Seurat - Guided Clustering Tutorial

Compiled: November 08, 2021

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).

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 our GitHub Wiki. For example, the count matrix is stored in `pbmc[["RNA"]]@counts`.

```
library(dplyr)
library(Seurat)
library(patchwork)
# Load the PBMC dataset
pbmc.data <- Read10X(data.dir = "../data/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.features = 200)
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"
##
## CD3D  4 . 10 . . 1 2 3 1 . . 2 7 1 . . 1 3 . 2  3 . . . . . 3 4 1 5
## TCL1A . .  . . . . . 1 . . . . . . . . . . . . . 1 . . . . . . . .
## MS4A1 . 6  . . . . . 1 1 1 . . . . . . . . . . 36 1 2 . . 2 . . . .
```

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
```

Standard pre-processing workflow 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. ## QC and selecting cells for further analysis 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-")
```

Where are QC metrics stored in Seurat?

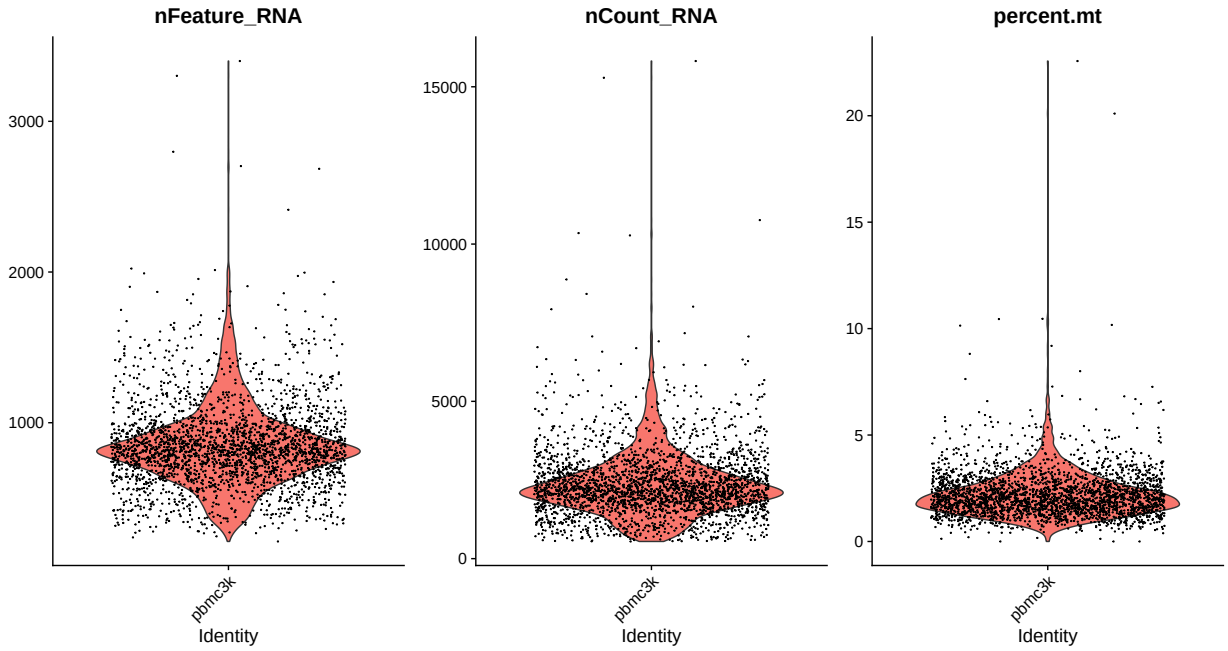
- The number of unique genes and total molecules are automatically calculated during `CreateSeuratObject()`
  - You can find them stored in the object meta data

```
# Show QC metrics for the first 5 cells
head(pbmc@meta.data, 5)
```

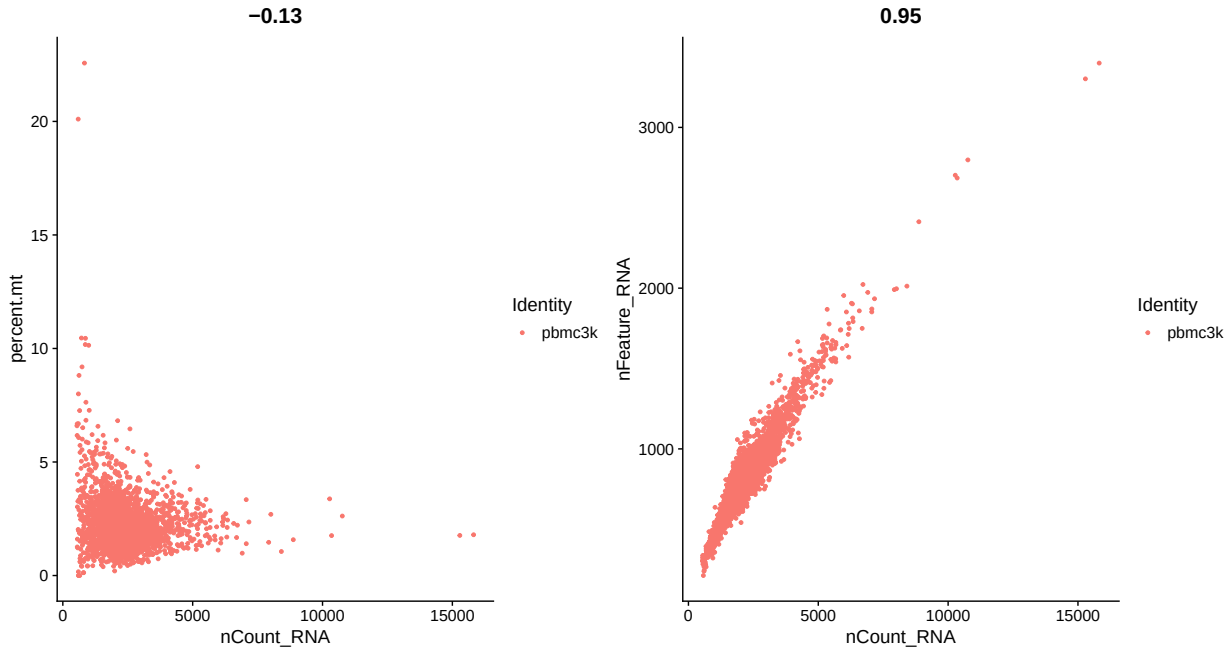
```
##                  orig.ident nCount_RNA nFeature_RNA percent.mt
## AAACATACAACCAC-1      pbmc3k       2419          779  3.0177759
## AAACATTGAGCTAC-1      pbmc3k       4903         1352  3.7935958
## AAACATTGATCAGC-1      pbmc3k       3147         1129  0.8897363
## AAACCGTGCTTCCG-1      pbmc3k       2639          960  1.7430845
## AAACCGTGTATGCG-1      pbmc3k        980          521  1.2244898
```

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

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




```r
# FeatureScatter is typically used to visualize feature-feature relationships, but can be used for anyt
plot1 <- FeatureScatter(pbmc, feature1 = "nCount_RNA", feature2 = "percent.mt")
plot2 <- FeatureScatter(pbmc, feature1 = "nCount_RNA", feature2 = "nFeature_RNA")
plot1 + plot2
```



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

Normalizing the data

After removing unwanted cells from the dataset, the next step is to normalize the data. By default, we employ a global-scaling normalization method "LogNormalize" 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)
```

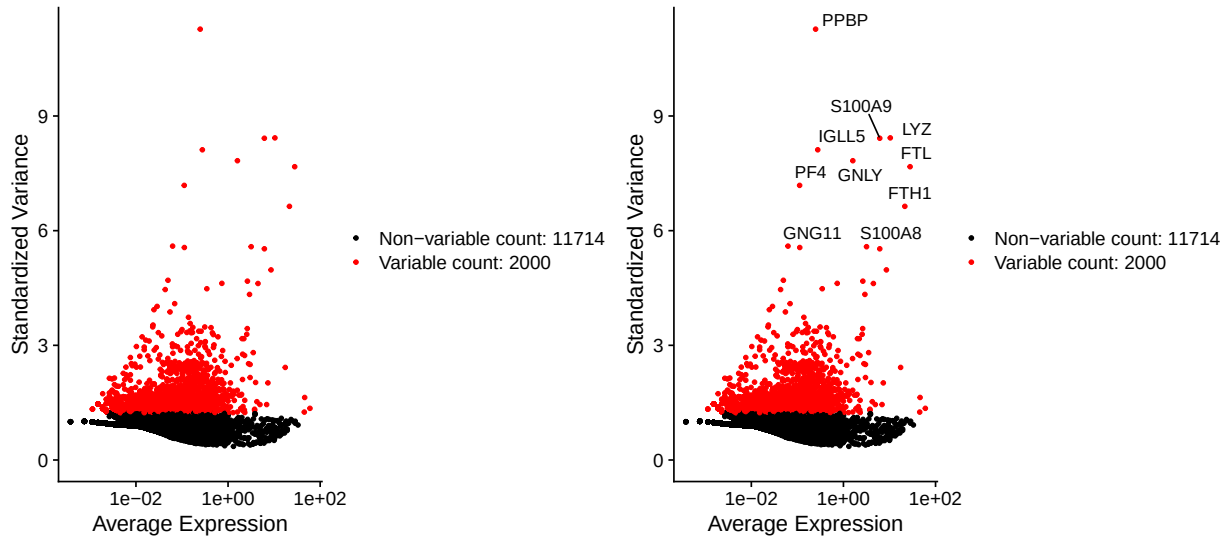
For clarity, in this previous line of code (and in future commands), we provide the default values for certain parameters in the function call. However, this isn't required and the same behavior can be achieved with:

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

Identification of highly variable features (feature selection)

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)
plot1 + plot2
```



** *# Scaling the data 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)
```

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.

*** # 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.

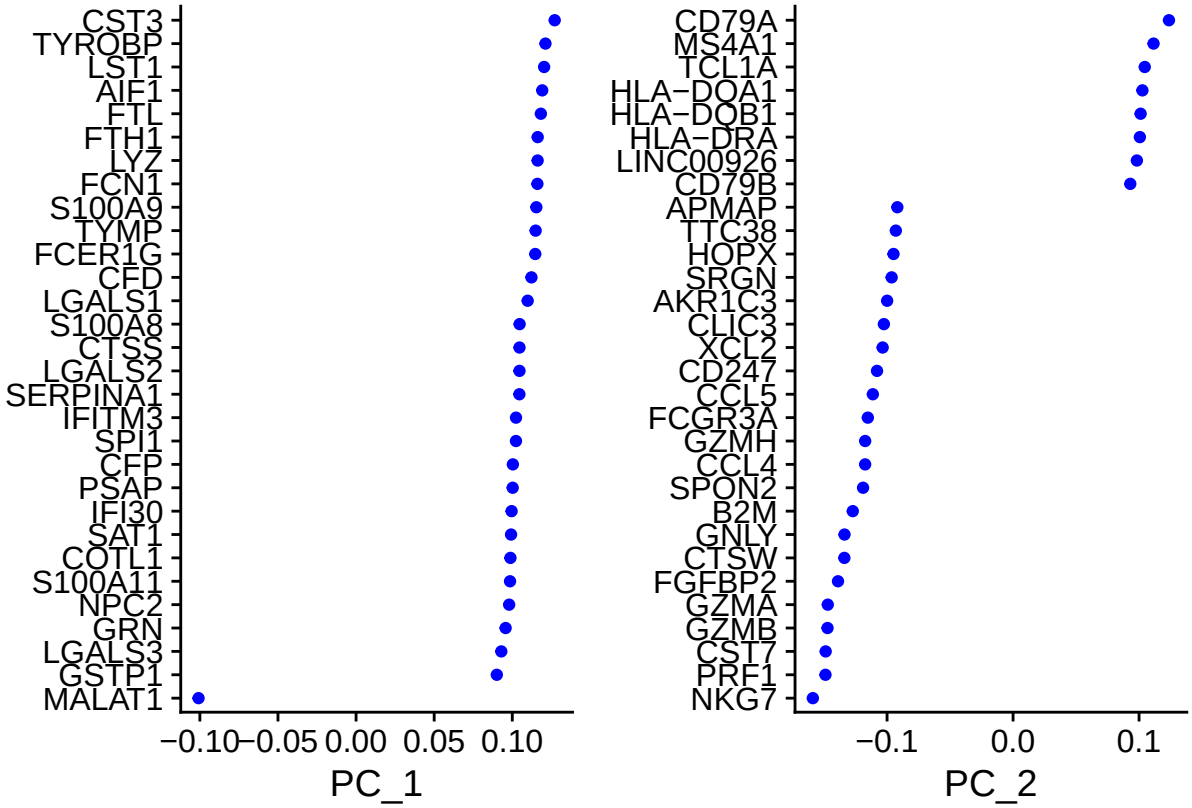
```
pbmc <- RunPCA(pbmc, features = VariableFeatures(object = pbmc))
```

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

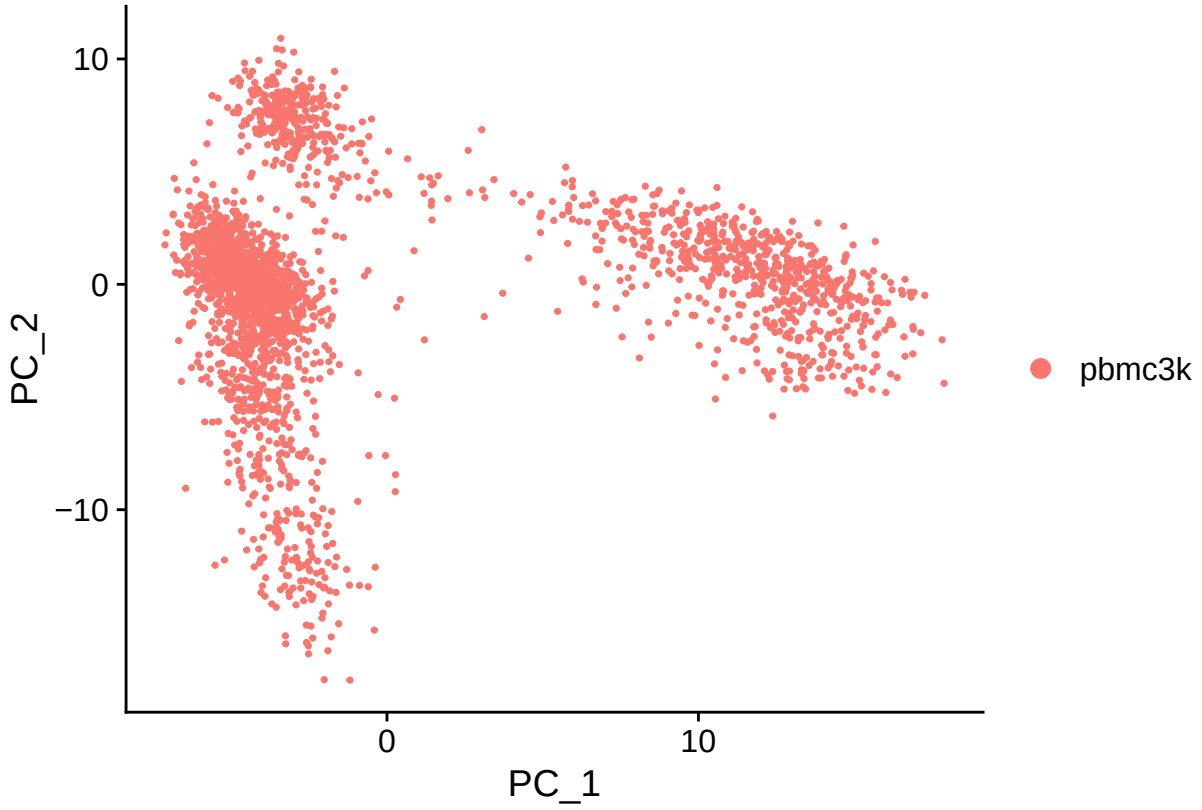
```
# Examine and visualize PCA results a few different ways
print(pbmc[['pca']], dims = 1:5, nfeatures = 5)
```

```
## 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')
```

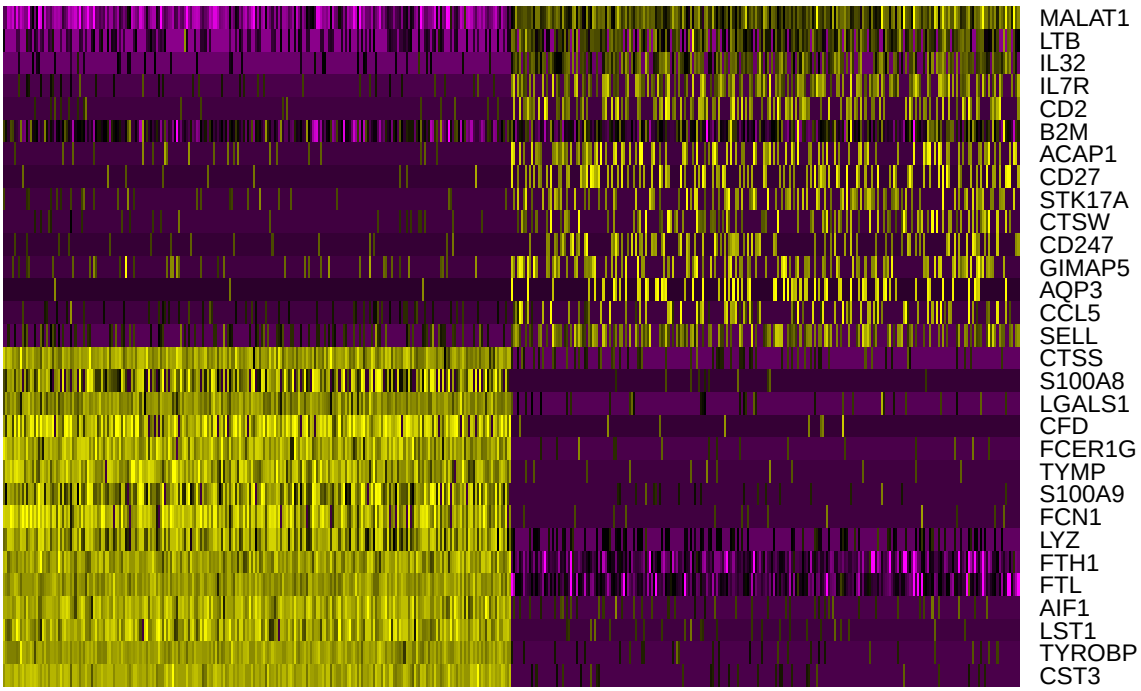



```
DimPlot(pbmc, reduction = 'pca')
```

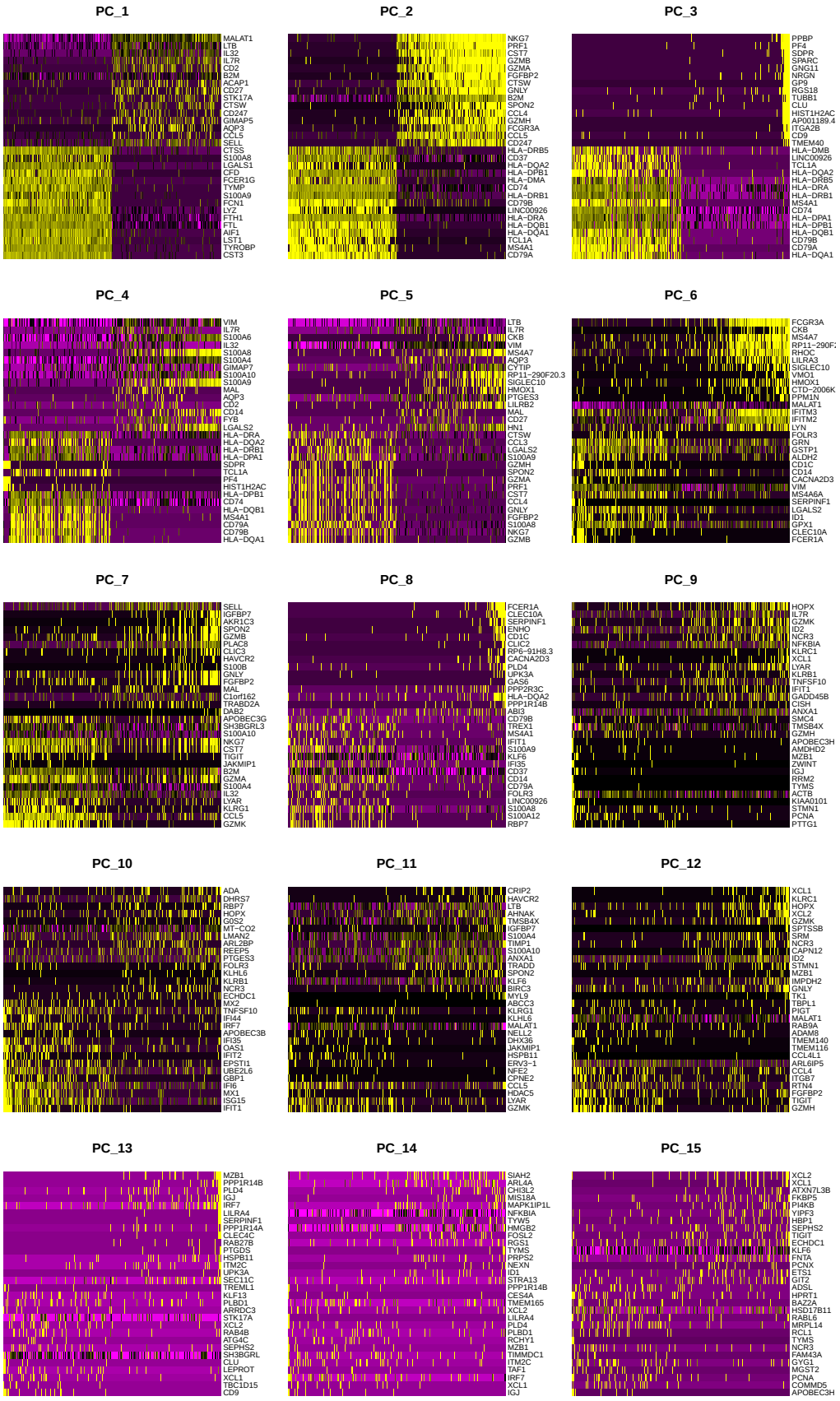


In particular `DimHeatmap()` allows for easy exploration of the primary sources of heterogeneity in a dataset, and can be useful when trying to decide which PCs to include for further downstream analyses. Both cells and features are ordered according to their PCA scores. Setting `cells` to a number plots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



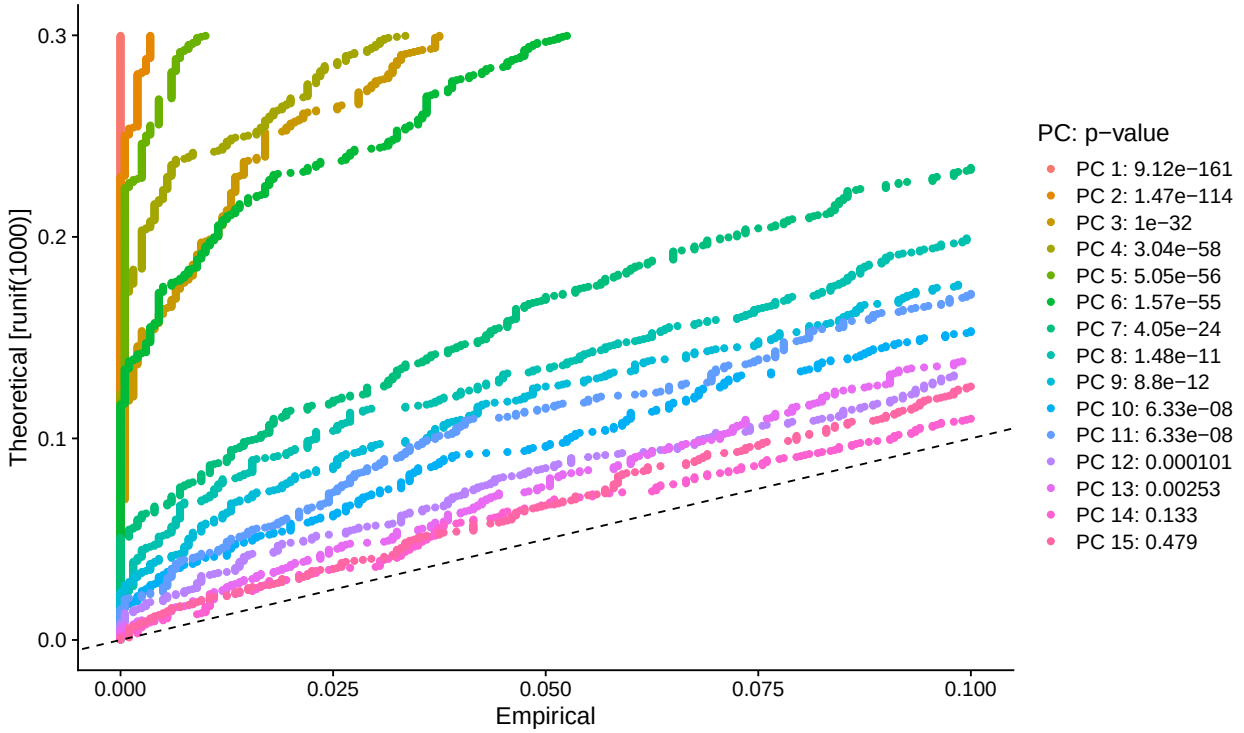
Determine the

'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? In Macosko *et al*, we implemented a resampling test inspired by the JackStraw procedure. We randomly permute a subset of the data (1% by default) and rerun PCA, constructing a 'null distribution' of feature scores, and repeat this procedure. We identify 'significant' PCs as those who have a strong enrichment of low p-value features.

```
# NOTE: This process can take a long time for big datasets, comment out for expediency. More approximat
pbmc <- JackStraw(pbmc, num.replicate = 100)
pbmc <- ScoreJackStraw(pbmc, dims = 1:20)
```

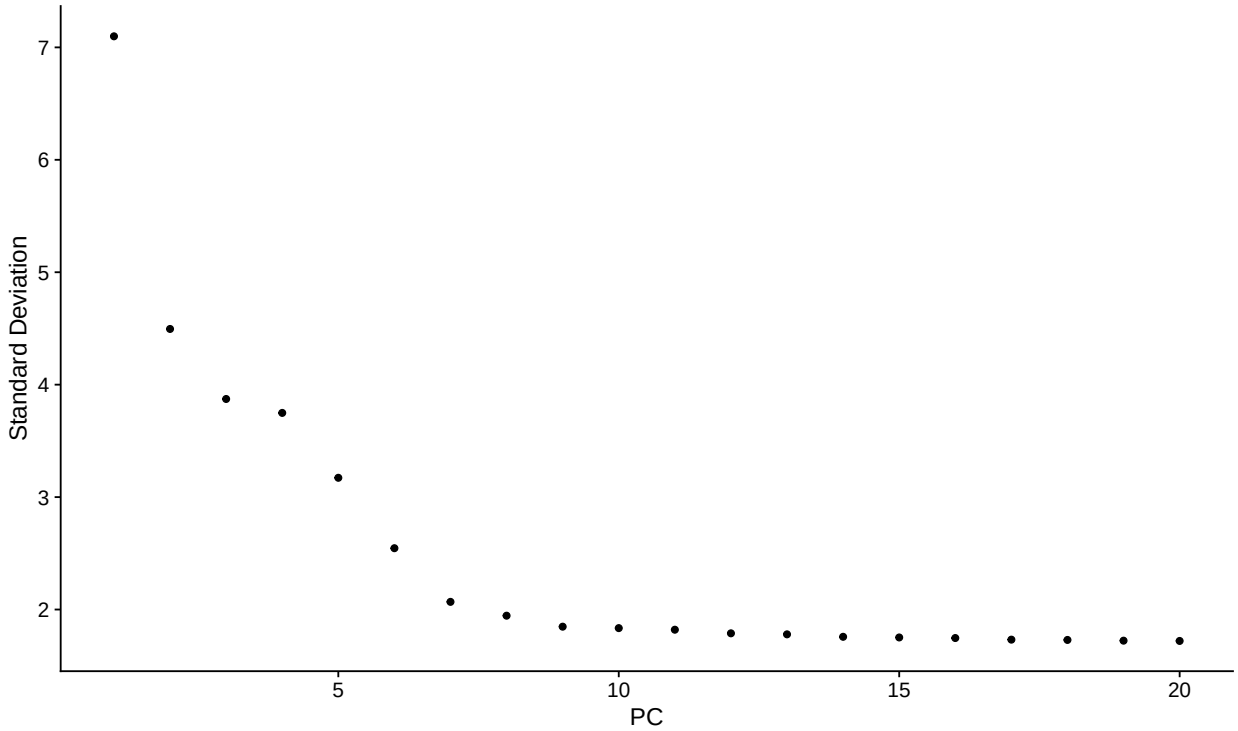
The `JackStrawPlot()` function provides a visualization tool for comparing the distribution of p-values for each PC with a uniform distribution (dashed line). 'Significant' PCs will show a strong enrichment of features with low p-values (solid curve above the dashed line). In this case it appears that there is a sharp drop-off in significance after the first 10-12 PCs.

```
JackStrawPlot(pbmc, dims = 1:15)
```



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. ** *# Cluster the 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)
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: 95965
##
## Running Louvain algorithm...
## Maximum modularity in 10 random starts: 0.8723
## Number of communities: 9
## Elapsed time: 0 seconds
```

```
# Look at cluster IDs of the first 5 cells
head(Idents(pbmc), 5)
```

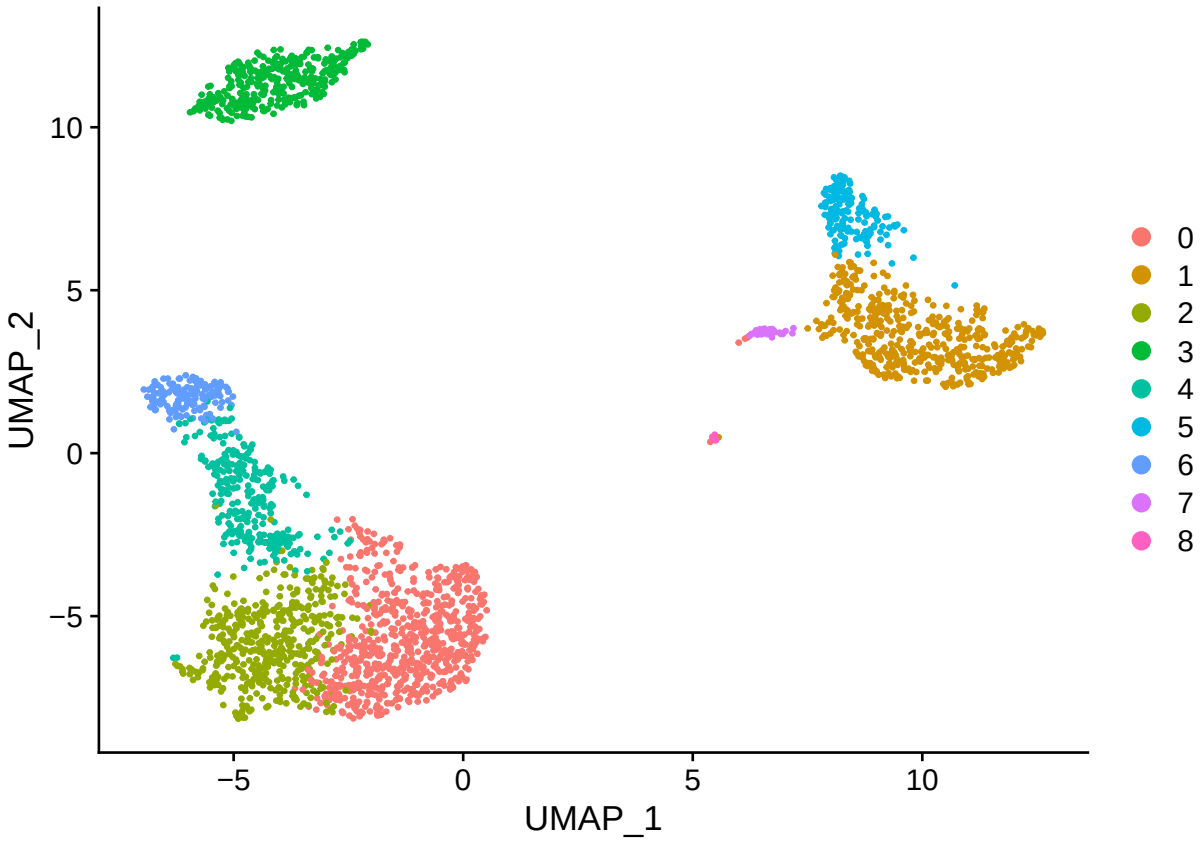
```
## AAACATACAACCAC-1 AAACATTGAGCTAC-1 AAACATTGATCAGC-1 AAACCGTGCTTCCG-1
##                2                3                2                1
## AAACCGTGTATGCG-1
##                6
## Levels: 0 1 2 3 4 5 6 7 8
```

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)
```

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



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 = "../output/pbmc_tutorial.rds")
```

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 2.593535e-91  1.2154360 0.949 0.466 3.556774e-87
## LTB  7.994465e-87  1.2828597 0.981 0.644 1.096361e-82
## CD3D 3.922451e-70  0.9359210 0.922 0.433 5.379250e-66
## IL7R 1.130870e-66  1.1776027 0.748 0.327 1.550876e-62
## LDHB 4.082189e-65  0.8837324 0.953 0.614 5.598314e-61
```

```
# 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     p_val_adj
## FCGR3A        2.150929e-209   4.267579 0.975 0.039 2.949784e-205
## IFITM3        6.103366e-199   3.877105 0.975 0.048 8.370156e-195
## CFD           8.891428e-198   3.411039 0.938 0.037 1.219370e-193
## CD68          2.374425e-194   3.014535 0.926 0.035 3.256286e-190
## RP11-290F20.3 9.308287e-191   2.722684 0.840 0.016 1.276538e-186
```

```
# find markers for every cluster compared to all remaining cells, report only the positive ones
pbmc.markers <- FindAllMarkers(pbmc, only.pos = TRUE, min.pct = 0.25, logfc.threshold = 0.25)
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 1.17e- 83       1.33 0.435 0.108 1.60e- 79 0       CCR7
##  2 1.74e-109       1.07 0.897 0.593 2.39e-105 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 7.99e- 87       1.28 0.981 0.644 1.10e- 82 2       LTB
##  6 2.61e- 59       1.24 0.424 0.111 3.58e- 55 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 4.93e-169       3.01 0.595 0.056 6.76e-165 4       GZMK
## 10 1.17e-178       2.97 0.957 0.241 1.60e-174 4       CCL5
## 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 6.82e-175       4.92 0.958 0.135 9.36e-171 6       GNLY
## 14 1.05e-265       4.89 0.986 0.071 1.44e-261 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-DPB1
## 17 3.68e-110       8.58 1     0.024 5.05e-106 8       PPBP
## 18 7.73e-200       7.24 1     0.01  1.06e-195 8       PF4
```

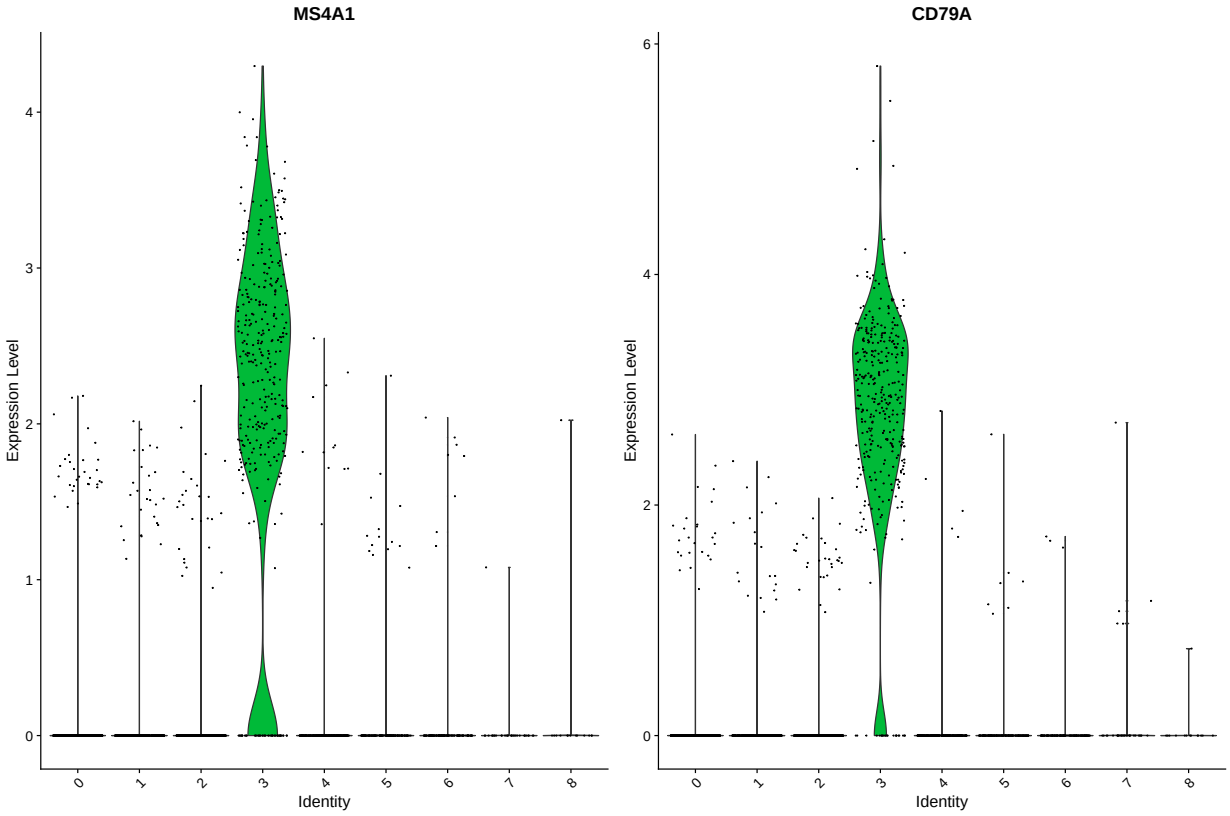
Seurat has several tests for differential expression which can be set with the test.use parameter (see our DE vignette for details). For example, the ROC test returns the 'classification power' for any individual marker (ranging from 0 - random, to 1 - perfect).

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

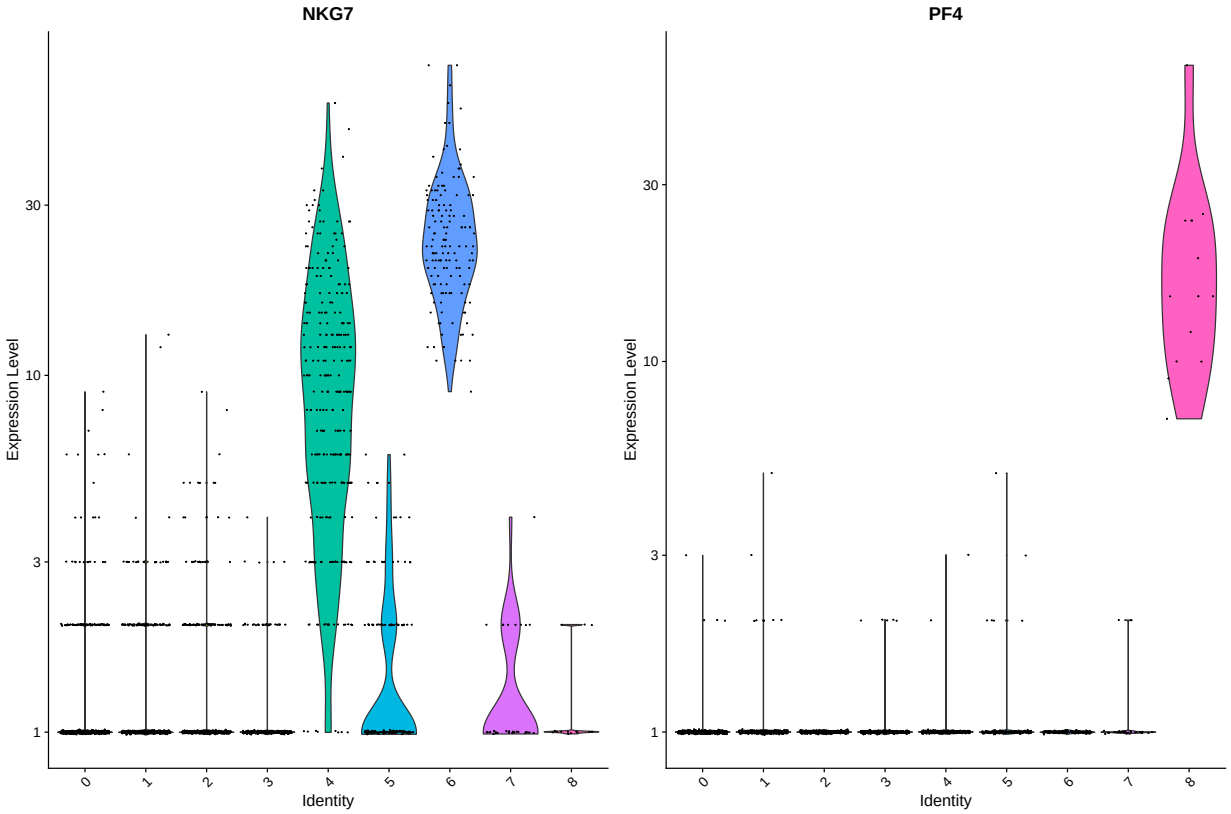
We include several tools for visualizing marker expression. `VlnPlot()` (shows expression probability distributions across clusters), and `FeaturePlot()` (visualizes feature expression on a tSNE or PCA plot) are

our most commonly used visualizations. We also suggest exploring `RidgePlot()`, `CellScatter()`, and `DotPlot()` as additional methods to view your dataset.

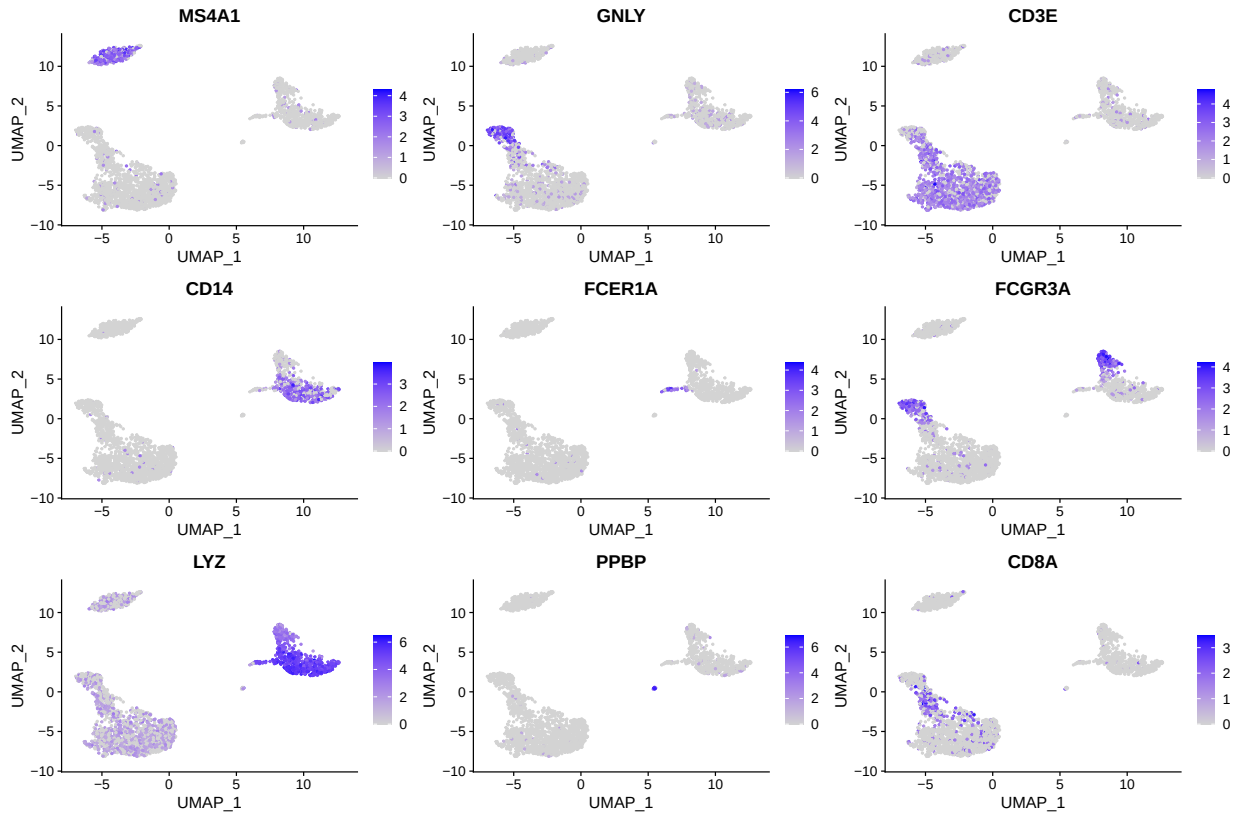
```
VlnPlot(pbmc, features = c("MS4A1", "CD79A"))
```




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

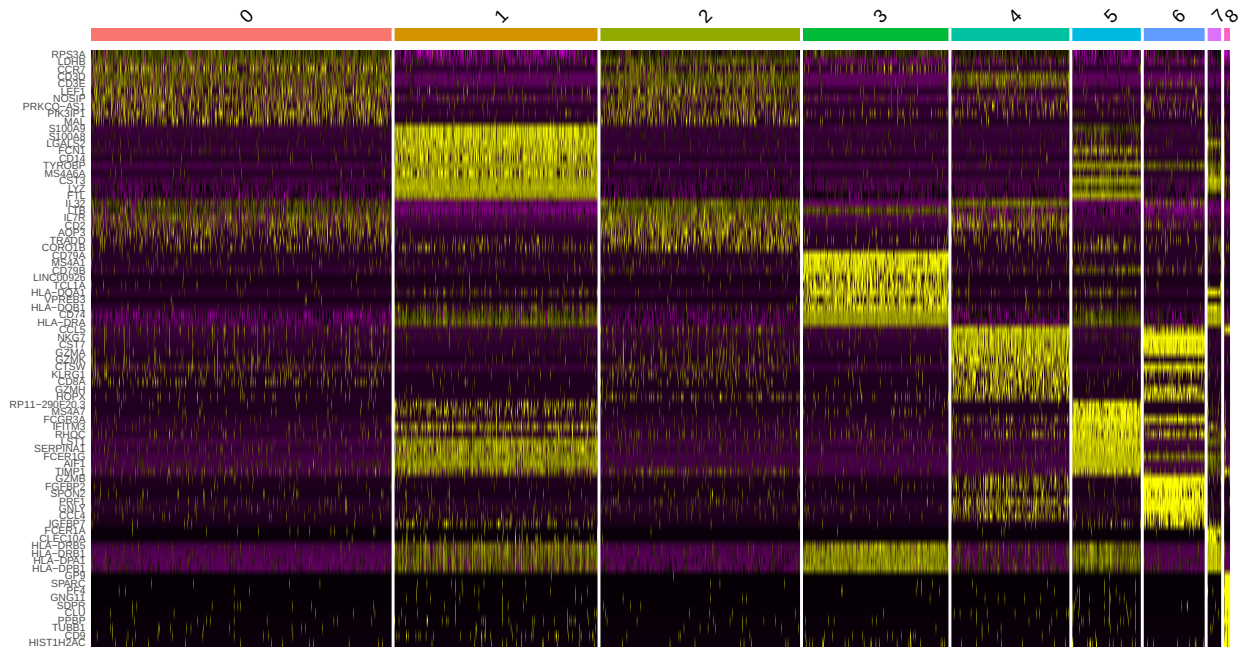



```
FeaturePlot(pbmc, features = c("MS4A1", "GNLY", "CD3E", "CD14", "FCER1A", "FCGR3A", "LYZ", "PPBP", "CD8
```



`DoHeatmap()` generates an expression heatmap for given cells and features. In this case, we are plotting the top 20 markers (or all markers if less than 20) for each cluster.

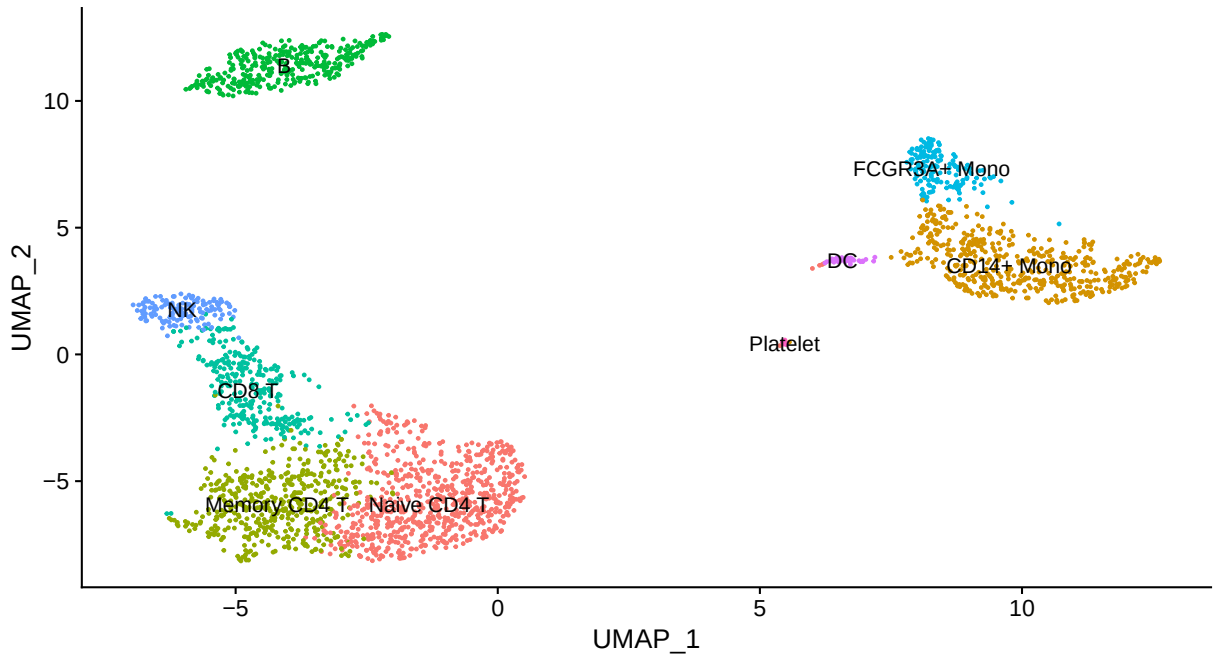
```
pbmc.markers %>% group_by(cluster) %>% top_n(n = 10, wt = avg_log2FC) -> top10
DoHeatmap(pbmc, features = top10$gene) + NoLegend()
```



*** # 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

```
new.cluster.ids <- c("Naive CD4 T", "CD14+ Mono", "Memory CD4 T", "B", "CD8 T", "FCGR3A+ Mono", "NK", "
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 = "../output/pbmc3k_final.rds")
```

Session Info

```
sessionInfo()
```

```
## R version 4.1.1 (2021-08-10)
## Platform: x86_64-conda-linux-gnu (64-bit)
## Running under: Ubuntu 20.04.2 LTS
##
## Matrix products: default
## BLAS/LAPACK: /home/verena/anaconda3/envs/seurat/lib/libopenblasp-r0.3.18.so
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=de_AT.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=de_AT.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=de_AT.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=de_AT.UTF-8 LC_IDENTIFICATION=C
```

```
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] ggplot2_3.3.5      patchwork_1.1.1    SeuratObject_4.0.2 Seurat_4.0.5
## [5] dplyr_1.0.7
##
## loaded via a namespace (and not attached):
##   [1] nlme_3.1-153          spatstat.sparse_2.0-0 matrixStats_0.61.0
##   [4] RcppAnnoy_0.0.19      RColorBrewer_1.1-2    httr_1.4.2
##   [7] sctransform_0.3.2     tools_4.1.1           utf8_1.2.2
##  [10] R6_2.5.1              irlba_2.3.3           rpart_4.1-15
##  [13] KernSmooth_2.23-20    uwot_0.1.10           mgcv_1.8-38
##  [16] lazyeval_0.2.2        colorspace_2.0-2      withr_2.4.2
##  [19] tidyselect_1.1.1      gridExtra_2.3         compiler_4.1.1
##  [22] cli_3.1.0             plotly_4.10.0         labeling_0.4.2
##  [25] scales_1.1.1          lmtest_0.9-39         spatstat.data_2.1-0
##  [28] ggridges_0.5.3        pbapply_1.5-0         goftest_1.2-3
##  [31] stringr_1.4.0         digest_0.6.28         spatstat.utils_2.2-0
##  [34] rmarkdown_2.11        pkgconfig_2.0.3       htmltools_0.5.2
##  [37] parallelly_1.28.1     highr_0.9             fastmap_1.1.0
##  [40] htmlwidgets_1.5.4     rlang_0.4.12          rstudioapi_0.13
##  [43] shiny_1.7.1           farver_2.1.0          generics_0.1.1
##  [46] zoo_1.8-9             jsonlite_1.7.2        ica_1.0-2
##  [49] magrittr_2.0.1        Matrix_1.3-4          Rcpp_1.0.7
##  [52] munsell_0.5.0         fansi_0.4.2           abind_1.4-5
##  [55] reticulate_1.22       lifecycle_1.0.1       stringi_1.7.5
##  [58] yaml_2.2.1            MASS_7.3-54           Rtsne_0.15
##  [61] plyr_1.8.6            grid_4.1.1            parallel_4.1.1
##  [64] listenv_0.8.0         promises_1.2.0.1      ggrepel_0.9.1
##  [67] crayon_1.4.2          deldir_1.0-6          miniUI_0.1.1.1
##  [70] lattice_0.20-45       cowplot_1.1.1         splines_4.1.1
##  [73] tensor_1.5            knitr_1.36            pillar_1.6.4
##  [76] igraph_1.2.8          spatstat.geom_2.3-0   future.apply_1.8.1
##  [79] reshape2_1.4.4        codetools_0.2-18      leiden_0.3.9
##  [82] glue_1.5.0            evaluate_0.14         data.table_1.14.2
##  [85] png_0.1-7             vctrs_0.3.8           httpuv_1.6.3
##  [88] polyclip_1.10-0       gtable_0.3.0          RANN_2.6.1
##  [91] purrr_0.3.4           spatstat.core_2.3-1   tidyr_1.1.4
##  [94] scattermore_0.7       future_1.23.0         xfun_0.28
##  [97] mime_0.12             xtable_1.8-4          RSpectra_0.16-0
## [100] later_1.2.0           survival_3.2-13       viridisLite_0.4.0
## [103] tibble_3.1.6          cluster_2.1.2         globals_0.14.0
## [106] fitdistrplus_1.1-6    ellipsis_0.3.2        ROCR_1.0-11
```

Pricing
API
Training
Blog
About