episcanpy Documentation

Release 0.1.6+78.gd8fb6a9

Anna Danese

Nov 05, 2019

CONTENTS

EpiScanpy is a toolkit to analyse single-cell open chromatin (scATAC-seq) and single-cell DNA methylation (for example scBS-seq) data. **EpiScanpy** is the epigenomic extension of the very popular scRNA-seq analysis tool **Scanpy** (Genome Biology, 2018) [Wolf18]. For more information, read scanpy documentation.

The documentation for epiScanpy is available here. If epiScanpy is useful to your research, consider citing epiScanpy.

Report issues and access the code on GitHub.

Note: Also see the release notes of `scanpy`.

Also see the release notes of `anndata`.

CHAPTER

ONE

VERSION 0.1.0 MAY 10, 2019

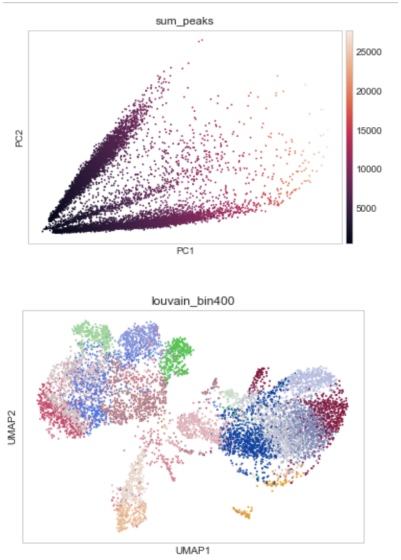
Initial release.

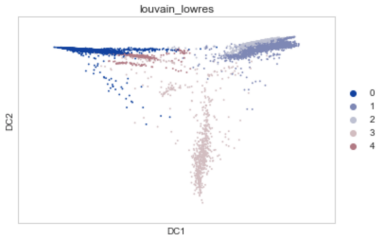
1.1 Tutorials

1.1.1 Single cell ATAC-seq

To get started, we recommend epiScanpy's analysis pipeline for scATAC-seq data for 10k PBMCs from 10x Genomics. This tutorial focuses on preprocessing, clustering, identification of cell types via known marker genes and trajectory inference. The tutorial can be found here.

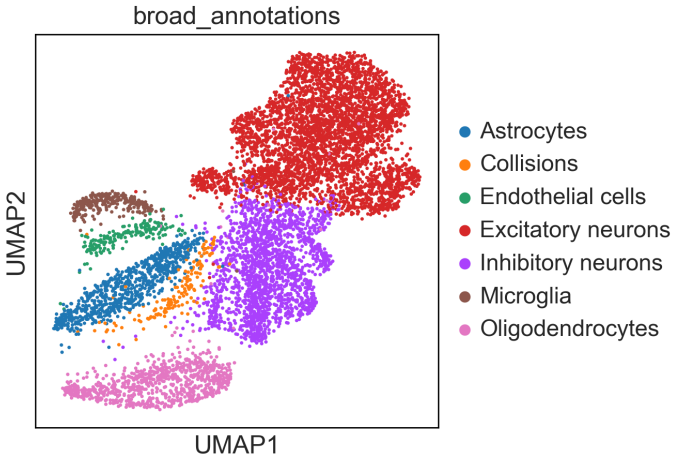
NB: The current tutorial is a beta version that do not include either optimal low embedding & clustering settings or proper cell type identifications. To check out performance see other tutorials below and an updated version will be available very soon.

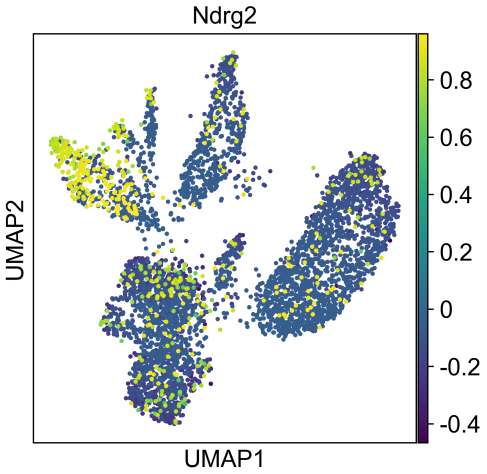


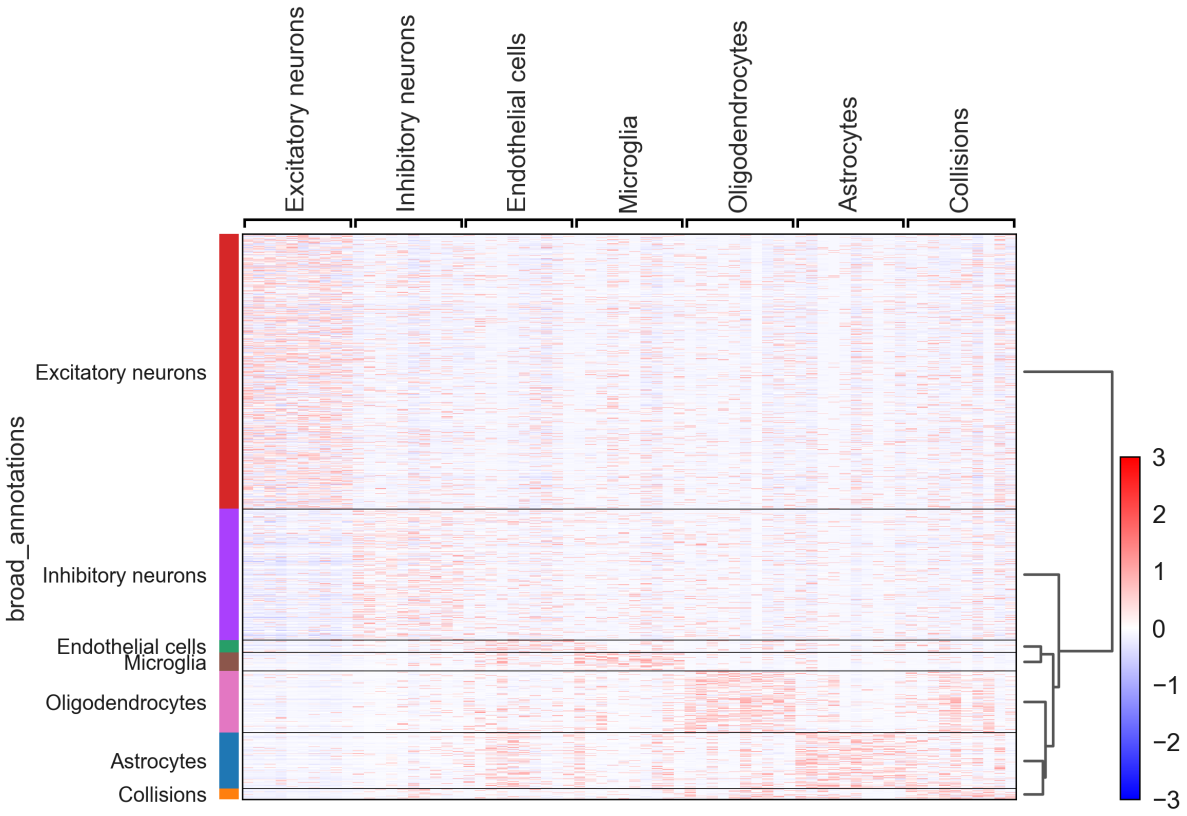


If you want to see how to build count matrices from ATAC-seq bam files for different set of annotations (like enhancers). The tutorial can be found here.

Soon available, there will be a tutorial on preprocessing, clustering and identification of cell types for the Cusanovich mouse scATAC-seq atlas [Cusanovich18] (prefrontal cortex data). In this tutorial we focus on the use of different feature space count matrices (peak and enhancer based count matrices).







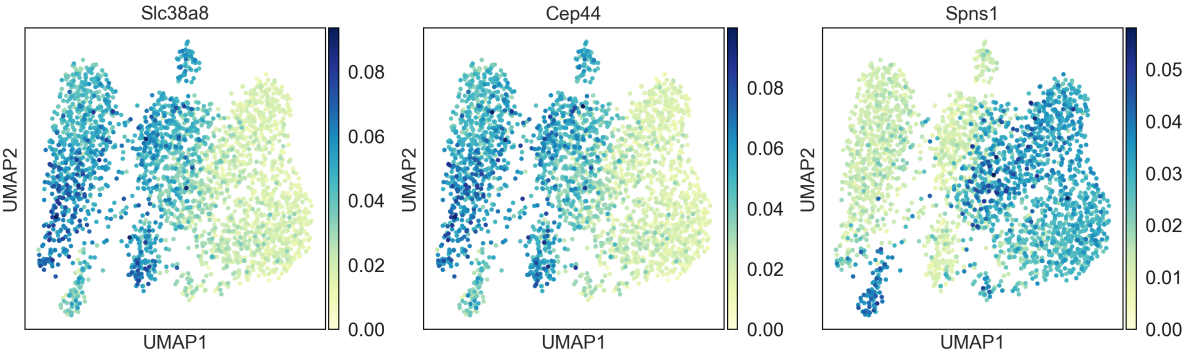
An additional tutorial on processing and clustering count matrices from the Cusanovich atlas. Here.

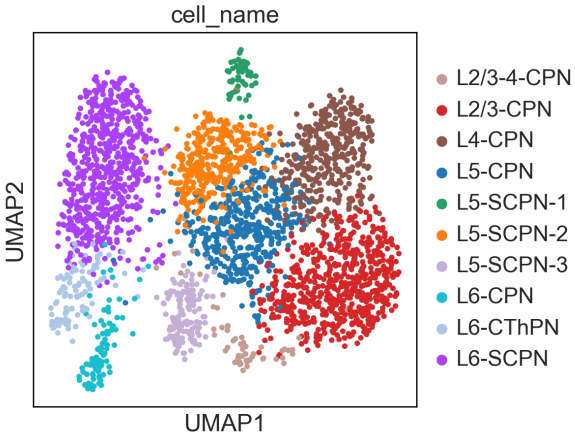
1.1.2 Single cell DNA methylation

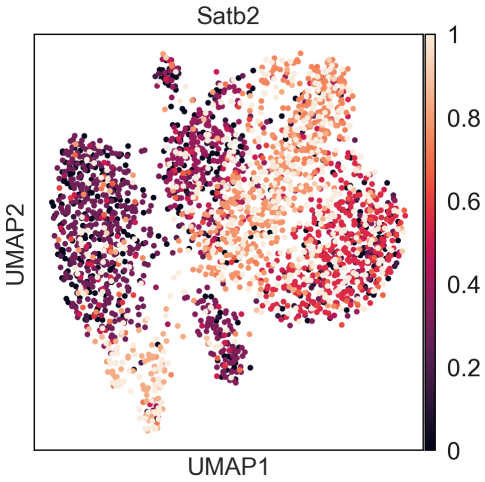
Here you can find a tutorial for the preprocessing, clustering and identification of cell types for single-cell DNA methylation data using the publicly available data from Luo et al. [Luo17].

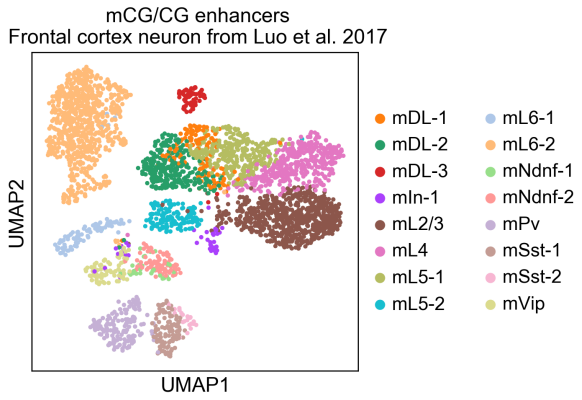
The first tutorial shows how to build the count matrices for the different feature spaces (windows, promoters) in different cytosine contexts. Here is the tutorial.

Then, there is a second tutorial on how to use them and compare the results. The data used comes from mouse brain (frontal cortex). It will be available very soon.









1.2 Usage Principles

Import the epiScanpy API as:

```
import episcanpy.api as epi
import anndata as ad
```

1.2.1 Workflow

The first step is to build the count matrix. Because single-cell epigenomic data types have different characteristics (count data in ATAC-seq versus methylation level in DNA methylation, for example), epiScanpy implements -omic specific approaches to build the count matrix. All the functions to build the count matrices (for ATAC, methylation or other) will use `epi.ct` (ct = count).

The first step is to load an annotation and then build the count matrix that will be either methylation or ATAC-seq specific. For example using `epi.ct`, e.g.:

```
epi.ct.load_features(file_features, **tool_params)  # to load annotation files
epi.ct.build_count_mtx(cell_file_names, omic="ATAC") # to build the ATAC-seq count␣
↪matrix
```

If you have an already build matrix, you can load it with any additional metadata (such as cell annotations or batches).

The count matrix, either the one that has been constructed or uploaded, with any additional informations (such as cell annotations or batches) are stored as an `AnnData` object. All functions for quality control and preprocessing are called using `epi.pp` (pp = preprocessing).

To visualise how common features are and what is the coverage distribution of the count matrix features, use:

```
epi.pp.commoness_features(adata, **processing_params)
epi.pp.coverage_cells(adata, **processing_params)
```

The next step, is the calculation of tSNE, UMAP, PCA etc. For that, we take advantage of the embedding into Scanpy and we use mostly Scanpy functions, which are called using `sc.tl` (tl = tool) [Wolf18]. For that, see Scanpy usage principles: <https://scanpy.readthedocs.io/en/latest/basic_usage.html>‘__. For example, to obtain cell-cell distance calculations or low dimensional representation we make use of the `adata` object, and store *n_obs* observations (cells) of *n_vars* variables (expression, methylation, chromatin features). For each tool, there typically is an associated plotting function in `sc.tl` and `sc.pl` (pl = plot)

```
sc.tl.tsne(adata, **tool_params)
sc.pl.tsne(adata, **plotting_params)
```

There are also epiScanpy specific tools and plotting functions that can be accessed using `epi.tl` and `epi.pl`

```
epi.tl.silhouette(adata, **tool_params)
epi.pl.silhouette(adata, **plotting_params)
epi.pl.prct_overlap(adata, **plotting_params)
```

1.2.2 Data structure

Similarly to Scanpy, the methylation and ATAC-seq matrices are stored as anndata object. For more information on the datastructure see here‘here <https://anndata.readthedocs.io/en/latest/>‘__

1.3 System Requirements

1.3.1 Hardware requirements

`epiScanpy` package requires only a standard computer with enough RAM to support the in-memory operations.

1.3.2 Software requirements

OS Requirements This package is supported for *macOS* and *Linux*. The package has been tested on the following systems: + macOS: Mojave (10.14.1)

1.3.3 Python Dependencies

EpiScanpy require a working version of `Python` (>= 3.5)

Additionally, this package `epiScanpy` depends on other Python dependencies and packages.:

```
anndata
matplotlib
numpy
pandas
pyliftOver
pysam
scanpy
scipy
scikit-learn
seaborn
```

1.4 Installation

1.4.1 Anaconda

If you do not have a working Python 3.5 or 3.6 installation, consider installing Miniconda (see Installing Miniconda). Then run:

```
conda install seaborn scikit-learn statsmodels numba
conda install -c conda-forge python-igraph louvain
conda create -n scanpy python=3.6 scanpy
```

Finally, run:

```
conda install -c annadanese episcanpy
```

Pull epiScanpy from PyPI (consider using `pip3` to access Python 3):

```
pip install episcanpy
```

1.4.2 Github

you can also install epiScanpy directly from Github:

```
pip install git+https://github.com/colomemaria/epiScanpy
```

1.5 API

Import epiScanpy's high-level API as:

```
import episcanpy.api as epi
```

1.5.1 Count Matrices: CT

Loading data, loading annotations, building count matrices, filtering of lowly covered methylation variables. Filtering of lowly covered cells.

Load features

In order to build a count matrix for either methylation or open chromatin data, loading the segmentation of the genome of interest or the set of features of interest is a prerequirement.

ct.load_features(file_features[, ...])	The function load features is here to transform a bed file into a usable set of units to measure methylation levels.
ct.make_windows(size[, chromosomes, max_length])	Generate windows/bins of the given size for the appropriate genome (default choice is human).
ct.size_feature_norm(loaded_feature, size)	If the features loaded are too smalls or of different sizes, it is possible to normalise them to a unique given size by extending the feature coordinate in both directions.
ct.plot_size_features(loaded_feature[, ...])	Plot the different feature sizes in an histogram.
ct.name_features(loaded_features)	Extract the names of the loaded features, specifying the chromosome they originated from.

episcanpy.ct.load_features

episcanpy.ct.**load_features**(*file_features, chromosomes=['1', '10', '11', '12', '13', '14', '15', '16', '17', '18', '19', '2', '20', '21', '22', '3', '4', '5', '6', '7', '8', '9', 'X', 'Y'], path='', sort=False*)

The function load features is here to transform a bed file into a usable set of units to measure methylation levels. It has to be a bed-like file. You also need to specify the chromosomes you use as a list of characteres like ['1', '7', '8', '9', 'X', 'Y', 'M']. The chromosome list you give as input can be not ordered. If you don't specify the chromosomes, the default is the human genome (including X, Y and mitochondrial DNA). THE BED FILE need to be sorted !! (Maybe I should add an option to sort the file).

The output is a dictionary where the keys are chromosomes and the value is a list containing [start, end, name] for every feature extracted.

Parameters

file_features the names of the bed file you want to load.

chromosomes chromosomes corresponding to the bed file. If not specified, it's human by default

path if you want to specify the path where your bed file is.

sort if True, the bed file is sorted based on starting coordinates.

episcanpy.ct.make_windows

episcanpy.ct.**make_windows**(*size, chromosomes=['1', '10', '11', '12', '13', '14', '15', '16', '17', '18', '19', '2', '20', '21', '22', '3', '4', '5', '6', '7', '8', '9', 'X', 'Y'], max_length=1000000000*)

Generate windows/bins of the given size for the appropriate genome (default choice is human).

Parameters

size size of the window you want

chromosomes Chromosomes of the species you are analysing

max_length maximum length given for every chromosome

episcanpy.ct.size_feature_norm

episcanpy.ct.**size_feature_norm**(*loaded_feature*, *size*)

If the features loaded are too smalls or of different sizes, it is possible to normalise them to a unique given size by extending the feature coordinate in both directions.

Parameters

loaded_feature : ***loaded feature to normalise***

size : ***desired size of the feature***

Returns Update the input features

episcanpy.ct.plot_size_features

episcanpy.ct.**plot_size_features**(*loaded_feature*, *bins=50*, *return_length=False*)

Plot the different feature sizes in an histogram.

episcanpy.ct.name_features

episcanpy.ct.**name_features**(*loaded_features*)

Extract the names of the loaded features, specifying the chromosome they originated from. It also contain the feature coordinates and an unique identifier.

Reading methylation file

Functions to read methylation files, extract methylation and buildthe count matrices:

ct.build_count_mtx(cells, annotation[, ...])	Build methylation count matrix for a given annotation.
ct.read_cyt_summary(sample_name, meth_type, ...)	Read file from which you want to extract the methylation level and (assuming it is like the Ecker/Methylpy format) extract the number of methylated read and the total number of read for the cytosines covered and in the right genomic context (CG or CH) :param sample_name: name of the file to read to extract key information.

Continued on next page

Table 2 – continued from previous page

ct.load_met_noimput(matrix_file[, path, save])	read the raw count matrix and convert it into an AnnData object.

episcanpy.ct.build_count_mtx

episcanpy.ct.**build_count_mtx**(*cells, annotation, path='', output_file=None, writing_option='a', meth_context='CG', chromosome=['1', '10', '11', '12', '13', '14', '15', '16', '17', '18', '19', '2', '20', '21', '22', '3', '4', '5', '6', '7', '8', '9', 'X', 'Y'], feature_names=None, threshold=1, ct_mtx=None, sparse=False*)

Build methylation count matrix for a given annotation. It either write the count matrix (if given an output file) or return it as a variable (numpy matrix). If you want to add cells to an already existing matrix (with the same annotations), you put the initial matrix as ct_mtx or you specify the matrix to write + writing option = a

if you want to write down the matrix as a sparse matrix you have to specify it (not implented yet)

I need to pay attention to where I am writing the output file.

Also, verbosity..

Pay attention, it does not average variables. If you want to process many small features such as tfbs, we advise to use the dedicated function.

Parameters

cells list of the file names to read to build the count matrix.

annotation loaded annotation to use to build the count matrix 'str' or 'list' depending of the number of matrices to build

path path to the input data.

output_file name files to write. 'str' or 'list' depending of the number of matrices to build

writing_option either 'w' if you want to erase potentialy already existing file or 'a' to append. 'str' or 'list' if you have a list of matrices and the writing options are differents

meth_context read methylation in 'CG' of 'CH' context

chromosome 'MOUSE' and 'HUMAN' (without mitochondrial genome) or list with chromosomes.

feature_names If you want to write down the name of the annotation features. 'Int' (or 'list' if you have multiple annotations)

thereshold the minimum of cytosines covered per annotation to calculate a methylation level. default=1 'Int' (or 'list' if you have multiple annotations with different thresholds)

ct_mtx numpy matrix containing the same set of annotations and for which you want to append. default: None

sparse Boolean, writing option as a normal or sparse matrix. default: False

episcanpy.ct.read_cyt_summary

episcanpy.ct.**read_cyt_summary**(*sample_name*, *meth_type*, *head*, *path*, *chromosome*)

Read file from which you want to extract the methylation level and (assuming it is like the Ecker/Methylpy format) extract the number of methylated read and the total number of read for the cytosines covered and in

the right genomic context (CG or CH) :param sample_name: name of the file to read to extract key information. :param meth_type: CG, CH or not specified :param head: file you are reading. The default value is the Methylpy/Ecker header :type head: if there is header that you don't want to read. An annotation in the :param path: :type path: path of the to access the file of the sample you want to read. :param chromosome: is the human genome (including mitochondrial and sexual chromosomes) :type chromosome: chromosomes if the species you are considering. default value

episcanpy.ct.load_met_noimput

episcanpy.ct.**load_met_noimput**(*matrix_file*, *path=''*, *save=False*)

read the raw count matrix and convert it into an AnnData object. write down the matrix as .h5ad (AnnData object) if save = True. Return AnnData object

Reading open chromatin(ATAC) file

ATAC-seq specific functions to build count matrices and load data:

ct.bld_atac_mtx(list_bam_files, loaded_feat)	Build a count matrix one set of features at a time.
ct.save_sparse_mtx(initial_matrix[, ...])	Convert regular atac matrix into a sparse Anndata:

episcanpy.ct.bld_atac_mtx

episcanpy.ct.**bld_atac_mtx**(*list_bam_files, loaded_feat, output_file_name=None, path=None, writing_option='a', header=None, mode='rb', check_sq=True, chromosomes=['1', '10', '11', '12', '13', '14', '15', '16', '17', '18', '19', '20', '21', '22', '2', '3', '4', '5', '6', '7', '8', '9', 'X', 'Y']*)

Build a count matrix one set of features at a time. It is specific of ATAC-seq data. It curently do not write down a sparse matrix. It writes down a regular count matrix as a text file.

Parameters

list_bam_files : ***input must be a list of bam file names. One for each cell to***
build the count matrix for

loaded_feat : ***the features for which you want to build the count matrix***

output_file_name : ***name of the output file. The count matrix that will be written***
down in the current directory. If this parameter is not specified, the output count amtrix will be named 'std_output_ct_mtx.txt'

path : ***path where to find the input file. The output file will be written down***

your current directory, it is not affected by this parameter. : ***in***

writing_option : ***standard writing options for the output file. 'a' or 'w'***
'a' to append to an already existing matrix. 'w' to overwrite any previously exisiting matrix. default: 'a'

header : ***if you want to write down the feature name specify this argument.***
Input must be a list.

mode : ***bamnostic argument 'r' or 'w' for read and write 'b' and 's' for bam or sa***
if only 'r' is specified, bamnostic will try to determine if the input is either a bam or sam file.

check_sq : ***bamnostic argument. when reading, check if SQ entries are present in h***

chromosomes : ***chromosomes of the species you are considering. default value***

is the human genome (not including mitochondrial genome). HUMAN = ['1', '10', '11', '12', '13', '14', '15', '16', '17', '18', '19', '20', '21', '22',

'2', '3', '4', '5', '6', '7', '8', '9','X', 'Y']

MOUSE = '1', '10', '11', '12', '13', '14', '15', '16', '17', '18', '19', '2', '3', '4', '5', '6', '7', '8', '9','X', 'Y']

Returns It does not return any object. The function write down the desired count matrix in a txt file

episcanpy.ct.save_sparse_mtx

episcanpy.ct.**save_sparse_mtx**(*initial_matrix*, *output_file='.h5ad'*, *path=''*, *omic='ATAC'*, *bed=False*, *save=True*)

Convert regular atac matrix into a sparse Anndata:

Parameters

initial_matrix : ***initial dense count matrix to load and convert into a sparse mat***

output_file : ***name of the output file for the AnnData object.***

output is the name of the input file with .h5ad extension : ***Default***

path : ***path to the input count matrix. The AnnData object is written in the curre***
not the location specified in path.

omic : ***'ATAC', 'RNA' or 'methylation' are the 3 currently recognised omics in epi***
However, other omic name can be accepted but are not yet recognised in other functions.
default: 'ATAC'

bed : ***boolean. If True it consider another input format (bedtools output format f***

save : ***boolean. If True, the sparse matrix is saved as h5ad file. Otherwise it is***

Returns It returns the loaded matrix as an AnnData object.

General functions

Functions non -omic specific:

ct.save_sparse_mtx(initial_matrix[, ...])	Convert regular atac matrix into a sparse Anndata:

1.5.2 Preprocessing: PP

Imputing missing data (methylation), filtering lowly covered cells or variables, correction for batch effect.

pp.coverage_cells(adata[, bins, key_added, ...])	Histogram of the number of open features (in the case of ATAC-seq data) per cell.
pp.commoness_features	
pp.select_var_feature(adata[, max_score, ...])	This function computes a variability score to rank the most variable features across all cells.
pp.binarize(adata[, copy])	convert the count matrix into a binary matrix.
pp.lazy(adata[, pp_pca, nb_pcs, ...])	Automatically computes PCA coordinates, loadings and variance decomposition, a neighborhood graph of observations, t-distributed stochastic neighborhood embedding (tSNE) Uniform Manifold Approximation and Projection (UMAP)
pp.load_metadata(adata, metadata_file[, ...])	Load observational metadata in adata.obs.
pp.read_ATAC_10x(matrix[, cell_names, ...])	Load sparse matrix (including matrices corresponding to 10x data) as AnnData objects.
pp.filter_cells(adata[, min_counts, ...])	Filter cell outliers based on counts and numbers of genes expressed.
pp.filter_features(data[, min_counts, ...])	Filter features based on number of cells or counts.
pp.normalize_total(adata[, target_sum, ...])	Normalize counts per cell.
pp.pca(adata[, n_comps, zero_center, ...])	Principal component analysis [Pedregosa11].
pp.normalize_per_cell(adata[, ...])	Normalize total counts per cell.
pp.regress_out(adata, keys[, n_jobs, copy])	Regress out unwanted sources of variation.
pp.subsample(data[, fraction, n_obs, ...])	Subsample to a fraction of the number of observations.
pp.downsample_counts(adata[, ...])	Downsample counts from count matrix.
pp.neighbors(adata[, n_neighbors, n_pcs, ...])	Compute a neighborhood graph of observations [McInnes18].
pp.sparse(adata[, copy])	Transform adata.X from a matrix or array to a csc sparse matrix.

episcanpy.pp.coverage_cells

episcanpy.pp.**coverage_cells**(*adata*, *bins=50*, *key_added=None*, *log=False*, *binary=None*, *xlabel=None*, *ylabel=None*, *title=None*, *color=None*, *edgecolor=None*, *save=None*)

Histogram of the number of open features (in the case of ATAC-seq data) per cell.

episcanpy.pp.select_var_feature

episcanpy.pp.**select_var_feature**(*adata*, *max_score=0.5*, *nb_features=None*, *show=True*, *copy=False*)

This function computes a variability score to rank the most variable features across all cells. Then it selects the most variable features according to either a specified number of features (nb_features) or a minimum variance score (min_score).

Parameters

adata : *adata object*

max_score : *max threshold of the variability score to retain features,*

0 is the score of the most variable features and 0.5 is the score of the least v

nb_features : ***default value is None, if specify it will select a the top most var***

the nb_features is larger than the total number of feature, it filters based on

show : ***default value True, it will plot the distribution of var.***

copy : ***return a new adata object if copy == True.***

Returns Depending on `copy`, returns a new AnnData object or overwrite the input

episcanpy.pp.binarize

episcanpy.pp.**binarize**(*adata*, *copy=False*)

convert the count matrix into a binary matrix.

Parameters

adata : ***AnnData object***

copy : ***return a new adata object if copy == True.***

Returns Depending on `copy`, returns a new AnnData object or overwrite the input

episcanpy.pp.lazy

episcanpy.pp.**lazy**(*adata*, *pp_pca=True*, *nb_pcs=50*, *n_neighbors=15*, *perplexity=30*, *method='umap'*, *metric='euclidean'*, *min_dist=0.5*, *spread=1.0*, *n_components=2*, *copy=False*)

Automatically computes PCA coordinates, loadings and variance decomposition, a neighborhood graph of observations, t-distributed stochastic neighborhood embedding (tSNE) Uniform Manifold Approximation and Projection (UMAP)

Parameters

adata : **AnnData** Annotated data matrix.

pp_pca : ***bool*** **(default:** ***True*****)** Computes PCA coordinates before the neighborhood graph

nb_pcs : ***int*** **(default: 50)** Number of principal component computed for PCA (and therefore neighbors, tsne and umap)

n_neighbors : ***int*** **(default: 15)** The size of local neighborhood (in terms of number of neighboring data points) used for manifold approximation. Larger values result in more global views of the manifold, while smaller values result in more local data being preserved. In general values should be in the range 2 to 100.

method : ***str*** **(default: 'umap')** Use 'umap' or 'gauss', kernel for computing connectivities. Gives very similar results.

metric : ***str*** **(default: 'euclidean')** A known metric's name or a callable that returns a distance.

perplexity : ***int*** **(default: 30)** The perplexity is related to the number of nearest neighbors that is used in other manifold learning algorithms. Larger datasets usually require a larger perplexity. Consider selecting a value between 5 and 50.

min_dist : *float* (default: 0.5) The effective minimum distance between embedded points. Smaller values will result in a more clustered/clumped embedding where nearby points on the manifold are drawn closer together, while larger values will result on a more even dispersal of points.

spread : *float* (default: 1.0) The effective scale of embedded points. In combination with *min_dist* this determines how clustered/clumped the embedded points are.

n_components : *int* (default: 2) The number of dimensions of the *UMAP* embedding.

copy : *bool* (default: *False*) Return a copy instead of writing to adata.

method='umap',

metric='euclidean',

min_dist=0.5,

spread=1.0,

n_components=2

Returns

Depending on *copy*, returns or updates *adata* with the following fields. X_pca : *adata.obsm*

PCA coordinates of data.

connectivities [sparse matrix (*.uns['neighbors']*, dtype *float32*)] Weighted adjacency matrix of the neighborhood graph of data points. Weights should be interpreted as connectivities.

distances [sparse matrix (*.uns['neighbors']*, dtype *float32*)] Instead of decaying weights, this stores distances for each pair of neighbors.

X_tsne [*np.ndarray* (*adata.obs*, dtype *float*)] tSNE coordinates of data.

X_umap [*adata.obsm*] UMAP coordinates of data.

episcanpy.pp.load_metadata

episcanpy.pp.**load_metadata**(*adata*, *metadata_file*, *path=''*, *separator=';'*)

Load observational metadata in adata.obs. Input metadata file as csv/txt and the adata object to annotate.

first raw of the metadata file is considered as a header first column contain the cell name

adata: initial AnnData object

metadata_file: csv file containing as a first column the cell names and in the rest of the columns any king of metadata to load

path: pathe to the metadata file

separator: ';' or ' ', character used to split the columns

Returns Annotated AnnData

episcanpy.pp.read_ATAC_10x

episcanpy.pp.**read_ATAC_10x**(*matrix*, *cell_names=''*, *var_names=''*, *path_file=''*)

Load sparse matrix (including matrices corresponding to 10x data) as AnnData objects. read the mtx file, tsv file coresponding to cell_names and the bed file containing the variable names

Parameters

matrix : *sparse count matrix*

cell_names : *optional, tsv file containing cell names*

var_names : *optional, bed file containing the feature names*

Returns AnnData object

episcanpy.pp.filter_cells

episcanpy.pp.**filter_cells**(*adata*, *min_counts=None*, *min_features=None*, *max_counts=None*, *max_features=None*, *inplace=True*, *copy=False*)

Filter cell outliers based on counts and numbers of genes expressed.

For instance, only keep cells with at least *min_counts* counts or *min_features* genes expressed. This is to filter measurement outliers, i.e. "unreliable" observations.

Only provide one of the optional parameters `min_counts`, `min_features`, `max_counts`, `max_features` per call.

Parameters

data The (annotated) data matrix of shape `n_obs` × `n_vars`. Rows correspond to cells and columns to genes.

min_counts Minimum number of counts required for a cell to pass filtering.

min_features Minimum number of genes expressed required for a cell to pass filtering.

max_counts Maximum number of counts required for a cell to pass filtering.

max_features Maximum number of genes expressed required for a cell to pass filtering.

inplace Perform computation inplace or return result.

Returns

Depending on `inplace`, returns the following arrays or directly subsets and annotates the data matrix:

cells_subset [`ndarray`] Boolean index mask that does filtering. `True` means that the cell is kept. `False` means the cell is removed.

number_per_cell [`ndarray`] Depending on what was tresholded (`counts` or `genes`), the array stores `n_counts` or `n_cells` per gene.

Examples

```
>>> adata = sc.datasets.krumsiek11()
>>> adata.n_obs
640
>>> adata.var_names
['Gata2' 'Gata1' 'Fog1' 'EKLF' 'Fli1' 'SCL' 'Cebpa'
 'Pu.1' 'cJun' 'EgrNab' 'Gfi1']
>>> # add some true zeros
>>> adata.X[adata.X < 0.3] = 0
>>> # simply compute the number of genes per cell
>>> sc.pp.filter_cells(adata, min_features=0)
>>> adata.n_obs
640
>>> adata.obs['n_features'].min()
```

(continues on next page)

(continued from previous page)

```
1
>>> # filter manually
>>> adata_copy = adata[adata.obs['n_features'] >= 3]
>>> adata_copy.obs['n_features'].min()
>>> adata.n_obs
554
>>> adata.obs['n_features'].min()
3
>>> # actually do some filtering
>>> sc.pp.filter_cells(adata, min_features=3)
>>> adata.n_obs
554
>>> adata.obs['n_features'].min()
3
```

episcanpy.pp.filter_features

episcanpy.pp.**filter_features**(*data*, *min_counts=None*, *min_cells=None*, *max_counts=None*, *max_cells=None*, *inplace=True*, *copy=False*)

Filter features based on number of cells or counts.

Keep features that have at least `min_counts` counts or are expressed in at least `min_cells` cells or have at most `max_counts` counts or are expressed in at most `max_cells` cells.

Only provide one of the optional parameters `min_counts`, `min_cells`, `max_counts`, `max_cells` per call.

Parameters

data An annotated data matrix of shape *n_obs* × *n_vars*. Rows correspond to cells and columns to genes.

min_counts Minimum number of counts required for a gene to pass filtering.

min_cells Minimum number of cells expressed required for a gene to pass filtering.

max_counts Maximum number of counts required for a gene to pass filtering.

max_cells Maximum number of cells expressed required for a gene to pass filtering.

inplace Perform computation inplace or return result.

Returns

Depending on *inplace*, returns the following arrays or directly subsets and annotates the data matrix

gene_subset [ndarray] Boolean index mask that does filtering. *True* means that the gene is kept. *False* means the gene is removed.

number_per_gene [ndarray] Depending on what was tresholded (*counts* or *cells*), the array stores *n_counts* or *n_cells* per gene.

episcanpy.pp.normalize_total

episcanpy.pp.**normalize_total**(*adata*, *target_sum=None*, *exclude_highly_expressed=False*, *max_fraction=0.05*, *key_added=None*, *layers=None*, *layer_norm=None*, *inplace=True*)

Normalize counts per cell.

If choosing `target_sum=1e6`, this is CPM normalization.

If `exclude_highly_expressed=True`, very highly expressed genes are excluded from the computation of the normalization factor (size factor) for each cell. This is meaningful as these can strongly influence the resulting normalized values for all other genes [Weinreb17].

Similar functions are used, for example, by Seurat [Satija15], Cell Ranger [Zheng17] or SPRING [Weinreb17].

Parameters

adata The annotated data matrix of shape `n_obs` × `n_vars`. Rows correspond to cells and columns to features.

target_sum If `None`, after normalization, each observation (cell) has a total count equal to the median of total counts for observations (cells) before normalization.

exclude_highly_expressed Exclude (very) highly expressed genes for the computation of the normalization factor (size factor) for each cell. A gene is considered highly expressed, if it has more than `max_fraction` of the total counts in at least one cell. The not-excluded genes will sum up to `target_sum`.

max_fraction If `exclude_highly_expressed=True`, consider cells as highly expressed that have more counts than `max_fraction` of the original total counts in at least one cell.

key_added Name of the field in `adata.obs` where the normalization factor is stored.

layers List of layers to normalize. Set to `'all'` to normalize all layers.

layer_norm

Specifies how to normalize layers:

- If *None*, after normalization, for each layer in *layers* each cell has a total count equal to the median of the *counts_per_cell* before normalization of the layer.
- If *'after'*, for each layer in *layers* each cell has a total count equal to *target_sum*.
- If *'X'*, for each layer in *layers* each cell has a total count equal to the median of total counts for observations (cells) of *adata.X* before normalization.

inplace Whether to update `adata` or return dictionary with normalized copies of `adata.X` and `adata.layers`.

Returns Returns dictionary with normalized copies of *adata.X* and *adata.layers* or updates *adata* with normalized version of the original *adata.X* and *adata.layers*, depending on *inplace*.

Example

```
>>> from anndata import AnnData
>>> import scanpy as sc
>>> sc.settings.verbosity = 2
>>> np.set_printoptions(precision=2)
>>> adata = AnnData(np.array([[3, 3, 3, 6, 6], [1, 1, 1, 2, 2], [1, 22, 1, 2,␣
↪2]]))
>>> adata.X
array([[ 3.,  3.,  3.,  6.,  6.],
       [ 1.,  1.,  1.,  2.,  2.],
       [ 1., 22.,  1.,  2.,  2.]], dtype=float32)
>>> X_norm = sc.pp.normalize_total(adata, target_sum=1, inplace=False)['X']
>>> X_norm
```

(continues on next page)

(continued from previous page)

```
array([[0.14, 0.14, 0.14, 0.29, 0.29],
       [0.14, 0.14, 0.14, 0.29, 0.29],
       [0.04, 0.79, 0.04, 0.07, 0.07]], dtype=float32)
>>> X_norm = sc.pp.normalize_total(adata, target_sum=1, exclude_highly_
↪expressed=True, max_fraction=0.2, inplace=False)['X']
The following highly-expressed genes are not considered during normalization␣
↪factor computation:
['1', '3', '4']
>>> X_norm
array([[ 0.5,  0.5,  0.5,  1. ,  1. ],
       [ 0.5,  0.5,  0.5,  1. ,  1. ],
       [ 0.5, 11. ,  0.5,  1. ,  1. ]], dtype=float32)
```

episcanpy.pp.pca

episcanpy.pp.**pca**(*adata*, *n_comps=50*, *zero_center=True*, *svd_solver='auto'*, *random_state=0*, *return_info=False*, *use_highly_variable=False*, *dtype='float32'*, *copy=False*, *chunked=False*, *chunk_size=None*)

Principal component analysis [Pedregosa11].

Computes PCA coordinates, loadings and variance decomposition. Uses the implementation of *scikit-learn* [Pedregosa11].

Parameters

data The (annotated) data matrix of shape `n_obs` × `n_vars`. Rows correspond to cells and columns to genes.

n_comps Number of principal components to compute.

zero_center If *True*, compute standard PCA from covariance matrix. If `False`, omit zero-centering variables (uses `TruncatedSVD`), which allows to handle sparse input efficiently. Passing `None` decides automatically based on sparseness of the data.

svd_solver

SVD solver to use:

'arpack' for the ARPACK wrapper in SciPy (`svds()`)

'randomized' for the randomized algorithm due to Halko (2009).

'auto' (the default) chooses automatically depending on the size of the problem.

random_state Change to use different initial states for the optimization.

return_info Only relevant when not passing an `AnnData`: see "**Returns**".

use_highly_variable Whether to use highly variable genes only, stored in `.var['highly_variable']`. By default uses them if they have been determined beforehand.

dtype Numpy data type string to which to convert the result.

copy If an `AnnData` is passed, determines whether a copy is returned. Is ignored otherwise.

chunked If `True`, perform an incremental PCA on segments of `chunk_size`. The incremental PCA automatically zero centers and ignores settings of `random_seed` and `svd_solver`. If `False`, perform a full PCA.

chunk_size Number of observations to include in each chunk. Required if `chunked=True` was passed.

Returns

X_pca [`scipy.sparse.spmatrix` or `numpy.ndarray`] If *data* is array-like and `return_info=False` was passed, this function only returns *X_pca*...

adata [`AnnData`] ...otherwise if `copy=True` it returns or else adds fields to `adata`:

.obsm['X_pca'] PCA representation of data.

.varm['PCs'] The principal components containing the loadings.

.uns['pca']['variance_ratio']) Ratio of explained variance.

.uns['pca']['variance'] Explained variance, equivalent to the eigenvalues of the covariance matrix.

episcanpy.pp.normalize_per_cell

episcanpy.pp.**normalize_per_cell**(*adata*, *counts_per_cell_after=None*, *counts_per_cell=None*, *key_n_counts=None*, *copy=False*, *layers=[]*, *use_rep=None*, *min_counts=1*)

Normalize total counts per cell.

Warning: Deprecated since version 1.3.7: Use `normalize_total()` instead. The new function is equivalent to the present function, except that

- the new function doesn't filter cells based on *min_counts*, use `filter_cells()` if filtering is needed.
- some arguments were renamed
- *copy* is replaced by *inplace*

Normalize each cell by total counts over all genes, so that every cell has the same total count after normalization.

Similar functions are used, for example, by Seurat [Satija15], Cell Ranger [Zheng17] or SPRING [Weinreb17].

Parameters

data : AnnData, *np.ndarray*, *sp.sparse* The (annotated) data matrix of shape *n_obs* × *n_vars*. Rows correspond to cells and columns to genes.

counts_per_cell_after : *float* or *None*, optional (default: *None*) If *None*, after normalization, each cell has a total count equal to the median of the *counts_per_cell* before normalization.

counts_per_cell : *np.array*, optional (default: *None*) Precomputed counts per cell.

key_n_counts : *str*, optional (default: *'n_counts'*) Name of the field in *adata.obs* where the total counts per cell are stored.

copy : *bool*, optional (default: *False*) If an `AnnData` is passed, determines whether a copy is returned.

min_counts : *int*, optional (default: 1) Cells with counts less than *min_counts* are filtered out during normalization.

Returns Returns or updates *adata* with normalized version of the original *adata.X*, depending on *copy*.

Examples

```
>>> adata = AnnData(
>>>     data=np.array([[1, 0], [3, 0], [5, 6]]))
>>> print(adata.X.sum(axis=1))
[  1.   3.  11.]
>>> sc.pp.normalize_per_cell(adata)
>>> print(adata.obs)
>>> print(adata.X.sum(axis=1))
   n_counts
0       1.0
1       3.0
2      11.0
[ 3.  3.  3.]
>>> sc.pp.normalize_per_cell(adata, counts_per_cell_after=1,
>>>                          key_n_counts='n_counts2')
>>> print(adata.obs)
>>> print(adata.X.sum(axis=1))
   n_counts  n_counts2
0       1.0        3.0
1       3.0        3.0
2      11.0        3.0
[ 1.  1.  1.]
```

episcanpy.pp.regress_out

episcanpy.pp.**regress_out**(*adata*, *keys*, *n_jobs=None*, *copy=False*)

Regress out unwanted sources of variation.

Uses simple linear regression. This is inspired by Seurat's *regressOut* function in R [Satija15].

Parameters

adata : AnnData The annotated data matrix.

keys : *str* or list of *str* Keys for observation annotation on which to regress on.

n_jobs : *int* or *None*, optional. If None is given, then the n_jobs seting is used (default: *None*) Number of jobs for parallel computation.

copy : *bool*, optional (default: *False*) If an AnnData is passed, determines whether a copy is returned.

Returns Depending on *copy* returns or updates *adata* with the corrected data matrix.

episcanpy.pp.subsample

episcanpy.pp.**subsample**(*data*, *fraction=None*, *n_obs=None*, *random_state=0*, *copy=False*)

Subsample to a fraction of the number of observations.

Parameters

data : AnnData, *np.ndarray*, *sp.sparse* The (annotated) data matrix of shape *n_obs* × *n_vars*. Rows correspond to cells and columns to genes.

fraction : *float* in [0, 1] or *None*, optional (default: *None*) Subsample to this *fraction* of the number of observations.

n_obs : *int* or *None*, optional (default: *None*) Subsample to this number of observations.

random_state : *int* or *None*, optional (default: 0) Random seed to change subsampling.

copy : *bool*, optional (default: *False*) If an AnnData is passed, determines whether a copy is returned.

Returns Returns *X[obs_indices], obs_indices* if data is array-like, otherwise subsamples the passed AnnData (*copy == False*) or returns a subsampled copy of it (*copy == True*).

episcanpy.pp.downsample_counts

episcanpy.pp.**downsample_counts**(*adata*, *counts_per_cell=None*, *total_counts=None*, *random_state=0*, *replace=False*, *copy=False*)

Downsample counts from count matrix.

If *counts_per_cell* is specified, each cell will downsampled. If *total_counts* is specified, expression matrix will be downsampled to contain at most *total_counts*.

Parameters

adata Annotated data matrix.

counts_per_cell Target total counts per cell. If a cell has more than 'counts_per_cell', it will be downsampled to this number. Resulting counts can be specified on a per cell basis by passing an array.Should be an integer or integer ndarray with same length as number of obs.

total_counts Target total counts. If the count matrix has more than *total_counts* it will be downsampled to have this number.

random_state Random seed for subsampling.

replace Whether to sample the counts with replacement.

copy If an AnnData is passed, determines whether a copy is returned.

Returns Depending on *copy* returns or updates an *adata* with downsampled *.X*.

episcanpy.pp.neighbors

episcanpy.pp.**neighbors**(*adata*, *n_neighbors=15*, *n_pcs=None*, *use_rep=None*, *knn=True*, *random_state=0*, *method='gauss'*, *metric='euclidean'*, *metric_kwds={}*, *copy=False*)

Compute a neighborhood graph of observations [McInnes18].

The neighbor search efficiency of this heavily relies on UMAP [McInnes18], which also provides a method for estimating connectivities of data points - the connectivity of the manifold (*method=='umap'*). If *method=='gauss'*, connectivities are computed according to [Coifman05], in the adaption of [Haghverdi16].

Parameters

adata Annotated data matrix.

n_neighbors The size of local neighborhood (in terms of number of neighboring data points) used for manifold approximation. Larger values result in more global views of the manifold, while smaller values result in more local data being preserved. In general values should be in the range 2 to 100. If *knn* is *True*, number of nearest neighbors to be searched. If *knn* is *False*, a Gaussian kernel width is set to the distance of the *n_neighbors* neighbor.

{n_pcs}

{use_rep}

knn If *True*, use a hard threshold to restrict the number of neighbors to *n_neighbors*, that is, consider a knn graph. Otherwise, use a Gaussian Kernel to assign low weights to neighbors more distant than the *n_neighbors* nearest neighbor.

random_state A numpy random seed.

method : {{*'umap'*, *'gauss'*, *'rapids'*, *None*}} (default: *'gauss'*) Use 'umap' [McInnes18] or 'gauss' (Gauss kernel following [Coifman05] with adaptive width [Haghverdi16]) for computing connectivities. Use 'rapids' for the RAPIDS implementation of UMAP (experimental, GPU only).

metric A known metric's name or a callable that returns a distance.

metric_kwds Options for the metric.

copy Return a copy instead of writing to adata.

Returns

Depending on *copy*, updates or returns *adata* with the following:

connectivities [sparse matrix (*.uns['neighbors']*, dtype *float32*)] Weighted adjacency matrix of the neighborhood graph of data points. Weights should be interpreted as connectivities.

distances [sparse matrix (*.uns['neighbors']*, dtype *float32*)] Instead of decaying weights, this stores distances for each pair of neighbors.

episcanpy.pp.sparse

episcanpy.pp.**sparse**(*adata*, *copy=False*)

Transform adata.X from a matrix or array to a csc sparse matrix.

Methylation matrices

Methylation specific count matrices.

pp.imputation_met(adata[, ...])	Impute missing values in methyaltion level matrices.
pp.load_met_noimput(matrix_file[, path, save])	read the raw count matrix and convert it into an AnnData object.
pp.readandimputematrix(file_name[, min_coverage])	Temporary function to load and impute methyaltion count matrix into an AnnData object

episcanpy.pp.imputation_met

episcanpy.pp.**imputation_met**(*adata*, *number_cell_covered=10*, *imputation_value='mean'*, *save=None*)

Impute missing values in methyaltion level matrices. The imputsation is based on the average methylation value of the given variable. It also filter out variables that are covered in an unsufficient number of cells in order to reduce the feature space to meaningful variables and discard potential coverage biases.

Parameters

adata : *AnnData object containing 'nan'*

number_cell_covered : *minimum number of cells to be covered in order to retain a*

imputation_value : ***imputation of the missing value can be made either on the mean***

Returns Return a new AnnData object

episcanpy.pp.load_met_noimput

episcanpy.pp.**load_met_noimput**(*matrix_file*, *path=''*, *save=False*)

read the raw count matrix and convert it into an AnnData object. write down the matrix as .h5ad (AnnData object) if save = True. Return AnnData object

episcanpy.pp.readandimputematrix

episcanpy.pp.**readandimputematrix**(*file_name*, *min_coverage=1*)

Temporary function to load and impute methyaltion count matrix into an AnnData object

Parameters

file_name : ***file name to read and load***

min_coverage : ***minimum number of cells covered for which we keep and impute a var***

Returns adata : AnnData Annotated data matrix.

1.5.3 Tools: TL

tl.rank_features(adata, groupby[, omic, ...])	It is a wrap-up function of scanpy sc.tl.rank_genes_groups function.
tl.silhouette(adata_name, cluster_annot[, ...])	Compute silhouette scores.
tl.lazy(adata[, pp_pca, copy])	Automatically computes PCA coordinates, loadings and variance decomposition, a neighborhood graph of observations, t-distributed stochastic neighborhood embedding (tSNE) Uniform Manifold Approximation and Projection (UMAP)
tl.load_markers(path, marker_list_file)	Convert list of known cell type markers from literature to a dictionary Input list of known marker genes First row is considered the header
tl.identify_cluster(adata, cell_type, ...[, ...])	Use markers of a given cell type to plot peak openness for peaks in promoters of the given markers Input cell type, cell type markers, peak promoter intersections
tl.top_feature_genes(adata, gtf_file[, ...])	
tl.find_genes	
tl.diffmap(adata[, n_comps, copy])	Diffusion Maps [Coifman05] [Haghverdi15] [Wolf18].
tl.draw_graph(adata[, layout, init_pos, ...])	Force-directed graph drawing [Islam11] [Jacomy14] [Chippada18].
tl.tsne(adata[, n_pcs, use_rep, perplexity, ...])	t-SNE [Maaten08] [Amir13] [Pedregosa11].
tl.umap(adata[, min_dist, spread, ...])	Embed the neighborhood graph using UMAP [McInnes18].
tl.louvain(adata[, resolution, ...])	Cluster cells into subgroups [Blondel08] [Levine15] [Traag17].
tl.leiden(adata[, resolution, restrict_to, ...])	Cluster cells into subgroups [Traag18].

episcanpy.tl.rank_features

episcanpy.tl.**rank_features**(*adata*, *groupby*, *omic=None*, *use_raw=True*, *groups='all'*, *reference='rest'*, *n_features=100*, *rankby_abs=False*, *key_added='rank_features_groups'*, *copy=False*, *method=''*, *corr_method='benjamini-hochberg'*, ***kwds*)

It is a wrap-up function of scanpy sc.tl.rank_genes_groups function. For more information see Scanpy documentations.

Here, we optimised the default features for the different kind of omics epiScanpy is analysing. Parameters for rankby_abs, method and corr_method are fixed byt the omic (if not specified otherwise)

In case you want to change all the settings, it is advise to directly use the Scanpy function or to specify the omic parameter as False.

If the omic of the input AnnData object is not specified (or incorrect), you can add it in omic (either 'RNA', 'ATAC' or 'methylation'). If the omic of the current matrix is not known by epiScanpy but you wnat to use settings of a known omic, specify which omic as a parameter.

episcanpy.tl.silhouette

episcanpy.tl.**silhouette**(*adata_name*, *cluster_annot*, *value='X_pca'*, *metric='euclidean'*, *key_added=None*, *copy=False*)

Compute silhouette scores.

It computes the general silhouette score as well as a silhouette score for every cell according to the cell cluster assigned to it.

Parameters

adata_name : *AnnData object*

cluster_annot : *observational variable corresponding to a cell clustering*

value : *measure used to build the silhouette plot (X_pca, X_tsne, X_umap)*

metric : *'euclidean'*

key_added : *key to save the computed silhouette scores*

Returns general silhouette score in 'uns' of the AnnData object individual silhouette scores in 'obs' of the AnnData object

Credit to sklearn script : https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html#sphx-glr-auto-examples-cluster-plot-kmeans-silhouette-analysis-py return score and silhouette plot. Still some work to do to finish the function. size=None but you can put 'large' if you want a bigger default figure size

episcanpy.tl.lazy

episcanpy.tl.**lazy**(*adata*, *pp_pca=True*, *copy=False*)

Automatically computes PCA coordinates, loadings and variance decomposition, a neighborhood graph of observations, t-distributed stochastic neighborhood embedding (tSNE) Uniform Manifold Approximation and Projection (UMAP)

Parameters

adata : AnnData Annotated data matrix.

pp_pca : *bool* (default: *True*) Computes PCA coordinates before the neighborhood graph

copy : *bool* (default: *False*) Return a copy instead of writing to adata.

Returns

Depending on *copy*, returns or updates *adata* with the following fields. X_pca : *adata.obsm*

PCA coordinates of data.

connectivities [sparse matrix (*.uns['neighbors']*, dtype *float32*)] Weighted adjacency matrix of the neighborhood graph of data points. Weights should be interpreted as connectivities.

distances [sparse matrix (*.uns['neighbors']*, dtype *float32*)] Instead of decaying weights, this stores distances for each pair of neighbors.

X_tsne [*np.ndarray* (*adata.obs*, dtype *float*)] tSNE coordinates of data.

X_umap [*adata.obsm*] UMAP coordinates of data.

episcanpy.tl.load_markers

episcanpy.tl.**load_markers**(*path*, *marker_list_file*)

Convert list of known cell type markers from literature to a dictionary Input list of known marker genes First row is considered the header

marker_list_file: file with the known marker genes (gene name is the 3rd column)

Returns cell_type_markers

episcanpy.tl.identify_cluster

episcanpy.tl.**identify_cluster**(*adata*, *cell_type*, *cell_type_markers*, *peak_promoter_file*, *gene_name_pos=5*, *path=''*, *n_peaks_per_cluster=1000*)

Use markers of a given cell type to plot peak openness for peaks in promoters of the given markers Input cell type, cell type markers, peak promoter intersections

cell_type: str, cell type that is to be investigated (must be the same as in the dictionary) cell_type_markers: dict, output of load_markers peak_promoter_file: tab-separated file that stores information about peak/promoter intersections and gene names for these promoters gene_name_pose: int, indicates the column in the peak_promoter_file where the gene name is stored path: str, path to the peak_promoter_file n_peaks_per_cluster: int, number of peaks per louvain cluster that should be searched for matches with the markers

Returns umap depicting peak openness for promoters of marker genes for a given cell type

episcanpy.tl.top_feature_genes

episcanpy.tl.**top_feature_genes**(*adata*, *gtf_file*, *extension=5000*)

episcanpy.tl.diffmap

episcanpy.tl.**diffmap**(*adata*, *n_comps=15*, *copy=False*)

Diffusion Maps [Coifman05] [Haghverdi15] [Wolf18].

Diffusion maps [Coifman05] has been proposed for visualizing single-cell data by [Haghverdi15]. The tool uses the adapted Gaussian kernel suggested by [Haghverdi16] in the implementation of [Wolf18].

The width ("sigma") of the connectivity kernel is implicitly determined by the number of neighbors used to compute the single-cell graph in `neighbors()`. To reproduce the original implementation using a Gaussian kernel, use *method=='gauss'* in `neighbors()`. To use an exponential kernel, use the default *method=='umap'*. Differences between these options shouldn't usually be dramatic.

Parameters

adata : AnnData Annotated data matrix.

n_comps : *int*, optional (default: 15) The number of dimensions of the representation.

copy : *bool* (default: *False*) Return a copy instead of writing to adata.

Returns

Depending on *copy*, returns or updates *adata* with the following fields.

X_diffmap [numpy.ndarray (*adata.obsm*)] Diffusion map representation of data, which is the right eigen basis of the transition matrix with eigenvectors as columns.

diffmap_evals [numpy.ndarray (*adata.uns*)] Array of size (number of eigen vectors). Eigenvalues of transition matrix.

episcanpy.tl.draw_graph

episcanpy.tl.**draw_graph**(*adata*, *layout='fa'*, *init_pos=None*, *root=None*, *random_state=0*, *n_jobs=None*, *adjacency=None*, *key_added_ext=None*, *copy=False*)

Force-directed graph drawing [Islam11] [Jacomy14] [Chippada18].

An alternative to tSNE that often preserves the topology of the data better. This requires to run `neighbors()`, first.

The default layout ('fa', *ForceAtlas2*) [Jacomy14] uses the package fa2 [Chippada18], which can be installed via *pip install fa2*.

Force-directed graph drawing describes a class of long-established algorithms for visualizing graphs. It has been suggested for visualizing single-cell data by [Islam11]. Many other layouts as implemented in igraph [Csardi06] are available. Similar approaches have been used by [Zunder15] or [Weinreb17].

Parameters

adata Annotated data matrix.

layout 'fa' (*ForceAtlas2*) or any valid igraph layout. Of particular interest are 'fr' (Fruchterman Reingold), 'grid_fr' (Grid Fruchterman Reingold, faster than 'fr'), 'kk' (Kamadi Kawai', slower than 'fr'), 'lgl' (Large Graph, very fast), 'drl' (Distributed Recursive Layout, pretty fast) and 'rt' (Reingold Tilford tree layout).

root Root for tree layouts.

random_state For layouts with random initialization like 'fr', change this to use different intial states for the optimization. If *None*, no seed is set.

adjacency Sparse adjacency matrix of the graph, defaults to *adata.uns['neighbors']['connectivities']*.

key_added_ext By default, append *layout*.

proceed Continue computation, starting off with 'X_draw_graph_'layout''.

init_pos *'paga'*/*True*, *None*/*False*, or any valid 2d-*.obsm* key. Use precomputed coordinates for initialization. If *False*/*None* (the default), initialize randomly.

copy Return a copy instead of writing to adata.

****kwds**

Parameters of chosen igraph layout. See e.g. fruchterman-reingold [Fruchterman91]. One of the most important ones is *maxiter*.

Returns

Depending on *copy*, returns or updates *adata* with the following field.

X_draw_graph_layout [*adata.obsm*] Coordinates of graph layout. E.g. for layout='fa' (the default), the field is called 'X_draw_graph_fa'

episcanpy.tl.tsne

episcanpy.tl.**tsne**(*adata*, *n_pcs=None*, *use_rep=None*, *perplexity=30*, *early_exaggeration=12*, *learning_rate=1000*, *random_state=0*, *use_fast_tsne=True*, *n_jobs=None*, *copy=False*)

t-SNE [Maaten08] [Amir13] [Pedregosa11].

t-distributed stochastic neighborhood embedding (tSNE) [Maaten08] has been proposed for visualizating single-cell data by [Amir13]. Here, by default, we use the implementation of *scikit-learn* [Pedregosa11]. You can achieve a huge speedup and better convergence if you install Multicore-tSNE by [Ulyanov16], which will be automatically detected by Scanpy.

Parameters

adata : AnnData Annotated data matrix.

{doc_n_pcs}

{use_rep}

perplexity : *float*, optional (default: 30) The perplexity is related to the number of nearest neighbors that is used in other manifold learning algorithms. Larger datasets usually require a larger perplexity. Consider selecting a value between 5 and 50. The choice is not extremely critical since t-SNE is quite insensitive to this parameter.

early_exaggeration : *float*, optional (default: 12.0) Controls how tight natural clusters in the original space are in the embedded space and how much space will be between them. For larger values, the space between natural clusters will be larger in the embedded space. Again, the choice of this parameter is not very critical. If the cost function increases during initial optimization, the early exaggeration factor or the learning rate might be too high.

learning_rate : *float*, optional (default: 1000) Note that the R-package "Rtsne" uses a default of 200. The learning rate can be a critical parameter. It should be between 100 and 1000. If the cost function increases during initial optimization, the early exaggeration factor or the learning rate might be too high. If the cost function gets stuck in a bad local minimum increasing the learning rate helps sometimes.

random_state : *int* or *None*, optional (default: 0) Change this to use different intial states for the optimization. If *None*, the initial state is not reproducible.

use_fast_tsne : *bool*, optional (default: *True*) Use the MulticoreTSNE package by D. Ulyanov if it is installed.

n_jobs : *int* or *None* (default: *sc.settings.n_jobs*) Number of jobs.

copy : *bool* (default: *False*) Return a copy instead of writing to adata.

Returns

Depending on *copy*, returns or updates *adata* with the following fields.

X_tsne [*np.ndarray* (*adata.obs*, dtype *float*)] tSNE coordinates of data.

episcanpy.tl.umap

episcanpy.tl.**umap**(*adata*, *min_dist=0.5*, *spread=1.0*, *n_components=2*, *maxiter=None*, *alpha=1.0*, *gamma=1.0*, *negative_sample_rate=5*, *init_pos='spectral'*, *random_state=0*, *a=None*, *b=None*, *copy=False*)

Embed the neighborhood graph using UMAP [McInnes18].

UMAP (Uniform Manifold Approximation and Projection) is a manifold learning technique suitable for visualizing high-dimensional data. Besides tending to be faster than tSNE, it optimizes the embedding such that it best reflects the topology of the data, which we represent throughout Scanpy using a neighborhood graph. tSNE, by contrast, optimizes the distribution of nearest-neighbor distances in the embedding such that these best match the distribution of distances in the high-dimensional space. We use the implementation of umap-learn [McInnes18]. For a few comparisons of UMAP with tSNE, see this preprint.

Parameters

adata : AnnData Annotated data matrix.

min_dist : *float*, optional (default: 0.5) The effective minimum distance between embedded points. Smaller values will result in a more clustered/clumped embedding where nearby points on the manifold are drawn closer together, while larger values will result on a more even dispersal of points. The value should be set relative to the `spread` value, which determines the scale at which embedded points will be spread out. The default of in the *umap-learn* package is 0.1.

spread : *float* (optional, default 1.0) The effective scale of embedded points. In combination with *min_dist* this determines how clustered/clumped the embedded points are.

n_components : *int*, optional (default: 2) The number of dimensions of the embedding.

maxiter : *int*, optional (default: *None*) The number of iterations (epochs) of the optimization. Called *n_epochs* in the original UMAP.

alpha : *float*, optional (default: 1.0) The initial learning rate for the embedding optimization.

gamma : *float* (optional, default 1.0) Weighting applied to negative samples in low dimensional embedding optimization. Values higher than one will result in greater weight being given to negative samples.

negative_sample_rate : *int* (optional, default 5) The number of negative edge/1-simplex samples to use per positive edge/1-simplex sample in optimizing the low dimensional embedding.

init_pos : *string* or *np.array*, optional (default: 'spectral')

How to initialize the low dimensional embedding. Called *init* in the original UMAP. Options are:

- Any key for *adata.obsm*.
- 'paga': positions from `paga()`.
- 'spectral': use a spectral embedding of the graph.
- 'random': assign initial embedding positions at random.
- A numpy array of initial embedding positions.

random_state : *int*, *RandomState* or *None*, optional (default: 0) If *int*, *random_state* is the seed used by the random number generator; If *RandomState*, *random_state* is the random number generator; If *None*, the random number generator is the *RandomState* instance used by *np.random*.

a : *float* (optional, default *None*) More specific parameters controlling the embedding. If *None* these values are set automatically as determined by *min_dist* and *spread*.

b : *float* (optional, default *None*) More specific parameters controlling the embedding. If *None* these values are set automatically as determined by *min_dist* and *spread*.

copy : *bool* (default: *False*) Return a copy instead of writing to adata.

method : {*'umap'*, *'rapids'*} (default: *'gauss'*) Use the original 'umap' implementation, or 'rapids' (experimental, GPU only)

Returns

Depending on *copy*, returns or updates *adata* with the following fields.

X_umap [*adata.obsm* field] UMAP coordinates of data.

episcanpy.tl.louvain

episcanpy.tl.**louvain**(*adata*, *resolution=None*, *random_state=0*, *restrict_to=None*, *key_added='louvain'*, *adjacency=None*, *flavor='vtraag'*, *directed=True*, *use_weights=False*, *partition_type=None*, *partition_kwargs=None*, *copy=False*)

Cluster cells into subgroups [Blondel08] [Levine15] [Traag17].

Cluster cells using the Louvain algorithm [Blondel08] in the implementation of [Traag17]. The Louvain algorithm has been proposed for single-cell analysis by [Levine15].

This requires having ran `neighbors()` or `bbknn()` first, or explicitly passing a `adjacency` matrix.

Parameters

adata The annotated data matrix.

resolution For the default flavor (`'vtraag'`), you can provide a resolution (higher resolution means finding more and smaller clusters), which defaults to 1.0. See "Time as a resolution parameter" in [Lambiotte09].

random_state Change the initialization of the optimization.

restrict_to Restrict the clustering to the categories within the key for sample annotation, tuple needs to contain `(obs_key, list_of_categories)`.

key_added Key under which to add the cluster labels. (default: `'louvain'`)

adjacency Sparse adjacency matrix of the graph, defaults to `adata.uns['neighbors']['connectivities']`.

flavor : {`'vtraag'`, `'igraph'`, `'rapids'`} Choose between to packages for computing the clustering. `'vtraag'` is much more powerful, and the default.

directed Interpret the `adjacency` matrix as directed graph?

use_weights Use weights from knn graph.

partition_type Type of partition to use. Only a valid argument if `flavor` is `'vtraag'`.

partition_kwargs Key word arguments to pass to partitioning, if `vtraag` method is being used.

copy Copy adata or modify it inplace.

Returns

None By default (`copy=False`), updates `adata` with the following fields:

adata.obs['louvain'] (pandas.Series, dtype category) Array of dim (number of samples) that stores the subgroup id (`'0'`, `'1'`, ...) for each cell.

AnnData When `copy=True` is set, a copy of `adata` with those fields is returned.

episcanpy.tl.leiden

episcanpy.tl.**leiden**(*adata*, *resolution=1*, ***, *restrict_to=None*, *random_state=0*, *key_added='leiden'*, *adjacency=None*, *directed=True*, *use_weights=True*, *n_iterations=-1*, *partition_type=None*, *copy=False*)

Cluster cells into subgroups [Traag18].

Cluster cells using the Leiden algorithm [Traag18], an improved version of the Louvain algorithm [Blondel08]. The Louvain algorithm has been proposed for single-cell analysis by [Levine15].

This requires having ran `neighbors()` or `bbknn()` first.

Parameters

adata The annotated data matrix.

resolution A parameter value controlling the coarseness of the clustering. Higher values lead to more clusters. Set to *None* if overriding *partition_type* to one that doesn't accept a *resolution_parameter*.

random_state Change the initialization of the optimization.

restrict_to Restrict the clustering to the categories within the key for sample annotation, tuple needs to contain *(obs_key, list_of_categories)*.

key_added *adata.obs* key under which to add the cluster labels. (default: *'leiden'*)

adjacency Sparse adjacency matrix of the graph, defaults to *adata.uns['neighbors']['connectivities']*.

directed Whether to treat the graph as directed or undirected.

use_weights If *True*, edge weights from the graph are used in the computation (placing more emphasis on stronger edges).

n_iterations How many iterations of the Leiden clustering algorithm to perform. Positive values above 2 define the total number of iterations to perform, -1 has the algorithm run until it reaches its optimal clustering.

partition_type Type of partition to use. Defaults to `RBConfigurationVertexPartition`. For the available options, consult the documentation for `find_partition()`.

copy Whether to copy *adata* or modify it inplace.

****partition_kwargs** Any further arguments to pass to *~leidenalg.find_partition* (which in turn passes arguments to the *partition_type*).

Returns

adata.obs[key_added] Array of dim (number of samples) that stores the subgroup id (*'0'*, *'1'*, ...) for each cell.

adata.uns['leiden']['params'] A dict with the values for the parameters *resolution*, *random_state*, and *n_iterations*.

1.5.4 Plotting: PL

The plotting module `episcanpy.plotting` largely parallels the `tl.*` and a few of the `pp.*` functions. For most tools and for some preprocessing functions, you'll find a plotting function with the same name.

pl.pca(adata[, color, feature_symbols, ...])	Scatter plot in PCA coordinates.
pl.pca_floadings	
pl.pca_overview(adata)	Plot PCA results.
pl.pca_variance_ratio(adata[, n_pcs, log, ...])	Plot the variance ratio.
pl.tsne(adata[, color, feature_symbols, ...])	Scatter plot in tSNE basis.
pl.umap(adata[, color, feature_symbols, ...])	Scatter plot in UMAP basis.
pl.diffmappl.draw_graph	
pl.rank_feat_groups(adata[, groups, ...])	Plot ranking of features.
pl.rank_feat_groups_violin(adata[, groups, ...])	Plot ranking of features for all tested comparisons.
pl.rank_feat_groups_dotplot(adata[, groups, ...])	Plot ranking of features using dotplot plot (see `dotplot()`)
pl.rank_feat_groups_stacked_violin(adata[, ...])	Plot ranking of features using stacked_violin plot (see `stacked_violin()`)
pl.rank_feat_groups_matrixplot(adata[, ...])	Plot ranking of features using matrixplot plot (see `matrixplot()`)
pl.rank_feat_groups_heatmap(adata[, groups, ...])	Plot ranking of features using heatmap plot (see `heatmap()`)
pl.rank_feat_groups_tracksplot(adata[, ...])	Plot ranking of features using heatmap plot (see `heatmap()`)
pl.cal_var(adata[, show])	
pl.violin(adata, keys[, groupby, log, ...])	Violin plot.
pl.scatter(adata[, x, y, color, use_raw, ...])	Scatter plot along observations or variables axes.
pl.ranking(adata, attr, keys[, dictionary, ...])	Plot rankings.
pl.clustermap(adata[, obs_keys, use_raw, ...])	Hierarchically-clustered heatmap.
pl.stacked_violin(adata, var_names[, ...])	Stacked violin plots.
pl.heatmap(adata, var_names[, groupby, ...])	Heatmap of the expression values of genes.
pl.dotplot(adata, var_names[, groupby, ...])	Makes a *dot plot* of the expression values of *var_names*.
pl.matrixplot(adata, var_names[, groupby, ...])	Creates a heatmap of the mean expression values per cluster of each var_names If groupby is not given, the matrixplot assumes that all data belongs to a single category.
pl.tracksplot(adata, var_names, groupby[, ...])	In this type of plot each var_name is plotted as a filled line plot where the y values correspond to the var_name values and x is each of the cells.
pl.dendrogram(adata, groupby[, ...])	Plots a dendrogram of the categories defined in *groupby*.
pl.correlation_matrix(adata, groupby[, ...])	Plots the correlation matrix computed as part of *sc.tl.dendrogram*.
pl.prct_overlap(adata, key_1, key_2[, norm, ...])	% or cell count corresponding to the overlap of different cell types between 2 set of annotations/clusters.
pl.overlap_heatmap(adata, key_1, key_2[, ...])	Heatmap of the cluster correspondance between 2 set of annaotations.
pl.cluster_composition(adata, cluster, condition)	
pl.silhouette(adata_name, cluster_annot[, ...])	Plot the product of tl.silhouette as a silhouette plot
pl.silhouette_tot(adata_name, cluster_annot)	Both compute silhouette scores and plot it.

episcanpy.pl.pca

episcanpy.pl.**pca**(*adata*, *color=None*, *feature_symbols=None*, *use_raw=None*, *layer=None*, *sort_order=True*, *groups=None*, *components=None*, *projection='2d'*, *legend_loc='right margin'*, *legend_fontsize=None*, *legend_fontweight=None*, *legend_fontoutline=None*, *size=None*, *color_map=None*, *palette=None*, *frameon=None*, *ncols=None*, *wspace=None*, *hspace=0.25*, *title=None*, *return_fig=None*, *show=None*, *save=None*)

Scatter plot in PCA coordinates.

Parameters

{adata_color_etc}

{scatter_bulk}

{show_save_ax}

Returns If *show==False* a `Axes` or a list of it.

episcanpy.pl.pca_overview

episcanpy.pl.**pca_overview**(*adata*)

Plot PCA results.

The parameters are the ones of the scatter plot. Call pca_ranking separately if you want to change the default settings.

Parameters

adata Annotated data matrix.

color : string or list of strings, optional (default: *None*) Keys for observation/cell annotation either as list *["ann1", "ann2"]* or string *"ann1,ann2,..."*.

use_raw : *bool*, optional (default: *True*) Use *raw* attribute of *adata* if present.

{scatter_bulk}

show : bool, optional (default: *None*) Show the plot, do not return axis.

save : *bool* or *str*, optional (default: *None*) If *True* or a *str*, save the figure. A string is appended to the default filename. Infer the filetype if ending on {{*'.pdf'*, *'.png'*, *'.svg'*}}.

episcanpy.pl.pca_variance_ratio

episcanpy.pl.**pca_variance_ratio**(*adata*, *n_pcs=30*, *log=False*, *show=None*, *save=None*)

Plot the variance ratio.

Parameters

n_pcs : *int*, optional (default: *30*) Number of PCs to show.

log : *bool*, optional (default: *False*) Plot on logarithmic scale..

show : *bool*, optional (default: *None*) Show the plot, do not return axis.

save : *bool* or *str*, optional (default: *None*) If *True* or a *str*, save the figure. A string is appended to the default filename. Infer the filetype if ending on {*'.pdf'*, *'.png'*, *'.svg'*}.

episcanpy.pl.tsne

episcanpy.pl.**tsne**(*adata*, *color=None*, *feature_symbols=None*, *use_raw=None*, *layer=None*, *sort_order=True*, *groups=None*, *components=None*, *projection='2d'*, *legend_loc='right margin'*, *legend_fontsize=None*, *legend_fontweight=None*, *legend_fontoutline=None*, *size=None*, *color_map=None*, *palette=None*, *frameon=None*, *ncols=None*, *wspace=None*, *hspace=0.25*, *title=None*, *return_fig=None*, *show=None*, *save=None*)

Scatter plot in tSNE basis.

Parameters

{adata_color_etc}

{edges_arrows}

{scatter_bulk}

{show_save_ax}

Returns If *show==False* a `Axes` or a list of it.

episcanpy.pl.umap

episcanpy.pl.**umap**(*adata*, *color=None*, *feature_symbols=None*, *use_raw=None*, *layer=None*, *sort_order=True*, *groups=None*, *components=None*, *projection='2d'*, *legend_loc='right margin'*, *legend_fontsize=None*, *legend_fontweight=None*, *legend_fontoutline=None*, *size=None*, *color_map=None*, *palette=None*, *frameon=None*, *ncols=None*, *wspace=None*, *hspace=0.25*, *title=None*, *return_fig=None*, *show=None*, *save=None*)

Scatter plot in UMAP basis.

Parameters

{adata_color_etc}

{edges_arrows}

{scatter_bulk}

{show_save_ax}

Returns If *show==False* a `Axes` or a list of it.

episcanpy.pl.rank_feat_groups

episcanpy.pl.**rank_feat_groups**(*adata*, *groups=None*, *n_features=20*, *feature_symbols=None*, *key='rank_features_groups'*, *fontsize=8*, *ncols=4*, *sharey=True*, *show=None*, *save=None*, *ax=None*)

Plot ranking of features.

Parameters

adata Annotated data matrix.

groups The groups for which to show the feature ranking.

feature_symbols Key for field in *.var* that stores feature symbols if you do not want to use *.var_names*.

n_features Number of feature to show.

fontsize Fontsize for feature names.

ncols Number of panels shown per row.

sharey Controls if the y-axis of each panels should be shared. But passing *sharey=False*, each panel has its own y-axis range.

{show_save_ax}

episcanpy.pl.rank_feat_groups_violin

episcanpy.pl.**rank_feat_groups_violin**(*adata*, *groups=None*, *n_features=20*, *feature_names=None*, *feature_symbols=None*, *use_raw=None*, *key='rank_features_groups'*, *split=True*, *scale='width'*, *strip=True*, *jitter=True*, *size=1*, *ax=None*, *show=None*, *save=None*)

Plot ranking of features for all tested comparisons.

Parameters

adata Annotated data matrix.

groups List of group names.

n_features Number of features to show. Is ignored if *feature_names* is passed.

feature_names List of features to plot. Is only useful if interested in a custom feature list, which is not the result of `epi.tl.rank_features()`.

feature_symbols Key for field in *.var* that stores feature symbols if you do not want to use *.var_names* displayed in the plot.

use_raw : *bool*, optional (default: *None*) Use *raw* attribute of *adata* if present. Defaults to the value that was used in `rank_genes_groups()`.

split Whether to split the violins or not.

scale See `violinplot()`.

strip Show a strip plot on top of the violin plot.

jitter If set to 0, no points are drawn. See `stripplot()`.

size Size of the jitter points.

{show_save_ax}

episcanpy.pl.rank_feat_groups_dotplot

episcanpy.pl.**rank_feat_groups_dotplot**(*adata*, *groups=None*, *n_features=10*, *groupby=None*, *key='rank_features_groups'*, *show=None*, *save=None*)

Plot ranking of features using dotplot plot (see `dotplot()`)

Parameters

adata Annotated data matrix.

groups The groups for which to show the feature ranking.

n_features Number of features to show.

groupby The key of the observation grouping to consider. By default, the groupby is chosen from the rank features groups parameter but other groupby options can be used. It is expected that groupby is a categorical. If groupby is not a categorical observation, it would be subdivided into *num_categories* (see `dotplot()`).

key Key used to store the ranking results in *adata.uns*.

{show_save_ax}

****kwds** Are passed to `dotplot()`.

episcanpy.pl.rank_feat_groups_stacked_violin

episcanpy.pl.**rank_feat_groups_stacked_violin**(*adata*, *groups=None*, *n_features=10*, *groupby=None*, *key='rank_features_groups'*, *show=None*, *save=None*)

Plot ranking of features using stacked_violin plot (see `stacked_violin()`)

Parameters

adata Annotated data matrix.

groups : *str* or *list* of *str* The groups for which to show the feature ranking.

n_features : *int*, optional (default: 10) Number of features to show.

groupby : *str* or *None*, optional (default: *None*) The key of the observation grouping to consider. By default, the groupby is chosen from the rank features groups parameter but other groupby options can be used. It is expected that groupby is a categorical. If groupby is not a categorical observation, it would be subdivided into *num_categories* (see `stacked_violin()`).

key Key used to store the ranking results in *adata.uns*.

{show_save_ax}

****kwds** Are passed to `stacked_violin()`.

episcanpy.pl.rank_feat_groups_matrixplot

episcanpy.pl.**rank_feat_groups_matrixplot**(*adata*, *groups=None*, *n_features=10*, *groupby=None*, *key='rank_features_groups'*, *show=None*, *save=None*)

Plot ranking of features using matrixplot plot (see `matrixplot()`)

Parameters

adata Annotated data matrix.

groups The groups for which to show the feature ranking.

n_features Number of features to show.

groupby The key of the observation grouping to consider. By default, the groupby is chosen from the rank features groups parameter but other groupby options can be used. It is expected that groupby is a categorical. If groupby is not a categorical observation, it would be subdivided into *num_categories* (see `matrixplot()`).

key Key used to store the ranking results in *adata.uns*.

{show_save_ax}

****kwds** Are passed to `matrixplot()`.

episcanpy.pl.rank_feat_groups_heatmap

episcanpy.pl.**rank_feat_groups_heatmap**(*adata*, *groups=None*, *n_features=10*, *groupby=None*, *key='rank_features_groups'*, *show=None*, *save=None*)

Plot ranking of features using heatmap plot (see `heatmap()`)

Parameters

adata : AnnData Annotated data matrix.

groups : *str* or *list* of *str* The groups for which to show the feature ranking.

n_features Number of features to show.

groupby The key of the observation grouping to consider. By default, the groupby is chosen from the rank features groups parameter but other groupby options can be used. It is expected that groupby is a categorical. If groupby is not a categorical observation, it would be subdivided into *num_categories* (see `heatmap()`).

key Key used to store the ranking results in *adata.uns*.

****kwds** Are passed to `heatmap()`.

{show_save_ax}

episcanpy.pl.rank_feat_groups_tracksplot

episcanpy.pl.**rank_feat_groups_tracksplot**(*adata*, *groups=None*, *n_features=10*, *groupby=None*, *key='rank_features_groups'*, *show=None*, *save=None*)

Plot ranking of features using heatmap plot (see `heatmap()`)

Parameters

adata Annotated data matrix.

groups The groups for which to show the feature ranking.

n_features Number of features to show.

groupby The key of the observation grouping to consider. By default, the groupby is chosen from the rank features groups parameter but other groupby options can be used. It is expected that groupby is a categorical. If groupby is not a categorical observation, it would be subdivided into *num_categories* (see `heatmap()`).

key Key used to store the ranking results in *adata.uns*.

****kwds** Are passed to `tracksplot()`.

{show_save_ax}

episcanpy.pl.cal_var

episcanpy.pl.**cal_var**(*adata*, *show=True*)

episcanpy.pl.violin

episcanpy.pl.**violin**(*adata*, *keys*, *groupby=None*, *log=False*, *use_raw=None*, *stripplot=True*, *jitter=True*, *size=1*, *layer=None*, *scale='width'*, *order=None*, *multi_panel=None*, *xlabel=''*, *rotation=None*, *show=None*, *save=None*, *ax=None*, ***kwds*)

Violin plot.

Wraps `seaborn.violinplot()` for `AnnData`.

Parameters

adata : AnnDataAnnData Annotated data matrix.

keys : str, Sequence[str]Union[str, Sequence[str]] Keys for accessing variables of *.var_names* or fields of *.obs*.

groupby : str, NoneOptional[str] (default: None) The key of the observation grouping to consider.

log : boolbool (default: False) Plot on logarithmic axis.

use_raw : bool, NoneOptional[bool] (default: None) Use *raw* attribute of *adata* if present.

stripplot : boolbool (default: True) Add a stripplot on top of the violin plot. See `stripplot()`.

jitter : float, boolUnion[float, bool] (default: True) Add jitter to the stripplot (only when stripplot is True) See `stripplot()`.

size : intint (default: 1) Size of the jitter points.

layer : str, NoneOptional[str] (default: None) Name of the AnnData object layer that wants to be plotted. By default adata.raw.X is plotted. If *use_raw=False* is set, then *adata.X* is plotted. If *layer* is set to a valid layer name, then the layer is plotted. *layer* takes precedence over *use_raw*.

scale : Literal_[area, count, width]Literal_[area, count, width] (default: 'width') The method used to scale the width of each violin. If 'width' (the default), each violin will have the same width. If 'area', each violin will have the same area. If 'count', a violin's width corresponds to the number of observations.

order : Sequence[str], NoneOptional[Sequence[str]] (default: None) Order in which to show the categories.

multi_panel : bool, NoneOptional[bool] (default: None) Display keys in multiple panels also when *groupby is not None*.

xlabel : strstr (default: '') Label of the x axis. Defaults to *groupby* if *rotation* is *None*, otherwise, no label is shown.

rotation : float, NoneOptional[float] (default: None) Rotation of xtick labels.

show : bool, NoneOptional[bool] (default: None) Show the plot, do not return axis.

save : bool, str, NoneUnion[bool, str, None] (default: None) If *True* or a *str*, save the figure. A string is appended to the default filename. Infer the filetype if ending on {*'.pdf'*, *'.png'*, *'.svg'*}.

ax : Axes, NoneOptional[Axes] (default: None) A matplotlib axes object. Only works if plotting a single component.

****kwds** Are passed to `violinplot()`.

Returns A `Axes` object if *ax* is *None* else *None*.

episcanpy.pl.scatter

episcanpy.pl.**scatter**(*adata*, *x=None*, *y=None*, *color=None*, *use_raw=None*, *layers=None*, *sort_order=True*, *alpha=None*, *basis=None*, *groups=None*, *components=None*, *projection='2d'*, *legend_loc='right margin'*, *legend_fontsize=None*, *legend_fontweight=None*, *legend_fontoutline=None*, *color_map=None*, *palette=None*, *frameon=None*, *right_margin=None*, *left_margin=None*, *size=None*, *title=None*, *show=None*, *save=None*, *ax=None*)

Scatter plot along observations or variables axes.

Color the plot using annotations of observations (*.obs*), variables (*.var*) or expression of genes (*.var_names*).

Parameters

adata : AnnDataAnnData Annotated data matrix.

x : str, NoneOptional[str] (default: None) x coordinate.

y : str, NoneOptional[str] (default: None) y coordinate.

color : str, Collection[str], NoneUnion[str, Collection[str], None] (default: None) Keys for annotations of observations/cells or variables/genes, or a hex color specification, e.g., *'ann1'*, *'#fe57a1'*, or *['ann1', 'ann2']*.

use_raw : bool, NoneOptional[bool] (default: None) Use *raw* attribute of *adata* if present.

layers : str, Collection[str], NoneUnion[str, Collection[str], None] (default: None) Use the *layers* attribute of *adata* if present: specify the layer for *x*, *y* and *color*. If *layers* is a string, then it is expanded to *(layers, layers, layers)*.

basis : Literal_[pca, tsne, umap, diffmap, draw_graph_fr], NoneOptional[Literal_[pca, tsne, umap, d String that denotes a plotting tool that computed coordinates.

sort_order : boolbool (default: True) For continuous annotations used as color parameter, plot data points with higher values on top of others.

groups : str, Iterable[str], NoneUnion[str, Iterable[str], None] (default: None) Restrict to a few categories in categorical observation annotation. The default is not to restrict to any groups.

components : str, Collection[str], NoneUnion[str, Collection[str], None] (default: None) For instance, *['1,2', '2,3']*. To plot all available components use *components='all'*.

projection : Literal_[2d, 3d]Literal_[2d, 3d] (default: '2d') Projection of plot (default: *'2d'*).

legend_loc : strstr (default: 'right margin') Location of legend, either *'on data'*, *'right margin'* or a valid keyword for the *loc* parameter of `Legend`.

legend_fontsize : int, float, Literal_[xx-small, x-small, small, medium, large, x-large, xx-large], Non Numeric size in pt or string describing the size. See `set_fontsize()`.

legend_fontweight : int, Literal_[light, normal, medium, semibold, bold, heavy, black], NoneUnion[i Legend font weight. A numeric value in range 0-1000 or a string. Defaults to *'bold'* if *legend_loc == 'on data'*, otherwise to *'normal'*. See `set_fontweight()`.

legend_fontoutline : float, NoneOptional[float] (default: None) Line width of the legend font outline in pt. Draws a white outline using the path effect `withStroke`.

size : int, float, NoneUnion[int, float, None] (default: None) Point size. If *None*, is automatically computed as 120000 / n_cells. Can be a sequence containing the size for each cell. The order should be the same as in adata.obs.

color_map : str, Colormap, NoneUnion[str, Colormap, None] (default: None)
Color map to use for continous variables. Can be a name or a Colormap instance (e.g. *"magma"*, *"viridis"* or *mpl.cm.cividis*), see get_cmap(). If *None*, the value of *mpl.rcParams["image.cmap"]* is used. The default *color_map* can be set using set_figure_params().

palette : Cycler, ListedColormap, str, Tuple[float, ...], Sequence[Union[str, Tuple[float, ..
Colors to use for plotting categorical annotation groups. The palette can be a valid ListedColormap name (*'Set2'*, *'tab20'*, ...), a Cycler object, or a sequence of matplotlib colors like *['red', '#ccdd11', (0.1, 0.2, 1)]* (see is_color_like()). If *None*, *mpl.rcParams["axes.prop_cycle"]* is used unless the categorical variable already has colors stored in *adata.uns["{var}_colors"]*. If provided, values of *adata.uns["{var}_colors"]* will be set.

frameon : bool, NoneOptional[bool] (default: None) Draw a frame around the scatter plot. Defaults to value set in set_figure_params(), defaults to *True*.

title : str, NoneOptional[str] (default: None) Provide title for panels either as string or list of strings, e.g. *['title1', 'title2', ...]*.

show : bool, NoneOptional[bool] (default: None) Show the plot, do not return axis.

save : bool, str, NoneUnion[bool, str, None] (default: None) If *True* or a *str*, save the figure. A string is appended to the default filename. Infer the filetype if ending on {*'.pdf'*, *'.png'*, *'.svg'*}.

ax : Axes, NoneOptional[Axes] (default: None) A matplotlib axes object. Only works if plotting a single component.

Returns If *show==False* a Axes or a list of it.

episcanpy.pl.ranking

episcanpy.pl.**ranking**(*adata*, *attr*, *keys*, *dictionary=None*, *indices=None*, *labels=None*, *color='black'*, *n_points=30*, *log=False*, *include_lowest=False*, *show=None*)

Plot rankings.

See, for example, how this is used in pl.pca_ranking.

Parameters

adata : AnnDataAnnData The data.

attr : Literal_[var, obs, uns, varm, obsm]Literal_[var, obs, uns, varm, obsm] The attribute of AnnData that contains the score.

keys : str, Sequence[str]Union[str, Sequence[str]] The scores to look up an array from the attribute of adata.

Returns Returns matplotlib gridspec with access to the axes.

episcanpy.pl.clustermap

episcanpy.pl.**clustermap**(*adata*, *obs_keys=None*, *use_raw=None*, *show=None*, *save=None*, ***kwds*)

Hierarchically-clustered heatmap.

Wraps seaborn.clustermap() for AnnData.

Parameters

adata : AnnDataAnnData Annotated data matrix.

obs_keys : str, NoneOptional[str] (default: None) Categorical annotation to plot with a different color map. Currently, only a single key is supported.

use_raw : bool, NoneOptional[bool] (default: None) Use *raw* attribute of *adata* if present.

show : bool, NoneOptional[bool] (default: None) Show the plot, do not return axis.

save : bool, str, NoneUnion[bool, str, None] (default: None) If *True* or a *str*, save the figure. A string is appended to the default filename. Infer the filetype if ending on {*'.pdf'*, *'.png'*, *'.svg'*}.

ax A matplotlib axes object. Only works if plotting a single component.

****kwds** Keyword arguments passed to `clustermap()`.

Returns If *show* is *False*, a `ClusterGrid` object (see `clustermap()`).

Examples

Soon to come with figures. In the meanwile, see `clustermap()`.

```
>>> import scanpy as sc
>>> adata = sc.datasets.krumsiek11()
>>> sc.pl.clustermap(adata, obs_keys='cell_type')
```

episcanpy.pl.stacked_violin

episcanpy.pl.**stacked_violin**(*adata*, *var_names*, *groupby=None*, *log=False*, *use_raw=None*, *num_categories=7*, *figsize=None*, *dendrogram=False*, *gene_symbols=None*, *var_group_positions=None*, *var_group_labels=None*, *standard_scale=None*, *var_group_rotation=None*, *layer=None*, *stripplot=False*, *jitter=False*, *size=1*, *scale='width'*, *order=None*, *swap_axes=False*, *show=None*, *save=None*, *row_palette='muted'*, ***kwds*)

Stacked violin plots.

Makes a compact image composed of individual violin plots (from `violinplot()`) stacked on top of each other. Useful to visualize gene expression per cluster.

Wraps `seaborn.violinplot()` for `AnnData`.

Parameters

adata : AnnDataAnnData Annotated data matrix.

var_names : str, Sequence[str], MappingUnion[str, Sequence[str], Mapping[str, Union[str, Se *var_names* should be a valid subset of *adata.var_names*. If *var_names* is a mapping, then the key is used as label to group the values (see *var_group_labels*). The mapping values should be sequences of valid *adata.var_names*. In this case either coloring or 'brackets' are used for the grouping of var names depending on the plot. When *var_names* is a mapping, then the *var_group_labels* and *var_group_positions* are set.

groupby : str, NoneOptional[str] (default: None) The key of the observation grouping to consider.

use_raw : bool, NoneOptional[bool] (default: None) Use *raw* attribute of *adata* if present.

log : boolbool (default: False) Plot on logarithmic axis.

num_categories : intint (default: 7) Only used if groupby observation is not categorical. This value determines the number of groups into which the groupby observation should be subdivided.

figsize : Tuple[float, float], NoneOptional[Tuple[float, float]] (default: None) Figure size when *multi_panel=True*. Otherwise the *rcParam['figure.figsize]* value is used. Format is (width, height)

dendrogram : bool, strUnion[bool, str] (default: False) If True or a valid dendrogram key, a dendrogram based on the hierarchical clustering between the *groupby* categories is added. The dendrogram information is computed using `scanpy.tl.dendrogram()`. If *tl.dendrogram* has not been called previously the function is called with default parameters.

gene_symbols : str, NoneOptional[str] (default: None) Column name in *.var* DataFrame that stores gene symbols. By default *var_names* refer to the index column of the *.var* DataFrame. Setting this option allows alternative names to be used.

var_group_positions : Sequence[Tuple[int, int]], NoneOptional[Sequence[Tuple[int, int]]] (Use this parameter to highlight groups of *var_names*. This will draw a 'bracket' or a color block between the given start and end positions. If the parameter *var_group_labels* is set, the corresponding labels are added on top/left. E.g. *var_group_positions=[(4,10)]* will add a bracket between the fourth *var_name* and the tenth *var_name*. By giving more positions, more brackets/color blocks are drawn.

var_group_labels : Sequence[str], NoneOptional[Sequence[str]] (default: None) Labels for each of the *var_group_positions* that want to be highlighted.

var_group_rotation : float, NoneOptional[float] (default: None) Label rotation degrees. By default, labels larger than 4 characters are rotated 90 degrees.

layer : str, NoneOptional[str] (default: None) Name of the AnnData object layer that wants to be plotted. By default adata.raw.X is plotted. If *use_raw=False* is set, then *adata.X* is plotted. If *layer* is set to a valid layer name, then the layer is plotted. *layer* takes precedence over *use_raw*.

stripplot : boolbool (default: False) Add a stripplot on top of the violin plot. See `stripplot()`.

jitter : float, boolUnion[float, bool] (default: False) Add jitter to the stripplot (only when stripplot is True) See `stripplot()`.

size : intint (default: 1) Size of the jitter points.

order : Sequence[str], NoneOptional[Sequence[str]] (default: None) Order in which to show the categories. Note: if *dendrogram=True* the categories order will be given by the dendrogram and *order* will be ignored.

scale : Literal_[area, count, width]Literal_[area, count, width] (default: 'width') The method used to scale the width of each violin. If 'width' (the default), each violin will have the same width. If 'area', each violin will have the same area. If 'count', a violin's width corresponds to the number of observations.

row_palette : strstr (default: 'muted') The row palette determines the colors to use for the stacked violins. The value should be a valid seaborn or matplotlib palette name (see `color_palette()`). Alternatively, a single color name or hex value can be passed, e.g. *'red'* or *'#cc33ff'*.

standard_scale : Literal_[var, obs], NoneOptional[Literal_[var, obs]] (default: None) Whether or not to standardize a dimension between 0 and 1, meaning for each variable or observation, subtract the minimum and divide each by its maximum.

swap_axes : boolbool (default: False) By default, the x axis contains *var_names* (e.g. genes) and the y axis the *groupby* categories. By setting *swap_axes* then x are the *groupby* categories and y the *var_names*. When swapping axes var_group_positions are no longer used

show : bool, NoneOptional[bool] (default: None) Show the plot, do not return axis.

save : bool, str, NoneUnion[bool, str, None] (default: None) If *True* or a *str*, save the figure. A string is appended to the default filename. Infer the filetype if ending on {*'.pdf'*, *'.png'*, *'.svg'*}.

ax A matplotlib axes object. Only works if plotting a single component.

****kwds** Are passed to `violinplot()`.

Returns List of `Axes`

Examples

```
>>> import scanpy as sc
>>> adata = sc.datasets.pbmc68k_reduced()
>>> markers = ['C1QA', 'PSAP', 'CD79A', 'CD79B', 'CST3', 'LYZ']
>>> sc.pl.stacked_violin(adata, markers, groupby='bulk_labels', dendrogram=True)
```

Using var_names as dict:

```
>>> markers = {'T-cell': 'CD3D', 'B-cell': 'CD79A', 'myeloid': 'CST3'}
>>> sc.pl.stacked_violin(adata, markers, groupby='bulk_labels', dendrogram=True)
```

See also:

rank_genes_groups_stacked_violin() to plot marker genes identified using the `rank_genes_groups()` function.

episcanpy.pl.heatmap

episcanpy.pl.**heatmap**(*adata*, *var_names*, *groupby=None*, *use_raw=None*, *log=False*, *num_categories=7*, *dendrogram=False*, *gene_symbols=None*, *var_group_positions=None*, *var_group_labels=None*, *var_group_rotation=None*, *layer=None*, *standard_scale=None*, *swap_axes=False*, *show_gene_labels=None*, *show=None*, *save=None*, *figsize=None*, ***kwds*)

Heatmap of the expression values of genes.

If *groupby* is given, the heatmap is ordered by the respective group. For example, a list of marker genes can be plotted, ordered by clustering. If the *groupby* observation annotation is not categorical the observation annotation is turned into a categorical by binning the data into the number specified in *num_categories*.

Parameters

adata : AnnDataAnnData Annotated data matrix.

var_names : str, Sequence[str], MappingUnion[str, Sequence[str], Mapping[str, Union[str, Se *var_names* should be a valid subset of *adata.var_names*. If *var_names* is a mapping, then the key is used as label to group the values (see *var_group_labels*). The mapping values should be sequences of valid *adata.var_names*. In this case either coloring or 'brackets' are used for the grouping of var names depending on the plot. When *var_names* is a mapping, then the *var_group_labels* and *var_group_positions* are set.

groupby : str, NoneOptional[str] (default: None) The key of the observation grouping to consider.

use_raw : bool, NoneOptional[bool] (default: None) Use *raw* attribute of *adata* if present.

log : boolbool (default: False) Plot on logarithmic axis.

num_categories : intint (default: 7) Only used if groupby observation is not categorical. This value determines the number of groups into which the groupby observation should be subdivided.

figsize : Tuple[float, float], NoneOptional[Tuple[float, float]] (default: None) Figure size when *multi_panel=True*. Otherwise the *rcParam['figure.figsize]* value is used. Format is (width, height)

dendrogram : bool, strUnion[bool, str] (default: False) If True or a valid dendrogram key, a dendrogram based on the hierarchical clustering between the *groupby* categories is added. The dendrogram information is computed using `scanpy.tl.dendrogram()`. If *tl.dendrogram* has not been called previously the function is called with default parameters.

gene_symbols : str, NoneOptional[str] (default: None) Column name in *.var* DataFrame that stores gene symbols. By default *var_names* refer to the index column of the *.var* DataFrame. Setting this option allows alternative names to be used.

var_group_positions : Sequence[Tuple[int, int]], NoneOptional[Sequence[Tuple[int, int]]] (
Use this parameter to highlight groups of *var_names*. This will draw a 'bracket' or a color block between the given start and end positions. If the parameter *var_group_labels* is set, the corresponding labels are added on top/left. E.g. *var_group_positions=[(4,10)]* will add a bracket between the fourth *var_name* and the tenth *var_name*. By giving more positions, more brackets/color blocks are drawn.

var_group_labels : Sequence[str], NoneOptional[Sequence[str]] (default: None) Labels for each of the *var_group_positions* that want to be highlighted.

var_group_rotation : float, NoneOptional[float] (default: None) Label rotation degrees. By default, labels larger than 4 characters are rotated 90 degrees.

layer : str, NoneOptional[str] (default: None) Name of the AnnData object layer that wants to be plotted. By default adata.raw.X is plotted. If *use_raw=False* is set, then *adata.X* is plotted. If *layer* is set to a valid layer name, then the layer is plotted. *layer* takes precedence over *use_raw*.

standard_scale : Literal_[var, obs], NoneOptional[Literal_[var, obs]] (default: None) Whether or not to standardize that dimension between 0 and 1, meaning for each variable or observation, subtract the minimum and divide each by its maximum.

swap_axes : boolbool (default: False) By default, the x axis contains *var_names* (e.g. genes) and the y axis the *groupby* categories (if any). By setting *swap_axes* then x are the *groupby* categories and y the *var_names*.

show_gene_labels : bool, NoneOptional[bool] (default: None) By default gene labels are shown when there are 50 or less genes. Otherwise the labels are removed.

show : bool, NoneOptional[bool] (default: None) Show the plot, do not return axis.

save : bool, str, NoneUnion[bool, str, None] (default: None) If *True* or a *str*, save the figure. A string is appended to the default filename. Infer the filetype if ending on {*'.pdf'*, *'.png'*, *'.svg'*}.

ax A matplotlib axes object. Only works if plotting a single component.

****kwds** Are passed to `matplotlib.pyplot.imshow()`.

Returns List of `Axes`

Examples

```
>>> import scanpy as sc
>>> adata = sc.datasets.pbmc68k_reduced()
>>> markers = ['C1QA', 'PSAP', 'CD79A', 'CD79B', 'CST3', 'LYZ']
>>> sc.pl.heatmap(adata, markers, groupby='bulk_labels', dendrogram=True, swap_
↪axes=True)
```

Using var_names as dict:

```
>>> markers = {'T-cell': 'CD3D', 'B-cell': 'CD79A', 'myeloid': 'CST3'}
>>> sc.pl.heatmap(adata, markers, groupby='bulk_labels', dendrogram=True)
```

See also:

rank_genes_groups_heatmap() to plot marker genes identified using the `rank_genes_groups()` function.

episcanpy.pl.dotplot

episcanpy.pl.**dotplot**(*adata*, *var_names*, *groupby=None*, *use_raw=None*, *log=False*, *num_categories=7*, *expression_cutoff=0.0*, *mean_only_expressed=False*, *color_map='Reds'*, *dot_max=None*, *dot_min=None*, *standard_scale=None*, *smallest_dot=0.0*, *figsize=None*, *dendrogram=False*, *gene_symbols=None*, *var_group_positions=None*, *var_group_labels=None*, *var_group_rotation=None*, *layer=None*, *show=None*, *save=None*, ***kwds*)

Makes a *dot plot* of the expression values of *var_names*.

For each var_name and each *groupby* category a dot is plotted. Each dot represents two values: mean expression within each category (visualized by color) and fraction of cells expressing the *var_name* in the category (visualized by the size of the dot). If *groupby* is not given, the dotplot assumes that all data belongs to a single category.

Note: A gene is considered expressed if the expression value in the *adata* (or *adata.raw*) is above the specified threshold which is zero by default.

An example of dotplot usage is to visualize, for multiple marker genes, the mean value and the percentage of cells expressing the gene accross multiple clusters.

Parameters

adata : AnnDataAnnData Annotated data matrix.

var_names : str, Sequence[str], MappingUnion[str, Sequence[str], Mapping[str, Union[str, Se

var_names should be a valid subset of *adata.var_names*. If *var_names* is a mapping, then the key is used as label to group the values (see *var_group_labels*). The mapping values should be sequences of valid *adata.var_names*. In this case either coloring or 'brackets' are used for the grouping of var names depending on the plot. When *var_names* is a mapping, then the *var_group_labels* and *var_group_positions* are set.

groupby : str, NoneOptional[str] (default: None) The key of the observation grouping to consider.

use_raw : bool, NoneOptional[bool] (default: None) Use *raw* attribute of *adata* if present.

log : boolbool (default: False) Plot on logarithmic axis.

num_categories : intint (default: 7) Only used if groupby observation is not categorical. This value determines the number of groups into which the groupby observation should be subdivided.

figsize : Tuple[float, float], NoneOptional[Tuple[float, float]] (default: None) Figure size when *multi_panel=True*. Otherwise the *rcParam['figure.figsize]* value is used. Format is (width, height)

dendrogram : bool, strUnion[bool, str] (default: False) If True or a valid dendrogram key, a dendrogram based on the hierarchical clustering between the *groupby* categories is added. The dendrogram information is computed using `scanpy.tl.dendrogram()`. If *tl.dendrogram* has not been called previously the function is called with default parameters.

gene_symbols : str, NoneOptional[str] (default: None) Column name in *.var* DataFrame that stores gene symbols. By default *var_names* refer to the index column of the *.var* DataFrame. Setting this option allows alternative names to be used.

var_group_positions : Sequence[Tuple[int, int]], NoneOptional[Sequence[Tuple[int, int]]] (
Use this parameter to highlight groups of *var_names*. This will draw a 'bracket' or a color block between the given start and end positions. If the parameter *var_group_labels* is set, the corresponding labels are added on top/left. E.g. *var_group_positions=[(4,10)]* will add a bracket between the fourth *var_name* and the tenth *var_name*. By giving more positions, more brackets/color blocks are drawn.

var_group_labels : Sequence[str], NoneOptional[Sequence[str]] (default: None) Labels for each of the *var_group_positions* that want to be highlighted.

var_group_rotation : float, NoneOptional[float] (default: None) Label rotation degrees. By default, labels larger than 4 characters are rotated 90 degrees.

layer : str, NoneOptional[str] (default: None) Name of the AnnData object layer that wants to be plotted. By default adata.raw.X is plotted. If *use_raw=False* is set, then *adata.X* is plotted. If *layer* is set to a valid layer name, then the layer is plotted. *layer* takes precedence over *use_raw*.

expression_cutoff : floatfloat (default: 0.0) Expression cutoff that is used for binarizing the gene expression and determining the fraction of cells expressing given genes. A gene is expressed only if the expression value is greater than this threshold.

mean_only_expressed : boolbool (default: False) If True, gene expression is averaged only over the cells expressing the given genes.

color_map : strstr (default: 'Reds') String denoting matplotlib color map.

dot_max : float, NoneOptional[float] (default: None) If none, the maximum dot size is set to the maximum fraction value found (e.g. 0.6). If given, the value should be a number between 0 and 1. All fractions larger than dot_max are clipped to this value.

dot_min : float, NoneOptional[float] (default: None) If none, the minimum dot size is set to 0. If given, the value should be a number between 0 and 1. All fractions smaller than dot_min are clipped to this value.

standard_scale : Literal_[var, group], NoneOptional[Literal_[var, group]] (default: None) Whether or not to standardize that dimension between 0 and 1, meaning for each variable or group, subtract the minimum and divide each by its maximum.

smallest_dot : floatfloat (default: 0.0) If none, the smallest dot has size 0. All expression levels with *dot_min* are plotted with this size.

show : bool, NoneOptional[bool] (default: None) Show the plot, do not return axis.

save : bool, str, NoneUnion[bool, str, None] (default: None) If *True* or a *str*, save the figure. A string is appended to the default filename. Infer the filetype if ending on {*'.pdf'*, *'.png'*, *'.svg'*}.

ax A matplotlib axes object. Only works if plotting a single component.

****kwds** Are passed to `matplotlib.pyplot.scatter()`.

Returns List of `Axes`

Examples

```
>>> import scanpy as sc
>>> adata = sc.datasets.pbmc68k_reduced()
>>> markers = ['C1QA', 'PSAP', 'CD79A', 'CD79B', 'CST3', 'LYZ']
>>> sc.pl.dotplot(adata, markers, groupby='bulk_labels', dendrogram=True)
```

Using var_names as dict:

```
>>> markers = {'T-cell': 'CD3D', 'B-cell': 'CD79A', 'myeloid': 'CST3'}
>>> sc.pl.dotplot(adata, markers, groupby='bulk_labels', dendrogram=True)
```

See also:

rank_genes_groups_dotplot: to plot marker genes identified using the `rank_genes_groups()` function.

episcanpy.pl.matrixplot

episcanpy.pl.**matrixplot**(*adata*, *var_names*, *groupby=None*, *use_raw=None*, *log=False*, *num_categories=7*, *figsize=None*, *dendrogram=False*, *gene_symbols=None*, *var_group_positions=None*, *var_group_labels=None*, *var_group_rotation=None*, *layer=None*, *standard_scale=None*, *swap_axes=False*, *show=None*, *save=None*, ***kwds*)

Creates a heatmap of the mean expression values per cluster of each var_names If groupby is not given, the matrixplot assumes that all data belongs to a single category.

Parameters

adata : AnnDataAnnData Annotated data matrix.

var_names : str, Sequence[str], MappingUnion[str, Sequence[str], Mapping[str, Union[str, Se *var_names* should be a valid subset of *adata.var_names*. If *var_names* is a mapping, then the key is used as label to group the values (see *var_group_labels*). The mapping values should be sequences of valid *adata.var_names*. In this case either coloring or 'brackets' are used for the grouping of var names depending on the plot. When *var_names* is a mapping, then the *var_group_labels* and *var_group_positions* are set.

groupby : str, NoneOptional[str] (default: None) The key of the observation grouping to consider.

use_raw : bool, NoneOptional[bool] (default: None) Use *raw* attribute of *adata* if present.

log : boolbool (default: False) Plot on logarithmic axis.

num_categories : intint (default: 7) Only used if groupby observation is not categorical. This value determines the number of groups into which the groupby observation should be subdivided.

figsize : Tuple[float, float], NoneOptional[Tuple[float, float]] (default: None) Figure size when *multi_panel=True*. Otherwise the *rcParam['figure.figsize]* value is used. Format is (width, height)

dendrogram : bool, strUnion[bool, str] (default: False) If True or a valid dendrogram key, a dendrogram based on the hierarchical clustering between the *groupby* categories is added. The dendrogram information is computed using `scanpy.tl.dendrogram()`. If *tl.dendrogram* has not been called previously the function is called with default parameters.

gene_symbols : str, NoneOptional[str] (default: None) Column name in *.var* DataFrame that stores gene symbols. By default *var_names* refer to the index column of the *.var* DataFrame. Setting this option allows alternative names to be used.

var_group_positions : Sequence[Tuple[int, int]], NoneOptional[Sequence[Tuple[int, int]]] (
Use this parameter to highlight groups of *var_names*. This will draw a 'bracket' or a color block between the given start and end positions. If the parameter *var_group_labels* is set, the corresponding labels are added on top/left. E.g. *var_group_positions=[(4,10)]* will add a bracket between the fourth *var_name* and the tenth *var_name*. By giving more positions, more brackets/color blocks are drawn.

var_group_labels : Sequence[str], NoneOptional[Sequence[str]] (default: None) Labels for each of the *var_group_positions* that want to be highlighted.

var_group_rotation : float, NoneOptional[float] (default: None) Label rotation degrees. By default, labels larger than 4 characters are rotated 90 degrees.

layer : str, NoneOptional[str] (default: None) Name of the AnnData object layer that wants to be plotted. By default adata.raw.X is plotted. If *use_raw=False* is set, then *adata.X* is plotted. If *layer* is set to a valid layer name, then the layer is plotted. *layer* takes precedence over *use_raw*.

standard_scale : Literal_[var, group], NoneOptional[Literal_[var, group]] (default: None) Whether or not to standardize that dimension between 0 and 1, meaning for each variable or group, subtract the minimum and divide each by its maximum.

show : bool, NoneOptional[bool] (default: None) Show the plot, do not return axis.

save : bool, str, NoneUnion[bool, str, None] (default: None) If *True* or a *str*, save the figure. A string is appended to the default filename. Infer the filetype if ending on {*'.pdf'*, *'.png'*, *'.svg'*}.

ax A matplotlib axes object. Only works if plotting a single component.

****kwds** Are passed to `matplotlib.pyplot.pcolor()`.

Returns List of `Axes`

Examples

```
>>> import scanpy as sc
>>> adata = sc.datasets.pbmc68k_reduced()
>>> markers = ['C1QA', 'PSAP', 'CD79A', 'CD79B', 'CST3', 'LYZ']
>>> sc.pl.matrixplot(adata, markers, groupby='bulk_labels', dendrogram=True)
```

Using var_names as dict:

```
>>> markers = {'T-cell': 'CD3D', 'B-cell': 'CD79A', 'myeloid': 'CST3'}
>>> sc.pl.matrixplot(adata, markers, groupby='bulk_labels', dendrogram=True)
```

See also:

rank_genes_groups_matrixplot() to plot marker genes identified using the `rank_genes_groups()` function.

episcanpy.pl.tracksplot

episcanpy.pl.**tracksplot**(*adata*, *var_names*, *groupby*, *use_raw=None*, *log=False*, *dendrogram=False*, *gene_symbols=None*, *var_group_positions=None*, *var_group_labels=None*, *layer=None*, *show=None*, *save=None*, *figsize=None*, ***kwds*)

In this type of plot each var_name is plotted as a filled line plot where the y values correspond to the var_name values and x is each of the cells. Best results are obtained when using raw counts that are not log.

groupby is required to sort and order the values using the respective group and should be a categorical value.

Parameters

adata : AnnDataAnnData Annotated data matrix.

var_names : str, Sequence[str], MappingUnion[str, Sequence[str], Mapping[str, Union[str, Se
var_names should be a valid subset of *adata.var_names*. If *var_names* is a mapping, then the key is used as label to group the values (see *var_group_labels*). The mapping values should be sequences of valid *adata.var_names*. In this case either coloring or 'brackets' are used for the grouping of var names depending on the plot. When *var_names* is a mapping, then the *var_group_labels* and *var_group_positions* are set.

groupby : strstr The key of the observation grouping to consider.

use_raw : bool, NoneOptional[bool] (default: None) Use *raw* attribute of *adata* if present.

log : boolbool (default: False) Plot on logarithmic axis.

num_categories Only used if groupby observation is not categorical. This value determines the number of groups into which the groupby observation should be subdivided.

figsize : Tuple[float, float], NoneOptional[Tuple[float, float]] (default: None) Figure size when *multi_panel=True*. Otherwise the *rcParam['figure.figsize]* value is used. Format is (width, height)

dendrogram : bool, strUnion[bool, str] (default: False) If True or a valid dendrogram key, a dendrogram based on the hierarchical clustering between the *groupby* categories is added. The dendrogram information is computed using `scanpy.tl.dendrogram()`. If *tl.dendrogram* has not been called previously the function is called with default parameters.

gene_symbols : str, NoneOptional[str] (default: None) Column name in *.var* DataFrame that stores gene symbols. By default *var_names* refer to the index column of the *.var* DataFrame. Setting this option allows alternative names to be used.

var_group_positions : Sequence[Tuple[int, int]], NoneOptional[Sequence[Tuple[int, int]]] (Use this parameter to highlight groups of *var_names*. This will draw a 'bracket' or a color block between the given start and end positions. If the parameter *var_group_labels* is set, the corresponding labels are added on top/left. E.g. *var_group_positions=[(4,10)]* will add a bracket between the fourth *var_name* and the tenth *var_name*. By giving more positions, more brackets/color blocks are drawn.

var_group_labels : Sequence[str], NoneOptional[Sequence[str]] (default: None) Labels for each of the *var_group_positions* that want to be highlighted.

var_group_rotation Label rotation degrees. By default, labels larger than 4 characters are rotated 90 degrees.

layer : str, NoneOptional[str] (default: None) Name of the AnnData object layer that wants to be plotted. By default adata.raw.X is plotted. If *use_raw=False* is set, then *adata.X* is plotted. If *layer* is set to a valid layer name, then the layer is plotted. *layer* takes precedence over *use_raw*.

show : bool, NoneOptional[bool] (default: None) Show the plot, do not return axis.

save : bool, str, NoneUnion[bool, str, None] (default: None) If *True* or a *str*, save the figure. A string is appended to the default filename. Infer the filetype if ending on {*'.pdf'*, *'.png'*, *'.svg'*}.

ax A matplotlib axes object. Only works if plotting a single component.

****kwds** Are passed to `heatmap()`.

Returns A list of `Axes`.

Examples

```
>>> import scanpy as sc
>>> adata = sc.datasets.pbmc68k_reduced()
>>> markers = ['C1QA', 'PSAP', 'CD79A', 'CD79B', 'CST3', 'LYZ']
>>> sc.pl.tracksplot(adata, markers, 'bulk_labels', dendrogram=True)
```

Using var_names as dict:

```
>>> markers = {'T-cell': 'CD3D', 'B-cell': 'CD79A', 'myeloid': 'CST3'}
>>> sc.pl.heatmap(adata, markers, groupby='bulk_labels', dendrogram=True)
```

See also:

pl.rank_genes_groups_tracksplot() to plot marker genes identified using the `rank_genes_groups()` function.

episcanpy.pl.dendrogram

episcanpy.pl.**dendrogram**(*adata*, *groupby*, *dendrogram_key=None*, *orientation='top'*, *remove_labels=False*, *show=None*, *save=None*)

Plots a dendrogram of the categories defined in *groupby*.

See `dendrogram()`.

Parameters

adata : AnnDataAnnData

groupby : strstr Categorical data column used to create the dendrogram

dendrogram_key : str, NoneOptional[str] (default: None) Key under with the dendrogram information was stored. By default the dendrogram information is stored under .uns['dendrogram_' + groupby].

orientation : strstr (default: 'top') Options are *top* (default), *bottom*, *left*, and *right*. Only when *show_correlation* is False.

remove_labels : boolbool (default: False)

{show_save_ax}

Returns matplotlib.axes.Axes

Examples

```
>>> import scanpy as sc
>>> adata = sc.datasets.pbmc68k_reduced()
>>> sc.tl.dendrogram(adata, 'bulk_labels')
>>> sc.pl.dendrogram(adata, 'bulk_labels')
```

episcanpy.pl.correlation_matrix

episcanpy.pl.**correlation_matrix**(*adata*, *groupby*, *show_correlation_numbers=False*, *dendrogram=None*, *figsize=None*, *show=None*, *save=None*, *ax=None*, ***kwds*)

Plots the correlation matrix computed as part of *sc.tl.dendrogram*.

Parameters

adata : AnnDataAnnData

groupby : strstr Categorical data column used to create the dendrogram

show_correlation_numbers : boolbool (default: False) If *show_correlation* is True, plot the correlation number on top of each cell.

dendrogram : bool, str, NoneUnion[bool, str, None] (default: None) If True or a valid dendrogram key, a dendrogram based on the hierarchical clustering between the *groupby* categories is added. The dendrogram information is computed using scanpy.tl.dendrogram(). If *tl.dendrogram* has not been called previously the function is called with default parameters.

figsize : Tuple[float, float], NoneOptional[Tuple[float, float]] (default: None) By default a figure size that aims to produce a squared correlation matrix plot is used. Format is (width, height)

{show_save_ax}

****kwds** Only if *show_correlation* is True: Are passed to matplotlib.pyplot.pcolormesh() when plotting the correlation heatmap. Useful values to pas are *vmax*, *vmin* and *cmap*.

Return type Axes, List[Axes]Union[Axes, List[Axes]]

Returns

Examples

```
>>> import scanpy as sc
>>> adata = sc.datasets.pbmc68k_reduced()
>>> sc.tl.dendrogram(adata, 'bulk_labels')
>>> sc.pl.correlation(adata, 'bulk_labels')
```

episcanpy.pl.prct_overlap

episcanpy.pl.**prct_overlap**(*adata*, *key_1*, *key_2*, *norm=False*, *ax_norm='row'*, *sort_index=False*)

% or cell count corresponding to the overlap of different cell types between 2 set of annotations/clusters.

Parameters

adata : *AnnData objet*

key_1 : *observational key corresponding to one cell division/ one set of clusters*

key_2 : *bservational key corresponding to one cell division/ one set of clusters*

norm : *normalise the ratio to the cell numbers given the total number of cells pe*

in key_1 : *cluster*

Returns Table containing the ratio of cells within a cluster

episcanpy.pl.overlap_heatmap

episcanpy.pl.**overlap_heatmap**(*adata*, *key_1*, *key_2*, *color='Blues'*, *norm=False*, *ax_norm='row'*, *sort_index=False*)

Heatmap of the cluster correspondance between 2 set of annaotations.

Parameters

adata : *AnnData objet*

key_1 : *observational key corresponding to one cell division/ one set of clusters*

key_2 : *bservational key corresponding to one cell division/ one set of clusters*

norm : *normalisation on the total number of cells per row/column or not.*

ax_norm : *normalistion (if norm=True) on row or col*

colors : *gradient displayed in the heatmap. "YlGnBu", "BuPu", "Greens", "Blues"*

Returns Heatmap.

episcanpy.pl.cluster_composition

episcanpy.pl.**cluster_composition**(*adata*, *cluster*, *condition*, *xlabel='cell cluster'*, *ylabel='cell count'*, *title=None*, *save=False*)

episcanpy.pl.silhouette

episcanpy.pl.**silhouette**(*adata_name*, *cluster_annot*, *key=None*, *xlabel=None*, *ylabel=None*, *title=None*, *size='large'*, *name_cluster=True*, *name_cluster_pos='left'*, *palette=None*, *save=None*)

Plot the product of tl.silhouette as a silhouette plot

Parameters

adata_name : *AnnData object*

cluster_annot : *observational variable corresponding to a cell clustering*

key : *specify name of precomputed silhouette scores if not standard*

Returns Silhouette plot

episcanpy.pl.silhouette_tot

episcanpy.pl.**silhouette_tot**(*adata_name*, *cluster_annot*, *value='X_pca'*, *metric='euclidean'*, *xlabel=None*, *ylabel=None*, *title=None*, *size='large'*, *name_cluster=True*, *name_cluster_pos='left'*, *palette=None*, *save=None*, *key_added=None*)

Both compute silhouette scores and plot it.

It computes the general silhouette score as well as a silhouette score for every cell according to the cell cluster assigned to it.

Parameters

adata_name : *AnnData object*

cluster_annot : *observational variable corresponding to a cell clustering*

value : *measure used to build the silhouette plot (X_pca, X_tsne, X_umap)*

metric : *'euclidean'*

key_added : *key to save the computed silhouette scores*

Returns

general silhouette score in 'uns' of the AnnData object individual silhouette scores in 'obs' of the AnnData object

Silhouette plot

Credit to sklearn script : https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html#sphx-glr-auto-examples-cluster-plot-kmeans-silhouette-analysis-py return score and silhouette plot. Still some work to do to finish the function. size=None but you can put 'large' if you want a bigger default figure size

1.6 References

BIBLIOGRAPHY

[Angerer16] Angerer *et al.* (2016), *destiny – diffusion maps for large-scale single-cell data in R*, Bioinformatics.

[Cusanovich18] Cusanovich, D. A. et al. A Single-Cell Atlas of In Vivo Mammalian Chromatin Accessibility. Cell 174, 1309–1324.e18 (2018).

[Luo17] Luo, C. et al. Single-cell methylomes identify neuronal subtypes and regulatory elements in mammalian cortex. Science 357, 600–604 (2017).

[Wolf18] Wolf, F. A., Angerer, P. & Theis, F. J. SCANPY: large-scale single-cell gene expression data analysis. Genome Biol. 19, 15 (2018).

PYTHON MODULE INDEX

e

INDEX

S

T

U

V