

# Welcome to the Demo!

This is your roadmap for processing single-cell RNA-seq data using Scanpy and related tools. We won't have time to cover everything in detail but we will point you at the considerations and directions available to you in the python ecosystem.

## Processing a Single Sample for QC

- Always perform QC on individual samples (before integration)
- You can apply this framework to your concatenated samples when processing the full cohort

1. QC & Filtering
2. Normalization & Dim Reduction
3. Downstream (not in demo)

Do not follow tutorials blind, adapt them to your biological system and expected cell types

There are multiple decision points during single cell QC, and you should make decisions based on your data and biological knowledge

```
%matplotlib inline

import scanpy as sc # scrublet needs scikit-image under the hood

import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd

#read in file - download this from the GitHub repository
adata = sc.read("data/adata_raw.h5ad")
```

## Data Audit

We load in the raw sample data. This is the output from **Cell Ranger**, which contains observations (cells), their features or vars (genes) and a matrix of X (gene counts per cell).

- the **AnnData** object is the standard data structure for single-cell data in Python
- it contains **DataFrames** for a) observations (**obs**) and b) variables (**var**)
- the main data matrix (**X**) is a compressed sparse matrix of gene counts per cell (calculated from Cell Ranger)

```
adata
```

```
AnnData object with n_obs × n_vars = 4323 × 31054
  obs: 'sample'
  var: 'gene_ids', 'feature_types'
```

## More Data Audit

```
# dataframe for vars (genes)
adata.var.head(3)
```

	gene_ids	feature_types
Xkr4	ENSMUSG00000051951	Gene Expression
Gm1992	ENSMUSG00000089699	Gene Expression
Gm37381	ENSMUSG00000102343	Gene Expression

```
# each file was 1 sample, so the sample name should be the same
# the index is the cell barcode unique ID
adata.obs.head(3)
```

	sample
AAACCCAAGATTAGCA-1-ALV5070A4_Oct2022	ALV5070A4_Oct2022
AAACCCAAGGCCGCTT-1-ALV5070A4_Oct2022	ALV5070A4_Oct2022
AAACCCAGTGCTCCGA-1-ALV5070A4_Oct2022	ALV5070A4_Oct2022

```
# cell names (row indices) and gene names (columns)
adata.obs_names[0:3], adata.var_names[0:3]
```

```
(Index(['AAACCCAAGATTAGCA-1-ALV5070A4_Oct2022',
        'AAACCCAAGGCCGCTT-1-ALV5070A4_Oct2022',
        'AAACCCAGTGCTCCGA-1-ALV5070A4_Oct2022'],
      dtype='object'),
 Index(['Xkr4', 'Gm1992', 'Gm37381'], dtype='object'))
```

## The Matrix

```
# view the matrix
adata.X
```

```
<Compressed Sparse Row sparse matrix of dtype 'float32'
  with 8165521 stored elements and shape (4323, 31054)>
```

```
# show min, max, median, mean, zero count prop
from scipy import stats
stats.describe(adata.X.data)
```

```
DescribeResult(nobs=8165521, minmax=(np.float32(1.0), np.float32(20726.0)), mean=np.float32(
```

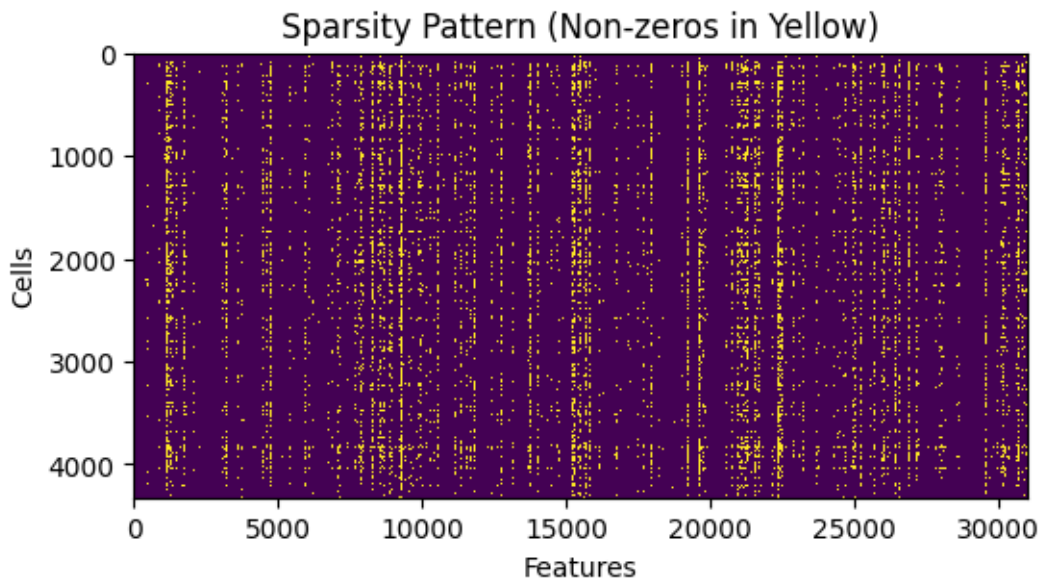
## Why CSR (Compressed Sparse Row)?

- Single-cell data is very sparse (most genes are not expressed in most cells)
- Compressed Sparse Row (CSR) format saves memory and speeds up computations

```
# don't need to code this!
mask = adata.X.toarray() != 0

plt.figure(figsize=(6, 3))
plt.imshow(mask, aspect="auto", cmap="viridis", interpolation="none")
plt.title("Sparsity Pattern (Non-zeros in Yellow)")
plt.xlabel("Features")
plt.ylabel("Cells")
print(f"percentage of non-zeros {round(100 * mask.sum() / mask.size)}")
```

percentage of non-zeros 6



## QC'ing the adata object

Three key metrics to look at **per CELL**

- **Mitochondrial percentage** - a high proportion of mitochondrial counts suggest a cell might be dying
- **Genes** - how many genes have at least 1 count?
- **Depth** - whats the total number of counts we have for a cell?

We need to:

### *Definite Outliers*

- 1) Label definite outlier cells
- 2) Remove definite outlier cells

### *Potential Outliers*

- 3) Label other QC Metrics
- 4) With choice to remove downstream

## Mitochondrial Genes

- Use string methods to label `var_names` that are mitochondrial:
  - create a `boolean` (`True` or `False`) array by querying if any of our `var_names` (gene names) start with `mt-` which all the mitochondrial genes exclusively do.
- You can use this method to label other sets of genes you might want to flag based on your biological knowledge

```
# returns a boolean array of whether the gene names start with "mt-"
adata.var_names.str.startswith("mt-")
```

```
array([False, False, False, ..., False, False, False], shape=(31054,))
```

```
adata.var['mt'] = adata.var_names.str.startswith('mt-') # assign boolean to column in var data
adata.var['mt'].value_counts()
```

```
mt
False    31041
True       13
Name: count, dtype: int64
```

## Calculating QC metrics

Once we have labelled certain genes with `boolean` labels, we can use one of scanpy's inbuilt functions to calculate qc metrics, and its called

- `sc.pp.calculate_qc_metrics`
  - the scanpy **package** (`sc`) has a **module** called `preprocessing` (`pp`) which contains a **function** `calculate_qc_metrics`.

```
sc.pp.calculate_qc_metrics(
    adata,
    expr_type='counts', var_type='genes', #defaults
    qc_vars=["mt"], #var column to use, can supply a list of columns
    percent_top=[20],
    inplace=True,
    log1p=True # run on logged data too
)
```

OMP: Info #276: omp\_set\_nested routine deprecated, please use omp\_set\_max\_active\_levels instea

## Inspect the new adata object

We can see that the function has added multiple `obs` and `var` metrics to our adata

- `n_genes_by_counts` is the number of genes with 1+ counts in a cell
- `total_counts` is the total number of counts per cell (library size)
- `pct_counts_mt` the percentage of counts per cell that are from mitochondrial genes.

```
adata.obs.columns, adata.var.columns
```

```
(Index(['sample', 'n_genes_by_counts', 'log1p_n_genes_by_counts',  
      'total_counts', 'log1p_total_counts', 'pct_counts_in_top_20_genes',  
      'total_counts_mt', 'log1p_total_counts_mt', 'pct_counts_mt'],  
      dtype='object'),  
 Index(['gene_ids', 'feature_types', 'mt', 'n_cells_by_counts', 'mean_counts',  
      'log1p_mean_counts', 'pct_dropout_by_counts', 'total_counts',  
      'log1p_total_counts'],  
      dtype='object'))
```

## Plot our QC metrics

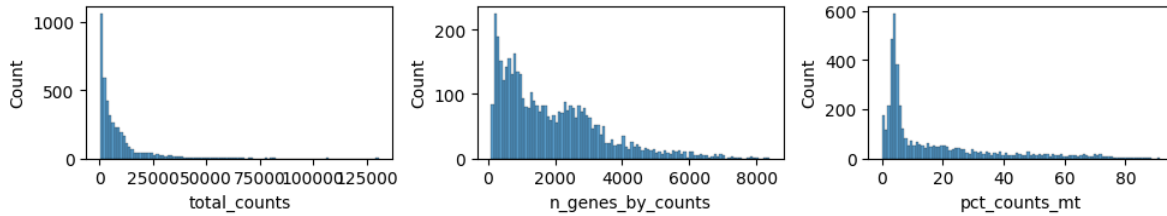
We want to eliminate cells with: - low overall counts, and low numbers of genes identified - poor quality for accurate identification - high amounts of mitochondrial counts - dying cells!

Next we can plot our qc metrics against one another and understand what we're looking for, and what we want to cut off.

## Plotting histograms

Scanpy doesn't have a built in histogram plotter, so let's use Seaborn to inspect the histograms first. - don't worry about coding these now

```
fig, axes = plt.subplots(1, 3, figsize=(10,2))  
sns.histplot(adata.obs["total_counts"], bins=100, kde=False, ax=axes[0])  
sns.histplot(adata.obs["n_genes_by_counts"], bins=100, kde=False, ax=axes[1])  
sns.histplot(adata.obs["pct_counts_mt"], bins=100, kde=False, ax=axes[2])  
  
plt.tight_layout()
```



- **total counts (library size)**: Sharp cut off for the minimum - cells with less than this were trimmed upstream due to the technology
- **n\_genes\_by\_counts (number of genes detected)**: Most cells have around 1k-3k genes detected (of ~20k...)
- **pct\_counts\_mt** (percentage of counts from mitochondrial genes)

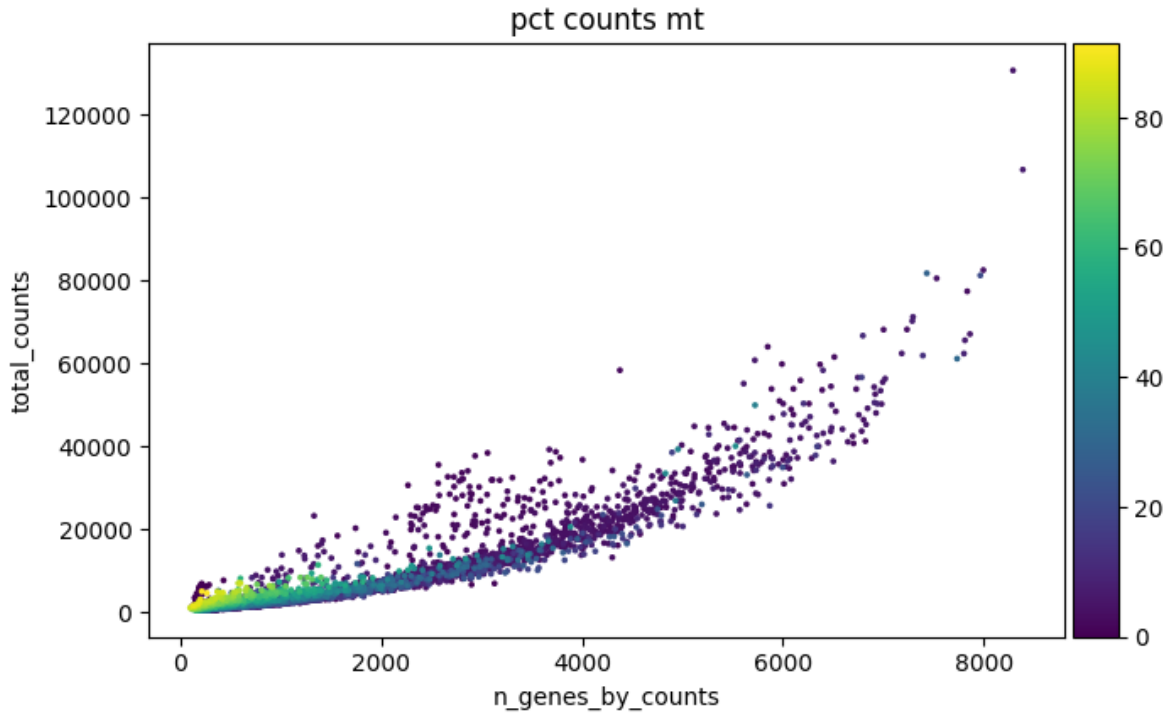
## Using Scanpy's `scatter()` method

Plotting using matplotlib or seaborn is great, but scanpy also conveniently has a plotting **module** (`pl`).

- `sc.pl.scatter(adata, <variable 1>, <variable 2>, color= <variable 3>)`
- `sc.pl.violin(adata, <variable 1>)`

Visualize our three favorite metrics with a coloured **scatter plot**

```
sc.pl.scatter(adata, x="n_genes_by_counts", y="total_counts", color="pct_counts_mt")
```



Warning: do not follow this blindly: some cells with high MT percentage may be biologically relevant (e.g. muscle cells). Always consider your biological context!

## Finding Outlier Cells & Setting thresholds

It's one thing to spot outliers, it's another to identify them robustly and reproducibly. In single cell data we have to process each sample individually and therefore these diagnostic plots can be different sample to sample.

We can apply **automatic thresholding** to solve this problem!

- we invite you to use MAD thresholding per sample
- find more information on the Single Cell Best practises book

## Quick and dirty version - hard thresholds

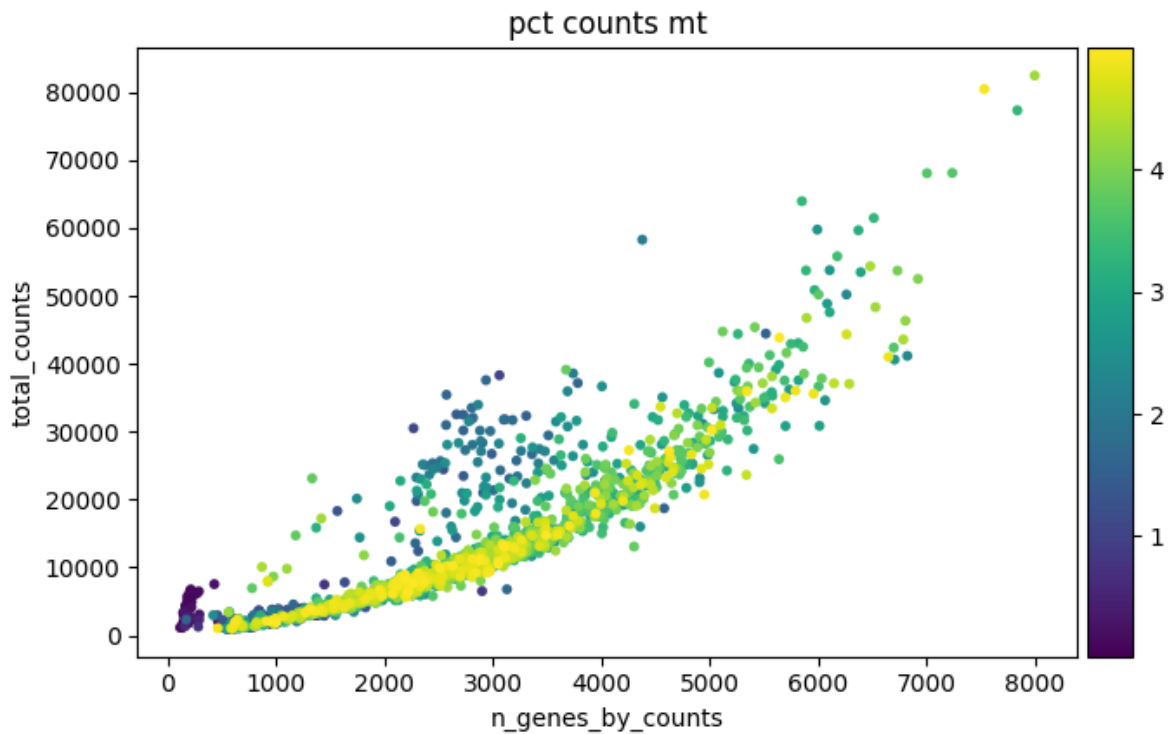
For the purposes of this workshop we will set manual thresholds using boolean selectors

```
adata.obs["outlier_mt"] = adata.obs["pct_counts_mt"] > 5 # mt % threshold
adata.obs["outlier_totalcounts"] = adata.obs["total_counts"] < 1000 # library size threshold

adata.obs["outlier"] = (adata.obs["outlier_mt"] | adata.obs["outlier_totalcounts"]) #union

adata = adata[adata.obs["outlier"] == False ] #keep the non outliers

sc.pl.scatter(adata, x="n_genes_by_counts", y="total_counts", color="pct_counts_mt")
```



## Scanpy Filtering

if you wanted to do a quick filtering, scanpy has two inbuilt functions where you can give a hard coded metric to filter the genes and cells by, but we advise using the automated thresholding approach

- `sc.pp.filter_cells(adata, min_genes=100)` #remove cells with poor signal
- `sc.pp.filter_genes(adata, min_cells=3)` #good to remove genes with little signal

## Doublets

- Each observation is a cell but actually that observation is a **droplet** and we presume it contains 1 cell
- in reality, some droplets can contain doublets - where two different cells are captured and sequenced as one

there are several algorithms to identify doublets - Scanpy comes with **scrublet**, but you can also download **DoubletDetector** from PyPI. - we want to **FLAG doublets**, decide on their removal later (dependent on experiment type) - We recommend running two different algorithms and taking the intersection of the predicted doublets. However if pressed for time, running one algorithm is sufficient

## Running Scrublet

First - **Scrublet** - the inbuilt scanpy function

- uses **scikit-image** under the hood, that's why we needed it in our env
- simulates artificial doublets from real data, scoring each real cell based on similarity to simulated doublets

```
sc.pp.scrublet(adata)
adata.obs[["doublet_score", "predicted_doublet"]].head(5) # adds score & boolean to obs
```

```
/Users/whittog/Documents/BABS/src/sc-fog/.venv/lib/python3.10/site-packages/scanpy/preprocess
adata.obs["doublet_score"] = scrubbed["obs"]["doublet_score"]
```

	doublet_score	predicted_doublet
AAACCCAAGGCCGCTT-1-ALV5070A4_Oct2022	0.064295	False
AAACCCAGTGCTCCGA-1-ALV5070A4_Oct2022	0.037975	False
AAAGAACAGTGCACAG-1-ALV5070A4_Oct2022	0.052257	False
AAAGAACCAGTCTGGC-1-ALV5070A4_Oct2022	0.035714	False
AAAGAACCATCCCGTT-1-ALV5070A4_Oct2022	0.037975	False

## Running Doublet Detection

Second - **DoubletDetection** - needs to be imported separately (and installed from PyPI)

- Uses synthetic doublets embedded in a clustered context, looking for enrichment of synthetic doublets in local neighborhoods

```
import doubletdetection
clf = doubletdetection.BoostClassifier()

labels = clf.fit(adata.X).predict()
scores = clf.doublet_score()

adata.obs["doublet_detector_score"] = scores # add score
adata.obs["doublet_detector_doublet"] = labels # add boolean
```

```
/Users/whittog/Documents/BABS/src/sc-fog/.venv/lib/python3.10/site-packages/tqdm/auto.py:21:
  from .autonotebook import tqdm as notebook_tqdm
100%|      | 10/10 [00:37<00:00,  3.71s/it]
```

```
# view concordance between two methods
pd.crosstab(
    adata.obs["doublet_detector_doublet"],
    adata.obs["predicted_doublet"]
)
```

	predicted_doublet	
doublet_detector_doublet	False	True
0.0	1677	4
1.0	8	2

## Scoring Cells based on Gene Expression “Signatures”

We often want to score cells based on the expression of sets of genes that represent biological processes or cell types

**Gene Set** - each cell gets a score based on the average expression of a set of genes

```
sc.tl.score_genes(adata, gene_list, score_name='score')
```

**scoring the cell cycle** - theres a specific function for scoring the cell cycle - each cell gets a score for S phase and G2M phase based on known gene sets, - additionally, each cell gets a category for its predicted cell cycle phase

```
s_genes_list = []
g2m_genes_list = []

sc.tl.score_genes_cell_cycle(adata, s_genes=s_genes_list, g2m_genes= g2m_genes_list)
```

## Dimensionality reduction in sc-RNA-Seq

- Dimension reduction methods aim to summarise high-dimensional data comprised of many variables with a smaller number of new dimensions
- We will focus on two dimension reduction methods commonly used in sc-RNA-Seq
- Principal Compnents Analysis (PCA)
  - A linear method that finds lines of best fit through multi-dimensional data sets
- Uniform Manifold Approximate Projection (UMAP)
  - A non-linear method that groups cells that are close in multi-dimensional space

## Why use multiple dimensional reduction techniques?

- In single-cell, we typically first summarise the expression data with PCA
- Some number of components are then selected to represent the data
- These components are then summarised further using UMAP for visualisation
- This helps to reduced over fitting and also minimises computation time

## Preparing Anndata for dimensionity reduction with PCA

- Before computing prinicpal components, we need to pre-process data:
  - Normalisation - account for variation due to sequencing depth across cells
  - Log-transformation - helps stabilise the variances
  - Identify highly-variable genes
  - Centering and Scaling - helps equalise the contribution of variables on different scales

## Stash the original counts in a layer

- We are going to be modifying the X data inplace
- We can keep a copy of the original X data in a layer called counts
- The `.copy()` method is really important - otherwise counts is a reference to `.X`

```
adata.layers['counts'] = adata.X.copy() # creates a deep copy
```

## Normalise

- The `normalize_total()` method adjusts all cells to have the same total counts
- The `target_sum` parameter defaults to the median total counts across all cells
- We will use a target of 10K for compatibility with CellTypist (later)

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

## Log transform the data

- The next pre-processing step is log transformation
- A *pseudocount* of 1 is added to all cells to avoid log transforming zeros
- Again, this modifies the `.X` data

```
sc.pp.log1p(adata)
```

## Stash the normalised and log1p data

- Save a copy for later use with CellTypist

```
adata.layers['log-norm10k'] = adata.X.copy()
```

## Identify highly-variable genes

- The `highly_variable_genes()` method of the scanpy pre-process module has various options to do this
- We are going to use the default `flavor='seurat'` approach on the normalised data
- This adds a column of bools to `adata.var`

```
sc.pp.highly_variable_genes(adata, batch_key="sample", flavor='seurat', n_top_genes=2000)
```

## Why only 2000 highly-variable genes?

- Our data set contains over 31k genes, but we only use the top 2000 highly variable genes
- Focusing on genes that actually vary across the set of cells:
  - reduces computational complexity
  - focuses on the genes that are actually useful to differentiate cell types

## Center and scale the data

- Centering and scaling sets the average expression of all genes to zero and adjusts standard deviation to be one
- **NB** This *densifies* the X data, increasing the memory footprint

```
sc.pp.scale(adata)
```

## Fit the PCA model

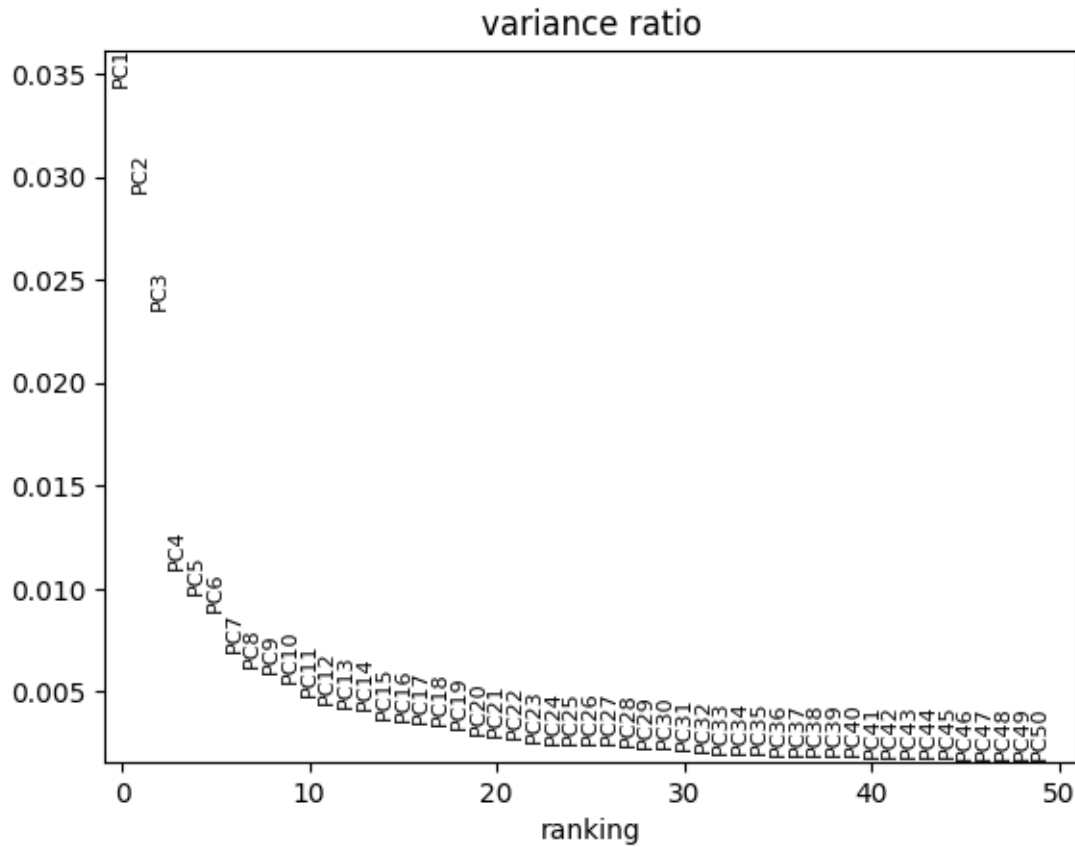
- This defaults to n\_comps=50
  - The PCA observation scores are stored in `.obsm['X_pca']`
  - Other results related to PCA (eg. variance explained) are stored in `.uns['pca']`

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

## Visualise explained variance

- We can get a feel for how many components are needed to extract the most useful structure in the data set (components after PC30 are not doing much)

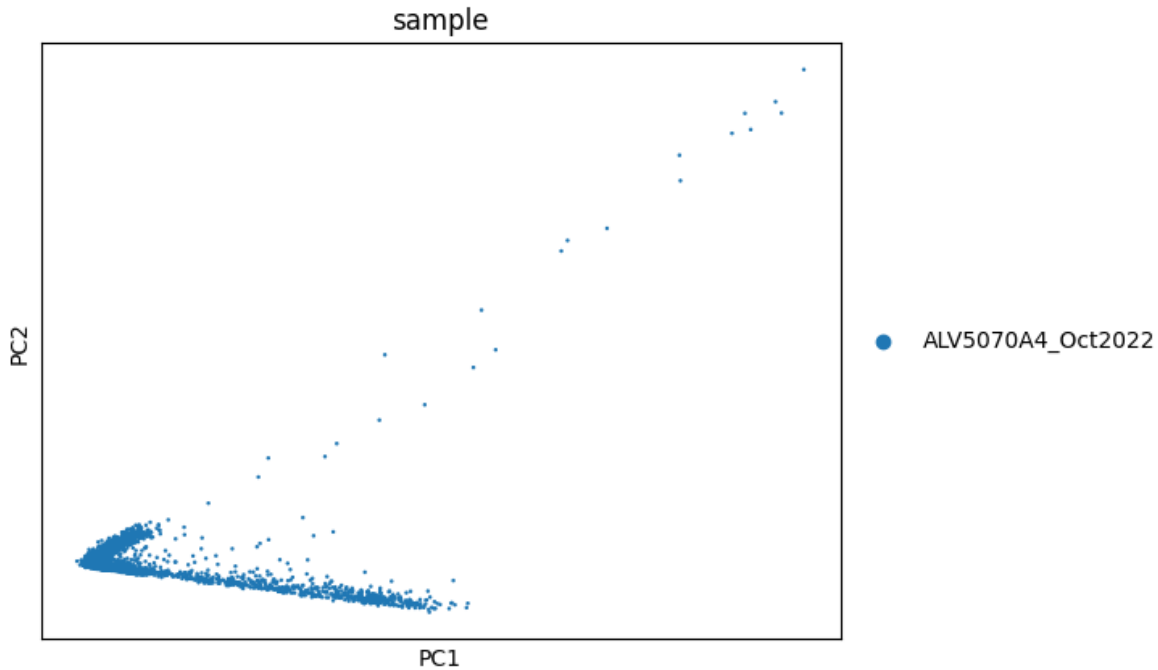
```
sc.pl.pca_variance_ratio(adata, n_pcs=50, log=False)
```



## Visualise pairs of PCs

- We can visualise cells with plots of pairs of PCs (dimensions)
- You could colour cells using metrics in your `adata.obs` dataframe

```
sc.pl.pca(adata, color='sample',
          dimensions=[(0, 1)], size=12)
```



## Visualising complex data with UMAP

- UMAP works in a fundamentally different way to PCA
- A shared nearest-neighbour graph is constructed, linking cells that are close in the high-dimensional space
- This graph is then embedded in a lower-dimensional space
- This makes it excellent for crushing complex data sets down into a 2D scatter plot

## Factors impacting UMAP

- We need to select a number of PCs to compute the shared nearest-neighbour graph from
  - scanpy defaults to 50
  - Its important to have enough to summarise the complex data
  - Including a few more unimportant PCs is less of a problem
- We need to select the number of nearest neighbours used to built the graph
  - start with the default (`n_neighbors=15`)
  - higher values emphasise global structure

## Compute the k-nearest neighbours graph

- The `.neighbors()` method from the preprocessing module computes the nearest neighbour graph
- The distances and connectivities get stored in `adata.obsp`
- The value of `k` is controlled by `n_neighbors` (defaults to 15)

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

## Embed the neighbourhood graph with UMAP

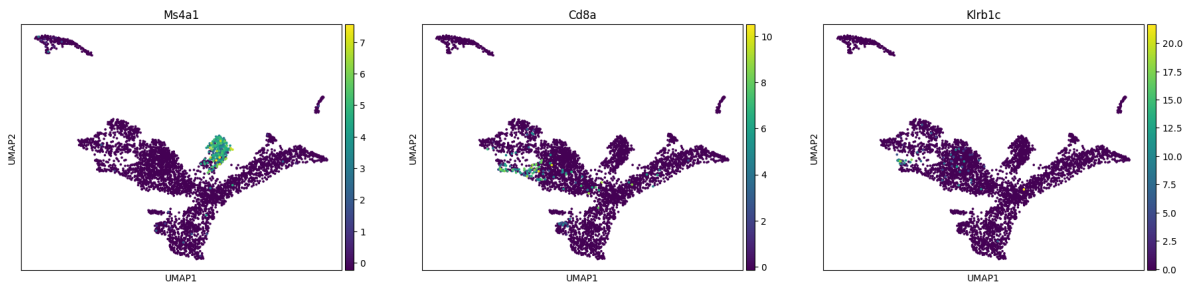
- The `sc.tl.umap()` method computes the neighbourhood embeddings
- The embeddings are stored in `adata.obsm` with the key `X_umap`

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

## Visualise the UMAP and colour by expression

- `sc.pl.umap()` plots the UMAP
- Supply a list of column names to colour cells
- This could be useful when manually investigating markers of cell identity

```
sc.pl.umap(adata, color=['Ms4a1', 'Cd8a', 'Klrb1c'])
```



## Cluster identification with marker genes

- We can see that *Ms4a1* forms a cluster of (likely) B-cells
- A typical workflow would be to run a clustering method (eg. `sc.tl.leiden()`) and identify clusters by marker genes expressed (`sc.tl.rank_genes_groups()`)
- This is important, but we will show you a quicker way (time is short)

## Cell annotation with reference atlases

We next focus on automated reference-based annotation with **CellTypist** using a pre-annotated atlas

```
# setup for this notebook
import warnings
warnings.filterwarnings('ignore', message=r'[\r\n]+.*Performance')
```

### What is atlas based annotation?

- We can use a pre-annotated reference dataset (or atlas) to assign cell types in our dataset.
- The reference atlas contains gene expression profiles for various cell types
- Many consortia around the world are working to generate atlases for different tissues and organisms
- There are too many to mention here (Google is your friend), but check out Cellxgene

### Key considerations when using atlas based annotation

**Reference Dataset Quality:** Must be well-annotated and biologically relevant

**Species Compatibility:** Ensure reference and query data are from the same species

**Batch Effects:** Differences in sequencing platforms or protocols can affect annotation accuracy

**Cell Type Coverage:** Reference must include the diversity of cell types expected in your sample - additionally, you may encounter novel cell types not present in the reference

### Load the celltypist package

- CellTypist is a Python package that uses logistic classification to perform automated cell type annotation
- It comes with several pre-trained models for different tissues and organisms, or you can train your own model using a custom reference dataset

```
import celltypist
```

## Load the model

- For this analysis we can use the `Adult_Mouse_Gut.pkl` that comes with Celltypist
- We can use the atlas because we expect to find all these cell types in our experimental data

```
model = celltypist.Model.load(  
    "data/Adult_Mouse_gut.pkl" # path to where yours is saved!  
)                               # ~/.celltypist/data/models/?  
print(model)
```

CellTypist model with 126 cell types and 8258 features

```
date: 2022-08-08 05:26:50.850176  
details: cell types in the adult mouse gut combined from eight datasets  
source: https://doi.org/10.1038/s41586-024-07251-0  
version: v2  
cell types: Activated CD4+ T cell, B cell, ..., vein Madcam1+  
features: 0610005C13Rik, 0610009B22Rik, ..., mt-Tc
```

## Prepare the adata

- Celltypist requires the data to be log-normalised, and specifically normalised to a total count of 10,000 per cell.
- We stashed data like this earlier in a layer called *log-norm10k*
- Lets switch layers around

```
adata.layers["scaled"] = adata.X.copy()  
  
adata.X = adata.layers["log-norm10k"]
```

## Run CellTypist

Now that our data is in the correct format we can run cell typist!

```
predictions = celltypist.annotate(adata, model=model, majority_voting=True)
```

```
Input data has 4323 cells and 31054 genes
Matching reference genes in the model
7822 features used for prediction
Scaling input data
Predicting labels
Prediction done!
Detected a neighborhood graph in the input object, will run over-clustering on the basis of
Over-clustering input data with resolution set to 5
Majority voting the predictions
Majority voting done!
```

## Copy useful columns from the predictions object

- `predictions.predicted_labels` contains information on the predicted cell types
- `predictions.probability_matrix` contains the probabilities for all possible cell types per cell - take the row maximum

```
adata.obs["celltypist_labels", "over_clustering", "majority_voting"] = (
    predictions.predicted_labels
)
adata.obs["celltypist_confidence"] = predictions.probability_matrix.max(axis=1)
```

## CellTypist annotation – What just happened?

### Prediction:

- Each cell's expression profile was compared to the reference model
- The classifier assigned a predicted cell type label to each cell
- It also calculated a confidence score, reflecting how strongly the model supports each prediction

### Results integration:

- Predictions were stored in `adata.obs["celltypist_labels"]`
- Confidence scores were stored in `adata.obs["celltypist_confidence"]`

## Interrogate the results

- At this point, you will want to take a good look over how well the cell types were predicted
- We are short on time here, but you could:
  - Visualise the counts of each predicted cell type (individually and for majority voting)
  - Visualise the distributions of confidence scores for each cell type

## Visualise annotations in UMAP space

We will now visualize the cell typist annotations in UMAP space to explore how the predicted cell types are distributed across our clusters. This allows us to:

- assess how successful the model predicts the cell types with `celltypist_confidence`
  - find any clusters of uncertain cell type predictions
- visualize the spatial distribution of different cell types in the UMAP embedding

```
sc.pl.umap(  
    adata,  
    color=["celltypist_labels", "majority_voting", "celltypist_confidence"],  
    legend_loc="on data",  
    title=['Cell Typist Cell Type Annotations', 'majority voting', 'celltypist score'],  
    frameon=False,  
)
```

