Cellustering Guided Tutorial

Zhecheng Ren (121090464)

This vignette offers you a guided tutorial on using `Cellustering`, a toolkit designed by Zhecheng Ren, Zhiyuan Gao, and Ang Li for the preprocessing and clustering analysis of single-cell RNA sequencing (scRNA-seq) data. In this tutorial, we will demonstrate the usage of our package through a streamlined cell-clustering analysis of a Peripheral Blood Mono-nuclear Cells (PBMC) 10X Genomics dataset. We will also illuminate the principles behind methodologies used in our package, and underscore the distinctive advantages that set `Cellustering` apart from previous solutions in the field.

```
library(cellustering)
```

Data Loading

We start by reading in the scRNA-seq data.

`read_10x()`: Create `Cellustering` instance from 10X Genomics data

`read_10x()` function reads in the data from a file directory containing the outputs of the Cell Ranger pipeline from 10X. Notice that the file format is required to be the Market Exchange Format (MEX). `read_10x()` combines the data in these files and generates a UMI count matrix. A `Cellustering` instance with count matrix stored in the `data` slot will be automatically created and returned.

```
# pbmc <- read_10x("/Users/ren/Documents/hg19")
# You can read in the data using directory like above
# Here we directly read in the data attached in our package like below
pbmc <- read_10x(cellustering_example("hg19"))
pbmc@data[c("CD3D", "TCL1A", "MS4A1"), 1:4]
#>       AAACATACAACCAC.1 AAACATTGAGCTAC.1 AAACATTGATCAGC.1 AAACCGTGCTTCCG.1
#> CD3D                4                0               10                0
#> TCL1A               0                0                0                0
#> MS4A1               0                6                0                0
```

Advantages of `Cellustering` over others

As the pivotal tool within the realm of scRNA-seq data analysis, Cell Ranger standardizes its output in the MEX format. `Cellustering` can directly read in the data in this format, eliminating the need for manual data conversion steps required by other packages such as `SC3` and `TSCAN`.

Connection to course materials

- Testing: `read_10x()` checks whether user inputs a valid directory containing files in the MEX format.
- Data structures: `read_10x()` manipulates data structures like matrix and data frame.
- Data loading: `read_10x()` reads in the data through a directory.

`Cellustering` class

In `Cellustering` package, every function operates on an instance of the `Cellustering` class. All data and plots produced during the analysis process are stored in the corresponding slots of this instance. The `Cellustering` class containing the following slots:

- data: Stores a data frame containing the UMI count matrix. Functions in the subsequent steps will access and may overwrite this data frame.
- quality: Stores the data and plots produced during the quality control steps.
- HVG: Stores the names and plots of genes selected during the feature selection step.
- reduced_dimension: Stores the data and plots produced during the dimension reduction step.
- clustering: Stores the data and plots produced during the clustering step.
- progress: Stores the progress of the whole analysis process.

The slots of an `Cellustering` instance can be accessed through the `@` operator.

`pbmc` data

The `pbmc` data is stored in a UMI count matrix with dimension 32738 $\times$ 2700. The values in this matrix represent the count of molecules for each gene (i.e. feature; row) that are detected in each cell (i.e. cellular barcode; column). The raw data can be found here.

```
dim(pbmc@data)
#> [1] 32738  2700
```

Quality Control

Before analysing the single-cell gene expression data, we must ensure that all cellular barcodes correspond to viable cells, and all genes do express in some cells. Therefore, the quality control (QC) is needed.

Cell QC is commonly performed based on three QC covariates: the number of counts per cell (count depth), the number of genes per cell (feature counts), and the fraction of counts from mitochondrial genes per cell (mitochondrial percent) (Ilicic et al, 2016; Griffiths et al, 2018). The outlier cells in these covariates can correspond to dying cells, cells whose membranes are broken, or doublets, thus awaiting to be filtered out.

- Indicators of dying cells and cells whose membranes are broken:
  - Low count depth.
  - Few detected genes.
  - High fraction of mitochondrial counts.
- Indicators of doublets:
  - High count depth.
  - High detected genes.

According to Luecken and Theis (2019), considering any of these three cell QC covariates in isolation can lead to misinterpretation of cellular signals. For example, cells with a comparatively high fraction of mitochondrial counts may be involved in respiratory processes. Thus, cell QC covariates should be considered jointly when univariate thresholding decisions are made.

Gene QC is commonly performed based on the number of counts per gene (total expression) and the number of cells per gene (cell counts). Genes with low values in these two covariates need to be filtered out.

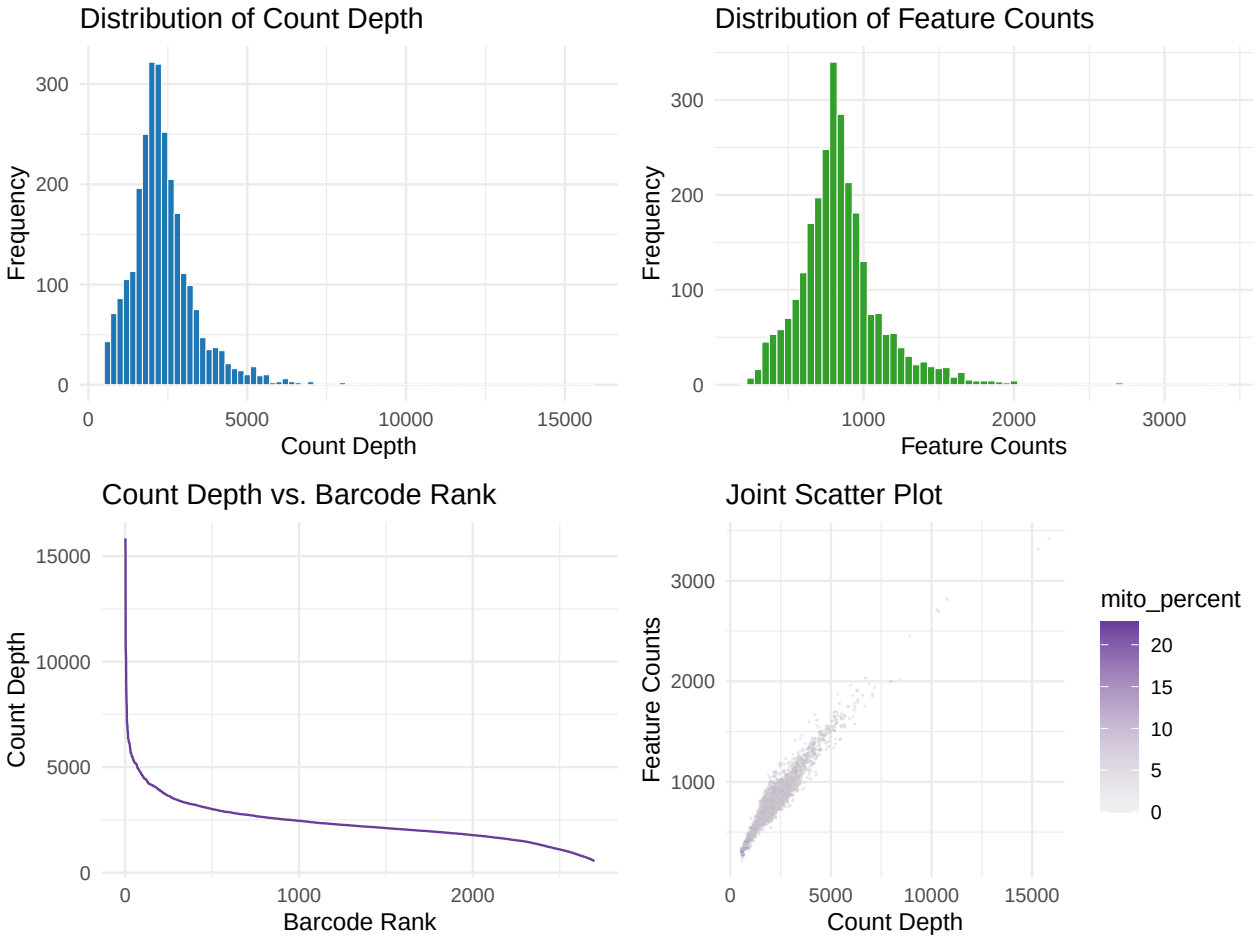
`qc_plot()`: Create plots to facilitate quality control

To help determine the thresholds for cell QC, two plots are created. The first plot contains the distributions of three cell QC covariates and their joint scatter. The second plot contains three vilon plots regarding these covariats.

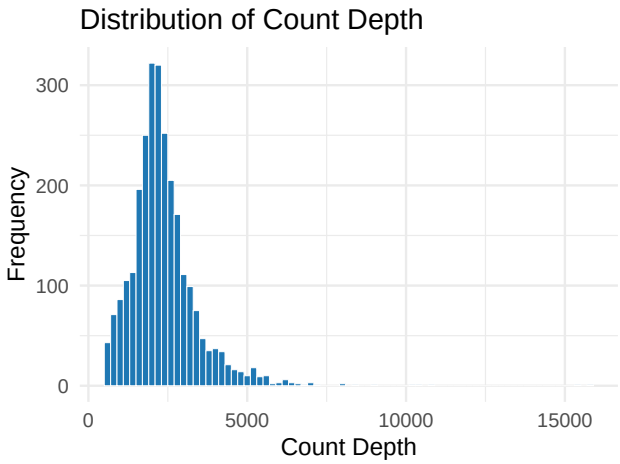
To help determine the thresholds for gene QC, one plot containing the truncated (x-axis value < 25) distributions of two gene QC covariates is generated.

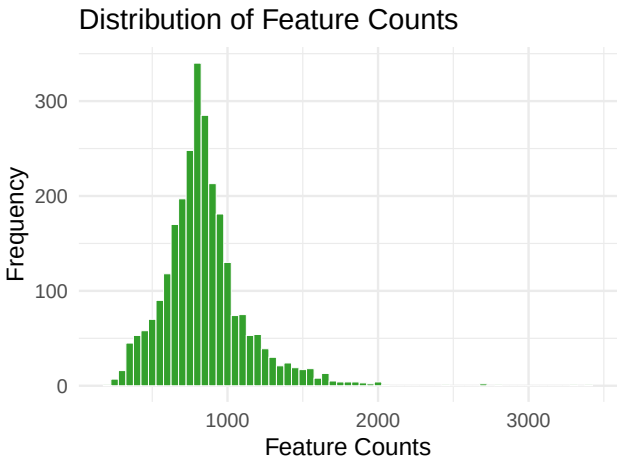
All relevant statistics and plots are added to the `quality` slot of the `Cellustering` instance. `qc_plot()` automatically shows the first plot for cell QC and returns the `Cellustering` instance.

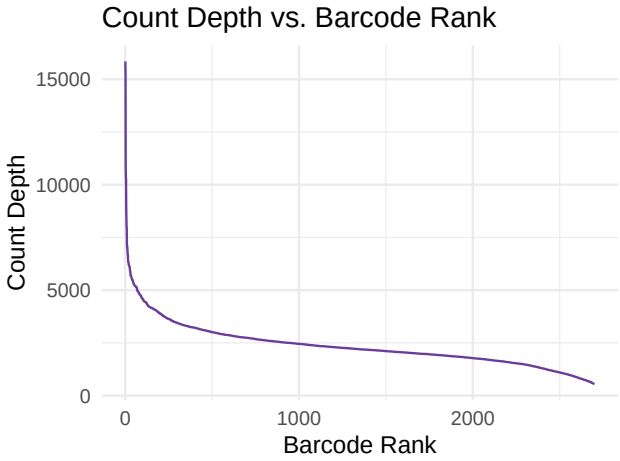
```
pbmc <- qc_plot(pbmc)
```

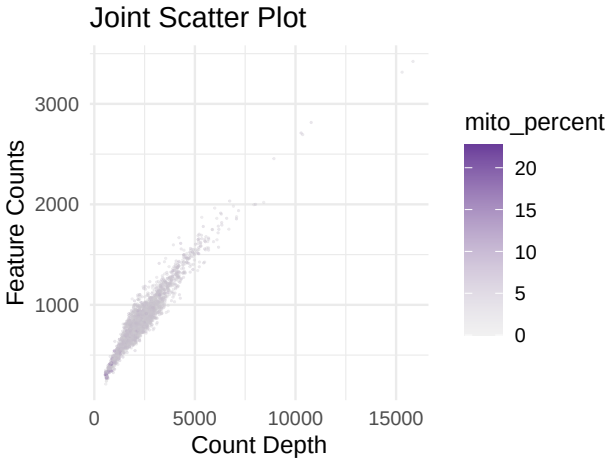



```
head(pbmc@quality$cell)
#>                  count_depth feature_count mito_percent
#> AAACATACAACCAC.1        2421           781     3.097893
#> AAACATTGAGCTAC.1        4903          1352     3.875178
#> AAACATTGATCAGC.1        3149          1131     1.079708
#> AAACCGTGCTTCCG.1        2639           960     1.970443
#> AAACCGTGTATGCG.1         981           522     1.427115
#> AAACGCACTGGTAC.1        2164           782     1.940850
head(pbmc@quality$feature)
#>              total_expression cell_count
#> MIR1302.10                  0          0
#> FAM138A                     0          0
#> OR4F5                       0          0
#> RP11.34P13.7                0          0
#> RP11.34P13.8                0          0
#> AL627309.1                  9          9
pbmc@quality$cell_plot
```

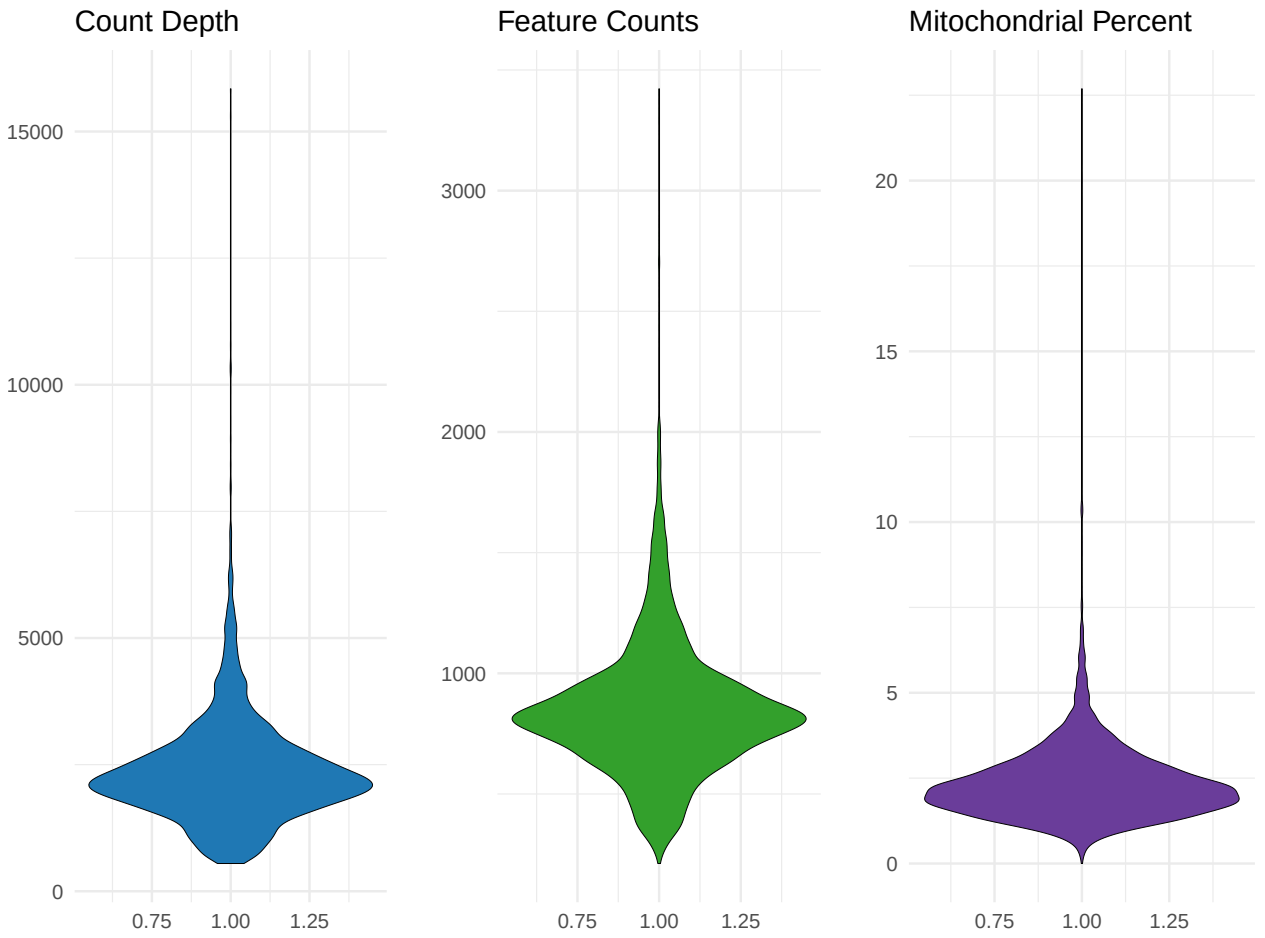




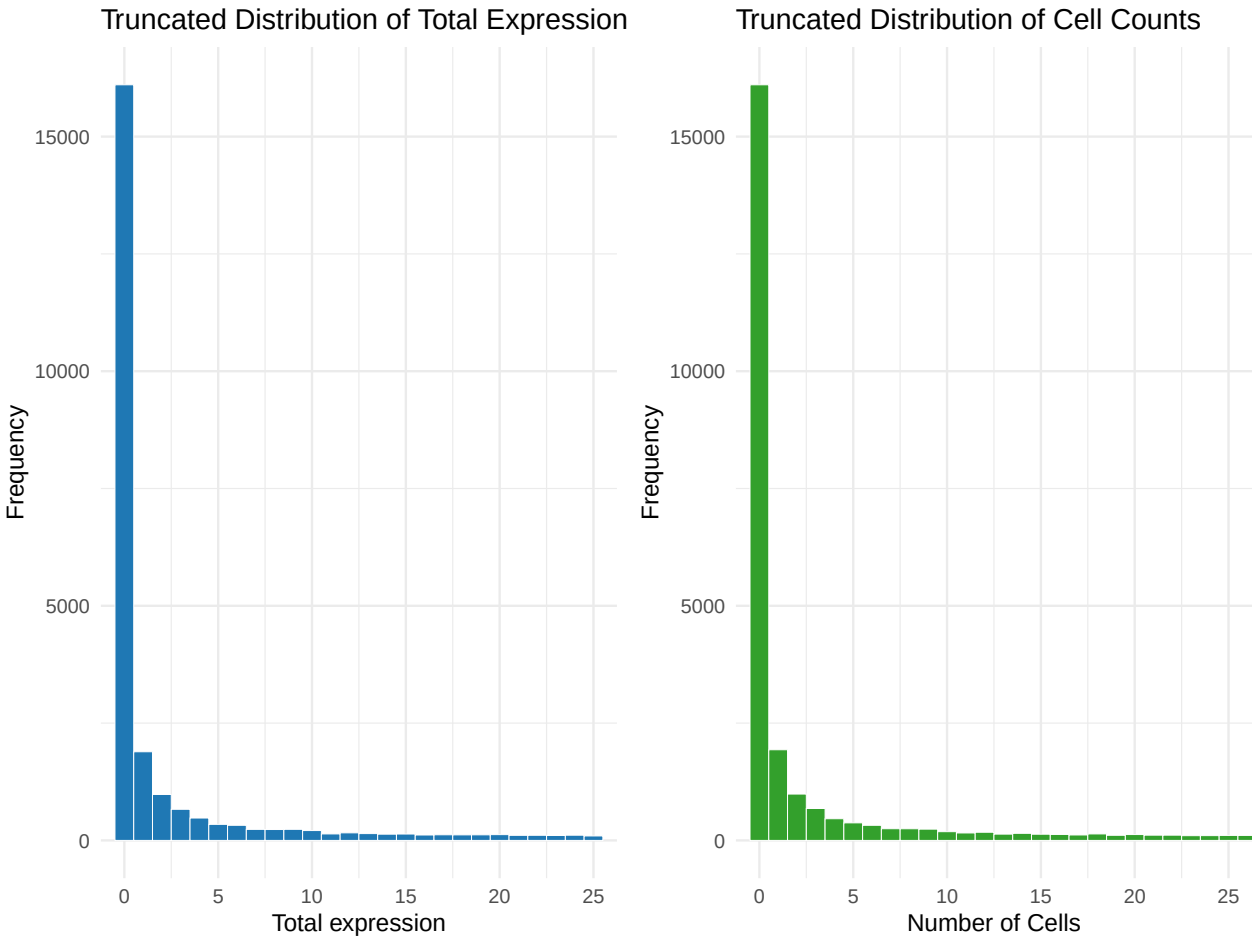





```
pbmc@quality$violin_plot
```




```
pbmc@quality$feature_plot
```



Advantages of `Cellustering` over others

For packages like `SC3` and `TSCAN`, they don't have built-in functions for QC.

For packages like `Seurat`, they have built-in functions to plot QC covariates, but

- The fraction of mitochondrial counts needs to be manually computed.
- The cell QC covariates are isolatedly plotted.
- Lacking a function to integrate different kinds of plots.
- Lacking plots for gene QC covariates.

Connection to course materials

- Testing: `qc_plot()` checks whether user inputs a `Cellustering` instance.
- Data frame manipulation: `qc_plot()` conducts column sum, row sum, and ranking over the data frame.
- ggplot2: `qc_plot()` uses `ggplot2` package to plot.
- Regular expression: `qc_plot()` uses regular expression and text manipulation to search for mitochondrial genes.

`qc_filter()`: Filter out low-quality cells and genes

According to the plots created by `qc_plot()`, we can set conditions over the QC covariates to filter out low-quality cells and genes. `qc_filter()` will do the filtering and return the `Cellustering` instance.

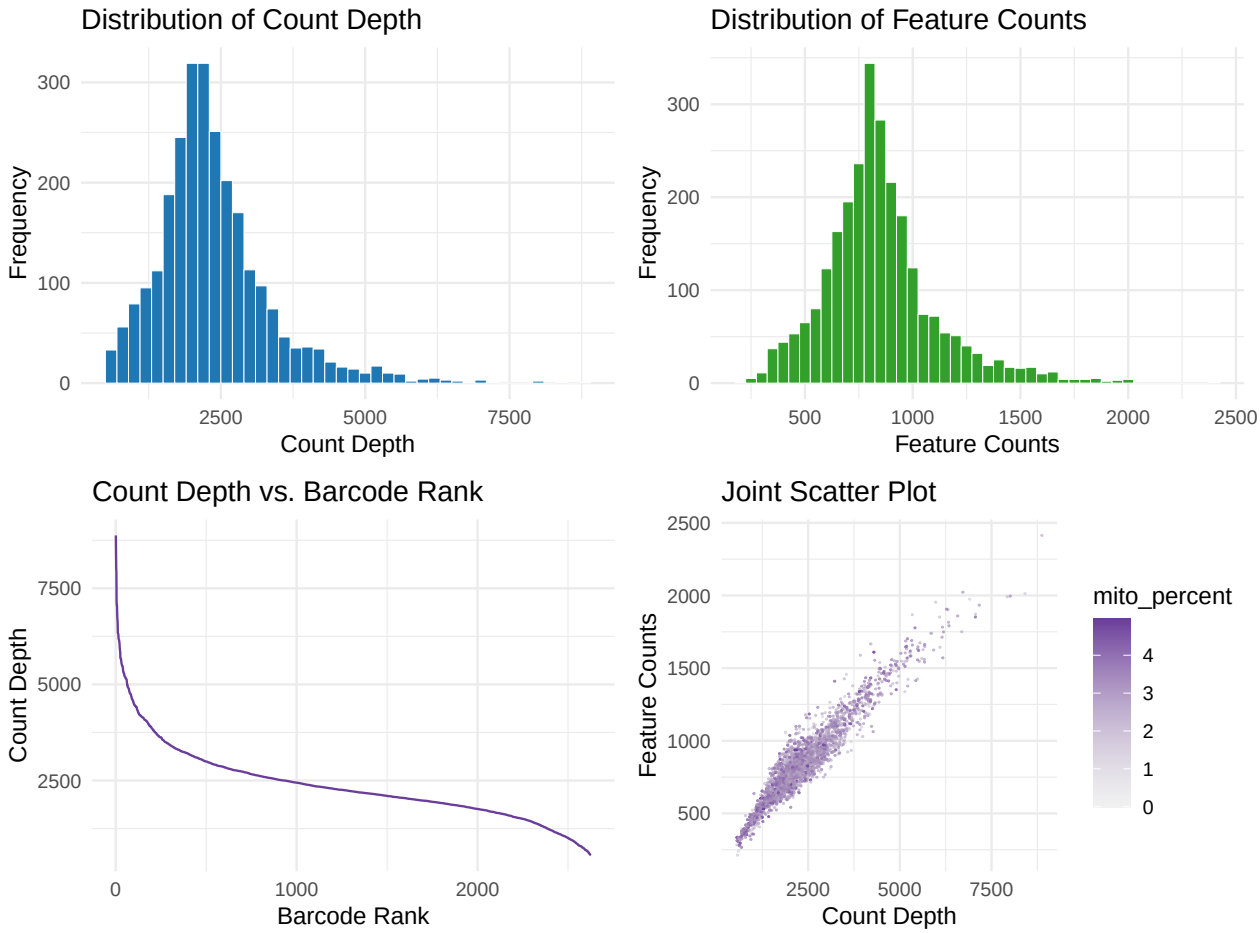
In the example here, we only retain cells and genes satisfying the following conditions:

- The number of genes per cell (feature counts) $>$ 200 & $<$ 2500.
- The fraction of counts from mitochondrial genes per cell (mitochondrial percent) $<$ 5%.
- The number of cells per gene (cell counts) $>$ 3.

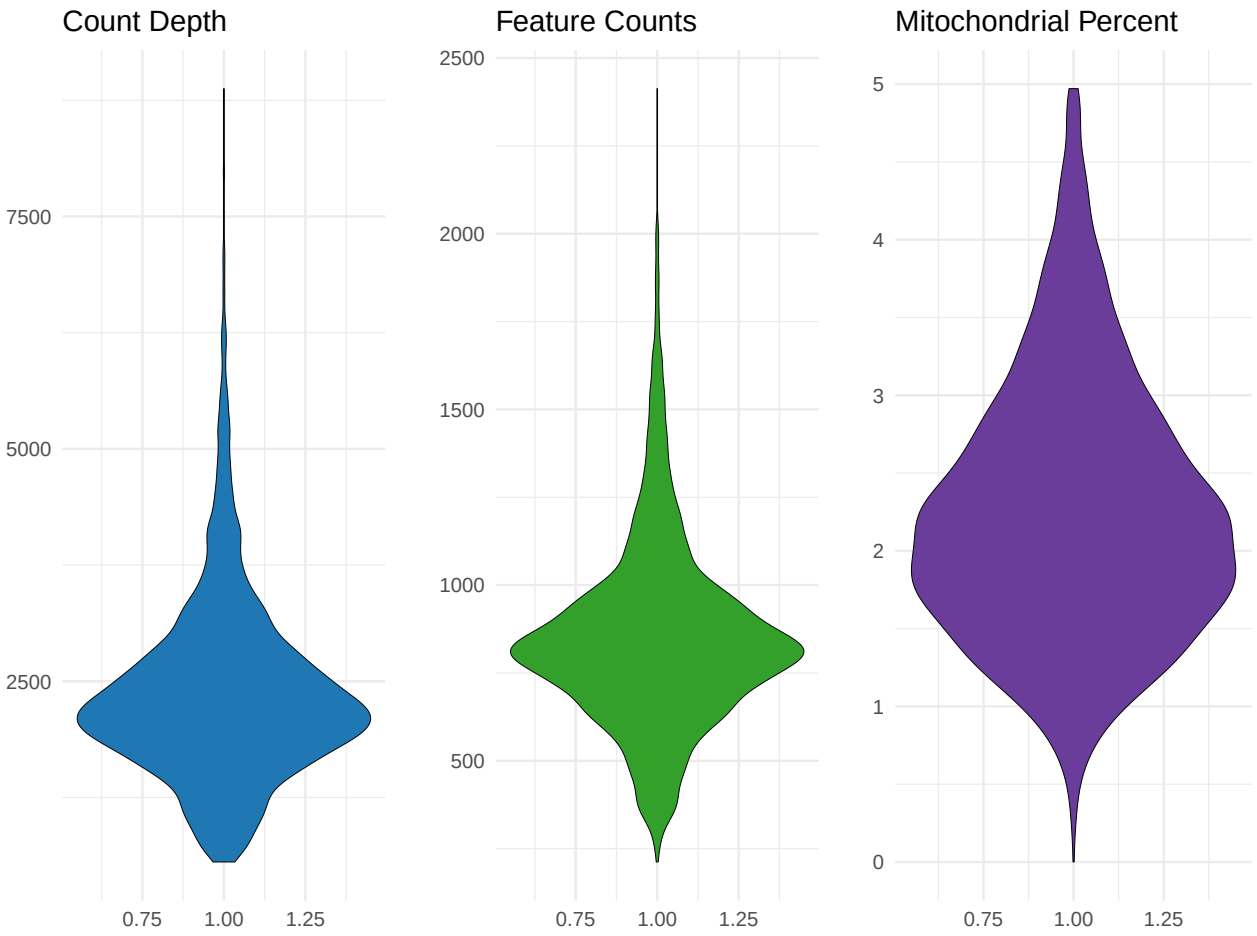
```
pbmc <- qc_filter(pbmc,
  min_feature = 200,
  max_feature = 2500,
  max_mito_percent = 5,
  min_cell = 3
)
```

We can check the dimension of the count matrix and QC plots again:

```
dim(pbmc@data)
#> [1] 13714  2626
pbmc <- qc_plot(pbmc)
```




```
pbmc@quality$violin_plot
```



We can see that 19024 cells and 74 genes are filtered out.

Advantages of `Cellustering` over others

For packages like `Seurat`, they don't have built-in functions for filtering, user has to manually filter out genes and cells.

Connection to course materials

- Testing: `qc_filter()` checks whether user input has correct data type and appropriate value.
- Indexing: `qc_filter()` uses indexing with booleans on-the-fly to achieve the filtering.

Data normalization

Due to the variations in total count depths among different cells, the magnitudes of gene expression counts of two cells may not be on a comparable scale. To remove the effect of count depth, we need to conduct cell-level normalization.

The most commonly used normalization protocol is count depth scaling, also referred to as Counts Per Million" or CPM normalization (Luecken & Theis, 2019). It scales the expression counts of different cells and unifies their count depths to 1e6. Variations of this method set the scale factors to be different factors of 10.

After normalization, the count matrix is typically log(x+1)-transformed. This transformation has three important effects:

- Distances between log-transformed expression values represent log fold changes, which are the canonical way to measure changes in expression.
- Log transformation mitigates (but does not remove) the mean–variance relationship in single-cell data (Brennecke et al, 2013).
- Log transformation reduces the skewness of the data to approximate the assumption of many downstream analysis tools that the data are normally distributed.

`normalize()`: Conduct cell-level normalization

The `normalize()` function allows us to normalize the count matrix with CPM protocol and its variation. We can also control whether to apply log(x+1) transformation by setting the `log_transformation` parameter. The count matrix in the `data` slot is overwritten by its normalized version. And the `Cellustering` instance is returned by the `normalize` function.

```
pbmc <- normalize(pbmc, scale_factor = 1e6, log_transformation = TRUE)
```

Advantages of `Cellustering` over others

Compared to other packages like `Seurat`, the variations of CPM are also employed by `Cellustering`. User is able to tune the scale factor since its value may impact the analysis in the presence of technical dropout effects.

Connection to course materials

- Testing: `normalize()` checks whether user input has correct data type and appropriate value.
- Dplyr: `normalize()` uses `Dplyr` to achieve CPM and its variations.

Feature Selection

A human single-cell RNA-seq dataset can have over 15,000 dimensions even after filtering out these low-count genes in the QC step. Many of these genes will not be informative for the clustering task. To ease the computational burden on downstream analysis, we consider keeping only genes that are "informative" of the variability in the data. According to Brennecke et al (2013), highly variable genes (HVGs) can effectively preserve the data variability. Typically, between 1,000 and 5,000 HVGs are selected for downstream analysis.

`find_HVG()`: Identify, visualize, and keep HVGs

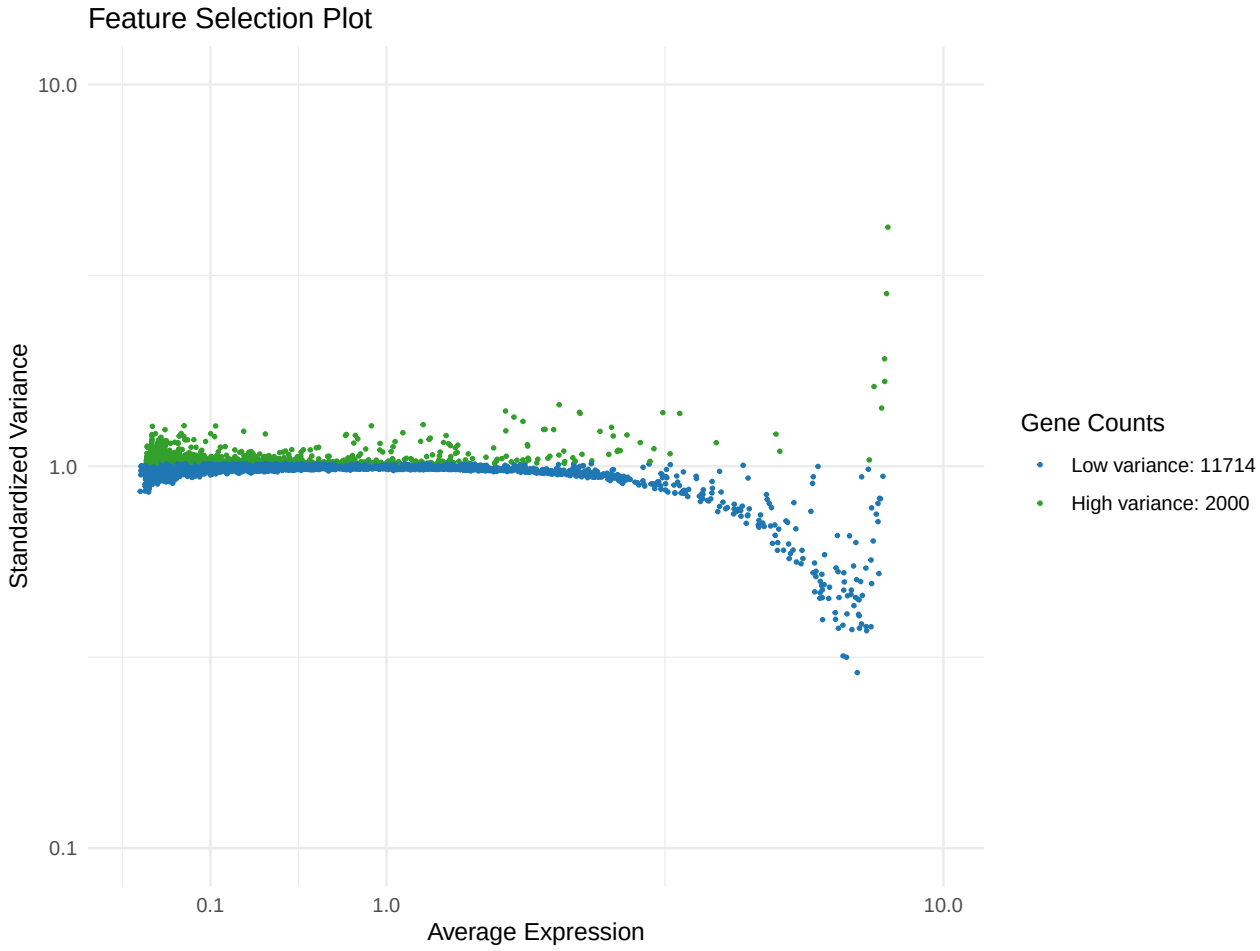
The `find_HVG()` function allows us to select HVGs through the variance stabilizing transformation (VST) algorithm. The algorithm takes the following steps:

1. The sample mean and sample variance of the expression counts for each gene are calculated.
2. A local polynomial regression model is fitted over the means and variances.
3. Predict each gene's sample variance using the fitted model and sample mean.
4. Standardize the count matrix using sample means and predicted variances.
5. Clipping the standardized values if it's too large.
6. Recompute the sample variance of the expression counts for each gene.
7. Rank the recomputed sample variances and select HVGs.

We can control the smoothness of the local polynomial regression model by tuning the `loess_span` parameter. The HVGs' names and relevant plots are stored in the `HVG` slot. The `Cellustering` instance is returned by the `normalize()` function.

```
pbmc <- find_HVG(pbmc, n_feature = 2000, loess_span = 0.5)
#> Warning in sqrt(predicted_vars): NaNs produced
pbmc@HVG$HVG_plot
```

```
#> Warning: Removed 18 rows containing missing values or values outside the scale range
#> (`geom_point()`).
```



Advantages of `Cellustering` over others

For packages like `Seurat`, they employ a binning method in the analysis to increase the accuracy at the cost of computing time. However, preliminary results from Klein et al (2015) suggest that downstream analysis is robust to the exact choice of the number of HVGs. While varying the number of HVGs, the authors reported similar low-dimensional representations in the PCA space, meaning selecting a small number of HVGs with high accuracy can be substituted by selecting more HVGs with lower accuracy. Based on that, `Cellustering` dispenses with the binning method and increases the speed of the algorithm.

Connection to course materials

- Testing: `find_HVG()` checks whether the count matrix has gone through the normalization step and whether user input has correct data type and appropriate value.
- Apply: `find_HVG()` uses `apply()` to compute the sample means and sample variances of gene expressions.
- Model fitting: `find_HVG()` fits a local polynomial regression model (by calling `loess()`) and make predictions on data.
- ggplot2: `find_HVG()` uses `ggplot2` package to plot.

Dimension Reduction

After feature selection, the dimensions of count matrix can be further reduced by dimension reduction algorithms, which capture the underlying structure in the data in a low-dimensional space. The two main objectives of dimension reduction are visualization and summarization.

- Visualization: Optimally describes the count matrix in two or three dimensions. The dimensions reduced to are used as coordinates for scatter plot to obtain a visual representation of the count matrix.
- Summarization: The number of output components is larger than two or three. It aims to reduce the count matrix to its essential components and find the inherent structure of the data.

`Cellustering` employs Principal Component Analysis (PCA) (Pearson, 1901), a dimension reduction technique that fulfills the above two objectives.

Principle of PCA:

Suppose the input is a dataset $D = \left\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)}\right\} \subset \mathbb{R}^D$*, with* D *being the original dimension. Our goal is to find a* K*-dimensional* $(K < D)$ *subspace* $\mathbb{S}$*, which consists of* K *orthonormal basis vectors* $\{\boldsymbol{u}_k\}_{k=1}^K$*. When projecting all points in* $\mathbb{R}^D$ *onto* $\mathbb{S}$*, it is desired that the structure or property of the original data is well preserved.*

Therefore, we set the objective as maximizing the variance of the reconstructions $\tilde{D} = \left\{\tilde{\mathbf{x}}^{(1)}, \ldots, \tilde{\mathbf{x}}^{(n)}\right\}$ *(We can easily show that this objective is equivalent to minimizing the reconstruction loss using Pythagoras Theorem):*

$$\max_{U, U^T U = I} \frac{1}{N} \sum_{i=1}^{N} \left\| \tilde{\mathbf{x}}^{(n)} - \tilde{\boldsymbol{\mu}} \right\|^2$$

where $\tilde{\boldsymbol{\mu}} = \frac{1}{N} \sum_{i=1}^{N} \tilde{\boldsymbol{x}}^{(n)}$ *is the mean of the reconstructions.*

Notice that the objective function doesn't contain the decision variable U*, therefore, we need to further process the objective function.*

According to the definition of $\tilde{\boldsymbol{x}}$ *and* $\mathbf{z}^{(n)}$*, we have:*

$$\begin{aligned} \tilde{\boldsymbol{\mu}} &= \frac{1}{N} \sum_{i=1}^{N} \tilde{\mathbf{x}}^{(n)} \\ &= \frac{1}{N} \sum_{i=1}^{N} \left(\boldsymbol{\mu} + U\mathbf{z}^{(n)}\right) \\ &= \boldsymbol{\mu} + \frac{U}{N} \sum_{i=1}^{N} \left(U^T \left(\mathbf{x}^{(n)} - \boldsymbol{\mu}\right)\right) \\ &= \boldsymbol{\mu} \end{aligned}$$

Therefore, the objective function can be converted into:

$$\begin{aligned} \frac{1}{N} \sum_{i=1}^{N} \left\| \tilde{\mathbf{x}}^{(n)} - \tilde{\boldsymbol{\mu}} \right\|^2 &= \frac{1}{N} \sum_{i=1}^{N} \left\| \boldsymbol{\mu} + U\mathbf{z}^{(n)} - \boldsymbol{\mu} \right\|^2 \\ &= \frac{1}{N} \sum_{i=1}^{N} \left\| \mathbf{z}^{(n)} \right\|^2 \end{aligned}$$

Utilizing the definition of $\mathbf{z}^{(n)}$*, we have:*

$$\begin{aligned} \max_{U, U^T U = I} \frac{1}{N} \sum_{i=1}^{N} \left\| \tilde{\mathbf{x}}^{(n)} - \tilde{\boldsymbol{\mu}} \right\|^2 &\Leftrightarrow \max_{U, U^T U = I} \frac{1}{N} \sum_{i=1}^{N} \left\| \mathbf{z}^{(n)} \right\|^2 \\ &\Leftrightarrow \max_{U, U^T U = I} \frac{1}{N} \sum_{i=1}^{N} \left\| U^T \left(\mathbf{x}^{(n)} - \boldsymbol{\mu}\right) \right\|^2 \\ &\Leftrightarrow \max_{U, U^T U = I} \frac{1}{N} \sum_{i=1}^{N} \operatorname{Trace}\left(U^T \left(\mathbf{x}^{(n)} - \boldsymbol{\mu}\right) \left(\mathbf{x}^{(n)} - \boldsymbol{\mu}\right)^T U \right) \end{aligned}$$

To solve this problem, we define the empirical covariance matrix as:

$$\Sigma = \frac{1}{N}\sum_{i=1}^{N}\left(\mathbf{x}^{(n)} - \boldsymbol{\mu}\right)\left(\mathbf{x}^{(n)} - \boldsymbol{\mu}\right)^T$$

Then the optimization problem becomes:

$$\max_{U, U^TU=I} \frac{1}{N}\sum_{i=1}^{N} \text{Trace}\left(U^T\Sigma U\right) = \sum_{k=1}^{K}\mathbf{u}_k^T\Sigma\mathbf{u}_k$$

The Lagrangian function is:

$$L\left(U, \Lambda_k\right) = \text{Trace}\left(U^T\Sigma U\right) + \text{Trace}\left(\Lambda_K^T\left(I - U^TU\right)\right)$$

where $\Lambda_K = \text{diag}\left(\hat{\lambda}_1, \ldots\ldots, \hat{\lambda}_k\right) \in \mathbb{R}^{K\times K}$.

The optimal solution satisfies:

$$\frac{\partial L\left(U, \Lambda_k\right)}{\partial U} = 2\Sigma U - 2U\Lambda_K = 0$$

which is equivalent to $\Sigma\boldsymbol{u}_k = \hat{\lambda}_k\boldsymbol{u}_k,\ k = 1, \ldots, K$.

Therefore, the primal optimal solution $\boldsymbol{u}_k$ *and the dual optimal solution* $\hat{\lambda}_k$ *are a pair of eigenvector and eigenvalue of* Σ. *And they satisfy* $U^TU = I$.

By SVD decomposition, we have:

$$\Sigma = Q\Lambda_D Q^T = \sum_{i=1}^{D}\lambda_i\mathbf{q}_i\mathbf{q}_i^T$$

- $Q = [\boldsymbol{q}_1, \ldots\ldots, \boldsymbol{q}_D] \in \mathbb{R}^{D\times D}$, $\boldsymbol{q}_i$ *corresponds to the i-th largest eigenvalue* λ_i.

- $\Lambda_D = \text{diag}\left(\lambda_1, \ldots\ldots, \lambda_D\right) \in \mathbb{R}^{D\times D}$, *and* $\lambda_1 \geq \cdots \geq \lambda_D$.

Plug into the objective function, we have:

$$\begin{array}{rcl} \sum_{k=1}^{K}\mathbf{u}_k{}^T\Sigma\mathbf{u}_k & = & \sum_{k=1}^{K}\mathbf{u}_k{}^T\left(\sum_{i=1}^{D}\lambda_i\mathbf{q}_i\mathbf{q}_i{}^T\right)\mathbf{u}_k \\ & = & \sum_{k=1}^{K}\sum_{i=1}^{D}\lambda_i\left(\mathbf{u}_k{}^T\mathbf{q}_i\right)\left(\mathbf{q}_i{}^T\mathbf{u}_k\right) \\ & = & \sum_{t\in T}\lambda_t \end{array}$$

where $T \subset \{1, \ldots, D\}$ *and* $|T| = K$.

Therefore, in order to maximize the objective function, we need to pick the top-K eigenvalues and their corresponding eigenvectors. As a result, we pick the top-K columns of Q *to form* U.

`scale_data()`: Scale and center the selected features

From the above derivation, we can see that PCA applies eigenvalue decomposition on a empirical covariance matrix. Therefore, `Cellustering` provides us the `scale_data()` function, which scales each HVG's expression counts to have mean 0 and variance 1, thus preparing for the subsequent PCA transformation.

`scale_data()` indexes the HVGs in the `data` slot with their names, stores the scaled HVG counts matrix in the `reduced_dimension` slot, and returns the `Cellustering` instance.

```
pbmc <- scale_data(pbmc)
pbmc@reduced_dimension$scaled_data[1:5, 1:4]
#>        AAACATACAACCAC.1 AAACATTGAGCTAC.1 AAACATTGATCAGC.1 AAACCGTGCTTCCG.1
#> RPS27         0.0374304      -0.09037671     0.5004711546        0.3059870
#> RPS14         0.3036236      -0.14136687    -0.5745393536       -1.0512154
#> RPS19         0.5267837       0.53713918     0.0006096829       -0.1688557
#> EEF1A1        0.3854507       0.25304065    -0.2187756621       -0.1955008
#> RPL18A        0.3538065       0.46123636    -0.9000086570        0.1146678

# Check if the row sums equal to 0
row_sums <- rowSums(pbmc@reduced_dimension$scaled_data)
all.equal(row_sums, rep(0, 2000), check.attributes = FALSE)
#> [1] TRUE

# Check if the row sd equal to 1
row_sds <- apply(pbmc@reduced_dimension$scaled_data, 1, sd)
all.equal(row_sds, rep(1, 2000), check.attributes = FALSE)
#> [1] TRUE
```

Connection to course materials

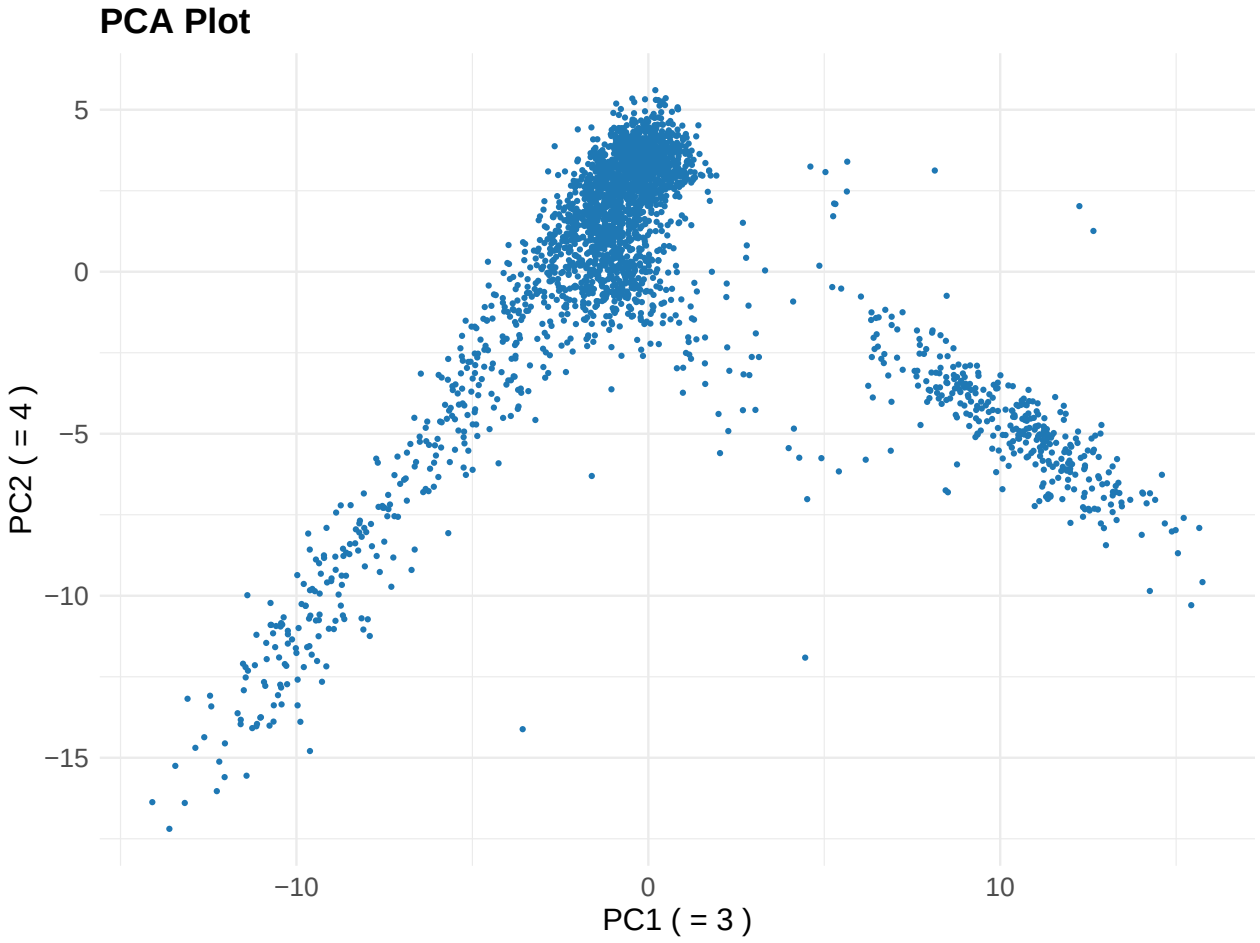
- Testing: `scale_data()` checks whether user inputs a `Cellustering` instance.
- Apply: `scale_data()` uses `apply()` to compute the standard deviation for each gene.
- Indexing: `scale_data()` uses indexing with name to select HVG data.

`principal_component_analysis()`: Apply PCA on data

From the above derivation, we can see the remarkable relationship between PCA and eigenvalue decomposition (or singular value decomposition). Therefore, the `principal_component_analysis()` function is provided to compute the eigenvalues and eigenvectors of the covariance matrix.

We can specify the two principal components (PCs) to visualize. A two-dimensional visualization of HVG count matrix is automatically outputed by `principal_component_analysis()`. This function stores all the relevant data and plots in the `reduced_dimension` slot, and returns the `Cellustering` instance.

```
pbmc <- principal_component_analysis(pbmc, PC1 = 3, PC2 = 4)
```



Connection to course materials

- Testing: `principal_component_analysis()` checks whether the HVG count matrix has gone through the scaling step and whether user input has correct data type and appropriate value.
- ggplot2: `principal_component_analysis()` uses `ggplot2` package to plot.

`select_proper_dimension`: Select dimensions for summarization

As mentioned before, we would like to pick out the essential PCs (say, PC 1 to PC L) of HVG count matrix and embed the data into this low-dimension space for the subsequent clustering analysis. To determine the value of L, `Cellustering` produces scree plot and profile log-likelihood plot to facilitate and automatically picks the optimized L.

Scree plot

From the above derivation for PCA, we can easily show that the reconstruction error $E(D, L) = \frac{1}{|D|}\sum_{i\in D}||\mathbf{x}_i - \tilde{\mathbf{x}}_i||^2$ is equivalent to $\sum_{k=L+1}^{D}\lambda_k$. Therefore, we can plot $\sum_{k=L+1}^{D}\lambda_k$ vs L and figure out the "regime change" (from relatively large errors to relatively small).

Profile log-likelihood plot

Since the scree plot is monotone, we cannot automate the detection of L by directly picking the minimal or maximal. One way to solve this is to use the profile log-likelihood plot.

- Consider partitioning the eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_{L_{max}}$ into two groups, depending on whether $k < L$ or $k > L$, where L is some threshold which we will determine.

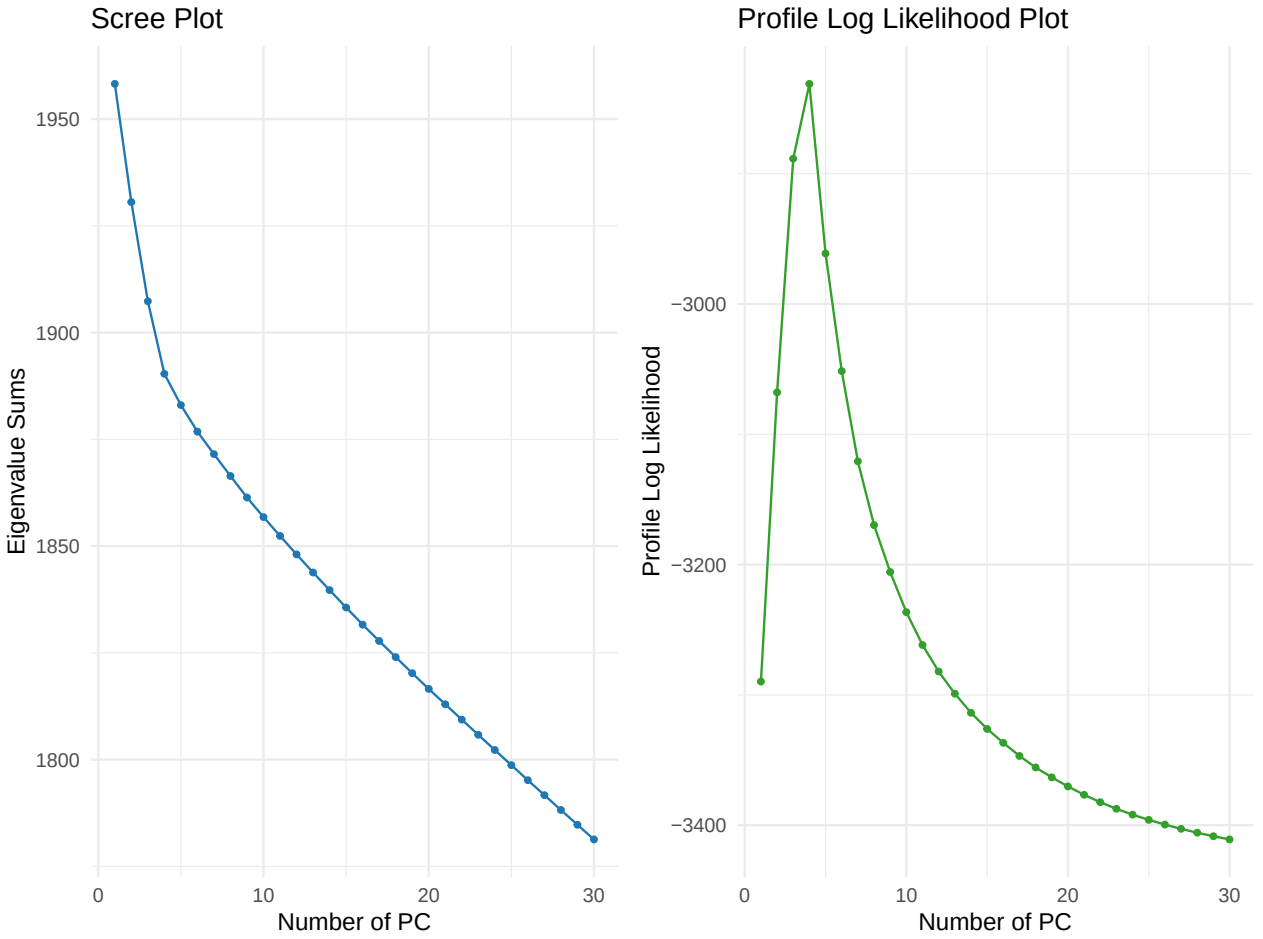
- To measure the quality of L, we assume a simple change-point model, where $\lambda_k \sim \mathcal{N}(\mu_1, \sigma^2)$ if $k \leq L$, and $\lambda_k \sim \mathcal{N}(\mu_2, \sigma^2)$ if $k > L$. within each regimes, assume the λ_k are iid.
- For each $L = 1 : L_{max}$, evaluate the profile log likelihood:

$$\begin{aligned} \mu_1(L) &= \frac{\sum_{k \leq L} \lambda_k}{L}, \ \mu_2(L) = \frac{\sum_{k > L} \lambda_k}{N - L} \\ \sigma^2(L) &= \frac{\sum_{k \leq L} (\lambda_k - \mu_1(L))^2 + \sum_{k > L} (\lambda_k - \mu_2(L))^2}{N} \\ \ell(L) &= \sum_{k=1}^{L} \log \mathcal{N}(\lambda_k | \mu_1(L), \sigma^2(L)) + \sum_{k=L+1}^{K} \log \mathcal{N}(\lambda_k | \mu_2(L), \sigma^2(L)) \end{aligned}$$

- Choose $L^* = arg\ max\ \ell(L)$.

The `select_proper_dimension()` function provides us the above to plots and suggests a dimension (L) to reduce to. The relevant data and plots are stored in the `reduced_dimension` slot, and the `Cellustering` instance is returned.

```
pbmc <- select_proper_dimension(pbmc)
#> [1] "The suggested dimension to reduce to: 4"
```



Advantages of `Cellustering` over others

For packages like `SC3` and `TSCAN`, they don't have a function to facilitate the selection of the dimension.

For packages like `Seurat`, they produces elbow plot (which functions just like scree plot), heatmap plot and others for facilitation. However, no automatic selection method is applied in these packages.

For packages like `SC3`, they employ the consensus clustering strategy (Kiselev et al, 2016) to analyze different choices of L. However, this procedure may take hours, and the clustering accuracy for large dataset is diminished since it cannot cluster all the cells at one time.

Connection to course materials

- Testing: `select_proper_dimension()` checks whether the count matrix has gone through the PCA step and whether user inputs a `Cellustering` instance.
- Vectorized computation: `select_proper_dimension()` employs vectorized computation in the production of the profile log-likelihood plot.

Cluster Analysis

Organizing cells into clusters is pivotal in single-cell analysis. Clusters are obtained by grouping cells based on the similarity of their gene expression profiles (the dimension-reduced version).

`Cellustering` uses the popular k-means clustering algorithm. This algorithm divides cells into k clusters by determining cluster centroids and assigning cells to the nearest cluster centroid. Centroid positions are iteratively optimized (MacQueen, 1967).

Principle of k-means clustering:

Given the dataset $\{\boldsymbol{x}_i\}_{i=1}^n$, k-means aims to find cluster centers $\boldsymbol{c} = \{\boldsymbol{c}_j\}_{j=1}^K$ and assgnments $\mathbf{r}$, by minimizing the sum of squared distances of the data points to their assigned cluster centers. In short, K-means will minimize the within-cluster variance, as follows:

$$\begin{aligned} &\min_{\mathbf{c},\mathbf{r}} J(\boldsymbol{c},\boldsymbol{r}) = \min_{\boldsymbol{c},\boldsymbol{r}} \sum_i^n \sum_k^K r_{ik} \left(\mathbf{x}_i - \mathbf{c}_k\right)^2 \\ &\text{Subject to} \quad \mathbf{r} \in \{0,1\}^{n*K}, \quad \sum_k^K r_{ik} = 1 \end{aligned}$$

where $r_{ik} = 1$ denotes $\boldsymbol{x}_i$ is assigned to cluster k.

The above optimization problem can be solved by coordinate descent algorithm, i.e., update $\boldsymbol{c}$ and $\boldsymbol{r}$ alternatively:

- *Given the cluster centers $\boldsymbol{c}$, update the assignments $\boldsymbol{r}$*
- *Given the assignments $\boldsymbol{r}$, update the cluster centers $\boldsymbol{c}$*

Therefore, the general procedure of the optimization of K-means clustering is:

- *Initialization: set K cluster centers $\boldsymbol{c}$ to random values*
- *Repeat the following steps until convergence (the assignments don't change):*
  - *Assignment: Given the cluster ceneters $\boldsymbol{c}$, update the assignments $\boldsymbol{r}$ by solving the following sub-problem:*

$$\min_{\boldsymbol{r}} \sum_i^n \sum_k^K r_{ik} \left(\mathbf{x}_i - \mathbf{c}_k\right)^2, \text{ subject to } \mathbf{r} \in \{0,1\}^{n \times K}, \sum_k^K r_{ik} = 1$$

Note that the assignment for each data $\boldsymbol{x}_i$ can be solved independently. i.e.

$$\min_{\boldsymbol{r}} \sum_k^K r_{ik} \left(\mathbf{x}_i - \mathbf{c}_k\right)^2, \text{ subject to } \mathbf{r}_i \in \{0,1\}^{1*K}, \sum_k^K r_{ik} = 1$$

It is easy to know that assign $\boldsymbol{x}_i$ to the closest cluster is the optimal solution.

$$k^* = \arg\min \left\{ (\mathbf{x}_i - \mathbf{c}_k)^2 \right\}_{k=1}^{K}, \text{ and } \sum_{k}^{K} r_{ik^*} = 1$$

- *Refitting: Given the assignments* $\boldsymbol{r}$*, update the cluster centers* $\boldsymbol{c}$*:*

$$\min_{\boldsymbol{c}} \sum_{i}^{n} \sum_{k}^{K} r_{ik} (\mathbf{x}_i - \mathbf{c}_k)^2$$

Note that $\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots\ldots, \boldsymbol{c}_K$ *can be optimized independently, as follows:*

$$\min_{c_k} \sum_{i}^{n} r_{ik} (\mathbf{x}_i - \mathbf{c}_k)^2$$

By setting the derivative w.r.t. $\boldsymbol{c}_k$ *as 0 , it is easy to obtain the optimal solution:*

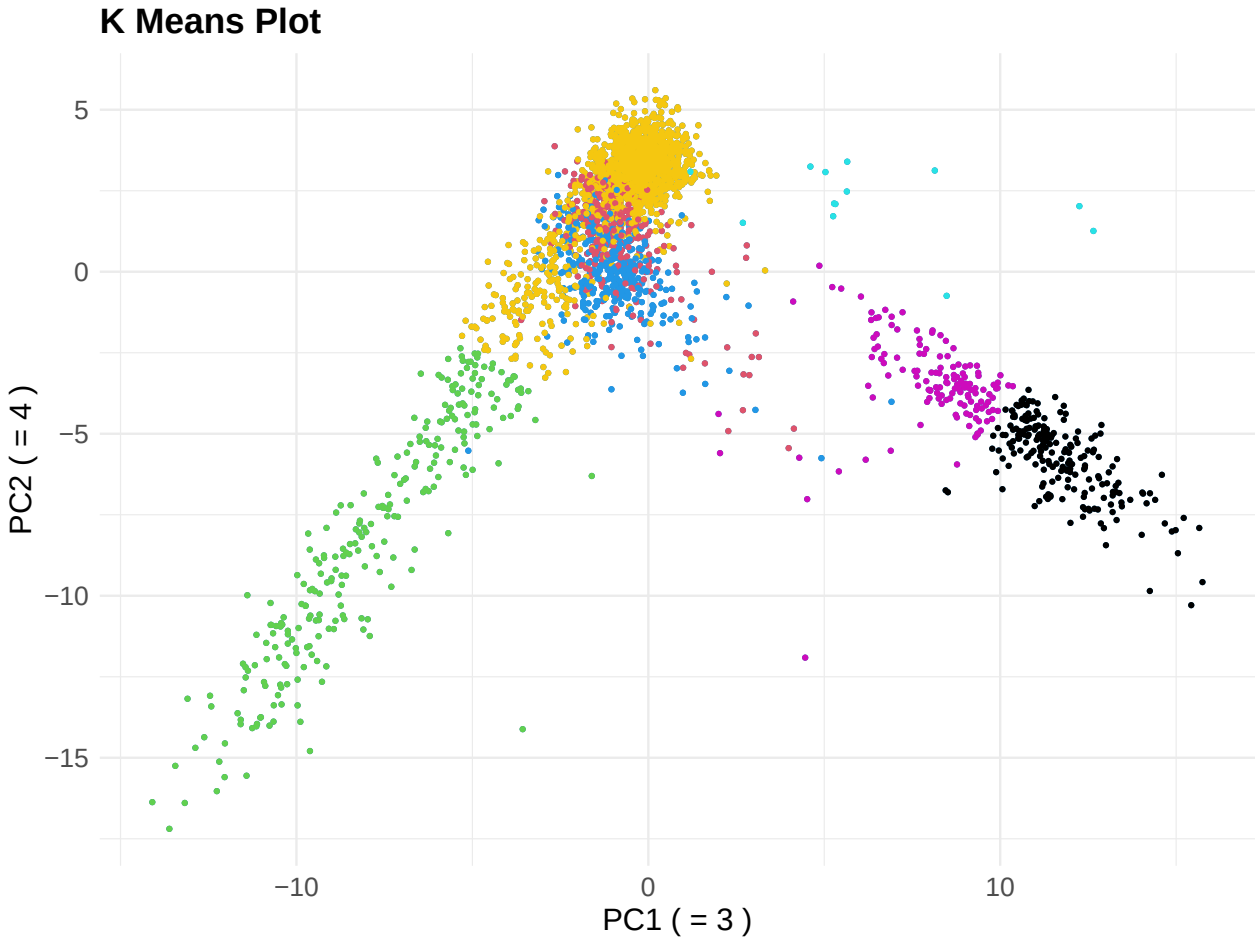
$$\sum_{i}^{n} 2 r_{ik} (\mathbf{x}_i - \mathbf{c}_k) = 0 \Rightarrow \boldsymbol{c}_k = \frac{\sum_i^n r_{ik} \mathbf{x}_i}{\sum_i^n r_{ik}}$$

Thus, $\boldsymbol{c}_k$ *is the center of the k-th cluster.*

`kmeans()`: Apply k-means clustering on data

The `kmeans()` function applies k-means clustering algorithm on the dimension-reduced HVG count matrix. The number of clusters k needs to be specified by us. The `kmeans()` function generates a two-dimensional plot based on the PCA results and clustering results. The relevant clustering information and plots are stored in the `clustering` slot, and the `Cellustering` instance is returned.

```
pbmc <- kmeans(pbmc, k = 7, dimensions = 4)
```



Advantages of `Cellustering` over others

For packages like `Seurat`, they use community detection methods like graph-partitioning algorithms. However, these algorithms cannot specify the exact number of clusters. User can only make rough adjustment on cluster numbers by tuning the `resolution` parameter.

Connection to course materials

- Testing: `kmeans()` checks whether the count matrix has gone through the PCA step and whether user input has correct data type and appropriate value.
- ggplot2: `kmeans()` uses `ggplot2` package to plot.

`compute_silhouette()`: Evaluate the clustering results

`Cellustering` uses Silhouette coefficient to evaluate the clustering results.

Silhouette coefficient for a single sample is formulated as:

$$s = \frac{b-a}{\max(a,b)} \Rightarrow s = \begin{cases} 1 - \frac{a}{b} & \text{if } a < b \\ 0 & \text{if } a = b \\ \frac{b}{a} - 1 & \text{if } a > b \end{cases}$$

It is easy to know that $s \in (-1, 1)$, and larger s value indicates better clustering performance. And the Silhouette coefficient s for a set of samples is defined as the mean of the Silhouette Coefficient for each sample.

The derived Silhouette coefficient is stored in the `clustering` slot and the `Cellustering` instance is returned.

```
pbmc <- compute_silhouette(pbmc)
#> [1] "The mean value of silhouette coefficients: 0.568468515377001"
```

Advantages of `Cellustering` over others

For packages like `Seurat`, they don't have evaluation matrices for clustering results.

Connection to course materials

- Testing: `compute_silhouette()` checks whether the count matrix has gone through the k-means step and whether user inputs a `Cellustering` instance.
- Data frame manipulation: `compute_silhouette()` manipulates over the data frame.