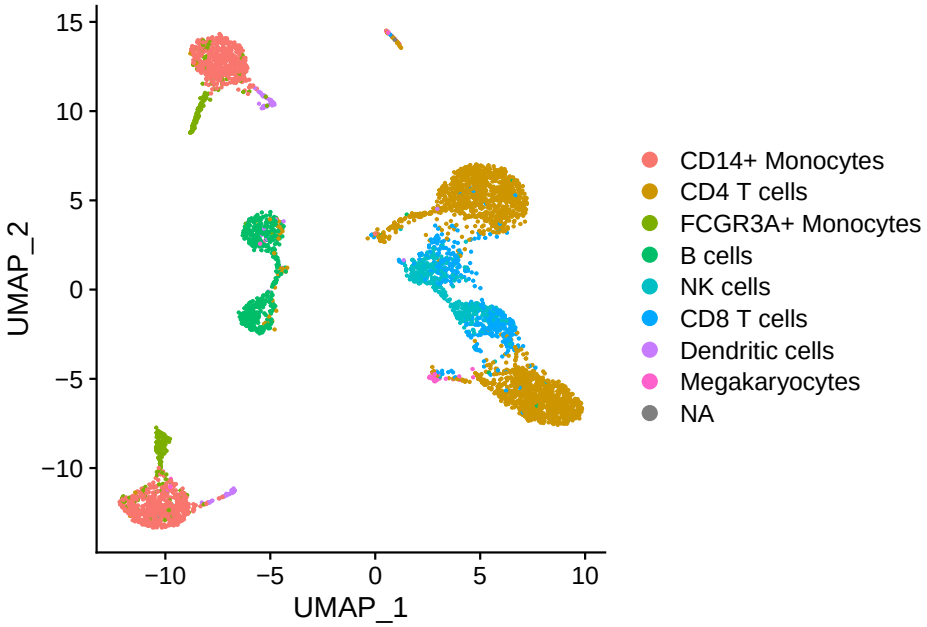
```
kang
#> An object of class Seurat
#> 7216 features across 5000 samples within 1 assay
```

```
#> Active assay: RNA (7216 features, 1228 variable features)
#>  2 dimensional reductions calculated: pca, umap
```

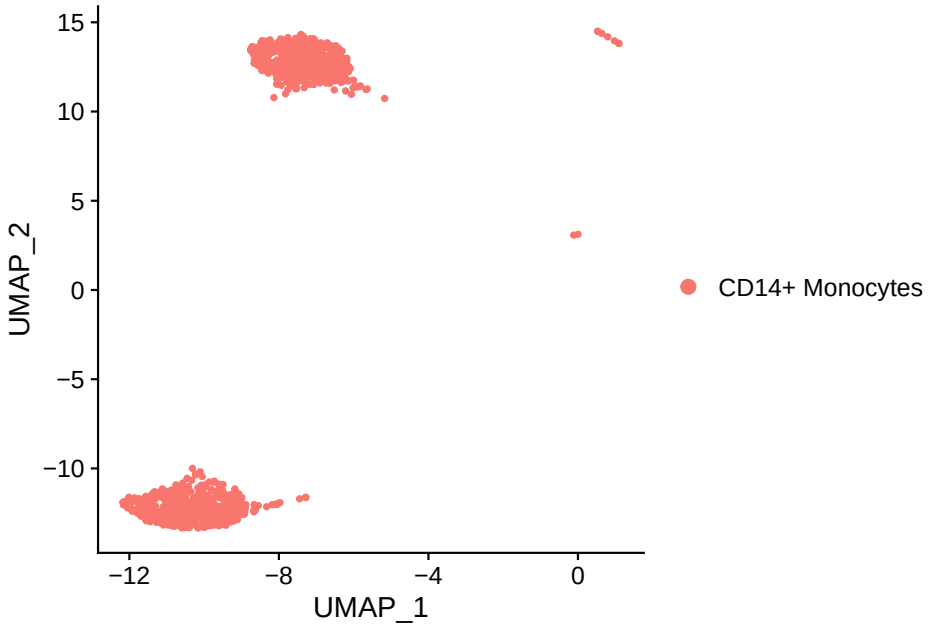
We might like to see the effect of IFN-beta stimulation on each cell type individually. For the purposes of this workshop, just going to test one cell type; CD14+ Monocytes

An easy way is to subset the object.

```
# Set idents to 'cell' column.
Idents(kang) <- kang$cell
DimPlot(kang)
```




```
kang.celltype <- kang[, kang$cell == "CD14+ Monocytes" ]
DimPlot(kang.celltype)
```



12.2 Default Wilcox test

To run this test, we change the Idents to the factor(column) we want to test. In this case, that's 'stim'.

```
# Change Ident to Condition
Idents(kang.celltype) <- kang.celltype$stim

# default, wilcox test
de_result_wilcox <- FindMarkers(kang.celltype,
            ident.1 = 'stim',
            ident.2 = 'ctrl',
            logfc.threshold = 0, # Give me ALL results
            min.pct = 0
            )

# Add average expression for plotting
de_result_wilcox$AveExpr<- rowMeans(kang.celltype[["RNA"]][rownames(de_result_wilcox),]
```

Look at the top differentially expressed genes.

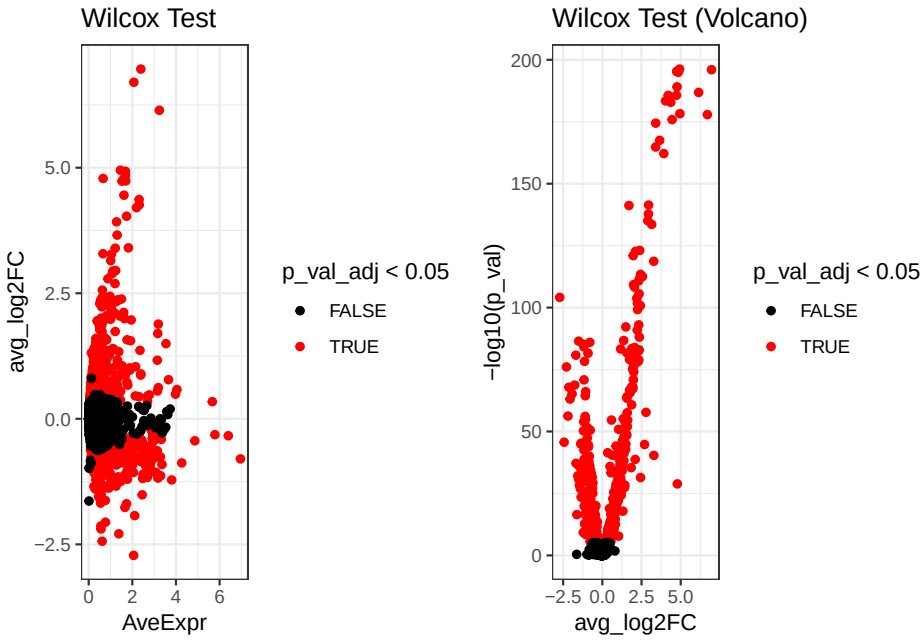
```
head(de_result_wilcox)
#>                 p_val avg_log2FC pct.1 pct.2     p_val_adj
#> RSAD2   5.541857e-197   4.928403 0.975 0.043 3.999004e-193
#> CXCL10  9.648067e-197   6.963650 0.973 0.038 6.962045e-193
#> IFIT3   4.988121e-196   4.736979 0.979 0.050 3.599428e-192
```

```
#> TNFSF10 1.116418e-195   4.847351 0.977 0.055 8.056075e-192
#> IFIT1   8.116699e-190   4.766009 0.950 0.026 5.857010e-186
#> ISG15   1.524836e-187   6.143242 0.998 0.320 1.100322e-183
#>           AveExpr
#> RSAD2   1.686530
#> CXCL10  2.388462
#> IFIT3   1.701247
#> TNFSF10 1.682688
#> IFIT1   1.584751
#> ISG15   3.239774
```

```
p1 <- ggplot(de_result_wilcox, aes(x=AveExpr, y=avg_log2FC, col=p_val_adj < 0.05)) +
  geom_point() +
  scale_colour_manual(values=c('TRUE'="red",'FALSE'="black")) +
  theme_bw() +
  ggtitle("Wilcox Test")


p2 <- ggplot(de_result_wilcox, aes(x=avg_log2FC, y=-log10(p_val), col=p_val_adj < 0.05)) +
  geom_point() +
  scale_colour_manual(values=c('TRUE'="red",'FALSE'="black")) +
  theme_bw() +
  ggtitle("Wilcox Test (Volcano)")

p1 + p2
```



12.3 Seurat Negative binomial

Negative binonial test is run almost the same way - just need to specify it under 'test.use'

```
# Change Ident to Condition
Idents(kang.celltype) <- kang.celltype$stim

# default, wilcox test
de_result_negbinom <- FindMarkers(kang.celltype,
            test.use="negbinom", # Choose a different test.
            ident.1 = 'stim',
            ident.2 = 'ctrl',
            logfc.threshold = 0, # Give me ALL results
            min.pct = 0
)

# Add average expression for plotting
de_result_negbinom$AveExpr<- rowMeans(kang.celltype[["RNA"]][rownames(de_result_negbino
```

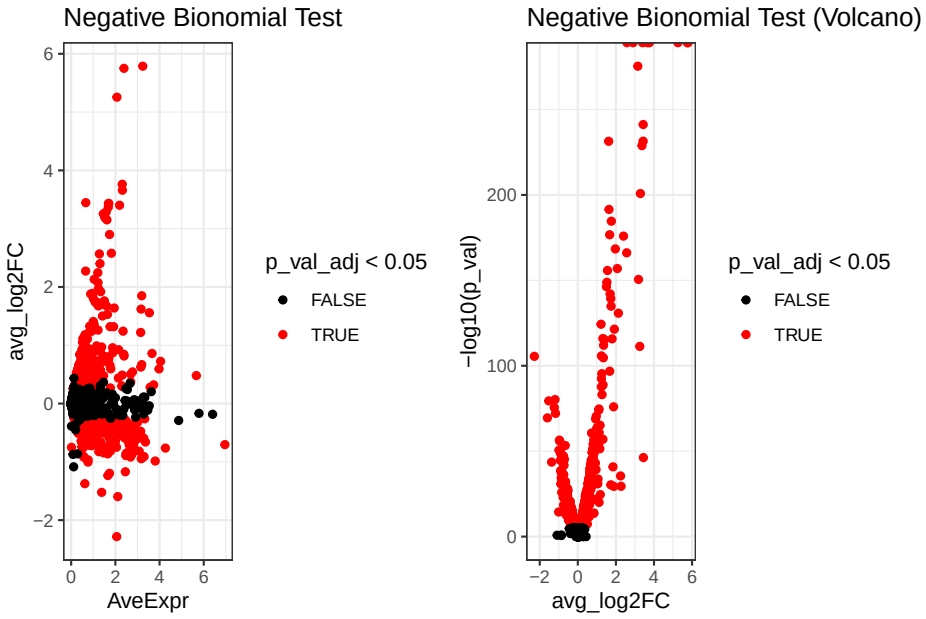
Look at the top differentially expressed genes.

```
head(de_result_negbinom)
#>         p_val avg_log2FC pct.1 pct.2 p_val_adj  AveExpr
#> ISG15       0   5.787224 0.998 0.320         0 3.239774
#> IFI6        0   2.578840 0.979 0.253         0 1.820140
#> CXCL10      0   5.750807 0.973 0.038         0 2.388462
#> LY6E        0   2.900314 0.973 0.133         0 1.736931
#> IFITM3      0   3.401677 0.994 0.267         0 2.190185
#> ISG20       0   3.659844 0.994 0.272         0 2.319655
```

```
p1 <- ggplot(de_result_negbinom, aes(x=AveExpr, y=avg_log2FC, col=p_val_adj < 0.05)) +
  geom_point() +
  scale_colour_manual(values=c('TRUE'="red",'FALSE'="black")) +
  theme_bw() +
  ggtitle("Negative Bionomial Test")


p2 <- ggplot(de_result_negbinom, aes(x=avg_log2FC, y=-log10(p_val), col=p_val_adj < 0.0
  geom_point() +
  scale_colour_manual(values=c('TRUE'="red",'FALSE'="black")) +
  theme_bw() +
  ggtitle("Negative Bionomial Test (Volcano)")

p1 + p2
```



12.4 Pseudobulk

Pseudobulk analysis is an option where you have biological replicates. It is essentially pooling the individual cell counts and treating your expreiment like a bulk RNAseq.

First, you need to build a pseudobulk matrix - the `AggregateExpression()` function can do this, once you set the 'Idents' of your seurat object to your grouping factor (here, thats a combination of individual+treatment called 'sample', instead of the 'stim' treatment column).

```
# Tools for bulk differential expression
library(limma)
#>
#> Attaching package: 'limma'
#> The following object is masked from 'package:BiocGenerics':
#>
#>     plotMA
library(edgeR)
#>
#> Attaching package: 'edgeR'
#> The following object is masked from 'package:SingleCellExperiment':
#>
#>     cpm
```

```
# Change idents to ind for grouping.
kang.celltype$sample <- factor(paste(kang.celltype$stim, kang.celltype$ind, sep="_"))
Idents(kang.celltype) <- kang.celltype$sample

# THen pool together counts in those groups
# AggregateExperssion returns a list of matricies - one for each assay requested (even
pseudobulk_matrix_list <- AggregateExpression( kang.celltype,  slot = 'counts', assays=
pseudobulk_matrix      <- pseudobulk_matrix_list[['RNA']]
colnames(pseudobulk_matrix) <- as.character(colnames(pseudobulk_matrix)) # Changes col
pseudobulk_matrix[1:5,1:4]
#>          ctrl_101 ctrl_1015 ctrl_1016 ctrl_1039
#> NOC2L           2         7         0         0
#> HES4            0         3         2         1
#> ISG15          31       185       236        41
#> TNFRSF18        0         3         4         2
#> TNFRSF4         0         2         0         0
```

Now it looks like a bulk RNAseq experiment, so treat it like one.

We can use the popular `limma` package for differential expression. Here is one tutorial, and the hefty reference manual is hosted by bioconductor.

In brief, this code below constructs a linear model for this experiment that accounts for the variation in individuals and treatment. It then tests for differential expression between 'stim' and 'ctrl' groups.

```
dge <- DGEList(pseudobulk_matrix)
dge <- calcNormFactors(dge)

# Remove _ and everything after it - yeilds stim group
stim <- gsub("_.*","",colnames(pseudobulk_matrix))

# Removing everything before the _ for the individua, then converting those numerical
ind  <- as.character(gsub(".*_","",colnames(pseudobulk_matrix)))

design <- model.matrix( ~0 + stim + ind)
vm  <- voom(dge, design = design, plot = FALSE)
fit <- lmFit(vm, design = design)

contrasts <- makeContrasts(stimstim - stimctrl, levels=coef(fit))
fit <- contrasts.fit(fit, contrasts)
fit <- eBayes(fit)

de_result_pseudobulk <- topTable(fit, n = Inf, adjust.method = "BH")
de_result_pseudobulk <- arrange(de_result_pseudobulk , adj.P.Val)
```

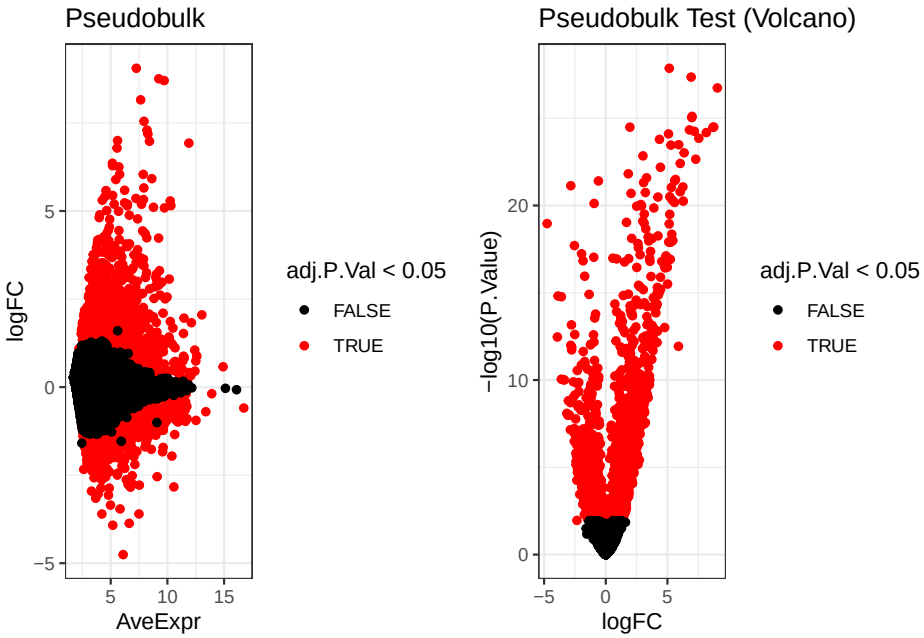
Look at the significantly differentially expressed genes:

```
head(de_result_pseudobulk)
#>           logFC   AveExpr        t      P.Value
#> ISG20  5.151090 10.311187 34.62460 1.377577e-28
#> ISG15  6.926462 11.895928 33.45672 4.402969e-28
#> CXCL11 9.051653  7.260525 32.07090 1.838679e-27
#> IFIT3  6.980913  8.420719 28.54319 9.234346e-26
#> HERC5  6.998957  5.602349 28.68162 7.853707e-26
#> TMSB10 1.959063 11.466981 27.48469 3.264041e-25
#>           adj.P.Val        B
#> ISG20  9.940598e-25 55.27733
#> ISG15  1.588591e-24 54.12183
#> CXCL11 4.422636e-24 51.56638
#> IFIT3  1.332701e-22 48.32088
#> HERC5  1.332701e-22 48.02111
#> TMSB10 2.950103e-22 47.34483
```

```
p1 <- ggplot(de_result_pseudobulk, aes(x=AveExpr, y=logFC, col=adj.P.Val < 0.05)) +
  geom_point() +
  scale_colour_manual(values=c('TRUE'="red",'FALSE'="black")) +
  theme_bw() +
  ggtitle("Pseudobulk")


p2 <- ggplot(de_result_pseudobulk, aes(x=logFC, y=-log10(P.Value), col=adj.P.Val < 0.05)) +
  geom_point() +
  scale_colour_manual(values=c('TRUE'="red",'FALSE'="black")) +
  theme_bw() +
  ggtitle("Pseudobulk Test (Volcano)")

p1 + p2
```



Discussion

These methods give different results. How would you decide which to use? How could you check an individual gene?

Chapter 13

Cell cycle Assignment

In some datasets, the phase of cell cycle that a cell is in (G1/G2M/S) can account for alot of the observed transcriptomic variation. There may be clustering by phase, or separation in the UMAP by phase.

Seurat provides a simple method for assigning cell cycle state to each cell. Other methods are available.

More information about assigning cell cycle states to cells is in the cell cycle vignette

```
# A list of cell cycle markers, from Tirosh et al, 2015, is loaded with Seurat.  We can
# segregate this list into markers of G2/M phase and markers of S phase
s.genes   <- cc.genes$s.genes
g2m.genes <- cc.genes$g2m.genes

# Use those lists with the cell cycle scoring function in Seurat.
pbmc <- CellCycleScoring(pbmc, s.features = s.genes, g2m.features = g2m.genes)
#> Warning: The following features are not present in the
#> object: DTL, UHRF1, MLF1IP, EXO1, CASP8AP2, BRIP1, E2F8,
#> not searching for symbol synonyms
#> Warning: The following features are not present in the
#> object: FAM64A, BUB1, HJURP, CDCA3, TTK, CDC25C, DLGAP5,
#> CDCA2, ANLN, GAS2L3, not searching for symbol synonyms
```

Which adds S.Score, G2M.Score and Phase calls to the metadata.

```
head(pbmc@meta.data)
#>                  orig.ident nCount_RNA nFeature_RNA
#> AAACATACAACCAC-1     pbmc3k       2419          779
#> AAACATTGAGCTAC-1     pbmc3k       4903         1352
#> AAACATTGATCAGC-1     pbmc3k       3147         1129
```

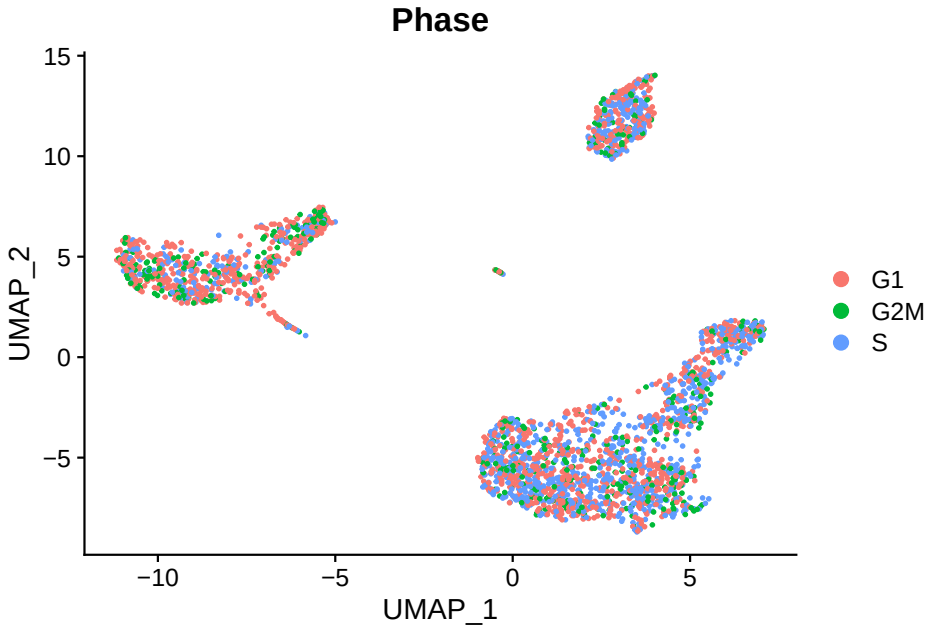
```
#> AAACCGTGCTTCCG-1     pbmc3k       2639          960
#> AAACCGTGTATGCG-1     pbmc3k        980          521
#> AAACGCACTGGTAC-1     pbmc3k       2163          781
#>                  percent.mt RNA_snn_res.0.5 seurat_clusters
#> AAACATACAACCAC-1  3.0177759               2               6
#> AAACATTGAGCTAC-1  3.7935958               3               1
#> AAACATTGATCAGC-1  0.8897363               2               0
#> AAACCGTGCTTCCG-1  1.7430845               1               5
#> AAACCGTGTATGCG-1  1.2244898               6               8
#> AAACGCACTGGTAC-1  1.6643551               2               0
#>                  RNA_snn_res.0.1 RNA_snn_res.0.2
#> AAACATACAACCAC-1               0               0
#> AAACATTGAGCTAC-1               3               3
#> AAACATTGATCAGC-1               0               0
#> AAACCGTGCTTCCG-1               1               1
#> AAACCGTGTATGCG-1               2               2
#> AAACGCACTGGTAC-1               0               0
#>                  RNA_snn_res.0.3 RNA_snn_res.0.4
#> AAACATACAACCAC-1               0               2
#> AAACATTGAGCTAC-1               3               3
#> AAACATTGATCAGC-1               0               2
#> AAACCGTGCTTCCG-1               1               1
#> AAACCGTGTATGCG-1               2               6
#> AAACGCACTGGTAC-1               0               2
#>                  RNA_snn_res.0.6 RNA_snn_res.0.7
#> AAACATACAACCAC-1               1               1
#> AAACATTGAGCTAC-1               3               3
#> AAACATTGATCAGC-1               1               1
#> AAACCGTGCTTCCG-1               2               2
#> AAACCGTGTATGCG-1               6               6
#> AAACGCACTGGTAC-1               1               1
#>                  RNA_snn_res.0.8 RNA_snn_res.0.9
#> AAACATACAACCAC-1               6               1
#> AAACATTGAGCTAC-1               2               2
#> AAACATTGATCAGC-1               1               1
#> AAACCGTGCTTCCG-1               4               4
#> AAACCGTGTATGCG-1               8               7
#> AAACGCACTGGTAC-1               1               1
#>                  RNA_snn_res.1 RNA_snn_res.1.1
#> AAACATACAACCAC-1             6               6
#> AAACATTGAGCTAC-1             2               2
#> AAACATTGATCAGC-1             1               1
#> AAACCGTGCTTCCG-1             4               4
#> AAACCGTGTATGCG-1             8               8
#> AAACGCACTGGTAC-1             1               1
```

```
#>                   RNA_snn_res.1.2 RNA_snn_res.1.3
#> AAACATACAACCAC-1               6               8
#> AAACATTGAGCTAC-1               2               2
#> AAACATTGATCAGC-1               1               0
#> AAACCGTGCTTCCG-1               4               5
#> AAACCGTGTATGCG-1               8               9
#> AAACGCACTGGTAC-1               1               0
#>                   RNA_snn_res.1.4 RNA_snn_res.1.5
#> AAACATACAACCAC-1               8               9
#> AAACATTGAGCTAC-1               2               2
#> AAACATTGATCAGC-1               0               1
#> AAACCGTGCTTCCG-1               5               4
#> AAACCGTGTATGCG-1               9               8
#> AAACGCACTGGTAC-1               0               1
#>                   RNA_snn_res.1.6 RNA_snn_res.1.7
#> AAACATACAACCAC-1               8               8
#> AAACATTGAGCTAC-1               1               1
#> AAACATTGATCAGC-1               0               0
#> AAACCGTGCTTCCG-1               3               3
#> AAACCGTGTATGCG-1               7               7
#> AAACGCACTGGTAC-1               0               0
#>                   RNA_snn_res.1.8 RNA_snn_res.1.9
#> AAACATACAACCAC-1               7               6
#> AAACATTGAGCTAC-1               1               1
#> AAACATTGATCAGC-1               0               0
#> AAACCGTGCTTCCG-1               3               3
#> AAACCGTGTATGCG-1               8               8
#> AAACGCACTGGTAC-1               0               0
#>                   RNA_snn_res.2 Naive_CD4_T1  cell_label
#> AAACATACAACCAC-1              6   1.22824523 Naive_CD4_T
#> AAACATTGAGCTAC-1              1  -0.08111043        <NA>
#> AAACATTGATCAGC-1              0  -0.37682601        <NA>
#> AAACCGTGCTTCCG-1              5  -0.72739714        <NA>
#> AAACCGTGTATGCG-1              8  -1.17396454        <NA>
#> AAACGCACTGGTAC-1              0  -0.63807586        <NA>
#>                   SingleR.labels     S.Score    G2M.Score
#> AAACATACAACCAC-1         T_cells  0.09853841 -0.044716507
#> AAACATTGAGCTAC-1          B_cell -0.02364305 -0.046889929
#> AAACATTGATCAGC-1         T_cells -0.02177266  0.074841537
#> AAACCGTGCTTCCG-1        Monocyte  0.03794398  0.006575446
#> AAACCGTGTATGCG-1         NK_cell -0.03309970  0.027910063
#> AAACGCACTGGTAC-1         T_cells -0.04814181 -0.078164839
#>                   Phase
#> AAACATACAACCAC-1      S
#> AAACATTGAGCTAC-1     G1
```

```
#> AAACATTGATCAGC-1   G2M
#> AAACCGTGCTTCCG-1     S
#> AAACCGTGTATGCG-1   G2M
#> AAACGCACTGGTAC-1    G1
```

We can then check the cell phase on the UMAP. In this dataset, phase isn't driving the clustering, and would not require any further handling.

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



Where a bias *is* present, your course of action depends on the task at hand. It might involve 'regressing out' the cell cycle variation when scaling data `ScaleData(kang, vars.to.regress="Phase")`, omitting cell-cycle dominated clusters, or just accounting for it in your differential expression calculations.

If you are working with non-human data, you will need to convert these gene lists, or find new cell cycle associated genes in your species.

Chapter 14

Data set integration with Harmony

Why do we need to do this?

You can have data coming from different samples, batches or experiments and you will need to combine them.

When data is collected from multiple samples, multiple runs of the single cell sequencing library preparation, or multiple conditions, cells of the same type may become separated in the UMAP and be put into several different clusters.

For the purpose of clustering and cell identification, we would like to remove such effects.

We will now look at GSE96583, another PBMC dataset. For speed, we will be looking at a subset of 5000 cells from this data. The cells in this dataset were pooled from eight individual donors. A nice feature is that genetic differences allow some of the cell doublets to be identified. This data contains two batches of single cell sequencing. One of the batches was stimulated with IFN-beta.

The data has already been processed as we have done with the first PBMC dataset, and can be loaded from `kang2018.rds`.

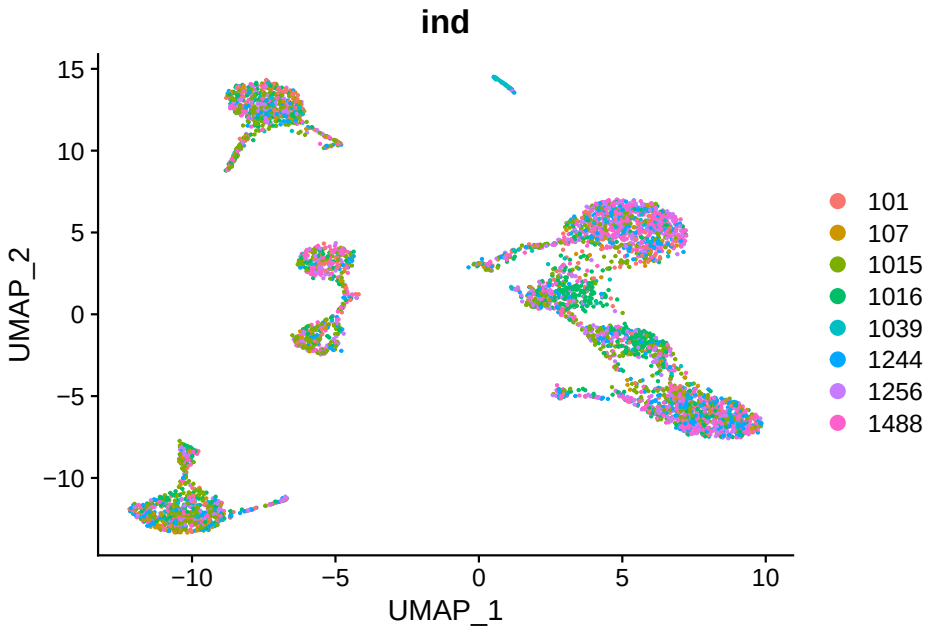
```
kang <- readRDS("data/kang2018.rds")

head(kang@meta.data)
#>                     orig.ident nCount_RNA nFeature_RNA  ind
#> AGGGCGCTATTTCC-1 SeuratProject       2020          523 1256
#> GGAGACGATTCGTT-1 SeuratProject        840          381 1256
```

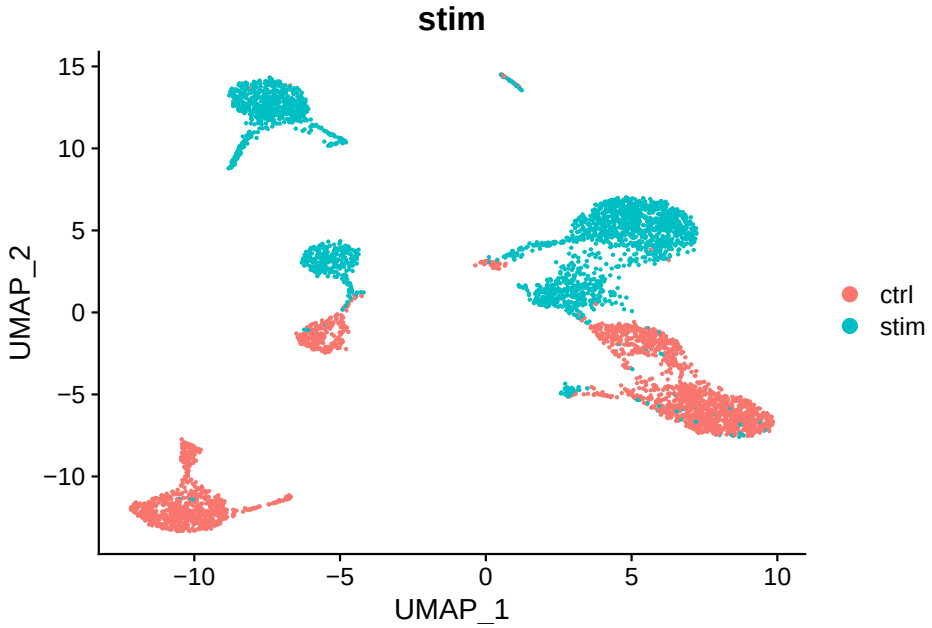
```
#> CACCGTTGTCGTAG-1 SeuratProject       3097          995 1016
#> TATCGTACACGCAT-1 SeuratProject       1011          540 1488
#> TGACGCCTTGCTTT-1 SeuratProject        570          367  101
#> TACGAGACCTATTC-1 SeuratProject       2399          770 1244
#>                  stim               cell multiplets
#> AGGGCGCTATTTCC-1 stim   CD14+ Monocytes    singlet
#> GGAGACGATTCGTT-1 stim       CD4 T cells    singlet
#> CACCGTTGTCGTAG-1 ctrl FCGR3A+ Monocytes    singlet
#> TATCGTACACGCAT-1 stim           B cells    singlet
#> TGACGCCTTGCTTT-1 ctrl       CD4 T cells       ambs
#> TACGAGACCTATTC-1 stim       CD4 T cells    singlet
```

- `ind` identifies a cell as coming from one of 8 individuals.
- `stim` identifies a cell as control or stimulated with IFN-beta.
- `cell` contains the cell types identified by the creators of this data set.
- `multiplets` classifies cells as singlet or doublet.

```
DimPlot(kang, reduction="umap", group.by="ind")
```

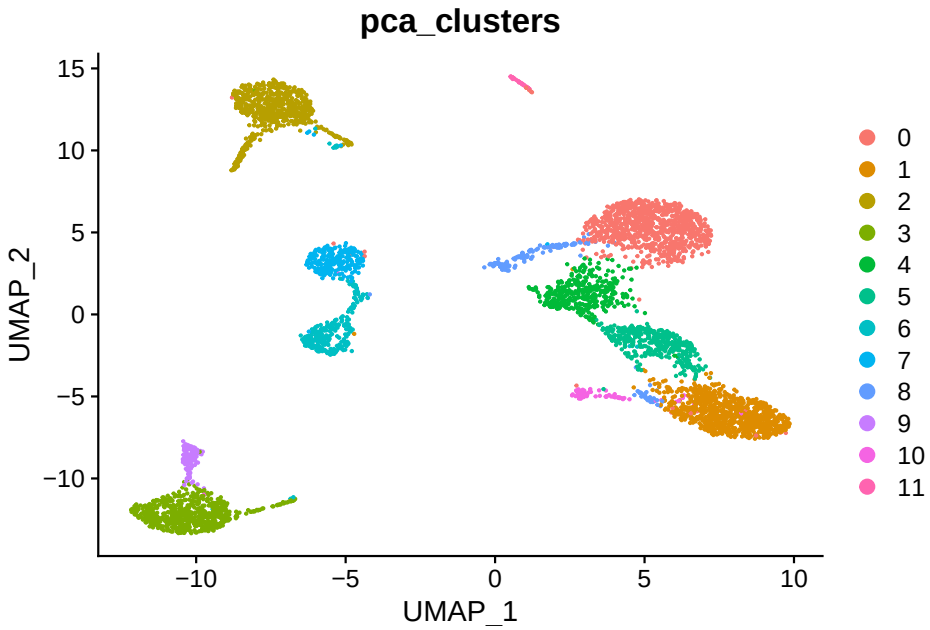



```
DimPlot(kang, reduction="umap", group.by="stim")
```




```
kang <- FindNeighbors(kang, reduction="pca", dims=1:10)
#> Computing nearest neighbor graph
#> Computing SNN
kang <- FindClusters(kang, resolution=0.25)
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 5000
#> Number of edges: 175130
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.9501
#> Number of communities: 12
#> Elapsed time: 0 seconds
kang$pca_clusters <- kang$seurat_clusters

DimPlot(kang, reduction="umap", group.by="pca_clusters")
```



There is a big difference between unstimulated and stimulated cells. This has split cells of the same type into pairs of clusters. If the difference was simply uniform, we could regress it out (e.g. using `ScaleData(..., vars.to.regress="stim")`). However, as can be seen in the PCA plot, the difference is not uniform and we need to do something cleverer.

We will use Harmony, which can remove non-uniform effects. We will try to remove both the small differences between individuals and the large difference between the unstimulated and stimulated cells.

Harmony operates only on the PCA scores. The original gene expression levels remain unaltered.

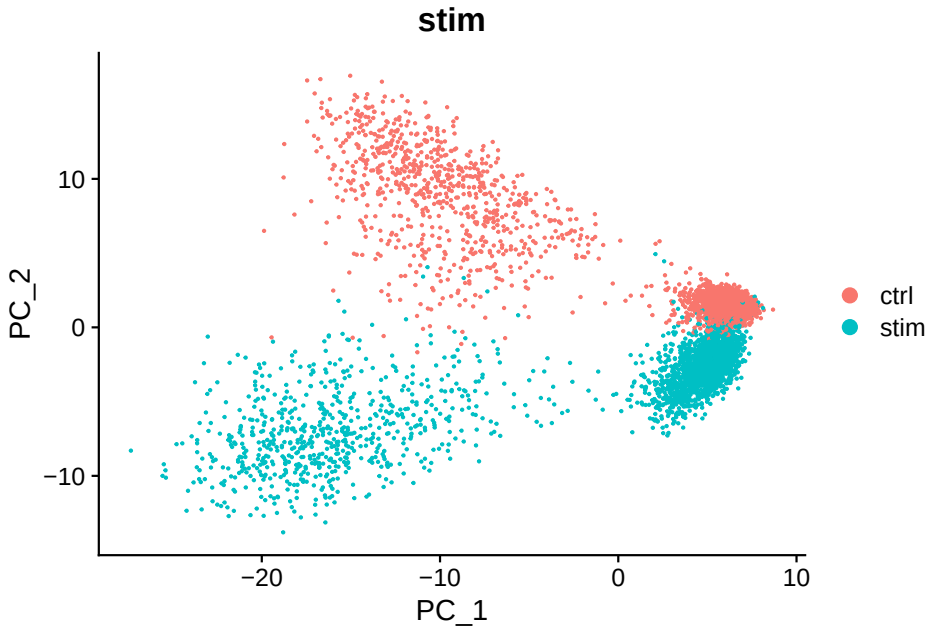
```
library(harmony)
#> Loading required package: Rcpp

kang <- RunHarmony(kang, c("stim", "ind"), reduction="pca")
#> Harmony 1/10
#> Harmony 2/10
#> Harmony 3/10
#> Harmony 4/10
#> Harmony 5/10
#> Harmony 6/10
#> Harmony 7/10
#> Harmony 8/10
#> Harmony 9/10
#> Harmony converged after 9 iterations
```

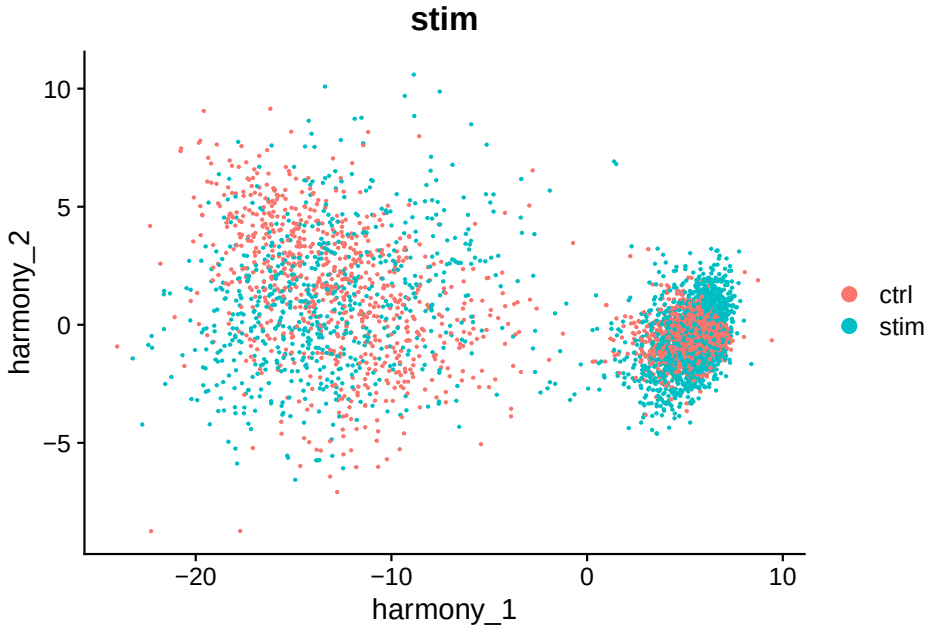
```
#> Warning: Invalid name supplied, making object name
#> syntactically valid. New object name is
#> Seurat..ProjectDim.RNA.harmony; see ?make.names for more
#> details on syntax validity
```

This has added a new set of reduced dimensions to the Seurat object, `kang$harmony` which is a modified version of the existing `kang$pca` reduced dimensions. The PCA plot shows a large difference between 'ctrl' and 'stim', but this has been removed in the harmony reduction.

```
DimPlot(kang, reduction="pca", group.by="stim")
```



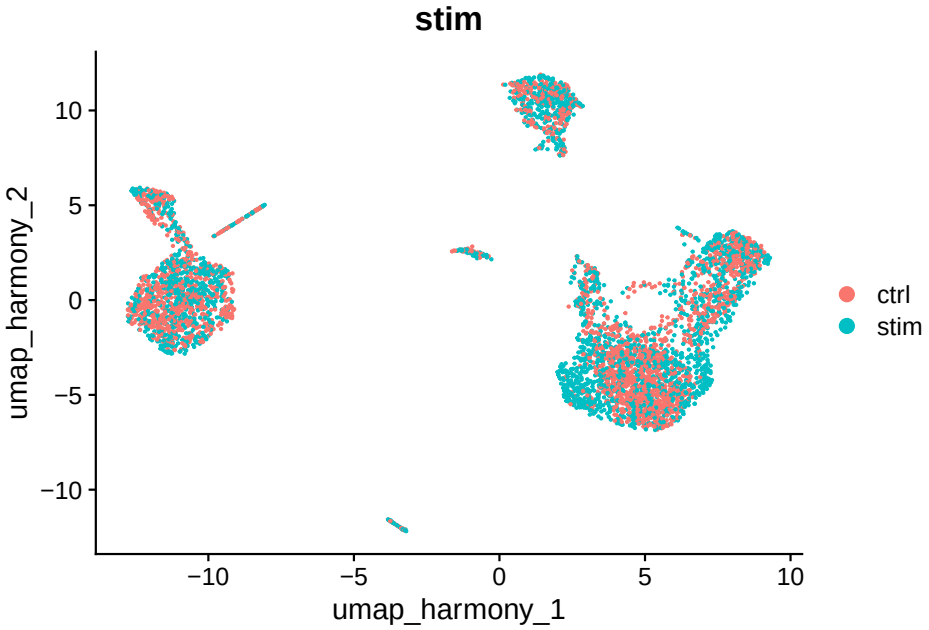

```
DimPlot(kang, reduction="harmony", group.by="stim")
```



We can use `harmony` the same way we used the `pca` reduction to compute a UMAP layout or to find clusters.

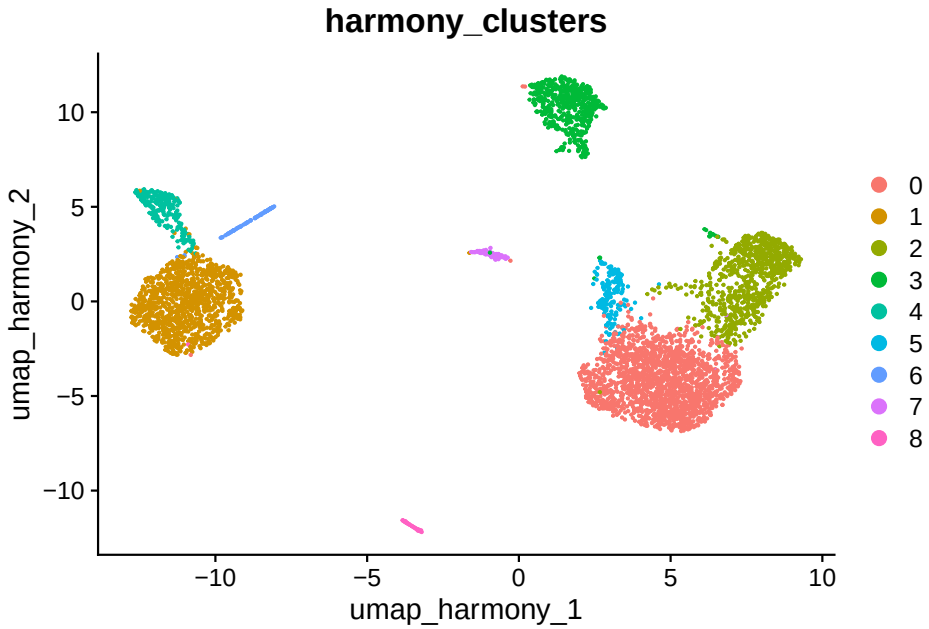
```
kang <- RunUMAP(kang, reduction="harmony", dims=1:10, reduction.name="umap_harmony")
#> 06:00:32 UMAP embedding parameters a = 0.9922 b = 1.112
#> 06:00:32 Read 5000 rows and found 10 numeric columns
#> 06:00:32 Using Annoy for neighbor search, n_neighbors = 30
#> 06:00:32 Building Annoy index with metric = cosine, n_trees = 50
#> 0%   10   20   30   40   50   60   70   80   90   100%
#> [----|----|----|----|----|----|----|----|----|----|
#> **************************************************|
#> 06:00:33 Writing NN index file to temp file /var/folders/my/cwpwt_rd1znflqb8gm8py41
#> 06:00:33 Searching Annoy index using 1 thread, search_k = 3000
#> 06:00:34 Annoy recall = 100%
#> 06:00:34 Commencing smooth kNN distance calibration using 1 thread with target n_ne
#> 06:00:35 Initializing from normalized Laplacian + noise (using irlba)
#> 06:00:35 Commencing optimization for 500 epochs, with 210490 positive edges
#> 06:00:39 Optimization finished
#> Warning: Cannot add objects with duplicate keys (offending
#> key: UMAP_), setting key to 'umap_harmony_'

DimPlot(kang, reduction="umap_harmony", group.by="stim")
```

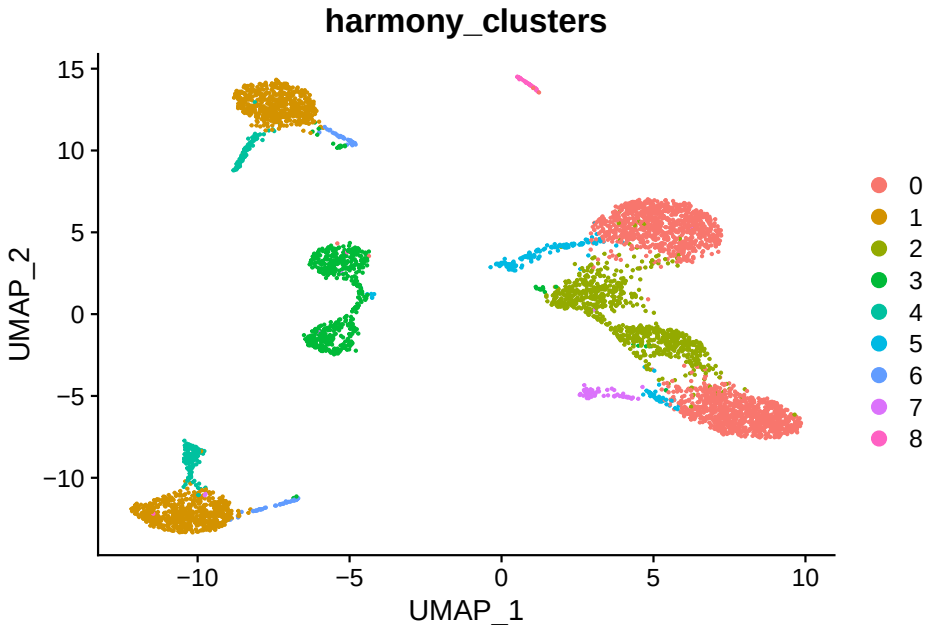



```
kang <- FindNeighbors(kang, reduction="harmony", dims=1:10)
#> Computing nearest neighbor graph
#> Computing SNN
kang <- FindClusters(kang, resolution=0.25)
#> Modularity Optimizer version 1.3.0 by Ludo Waltman and Nees Jan van Eck
#>
#> Number of nodes: 5000
#> Number of edges: 171396
#>
#> Running Louvain algorithm...
#> Maximum modularity in 10 random starts: 0.9324
#> Number of communities: 9
#> Elapsed time: 0 seconds
kang$harmony_clusters <- kang$seurat_clusters

DimPlot(kang, reduction="umap_harmony", group.by="harmony_clusters")
```




```
DimPlot(kang, reduction="umap", group.by="harmony_clusters")
```



Having found a good set of clusters, we would usually perform differential expression analysis on the original data and include batches/runs/individuals as predictors in the linear model. In this example we could now compare unstimulated and stimulated cells within each cluster. A particularly nice statisti-

cal approach that is possible here would be to convert the counts to pseudo-bulk data for the eight individuals, and then apply a bulk RNA-Seq differential expression analysis method. However there is still the problem that unstimulated and stimulated cells were processed in separate batches.

Chapter 15

Resources

Useful resources for next steps.

15.1 Help and fruther Resources

Seurat Vignettes

https://satijalab.org/seurat/index.html

There are a good many Seurat vigettes for different aspects of the Seurat package. E.g.

- Guided Clustering tutorial : We've just worked through this
- Differential expression : An Exploration of differential expression methods within Seurat
- Data integration : Seurat's data integration is a popular method to combine different datasets into one joint analysis.

Seurat Cheatsheet

https://satijalab.org/seurat/articles/essential_commands.html

A useful resource for asking; How can I do 'X' with my seurat object?

OSCA

https://bioconductor.org/books/release/OSCA/

An comprehensive resource for analysis approaches for single cell data. This uses the SingleCellExperiment bioconductor ecosystem, but alot of the same principle still apply.

This includes a good discussion of useing pseudobulk approaches, worth checking out for differential expression analyses.

MBP training Reading list

https://monashbioinformaticsplatform.github.io/Single-Cell-Workshop/

A workshop page for a previous workshop (upon which this one is based) run by Monash Bioinformatics Platform - down the bottom there is an extensive list of useful single cell links and resources.

Biocommons Single Cell Omics

https://www.biocommons.org.au/single-cell-omics

Join the single cell omics community resources being setup by biocommons.

15.2 Data

Demo 10X data

https://www.10xgenomics.com/resources/datasets

10X genomics have quite a few example datasets availble for download (including PBMC3k). This is a useful resource if you want to see what the 'raw' data looks like for a particular technology.

GEO

https://www.ncbi.nlm.nih.gov/geo/

Many papers publish their raw single cell data in GEO. Formats vary, but often you can find the counts matrix.

Seurat data

https://github.com/satijalab/seurat-data

Package for obtaining a few datasets as seurat objects.

15.3 Analysis Tools

A handful of the many tools that might be worth checking out for next steps.

Cyclone

https://pubmed.ncbi.nlm.nih.gov/26142758/

Part of the scran package, cyclone is a(nother) method for determining cell phase. Doco

Harmony

https://portals.broadinstitute.org/harmony/articles/quickstart.html

Method for integration of multiple single cell datasets.

SingleR

http://bioconductor.org/books/release/SingleRBook/

There is extensive documentation for the singleR package in the 'singleR' book.

Scrublet

https://github.com/swolock/scrublet

A python based tool for doublet detection. One of many tools in this space.

ScVelo

https://scvelo.readthedocs.io/

A package for single cell RNA velocity analysis, useful for developmental/pseudotime trajectories. Python/scanpy based.

Monocle

https://cole-trapnell-lab.github.io/monocle3/

A package for single cell developmental//pseudotime trajectory analysis.

TidySeurat

https://stemangiola.github.io/tidyseurat/

For fans of tidyverse-everything, there's tidyseurat. Example workflow here

---

15.4 Preprocessing Tools

Tooks that process raw sequencing data into counts matricies

Cell Ranger

https://support.10xgenomics.com/single-cell-gene-expression/software/pipelines/latest/what-is-cell-ranger

CellRanger is the 10X tool that takes raw fastq sequence files and produces the counts matricies that are the starting point for today's analysis. It only works for 10X data.

STARSolo

STAR is an aligner (which is actually used within cell ranger). STARSolo is a tool for producing counts matricies, and is configurable enough for use with multiple technologies.

https://github.com/alexdobin/STAR/blob/master/docs/STARsolo.md