Processing scRNA-seq Data with Scanpy and Automatic Cluster Annotation with CellTypist

Introduction

In this post, we will explore an example of single-cell RNA-seq data analysis using Scanpy and the automatic cell annotation with the Python module, CellTypist. In a previous entry (https://biologydatascience.wordpress.com/2022/08/05/workflow-estandard-per-analitzar-dades-dexperiments-single-cell-rna-seq-amb-seurat/), we looked at how to analyze this type of data with Seurat. While Seurat is a package for analyzing scRNA-seq data in R, Scanpy is its counterpart in a Python environment. The data we will work with in this example can be downloaded from the 10x Genomics website via this link. This scRNA-seq dataset comes from PBMCs (peripheral blood mononuclear cells), which are immune system cells found in the blood characterized by having a single round nucleus. The main cell types we expect to find are:

- Lymphocytes (T cells, B cells, and natural killers).
- Monocytes.
- Dendritic cells. PBMCs are isolated from the rest of the blood components through density gradient centrifugation and play an essential role in biomedical research as they are used as a model in studying the immune response in diseases.

Once we obtain the single-cell sequencing data from our sample, the usual practice is to preprocess it with a pipeline that allows us to obtain the count matrix (genes x cells or cells x genes). For data generated with 10x Genomics technology, this pipeline is usually the one proposed by the same company (Cell Ranger). scRNA-seq data analyses usually have well-defined phases: preprocessing (generation of the count matrix), data processing and annotation, downstream analysis (differential expression analysis, cellular communication, trajectories, etc.). In the example at hand, we will see how to perform data processing and annotation with Scanpy and CellTypist.

Loading Necessary Modules and Creating the AnnData Object

Once we have started our Python work environment, we need to ensure that the Scanpy, Scrublet (for doublet detection), CellTypist, Numpy, Matplotlib, and adjustText (to improve one of the graphics we will obtain) modules are installed. The first step in our analysis is to load the modules:

```
>>> import scanpy as sc
>>> import scrublet as scr
>>> import celltypist
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from adjustText import adjust_text
```

Next, we load the count matrix and create an AnnData object using the `read_10x_h5()` function, where we only need to specify the path to the h5 file obtained from the preprocessing pipeline (or, in this case, downloaded from the web):

```
>>> file_path = "data/pbmc_10k.h5"
>>> adata = sc.read_10x_h5(file_path)
```

Once the object is created, we evaluate it to confirm it contains the information we expect:

```
>>> adata
AnnData object with n_obs × n_vars = 11769 × 33538
    var: 'gene_ids', 'feature_types', 'genome'
```

AnnData objects are designed to store and analyze scRNA-seq data. The core of an AnnData object is a data matrix (with cells in rows and genes in columns) and its annotations. The different parts of the object are:

- `.X`: The main matrix containing expression values.
- `.obs`: Cell-level metadata (cluster annotations, experimental conditions, quality metrics, etc.).
- `.var`: Feature-level metadata (such as gene names, identifiers, and any other characteristics we want to store about each gene).
- `.obsm` and `.varm`: Multidimensional annotations (dimensional reductions like PCA or UMAP).
- `.uns`: Unstructured metadata for storing additional information (visualization settings, cluster colors, etc.). The last step before starting data processing is to use the `var_names_make_unique()` function to ensure all genes have unique names. This is important because, in many cases, gene names are repeated as there is no consensus on which coding sequence represents the common name.

```
>>> adata.var_names_make_unique()
```

We can now proceed to data processing and annotation. Typically, this step is divided into different sections:

- **Quality Control**: The goal is to eliminate poor-quality cells, whether because they are not cells (they are ambient RNA from other cells) or because they are somehow damaged cells.
- **Normalization and Multivariate Analysis**: In this part of the analysis, the goal is to normalize the data so that the expression of the cells is comparable among themselves, select the most variable genes, standardize the expression of these genes, reduce dimensionality, and cluster the cells.
- **Cell Annotation**: In the last part, the aim is to find the identity of the cells based on the transcriptional profile of each cell cluster.

Quality Control

To identify low-quality cells, we rely on three variables:

- **Total number of transcripts and genes detected in each cell**: The assumption is that cells with very few transcripts and genes likely do not represent actual cells but rather ambient RNA. Conversely, cells with extremely high values may be doublets (two cells instead of one).
- **Percentage of transcripts coming from the mitochondrial genome (hereafter, mitochondrial percentage)**: Here, the assumption is that cells with a very high mitochondrial percentage correspond to cells where the cell membrane has broken and the cytoplasmic mRNA has been lost, but the mitochondrial mRNA is still present.

Towards the end of the entry, as nuclear markers can also help us identify poor-quality cells, especially in more complex tissue samples. To provide additional protection against doublets, we will use specific software for their detection.

Automatic Doublet Identification with Scrublet Before starting quality control, we will use the Scrublet library to detect possible doublets. Scrublet works as follows:

- **Doublet Simulation**: Scrublet creates synthetic doublets by averaging the expression profiles of random pairs of cells, mimicking real doublets.
- **Dimensionality Reduction**: Both real cells and synthetic doublets are integrated into a reduced-dimensionality space (using PCA).
- **Doublet Score**: Using a machine learning algorithm, Scrublet calculates a "doublet score" for each cell based on its similarity to the synthetic doublets in the principal component space.

Once doublets are detected, Scrublet finds a score threshold to classify cells as doublets or "singlets." This is based on the reported doublet ratios for 10x Genomics technology, which is 1% per 1000 cells. Since we have recovered around 10,000 cells, we expect about 10% of doublets.

First, we extract the count matrix:

```
>>> scrub = scr.Scrublet(adata.X.toarray())
```

Then, we use the Scrublet algorithm to detect doublets:

```
>>> doublet_scores, predicted_doublets = scrub.scrub_doublets()
```

The above line of code generates two objects: doublet_scores, with the scores for each cell, and predicted_doublets, with the classification of "singlet" or doublet.

Study of the Distribution of Usual Quality Metrics To start, we classify each gene based on whether it is found in the mitochondrial genome or not and then calculate the percentage of transcripts coming from the mitochondrial genome. We do this with the `str.startswith()` method:

```
>>> adata.var["mt"] = adata.var_names.str.startswith("MT-")
```

Since the genes found in the mitochondrial genome begin with the prefix "MT-", we add a new column called "mt" to the gene metadata, which will contain True or False depending on whether the gene name starts with "MT-" or not.

```
sc.pp.calculate_qc_metrics(
    adata, qc_vars=["mt"], inplace=True, log1p=True
)
```

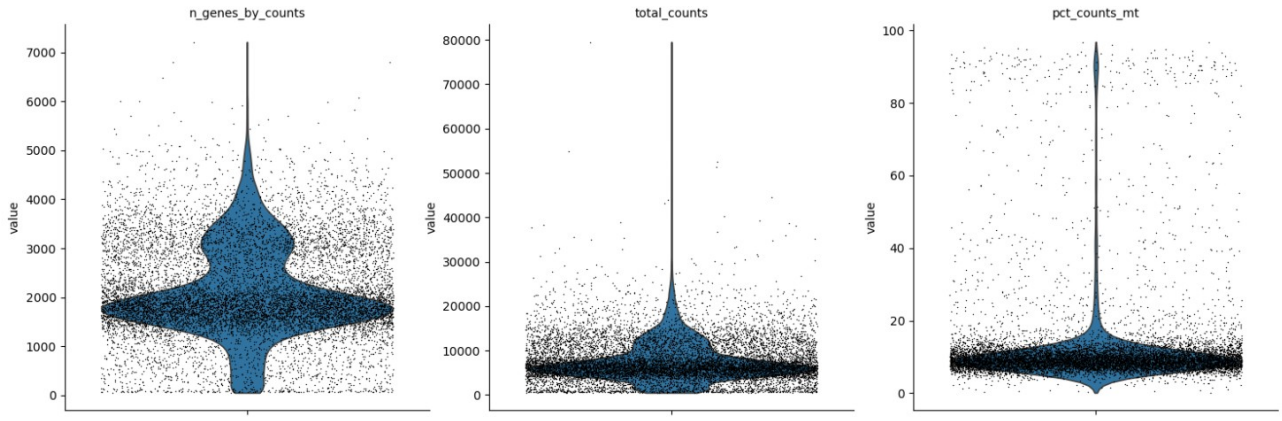
Now, we calculate the main quality metrics with the `calculate_qc_metrics()` function. This function calculates many different quality metrics. The most notable for each cell are:

- **total_counts**: total number of transcripts detected.
- **n_genes_by_counts**: total number of genes with at least one detected transcript.
- **pct_counts_mt**: percentage of transcripts associated with mitochondrial genes (because we put it in the qc_vars argument)

Since we used the `log1p=True` argument, the function will also return the same variables log-transformed (natural logarithm). The `inplace=True` argument adds the calculated variables directly to the metadata.

Once we have all the quality variables calculated, the next step is to study their distribution. We will do this with violin plots, with the values of each cell superimposed:

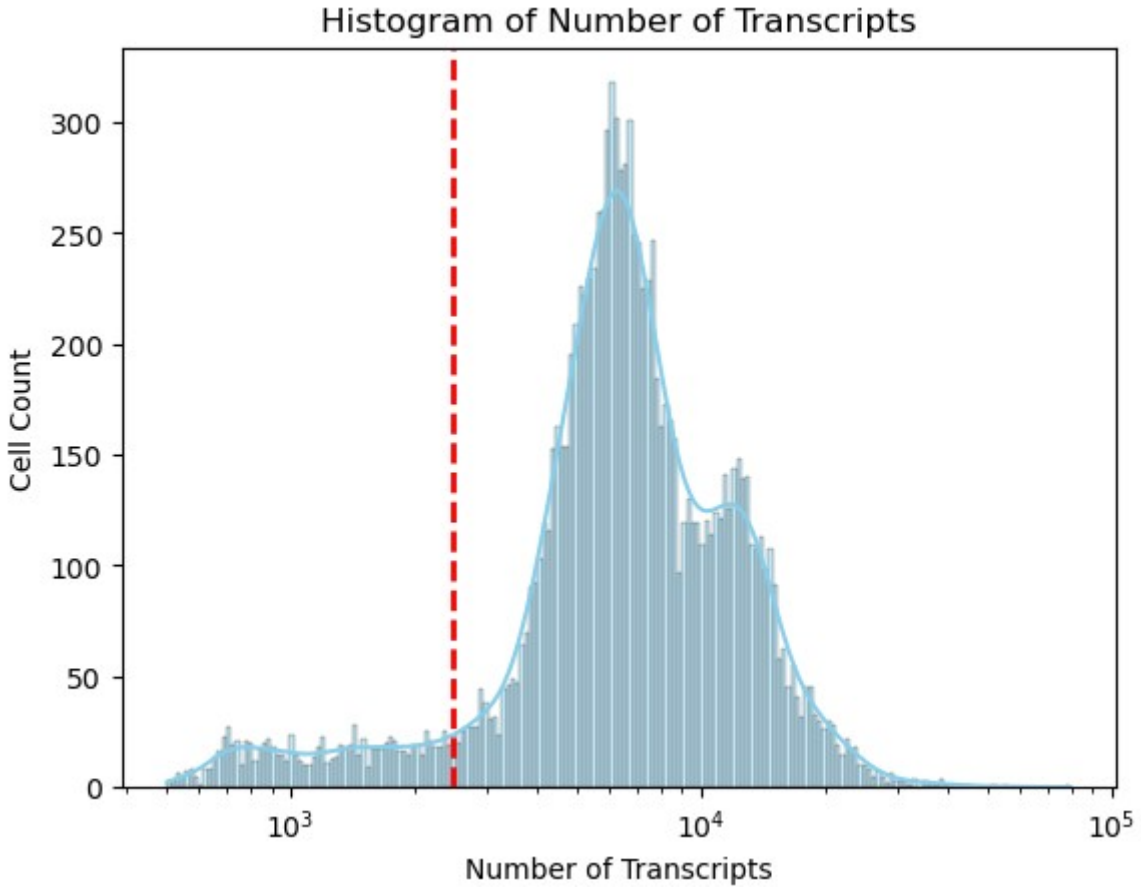
```
sc.pl.violin(
    adata,
    ["n_genes_by_counts", "total_counts", "pct_counts_mt"],
    jitter=0.4,
    multi_panel=True
)
```



Although in this distribution we can differentiate different populations of cells and, in the case of mitochondrial percentage, identify poor-quality cells, we can better study the distributions with a histogram. For the total number of transcripts:

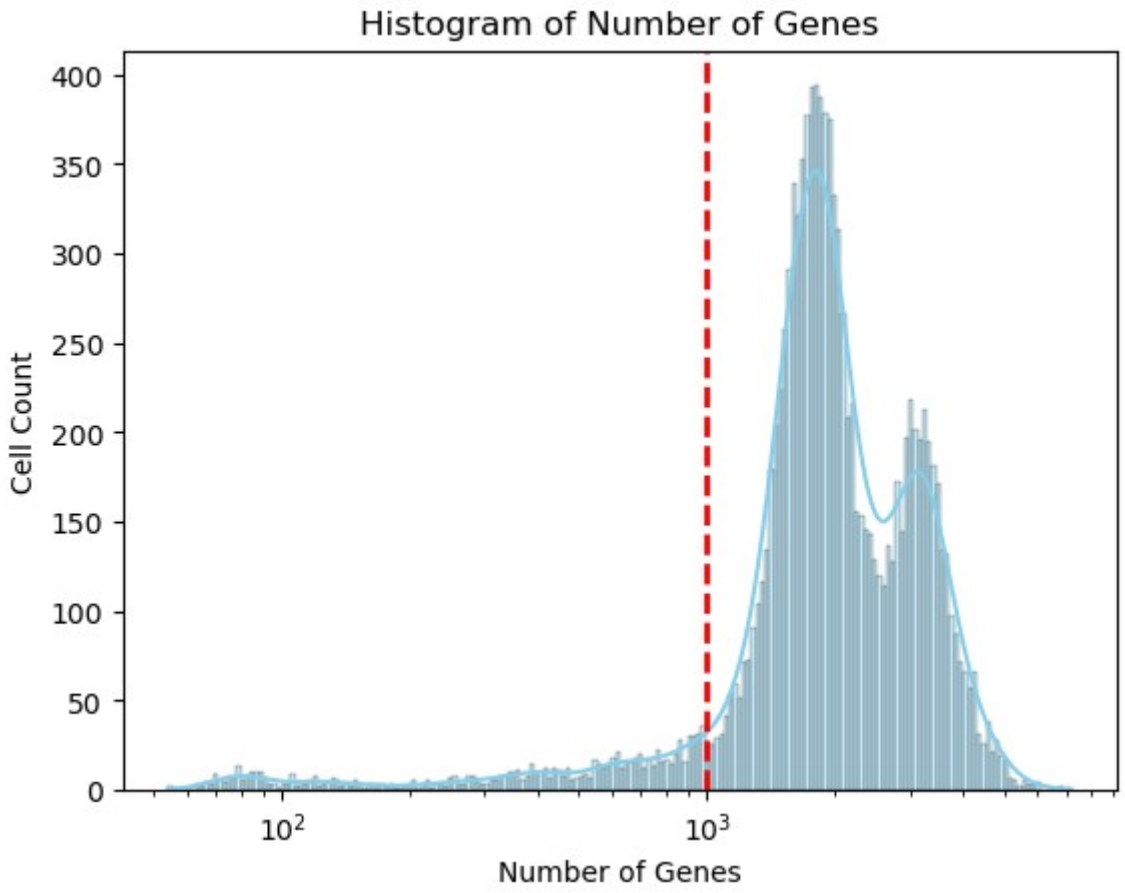
```
min_total_counts = 2500
total_counts_data = adata.obs['total_counts']
sns.histplot(total_counts_data, bins=200, kde=True, color='skyblue', edgecolor='black',
log_scale=True)
```

```
plt.axvline(x=min_total_counts, color='red', linestyle='--', linewidth=2, label='Threshold = 2500')
plt.xlabel('Number of Transcripts')
plt.ylabel('Cell Count')
plt.title('Histogram of Number of Transcripts')
plt.show()
```



We do the same for the total number of detected genes:

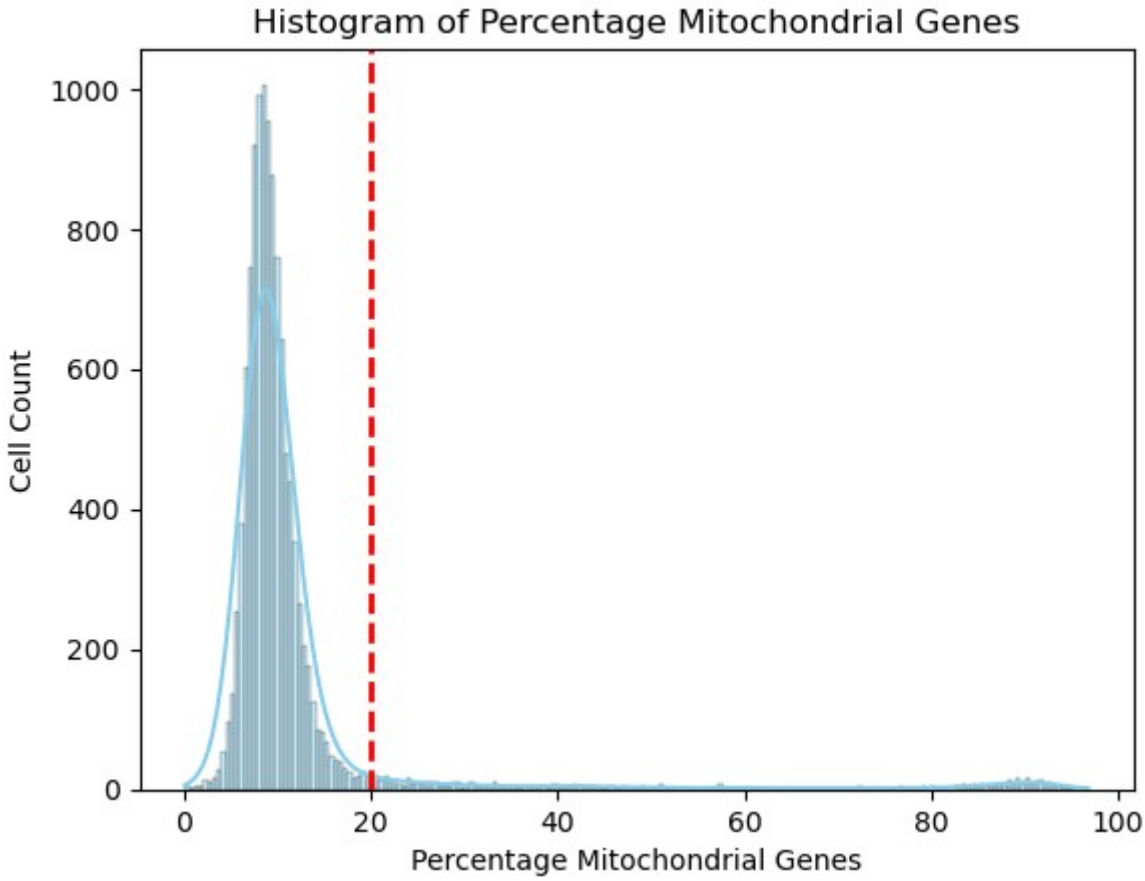
```
min_genes = 1000
total_genes_data = adata.obs['n_genes_by_counts']
sns.histplot(total_genes_data, bins=200, kde=True, color='skyblue', edgecolor='black',
log_scale=True)
plt.axvline(x=min_genes, color='red', linestyle='--', linewidth=2, label='Threshold = 1000')
plt.xlabel('Number of Genes')
plt.ylabel('Cell Count')
plt.title('Histogram of Number of Genes')
plt.show()
```



Here we set the threshold values to a minimum of 2500 detected transcripts and 1000 genes per cell. At the end of this entry, we will see how we arrived at these values. Here we only say that the lower values correspond to poor-quality cells and less complex and contaminating cells, the platelets (which we do not expect to find in a PBMC preparation).

Regarding the mitochondrial percentage:

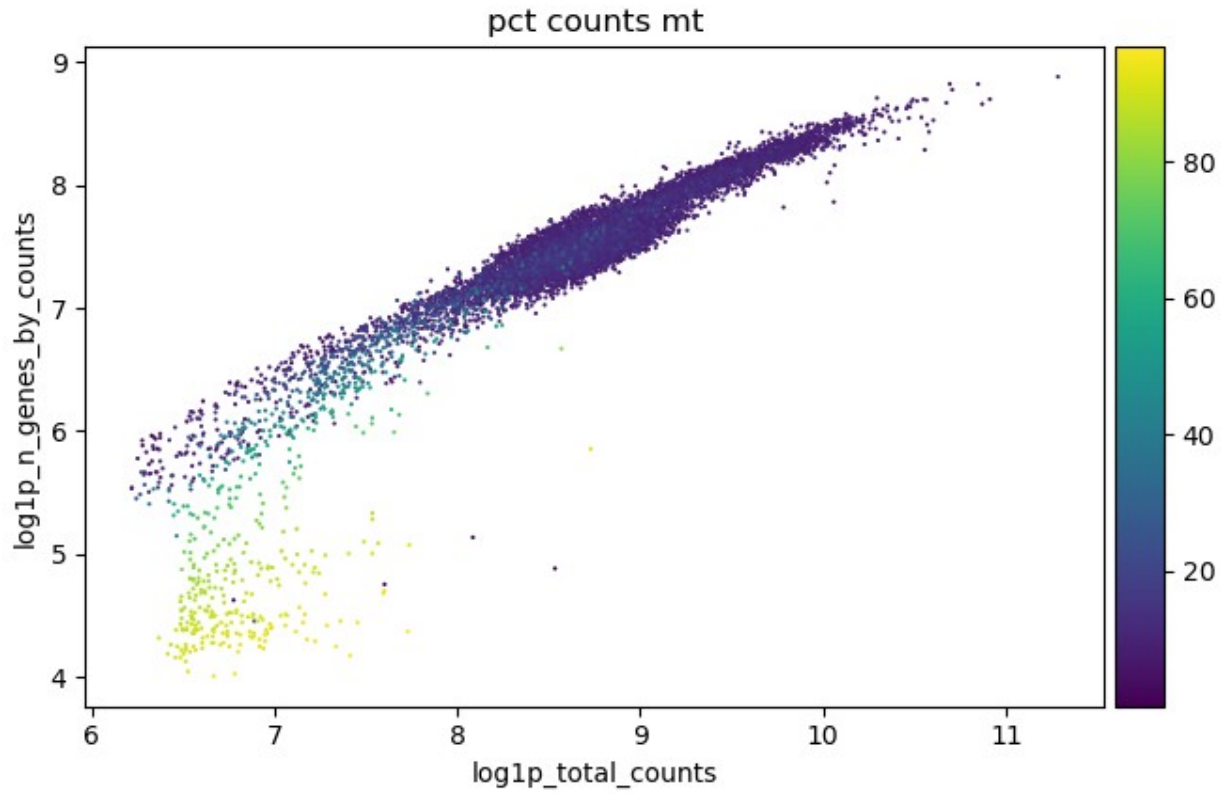
```
max_mito = 20
pct_counts_mt_data = adata.obs['pct_counts_mt']
sns.histplot(pct_counts_mt_data, bins=200, kde=True, color='skyblue', edgecolor='black',
log_scale=False)
plt.axvline(x=max_mito, color='red', linestyle='--', linewidth=2, label='Threshold = 15')
plt.xlabel('Percentage Mitochondrial Genes')
plt.ylabel('Cell Count')
plt.title('Histogram of Percentage Mitochondrial Genes')
plt.show()
```



In this case, it is clear that the vast majority of cells have values below 20%, and a group of cells have values above 80%.

Finally, it is also useful to look at the three metrics together. In their logarithmic version, we can better appreciate the differences between good and poor-quality cells, at least those related to those with a high content of mitochondrial genes:

```
>>> sc.pl.scatter(adata, x='log1p_total_counts', y='log1p_n_genes_by_counts', color='pct_counts_mt')
```



Although we will see in more detail how we arrived at the threshold values at the end of the post, in these scatter plots of the three measures together, we can see that the bulk of the cells have more than 2500 transcripts, more than 1000 detected genes, and a mitochondrial percentage below 20%.

Removal of Poor-Quality Cells and Unexpressed Genes

We will create a new metadata variable to classify poor-quality cells and doublets:

```
adata.obs['bad_quality'] = (
    (adata.obs['total_counts'] < 2500) |
    (adata.obs['total_counts'] > 30000) |
    (adata.obs['n_genes_by_counts'] < 1000) |
    (adata.obs['n_genes_by_counts'] > 5500) |
    (adata.obs['pct_counts_mt'] > 20) |
    (adata.obs['predicted_doublets'])
)
```

We can count the good and bad quality cells using the `value_counts()` function:

```
>>> adata.obs['bad_quality'].value_counts()
bad_quality
False    10000
True      1769
Name: count, dtype: int64
```

In total, we will discard 1769 poor-quality cells and doublets. Now, we can remove these cells that we have marked:

```
>>> adata = adata[(~adata.obs['bad_quality'])].copy()
>>> adata
AnnData object with n_obs × n_vars = 10321 × 33538
    obs: 'doublet_scores', 'predicted_doublets', 'n_genes_by_counts', 'log1p_n_genes_by_counts',
'total_counts', 'log1p_total_counts', 'pct_counts_in_top_50_genes', 'pct_counts_in_top_100_genes',
'pct_counts_in_top_200_genes', 'pct_counts_in_top_500_genes', 'total_counts_mt',
'log1p_total_counts_mt', 'pct_counts_mt', 'bad_quality'
    var: 'gene_ids', 'feature_types', 'genome', 'mt', 'ribo', 'hb', 'n_cells_by_counts',
'mean_counts', 'log1p_mean_counts', 'pct_dropout_by_counts', 'total_counts', 'log1p_total_counts'
```

The ~ symbol allows us to invert the selection (select False instead of True).

Normalization and Multivariate Analysis

Normalization The last step before proceeding to annotate the cells is the normalization of the data and multivariate analysis. Data normalization corrects differences in the total number of transcripts detected in each cell:

```
>>> sc.pp.normalize_total(adata, target_sum=1e4)
>>> sc.pp.log1p(adata)
```

Automatically, the first function scales the number of transcripts so that all cells have the same total number of transcripts (in this case, 10000):

$$x_{ij}^{\text{normalized}} = x_{ij} \times \frac{\text{target\_sum}}{\sum_j x_{ij}}$$

This transformation is important for comparing expression values between cells so that the differences we see are not due to differences in library size.

On the other hand, the second function takes the natural logarithm of the data (adding one to avoid having values of 0):

$$x^{\text{transformed}} = \ln(x+1)$$

In this case, this transformation is useful for stabilizing the variance of the data and obtaining a distribution of gene expression more suitable for the multivariate statistical techniques that we will use next for differential expression analysis between groups of cells.

Multivariate Analysis The following steps fall within multivariate analysis. The goal of these is to form groups of cells (based on transcriptomic similarity) under the assumption that these will represent different cellular types or states.

The first phase is to select the genes (variable selection) that we will use to form the cell clusters. This is done by finding the 2000 genes that show the most variability among cells (this is an arbitrary number that usually works well), taking into account technical background noise. The assumption here is that these are the genes that contain the most relevant biological information and reduce the noise contributed by genes

with little variance and that are not informative. Additionally, working with a smaller set of genes improves computational efficiency. The line of code that allows us to identify the most variable genes is the following:

```
>>> sc.pp.highly_variable_genes(adata, n_top_genes=2000)
```

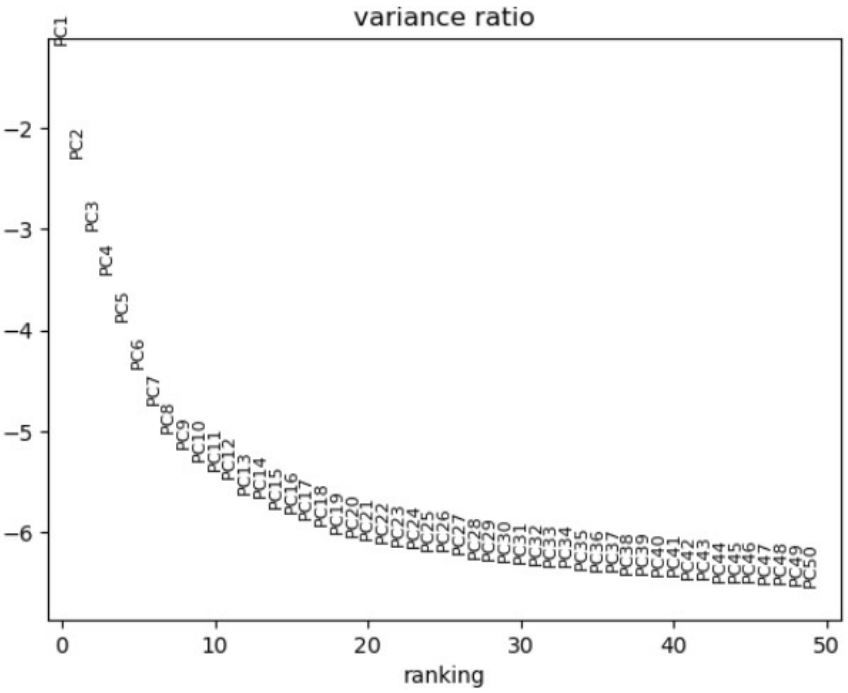
Briefly, this function divides the genes into intervals according to their average expression, and within each interval, it calculates a normalized dispersion (variance divided by the mean). The 2000 genes with the highest normalized dispersion are selected as the most variable.

Now, the next step is to perform principal component analysis. The goal is to reduce the number of dimensions (the 2000 most variable genes) to a much smaller number (by default, the first 50 principal components) that captures the most important expression patterns among cells:

```
>>> sc.tl.pca(adata)
```

Once the first 50 principal components are found, it is always a good idea to inspect the explained variance for all of them:

```
>>> sc.pl.pca_variance_ratio(adata, n_pcs=50, log=True)
```



On the Y-axis, we have the logarithm of the variance of each principal component relative to the total variance:

$$\text{Log(Variance Ratio)} = \ln\left(\frac{\text{Variance}_{PC_i}}{\text{Total Variance}}\right)$$

We can see that the first 50 principal components explain the majority of the variance. While it is important to select a sufficient number of principal components to continue multivariate analysis, there is usually no harm in selecting a high number. In this example, we will continue the analysis with the first 50 principal components. The next step is to perform clustering of the cells using an unsupervised clustering algorithm.

First, we will construct a graph of nearest neighbors (kNN; by default, it finds the 15 nearest neighbors for each cell) among the cells. This will relate the most similar cells to each other based on the information contained in the first 50 principal components:

```
>>> sc.pp.neighbors(adata)
```

Once we have the neighbor graph, we will find the clusters with the following function:

```
>>> sc.tl.leiden(adata, flavor="igraph", n_iterations=2, resolution=1)
```

This function uses the Leiden algorithm to define communities from a neighbor graph. The algorithm groups cells into clusters in such a way that the connections within a cluster are greater than between clusters. The parameter flavor="igraph" indicates that we want to use the Leiden implementation of the igraph package, while n_iterations=2 is the number of iterations the algorithm will make to refine the clustering. Two iterations is the default number for the igraph package, it is faster, and it is usually sufficient to obtain good results. On the other hand, the resolution=1 parameter defines how we want the clusters to be: values above 1 correspond to a higher number of clusters (a finer and more detailed partition) while values below 1 will result in a smaller and larger number of clusters (a more general partition). Often, the definition of this value is found by trial and error, but it helps to know the approximate number of cellular types or states that we expect to find. The cluster to which each cell belongs is saved in the metadata in the 'leiden' column:

```
>>> adata.obs['leiden']
AAACCCAAGCGCCCAT-1     0
AAACCCAAGGTTCCGC-1     2
AAACCCACAGAGTTGG-1     3
AAACCCACAGGTATGG-1     4
AAACCCACATAGTCAC-1     5
                      ..
TTTGTTGGTGCGTCGT-1     1
TTTGTTGGTGTCATGT-1     6
TTTGTTGGTTTGAACC-1    11
TTTGTTGTCCAAGCCG-1     7
TTTGTTGTCTTACTGT-1     1
Name: leiden, Length: 10321, dtype: category
Categories (21, object): ['0', '1', '2', '3', ..., '17', '18', '19', '20']
```

With the following line of code, we can count the number of cells contained in each of the clusters:

```
>>> adata.obs['leiden'].value_counts()
leiden
0     1418
6     1383
1     1275
3     1124
11     779
9      600
16     592
4      578
5      575
7      455
8      334
12     326
15     280
```

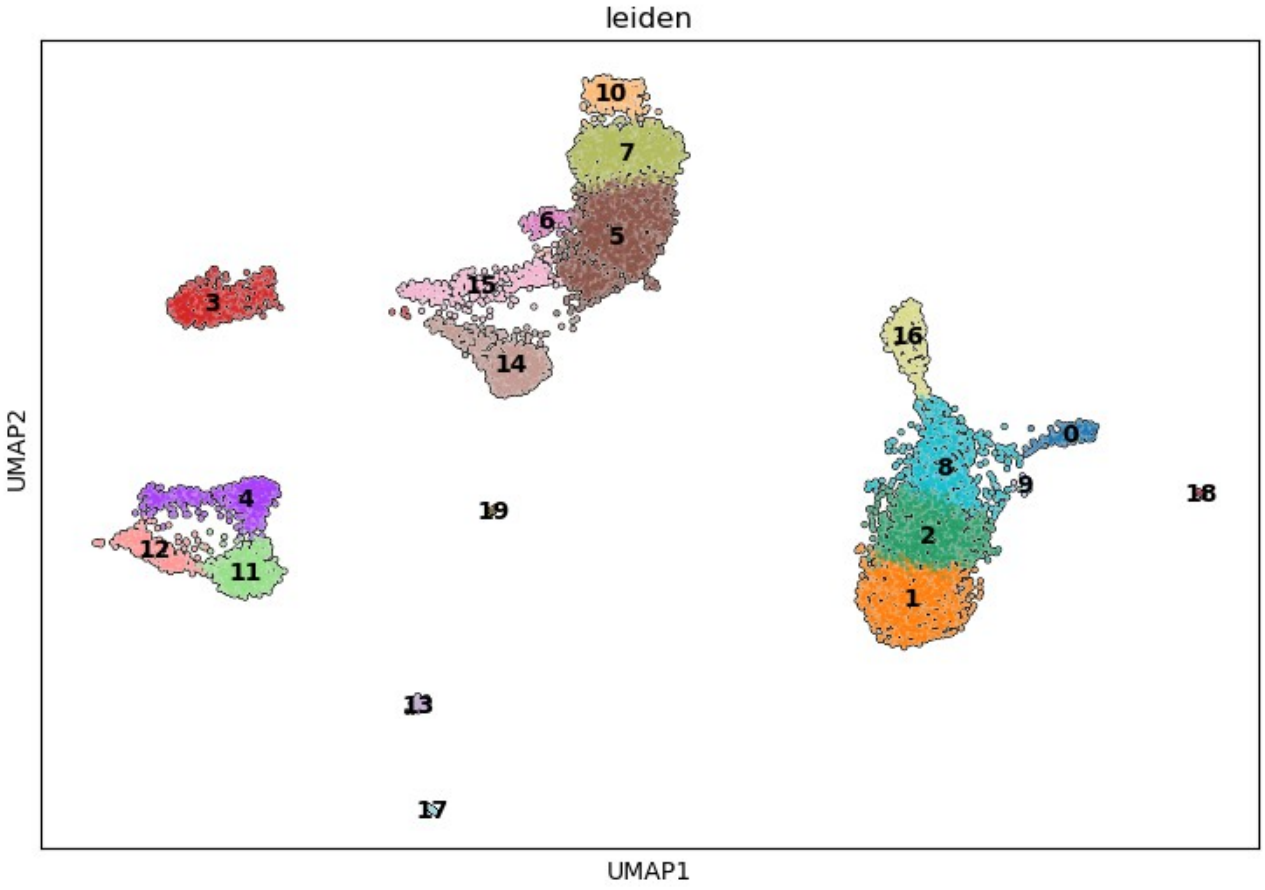
```
14     253
2      124
13      79
10      72
17      24
20      22
18      14
19      14
Name: count, dtype: int64
```

The last step before proceeding to annotate the different cellular communities we have found is to visualize the transcriptome of our cells in two dimensions. To do this, we will further reduce the dimensionality of the data (we now have 50 dimensions, the first 50 principal components), but this time non-linearly using a UMAP:

```
>>> sc.tl.umap(adata)
```

UMAP is a dimensionality reduction technique that allows projecting the information of the first 50 principal components into a two-dimensional space for visualization and exploration. In this way, the previous function calculates two UMAP coordinates for each cell using the previously calculated kNN graph, in such a way that it will preserve the relationships between them: similar cells will be close in the new coordinates, while very different cells will be far apart. Next, we can visualize our data and the cell groupings:

```
sc.pl.umap(
    adata,
    color="leiden",
    legend_loc='on data',
    add_outline=True
)
```



In the previous code, the parameter `color="leiden"`, indicates that we want to paint the cells based on the cluster they belong to; `legend_loc='on data'` serves to place the legend at the center of each cluster; and `add_outline=True` draws the outline around the point clouds. In the UMAP graphs, each point represents a cell.

Cell Annotation with CellTypist

We are now ready to use the CellTypist module to automatically annotate the cells. CellTypist is a tool that contains machine learning models to predict cell types in scRNA-seq data. In addition to making predictions, the tool also returns a confidence score for each cell's prediction.

First, we need to download the models:

```
>>> celltypist_model = celltypist.models.download_models()
```

Next, we can view the description of each model with the following line of code:

```
>>> celltypist.models.models_description()
   Detailed model information can be found at `https://www.celltypist.org/models`
model   description
0       Immune_All_Low.pkl     immune sub-populations combined from 20 tissue...
1       Immune_All_High.pkl    immune populations combined from 20 tissues of...
2       Adult_COVID19_PBMC.pklperipheral blood mononuclear cell types from C...
```

```
3       Adult_CynomolgusMacaque_Hippocampus.pkl     cell types from the hippocampus of adult
cynom...
4       Adult_Human_MTG.pkl   cell types and subtypes (10x-based) from the a...
5       Adult_Human_PancreaticIslet.pkl      cell types from pancreatic islets of healthy a...
6       Adult_Human_PrefrontalCortex.pkl     cell types and subtypes from the adult human d...
7       Adult_Human_Skin.pkl  cell types from human healthy adult skin
8       Adult_Human_Vascular.pkl      vascular populations combined from multiple ad...
9       Adult_Mouse_Gut.pkl   cell types in the adult mouse gut combined fro...
10      Adult_Mouse_OlfactoryBulb.pkl cell types from the olfactory bulb of adult mice
11      Adult_Pig_Hippocampus.pkl     cell types from the adult pig hippocampus
12      Adult_RhesusMacaque_Hippocampus.pkl  cell types from the hippocampus of adult rhesu...
13      Autopsy_COVID19_Lung.pkl      cell types from the lungs of 16 SARS-CoV-2 inf...
14      COVID19_HumanChallenge_Blood.pkl     detailed blood cell states from 16 individuals...
15      COVID19_Immune_Landscape.pkl  immune subtypes from lung and blood of COVID-1...
16      Cells_Adult_Breast.pklcell types from the adult human breast
17      Cells_Fetal_Lung.pkl  cell types from human embryonic and fetal lungs
18      Cells_Human_Tonsil.pkltonsillar cell types from humans (3-65 years)
19      Cells_Intestinal_Tract.pkl    intestinal cells from fetal, pediatric (health...
20      Cells_Lung_Airway.pkl cell populations from scRNA-seq of five locati...
21      Developing_Human_Brain.pkl    cell types from the first-trimester developing...
22      Developing_Human_Gonads.pkl   cell types of human gonadal and adjacent extra...
23      Developing_Human_Hippocampus.pkl     cell types from the developing human hippocampus
24      Developing_Human_Organs.pkl   cell types of five endoderm-derived organs in ...
25      Developing_Human_Thymus.pkl   cell populations in embryonic, fetal, pediatri...
26      Developing_Mouse_Brain.pkl    cell types from the embryonic mouse brain betw...
27      Developing_Mouse_Hippocampus.pkl     cell types from the mouse hippocampus at postn...
28      Fetal_Human_AdrenalGlands.pkl cell types of human fetal adrenal glands from ...
29      Fetal_Human_Pancreas.pkl      pancreatic cell types from human embryos at 9-...
30      Fetal_Human_Pituitary.pkl     cell types of human fetal pituitaries from 7 t...
31      Fetal_Human_Retina.pklcell types from human fetal neural retina and ...
32      Fetal_Human_Skin.pkl  cell types from developing human fetal skin
33      Healthy_Adult_Heart.pkl       cell types from eight anatomical regions of th...
34      Healthy_COVID19_PBMC.pkl      peripheral blood mononuclear cell types from h...
35      Healthy_Human_Liver.pkl       cell types from scRNA-seq and snRNA-seq of the...
36      Healthy_Mouse_Liver.pkl       cell types from scRNA-seq and snRNA-seq of the...
37      Human_AdultAged_Hippocampus.pkl      cell types from the hippocampus of adult and a...
38      Human_Colorectal_Cancer.pkl   cell types of colon tissues from patients with...
39      Human_Developmental_Retina.pkl       cell types from human fetal retina
40      Human_Embryonic_YolkSac.pkl   cell types of the human yolk sac from 4-8 post...
41      Human_Endometrium_Atlas.pkl   endometrial cell types integrated from seven d...
42      Human_IPF_Lung.pkl    cell types from idiopathic pulmonary fibrosis,...
43      Human_Longitudinal_Hippocampus.pkl   cell types from the adult human anterior and p...
44      Human_Lung_Atlas.pkl  integrated Human Lung Cell Atlas (HLCA) combin...
45      Human_PF_Lung.pkl     cell types from different forms of pulmonary f...
46      Human_Placenta_Decidua.pkl    cell types from first-trimester human placenta...
47      Lethal_COVID19_Lung.pkl       cell types from the lungs of individuals who d...
48      Mouse_Dentate_Gyrus.pkl       cell types from the dentate gyrus in perinatal...
49      Mouse_Isocortex_Hippocampus.pkl      cell types from the adult mouse isocortex (neo...
50      Mouse_Postnatal_DentateGyrus.pkl     cell types from the mouse dentate gyrus in pos...
51      Mouse_Whole_Brain.pkl cell types from the whole adult mouse brain
52      Nuclei_Lung_Airway.pklcell populations from snRNA-seq of five locati...
53      Pan_Fetal_Human.pkl   stromal and immune populations from the human ...
```

In this example, since the sample is from PBMCs, we will use the model called Healthy_COVID19_PBMC. The predictions will be made by calling the `celltypist.annotate()` function:

```
predictions = celltypist.annotate(
    adata,
```

```
    model='Healthy_COVID19_PBMC.pkl',
    majority_voting=True,
    over_clustering='leiden'
)
   Input data has 10000 cells and 19951 genes
   Matching reference genes in the model
   3376 features used for prediction
   Scaling input data
   Predicting labels
   Prediction done!
   Majority voting the predictions
   Majority voting done!
```

The `majority_voting=True` parameter assigns a single identity to each of the clusters we identified earlier. CellTypist first predicts each cell individually and then assigns the majority identity to each cluster. The `over_clustering='leiden'` parameter indicates which column in the metadata contains the clusters.

Now, we convert the "predictions" object into an AnnData object, which will be identical to the one we already had, but with the predictions and confidence scores added to the metadata:

```
>>> adata = predictions.to_adata()
```

If we evaluate the object and take a look at its metadata, we will see the new columns:

```
>>> adata
AnnData object with n_obs × n_vars = 10000 × 19951
    obs: 'doublet_scores', 'predicted_doublets', 'n_genes_by_counts', 'log1p_n_genes_by_counts',
'total_counts', 'log1p_total_counts', 'pct_counts_in_top_50_genes', 'pct_counts_in_top_100_genes',
'pct_counts_in_top_200_genes', 'pct_counts_in_top_500_genes', 'total_counts_mt',
'log1p_total_counts_mt', 'pct_counts_mt', 'bad_quality', 'leiden', 'predicted_labels',
'over_clustering', 'majority_voting', 'conf_score'
    var: 'gene_ids', 'feature_types', 'genome', 'mt', 'ribo', 'hb', 'n_cells_by_counts',
'mean_counts', 'log1p_mean_counts', 'pct_dropout_by_counts', 'total_counts', 'log1p_total_counts',
'n_cells', 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg', 'pca', 'neighbors', 'leiden', 'umap', 'leiden_colors',
'majority_voting_colors', 'predicted_labels_colors'
    obsm: 'X_pca', 'X_umap'
    varm: 'PCs'
    obsp: 'distances', 'connectivities'
```

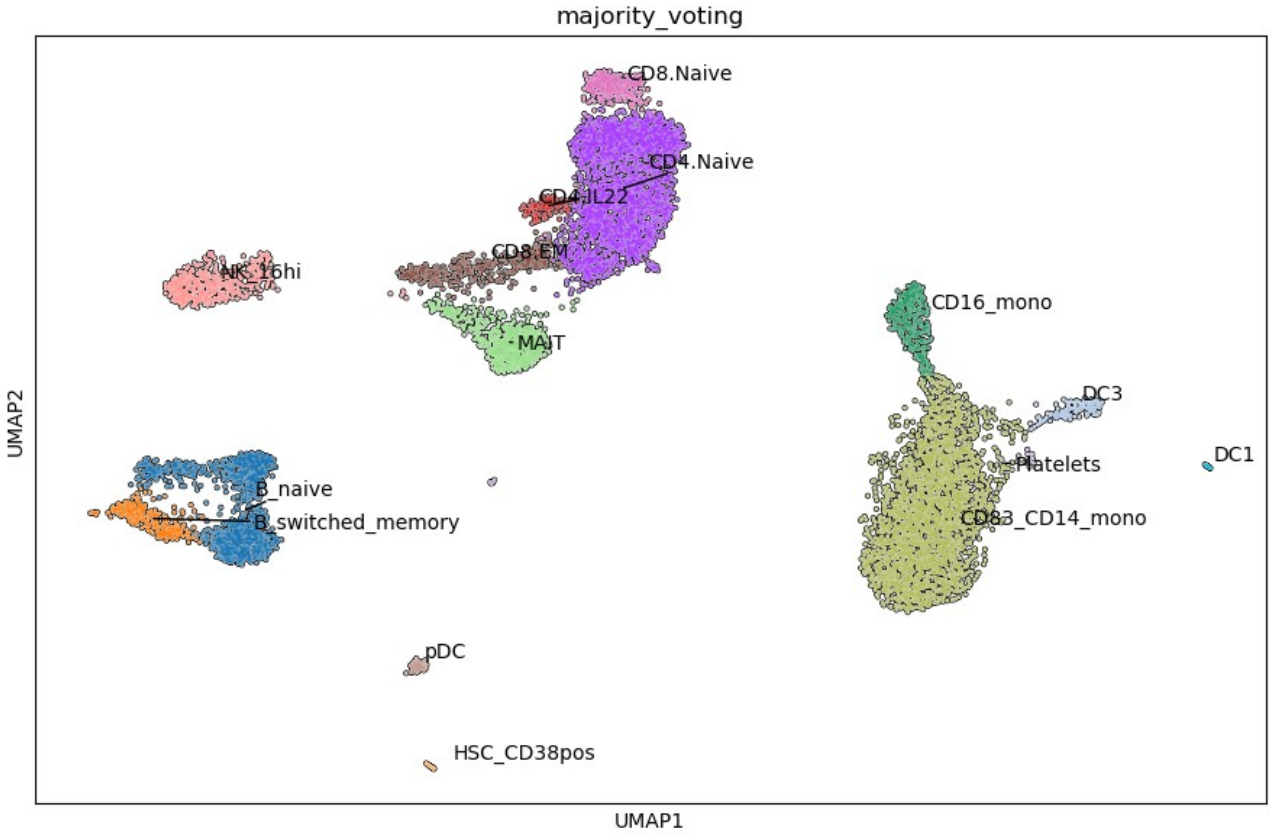
The 'predicted_labels' column contains the predictions at the cell level, while the decisions at the cluster level are found in 'majority_voting'. The 'over_clustering' column contains the same information as the 'leiden' column, as we have indicated. Finally, the 'conf_score' column contains the scores of confidence in the prediction.

Now we can take the last step: visualizing on the UMAP how the annotation turned out. To ensure that the labels on the UMAP do not overlap, we will use this code (taken from https://github.com/scverse/scanpy/issues/1513):

```
def gen_mpl_labels(
    adata, groupby, exclude=(), ax=None, adjust_kwargs=None, text_kwargs=None
):
```

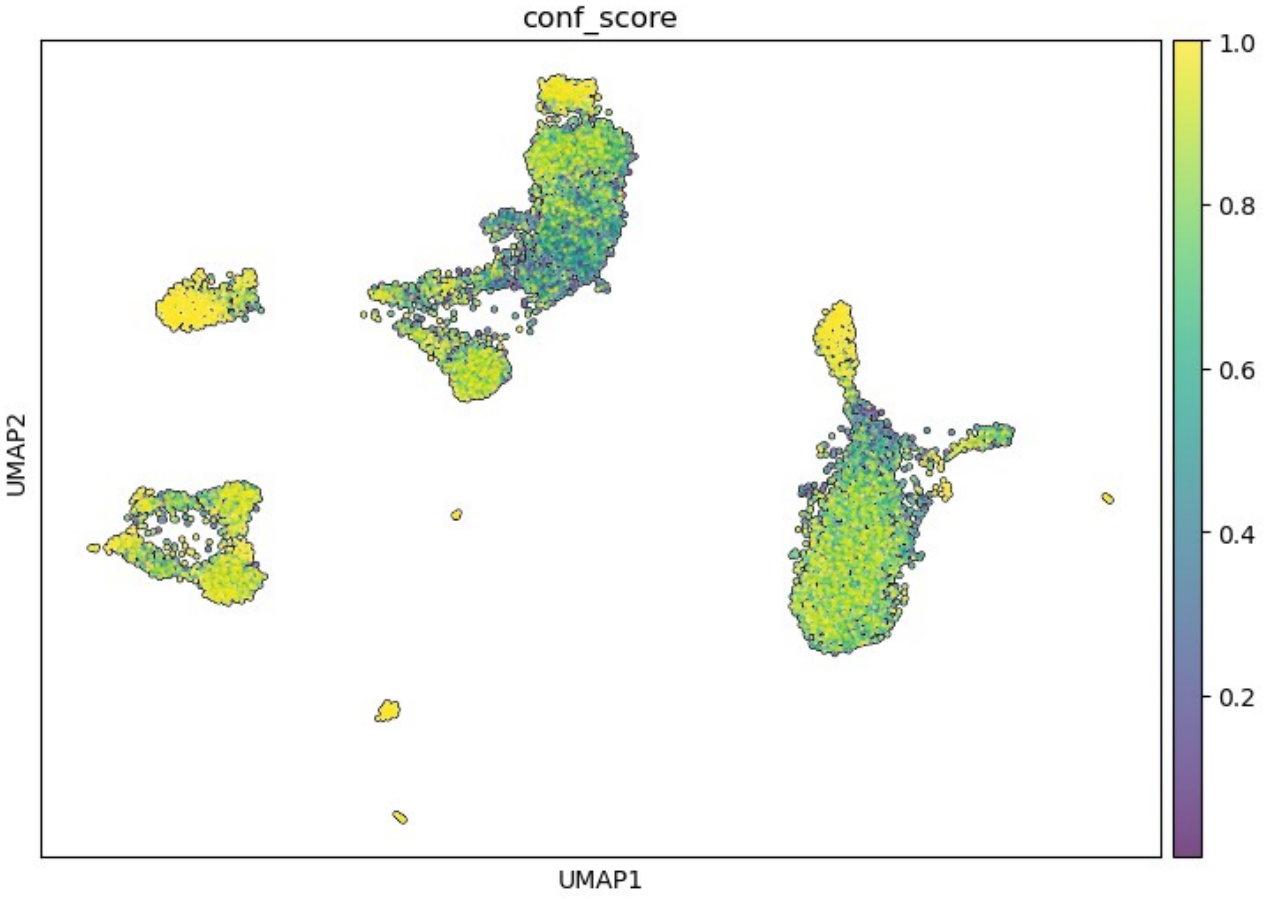
```
    if adjust_kwargs is None:
        adjust_kwargs = {"text_from_points": False}
    if text_kwargs is None:
        text_kwargs = {}

    medians = {}

    for g, g_idx in adata.obs.groupby(groupby).groups.items():
        if g in exclude:
            continue
        medians[g] = np.median(adata[g_idx].obsm["X_umap"], axis=0)

    if ax is None:
        texts = [
            plt.text(x=x, y=y, s=k, **text_kwargs) for k, (x, y) in medians.items()
        ]
    else:
        texts = [ax.text(x=x, y=y, s=k, **text_kwargs) for k, (x, y) in medians.items()]

    adjust_text(texts, **adjust_kwargs)

ax = sc.pl.umap(
    adata,
    color="majority_voting",
    show=False,
    legend_loc=None,
    frameon=True,
    add_outline=True
)
gen_mpl_labels(
    adata,
    "majority_voting",
    exclude=("None",),  # This was before we had the `nan` behavior
    ax=ax,
    adjust_kwargs=dict(arrowprops=dict(arrowstyle='-', color='black')),
    text_kwargs=dict(fontsize=10),

)
fig = ax.get_figure()
fig.tight_layout()
plt.show()
```

This function and subsequent code are used to generate a UMAP plot where each cell is colored according to its majority voting result. The function `gen_mpl_labels` is designed to position the labels effectively so they don't overlap, making the visualization clearer and more informative. First, we defined a function to properly separate the labels, then we created the graph and called the function to modify the location of the labels. This is the result:



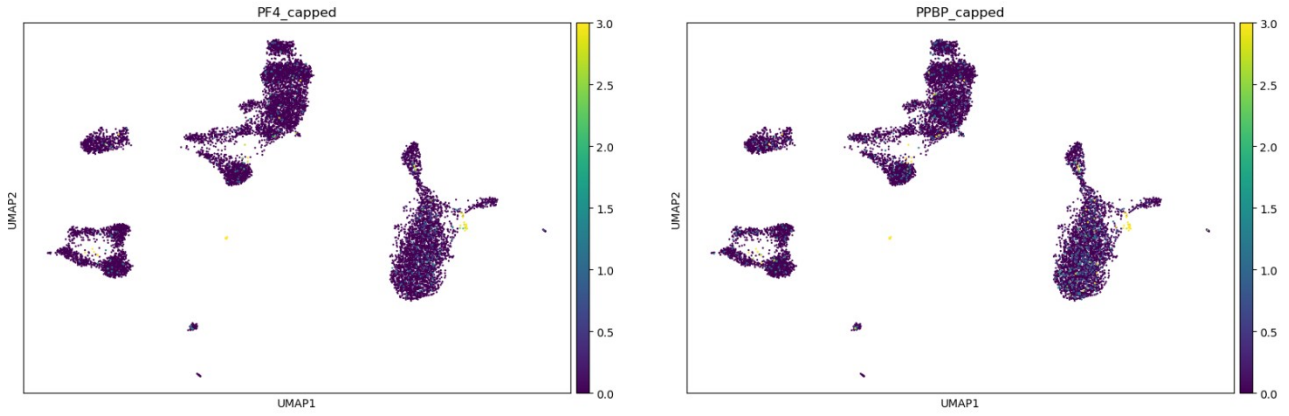
Except for the platelets, which seem to have been divided into two clusters, the rest looks very promising. We can also see the confidence scores:

```
sc.pl.umap(
    adata,
    color=['conf_score'],
    legend_loc='on data',
    add_outline=True,
    show=False
)
```



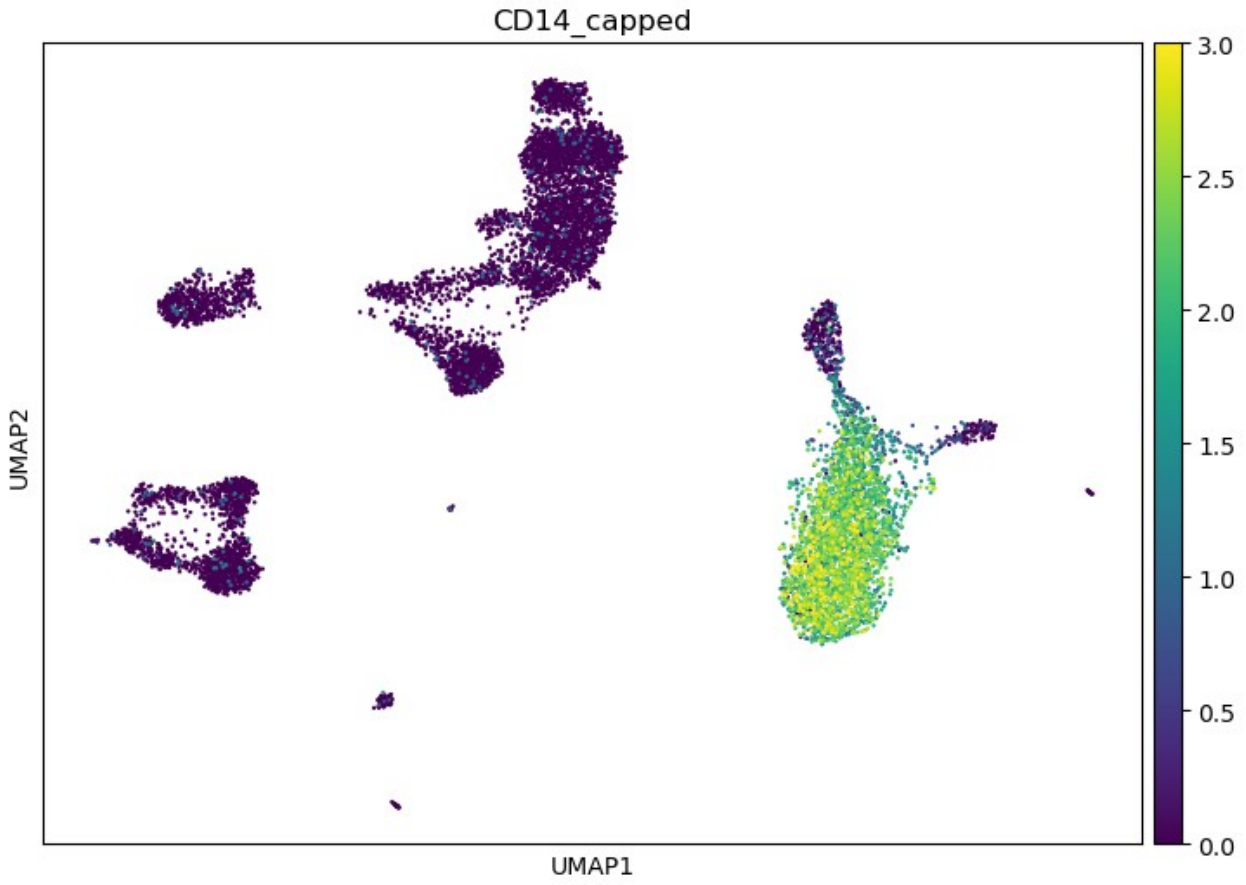
Overall, the predictions seem quite reliable. We can investigate this cluster of platelets more deeply by visualizing on the UMAP the expression of two markers. We will also study the expression of CD14, a monocyte marker, to see that one of the two clusters probably represents a cluster of doublets between monocytes and platelets that was not recognized by Scrublet. First, we visualize the expression of PF4 and PPBP (platelet markers). To facilitate interpretation, we will reduce the maximum expression value on the color scale:

```
max_cutoff = 3
gene = 'PF4'
adata.obs[f'{gene}_capped'] = np.minimum(adata[:, gene].X.toarray().flatten(), max_cutoff)
gene2 = 'PPBP'
adata.obs[f'{gene2}_capped'] = np.minimum(adata[:, gene2].X.toarray().flatten(), max_cutoff)
sc.pl.umap(adata, color=[f'{gene}_capped', f'{gene2}_capped'], cmap='viridis')
```



The platelet clusters clearly express these two markers. Now let's see the expression of the CD14 gene (monocyte marker):

```
gene = 'CD14'
adata.obs[f'{gene}_capped'] = np.minimum(adata[:, gene].X.toarray().flatten(), max_cutoff)
sc.pl.umap(adata, color=[f'{gene}_capped'], cmap='viridis')
```

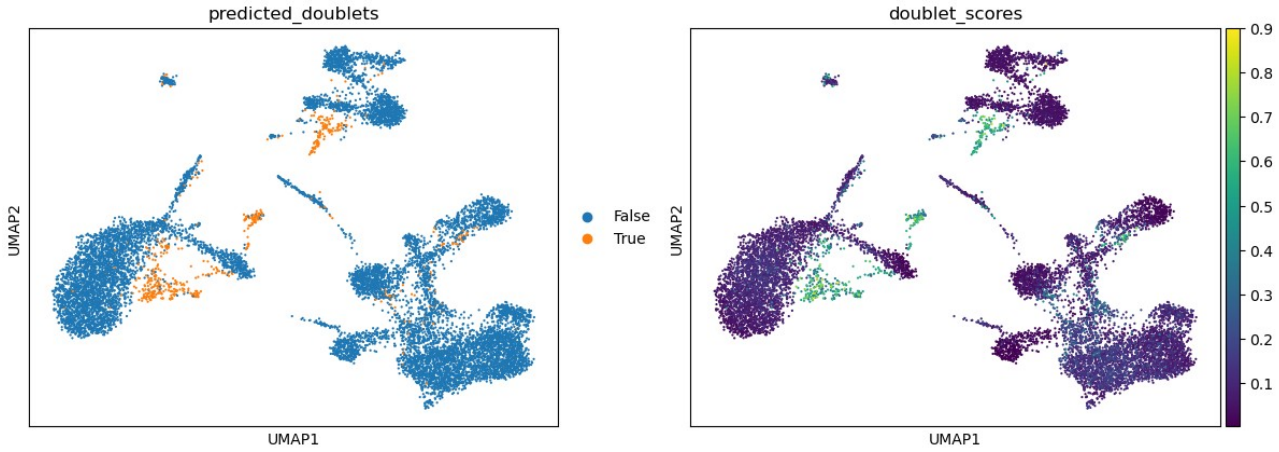


Here it is clear that one of the platelet clusters also expresses this monocyte marker. It likely represents a cluster of doublets of monocytes and platelets. Since platelets are not of interest in PBMC preparations and are considered contaminants, we can opt to ignore them in subsequent analyses or remove them from the data and process them again.

Definition of Quality Variable Thresholds

An efficient way to decide on the thresholds for quality variables is to process the data without performing any quality control (i.e., with all cells coming out of Cell Ranger) and visualize the quality variable values on a UMAP. We'll start by visualizing doublet predictions. We expect different types of doublets to form clusters. We won't put the code to generate the object again, but it's the same as what we used before skipping the removal of poor-quality cells:

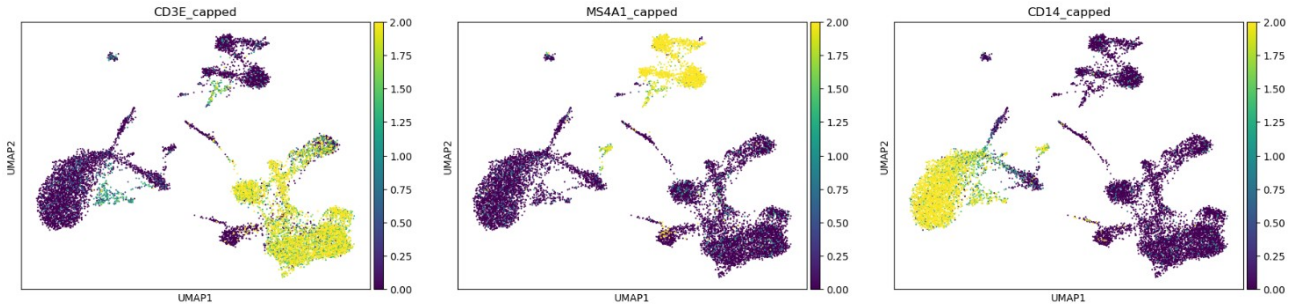
```
sc.pl.umap(
    adata,
    color=["predicted_doublets", "doublet_scores"]
)
```



In this case, it is clearly observed that the doublets cluster. If they didn't, it would be a good idea to keep these cells and wait for future subclusterings (for example, if we decide to isolate T cells to annotate the different subtypes) to see if we find these clusters.

Now we can investigate the typology of these doublets by visualizing the expression of canonical markers for T cells, B cells, and monocytes. Again, we will limit the maximum expression values of the scale to improve interpretation:

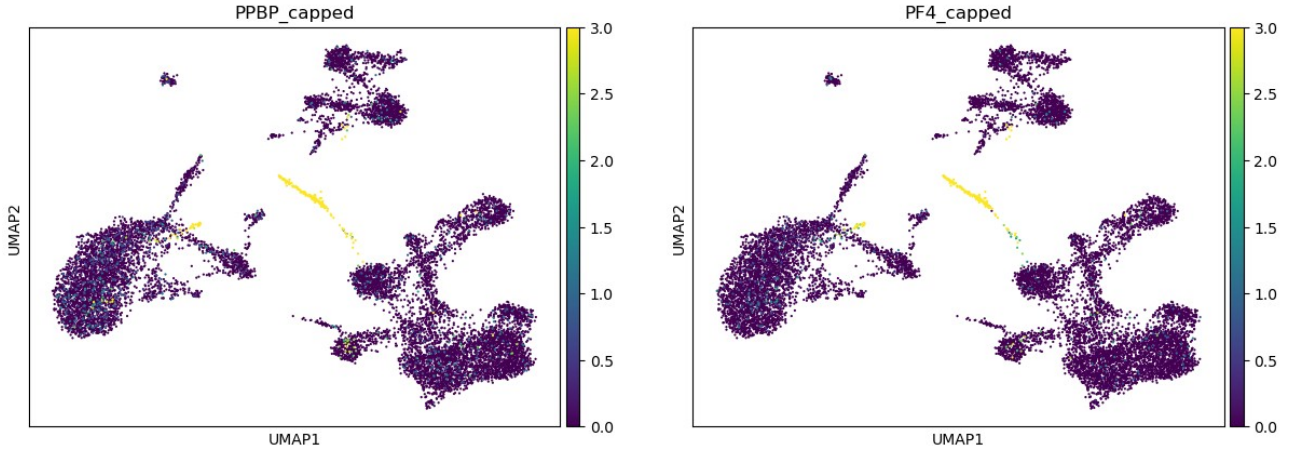
```
max_cutoff = 2
gene = 'CD3E'
adata.obs[f'{gene}_capped'] = np.minimum(adata[:, gene].X.toarray().flatten(), max_cutoff)
gene2 = 'MS4A1'
adata.obs[f'{gene2}_capped'] = np.minimum(adata[:, gene2].X.toarray().flatten(), max_cutoff)
gene3 = 'CD14'
adata.obs[f'{gene3}_capped'] = np.minimum(adata[:, gene3].X.toarray().flatten(), max_cutoff)
sc.pl.umap(adata, color=[f'{gene}_capped', f'{gene2}_capped', f'{gene3}_capped'], cmap='viridis')
```



The three major clusters of doublets correspond to B cells (MS4A1 gene) with T cells (CD3E gene), T cells with monocytes (CD14 gene), and B cells with monocytes.

For other quality metrics, it is necessary to first identify the platelets. We can easily do this with the expression of the PF4 and PPBP genes:

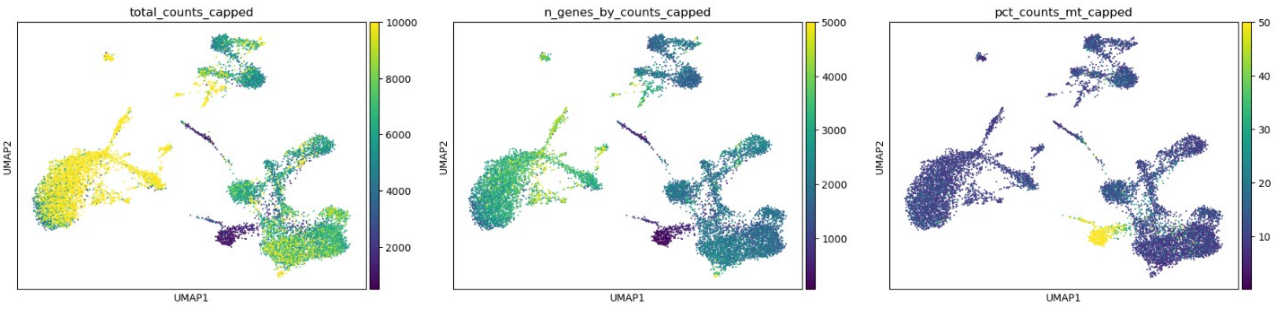
```
max_cutoff = 3
gene = 'PPBP'  # Replace with your gene of interest
adata.obs[f'{gene}_capped'] = np.minimum(adata[:, gene].X.toarray().flatten(), max_cutoff)
gene2 = 'PF4'  # Replace with your gene of interest
adata.obs[f'{gene2}_capped'] = np.minimum(adata[:, gene2].X.toarray().flatten(), max_cutoff)

# Plot the UMAP with the capped expression values
sc.pl.umap(adata, color=[f'{gene}_capped', f'{gene2}_capped'], cmap='viridis')
```



Now we'll explore the total number of transcripts and genes and the mitochondrial percentage. We'll limit the color scales to facilitate interpretation:

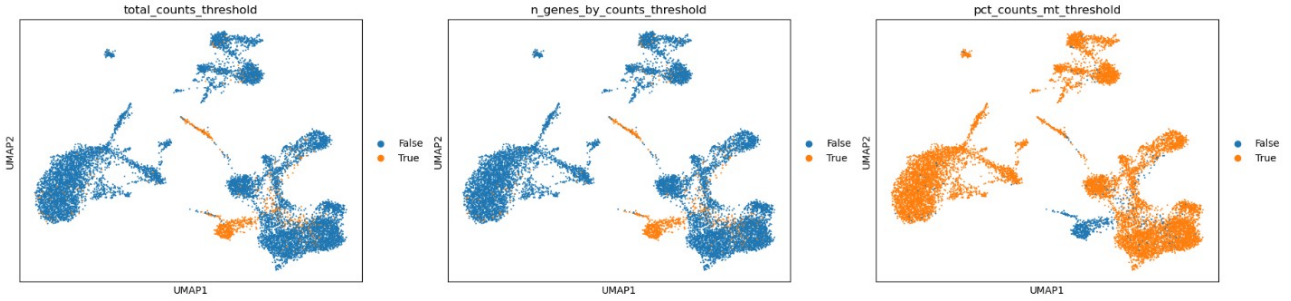
```
adata.obs['total_counts_capped'] = np.minimum(
    adata.obs['total_counts'].values.astype(float), 10000
)
adata.obs['n_genes_by_counts_capped'] = np.minimum(
    adata.obs['n_genes_by_counts'].values.astype(float), 5000
)
adata.obs['pct_counts_mt_capped'] = np.minimum(
    adata.obs['pct_counts_mt'].values.astype(float), 50
)
sc.pl.umap(
```

```
    adata,
    color = ['total_counts_capped', 'n_genes_by_counts_capped', 'pct_counts_mt_capped']
)
```



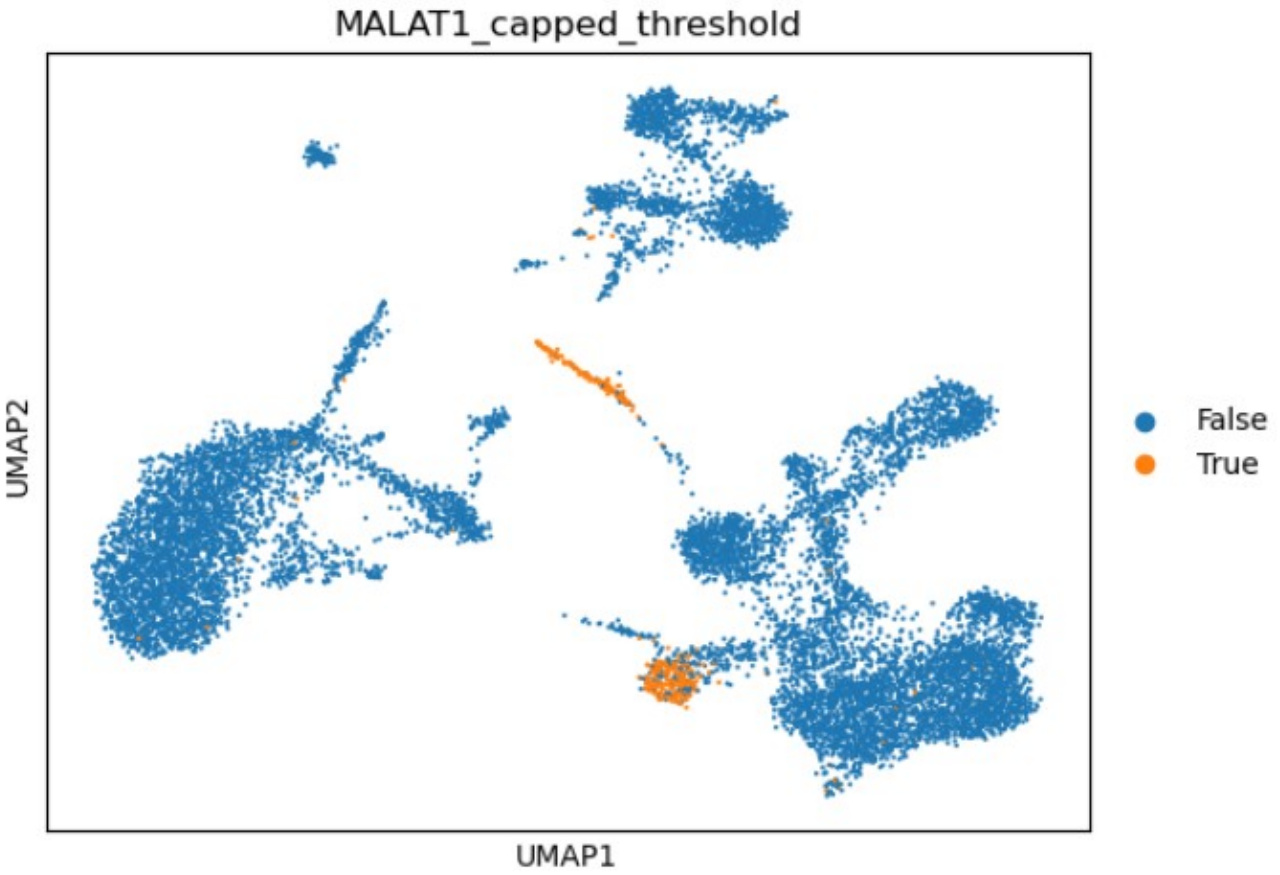
In these UMAPs, we clearly see how the platelets are cells of low complexity (low number of transcripts and genes). We also clearly see a group of poor-quality cells. If we now visualize the cells that are included with the filters we have set, we will see that they clearly select the poor-quality cells and most of the platelets:

```
adata.obs["total_counts_threshold"] = adata.obs.total_counts < 2500
adata.obs['n_genes_by_counts_threshold'] = adata.obs.n_genes_by_counts < 1000
adata.obs['pct_counts_mt_threshold'] = adata.obs.pct_counts_mt < 20
sc.pl.umap(adata,          color=['total_counts_threshold',          'n_genes_by_counts_threshold',
'pct_counts_mt_threshold'])
```



Although it may not seem necessary in these data, in more complex tissue data it can be much more difficult to identify poor-quality cells since the content of ambient RNA (that which is in the solution where we have the cells originating from cells that have broken during the experimental process) can be very high and make it difficult to distinguish what is a cell and what is not. In these cases (and also to identify platelets and erythrocytes), it is advisable to include some nuclear marker, such as the percentage of transcripts with introns or the expression of MALAT1 (a long non-coding RNA located in the nucleus and highly expressed in all nucleated cells). If we apply this to these data, we will see that it is capable of identifying most of the poor-quality cells and platelets. In other words, it will identify the anucleated cells, either because biologically they do not have one (like platelets), because the lysis has not been total and the nuclear transcripts have not been sequenced, or because they are empty droplets:

```
gene = 'MALAT1'  # Replace with your gene of interest
adata.obs[f'{gene}_expression'] = adata[:, gene].X.toarray().flatten()
adata.obs['MALAT1_threshold'] = adata.obs.MALAT1_capped < 4.3
sc.pl.umap(adata, color='MALAT1_capped_threshold')
```



Conclusion

In this post, we have seen how we can process an scRNA-seq sample with Scanpy and automatically annotate the cells with CellTypist. All the code used can be found on the blog's GitHub. The same principles apply when working with more than one sample, but the details will be left for a future post.