Scanpy tutorial

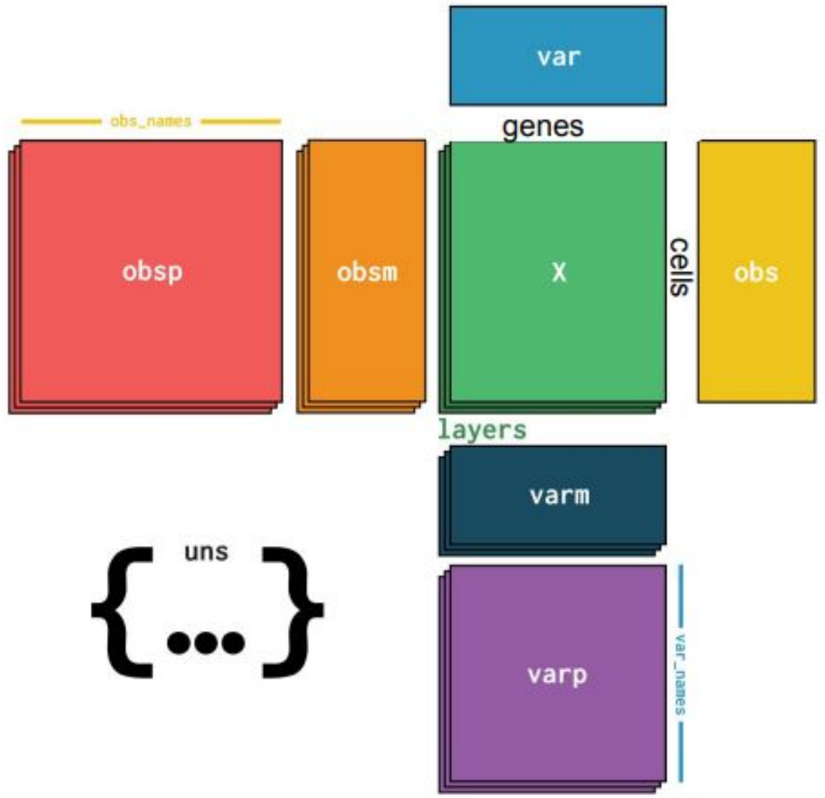
Lesson 8

Scanpy

- A comprehensive toolkit for Single Cell analysis
- scRNAseq poses unique challenges due to its high dimensionality and inherent noise
- Various tools have been developed to assess these problems, but Scanpy stands out as a comprehensive and versatile toolkit
- Wide range of functionalities, modular design allows to seamlessly perform quality control, dimensionality reduction, clustering, trajectory inference…
- Application in biology, immunology, disease research and pharmacology

Annotated data object

- Python package for handling annotated data matrices
- ***n x d*** dimensions
- ***n*** observation, cells (rows) each represented by a ***d***-dimensional vector of features, genes (columns)
- .X contains gene expressions, supports sparsity to save space
- Value in intersection of row (cell) and column (gene) quantifies gene transcriptomes of that specific gene found in that specific cell
- Zeros are not included in sparse shapes



Annotated data object

- Both rows and columns are indexed
- ***.obs*** – pandas dataframe containing row indices (cell barcodes) and observations' metadata (donor info, cluster names, coordinates etc.)
- ***.var*** – pandas dataframe containing column indices (gene ids) and variables' metadata
- ***.obs_names, .var_names*** – index names
- ***.n_obs, n_var*** – anndata dimensions
- ***.obsm*** – dictionary that contains metadata dense matrices about observations (***n x ?***, embeddings obtained from PCA, UMAP…)
- ***.varm*** – dictionary that contains metadata dense matrices about variables (***d x ?***, coefficients obtained via factor analysis)
- ***.uns*** – unstructured metadata, dictionaries or lists with some general information useful in the analysis (cluster to colour mapping)

Starting with anndata

- pip install anndata – package installation
- pip install scanpy
- import anndata as ad – import package
- import scanpy as sc
- Read from a file:
  - adata = ad.read_h5ad(path) - default file extension
  - adata = ad.read_mtx(path), read_csv, read_txt, read_hdf
- Create from scratch:
  - adata = ad.Anndata(sparse_example) – results in anndata with sparse_example as .X and obs_names and var_names as numbers 1, 2, 3…
  - obs_names and var_names can be additionaly changed
  - adata = ad.Anndata(sparse_example, obs_example, var_example) – creates anndata with .X, .obs and .var, but obs_names and var_names need to be added additionally

Accessing anndata objects

- Subsetting anndata by index (not typical):
  - By observation

    subset_adata = adata[10:20, :] – rows 10 – 20 (cells), all columns (genes)
  - By variables

    subset_adata = adata[:, 100:200] – all rows (cells), columns 100 – 200 (genes)
  - Individual cell

    cell_data = adata[5, :] – cell 5 with all genes
- Subsetting anndata by obs/var_names:
  - *subset_adata = adata[[‘Cell_1’, ‘Cell_2’], [‘Gene_5’, ‘Gene_1900’]]* – result: View of AnnData object with n_obs × n_vars = 2 × 2
- Conditional subsetting:
  - Based on observation

    subset_adata = adata[adata.obs['cell_type'] == 'T cell', :] – returns subset of original anndata where cell_type has ‘T cell’ value with all genes

Accessing anndata objects

- Conditional subsetting:
  - Based on variables

    subset_adata = adata[:, adata.var['gene_type'] == 'protein_coding’] - returns all cells and genes responsible for protein coding

    subset_adata = adata[:, adata.var['chromosome'].isin(['chr1', 'chr2', 'chr3'])] - return all cells and genes located on specific chromosomes

  - Combined conditional slicing:

    subset_adata = adata[(adata.obs['cell_type'] == 'T cell') & (adata.obs[‘total_counts’] > 500), :] - select T cells with total count higher than 500

  - Combined conditional slicing:

    subset_adata = adata[~(adata.obs['cell_type'] == 'T cell'), :] - select cells not labeled as 'T cell’

    subset_adata = adata[:, ~adata.var['gene_type'].isin(['pseudogene', 'coding_RNA'])] - select genes not annotated as pseudogenes and non-coding RNAs

- Different ways of subsetting can be combined creating complex slicing conditions

Anndata layers

- Layers in Anndata objects are used to store additional data or transformations of the main data matrix (.X)
- Access layers using ***.layers*** attribute, which is dictionary-like object

  adata.layers[‘layer_name’]
- Adding layers

  adata.layers[‘log_transformed’] = np.log1p(adata.X)
- Various operations and analyses can be performed on layers in Anndata objects, similar to working with the main data matrix (visualization, dimensionality reduction etc.)
- Common layer types
  - 'raw': Typically used to store raw unnormalized count data.
  - 'norm_data': Used to store normalized expression data.
  - 'scaled': Used to store scaled or transformed data, such as log-transformed or scaled count data.

Other operations with Anndata objects

- Adding aligned metadata

  adata.obs[“annotation"] = annotation_list

- Conversion to DataFrames

  new_df = adata.to_df()

- Copy operation

  ad_copy = adata.copy() – creates a new identical object of anndata in memory

- Writing anndata to file

  adata.write(output.h5ad’)

- *adata.var_names_make_unique()* – ensuring that the variable (gene) names are unique. Dupplicates get suffixes appended to make them unique. Useful when merging or concatenate multiple datasets with overlapping sets of variables

View of Anndata

- Indexing, slicing and adding new metadata create View of Anndata object
- A view object is a reference to the original data, not a copy
- Changing .X attribute of a view makes changes to the original anndata
- **Careful!** Changing views .obs or .var (although a reference to the original) causes the making of a copy of the original object in the memory (not a reference anymore)

Starting with scanpy

- pip install scanpy – installation of the package
- import scanpy as sc – importing the package
- Configuring Python environment (optional):

  sc.settings.verbosity = *3* – receive errors (0), warnings (1), info (2) and hints (3)

  sc.logging.print_header() – prints header for logging purposes, provides software versions and dependencies

  sc.settings.set_figure_params(dpi=80, facecolor=*"white")* – sets parameters for figures generated by Scanpy, resolution (dots per inch – dpi) to 80 and back colour of figure to white

Task

- We are using GTEX-1HSMQ-5005 data set
- It contains 14k cells from lung tissue
- Obtained from 3 healthy human donors
- Goal:
  - Analyse and prepare the data set
  - Find meaningful clusters and annotate them
- Link to code - https://colab.research.google.com/drive/1t4Ov1k1Idjp3TW1Z-QmEgP8DdhZNvEtF#scrollTo=OKG457pckoZg

Preprocessing - Filtering data

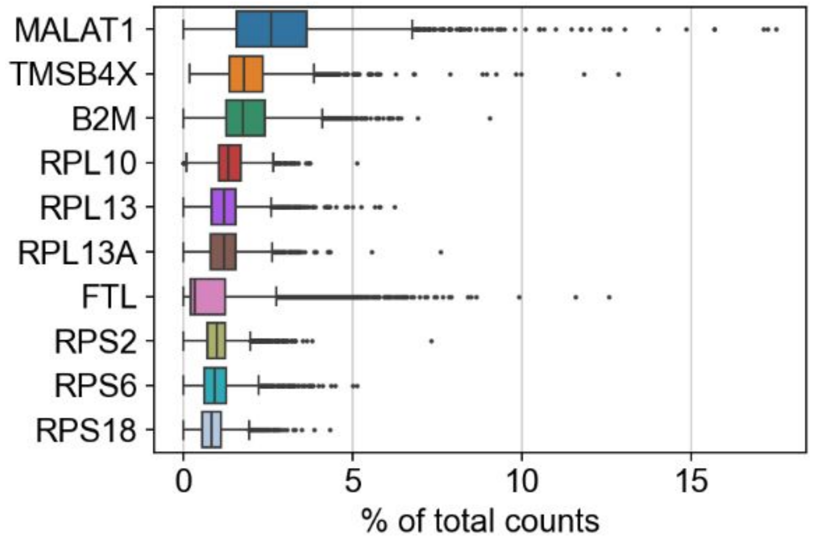
- Show most expressed genes in the data set:

  sc.pl.highest_expr_genes(adata, n_top=10)

- Are they necessarily most important for the analysis?
- Filter cells that have low number of genes (< 200), low-quality cells – removes noise
- Filter genes that are present in low number of cells (< 3), cell-specific genes, removes noise and bias

  sc.pp.filter_cells(adata, min_genes=200)

  sc.pp.filter_genes(adata, min_cells=3)



Preprocessing - Calculate metrics

- Mitochondrial genes – genes that come from mitochondria, not nucleus. Not important for the analysis
- High proportions indicate poor-quality cells, hence should be filtered out
- First, annotate mitochondrial genes (their names start with “MT-”):

  adata.var["mt"] = adata.var_names.str.startswith("MT-") – returns Boolean values

Preprocessing - Calculate metrics

- Calculate metrics for each cell separately:

  sc.pp.calculate_qc_metrics(adata, qc_vars=["mt"], percent_top=None, log1p=False, inplace=True) – calculates standart metrics and percent of mitochondrial genes in place (results are stored in the same anndata object in obs)

  - *n_genes_by_count* – number of different genes expressed (have non-zero value) in a cell
  - *total_counts* – total sum of gene expression counts per cell
  - *pct_count_mt* – percentage of counts derived from mitochondrial genes in a cell relative to the total counts in the cell
  - *and more…*

- Visualize metrics with violin plot:

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

Preprocessing - Filter cells by metrics

- Visualize metrics:

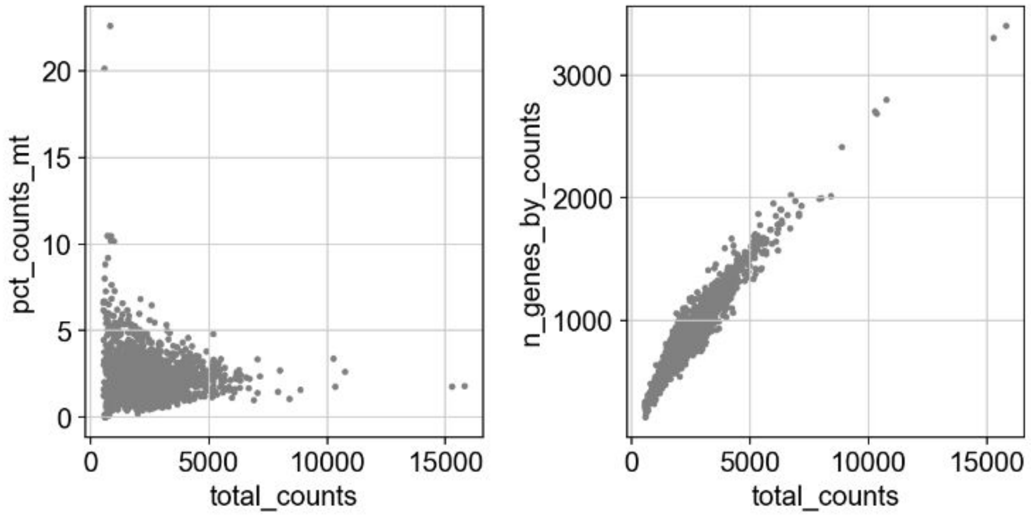
  sc.pl.scatter(adata, x="total_counts", y="pct_counts_mt")

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

- Cut outliers – cells with high percent of mt genes and cells with high number of expressed genes – it may indicate potential doublets (two cells captured together) or other technical errors

  adata = adata[adata.obs.n_genes_by_counts < 2500, :]

  adata = adata[adata.obs.pct_counts_mt < 5, :].copy()



Preprocessing - Normalization

- Total count normalization - normalize each cell's gene expression counts by its total count and scale to a common total count across all cells (usually 10000)
- This ensures that each cell contributes equally to downstream analyses regardless of its size

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

$$normalized\_expr = \frac{expression}{total\_counts} * 10000$$

Normalisation - example

- Ex. There are 2 cells each represented with 3 genes:
  - Cell_1 gene expression counts: 2, 3, 5
  - Cell_2 gene expression counts: 1000, 1500, 2500
- These gene expressions follow same proportions in both cells - biologically they should represent the same type of cell, but no computer algorithm can see these cells as similar due to the great difference in their expression counts. That's when normalisation shows useful
- After normalisation cells look like:
  - Cell_1: 2000, 3000, 5000
  - Cell_2: 2000, 3000, 5000
- Now the similarity between these cells is evident both to humans and computers

Preprocessing – Log transformation

- It's common to perform a log transformation on gene expression counts before conducting downstream analyses

  sc.pp.log1p()

  $log\_expr = log(1 + expression)$

- Many genes having low expression counts and a few genes having very high counts
- Log transformation reduces the dynamic range of expression values, making it easier to visualize and analyze the data
- Stabilizes variance – important for dimensionality reduction and clustering
- Improves the performance of statistical tests and machine learning algorithms
- Natural logarithm (base e)
- Adds constant 1 helps avoid taking a logarithm of zero (undefined)

Preprocessing – Highly variable genes

- Highly variable genes (HVGs) – genes that exhibit significant variation in expression levels across cells
- Variations due to differences in cell types, developmental stages, environmental conditions etc.
- Identification through statistical analysis of expression variability
- Biological significance – hvgs play key roles in cellular processes, cell fate determination, differentiation, response to stimuli and disease mechanisms
- In scRNA-seq data analysis identifying and focusing on highly variable genes is important for dimensionality reduction, clustering, visualization, and downstream analyses

  sc.pp.highly_variable_genes(adata, min_mean=0.0125, max_mean=3, min_disp=0.5) – parameters optional

  sc.pl.highly_variable_genes(adata) – visualization of hvgs' dispersion

Preprocessing – Regressing out and scaling

- Regression-based normalization by removing the effects of unwanted sources (technical sources) of variation that are not of biological interest

  sc.pp.regress_out(adata, ["total_counts", "pct_counts_mt"]) – these variables are often considered as technical covariates or batch effects that can confound downstream analyses if not properly addressed

- Scaling transformation performs z-score normalization

  sc.pp.scale(adata, max_value=10) – max value optional

  $x_i' = \frac{x_i - \mu}{\sigma}$ x_i is the expression value of the feature in the i-th cell, μ is the mean expression value of the feature across all cells, σ is the standard deviation of the expression values of the feature across all cells

- This process centers the distribution of expression values around zero with a standard deviation of one

- *regress_out* and *scale* methods change sparse matrix shape to dense matrix

Dimensionality reduction

- Dimensionality reduction is a process used in data analysis and machine learning to reduce the number of features or variables in a dataset while preserving its important underlying structure or patterns
- It involves transforming high-dimensional data into a lower-dimensional representation that retains as much relevant information as possible
- Several benefits: visualization, computational efficiency

PCA

- Principal Component Analysis (PCA) - linear dimensionality reduction technique that transforms the data into a new coordinate system, where each dimension (principal component) captures the maximum variance in the data

  sc.tl.pca(adata, svd_solver="arpack") – *n_comps* parameter determines number of components (new features) to be calculated (default 50); by default it uses only HVGs if they have been determined, to use all genes set parameter use_highly_variable = False; adds resulting values *'X_pca'* to *.obsm*

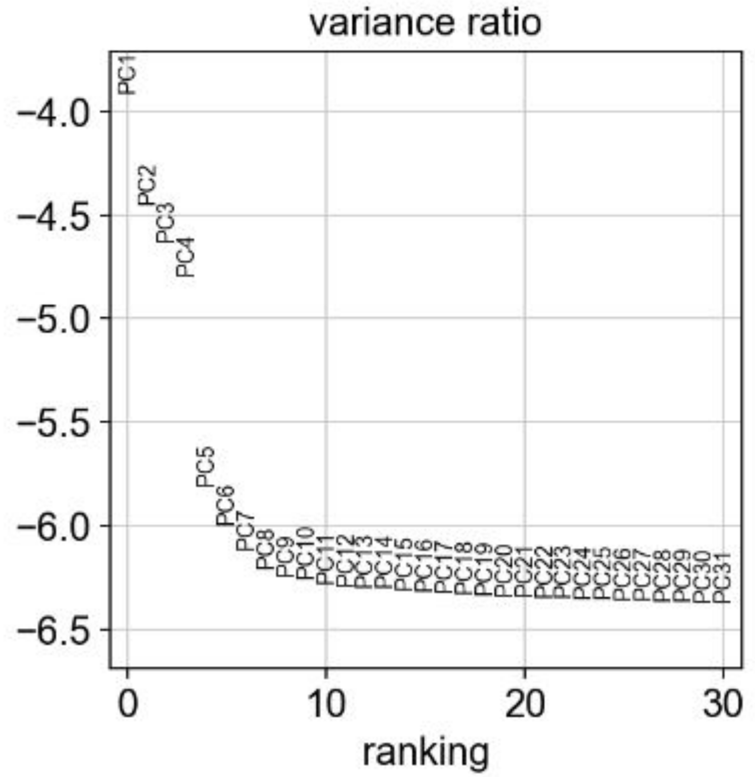
  sc.pl.pca(adata) – visualize

PCA

- Inspect the contribution of single PCs to the total variance in the data in order to determine how many PCs to consider (used in clustering algorithms or tSNE)

  sc.pl.pca_variance_ratio(adata, log=True) - provides a graphical representation of the proportion of variance contributed by each principal component

- Scree plot – common approach in deciding how many PCs to retain, shows eigenvalues of each PC in decreasing order
- The number of principal components to retain can be chosen based on where the scree plot begins to level off
- Usually a rough estimate of the number of PCs is enough



Neighbourhood graph

- Calculate the neighbourhood graph:

  sc.pp.neighbors(adata, n_neighbors=10, n_pcs=40) - uses 40 PCs to calculate, *n_neighbors* specifies the number of nearest neighbors to consider when constructing the graph

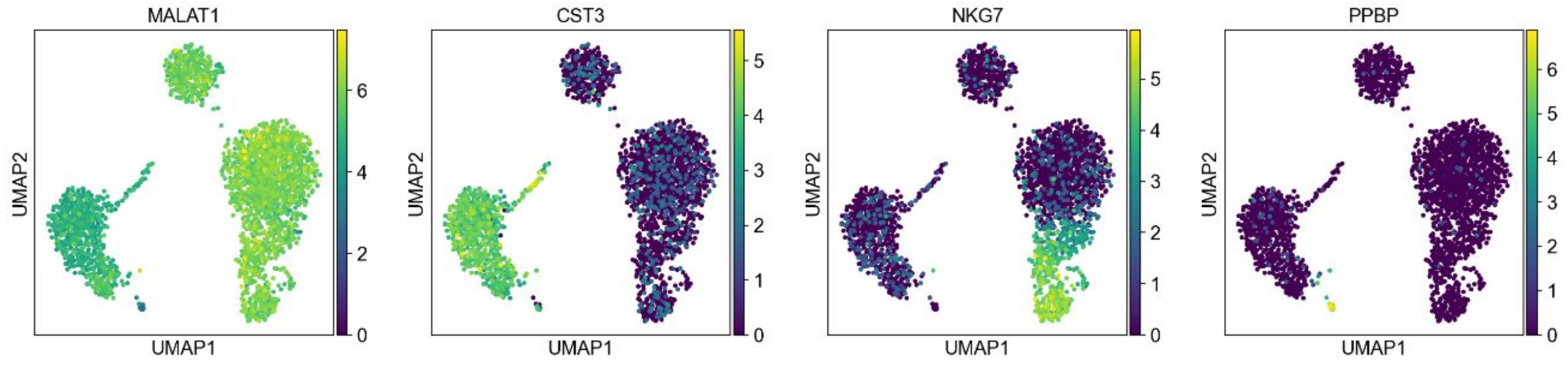
- Represents the pairwise similarities or distances between cells in the data set
- Typically uses Euclidean distance or Cosine similarity as metric
- Fundamental data structure for some dimensionality reduction techniques (UMAP, tSNE) and clustering algorithms (Louvain, Leiden)
- As a result adds *‘distances’* and *‘connectivities’* to *.obsp* and *‘neighbors’* to *.uns*

Visualization

- UMAP – Uniform Manifold Approximation and Projection – another dimensionality reduction technique
- Provides embedding (usually 2 dimensions) for visualization of high-dimensional data such as scRNA-seq
- Preserves local and global structure in the data
- Uses neighbourhood graph for calculating

  sc.tl.umap(adata) – adds *'X_umap'* to *.obsm*

  sc.pl.umap(adata, color=["MALAT1", "CST3", "NKG7", "PPBP"]) – visualize specific genes' expressions



Clustering

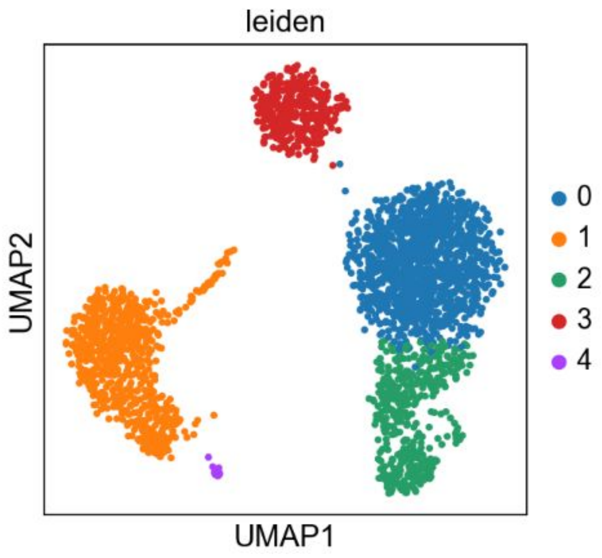
- Grouping similar objects into clusters based on their characteristics or attributes
- Leiden – community detection algorithm based on optimising modularity

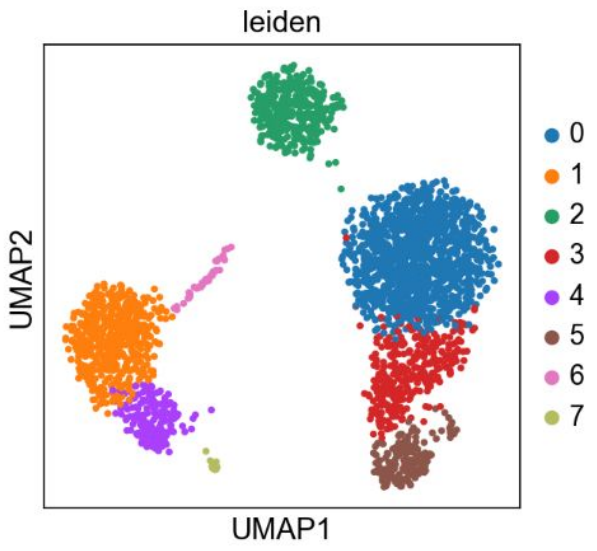
  pip install leidenalg

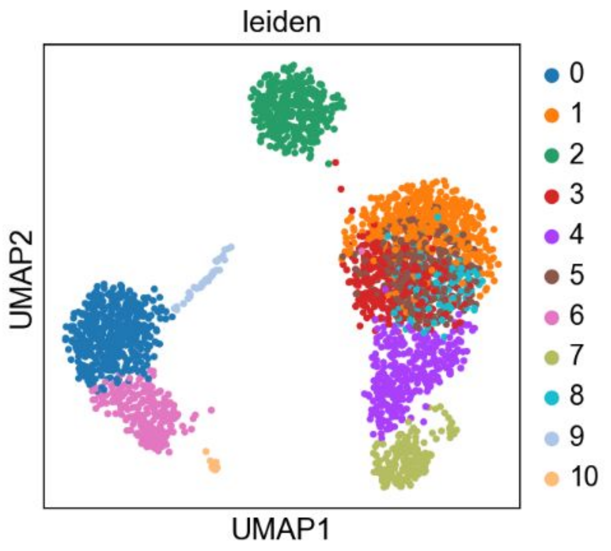
  sc.tl.leiden(adata, resolution=0.9) – resolution optional (default = 1), affects number of clusters in the resulting clusterin (higher value – higher number of clusters); adds *'leiden'* annotations to *.obs*

  sc.pl.umap(adata, color="leiden") – visualize clusters with UMAP

  Ex. Results for resolutions 0.3, 0.9 and 1.3







Marker genes

- Specific genes that are used to identify or characterize a particular cell type (cluster) or condition
- Associated with unique expression patterns or functions that distinguish them from other genes

  sc.tl.rank_genes_groups(adata, "leiden", method="t-test") – finds marker genes for all *'leiden'* clusters; by default uses *.raw* if it exists

  sc.pl.rank_genes_groups(adata, n_genes=25, sharey=False) – plots top 25 marker genes for every cluster
- Other viable methods: *'wilcoxon'* (recommended), *'logreg'*

Cell annotation

- Assigning each cell (or cluster of cells) a cell type
- pip install decoupler
- pip install omnipath

markers = dc.get_resource('PanglaoDB’) - PanglaoDB, a database of cell type markers, part of Omnipath

- Next filter to leave just human specific genes and remove doubles

```
markers = markers[markers['human'] & markers['canonical_marker'] &
(markers['human_sensitivity'] > 0.5)]
```

```
markers = markers[~markers.duplicated(['cell_type', 'genesymbol’])]
```

- Results are stored in .obsm[‘ora_estimate’]

Cell annotation

- Remove infinite values and set them to max value

acts = dc.get_acts(adata, obsm_key='ora_estimate’) – new anndata object with ora values

acts_v = acts.X.ravel()

max_e = np.nanmax(acts_v[np.isfinite(acts_v)])

acts.X[~np.isfinite(acts.X)] = max_e

df = dc.rank_sources_groups(acts, groupby='leiden', reference='rest', method='t-test_overestim_var’)-

uses ‘leiden’ clusters to identify “marker” cell types per cluster using statistical test

ctypes_dict = df.groupby('group').head(num_types).groupby('group')['names'].apply(lambda x: list(x)).to_dict() – forms a dictionary of top num_types predicted types for every cluster

Exercise 3 (bonus)

- Analyze the entire GTEX-1HSMQ-5005 dataset, including all batches as shown in this lecture
- Cluster the data and calculate the Davies-Bouldin score for the clusters you obtain in the UMAP plain
- Then, repeat the analysis and clustering process, but this time, skip the normalization step
- Calculate the Davies-Bouldin score again without normalization
- Finally, compare the scores you obtained from the two different approaches and provide conclusions.
- Upload the jupyter notebook with executed cells to the public Github repository and send a link to it to pedjao@etf.rs before Thursday, 9th of May, 23:59.

- 10 extra points

Contact

- Lazar Smiljkovic
- sl225023p@student.etf.bg.ac.rs