celloracle

Release 0.6.9

Samantha Morris Lab

Jul 16, 2021

CONTENTS

CellOracle is a python library for the in silico gene perturbation analysis using single-cell omics data and Gene Regulatory Network models.

Source code is available at celloracle GitHub repository

For more information, please read our bioarxiv preprint: CellOracle: Dissecting cell identity via network inference and in silico gene perturbation

Note:

Documentation is also available as a pdf file.

`pdf documentation`

Warning: CellOracle is still under development. It is a beta version. Functions in this package may change in the future release.

CHAPTER

ONE

NEWS

- 7/16/2021: We overhauled our documentation and tutorial codes. Please re-download tutorial notebooks if you have old one. Also, we are updating CellOracle frequently. Please install the latest version of CellOracle if you have an old version. The latest version is 0.7.1.

CHAPTER

TWO

CONTENTS

2.1 Installation

CellOracle uses several python libraries and R libraries. Please follow this guide below to install CellOracle and its dependent software.

2.1.1 Docker image

- Pre-built docker image is available through Docker Hub .

```
docker push kenjikamimoto126/celloracle_ubuntu:latest
```

- This docker image was built based on Ubuntu 20.04.
- Python dependent packages and celloracle are installed under an anaconda environment, celloracle_env. This environment will be activated automatically when you log in.
- R dependent libraries for network analysis are installed. Also, Seurat V3, Monocle3, and Cicero are installed.
- After logging in, the user switches from the root user to the following user. Username: user. Password: pass.

2.1.2 Install CellOracle

System Requirements

- Operating system: macOS or Linux are highly recommended. CellOraclewas developed and tested in Linux and macOS.
- We found that the celloracle calculation may be EXTREMELY SLOW under an environment of Windows Subsystem for Linux (WSL). We do not recommend using WSL.
- While you can install CellOracle in Windows OS, please do so at your own risk and responsibility. We DO NOT provide any support for the use in the Windows OS.
- Memory: 8 G byte or more. Memory usage also depends on your scRNA-seq data. Especially in silico perturbation requires large amount of memory.
- CPU: Core i5 or better processor. GRN inference supports multicore calculation. Higher number of CPU cores enables fast calculation.

Python Requirements

- CellOracle was developed with python 3.6. We do not support python 2.7x or python <=3.5.
- Please install all dependent libraries before installing CellOracle according to the instructions below.
- CellOracle is still a beta version and it is not available through PyPI or anaconda distribution yet. Please install CellOracle from our GitHub repository according to the instruction below.

Quick CellOracle installation using pip

You can install CellOracle and its all dependencies by the following command.

```
pip install git+https://github.com/morris-lab/CellOracle.git
```

You may have an error in the installation process of CellOracle dependent libraries. If you have an error, please install dependent libraries stepwise.

Python dependent library installation troubleshooting

0. (Optional) Make a new conda environment

This step is optional, but we recommend installing CellOracle in an independent conda environment to avoid dependent software conflicts. Please make a new python environment for celloracle and install dependent libraries in it.

```
conda create -n celloracle_env python=3.6
conda activate celloracle_env
```

1. Add conda channels

Installation of some libraries requires non-default anaconda channels. Please add the channels below. Instead, you can explicitly enter the channel when you install a library.

```
conda config --add channels defaults
conda config --add channels bioconda
conda config --add channels conda-forge
```

2. Install velocyto

Please install velocyto with the following commands or the author's instruction .

```
conda install numpy scipy cython numba matplotlib scikit-learn h5py>=3.1.0 click␣
↪pysam llvm louvain
```

Then

```
pip install velocyto
```

It was reported that some compile errors might occur during the installation of velocyto on MacOS. Various errors were reported, and you need to find the best solution depending on your error. You may find the solution with these links below.

- Solution 1: Install Xcode. Please try this first.
- Solution 2: Install macOS_SDK_headers. This solution is needed in addition to Solution-1 if your OS is macOS Mojave.
- Solution 3. This is the solution reported by a CellOracle user. Thank you very much!
- Other solutions on Velocyto GitHub issue page

3. Install scanpy

Please install scanpy with the following commands or the author's instruction .

```
conda install scanpy
```

4. Install other python libraries

Please install other python libraries below with the following commands.

```
conda install goatools pyarrow tqdm joblib jupyter gimmemotifs==0.14.4 genomepy==0.8.4
```

5. install celloracle

```
pip install git+https://github.com/morris-lab/CellOracle.git
```

R requirements

CellOracle uses R libraries to calculate network graph score. Please install R (>=3.5) and R libraries below.

```
install.packages("igraph")
install.packages("rnetcarto")
install.packages("linkcomm")
```

If you have an error when installing these R libraries above, please look at the troubleshooting tips below.

R dependent library installation troubleshooting

igraph

Please install `igraph` with the following r-script or the author's instruction .

In R console,

```
install.packages("igraph")
```

If you get an error during installation, please check compilers. This GitHub issue page is helpful.

linkcomm

Please install `linkcomm` with the following r-script or the author's instruction .

In R console,

```
install.packages("linkcomm")
```

rnetcarto

Please install `rnetcarto` with the following r-script or the author's instruction . `rnetcarto` requires the GNU scientific libraries .

If you use ubuntu, you can install the GNU scientific libraries as follows.

```
sudo apt-get install libgsl-dev
```

In R console,

```
install.packages("rnetcarto")
```

Check installation

Check python library installation status

You can check the installed library version as follows.

In python console,

```
import celloracle as co
co.check_python_requirements()
```

Check R library installation status

Please make sure that all R libraries are installed using the following function.

```
import celloracle as co
co.test_R_libraries_installation()
```

The following message will be shown when all R libraries are appropriately installed.

R path: /usr/lib/R/bin/R
checking R library installation: igraph -> OK
checking R library installation: linkcomm -> OK
checking R library installation: rnetcarto -> OK

The first line above is your R path. If you want to use another R program installed at a different place, please set a new R path with the following command.

```
co.network_analysis.set_R_path("ENTER YOUR R PATH HERE")
```

Optional R libraries for input data preparation

We provide many working examples for input data preparation. These R packages below are not in the part of the CellOracle library itself and not necessary. However you can use them in the input data preparation step if you want. Please install them on demand. If you want to try CellOracle main tutorials, networkanalysis and simulation, you DO NOT need to install the libraries below.

- Seurat
- Cicero

2.2 Tutorial

This tutorial aims to introduce how to use CellOracle functions using the demo dataset. Once you get used to CellOracle codes, please replace demo data with your data to investigate it.

2.2.1 What the tutorial covers

1. Main celloracle analysis

- *GRN model construction and Network analysis*: This notebook introduces how to construct sample-specific GRN models. It also contains examples of network analysis with graph theory.
- *in silico gene perturbation with GRNs* : This notebook performs in silico gene perturbation analysis using GRN models.

Note: Demo dataset is available in the tutorial notebooks above. You can try CellOracle even if you do not have any data.

---

2. How to prepare input data

We recommend getting started with CellOracle using demo dataset. Please get used to CellOracle analysis with them first. When you want to apply CellOracle to your scRNA-seq or scATAC dataset, please refer to the following tutorials to know how to prepare input data.

- *scRNA-seq data preparation*: This notebook explains preprocessing steps for scRNA-seq data.
- *Base GRN input data preparation*: This tutorial explains how to prepare input data for TF motif scan.
- *Transcription factor binding motif scan*: This tutorial describes the TF motif scan pipeline for base-GRN construction.

Warning: In the input data preparation, we introduce how to prepare input data using some other libraries. But **the input data preparation notebook is NOT CellOracle analysis itself**, and we just provide an example how to leverage pre-existing tools **to prepare input data**. CellOracle is not just pipeline that is made of pre-existing tools.

2.2.2 Prerequisites

- This tutorial assumes that you have some Python programming experience. In particular, we assume you are familiar with Python data science libraries: jupyter, pandas, and matplotlib.
- Also, this tutorial assume that you are familiar with basic scRNA-seq data analysis. In particular, we assume you have some experience of scRNA-seq analysis using Scanpy and Anndata , which is a python toolkit for single-cell analysis. You can use scRNA-seq data processed with Seurat . But the Seurat data need to be converted into Anndata format in advance to CellOracle analysis. See this tutorial for detail.
- CellOracle provides pre-build base-GRN, and it is not necessary to construct custom base-GRN. But if you want to construct custom base-GRN from your scATAC-seq data, we recommend using Cicero . In this case, please get used to Cicero, basic scATAC-seq data analysis, and TF motif analysis in advance to start constructing base-GRN.

2.2.3 Code and data availability

- All jupyter notebook files are available here .
- Also, we provide link for the notebook in each section.
- You can download demo input data using the notebooks.
- We provide intermediate files. You can start at any section.

2.2.4 Getting started

If you run CellOracle for the first time, please start with the *GRN model construction and Network analysis*. And then, please proceed to *in silico gene perturbation with GRNs*. We provide demo scRNA-seq dataset and base-GRN data as follows. You can load these data using the CellOracle data loading function.

- scRNA-seq data: Hematopoiesis dataset published by Paul et al (2015) .
- Base-GRN: Base-GRN generated from Mouse sci-ATAC-seq atlas dataset .

You can easily start CellOracle analysis with this dataset. You can reproduce hematopoiesis network analysis and perturbation simulation results that are shown in our bioarxiv preprint .

2.2.5 Index

GRN model construction and Network analysis

GRN model construction and Network analysis

Please download notebooks from here . Or please click below to view the content.

Overview

This notebook describes how to construct GRN models. Please read our paper first to know about the CellOracle algorithm.

Notebook file

Notebook file is available at CellOracle GitHub. https://github.com/morris-lab/CellOracle/blob/master/docs/notebooks/04_Network_analysis/Network_analysis_with_Paul_etal_2015_data.ipynb

Data

CellOracle uses two input data below for the GRN model construction.

- **Input data1: scRNA-seq data**. Please look at the previous section to know the scRNA-seq data preprocessing method. https://morris-lab.github.io/CellOracle.documentation/tutorials/scrnaprocess.html
- **Input data2: Base-GRN**. Base-GRN is a binary matrix (or list) that represents the TF-target gene connection. Please look at our paper to know the concept of base-GRN.
- CellOracle typically uses base-GRN constructed from scATAC-seq. If you want to create custom base-GRN from your data, please look at another notebook on how to get base-GRN from your scATAC-seq data. https://morris-lab.github.io/CellOracle.documentation/tutorials/base_grn.html
- If you do not have any scATAC-seq data that correspond / similar to the cell type of the scRNA-seq data, please use pre-built base-GRN.
- We provide multiple options for pre-built base-GRN. For mouse analysis, we recommend using base-GRN constructed from the mouse sciATAC-seq atlas dataset. It includes various tissue and various cell types. Another option is base-GRN constructed from promoter sequence. We provide promoter base-GRN for ten species.

What you can do

After constructing the CellOracle GRN model, you can do two analyses.

1. **in silico TF perturbation** to simulate cell identity shift. CellOracle uses the GRN model to simulate cell identity shift in response to TF perturbation. For this analysis, you need to construct GRN models in this notebook first.
2. **Network analysis** using graph theory. You can analyze the GRN model itself. We provide several functions for Network analysis using graph theory.

- CellOracle construct cluster-wise GRN model. You can compare the GRN model structure between clusters. By comparing GRN models, you can investigate the cell type-specific GRN configuration and rewiring process of this GRN.
- You can export the network models. You can analyze the GRN model using any method you like.

Custom data class / object

In this notebook, CellOracle uses two custom classes, `Oracle` and `Links`.

- `Oracle` is the main class in the CellOracle package. It will do almost all calculations of GRN model construction and TF perturbation simulation. `Oracle` will do the following calculation sequentially.

1. Import scRNA-sequence data. Please look at another notebook to learn preprocessing method.
2. Import base-GRN data.
3. scRNA-seq data processing.
4. GRN model construction.
5. in silico petrurbation. We will describe how to do it in the following notebook.

- `Links` is a class to store GRN data. Also, it has many functions for network analysis and visualization.

0. Import libraries

[1]:
```
# 0. Import

import os
import sys

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

[2]:
```
import celloracle as co
co.__version__
```

[2]:
```
'0.6.17'
```

[3]:
```
# visualization settings
%config InlineBackend.figure_format = 'retina'
%matplotlib inline

plt.rcParams['figure.figsize'] = [6, 4.5]
plt.rcParams["savefig.dpi"] = 300
```

Celloracle uses some R libraries in network analysis. Please make sure that all dependent R libraries are installed on your computer. You can test the installation with the following command.

[4]:
```
co.test_R_libraries_installation()
```

```
R path: /usr/bin/R
checking R library installation: igraph -> OK
checking R library installation: linkcomm -> OK
checking R library installation: rnetcarto -> OK
```

[5]:
```
save_folder = "figures"
os.makedirs(save_folder, exist_ok=True)
```

1. Load data

Please refer to the previous notebook in the tutorial for an example of how to process scRNA-seq data. https://morris-lab.github.io/CellOracle.documentation/tutorials/scrnaprocess.html

We need scRNA-seq data as anndata.

> **This CellOracle tutorial notebook assume the user have a basic knoledge and experience of scRNA-seq analysis with scanpy and anndata.** This notebook do not intend to give introductory knowledge about scanpy and anndata. If you are not familiar with them, please look at the documentation and tutorials of annata (https://anndata.readthedocs.io/en/stable/) and Scanpy (https://scanpy.readthedocs.io/en/stable/).

[6]:
```
# Load data. !!Replace the data path below when you use another data.
# adata = sc.read_h5ad("DATAPATH")

# Here, we will use a hematopoiesis data by Paul 2015.
# You can load preprocessed data using a celloracle function as follows.
adata = co.data.load_Paul2015_data()
adata
```

[6]:
```
AnnData object with n_obs × n_vars = 2671 × 1999
    obs: 'paul15_clusters', 'n_counts_all', 'n_counts', 'louvain', 'cell_type',
↪'louvain_annot', 'dpt_pseudotime'
    var: 'n_counts'
    uns: 'cell_type_colors', 'diffmap_evals', 'draw_graph', 'iroot', 'louvain',
↪'louvain_annot_colors', 'louvain_colors', 'louvain_sizes', 'neighbors', 'paga',
↪'paul15_clusters_colors', 'pca'
    obsm: 'X_diffmap', 'X_draw_graph_fa', 'X_pca'
    varm: 'PCs'
    layers: 'raw_count'
    obsp: 'connectivities', 'distances'
```

If your scRNA-seq data includes more than 20-30K cells, we recommend doing downsampling. It is because the later simulation process will require large amount of memory if you have large data.

Also, please pay attention to the number of genes. If you are following the instruction in the previous tutorial notebook, the scRNA-seq data should include only top 2~3K variable genes. If you have more than 3K genes, it might cause problems in the later steps.

[7]:
```
print(f"Cell number is :{adata.shape[0]}")
print(f"Gene number is :{adata.shape[1]}")
```

```
Cell number is :2671
Gene number is :1999
```

[8]:
```
# Random downsampling into 30K cells if the anndata include more than 30 K cells.
n_cells_downsample = 30000

if adata.shape[0] > n_cells_downsample:
    # Let's dowmsample into 30K cells
    sc.pp.subsample(adata, n_obs=n_cells_downsample, random_state=123)
```

[9]:
```
print(f"Cell number is :{adata.shape[0]}")
```

```
Cell number is :2671
```

For the GRN inference, celloracle needs base-GRN. - There are several ways to make base-GRN. We can typically generate TF information from scATAC-seq data or bulk ATAC-seq data. Please refer to the first step of the tutorial for

the details of this process. https://morris-lab.github.io/CellOracle.documentation/tutorials/base_grn.html

- If you do not have your scATAC-seq data, you can use some built-in base-GRN data.
- Base-GRN made from mouse sci-ATAC-seq atlas dataset: The built-in base-GRN was made from various tissue/cell-types (http://atlas.gs.washington.edu/mouse-atac/). We recommend using this for mouse scRNA-seq data. Please load this data as follows.

```
base_GRN = co.data.load_mouse_scATAC_atlas_base_GRN()
```

- Promoter base-GRN: We provide base-GRN made from promoter DNA-sequencing for ten species. You can load this data as follos.
- For Human: `base_GRN = co.data.load_human_promoter_base_GRN()`

[10]:
```
# Load TF info which was made from mouse cell atlas dataset.
base_GRN = co.data.load_mouse_scATAC_atlas_base_GRN()

# Check data
base_GRN.head()
```

[10]:

	peak_id	gene_short_name	9430076c15rik	Ac002126.6	\
0	chr10_100050979_100052296	4930430F08Rik	0.0	0.0	
1	chr10_101006922_101007748	SNORA17	0.0	0.0	
2	chr10_101144061_101145000	Mgat4c	0.0	0.0	
3	chr10_10148873_10149183	9130014G24Rik	0.0	0.0	
4	chr10_10149425_10149815	9130014G24Rik	0.0	0.0	

	Ac012531.1	Ac226150.2	Afp	Ahr	Ahrr	Aire	...	Znf784	Znf8	Znf816	\
0	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
1	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
2	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
3	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
4	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	

	Znf85	Zscan10	Zscan16	Zscan22	Zscan26	Zscan31	Zscan4
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	1.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	1.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0

[5 rows x 1095 columns]

2. Initiate Oracle object

We can use Oracle for the data preprocessing and GRN inference steps. The Oracle object stores all of the necessary information and does the calculations with its internal functions. We instantiate an Oracle object, then input the gene expression data (anndata) and a TFinfo into the Oracle object.

[19]:
```
# Instantiate Oracle object
oracle = co.Oracle()
```

For the celloracle analysis, the anndata shoud include (1) gene expression count, (2) clustering information, (3) trajectory (dimensional reduction embeddings) data. Please refer to another notebook for more information on anndata preprocessing.

When you load a scRNA-seq data, please enter **the name of clustering data** and **dimensional reduction data.** - The clustering data should be to be stored in the attribute of `obs` in the anndata. > You can check it by the following

command. > > `adata.obs.columns`

- Dimensional reduction data suppose to be stored in the attribute of "obsm" in the anndata. > You can check it by the following command. > > `adata.obsm.keys()`

[20]:
```
# Show data name in anndata
print("metadata columns :", list(adata.obs.columns))
print("dimensional reduction: ", list(adata.obsm.keys()))
```

```
metadata columns : ['paul15_clusters', 'n_counts_all', 'n_counts', 'louvain', 'cell_
↪type', 'louvain_annot', 'dpt_pseudotime']
dimensional reduction:  ['X_diffmap', 'X_draw_graph_fa', 'X_pca']
```

[21]:
```
# In this notebook, we use raw mRNA count as an input of Oracle object.
adata.X = adata.layers["raw_count"].copy()

# Instantiate Oracle object.
oracle.import_anndata_as_raw_count(adata=adata,
                                   cluster_column_name="louvain_annot",
                                   embedding_name="X_draw_graph_fa")
```

[22]:
```
# You can load TF info dataframe with the following code.
oracle.import_TF_data(TF_info_matrix=base_GRN)

# Alternatively, if you saved the informmation as a dictionary, you can use the code␣
↪below.
# oracle.import_TF_data(TFdict=TFinfo_dictionary)
```

We can add additional TF-target gene pair manually.

For example, if there is a study or database that includes specific TF-target pairs, you can use such information in the following way.

2.3.1. Make dictionary

Here, we will introduce how to manually add TF-target gene pair data.

As an example, we will use TF binding data that was published in supplemental table 4 in the paper. (http://doi.org/10.1016/j.cell.2015.11.013).

You can dowmload this file by running the following command. If it fails, please download manually. https://raw.githubusercontent.com/morris-lab/CellOracle/master/docs/demo_data/TF_data_in_Paul15.csv

In order to import TF data into the Oracle object, we need to convert them into a python dictionary. The dictionary keys is a target gene, and dictionary value is a list of regulatory candidate TFs.

[3]:
```
# Download file.
!wget https://raw.githubusercontent.com/morris-lab/CellOracle/master/docs/demo_data/
↪TF_data_in_Paul15.csv

# If you are using macOS, please try the following command.
#!curl -O https://raw.githubusercontent.com/morris-lab/CellOracle/master/docs/demo_
↪data/TF_data_in_Paul15.csv
```

```
--2021-06-09 15:13:52--  https://raw.githubusercontent.com/morris-lab/CellOracle/
↪master/docs/notebooks/04_Network_analysis/TF_data_in_Paul15.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.110.133,␣
↪185.199.109.133, 185.199.108.133, ...
```

(continues on next page)

(continued from previous page)

```
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.110.133|:
↪443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1768 (1.7K) [text/plain]
Saving to: 'TF_data_in_Paul15.csv'

TF_data_in_Paul15.c 100%[===================>]   1.73K  --.-KB/s    in 0s

2021-06-09 15:13:52 (11.8 MB/s) - 'TF_data_in_Paul15.csv' saved [1768/1768]
```

[23]:
```
# We have TF and its target gene information. This is from a supplemental Fig of Paul␣
↪et. al, (2015).
Paul_15_data = pd.read_csv("TF_data_in_Paul15.csv")
Paul_15_data
```

[23]:

	TF	Target_genes
0	Cebpa	Abcb1b, Acot1, C3, Cnpy3, Dhrs7, Dtx4, Edem2, ...
1	Irf8	Abcd1, Aif1, BC017643, Cbl, Ccdc109b, Ccl6, d6...
2	Irf8	1100001G20Rik, 4732418C07Rik, 9230105E10Rik, A...
3	Klf1	2010011I20Rik, 5730469M10Rik, Acsl6, Add2, Ank...
4	Sfpi1	0910001L09Rik, 2310014H01Rik, 4632428N05Rik, A...

[24]:
```
# Make dictionary: dictionary Key is TF, dictionary Value is list of target genes
TF_to_TG_dictionary = {}

for TF, TGs in zip(Paul_15_data.TF, Paul_15_data.Target_genes):
    # convert target gene to list
    TG_list = TGs.replace(" ", "").split(",")
    # store target gene list in a dictionary
    TF_to_TG_dictionary[TF] = TG_list

# We have to make a dictionary, in which a Key is Target gene and value is TF.
# We invert the dictionary above using a utility function in celloracle.
TG_to_TF_dictionary = co.utility.inverse_dictionary(TF_to_TG_dictionary)
```

```
HBox(children=(FloatProgress(value=0.0, max=178.0), HTML(value='')))
```

2.3.2. Add TF informatio dictionary into the oracle object

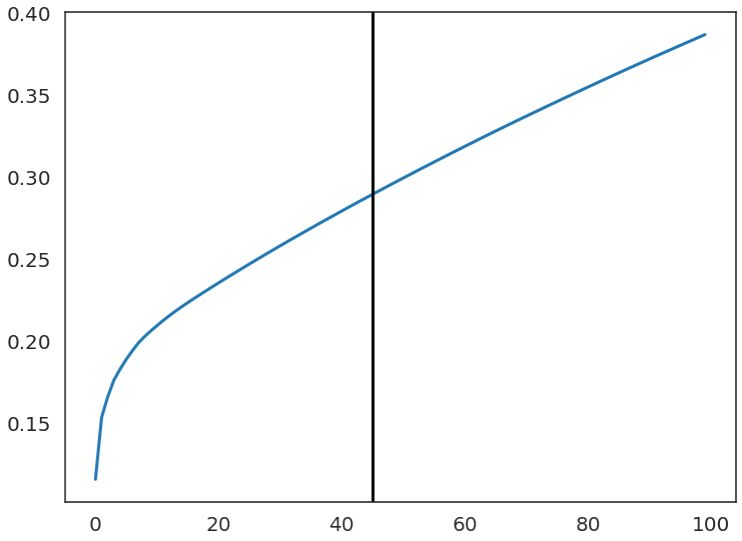
[25]:
```
# Add TF information
oracle.addTFinfo_dictionary(TG_to_TF_dictionary)
```

3. Knn imputation

Celloracle uses the same strategy as velocyto for visualizing cell transitions. This process requires KNN imputation in advance.

For the KNN imputation, we need PCA and PC selection first.

[26]:
```
# Perform PCA
oracle.perform_PCA()

# Select important PCs
plt.plot(np.cumsum(oracle.pca.explained_variance_ratio_)[:100])
n_comps = np.where(np.diff(np.diff(np.cumsum(oracle.pca.explained_variance_ratio_))>0.
↪002))[0][0]
plt.axvline(n_comps, c="k")
print(n_comps)
n_comps = min(n_comps, 50)
```



Estimate the optimal number of nearest neighbors for KNN imputation.

[27]:
```
n_cell = oracle.adata.shape[0]
print(f"cell number is :{n_cell}")
```

```
cell number is :2671
```

[28]:
```
k = int(0.025*n_cell)
print(f"Auto-selected k is :{k}")
```

```
Auto-selected k is :66
```

[29]:
```
oracle.knn_imputation(n_pca_dims=n_comps, k=k, balanced=True, b_sight=k*8,
                      b_maxl=k*4, n_jobs=4)
```

4. Save and Load.

You can save `Oracle` object using `Oracle.to_hdf5(FILE_NAME.celloracle.oracle)`.

Pleasae use `co.load_hdf5(FILE_NAME.celloracle.oracle)` to load the saved file.

[ ]:
```
# Save oracle object.
oracle.to_hdf5("Paul_15_data.celloracle.oracle")
```

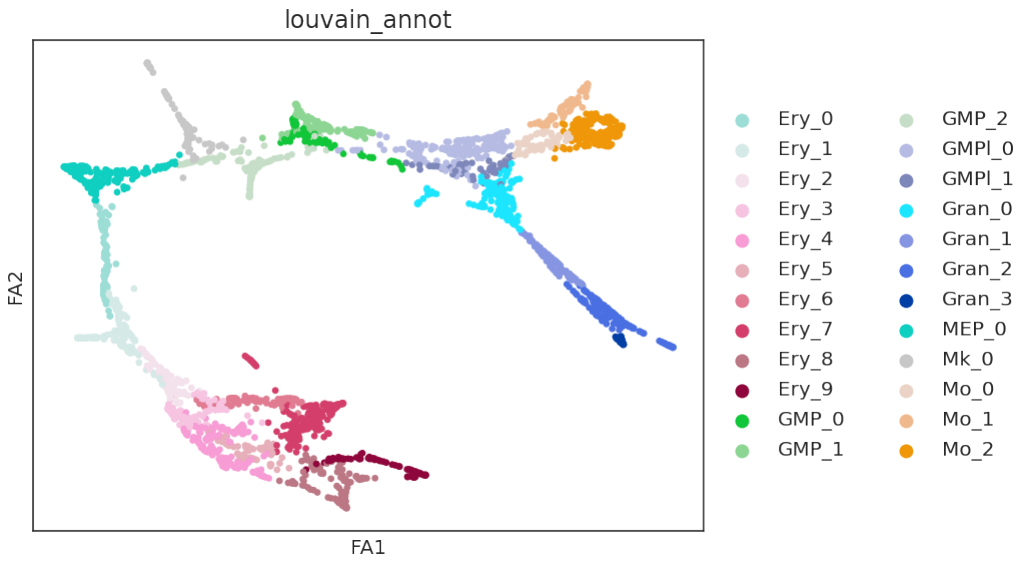
[ ]:
```
# Load file.
oracle = co.load_hdf5("Paul_15_data.celloracle.oracle")
```

5. GRN calculation

The next step is constructing a cluster-specific GRN for all clusters.

- You can calculate GRNs with the `get_links` function, and the function returns GRNs as a `Links` object. The `Links` object stores inferred GRNs and the corresponding metadata. You can do network analysis with the `Links` object.
- The GRN will be calculated for each cluster/sub-group. In the example below, we construct GRN for each unit of the "louvain_annot" clustering.

[32]:
```
# check data
sc.pl.draw_graph(oracle.adata, color="louvain_annot")
```



[ ]:
```
%%time
# Calculate GRN for each population in "louvain_annot" clustering unit.
# This step may take long time.(~30 minutes)
links = oracle.get_links(cluster_name_for_GRN_unit="louvain_annot", alpha=10,
                         verbose_level=10, test_mode=False)
```

Although celloracle has many functions for network analysis, you can analyze GRNs by hand if you choose. The raw GRN data is stored as a dictionary of dataframe in the attribute of `links_dict`.

For example, you can get the GRN for the "Ery_0" cluster with the following commands.

[34]:
```
links.links_dict.keys()
```

[34]:
```
dict_keys(['Ery_0', 'Ery_1', 'Ery_2', 'Ery_3', 'Ery_4', 'Ery_5', 'Ery_6', 'Ery_7',
↪'Ery_8', 'Ery_9', 'GMP_0', 'GMP_1', 'GMP_2', 'GMPl_0', 'GMPl_1', 'Gran_0', 'Gran_1',
↪ 'Gran_2', 'Gran_3', 'MEP_0', 'Mk_0', 'Mo_0', 'Mo_1', 'Mo_2'])
```

[35]:
```
links.links_dict["Ery_0"]
```

[35]:

	source	target	coef_mean	coef_abs	p	-logp
0	Nfe2	0610007L01Rik	0.003554	0.003554	1.443188e-02	1.840677
1	Id2	0610007L01Rik	0.001891	0.001891	1.711869e-01	0.766530
2	Zbtb1	0610007L01Rik	-0.002724	0.002724	6.442622e-03	2.190937
3	Elf1	0610007L01Rik	0.006480	0.006480	1.326915e-06	5.877157
4	Hnf4a	0610007L01Rik	0.001538	0.001538	3.538728e-01	0.451153
...	...	...	...	...	...	...
74467	Stat3	Zyx	0.022902	0.022902	7.967776e-09	8.098663
74468	Ets1	Zyx	0.015078	0.015078	2.280509e-05	4.641968
74469	Nfkb1	Zyx	0.015030	0.015030	3.214934e-07	6.492828
74470	Fli1	Zyx	0.012840	0.012840	6.909677e-05	4.160542
74471	Klf4	Zyx	-0.003232	0.003232	1.250538e-05	4.902903

[74472 rows x 6 columns]

You can export the file as follows.

[36]:
```
# Set cluster name
cluster = "Ery_0"

# Save as csv
#links.links_dict[cluster].to_csv(f"raw_GRN_for_{cluster}.csv")
```

The links object has a color information in an attribute, `palette`. This information is used for the visualization

The sample will be visualized in that order. Here we can change color and order.

[43]:
```
# Show the contents of pallete
links.palette
```

[43]:

	palette
Ery_0	#9CDED6
Ery_1	#D5EAE7
Ery_2	#F3E1EB
Ery_3	#F6C4E1
Ery_4	#F79CD4
Ery_5	#E6AFB9
Ery_6	#E07B91
Ery_7	#D33F6A
Ery_8	#BB7784
Ery_9	#8E063B
GMP_0	#11C638
GMP_1	#8DD593
GMP_2	#C6DEC7
GMPl_0	#B5BBE3
GMPl_1	#7D87B9
Gran_0	#1CE6FF
Gran_1	#8595E1
Gran_2	#4A6FE3
Gran_3	#023FA5
MEP_0	#0FCFC0

(continues on next page)

(continued from previous page)

```
Mk_0    #C7C7C7
Mo_0    #EAD3C6
Mo_1    #F0B98D
Mo_2    #EF9708
```

[46]:
```
# Change the order of pallete
order = ['MEP_0', 'Mk_0', 'Ery_0',
         'Ery_1', 'Ery_2', 'Ery_3', 'Ery_4', 'Ery_5', 'Ery_6', 'Ery_7', 'Ery_8', 'Ery_
↪9',
         'GMP_0', 'GMP_1', 'GMP_2', 'GMPl_0', 'GMPl_1',
         'Mo_0', 'Mo_1', 'Mo_2',
         'Gran_0', 'Gran_1', 'Gran_2', 'Gran_3']
links.palette = links.palette.loc[order]
links.palette
```

[46]:
```
        palette
MEP_0   #0FCFC0
Mk_0    #C7C7C7
Ery_0   #9CDED6
Ery_1   #D5EAE7
Ery_2   #F3E1EB
Ery_3   #F6C4E1
Ery_4   #F79CD4
Ery_5   #E6AFB9
Ery_6   #E07B91
Ery_7   #D33F6A
Ery_8   #BB7784
Ery_9   #8E063B
GMP_0   #11C638
GMP_1   #8DD593
GMP_2   #C6DEC7
GMPl_0  #B5BBE3
GMPl_1  #7D87B9
Mo_0    #EAD3C6
Mo_1    #F0B98D
Mo_2    #EF9708
Gran_0  #1CE6FF
Gran_1  #8595E1
Gran_2  #4A6FE3
Gran_3  #023FA5
```

6. Network preprocessing

Using base-GRN, CellOracle constructs GRN models as lits of a directed edge between TF and its target gene. We need to remove weak edges or insignificant edges before doing network analysis.

We filter the network edges as follows.

1. Remove uncertain network edges based on the p-value.
2. Remove weak network edge. In this tutorial, we pick up the top 2000 edges by edge strength.

The raw network data is stored as an attribute, `links_dict`, while filtered network data is stored in `filtered_links`.

[5]:
```
links = co.data.load_tutorial_links_object()
```

[16]:
```
links.filter_links(p=0.001, weight="coef_abs", threshold_number=2000)
```

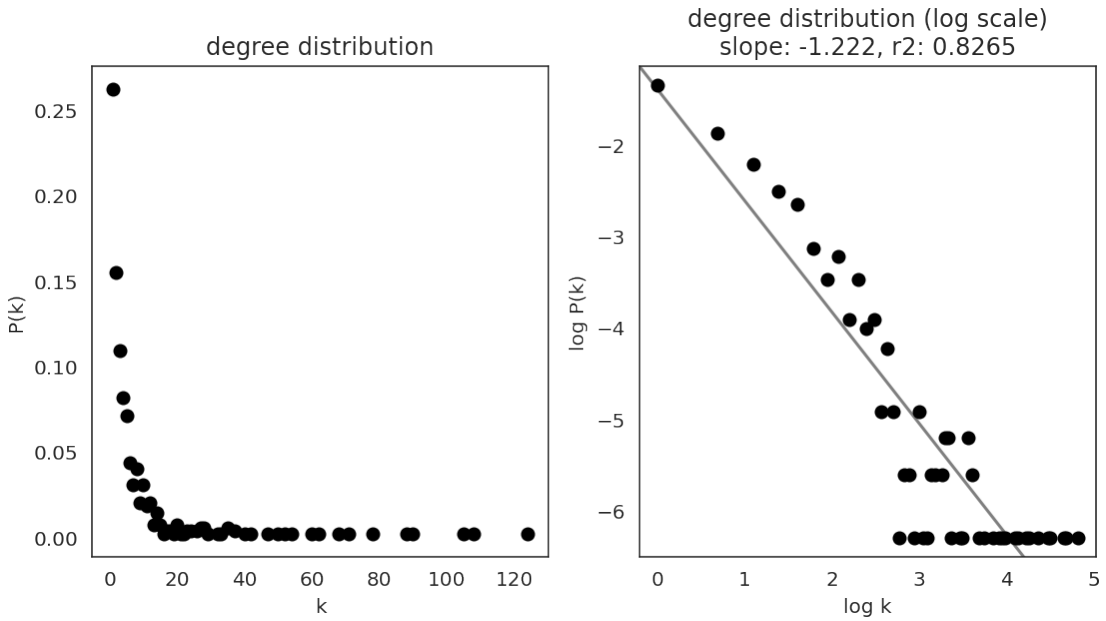
In the first step, we examine the network degree distribution.

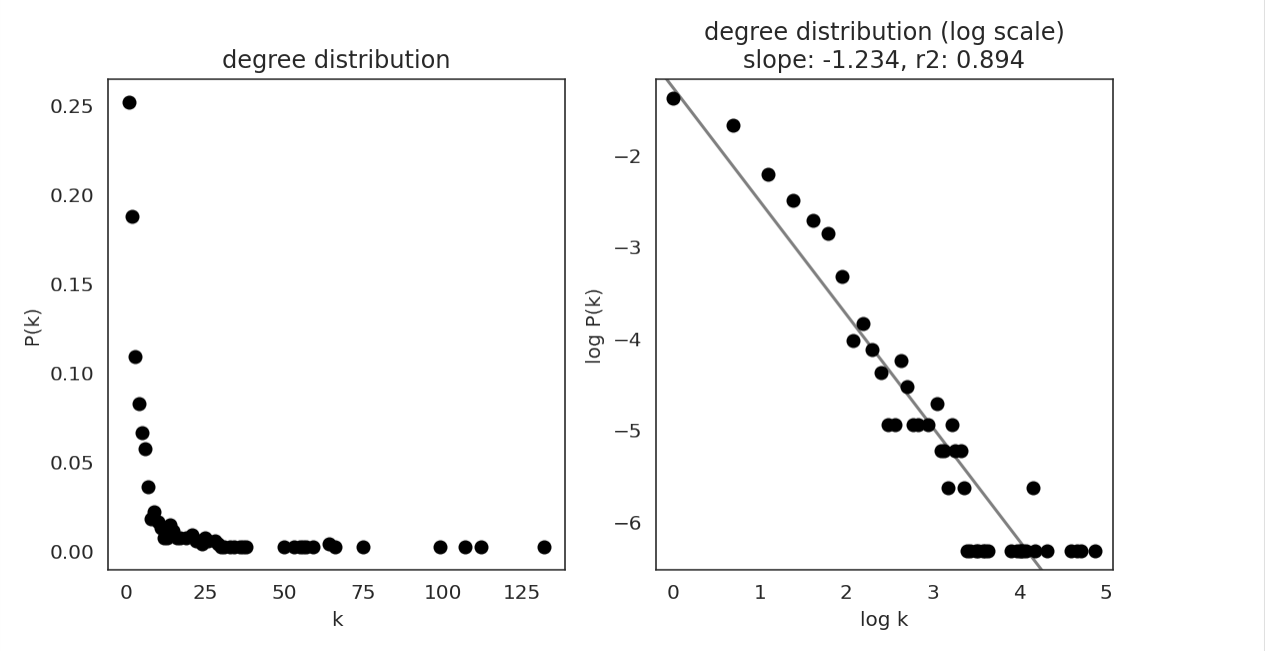
> Network degree, which is the number of edges for each node, is one of the important metrics used to investigate the network structure (https://en.wikipedia.org/wiki/Degree_distribution).

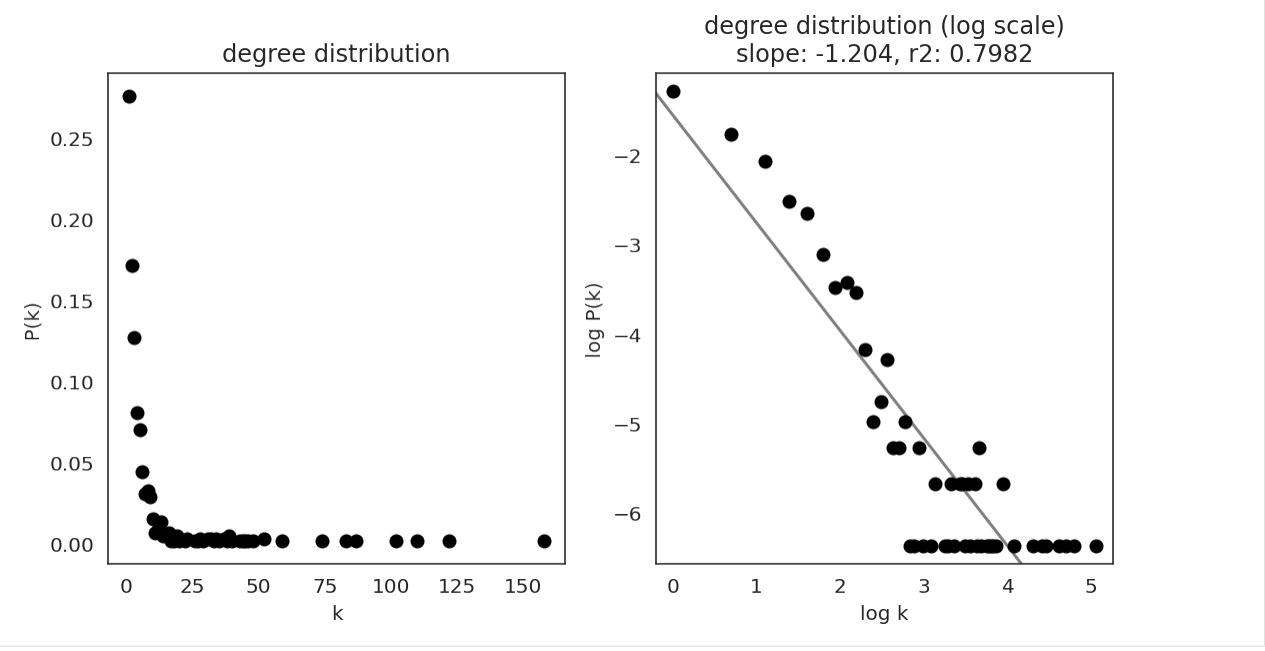
Please keep in mind that the degree distribution may change depending on the filtering threshold.

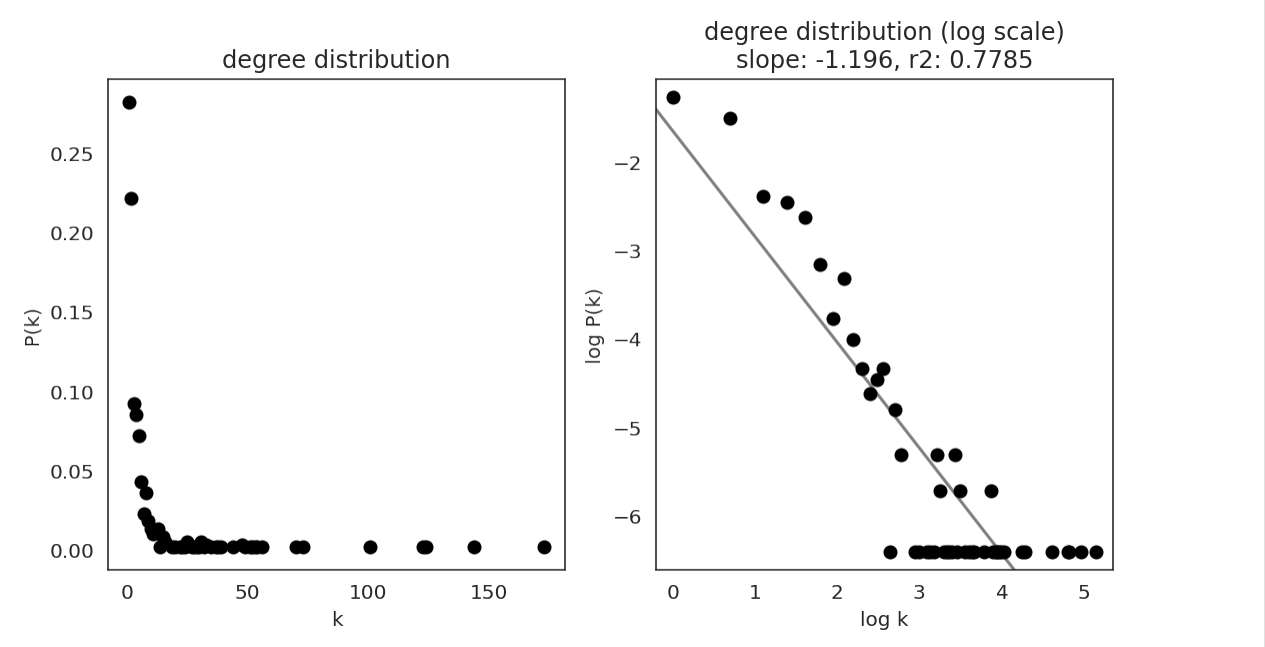
[18]:
```
plt.rcParams["figure.figsize"] = [9, 4.5]
```

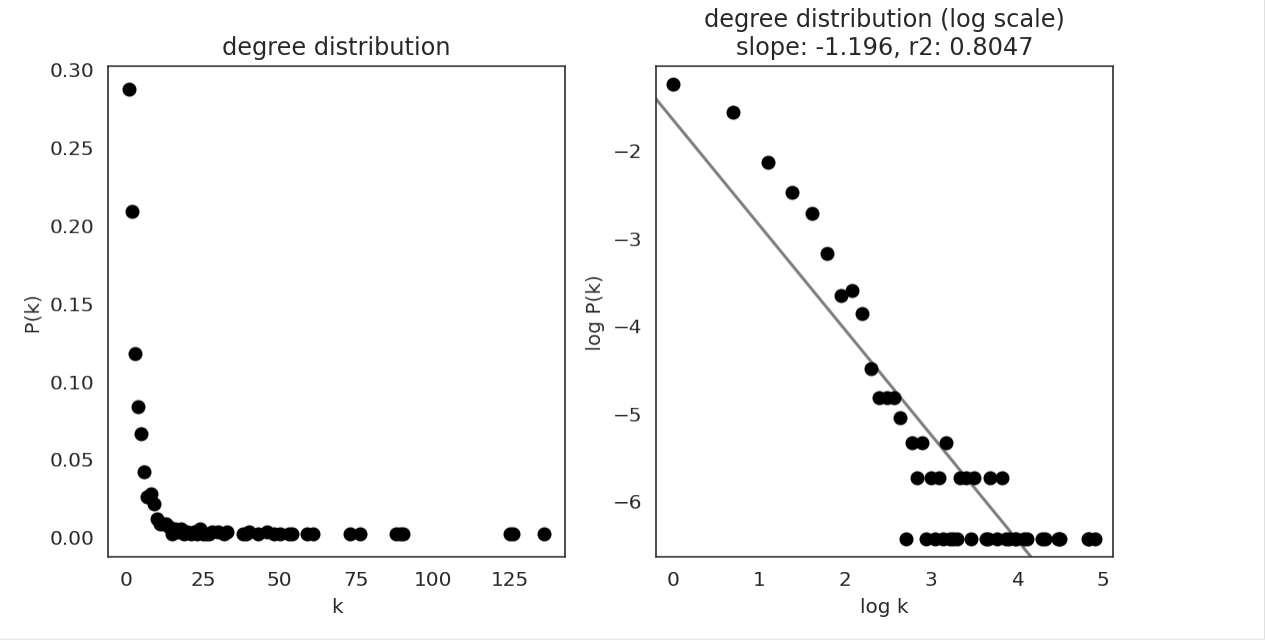
[19]:
```
links.plot_degree_distributions(plot_model=True,
                                                #save=f"{save_folder}/degree_
↪distribution/",
                                                )
```

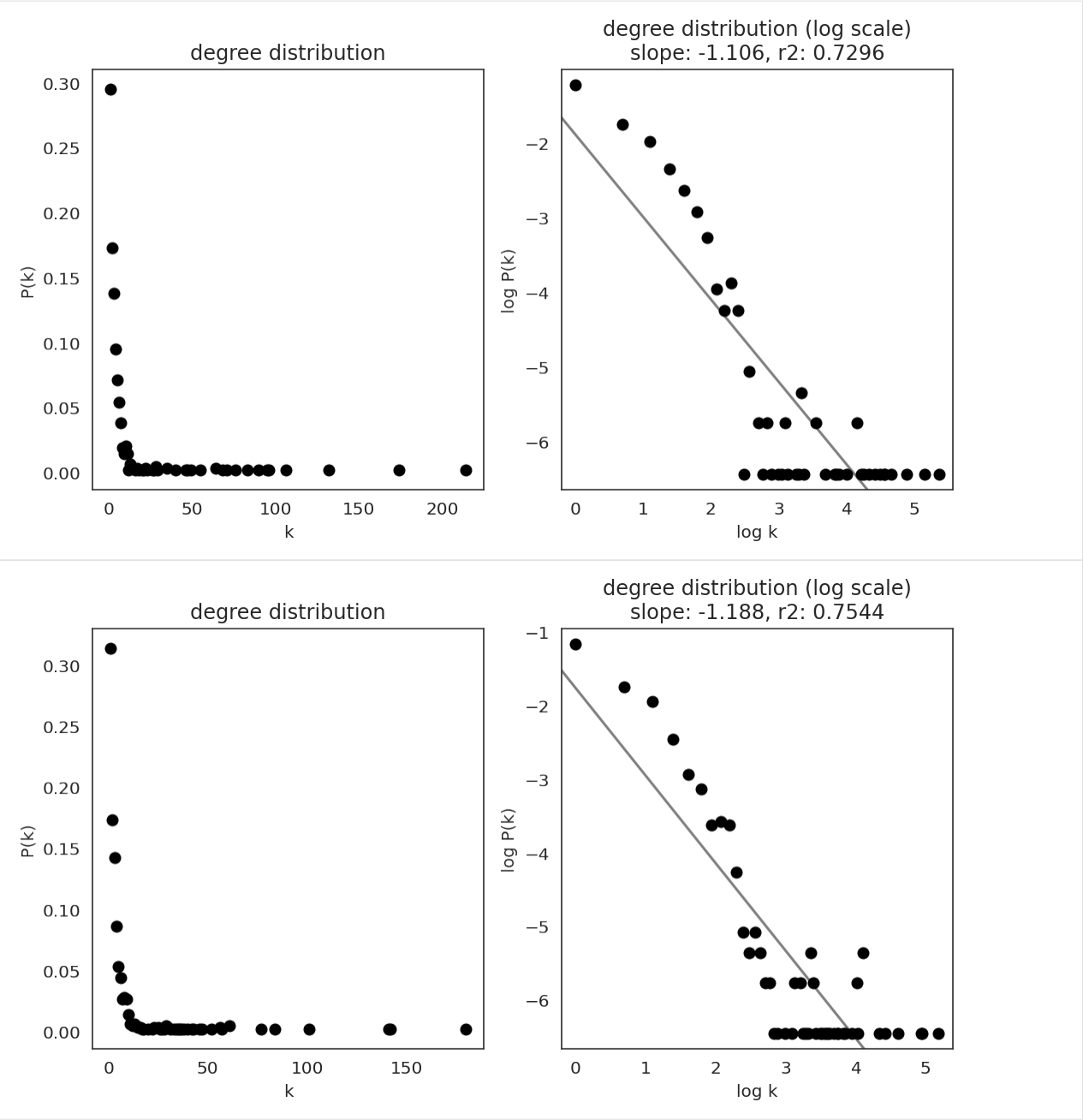


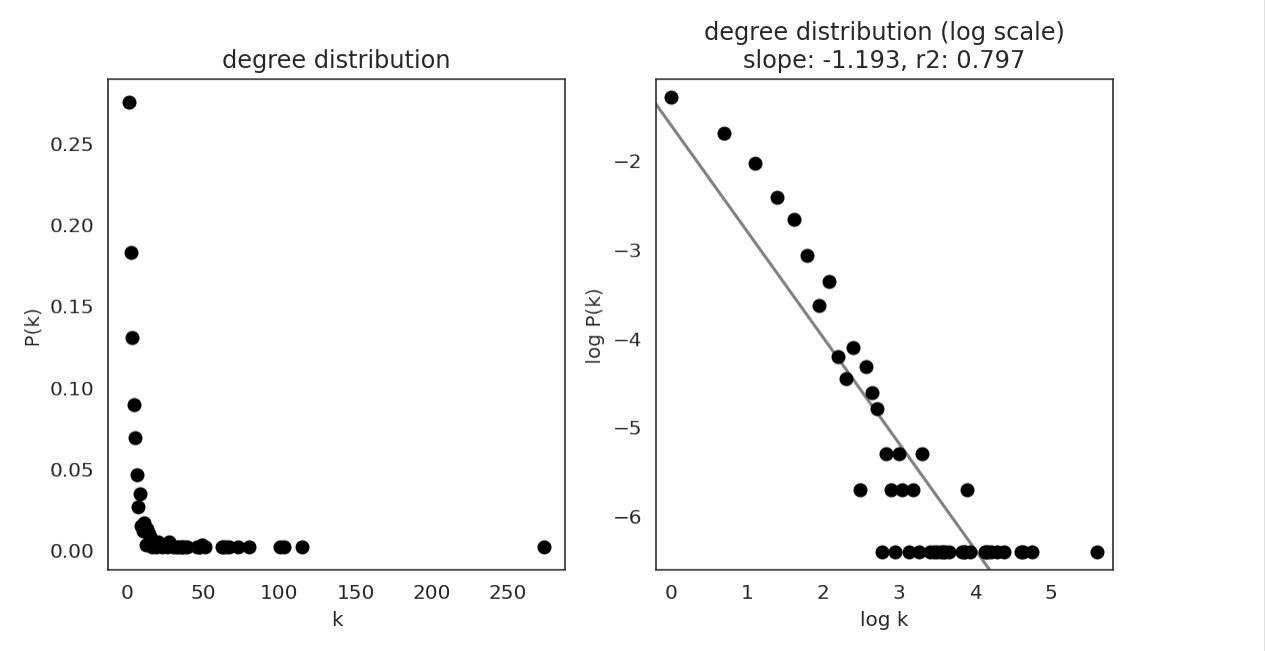

degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.234, r2: 0.894
log P(k)
log k

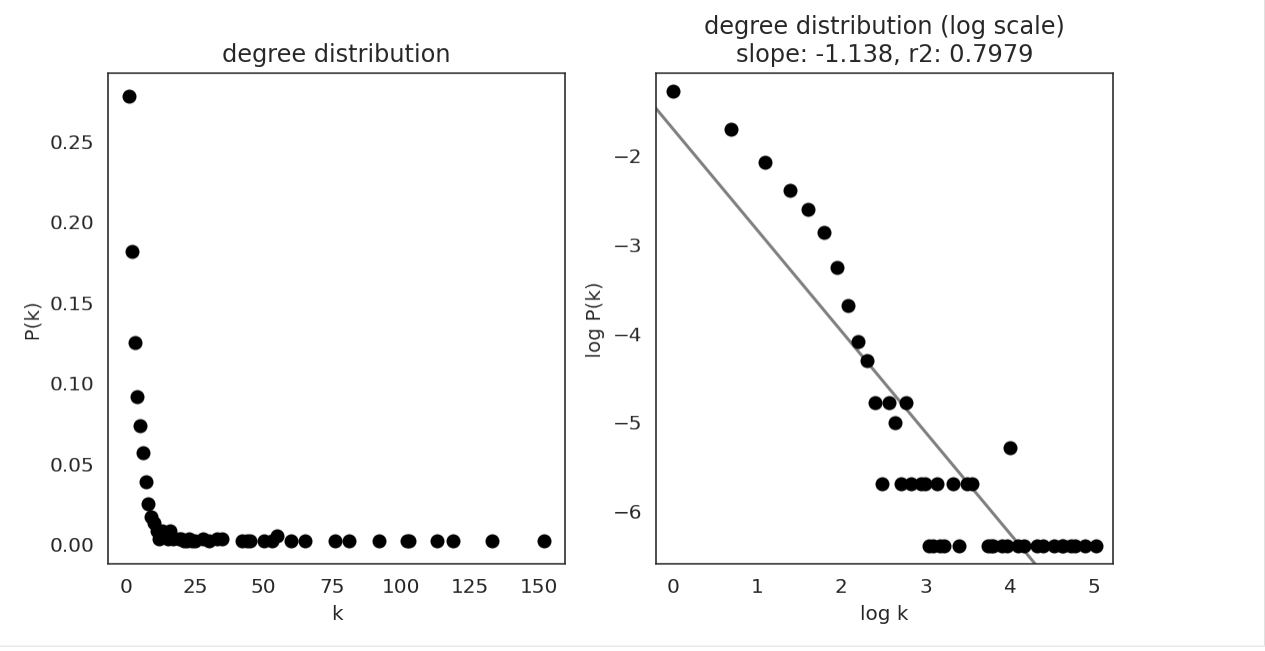

degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.204, r2: 0.7982
log P(k)
log k

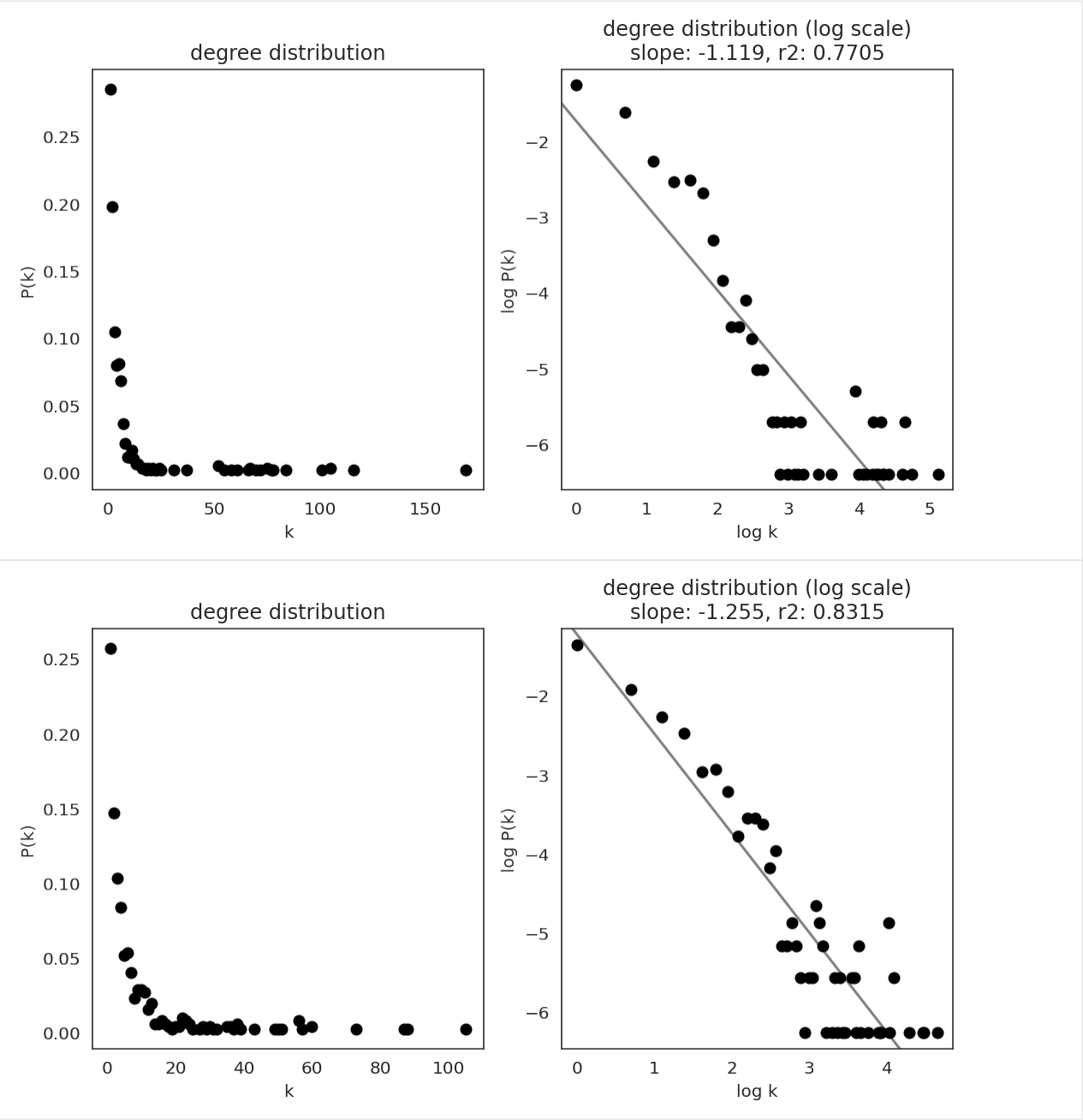

degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.196, r2: 0.7785
log P(k)
log k

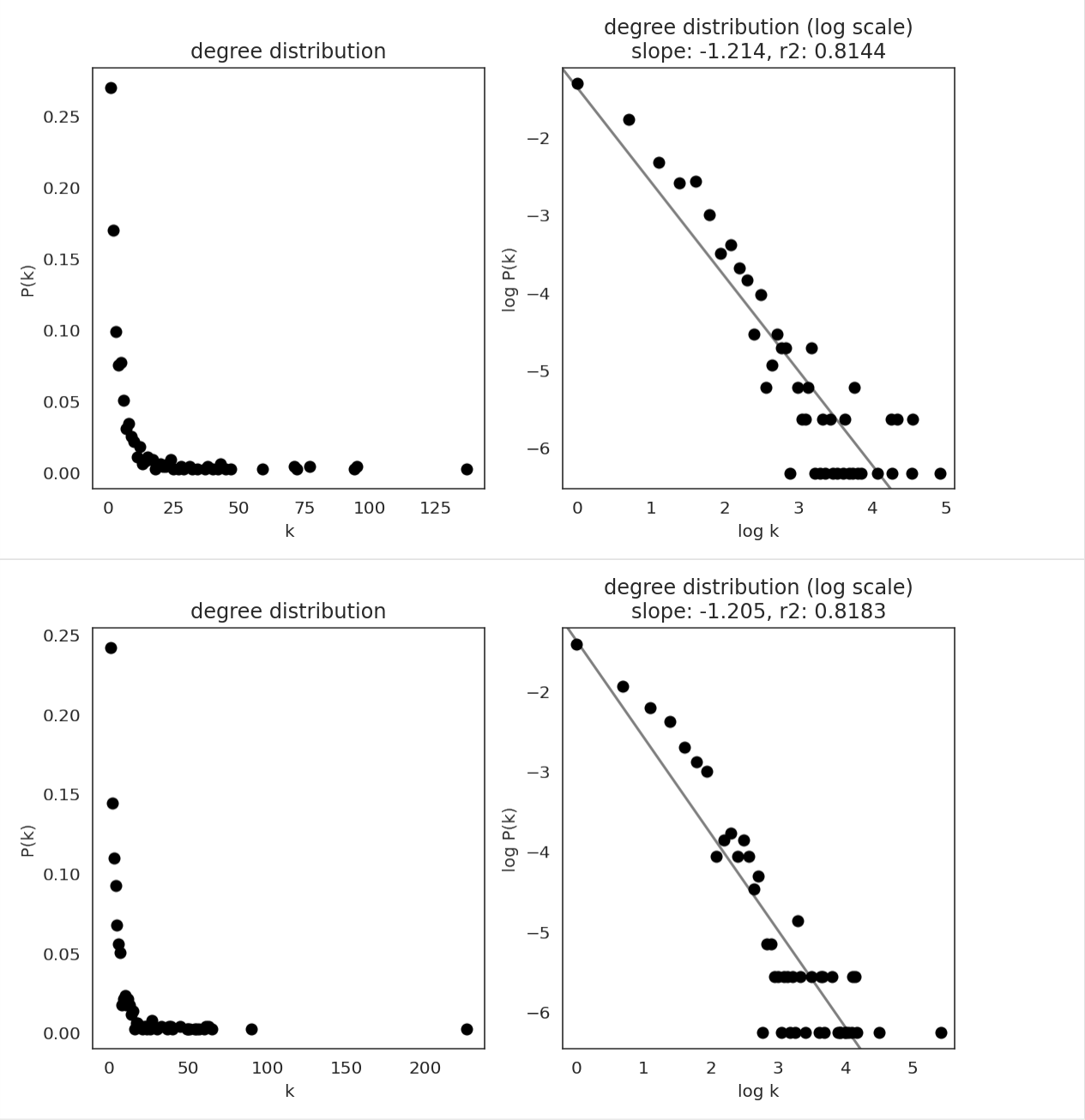

degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.196, r2: 0.8047
log P(k)
log k

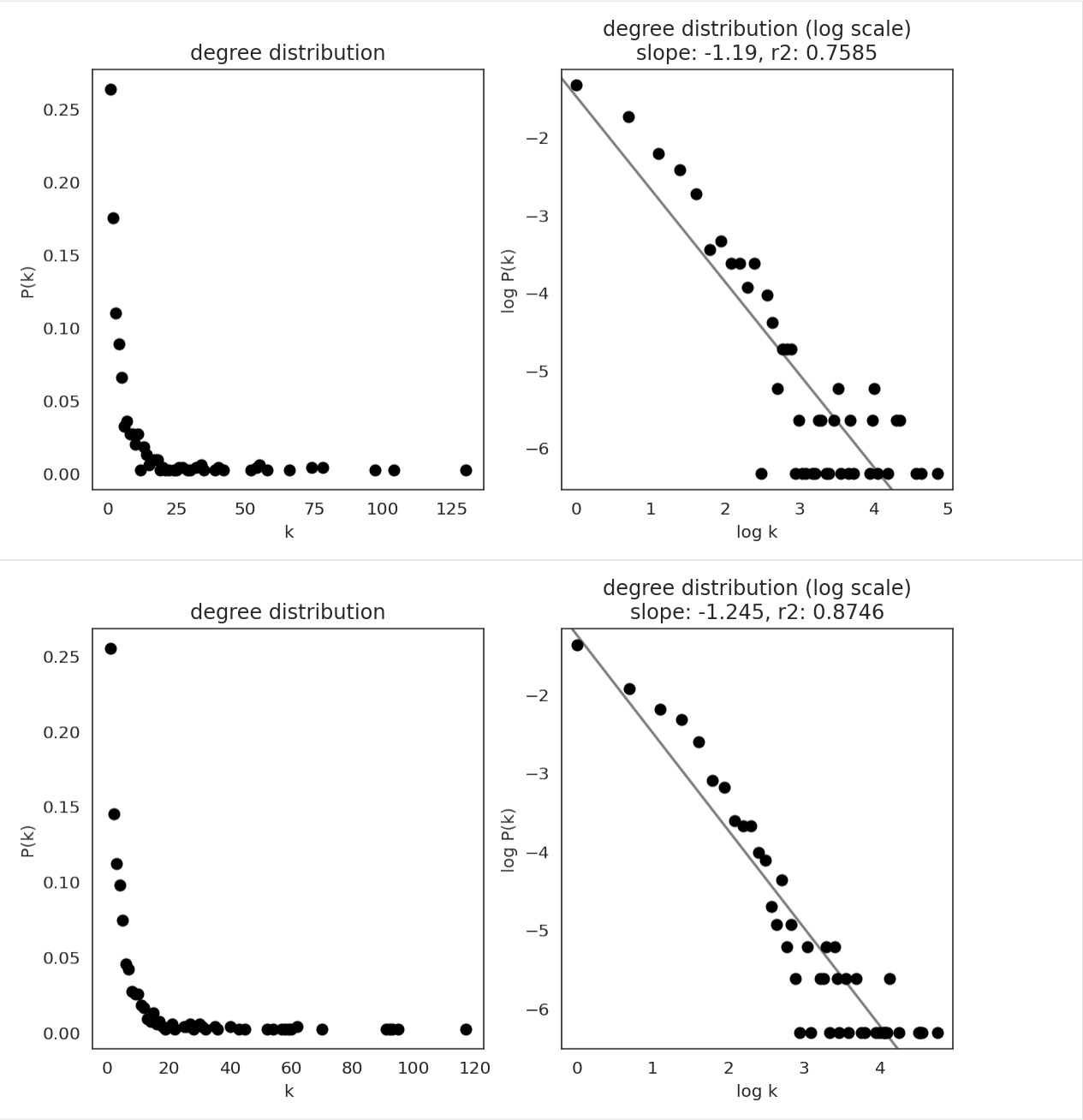

degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.106, r2: 0.7296
log P(k)
log k
degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.188, r2: 0.7544
log P(k)
log k

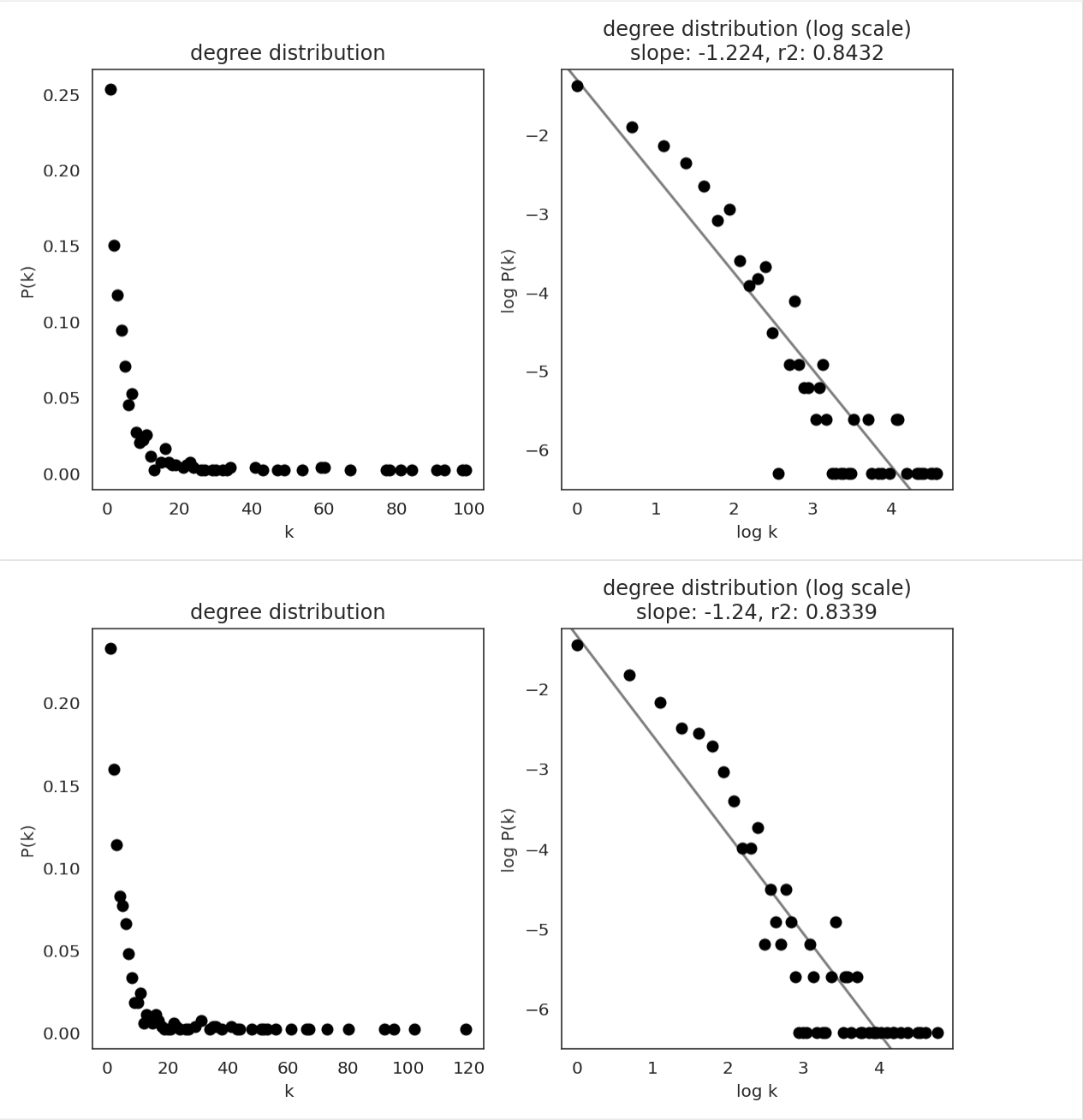

degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.193, r2: 0.797
log P(k)
log k

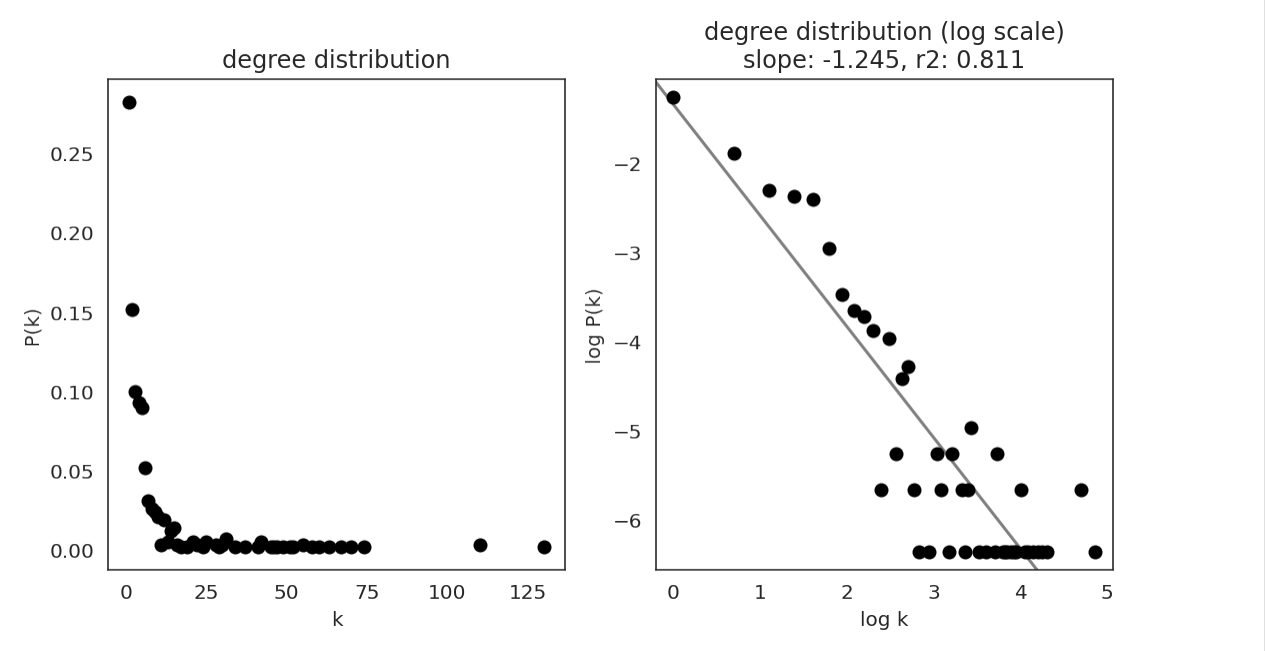

degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.138, r2: 0.7979
log P(k)
log k

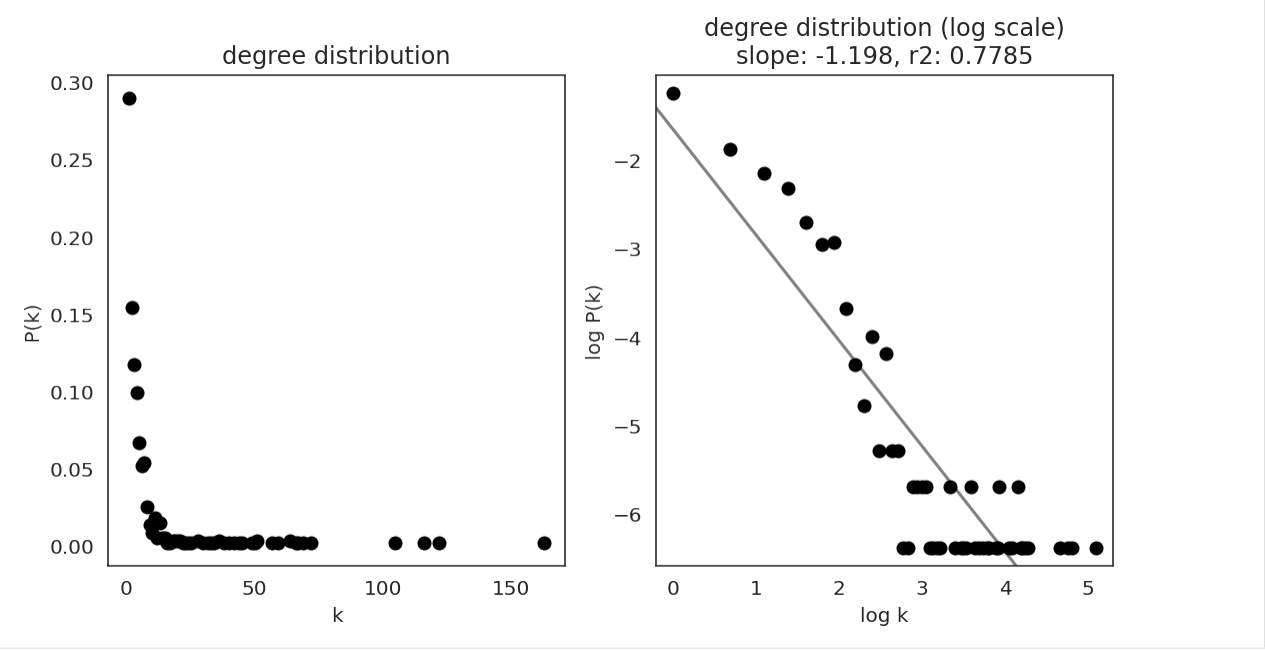

degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.119, r2: 0.7705
log P(k)
log k
degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.255, r2: 0.8315
log P(k)
log k

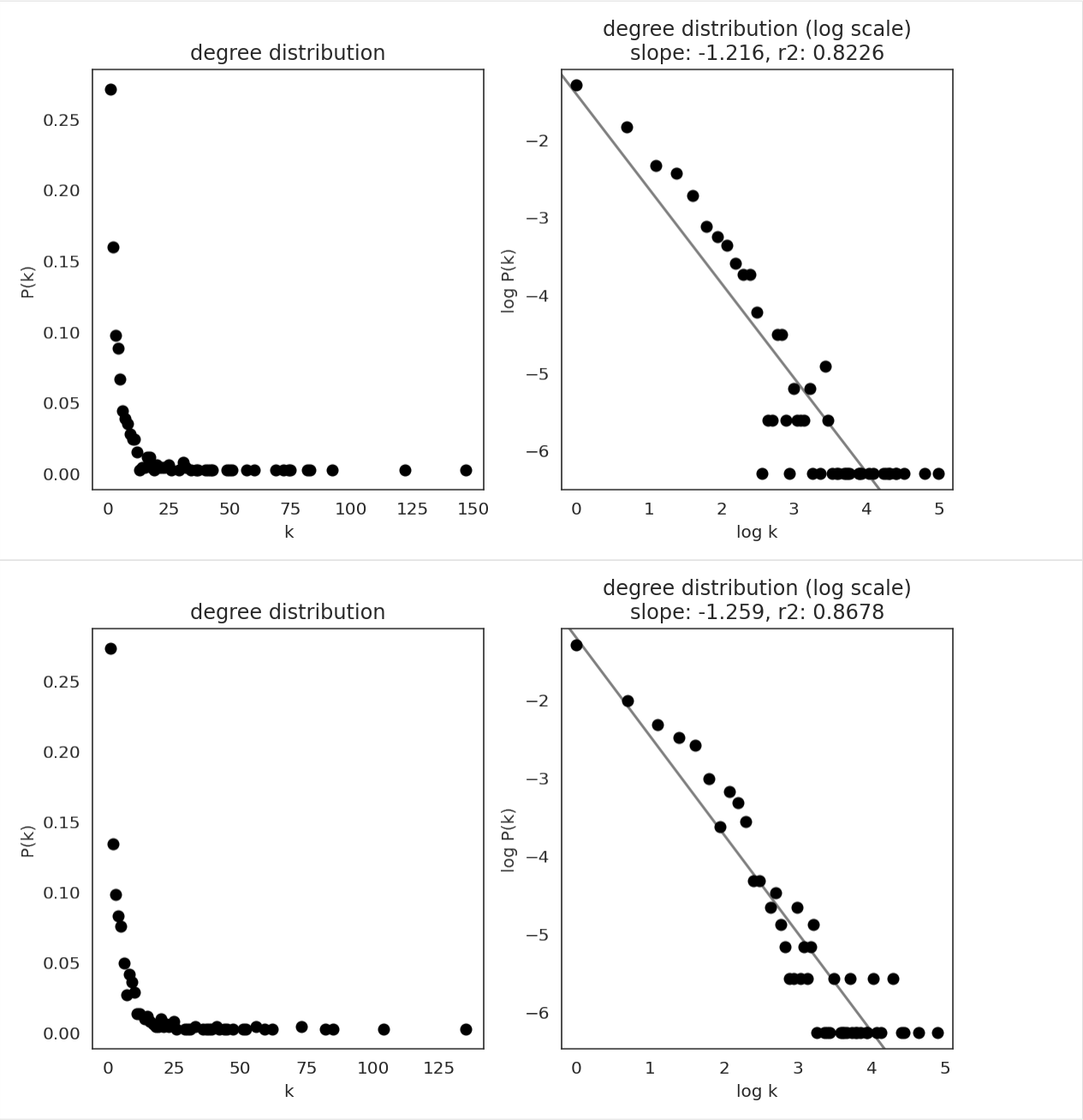

degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.214, r2: 0.8144
log P(k)
log k
degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.205, r2: 0.8183
log P(k)
log k

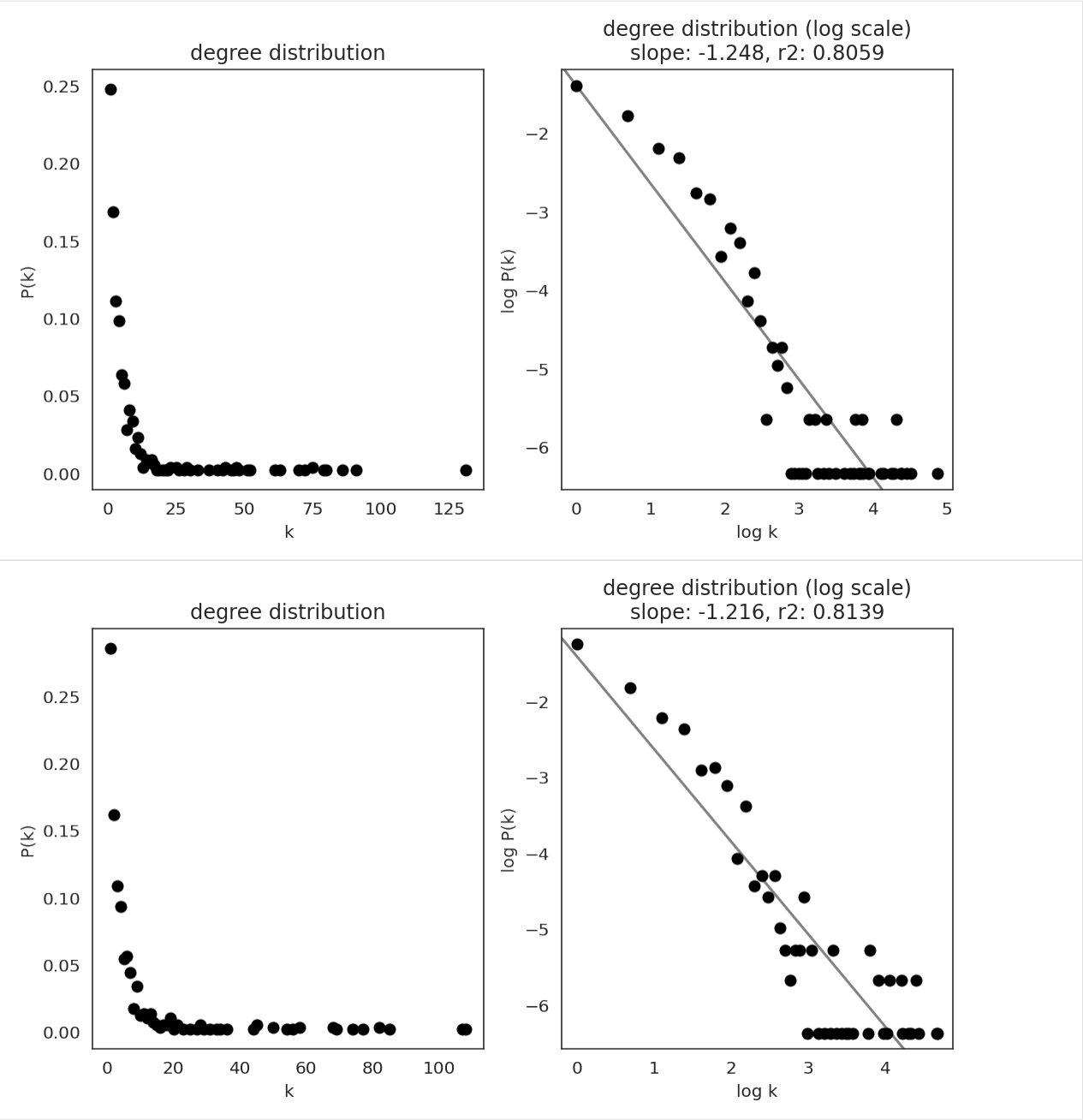

degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.19, r2: 0.7585
log P(k)
log k
degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.245, r2: 0.8746
log P(k)
log k

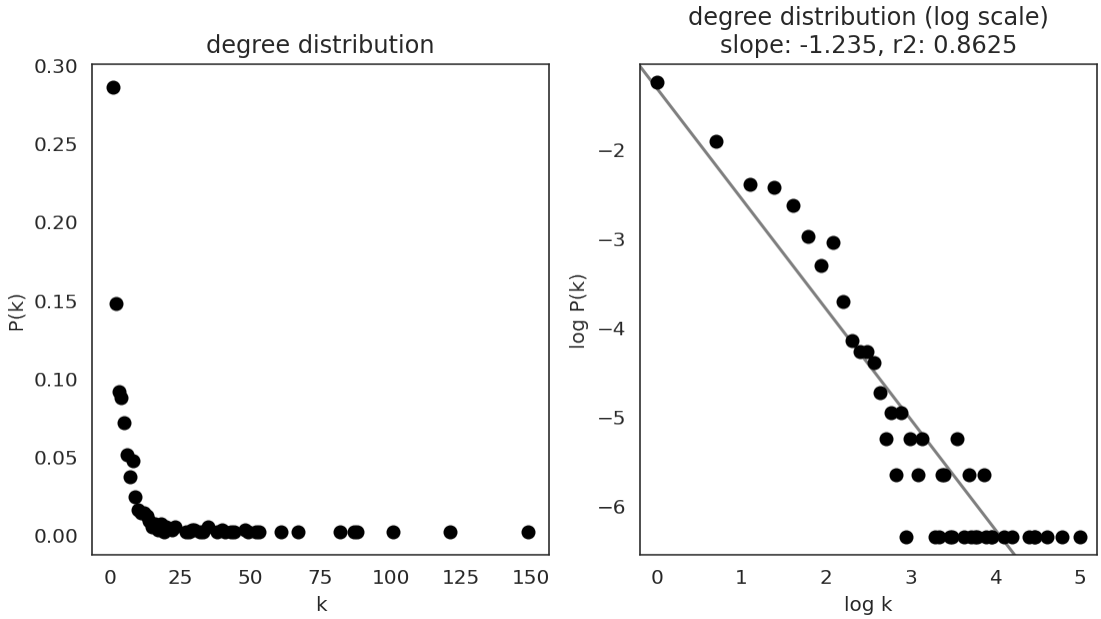

degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.224, r2: 0.8432
log P(k)
log k
degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.24, r2: 0.8339
log P(k)
log k


degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.245, r2: 0.811
log P(k)
log k


degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.198, r2: 0.7785
log P(k)
log k


degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.216, r2: 0.8226
log P(k)
log k
degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.259, r2: 0.8678
log P(k)
log k


degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.248, r2: 0.8059
log P(k)
log k
degree distribution
P(k)
k
degree distribution (log scale)
slope: -1.216, r2: 0.8139
log P(k)
log k



[50]:
```
plt.rcParams["figure.figsize"] = [6, 4.5]
```

Next, we calculate several network score using some R libraries. Please make sure that R libraries are installed in your PC before running the command below.

[25]:
```
# Calculate network scores. It takes several minutes.
links.get_score()
```

```
processing... batch 1/3
Ery_0: finished.
Ery_1: finished.
Ery_2: finished.
Ery_3: finished.
Ery_4: finished.
Ery_5: finished.
Ery_6: finished.
Ery_7: finished.
processing... batch 2/3
Ery_8: finished.
Ery_9: finished.
GMP_0: finished.
GMP_1: finished.
GMPl_0: finished.
Gran_0: finished.
Gran_1: finished.
Gran_2: finished.
processing... batch 3/3
MEP_0: finished.
Mk_0: finished.
Mo_0: finished.
Mo_1: finished.
```

The score is stored as a attribute `merged_score`.

[51]:
```
links.merged_score.head()
```

[51]:
```
       degree_all  degree_in  degree_out  clustering_coefficient  \
Mycn           42          0          42                0.003484
Ybx1           68         10          58                0.032924
Nfe2          124          7         117                0.025702
Gata2         108          8         100                0.031499
Myc            78          7          71                0.038628

       clustering_coefficient_weighted  degree_centrality_all  \
Mycn                          0.003821               0.076642
Ybx1                          0.033228               0.124088
Nfe2                          0.026156               0.226277
Gata2                         0.033937               0.197080
Myc                           0.042569               0.142336

       degree_centrality_in  degree_centrality_out  betweenness_centrality  \
Mycn               0.000000               0.076642                       0
Ybx1               0.018248               0.105839                    1290
Nfe2               0.012774               0.213504                    1556
Gata2              0.014599               0.182482                    1572
Myc                0.012774               0.129562                    1507

       closeness_centrality  ...  assortative_coefficient  \
Mycn               0.000010  ...                -0.124407
Ybx1               0.000004  ...                -0.124407
Nfe2               0.000008  ...                -0.124407
Gata2              0.000004  ...                -0.124407
Myc                0.000005  ...                -0.124407

       average_path_length  community_edge_betweenness  community_random_walk  \
Mycn              2.523462                           1                      1
Ybx1              2.523462                           2                      6
Nfe2              2.523462                           1                      1
Gata2             2.523462                           3                      1
Myc               2.523462                           4                      1

       community_eigenvector  module  connectivity  participation  \
Mycn                       1       5      3.591559       0.511338
Ybx1                       3       4      5.558769       0.608564
Nfe2                       1       3      5.267448       0.727107
Gata2                      4       3      4.823948       0.705761
Myc                        4       4      4.320821       0.709730

                role  cluster
Mycn   Connector Hub    Ery_0
Ybx1   Connector Hub    Ery_0
Nfe2   Connector Hub    Ery_0
Gata2  Connector Hub    Ery_0
Myc    Connector Hub    Ery_0

[5 rows x 22 columns]
```

Save processed GRN. We use this file in in silico TF perturbation analysis.

[52]:
```
# Save Links object.
links.to_hdf5(file_path="links.celloracle.links")
```

[6]:
```
# You can load files with the following command.
links = co.load_hdf5(file_path="links.celloracle.links")
```

If you are not interested in Network analysis and jut want to do TF perturbation simulation, you can skip the network analysis described below. Please go to next step: in silico gene perturbation with GRNs

https://morris-lab.github.io/CellOracle.documentation/tutorials/simulation.html

7. Network analysis; Network score for each gene

The `Links` class has many functions to visualize network score. See the documentation for the details of the functions.

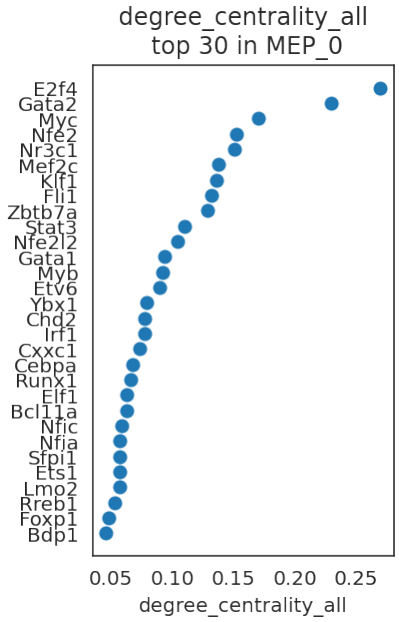
We have calculated several network scores using different centrality metrics. >The centrality score is one of the important indicators of network structure (https://en.wikipedia.org/wiki/Centrality).

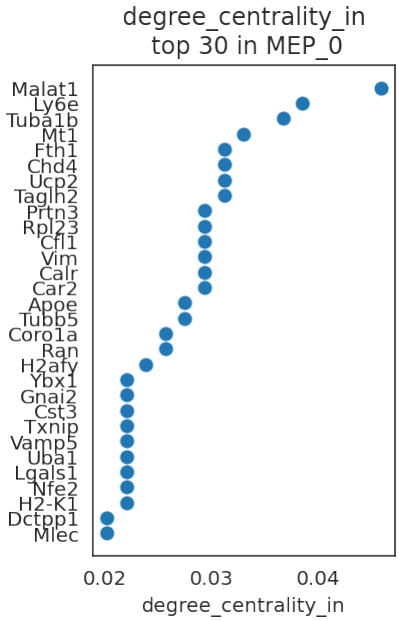
Let's visualize genes with high network centrality.

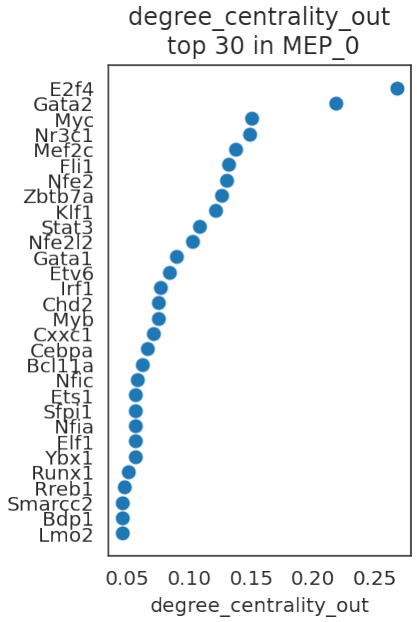
[53]:
```
# Check cluster name
links.cluster
```

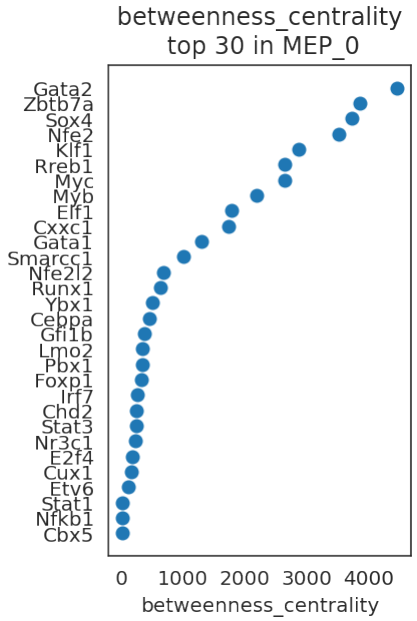
[53]:
```
['Ery_0',
 'Ery_1',
 'Ery_2',
 'Ery_3',
 'Ery_4',
 'Ery_5',
 'Ery_6',
 'Ery_7',
 'Ery_8',
 'Ery_9',
 'GMP_0',
 'GMP_1',
 'GMP_2',
 'GMPl_0',
 'GMPl_1',
 'Gran_0',
 'Gran_1',
 'Gran_2',
 'Gran_3',
 'MEP_0',
 'Mk_0',
 'Mo_0',
 'Mo_1',
 'Mo_2']
```

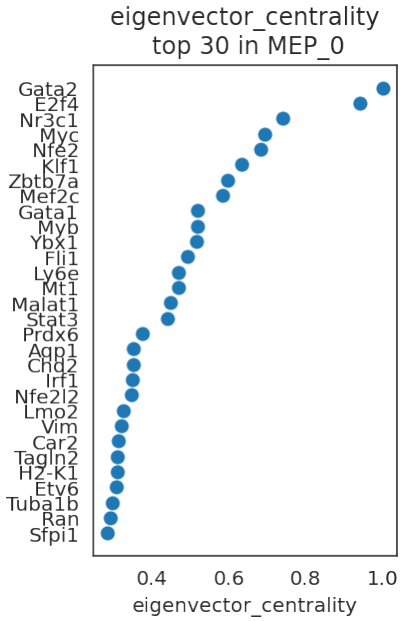
[54]:
```
# Visualize top n-th genes that have high scores.
links.plot_scores_as_rank(cluster="MEP_0", n_gene=30, save=f"{save_folder}/ranked_
↪score")
```

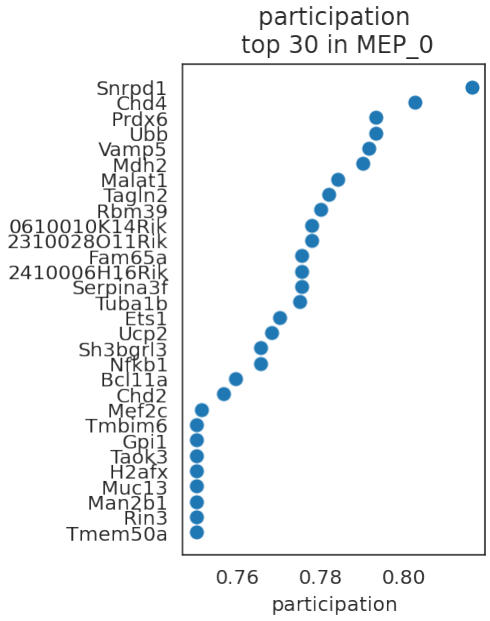

degree_centrality_all
top 30 in MEP_0
E2f4
Gata2
Myc
Nfe2
Nr3c1
Mef2c
Klf1
Fli1
Zbtb7a
Stat3
Nfe2l2
Gata1
Myb
Etv6
Ybx1
Chd2
Irf1
Cxxc1
Cebpa
Runx1
Elf1
Bcl11a
Nfic
Nfia
Sfpi1
Ets1
Lmo2
Rreb1
Foxp1
Bdp1
0.05
0.10
0.15
0.20
0.25
degree_centrality_all


degree_centrality_in
top 30 in MEP_0
Malat1
Ly6e
Tuba1b
Mt1
Fth1
Chd4
Ucp2
Taglh2
Prtn3
Rpl23
Cfl1
Vim
Calr
Car2
Apoe
Tubb5
Coro1a
Ran
H2afy
Ybx1
Gnai2
Cst3
Txnip
Vamp5
Uba1
Lgals1
Nfe2
H2-K1
Dctpp1
Mlec
0.02
0.03
0.04
degree_centrality_in


degree_centrality_out
top 30 in MEP_0
E2f4
Gata2
Myc
Nr3c1
Mef2c
Fli1
Nfe2
Zbtb7a
Klf1
Stat3
Nfe2l2
Gata1
Etv6
Irf1
Chd2
Myb
Cxxc1
Cebpa
Bcl11a
Nfic
Ets1
Spi1
Nfia
Elf1
Ybx1
Runx1
Rreb1
Smarcc2
Bdp1
Lmo2
0.05
0.10
0.15
0.20
0.25
degree_centrality_out

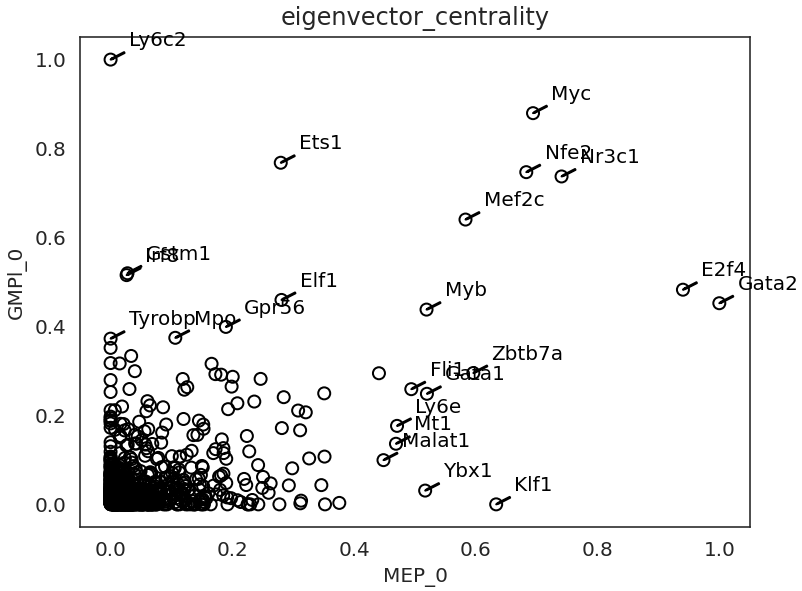

betweenness_centrality
top 30 in MEP_0
Gata2
Zbtb7a
Sox4
Nfe2
Klf1
Rreb1
Myc
Myb
Elf1
Cxxc1
Gata1
Smarcc1
Nfe2l2
Runx1
Ybx1
Cebpa
Gfi1b
Lmo2
Pbx1
Foxp1
Irf7
Chd2
Stat3
Nr3c1
E2f4
Cux1
Etv6
Stat1
Nfkb1
Cbx5
0
1000
2000
3000
4000
betweenness_centrality



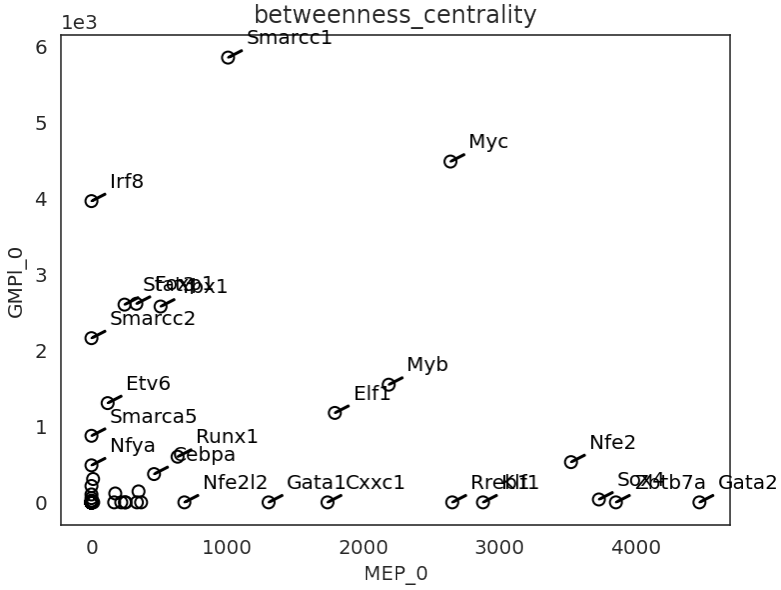


By comparing network scores between two clusters, we can analyze differences in GRN structure.

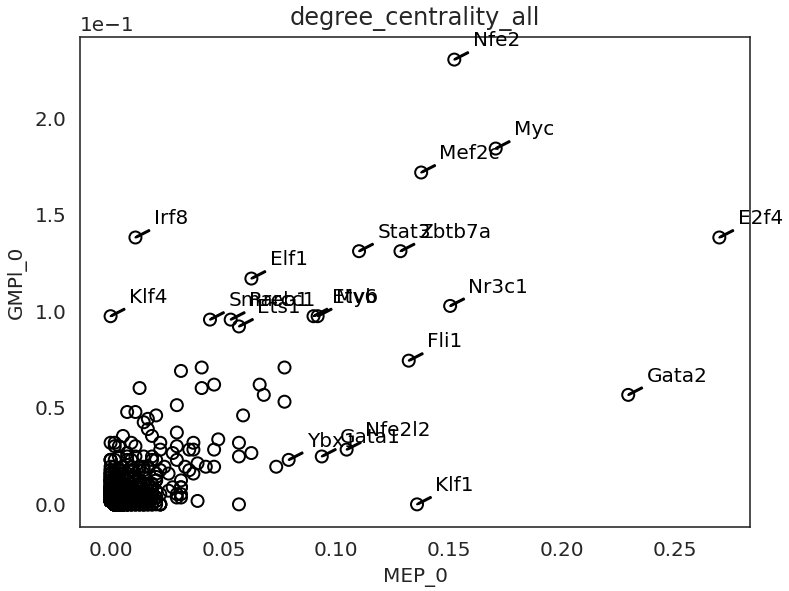
[55]:
```
plt.ticklabel_format(style='sci',axis='y',scilimits=(0,0))
links.plot_score_comparison_2D(value="eigenvector_centrality",
                               cluster1="MEP_0", cluster2="GMP1_0",
                               percentile=98, save=f"{save_folder}/score_comparison")
```



[56]:
```
plt.ticklabel_format(style='sci',axis='y',scilimits=(0,0))
links.plot_score_comparison_2D(value="betweenness_centrality",
                               cluster1="MEP_0", cluster2="GMP1_0",
                               percentile=98, save=f"{save_folder}/score_comparison")
```



[57]:
```
plt.ticklabel_format(style='sci',axis='y',scilimits=(0,0))
links.plot_score_comparison_2D(value="degree_centrality_all",
                               cluster1="MEP_0", cluster2="GMP1_0",
                               percentile=98, save=f"{save_folder}/score_comparison")
```

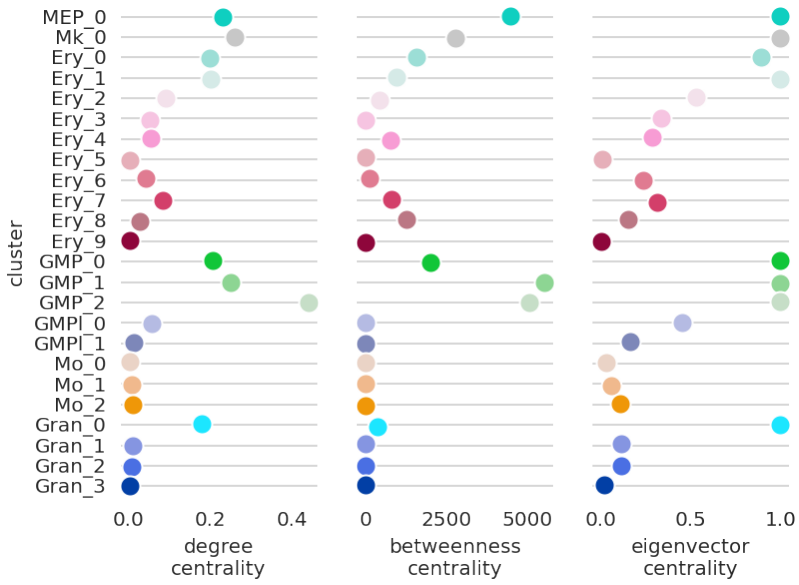


In the following session, we focus on how a gene's network score changes during the differentiation.

Using Gata2, we introduce how to visualize networks scores dynamics.

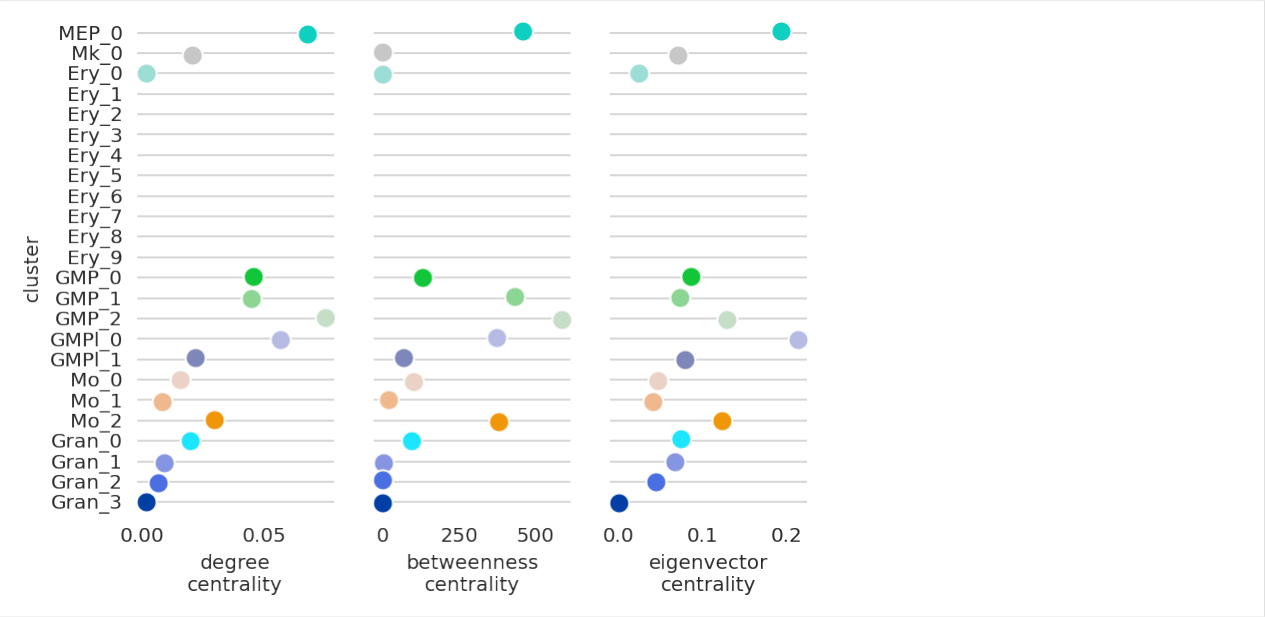
Gata2 is known to play an essential role in the early MEP and GMP populations. .

[58]:
```
# Visualize Gata2 network score dynamics
links.plot_score_per_cluster(goi="Gata2", save=f"{save_folder}/network_score_per_gene/
↪")
```



If a gene have no connections in a cluster, it is impossible to calculate network degree scores. Thus the scores will not be shown. For example, Cebpa have no connection in the erythloids clusters, and there is no degree scores for Cebpa in these clusters as follows.

[59]:
```
links.plot_score_per_cluster(goi="Cebpa")
```



You can check filtered network edge as follows.

[62]:
```
cluster_name = "Ery_1"
filtered_links_df = links.filtered_links[cluster_name]
filtered_links_df.head()
```

[62]:

	source	target	coef_mean	coef_abs	p	-logp
5480	Gata2	Apoe	0.100094	0.100094	6.274381e-16	15.202429
5496	Zfhx3	Apoe	0.098389	0.098389	1.749953e-13	12.756974
68791	Hnf4a	Top2a	0.098258	0.098258	7.209973e-10	9.142066
48857	E2f4	Phf10	0.095547	0.095547	1.429657e-13	12.844768
5470	Nfatc3	Apoe	-0.095185	0.095185	2.385889e-14	13.622350

You can confirm that there is no Cebpa connection in Ery_0 cluster.

[63]:
```
filtered_links_df[filtered_links_df.source == "Cebpa"]
```

[63]:
```
Empty DataFrame
Columns: [source, target, coef_mean, coef_abs, p, -logp]
Index: []
```

We can calculate gene cartography as follows.

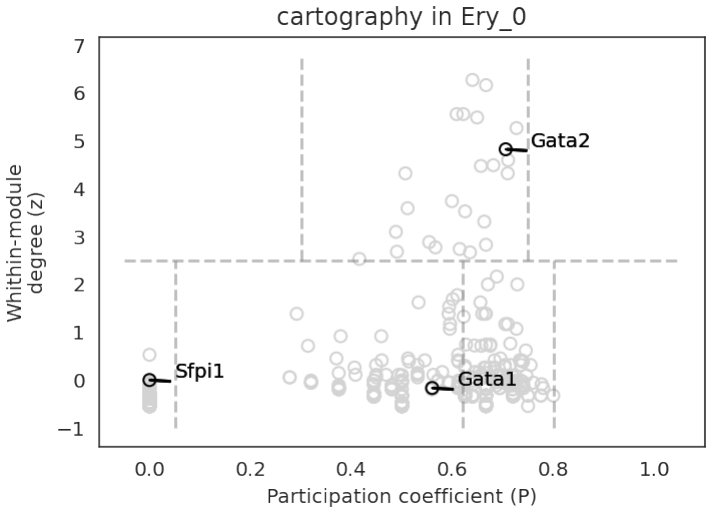
The gene cartography will be calculated for the GRN in each cluster.

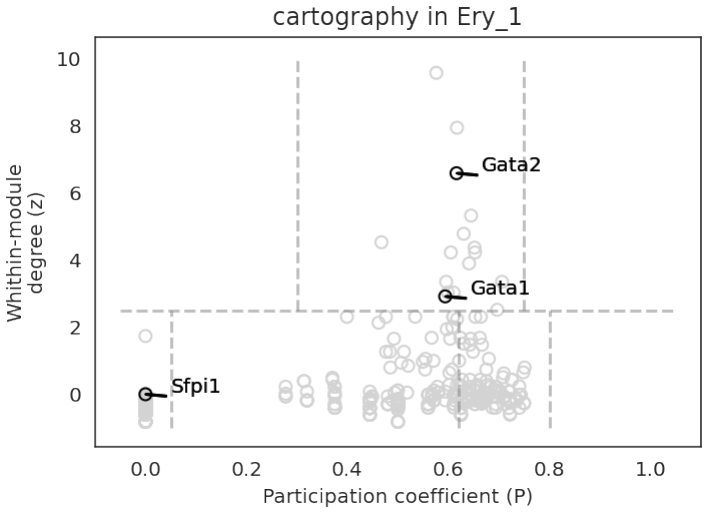
> Gene cartography is a method for gene network analysis. The method classifies gene into several groups using the network module structure and connections. For more information on gene cartography, please refer to the following paper (https://www.nature.com/articles/nature03288).

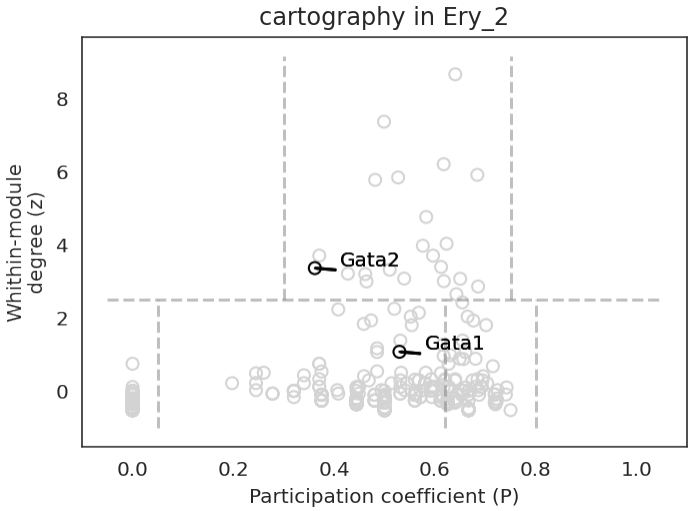
[64]:
```
# Plot cartography as a scatter plot
links.plot_cartography_scatter_per_cluster(scatter=True,
                                           kde=False,
                                           gois=["Gata1", "Gata2", "Sfpi1"], #␣
↪Highlight genes of interest
                                           auto_gene_annot=False,
                                           args_dot={"n_levels": 105},
                                           args_line={"c":"gray"}, save=f"{save_
↪folder}/cartography")
```

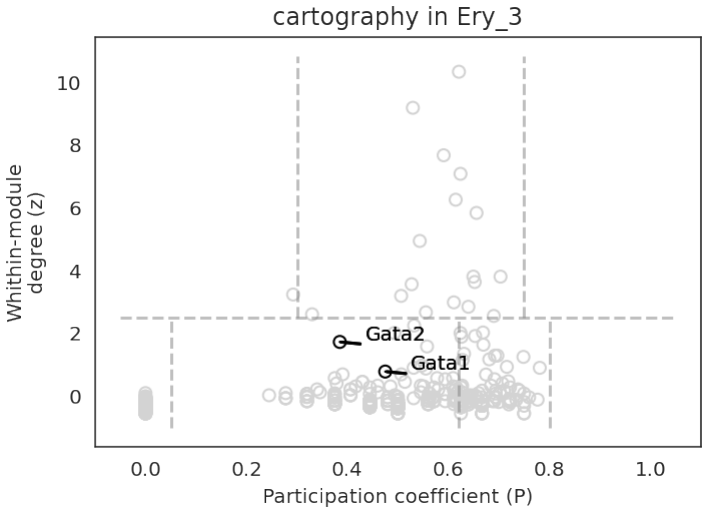
(continues on next page)

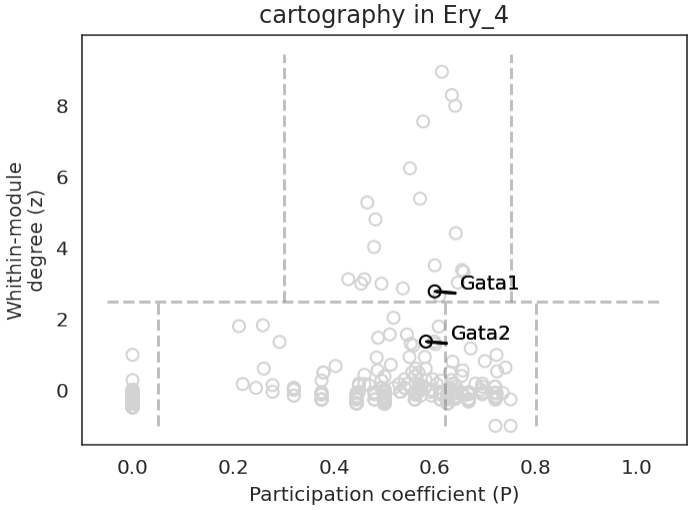
(continued from previous page)

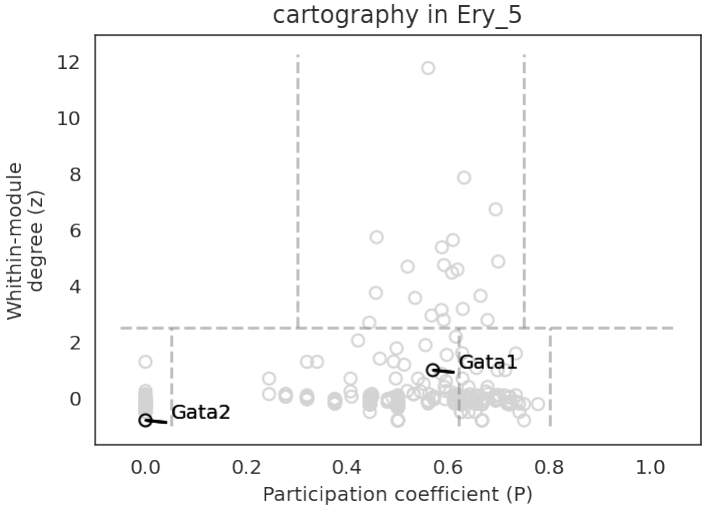


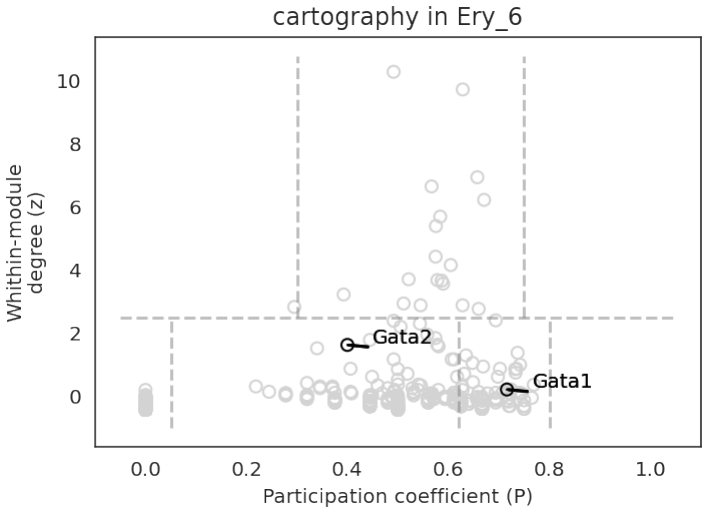


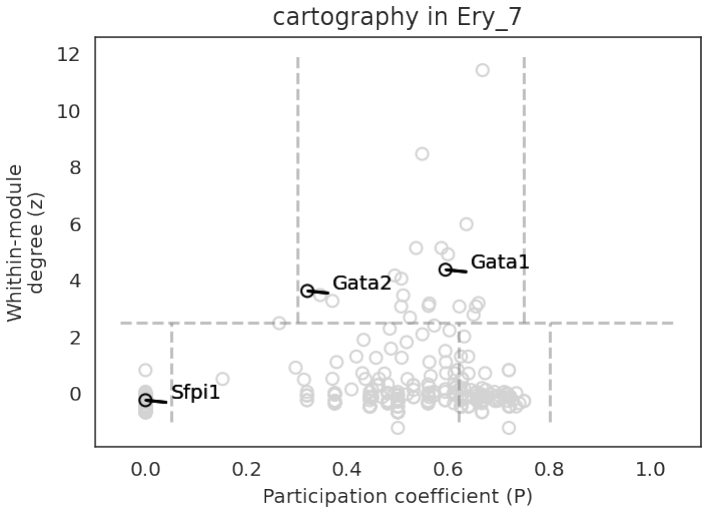


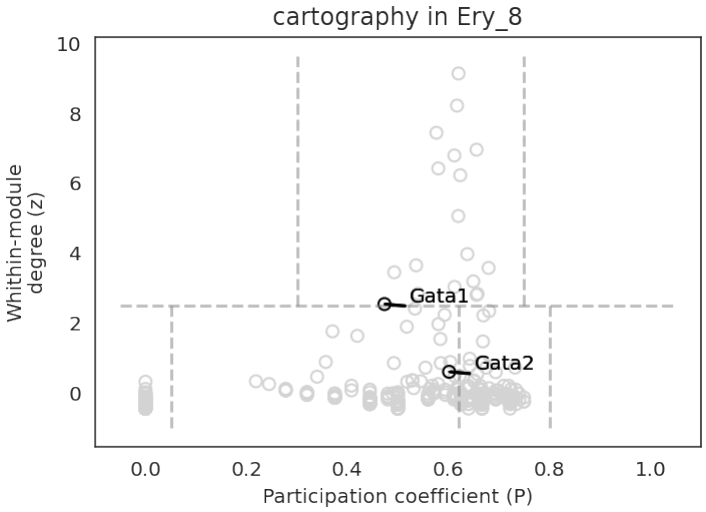

cartography in Ery_3
Whithin-module degree (z)
Participation coefficient (P)
Gata2
Gata1

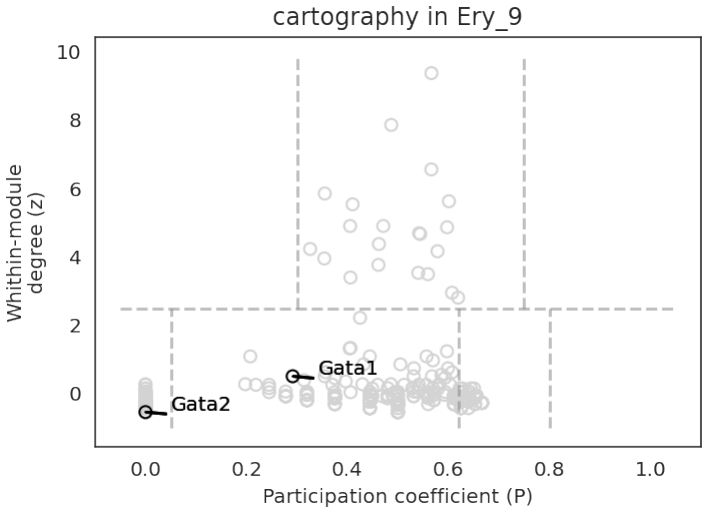

cartography in Ery_4
Whithin-module degree (z)
Participation coefficient (P)
Gata1
Gata2

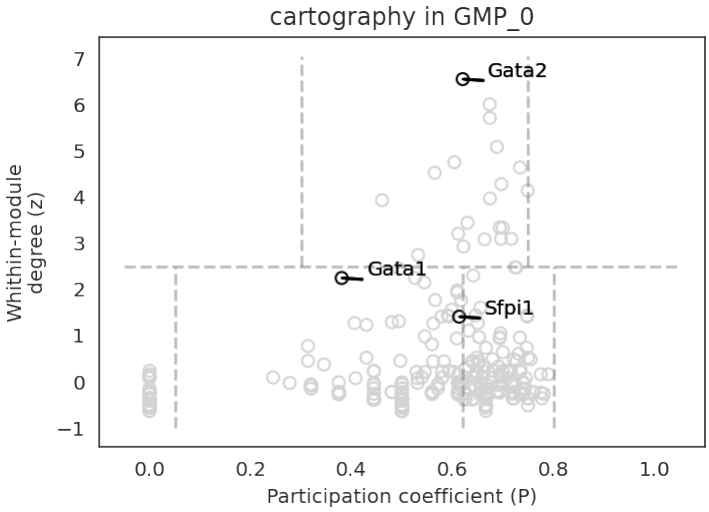

cartography in Ery_5
Whithin-module degree (z)
Participation coefficient (P)
Gata1
Gata2

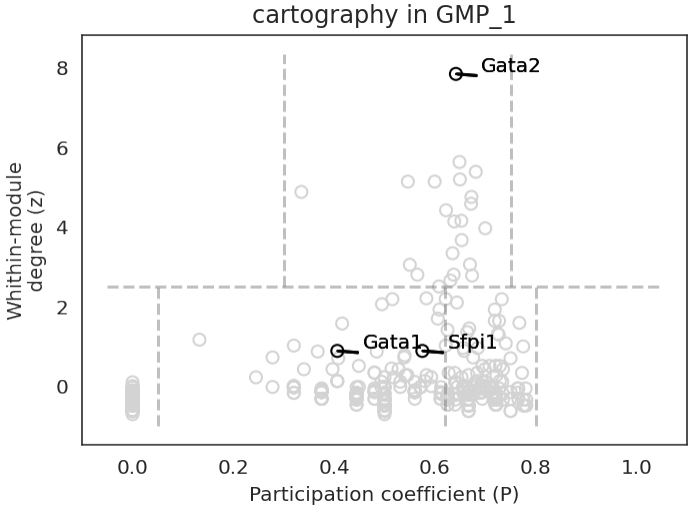

cartography in Ery_6
Whithin-module degree (z)
Participation coefficient (P)
Gata2
Gata1
0
2
4
6
8
10
0.0
0.2
0.4
0.6
0.8
1.0

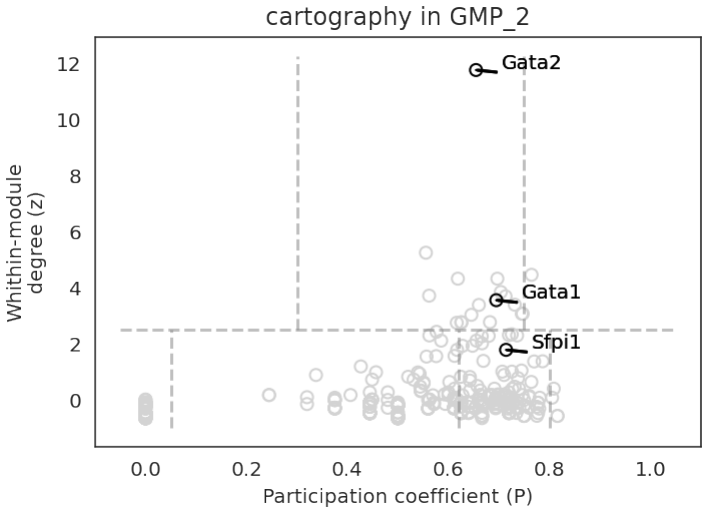

cartography in Ery_7
Whithin-module degree (z)
Participation coefficient (P)
Gata1
Gata2
Sfpi1
0
2
4
6
8
10
12
0.0
0.2
0.4
0.6
0.8
1.0

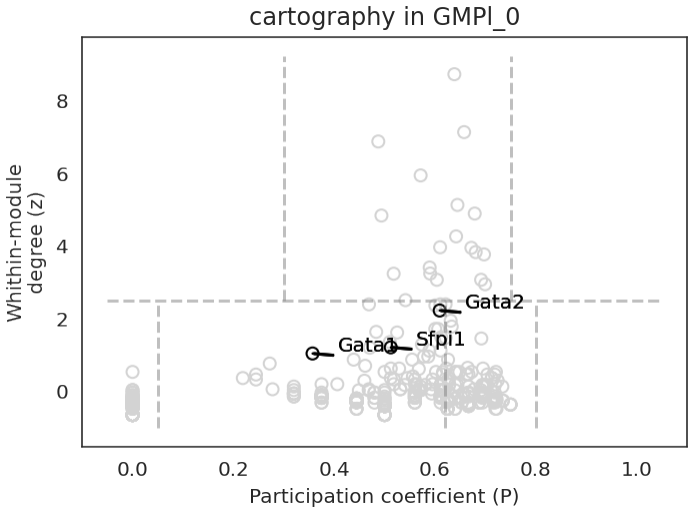

cartography in Ery_8
Whithin-module degree (z)
Participation coefficient (P)
Gata1
Gata2
0
2
4
6
8
10
0.0
0.2
0.4
0.6
0.8
1.0

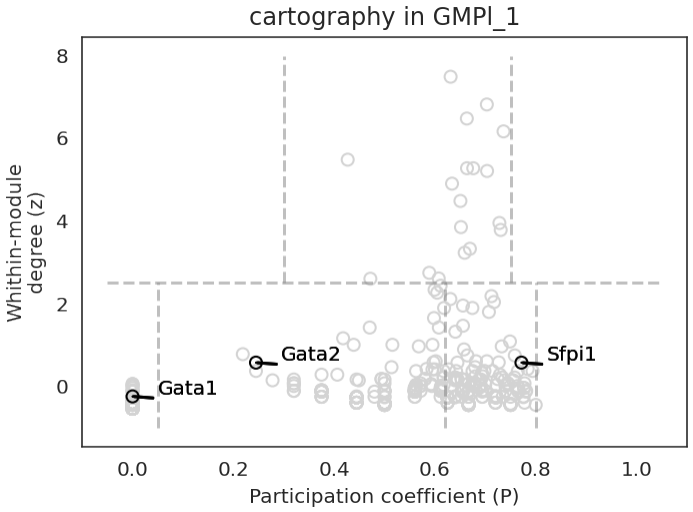

cartography in Ery_9
Whithin-module degree (z)
Participation coefficient (P)
Gata1
Gata2

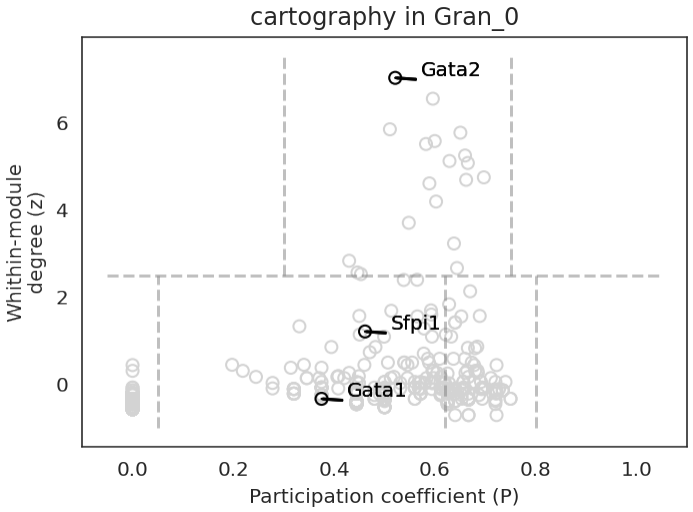

cartography in GMP_0
Whithin-module degree (z)
Participation coefficient (P)
Gata2
Gata1
Sfpi1

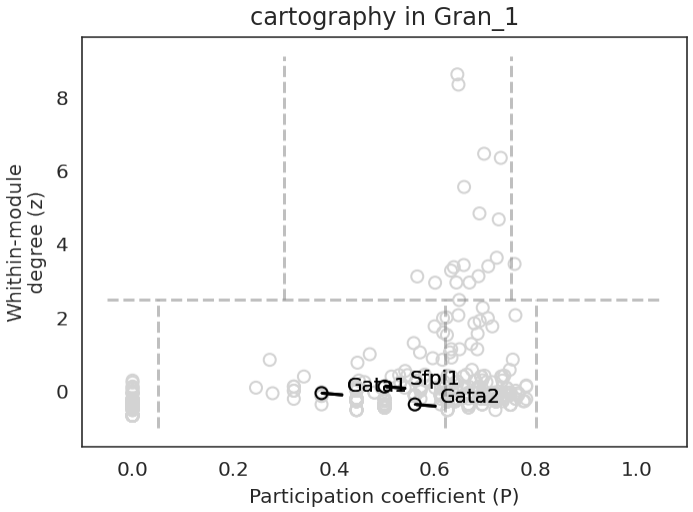

cartography in GMP_1
Whithin-module degree (z)
Participation coefficient (P)
Gata2
Gata1
Sfpi1

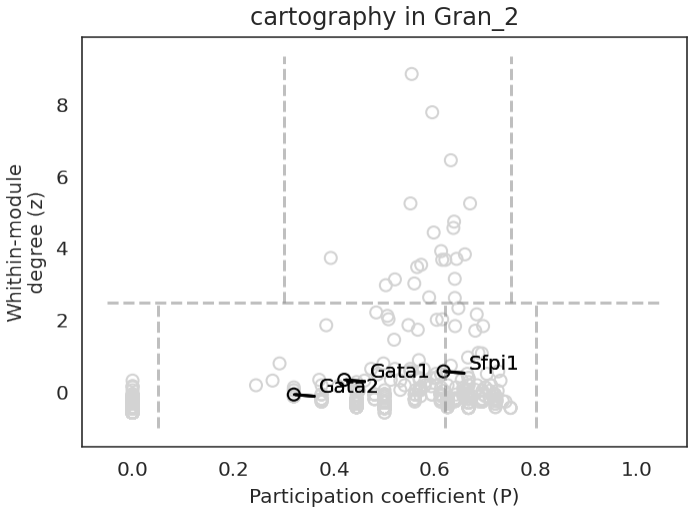

cartography in GMP_2
Gata2
Gata1
Sfpi1
Whithin-module degree (z)
Participation coefficient (P)
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10
12

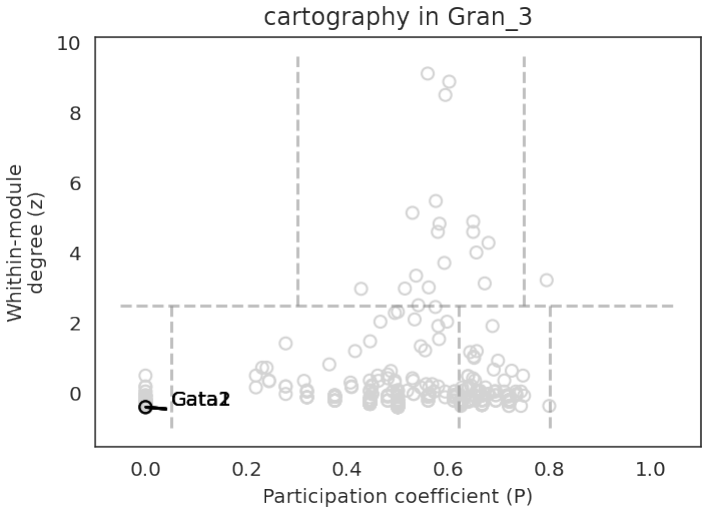

cartography in GMPl_0
Gata2
Gata1
Sfpi1
Whithin-module degree (z)
Participation coefficient (P)
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8

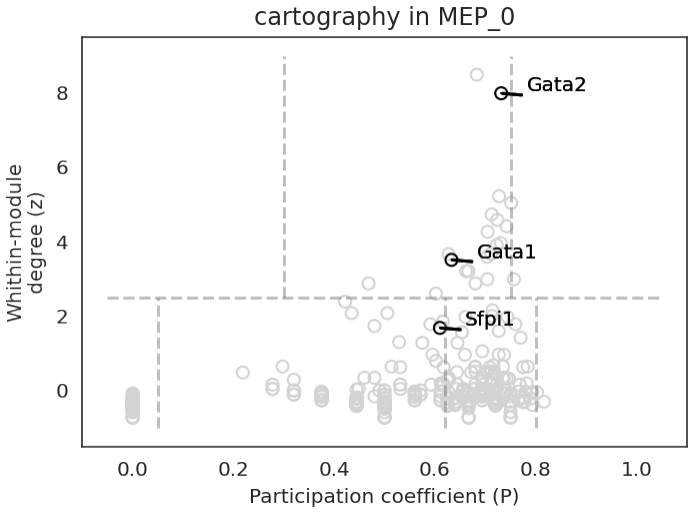

cartography in GMPl_1
Gata2
Gata1
Sfpi1
Whithin-module degree (z)
Participation coefficient (P)
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8

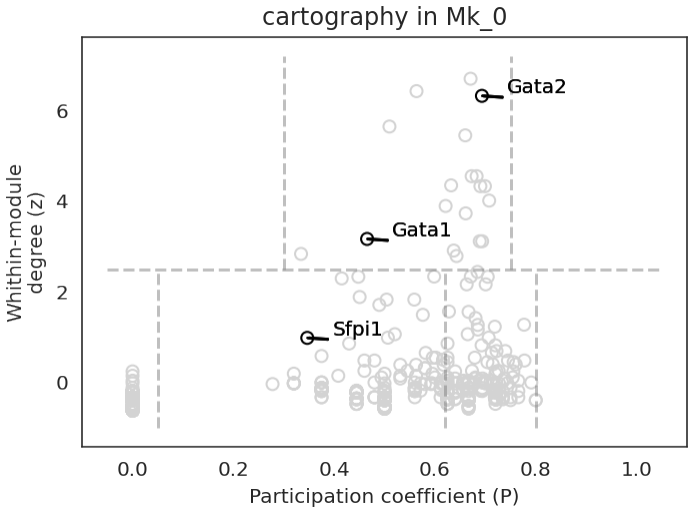

cartography in Gran_0
Gata2
Sfpi1
Gata1
Whithin-module degree (z)
Participation coefficient (P)
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6

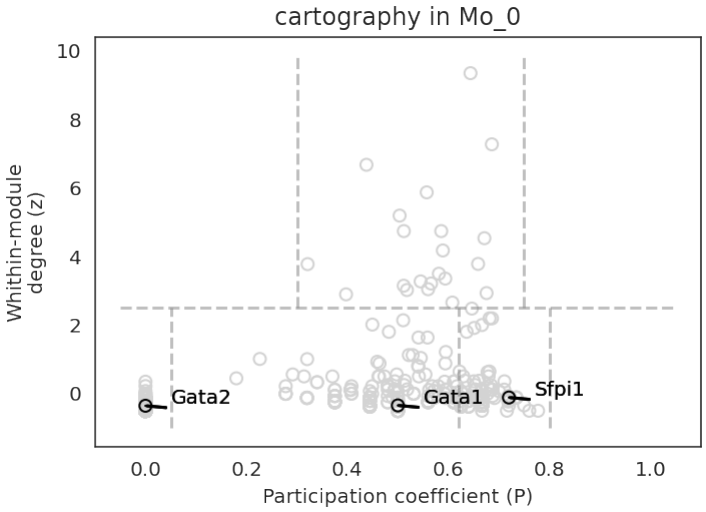

cartography in Gran_1
Gata1
Sfpi1
Gata2
Whithin-module degree (z)
Participation coefficient (P)
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8

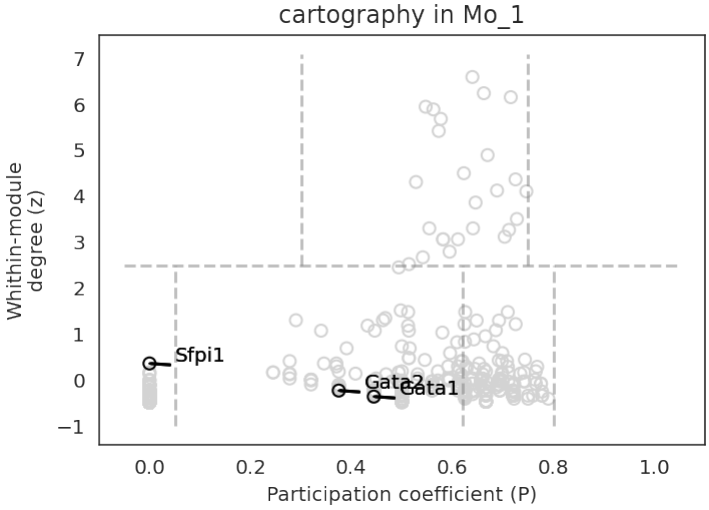

cartography in Gran_2
Gata1
Sfpi1
Gata2
Whithin-module degree (z)
Participation coefficient (P)
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8

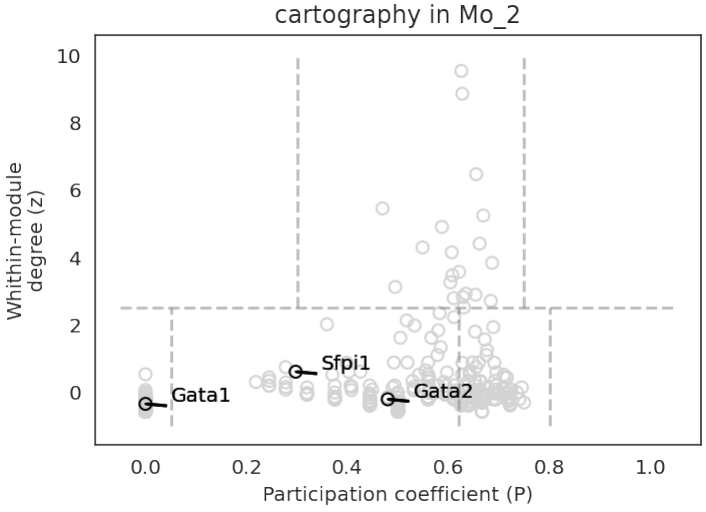

cartography in Gran_3
Whithin-module degree (z)
Participation coefficient (P)
Gata2


cartography in MEP_0
Whithin-module degree (z)
Participation coefficient (P)
Gata2
Gata1
Sfpi1

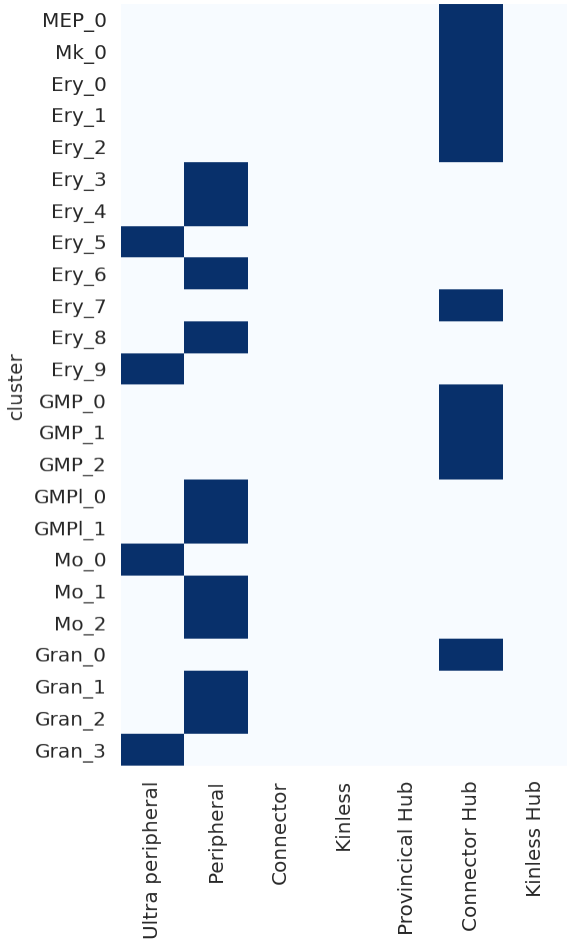

cartography in Mk_0
Whithin-module degree (z)
Participation coefficient (P)
Gata2
Gata1
Sfpi1


cartography in Mo_0
Whithin-module degree (z)
Participation coefficient (P)
Gata2
Gata1
Sfpi1


cartography in Mo_1
Whithin-module degree (z)
Participation coefficient (P)
Sfpi1
Gata2
Gata1


cartography in Mo_2
Whithin-module degree (z)
Participation coefficient (P)
Sfpi1
Gata1
Gata2

[22]:
```
plt.rcParams["figure.figsize"] = [4, 7]
# Plot the summary of cartography analysis
links.plot_cartography_term(goi="Gata2",
                            # save=f"{save_folder}/cartography",
                           )
```

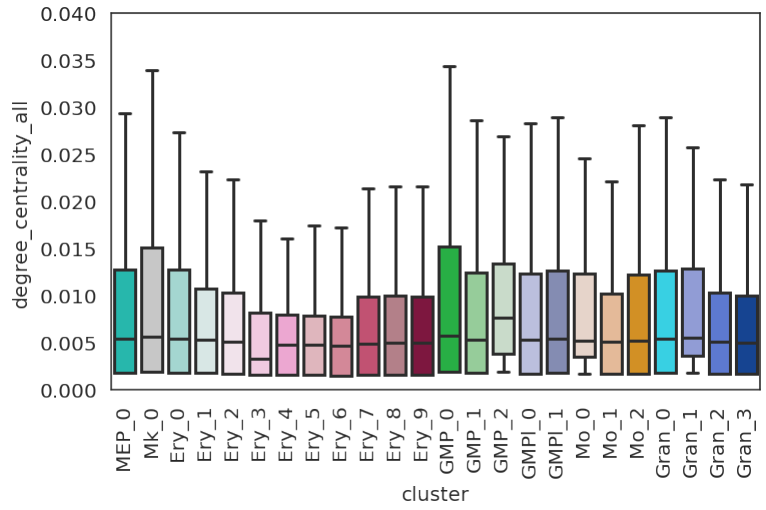


8. Network analysis; network score distribution

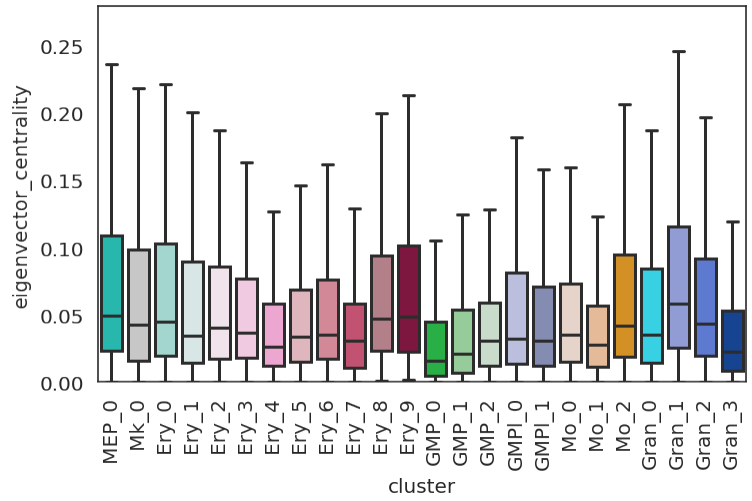
Next, we visualize the distribution of network score to get insight into the global trend of the GRNs.

[24]:
```
plt.rcParams["figure.figsize"] = [6, 4.5]
```

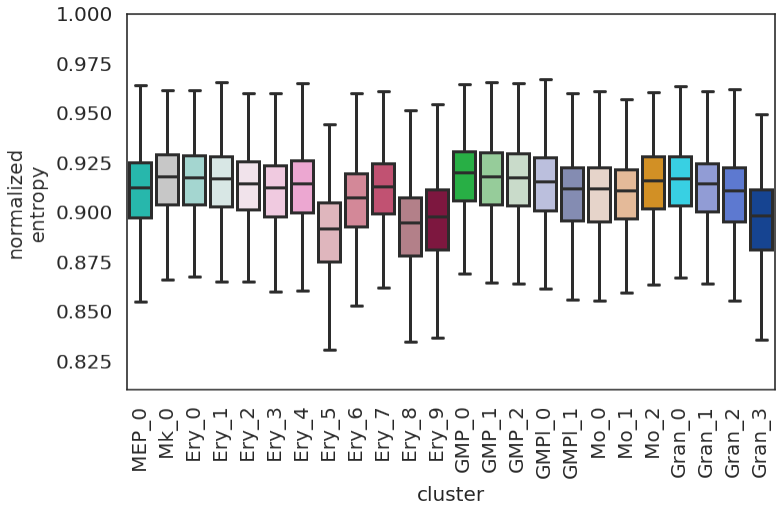
[25]:
```
plt.subplots_adjust(left=0.15, bottom=0.3)
plt.ylim([0,0.040])
links.plot_score_discributions(values=["degree_centrality_all"], method="boxplot",
                               #save=f"{save_folder}",
                              )
```



[69]:
```
plt.subplots_adjust(left=0.15, bottom=0.3)
plt.ylim([0, 0.28])
links.plot_score_discributions(values=["eigenvector_centrality"], method="boxplot",␣
↪save=f"{save_folder}")
```



[70]:
```
plt.subplots_adjust(left=0.15, bottom=0.3)
links.plot_network_entropy_distributions(save=f"{save_folder}")
```



Please go to next step: in silico gene perturbation with GRNs**

https://morris-lab.github.io/CellOracle.documentation/tutorials/simulation.html

[ ]:

in silico gene perturbation with GRNs

in silico gene perturbation with GRNs

`celloracle` leverage GRNs to simulate signal propagation inside a cell. We can estimate the effect of gene perturbation by the simulation with GRNs.

The jupyter notebook files and data used in this tutorial are available here .

Python notebook

Overview

This notebook describes how to do in silico TF perturbation using GRN models. Please read our paper first to know about the CellOracle algorithm.

Notebook file

Notebook file is available at CellOracle GitHub. https://github.com/morris-lab/CellOracle/blob/master/docs/notebooks/05_simulation/Gata1_KO_simulation_with_Paul_etal_2015_data.ipynb

Data

In this notebook, CellOracle uses two input data below for the GRN model construction.

- **Input data1: Oracle object**. Please look at the previous notebook to know how to make a `Oracle` object from scRNA-seq. https://morris-lab.github.io/CellOracle.documentation/notebooks/04_Network_analysis/Network_analysis_with_Paul_etal_2015_data.html

In this tutorial, we use demo data made from hematopoiesis scRNA-seq data. We can load the demo `Oracle` object as follows.

```
oracle = co.data.load_tutorial_oracle_object()
```

- **Input data2: Links object**. `Links` object is a class to store GRN data. We need GRN models stored as a `Links` object for simulation. In this tutorial, we use demo GRNs made from hematopoiesis scRNA-seq data and mouse sciATAC-seq atlas base-GRN. We can load the demo `Links` object as follows.

```
links = co.data.load_tutorial_links_object()
```

What you can do

In this notebook, we perform two analyzes.

1. **in silico TF perturbation** to simulate cell identity shift. CellOracle uses the GRN model to simulate cell identity shift in response to TF perturbation. For this analysis, you need to construct GRN models in this notebook first.
2. **Compare simulation vector with development vectors**. In order to properly interpret the simulation results, it is very important to consider the natural direction of development. First, I will show you how to get a pseudotime gradient vector field that represents the direction of development. Then, we compare the CellOracle TF perturbation simulation vector field with the development vector field by calculating the inner product score. See below for more information.

Custom data class / object

In this notebook, CellOracle uses four custom classes, `Oracle`, `Links`, `Gradient_calculator`, and `Oracle_development_module`.

- `Oracle` is the main class in the CellOracle package. It will do almost all calculations of GRN model construction and TF perturbation simulation.
- `Links` is a class to store GRN data.
- `Gradient_calculator` calculates development vector field using pseudotime information. We need the pseudotime data for this calculation. Please see another notebook how to get pseudotime data.
- `Oracle_development_module` integrates `Oracle` object data and `Gradient_calculator` object data to analyze how TF perturbation affects on the developmental process. It have many visualization functions.

0. Import libraries

[1]:
```
import os
import sys

import matplotlib.colors as colors
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scanpy as sc
import seaborn as sns
```

This notebook was made with celloracle version 0.7.0. Please use celloracle>=0.7.0. Otherwise you may get an error.

[2]:
```
import celloracle as co
co.__version__
```

[2]:
```
'0.7.0'
```

[3]:
```
#plt.rcParams["font.family"] = "arial"
plt.rcParams["figure.figsize"] = [6,6]
%config InlineBackend.figure_format = 'retina'
plt.rcParams["savefig.dpi"] = 600

%matplotlib inline
```

[4]:
```
# Make folder to save plots
save_folder = "figures"
os.makedirs(save_folder, exist_ok=True)
```

1. Load data

Load the oracle object. See the previous notebook for the notes on how to prepare the oracle object.

[5]:
```
# oracle = co.load_hdf5("ORACLE OBJECT PATH")

# Here, we load tutorial oracle object.
oracle = co.data.load_tutorial_oracle_object()
oracle
```

[5]:
```
Oracle object

Meta data
    celloracle version used for instantiation: 0.6.11
    n_cells: 2671
    n_genes: 1999
    cluster_name: louvain_annot
    dimensional_reduction_name: X_draw_graph_fa
    n_target_genes_in_TFdict: 21259 genes
    n_regulatory_in_TFdict: 1093 genes
    n_regulatory_in_both_TFdict_and_scRNA-seq: 90 genes
    n_target_genes_both_TFdict_and_scRNA-seq: 1850 genes
    k_for_knn_imputation: 66
Status
```

(continues on next page)

(continued from previous page)

```
    Gene expression matrix: Ready
    BaseGRN: Ready
    PCA calculation: Done
    Knn imputation: Done
    GRN calculation for simulation: Not finished
```

In the previous notebook, we calculated GRNs. Now, we will use these GRNs for simulation. We import GRNs which were saved in the Links object.

[6]:
```
# links = co.load_hdf5("YOUR LINK OBJCT PATH")

# Here, we load demo links object for the training purpose.
links = co.data.load_tutorial_links_object()
```

2. Make predictive models for simulation

We will fit ridge regression models again. This process takes less time than the GRN inference in the previous notebook because we use already filtered GRN models.

[7]:
```
links.filter_links()
oracle.get_cluster_specific_TFdict_from_Links(links_object=links)
oracle.fit_GRN_for_simulation(alpha=10, use_cluster_specific_TFdict=True)
```

```
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
```

```
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
HBox(children=(FloatProgress(value=0.0, max=1999.0), HTML(value='')))
```

3. in silico TF Perturbation analysis

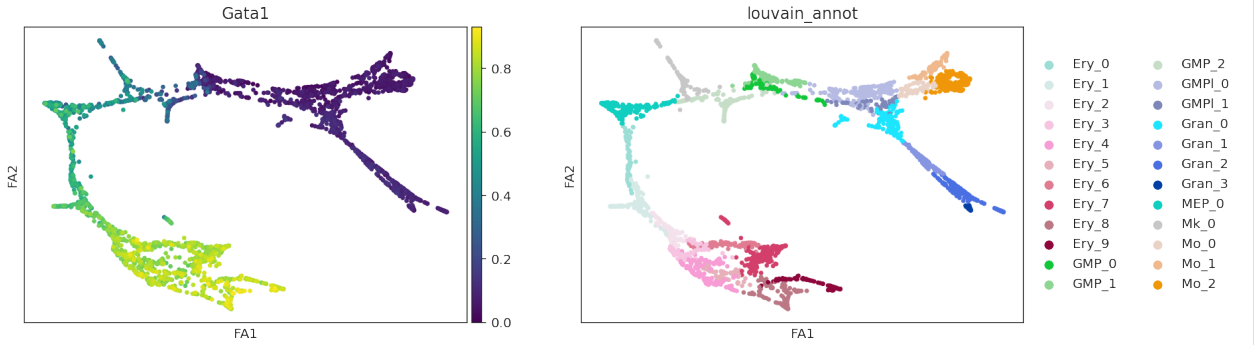
Next, we will simulate the TF perturbation effects on cell identity to investigate its function and regulatory mechanism. See the celloracle paper for the details and scientific premise on the algorithm.

In this notebook, we'll show an example of the simulation; we'll simulate knock-out of Gata1 gene in the hematopoiesis trajectory.

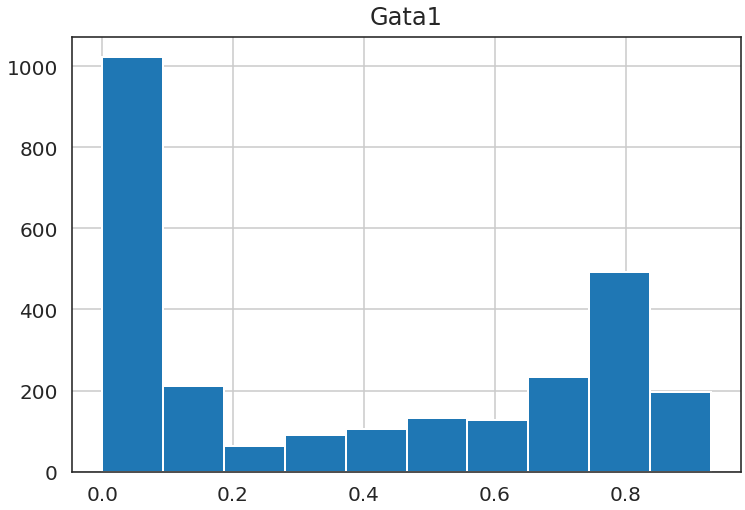
Previous studies have shown that Gata1 is one of the TFs that regulates cell fate decisions in myeloid progenitors. Additionally, Gata1 has been shown to affect erythroid cell differentiation.

Here, we will analyze Gata1 for the demonstration of celloracle; Celloracle try to recapitulate the previous findings of Gata1 gene above.

[8]:
```
# Check gene expression
goi = "Gata1"
sc.pl.draw_graph(oracle.adata, color=[goi, oracle.cluster_column_name],
                 layer="imputed_count", use_raw=False, cmap="viridis")
```



[9]:
```
# Plot gene expression in histogram
sc.get.obs_df(oracle.adata, keys=[goi], layer="imputed_count").hist()
plt.show()
```



- You can use any gene expression value to enter in silico perturbations, but please avoid extremely high values that are far from the natural gene expression range. The upper limit allowed is twice the maximum gene expression.

Here we simulate Gata1 KO; we predict what happens to the cells if Gata1 gene expression changed into 0.

[10]:
```
# Enter perturbation conditions to simulate signal propagation after the perturbation.
oracle.simulate_shift(perturb_condition={goi: 0.0},
                      n_propagation=3)
```

```
Variability score of Gene Gata1 is too low. Simulation accuracy may be poor with this␣
↪gene.
```

- The steps above simulated global future gene expression shift after perturbation. This prediction is based on iterative calculations of signal propagation within the GRN. Please look at our paper for more information.
- The next step is to calculate the probability of cell state transitions based on the simulation data. You can use the transition probabilities between cells to predict how cells will change after a perturbation.
- This transition probability will be used later.

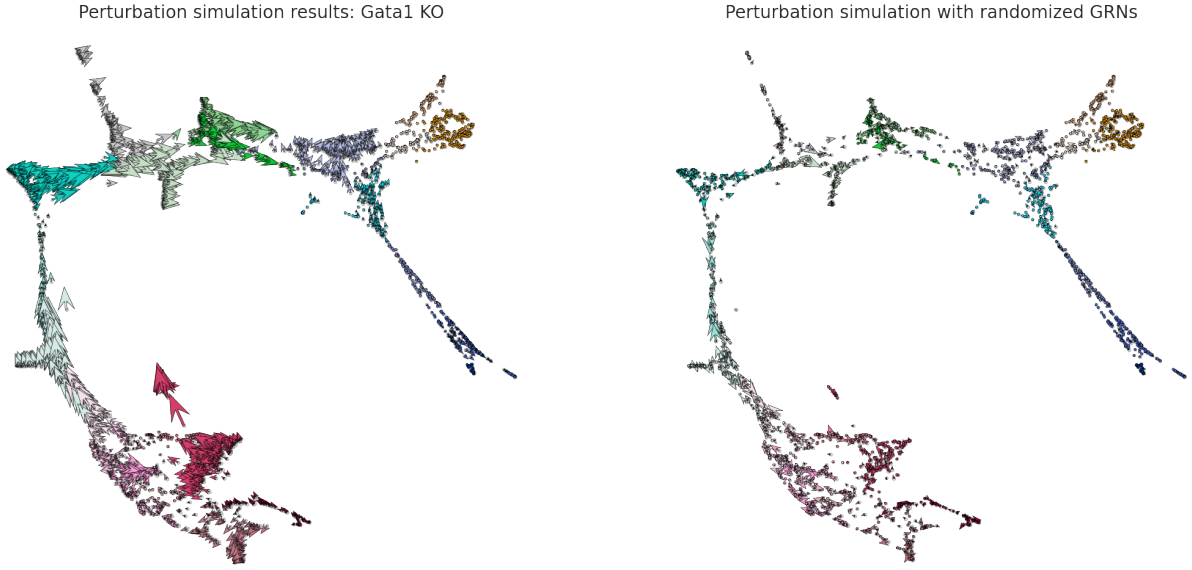
[11]:
```
# Get transition probability
oracle.estimate_transition_prob(n_neighbors=200,
                                knn_random=True,
                                sampled_fraction=1)

# Calculate embedding
oracle.calculate_embedding_shift(sigma_corr = 0.05)
```

4. Visualization

Caution: It is very important to find optimal `scale` parameter.

- We need to adjust the `scale` parameter. Please seek to find the optimal `scale` parameter that provides good visualization.
- If you don't see any vector, you can try the smaller `scale` parameter to magnify vector length. However, if you see large vectors in the right panel, which is a randomized simulation, it means that the scale parameters are too small.

[12]:
```
fig, ax = plt.subplots(1, 2,  figsize=[15, 7])

scale = 25
# Show quiver plot
oracle.plot_quiver(scale=scale, ax=ax[0])
ax[0].set_title(f"Perturbation simulation results: {goi} KO")

# Show quiver plot that was calculated with randomized GRN.
oracle.plot_quiver_random(scale=scale, ax=ax[1])
ax[1].set_title(f"Perturbation simulation with randomized GRNs")

plt.show()
```



4.2. Vector field graph

We can visualize simulation result as a vector field graph. Single cell transition vectors are grouped by grid point.

4.2.1 Find parameters for n_grid and min_mass

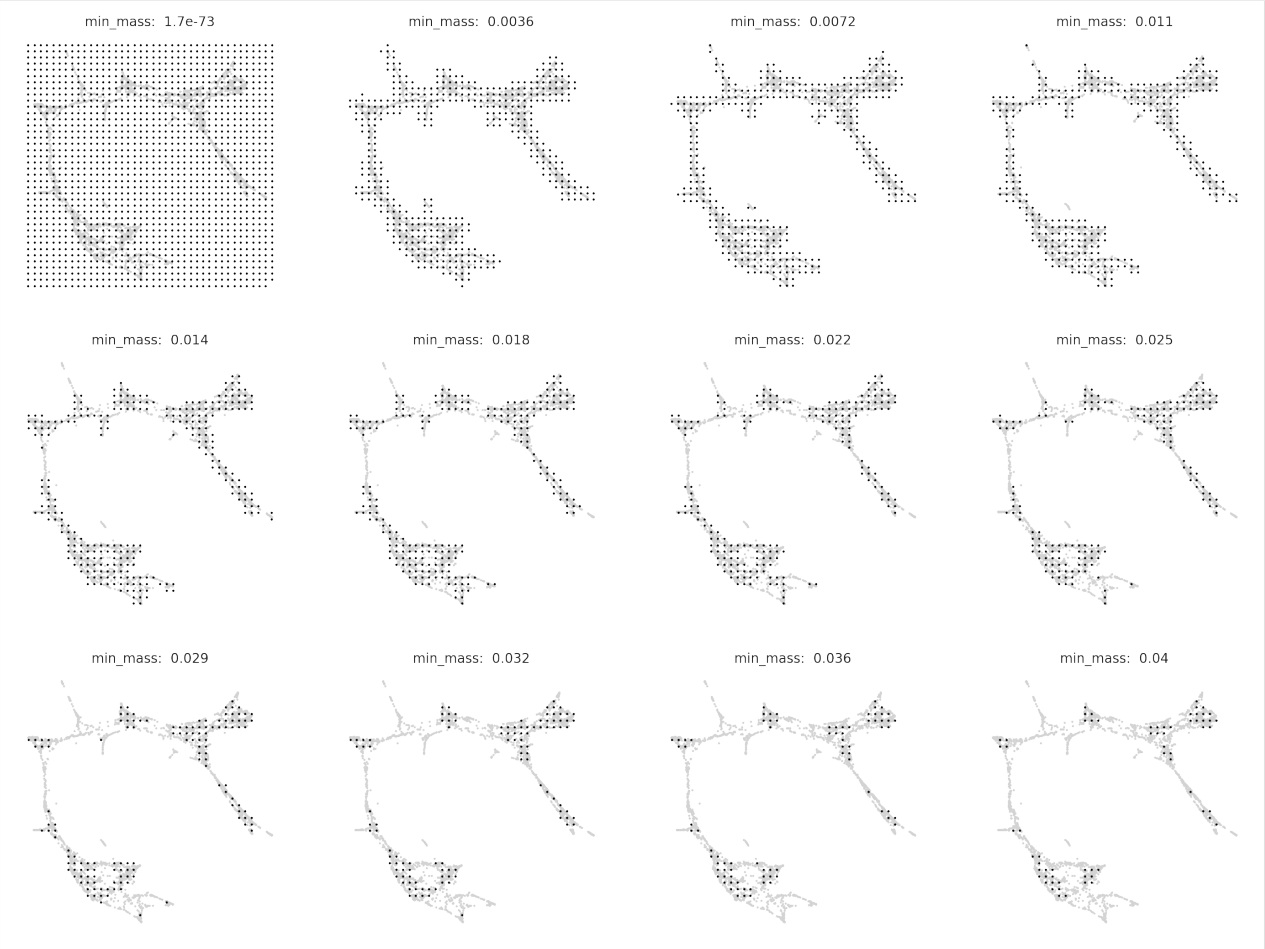
`n_grid`: Number of grid point.

`min_mass`: Threshold value for the cell density The appropriate values for these parameters depends on the data. Please find appropriate values as follows.

[13]:
```
# n_grid = 40 is a good point to start with.
n_grid = 40
oracle.calculate_p_mass(smooth=0.8, n_grid=n_grid, n_neighbors=200)
```

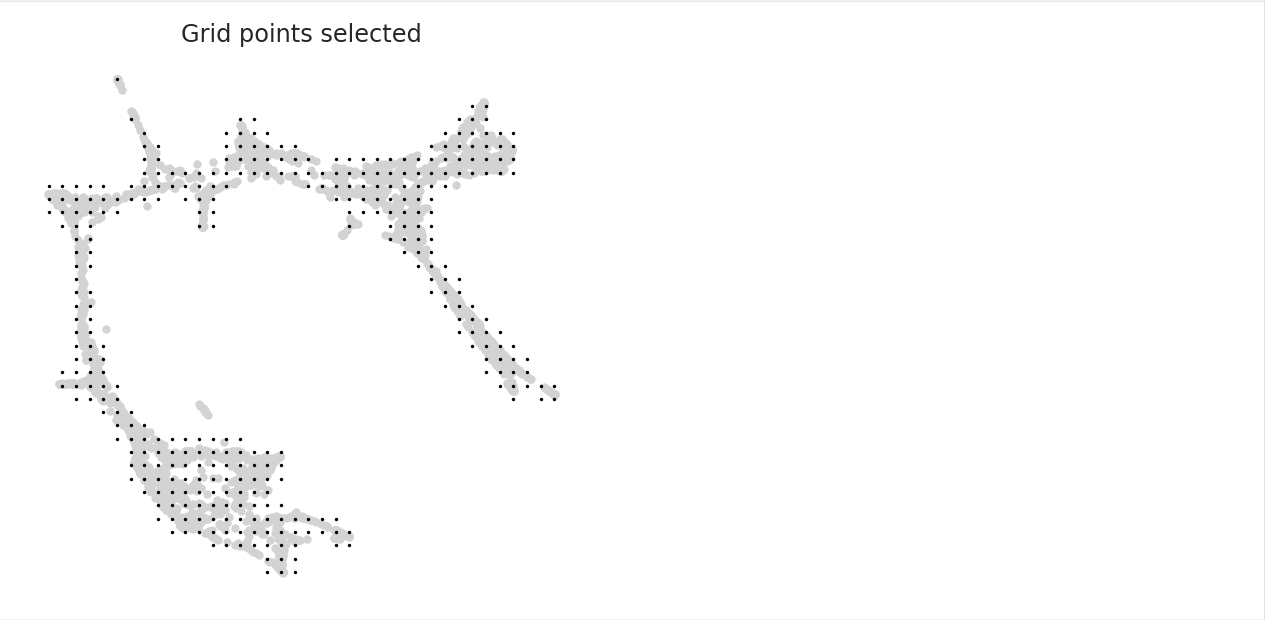
Please run `oracle.suggest_mass_thresholds()` to find appropriate `min_mass` parameter. It will give you some examples.

[14]:
```
# Search for best min_mass.
oracle.suggest_mass_thresholds(n_suggestion=12)
```



According to the results, the appropriate `min_mass` is around 0.011.

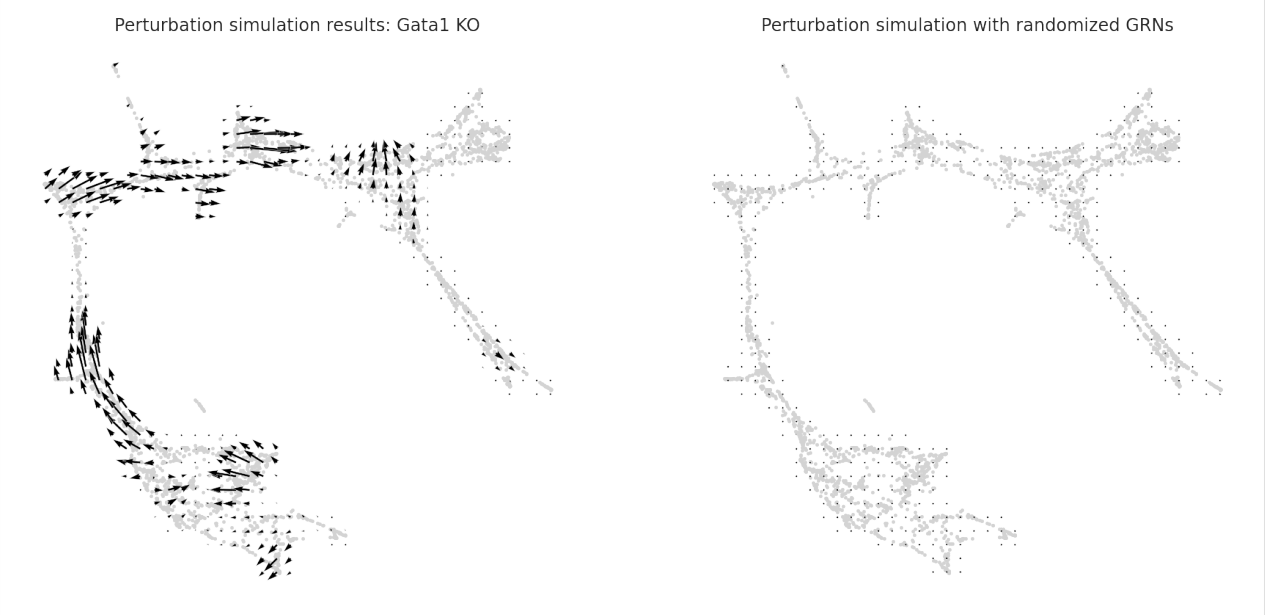
[15]:
```
min_mass = 0.01
oracle.calculate_mass_filter(min_mass=min_mass, plot=True)
```



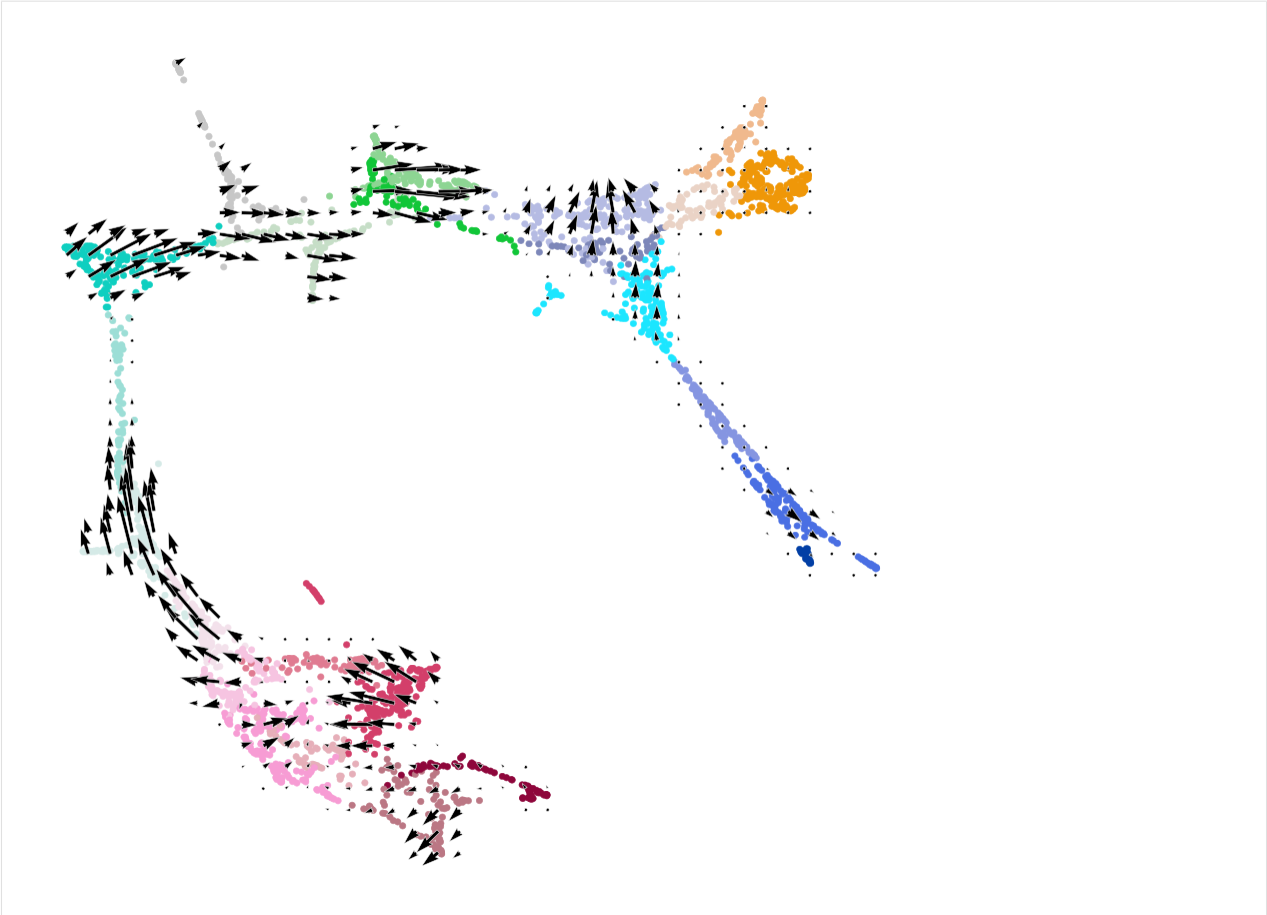
4.2.2 Plot vector fields

- Again, we need to adjust the `scale` parameter. Please seek to find the optimal `scale` parameter that provides good visualization.
- If you don't see any vector, you can try the smaller `scale` parameter to magnify vector length. However, if you see large vectors in the right panel, which is a randomized simulation, it means that the scale parameters are too small.

[17]:
```
fig, ax = plt.subplots(1, 2,  figsize=[15, 7])

scale_simulation = 0.5
# Show quiver plot
oracle.plot_simulation_flow_on_grid(scale=scale_simulation, ax=ax[0])
ax[0].set_title(f"Perturbation simulation results: {goi} KO")

# Show quiver plot that was calculated with randomized GRN.
oracle.plot_simulation_flow_random_on_grid(scale=scale_simulation, ax=ax[1])
ax[1].set_title(f"Perturbation simulation with randomized GRNs")

plt.show()
```



[19]:
```
# Plot vector field with cell cluster
fig, ax = plt.subplots(figsize=[8, 8])

oracle.plot_cluster_whole(ax=ax, s=10)
oracle.plot_simulation_flow_on_grid(scale=scale_simulation, ax=ax, show_
↪background=False)
```



5. Compare simulation vector with development vectors

- As shown above, we can use celloracle's simulation to infer how TF perturbations affect cell identity. The simulation results are provided in the form of a vector field map.
- To interpret the results, it is necessary to take into account the direction of natural differentiation. We will compare the simulated perturbation vectors with the development vector. By comparing them, we can intuitively understand how TF is involved in cell fate determination during development. This perspective is also important for the estimation of experimental perturbation results
- Here, we show an example to calculate the vector field of development using **pseudotime gradient**. In short, the process is as follows.

1. Transfer **pseudotime data** into n x n grid point.
2. Calculate the 2D gradient of pseudotime to get vector field
3. Compare in silico TF perturbation vector field with development vector field by calculating inner product between these two vectors.

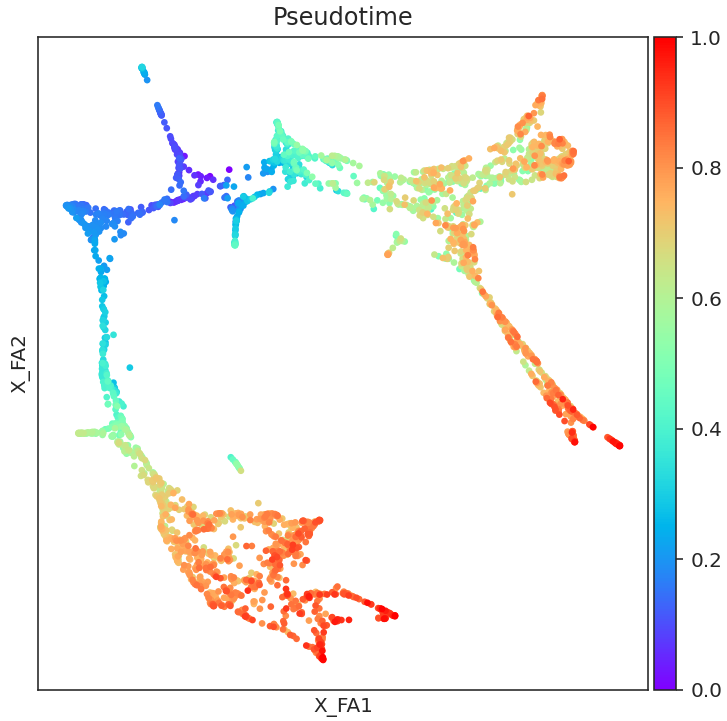
- Also, there are many other options to get vector field of development flow from scRNA-seq data, and you can select another option. For example, RNA velocity analysis is a good way to estimate the direction of cell differentiation. Choose the method that best suits your data.

In the analysis below, we need to use **pseudotime** data. Pseudotime data is included in the demo data. **If you try to analyze your scRNA-seq data, please calculate pseudotime before starting this analysis.**

We provide a tutorial notebook introducing how to calculate pseudotime. https://morris-lab.github.io/CellOracle.documentation/tutorials/pseudotime.html

We use pseudotime data for an input of this analysis. Pleas calculate continuous pseudotime in advance. Please look at another notebook for details on how to calculate pseudotime.

[20]:
```
# Visualize pseudotime
fig, ax = plt.subplots(figsize=[6,6])

sc.pl.embedding(adata=oracle.adata, basis=oracle.embedding_name, ax=ax, cmap="rainbow
↪",
                color=["Pseudotime"])
```



[21]:
```
from celloracle.applications import Gradient_calculator

# Instantiate Gradient calculator object
gradient = Gradient_calculator(oracle_object=oracle, pseudotime_key="Pseudotime")
```

We need to select `n_grid` and `min_mass` to make grid point. `n_grid`: Number of grid point.

We already know approproate values for them. Please set the same values as step 4.2.1 above.

[22]:
```
gradient.calculate_p_mass(smooth=0.8, n_grid=n_grid, n_neighbors=200)
gradient.calculate_mass_filter(min_mass=min_mass, plot=True)
```

Next, we will transfer pseudotime data into grid points. For this calculation we can chose two method.

- knn: K-Nearesr Neighbor regressor. You need to set number of neighbor. Please adjust `n_knn` searching for best results.

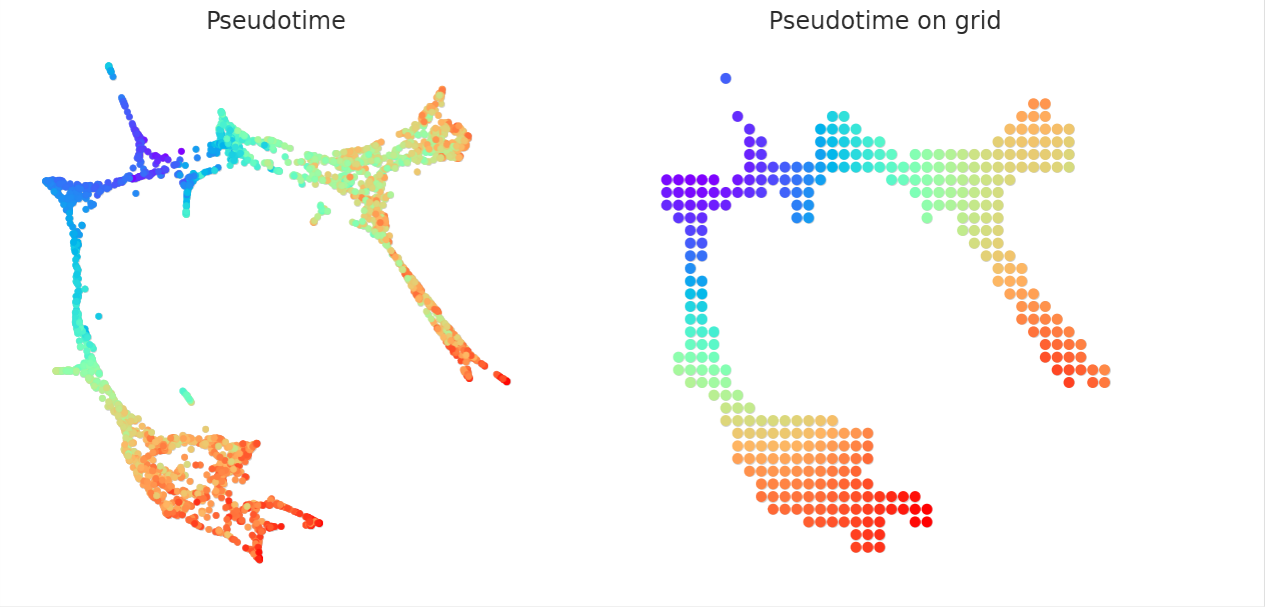
```
gradient.transfer_data_into_grid(args={"method": "knn", "n_knn":50})
```

- polynomial: Polynomial regression using x-axis and y-axis of dimensional reduction space.

In general, this method will be more robust. Please use this method if k-nn does not work. `n_poly` is the number of degree for the polynomial regression model. Please try to find appropriate`n_poly` searching for best results.

```
gradient.transfer_data_into_grid(args={"method": "polynomial", "n_poly":3})
```

[23]:
```
gradient.transfer_data_into_grid(args={"method": "polynomial", "n_poly":3}, plot=True)
```

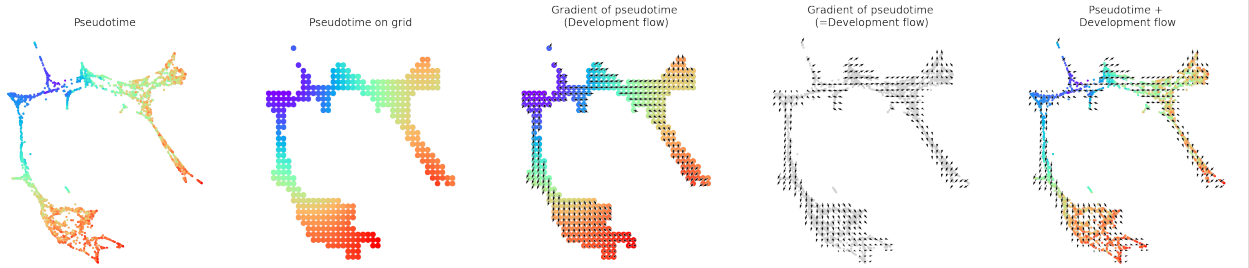


Calculate 2D vector map that represents the gradient of pseudotime. After the gradient calculation, the length of the

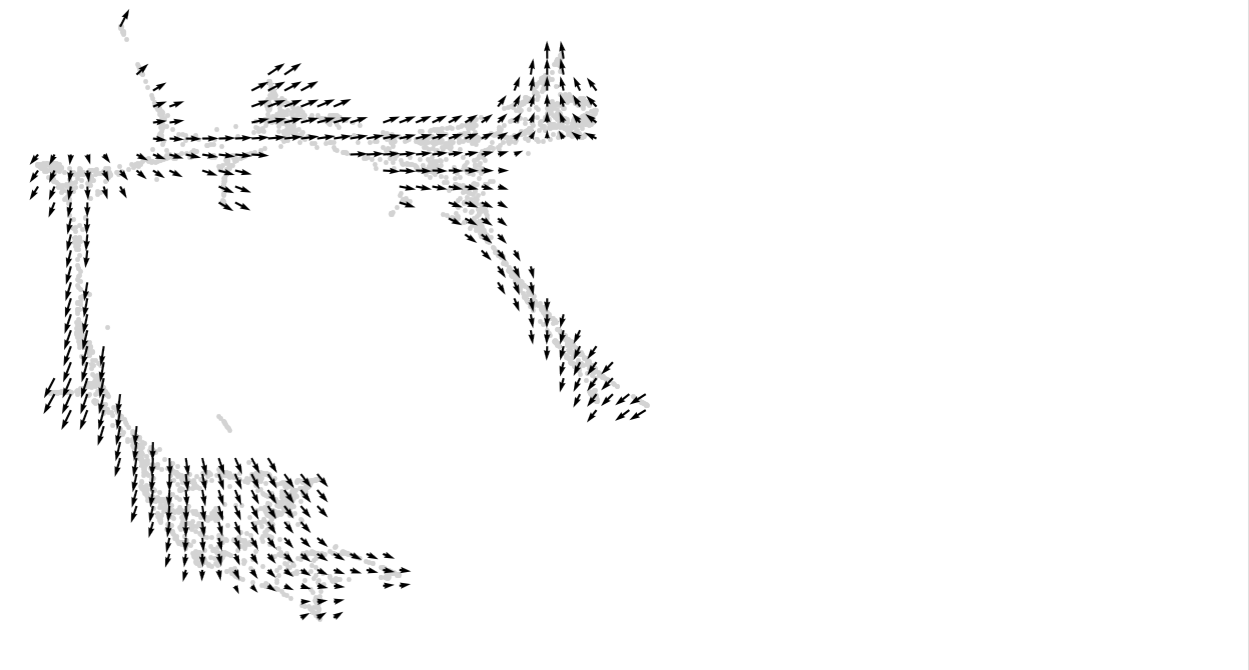
vector will be normalized automatically.

Please adjust `scale` parameter to adjust vector length.

[24]:
```
# Calculate graddient
gradient.calculate_gradient()

# Show results
scale_dev = 40
gradient.visualize_results(scale=scale_dev, s=5)
```



[25]:
```
# Visualize results
fig, ax = plt.subplots(figsize=[6, 6])
gradient.plot_dev_flow_on_grid(scale=scale_dev, ax=ax)
```

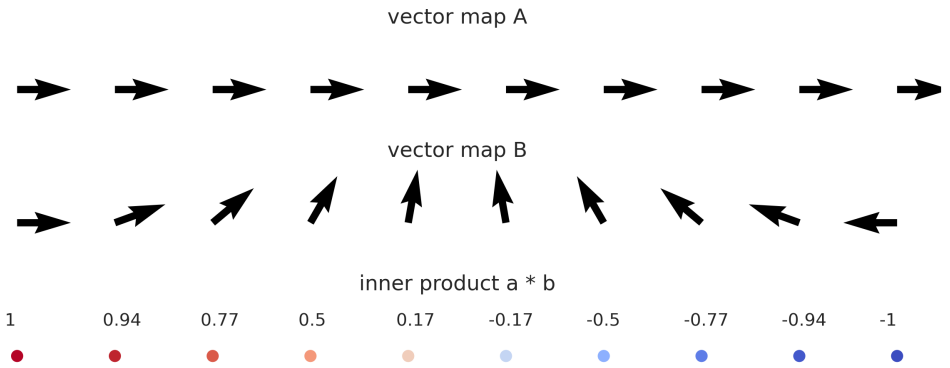


[26]:
```
# Save gradient object if you want.
#gradient.to_hdf5("../data/Paul_etal.celloracle.gradient")
```

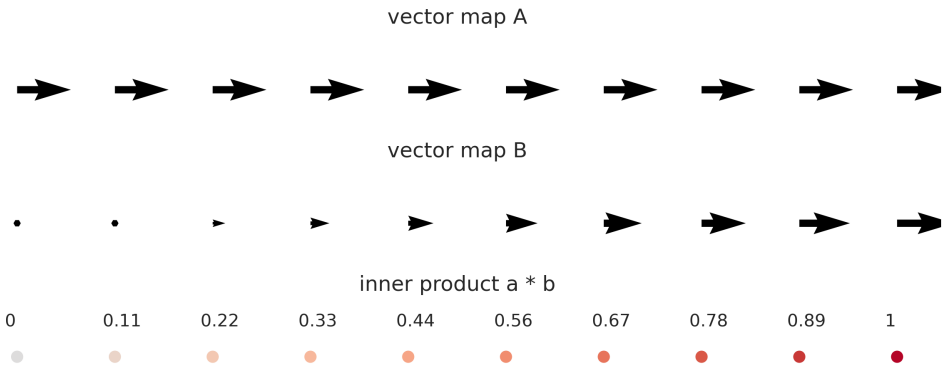
We will use the inner product to compare the 2D vector map of perturb-simulation and development quantitatively. > If you are not familiar with Inner product / Dot product, please see https://en.wikipedia.org/wiki/Dot_product

- The inner product represents the similarity between two vectors.
- Using the inner product, we compare the 2D vector field of perturbation simulation and development flow.

- Inner product can be a positive value when two vectors are pointing in the same direction.
- Inner product can be a negative value when two vectors are pointing in the opposite direction.



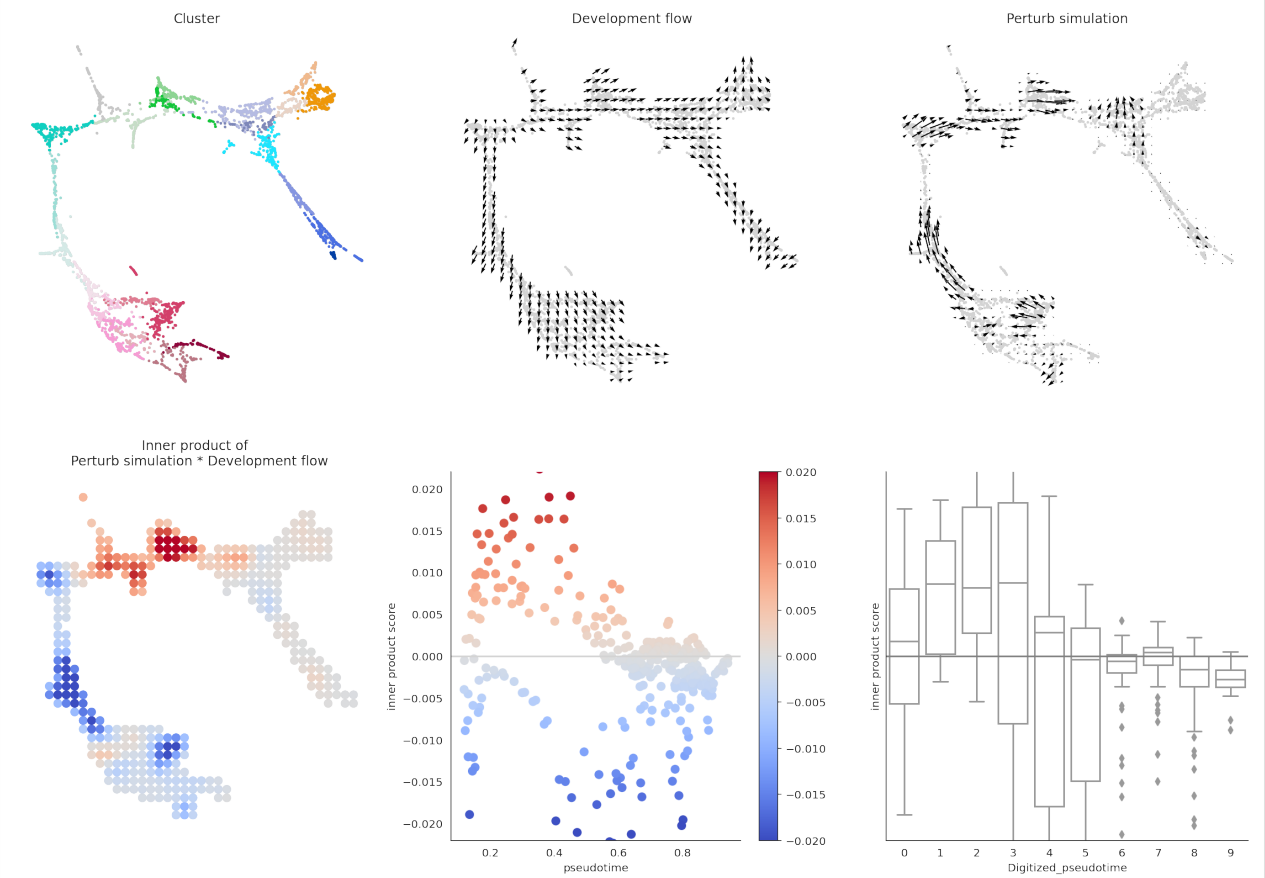
- The length of vector also affects the absolute value of inner product value.



In summary, - **a negative inner product** means that perturbation might **block differentiation**. - **a positive inner product** means that perturbation might **promote differentiation**.

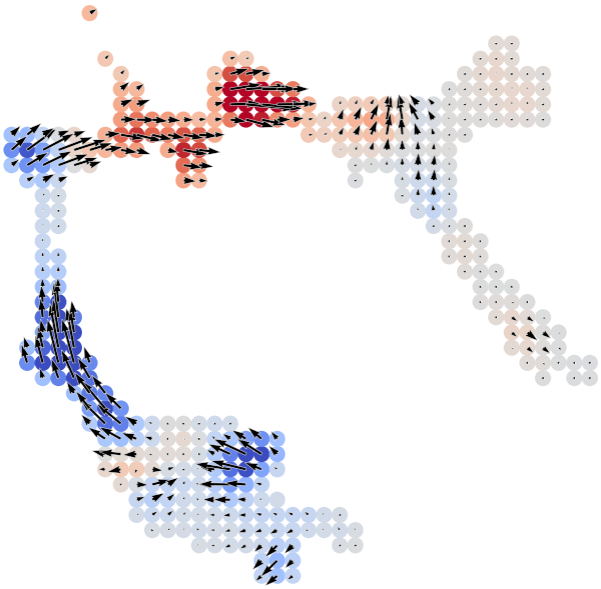
[27]:
```
from celloracle.applications import Oracle_development_module

# Make Oracle_development_module to compare two vector field
dev = Oracle_development_module()

# Load development flow
dev.load_differentiation_reference_data(gradient_object=gradient)

# Load simulation result
dev.load_perturb_simulation_data(oracle_object=oracle)


# Calculate inner produc scores
dev.calculate_inner_product()
dev.calculate_digitized_ip(n_bins=10)
```

[28]:
```
# Let's visualize the results
dev.visualize_development_module_layout_0(s=5,
                                          scale_for_simulation=scale_simulation,
                                          s_grid=50,
                                          scale_for_pseudotime=scale_dev,
                                          vm=0.02)
```



[29]:
```
# Show inner product score
fig, ax = plt.subplots(figsize=[6, 6])
dev.plot_inner_product_on_grid(vm=0.02, s=50, ax=ax)
```

[30]:
```
# Show inner product score with perturbation simulation vector field
fig, ax = plt.subplots(figsize=[6, 6])
dev.plot_inner_product_on_grid(vm=0.02, s=50, ax=ax)
dev.plot_simulation_flow_on_grid(scale=scale_simulation, show_background=False, ax=ax)
```



6. Focus on a single development lineage to interpret the results in detail

So far, we have used `Oracle_development_module` to analyze the whole cell population. If you input the index for the cells of interest, Oracle_development_module will analyze subset data.

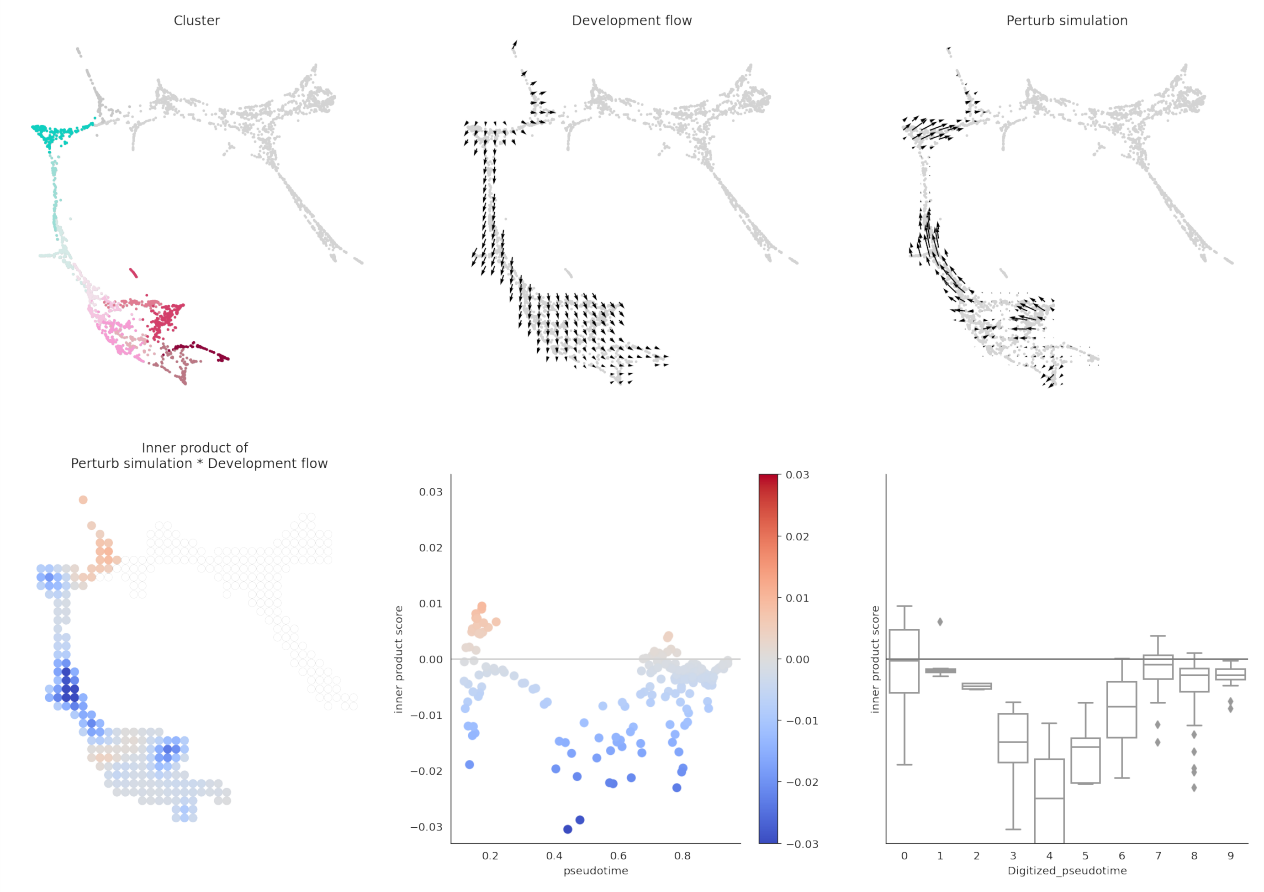
In this example, let's analyze MEP and its progenies.

[31]:
```
# Get cell index list for the cells of interest
clusters = ['Ery_0', 'Ery_1', 'Ery_2', 'Ery_3', 'Ery_4', 'Ery_5', 'Ery_6',
            'Ery_7', 'Ery_8', 'Ery_9', 'MEP_0', 'Mk_0']
cell_idx = np.where(oracle.adata.obs["louvain_annot"].isin(clusters))[0]

# Check
print(cell_idx)
```

```
[   0    2    4 ... 2666 2668 2670]
```

[32]:
```
dev = Oracle_development_module()

# Load development flow
dev.load_differentiation_reference_data(gradient_object=gradient)

# Load simulation result
dev.load_perturb_simulation_data(oracle_object=oracle,
                                 cell_idx_use=cell_idx, # Enter cell id list
                                 name="Lineage_MEP" # Name of this cell group. You␣
↪can enter arbitrary name.
                                 )

# Calculate stats
dev.calculate_inner_product()
dev.calculate_digitized_ip(n_bins=10)
```

[33]:
```
# Let's visualize the results
dev.visualize_development_module_layout_0(s=5,
                                          scale_for_simulation=scale_simulation,
                                          s_grid=50,
                                          scale_for_pseudotime=scale_dev,
                                          vm=0.03)
```



[ ]:

[ ]:

Prepare input data

scRNA-seq data preparation

Overview

In advance to CellOrale analysis, scRNA-seq data should be processed. Please prepare scRNA-seq data as an *anndata* using *scanpy*.

Note: *scanpy* is a python toolkit for scRNA-seq data analysis. If you are new to scanpy, pelase read the documentation to learn it in advance.

- scanpy documentation: https://scanpy.readthedocs.io/en/stable/
- anndata documentation: https://anndata.readthedocs.io/en/latest/

Warning: In this section, we intend to introduce an example of how to prepare the **input data** for CellOracle analysis. **This is NOT the CellOracle analysis itself.** We do NOT use celloracle in this notebook.

A. scRNA-seq data preprocessing with scanpy

Please download notebooks from here . Or please click below to view the content.

Overview

This notebook will show an example of how to process scRNA-seq data using scRNA-seq data of hematopoiesis. (Paul, F., Arkin, Y., Giladi, A., Jaitin, D. A., Kenigsberg, E., Keren-Shaul, H., et al. (2015). Transcriptional Heterogeneity and Lineage Commitment in Myeloid Progenitors. Cell, 163(7), 1663–1677. http://doi.org/10.1016/j.cell.2015.11.013).

You can easily download this scRNA-seq data with a scanpy function.

Notebook file

Notebook file is available at CellOracle GitHub. https://github.com/morris-lab/CellOracle/blob/master/docs/notebooks/03_scRNA-seq_data_preprocessing/scanpy_preprocessing_with_Paul_etal_2015_data.ipynb

Steps

We need to do following scRNA-seq processing.

1. **Variable gene selection and normalization.**
2. **Log transformation.** Although we need to do log-transformation, CellOracle also needs the raw gene expression value in later process. We keep raw count data in a layer of anndata.
3. **Cell clustering.**
4. **Dimensional reduction.** We need to prepare 2D embedding data. Make sure that the 2D embedding properly represents the identity of the cell. Good Cell Oracle simulation results cannot be obtained if there is biologically inappropriate embedded data.

Caution

- This notebook is intended to explain **how to prepare the input data for CellOracle analysis**. This is **NOT** the CellOracle analysis itself. Also, this notebook does NOT use `celloracle` in this notebook.
- Instead, we use `scanpy` and `anndata` to process and store scRNA-seq data. If you are new to these packages, pelase read the documentation to learn them in advance.
- `scanpy` documentation: https://scanpy.readthedocs.io/en/stable/
- `anndata` documentation: https://anndata.readthedocs.io/en/latest/

0. Import libraries

[1]:
```
import os
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scanpy as sc
```

[9]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
plt.rcParams["savefig.dpi"] = 300
plt.rcParams["figure.figsize"] = [6, 4.5]
```

1. Load data

[3]:
```
# Download dataset. You can change the code blow if you use another data.
adata = sc.datasets.paul15()
```

```
WARNING: In Scanpy 0.*, this returned logarithmized data. Now it returns non-
↪logarithmized data.
```

```
... storing 'paul15_clusters' as categorical
Trying to set attribute `.uns` of view, making a copy.
```

2. Filtering

[4]:
```
# Only consider genes with more than 1 count
sc.pp.filter_genes(adata, min_counts=1)
```

3. Normalization

[5]:
```
# Normalize gene expression matrix with total UMI count per cell
sc.pp.normalize_per_cell(adata, key_n_counts='n_counts_all')
```

4. Identification of highly variable genes

This step is essential. Please do not skip this step.

> By removing non-variable genes, we can reduce the calculation time during the GRN reconstruction and simulation. Also, it will improve the accuracy of GRN inference by removing noisy genes. We recommend using the top 2000~3000 variable genes.

[6]:
```
# Select top 2000 highly-variable genes
filter_result = sc.pp.filter_genes_dispersion(adata.X,
                                              flavor='cell_ranger',
                                              n_top_genes=2000,
                                              log=False)
```

(continues on next page)

(continued from previous page)

```
# Subset the genes
adata = adata[:, filter_result.gene_subset]

# Renormalize after filtering
sc.pp.normalize_per_cell(adata)
```

```
Trying to set attribute `.obs` of view, making a copy.
```

5. Log transformation

- We will do log transformation and scaling because these are necessary for PCA, clustering, and differential gene calculations.
- We also need non-transformed gene expression data for celloracle analysis. Thus, **we need to keep gene expression data as a separate layer of anndata before the log transformation.**

```
adata.layers["raw_count"] = adata.raw.X.copy()
```

[7]:
```
# keep raw cont data before log transformation
adata.raw = adata
adata.layers["raw_count"] = adata.raw.X.copy()


# Log transformation and scaling
sc.pp.log1p(adata)
sc.pp.scale(adata)
```

6. PCA and find neighbors

This step is necessary to perform later dimensional reduction and clustering.

[ ]:
```
# PCA
sc.tl.pca(adata, svd_solver='arpack')

# Diffusion map
sc.pp.neighbors(adata, n_neighbors=4, n_pcs=20)

sc.tl.diffmap(adata)
# Calculate neihbors again based on diffusionmap
sc.pp.neighbors(adata, n_neighbors=10, use_rep='X_diffmap')
```

7. Cell clustering

[11]:
```
sc.tl.louvain(adata, resolution=0.8)
```

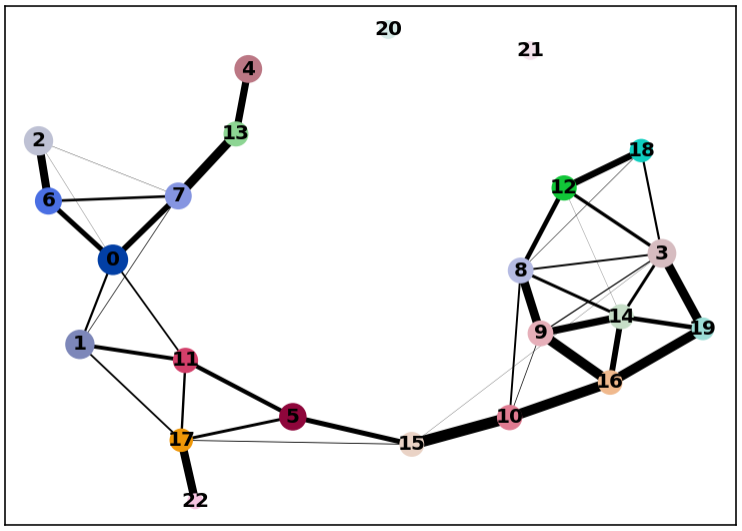
8. Dimensional reduction using PAGA and Force

- Dimensional reduction is one of the most important parts of the scRNA-seq analysis.Celloracle needs dimensional reduction embeddings to simulate cell transition.
- Please choose a proper algorithm for dimensional reduction for your scRNA-seq data so that the embedding appropriately represents its developmental trajectory. We recommend using one of the following dimensional reduction algorithms (or trajectory inference algorithms)
- UMAP: https://scanpy.readthedocs.io/en/stable/generated/scanpy.tl.umap.html#scanpy.tl.umap
- TSNE: https://scanpy.readthedocs.io/en/stable/generated/scanpy.tl.tsne.html#scanpy.tl.tsne
- Diffusion map: https://scanpy.readthedocs.io/en/stable/generated/scanpy.tl.diffmap.html#scanpy.tl.diffmap
- Force-directed graph drawing: https://scanpy.readthedocs.io/en/stable/generated/scanpy.tl.draw_graph.html#scanpy.tl.draw_graph
- In this example, we use a workflow introduced in the scanpy trajectory inference tutorial. https://scanpy-tutorials.readthedocs.io/en/latest/paga-paul15.html This method is combination of three algorithms:diffusion map, force-directed graph, PAGA.
- Step1: Calculate PAGA graph. PAGA data will be used for the initial status of force-directed graph calculation.
- Step2: Force-directed graph calculation.

[12]:
```
# PAGA graph construction
sc.tl.paga(adata, groups='louvain')
```

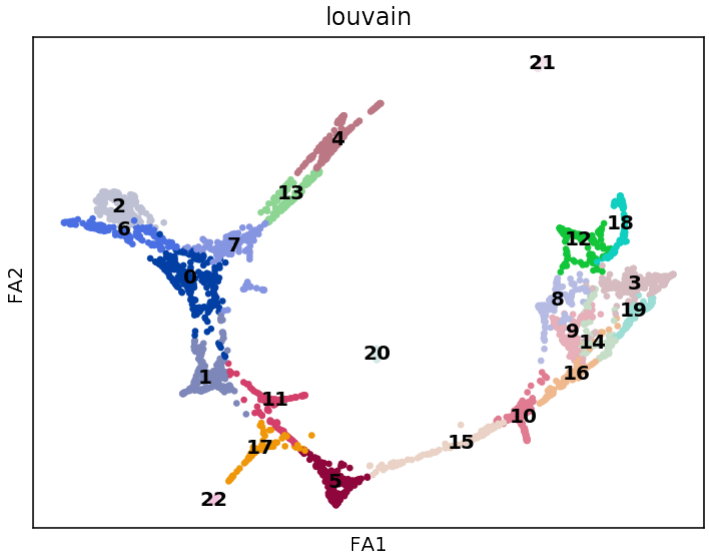
[14]:
```
plt.rcParams["figure.figsize"] = [6, 4.5]
```

[15]:
```
sc.pl.paga(adata)
```



[16]:
```
sc.tl.draw_graph(adata, init_pos='paga', random_state=123)
```

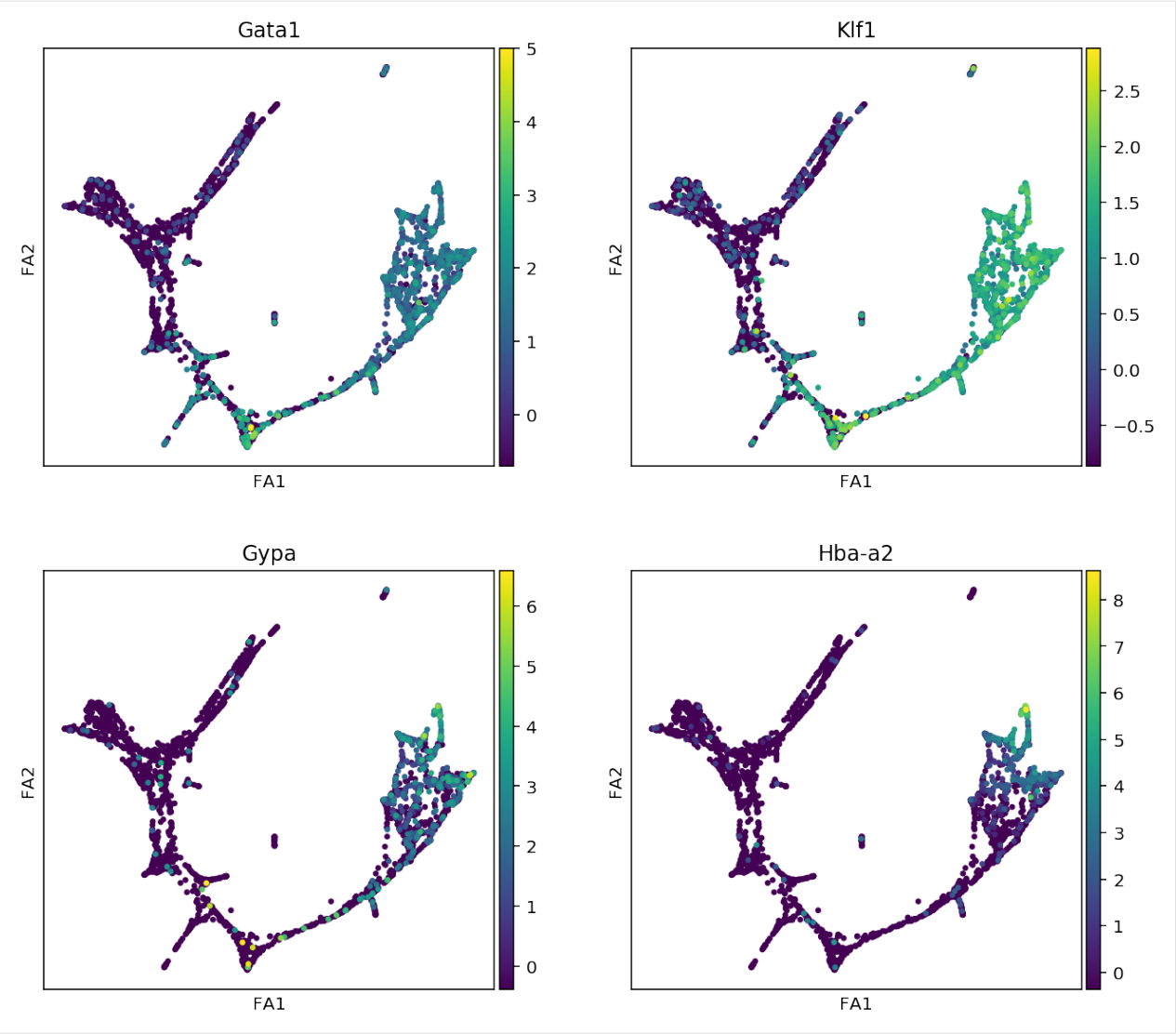
[17]:
```
sc.pl.draw_graph(adata, color='louvain', legend_loc='on data')
```

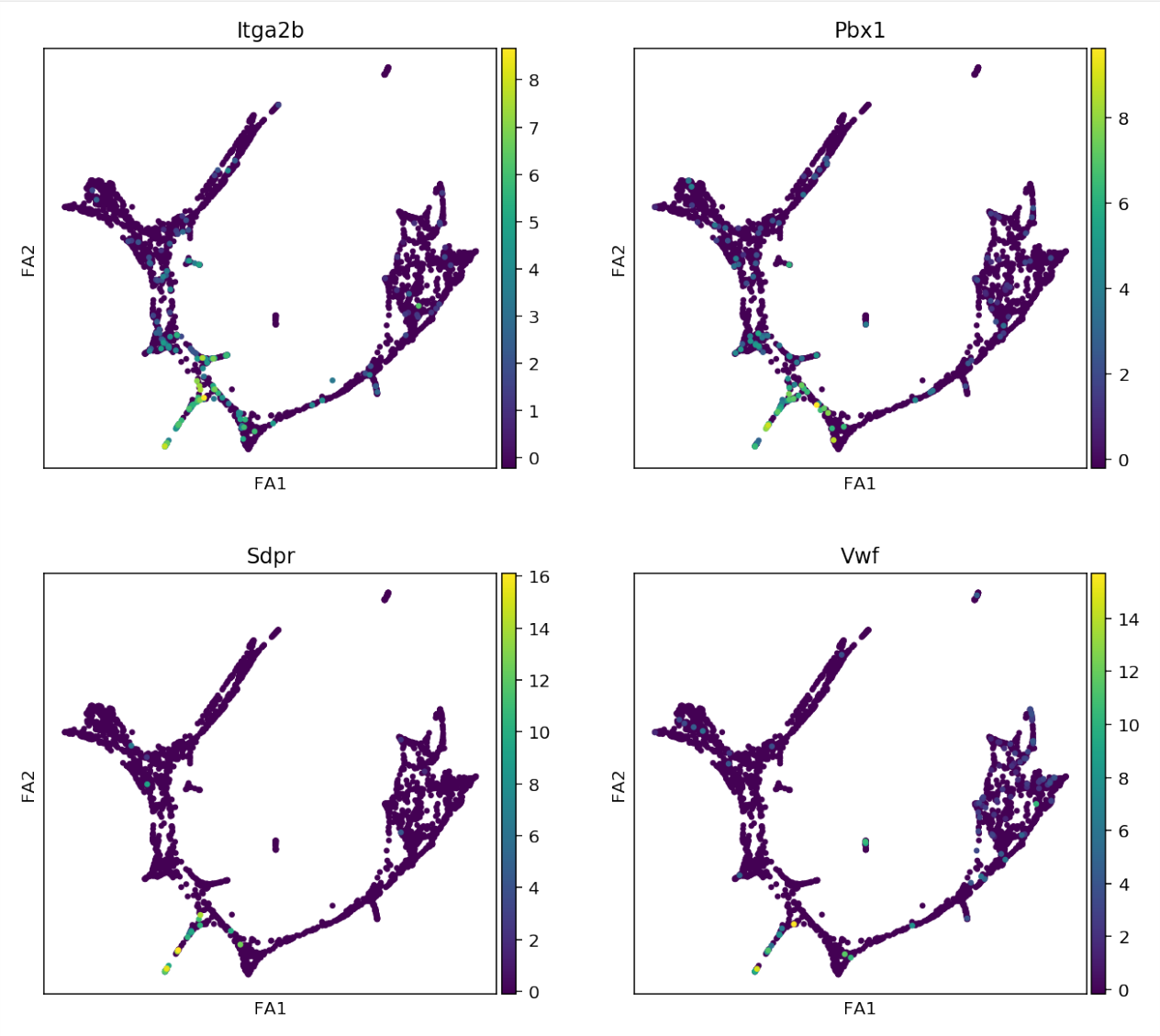


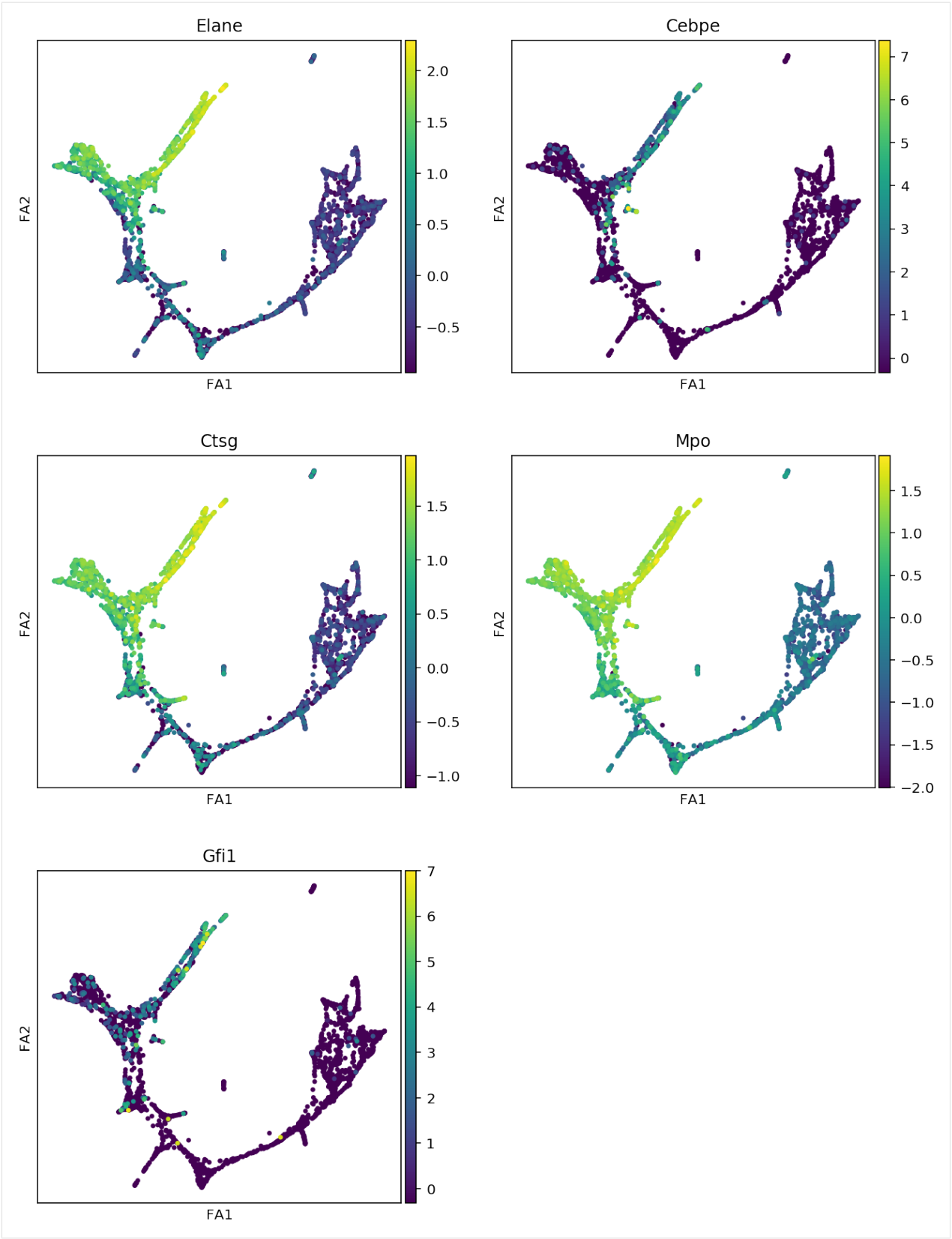
9. Check data

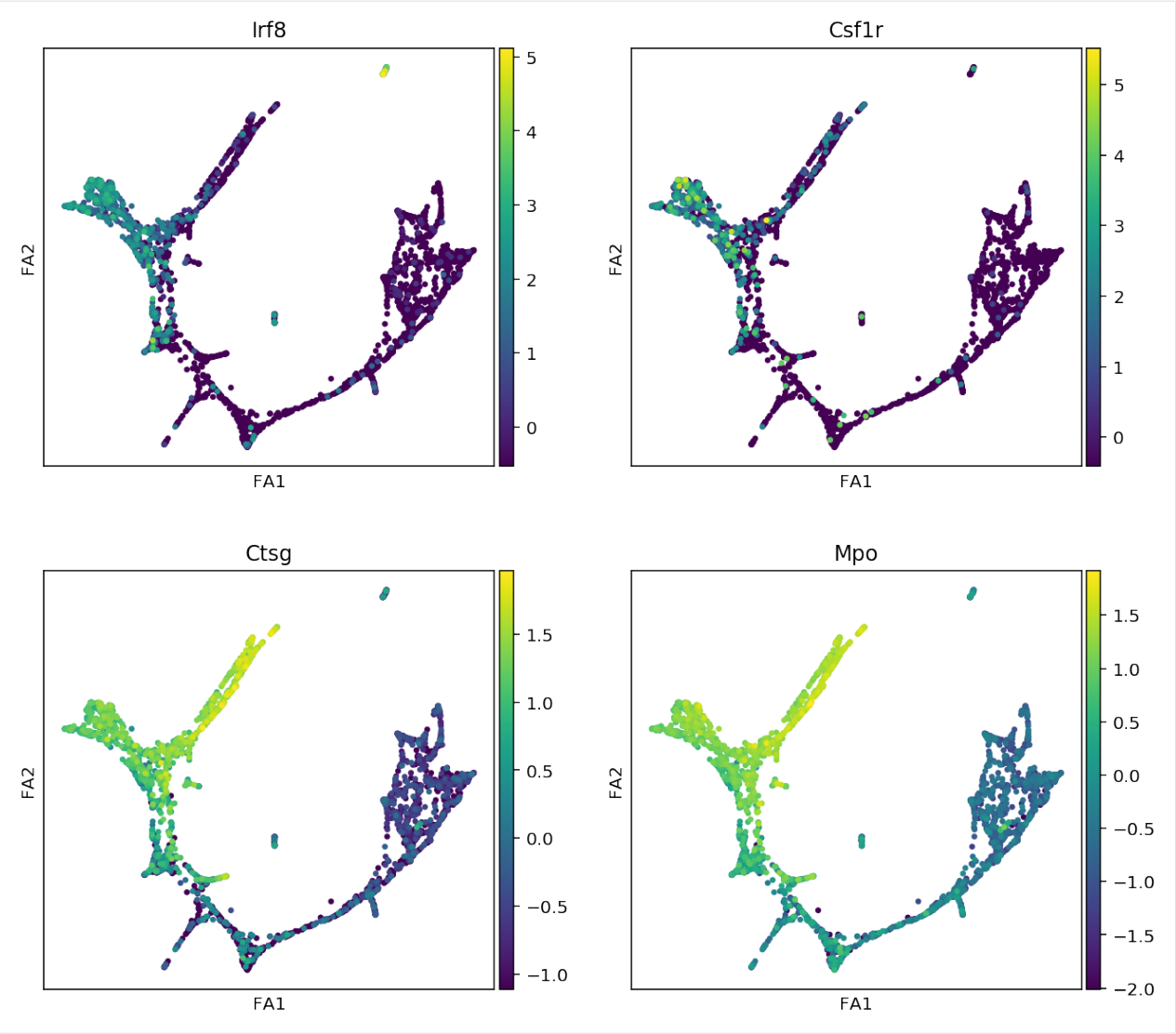
[18]:
```
plt.rcParams["figure.figsize"] = [4.5, 4.5]
```

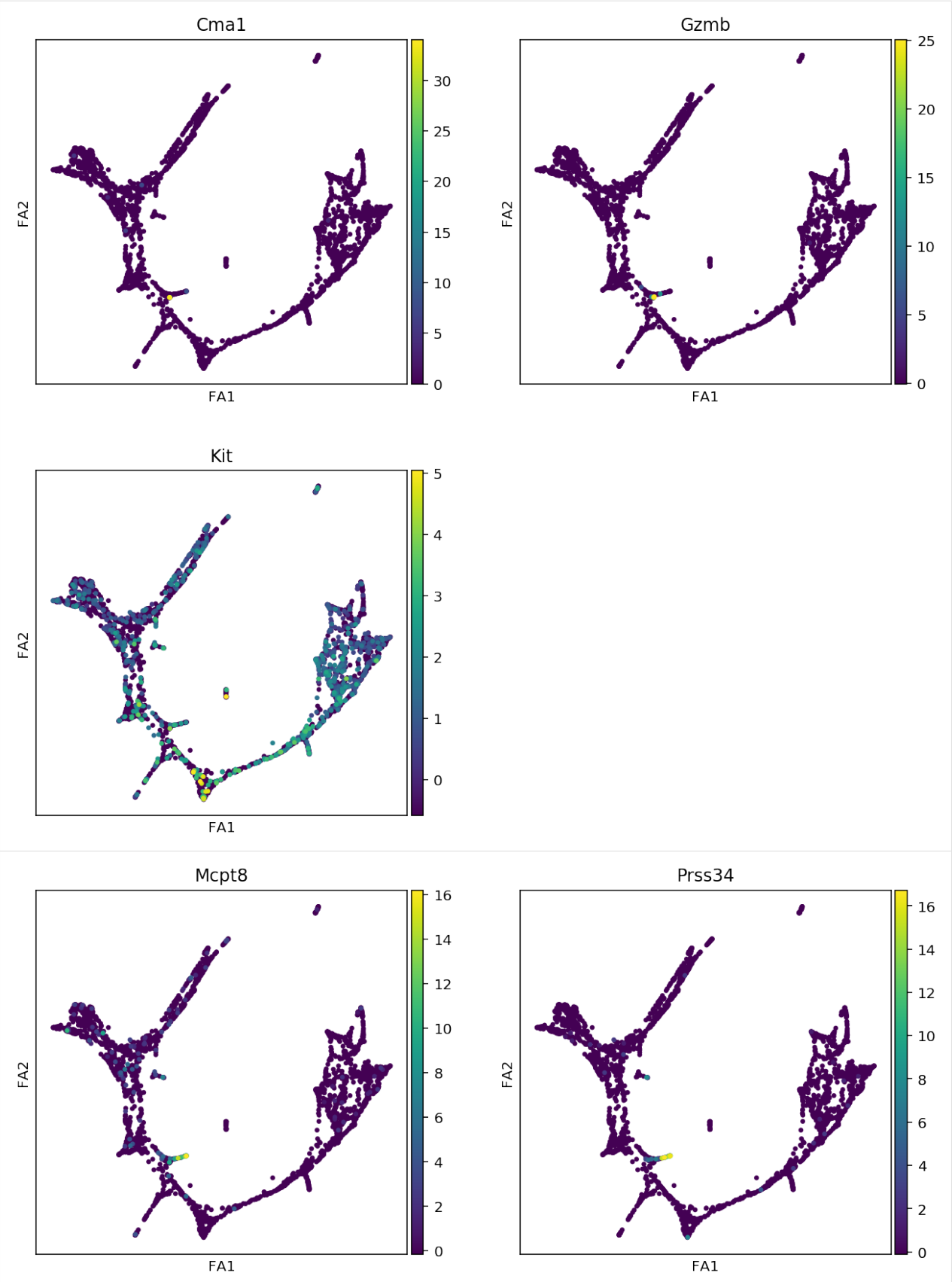
[19]:
```
markers = {"Erythroids":["Gata1", "Klf1", "Gypa", "Hba-a2"],
           "Megakaryocytes":["Itga2b", "Pbx1", "Sdpr", "Vwf"],
            "Granulocytes":["Elane", "Cebpe", "Ctsg", "Mpo", "Gfi1"],
            "Monocytes":["Irf8", "Csf1r", "Ctsg", "Mpo"],
            "Mast_cells":["Cma1", "Gzmb", "Kit"],
            "Basophils":["Mcpt8", "Prss34"]
            }

for cell_type, genes in markers.items():
    print(f"marker gene of {cell_type}")
    sc.pl.draw_graph(adata, color=genes, use_raw=False, ncols=2)
    plt.show()
```


Gata1
Klf1
Gypa
Hba-a2
FA1
FA2
0
1
2
3
4
5
6
7
8
−0.5
0.0
0.5
1.0
1.5
2.0
2.5


Itga2b
Pbx1
Sdpr
Vwf
FA1
FA2
0
1
2
3
4
5
6
7
8
10
12
14
16


Elane
FA2
FA1
2.0
1.5
1.0
0.5
0.0
−0.5
Cebpe
FA2
FA1
7
6
5
4
3
2
1
0
Ctsg
FA2
FA1
1.5
1.0
0.5
0.0
−0.5
−1.0
Mpo
FA2
FA1
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
Gfi1
FA2
FA1
7
6
5
4
3
2
1
0

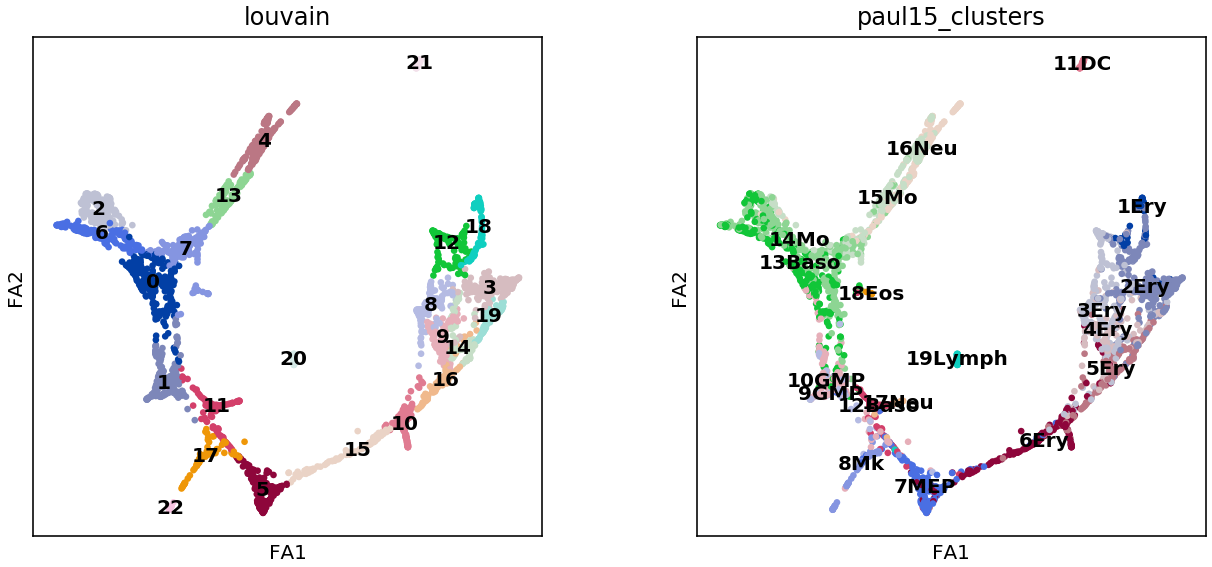

Irf8
Csf1r
Ctsg
Mpo
FA1
FA2
5
4
3
2
1
0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0


Cma1
FA2
FA1
30
25
20
15
10
5
0
Gzmb
FA2
FA1
25
20
15
10
5
0
Kit
FA2
FA1
5
4
3
2
1
0
Mcpt8
FA2
FA1
16
14
12
10
8
6
4
2
0
Prss34
FA2
FA1
16
14
12
10
8
6
4
2
0

10. [Optional step] Make annotation for cluster

Based on the marker gene expression and previous reports, we will manually annotate each cluster.

[20]:
```
sc.pl.draw_graph(adata, color=['louvain', 'paul15_clusters'],
                 legend_loc='on data')
```



[21]:
```
# Check current cluster name
cluster_list = adata.obs.louvain.unique()
cluster_list
```

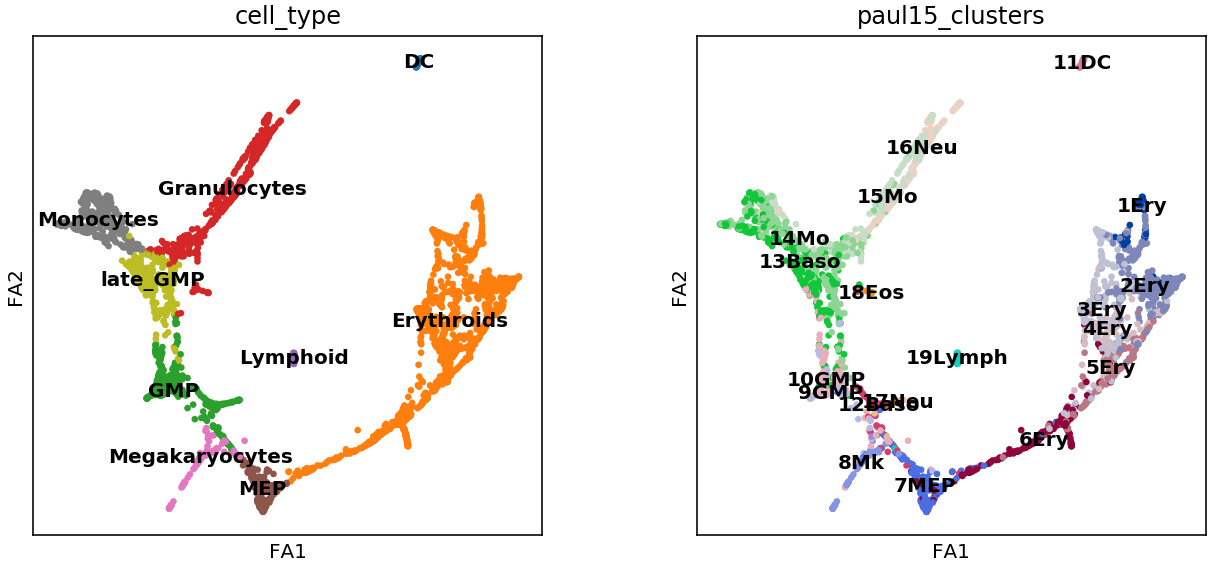
[21]:
```
[5, 2, 12, 13, 0, ..., 6, 20, 14, 15, 21]
Length: 23
Categories (23, object): [5, 2, 12, 13, ..., 20, 14, 15, 21]
```

[22]:
```
# Make anottation dictionary
annotation = {"MEP":[5],
              "Erythroids": [15, 10, 16, 9, 8, 14, 19, 3, 12, 18],
              "Megakaryocytes":[17, 22],
              "GMP":[11, 1],
              "late_GMP" :[0],
              "Granulocytes":[7, 13, 4],
              "Monocytes":[6, 2],
              "DC":[21],
              "Lymphoid":[20]}

# change dictionary format
annotation_rev = {}
for i in cluster_list:
    for k in annotation:
        if int(i) in annotation[k]:
            annotation_rev[i] = k

# check dictionary
annotation_rev
```

[22]:
```
{'5': 'MEP',
 '2': 'Monocytes',
 '12': 'Erythroids',
 '13': 'Granulocytes',
 '0': 'late_GMP',
 '10': 'Erythroids',
 '3': 'Erythroids',
 '18': 'Erythroids',
 '11': 'GMP',
 '7': 'Granulocytes',
 '8': 'Erythroids',
 '22': 'Megakaryocytes',
 '16': 'Erythroids',
 '1': 'GMP',
 '17': 'Megakaryocytes',
 '4': 'Granulocytes',
 '19': 'Erythroids',
 '9': 'Erythroids',
 '6': 'Monocytes',
 '20': 'Lymphoid',
 '14': 'Erythroids',
 '15': 'Erythroids',
 '21': 'DC'}
```

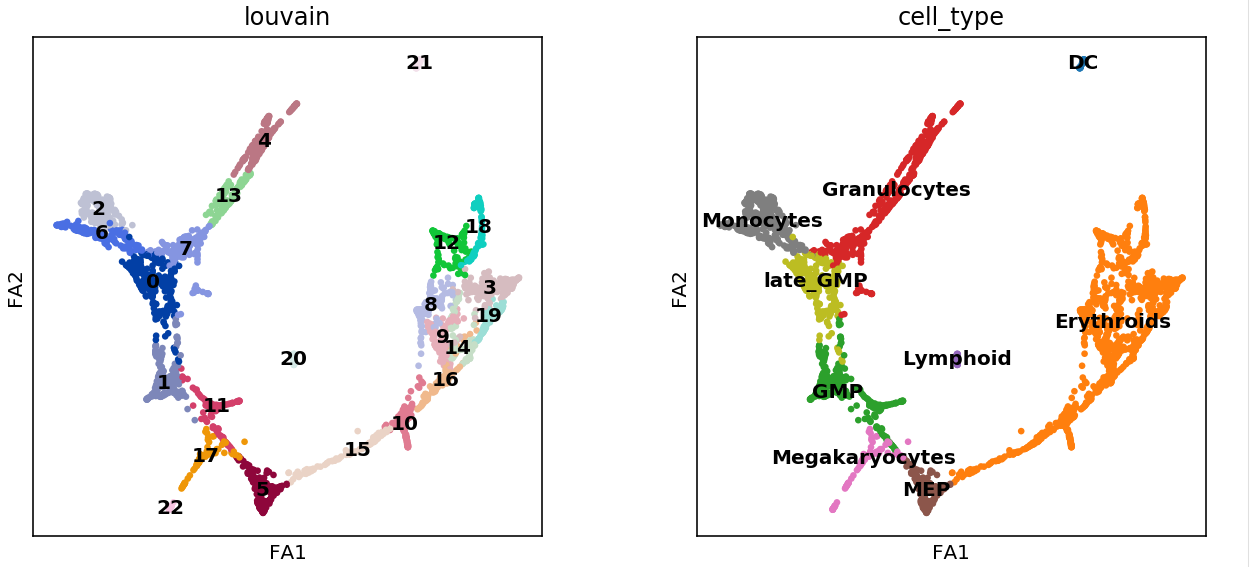
[23]:
```
adata.obs["cell_type"] = [annotation_rev[i] for i in adata.obs.louvain]
```

[24]:
```
# check results
sc.pl.draw_graph(adata, color=['cell_type', 'paul15_clusters'],
                 legend_loc='on data')
```



We'll make another annotation manually for each Louvain clusters.

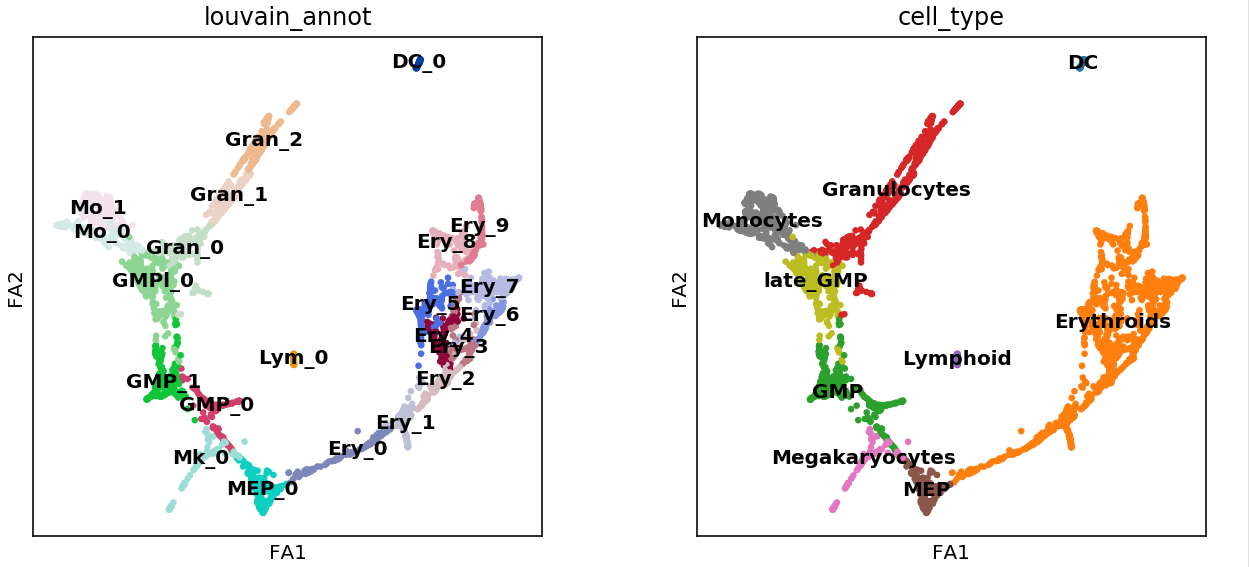
[25]:
```
sc.pl.draw_graph(adata, color=['louvain', 'cell_type'],
                 legend_loc='on data')
```



[26]:
```
annotation_2 = {'5': 'MEP_0',
                '15': 'Ery_0',
                '10': 'Ery_1',
                '16': 'Ery_2',
                '14': 'Ery_3',
                '9': 'Ery_4',
                '8': 'Ery_5',
                '19': 'Ery_6',
                '3': 'Ery_7',
                '12': 'Ery_8',
                '18': 'Ery_9',
                '17': 'Mk_0',
                '22': 'Mk_0',
                '11': 'GMP_0',
                '1': 'GMP_1',
                '0': 'GMPl_0',
                '7': 'Gran_0',
                '13': 'Gran_1',
                '4': 'Gran_2',
                '6': 'Mo_0',
                '2': 'Mo_1',
                '21': 'DC_0',
                '20': 'Lym_0'}
```

[27]:
```
adata.obs["louvain_annot"] = [annotation_2[i] for i in adata.obs.louvain]
```

[28]:
```
# Check result
sc.pl.draw_graph(adata, color=['louvain_annot', 'cell_type'],
                 legend_loc='on data')
```



11. [Optional step] Subset cells

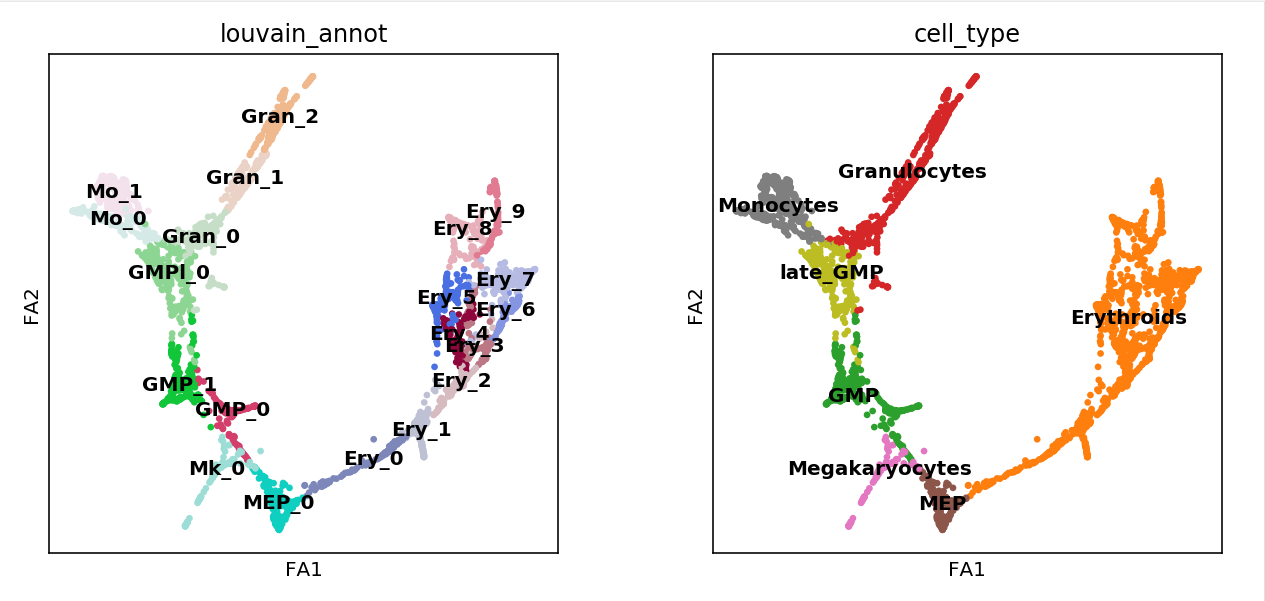
In this tutorial, we are using scRNA-seq data of hematopoiesis. In the latter part, we will focus on the cell fate decision in the myeloid lineage. So we will remove non-myeloid cell cluster; DC and Lymphoid cell cluster.

[29]:
```
adata.obs.cell_type.unique()
```

[29]:
```
[MEP, Monocytes, Erythroids, Granulocytes, late_GMP, GMP, Megakaryocytes, Lymphoid,␣
↪DC]
Categories (9, object): [MEP, Monocytes, Erythroids, Granulocytes, ..., GMP,␣
↪Megakaryocytes, Lymphoid, DC]
```

[30]:
```
cell_of_interest = adata.obs.index[~adata.obs.cell_type.isin(["Lymphoid", "DC"])]
adata = adata[cell_of_interest, :]
```

[31]:
```
# check result
sc.pl.draw_graph(adata, color=['louvain_annot', 'cell_type'],
                 legend_loc='on data')
```



12. Save processed data

```
[ ]: adata.write_h5ad("data/Paul_etal_15.h5ad")
```

B. scRNA-seq data preprocessing with Seurat

R notebook . . . comming in the future update.

Note: If you use `Seurat` for preprocessing, you need to convert the scRNA-seq data (Seurat object) into anndata to analyze the data with `celloracle`. `celloracle` has a python API and command-line API to convert a Seurat object into an anndata. Please go to the documentation of celloracle's API documentation for more information.

Pseudotime calculation

To interpret the celloracle simulation results, it is important to compare the simulated cell identity shift vector with the direction of natural development. We leverage pseudotime data to create development vector field.

Please download notebooks from here . Or please click below to view the content.

Overview

Aim

To interpret the celloracle simulation results, it is essential to compare the direction of the perturbation effect with natural differentiation. By comparing them, you can intuitively understand how TF is involved in cell fate determination during development. This perspective is also needed when estimating experimental perturbation results using celloracle simulations.

Method summary

For that purpose, we will introduce how to calculate the direction of differentiation using "pseudotime estimation" and "gradient calculation". Here's an overview of how to do this:

1. Calculate the pseudotime using the diffusion pseudotime method (dpt).
2. Transfer pseudotime data to grid points
3. Calculate the 2D gradient vector field using the pseudotime on the grid points
4. Compute the inner product value between the 2D gradient vector and the celloracle simulation vector to compare the simulated cell identity shift direction with the development direction.

In this notebook, we will do step1: pseudotime calculation. The pseudotime calculation part consists of these steps below. 1. Set lineage information and split the cells into several lineage brahches 2. Set root cells manually 3. Calculate pseudotime with dpt algorithm. 4. Re-aggregate scRNA-seq data into one data

Custom class / object

`Pseudotime_calculator`: This is a class for the pseudotime calculation. This class help us calculate pseudotime from scRNA-seq data. We need to specify a root cell. Also, scRNA-seq need to have a diffusion map >Under the hood, Pseudotime_calculator uses "dpt" algorithm. For more information of dpt algorithm and root cell, please look at the scanpy web documentation. https://scanpy.readthedocs.io/en/stable/api/scanpy.tl.dpt.html#scanpy.tl.dpt

Data

Pseudo-time calculation requires preprocessed scRNA-seq data in anndata format. You need to do neighbor calculation and diffusion map calculation in advance. If you have processed the scRNA-seq data according to our tutorial (link), these calculations have already been performed. - Neighbor calculation: https://scanpy.readthedocs.io/en/stable/generated/scanpy.pp.neighbors.html#scanpy.pp.neighbors - Diffusion map calculation: https://scanpy.readthedocs.io/en/stable/generated/scanpy.tl.diffmap.html#scanpy.tl.diffmap

Install additional python package

This notebook we recommend using another python package, `plotly`.

Please install `plotly` in advance.

`pip install plotly`

> Plotly is a toolkit for interactive visualization. We recommend using plotly to pick up root cells in this notebook. For more information, please look at plotly web site. https://plotly.com

Caution

Here, we will introduce an example of a pseudotime calculation method using the diffusion pseudotime method. This is NOT celloracle analysis itself. If you want to use another different algorithm for the pseudotime calculation, you can use anything.

0. Import libraries

0.1. Import public libraries

[3]:
```
import copy
import glob
import time
import os
import shutil
import sys

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scanpy as sc
import seaborn as sns
from tqdm.notebook import tqdm

#import time
```

0.2. Import our library

[2]:
```
import celloracle as co
from celloracle.applications import Pseudotime_calculator
co.__version__
```

[2]:
```
'0.7.0'
```

0.3. Plotting parameter setting

[20]:
```
#plt.rcParams["font.family"] = "arial"
plt.rcParams["figure.figsize"] = [5,5]
%config InlineBackend.figure_format = 'retina'
plt.rcParams["savefig.dpi"] = 300

%matplotlib inline
```

1. Load data

We can add pseudotime calculation to an oracle object or to anndata.

- If you have oracle object, please run **1.1.[Option1] Load oracle data.**
- If you have not made oracle object yet and want to calculate pseudotime using anndata, please run **1.2.[Option2] Load anndata.**

In this notebook, we load demo oracle object to add pseudotime.

1.1. [Option1] Load oracle data

[13]:
```
# Load demo scRNA-seq data.
oracle = co.data.load_tutorial_oracle_object()

# Instantiate pseudotime object using oracle object.
pt = Pseudotime_calculator(oracle_object=oracle)
```

1.2. [Option2] Load anndata

[16]:
```
# Load demo scRNA-seq data.
adata = co.data.load_Paul2015_data()

# Instantiate pseudotime object using anndata object.
pt = Pseudotime_calculator(adata=adata,
                           obsm_key="X_draw_graph_fa", # Dimensional reduction data␣
↪name
                           cluster_column_name="louvain_annot" # Clustering data name
                           )
```

2. Pseudotime calculation

2.1. Add lineage information

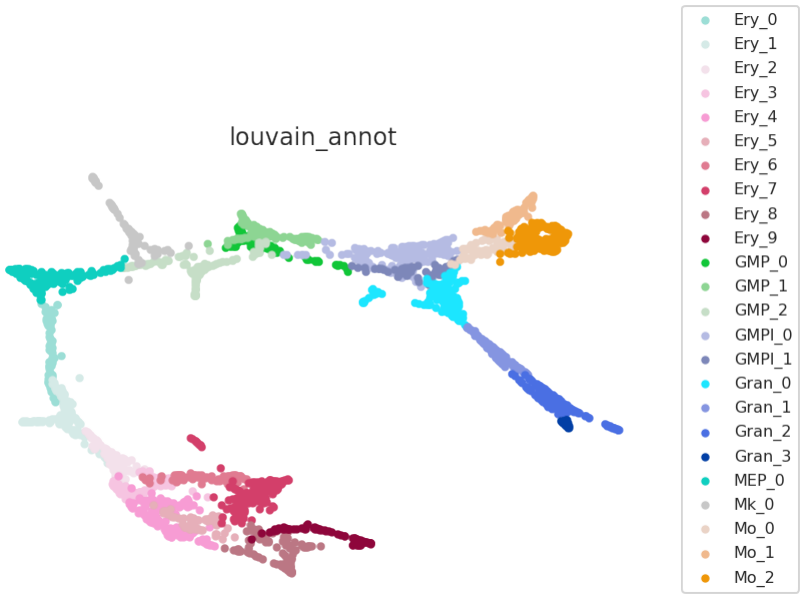
We will calculate pseudotime for each lineage. We need to set lineage information first.

2.1.1 Check clustering unit

[17]:
```
print("Clustering name: ", pt.cluster_column_name)
print("Cluster list", pt.cluster_list)
```

```
Clustering name:  louvain_annot
Cluster list ['Ery_0', 'Ery_1', 'Ery_2', 'Ery_3', 'Ery_4', 'Ery_5', 'Ery_6', 'Ery_7',
↪'Ery_8', 'Ery_9', 'GMP_0', 'GMP_1', 'GMP_2', 'GMPl_0', 'GMPl_1', 'Gran_0', 'Gran_1',
↪ 'Gran_2', 'Gran_3', 'MEP_0', 'Mk_0', 'Mo_0', 'Mo_1', 'Mo_2']
```

[23]:
```
# Check data
pt.plot_cluster(fontsize=8)
```



2.1.2. Define lineage

We will make lineage annotation on the scRNA-seq data. For example, this scRNA-seq data include roughly two lineages: megakaryocytes-erythroid (ME) lineage and granulocytes-monocyte (GM) lineage.

To get better pseudotime information, calculate the pseudotime for each cell lineage individually. Then, all pseudotime information of each lineage are merged into one.

Here is an example of setting lineage information. Lineage structure and number may vary depending on the data. Please adjust them on demand.

[35]:
```
# These cluster can be classified into either MEP lineage or GMP lineage

clusters_in_ME_lineage = ['Ery_0', 'Ery_1', 'Ery_2', 'Ery_3', 'Ery_4', 'Ery_5',
                          'Ery_6', 'Ery_7', 'Ery_8', 'Ery_9', 'MEP_0', 'Mk_0']
clusters_in_GM_lineage = ['GMP_0', 'GMP_1', 'GMP_2', 'GMPl_0', 'GMPl_1', 'Gran_0',
                          'Gran_1', 'Gran_2', 'Gran_3', 'Mo_0', 'Mo_1', 'Mo_2']

# Make dictionary
lineage_dictionary = {"Lineage_ME": clusters_in_ME_lineage,
           "Lineage_GM": clusters_in_GM_lineage}
```

(continues on next page)

(continued from previous page)

```
# Inpur lineage information into pseudotime object
pt.set_lineage(lineage_dictionary=lineage_dictionary)

# Visualize lineage information
pt.plot_lineages()
```

2.2. Add root cell information

The pseudotime calculation with dpt requires to input root cell. We will manyally estimate root cell for each lineage.

Please read documentation (https://scanpy.readthedocs.io/en/stable/api/scanpy.tl.dpt.html#scanpy.tl.dpt) to find detailed information about dpt algorithm and root cells

2.2.1. (optional) Interactive visualization of cell name

This notebook we recommend using another python package, `plotly`.

Please install `plotly` in advance.

`pip install plotly`

> Plotly is a toolkit for interactive visualization. We recommend using plotly to pick up root cells in this notebook. For more information, please look at plotly web site. https://plotly.com

Using plotly, we can visualize cell name interactively. It helps us pick up a root cell. This is an example image. **|6fd38db5fcf34a6fa24413839b8e2112|**

https://raw.githubusercontent.com/morris-lab/CellOracle/master/docs/demo_data/screenshot_01.png

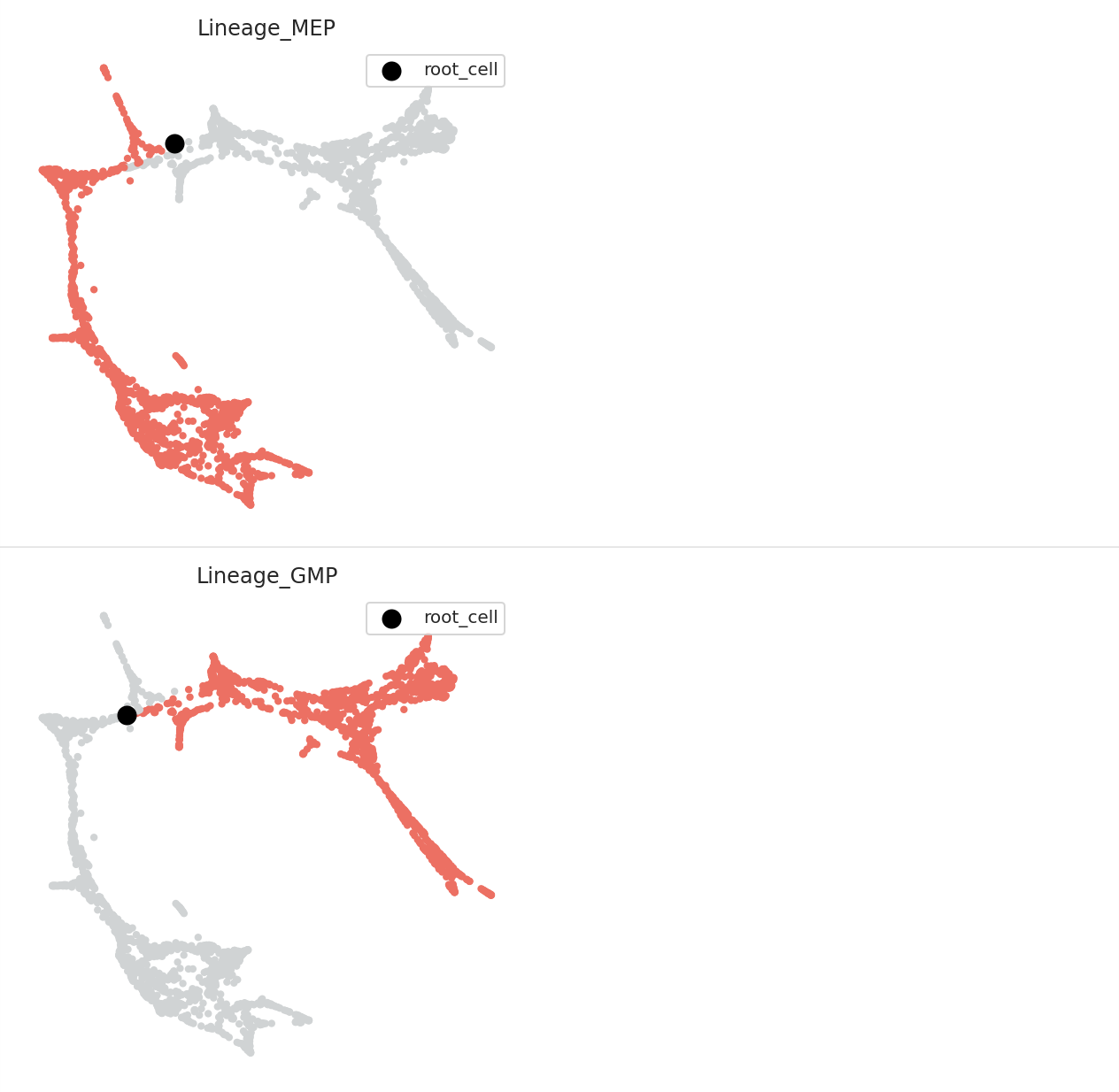
[ ]:
```
# Show interactive plot using plotly. Please make sure you installed plotly.

try:
    import plotly.express as px
    def plot(adata, embedding_key, cluster_column_name):
        embedding = adata.obsm[embedding_key]
        df = pd.DataFrame(embedding, columns=["x", "y"])
        df["cluster"] = adata.obs[cluster_column_name].values
        df["label"] = adata.obs.index.values
        fig = px.scatter(df, x="x", y="y", hover_name=df["label"], color="cluster")
        fig.show()

    plot(adata=pt.adata,
         embedding_key=pt.obsm_key,
         cluster_column_name=pt.cluster_column_name)
except:
    print("Found error. Did you install plotly? Please read the instruction above.")
```

2.2.2. Select root cell for each lineage

[38]:
```
#  Estimated root cell name for each lineage
root_cells = {"Lineage_MEP": "1539", "Lineage_GMP": "2244"}
pt.set_root_cells(root_cells=root_cells)
```

2.2.3. Visualize root cells

[11]:
```
# Check root cell and lineage
pt.plot_root_cells()
```



2.3. Pseudotime calculation

You need to do neighbor calculation and diffusion map calculation in advance. If you have processed the scRNA-seq data according to our tutorial, these calculations have already been performed. - Neighbor calculation: https://scanpy.readthedocs.io/en/stable/generated/scanpy.pp.neighbors.html#scanpy.pp.neighbors - Diffusion map calculation: https://scanpy.readthedocs.io/en/stable/generated/scanpy.tl.diffmap.html#scanpy.tl.diffmap

2.3.1. Check diffusion map

[36]:
```
# Check diffusion map data.
"X_diffmap" in pt.adata.obsm
```

[36]:
```
True
```

Calculate diffusion map if your adata does not have diffusion map data

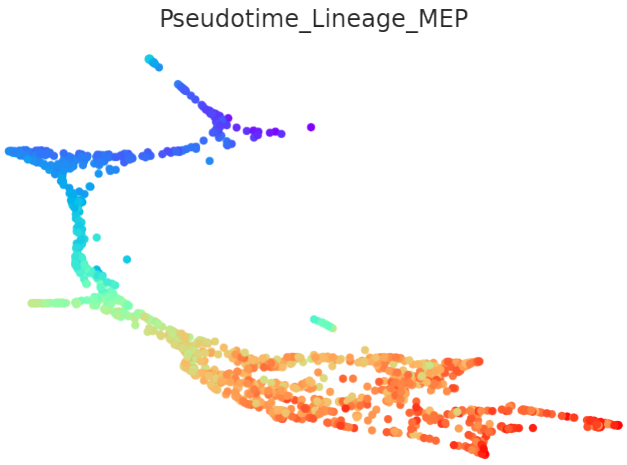
[13]:
```
# sc.pp.neighbors(pt.adata, n_neighbors=30)
# sc.tl.diffmap(pt.adata)
```

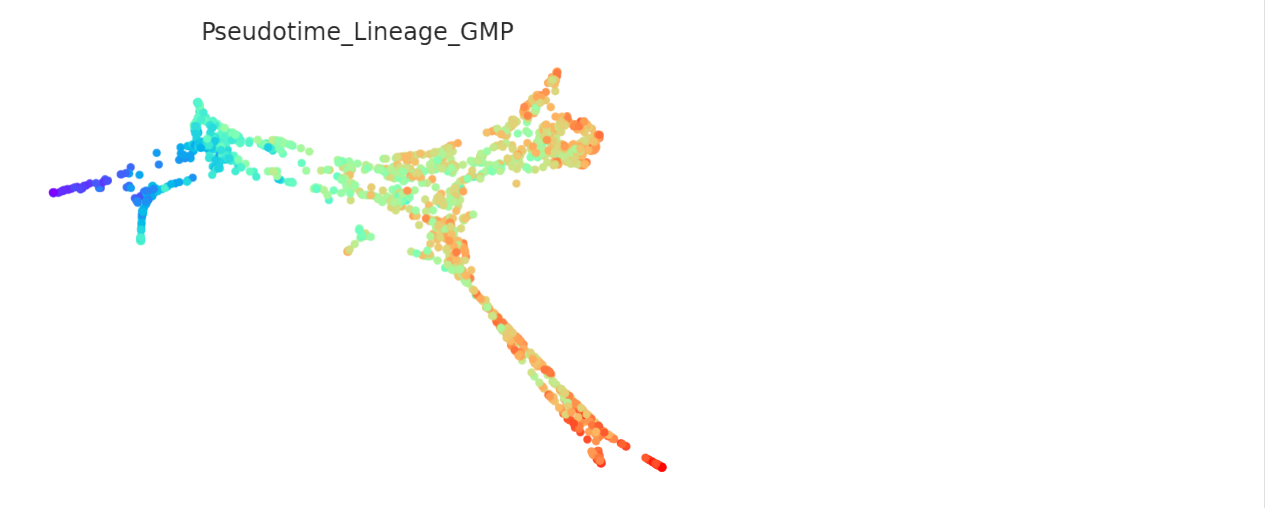
Diffusion maps can be calculated in the anther dimensionality reduction space. Please adjust this parameter "use_rep" to get another better results if you have a issue in the following calculation.

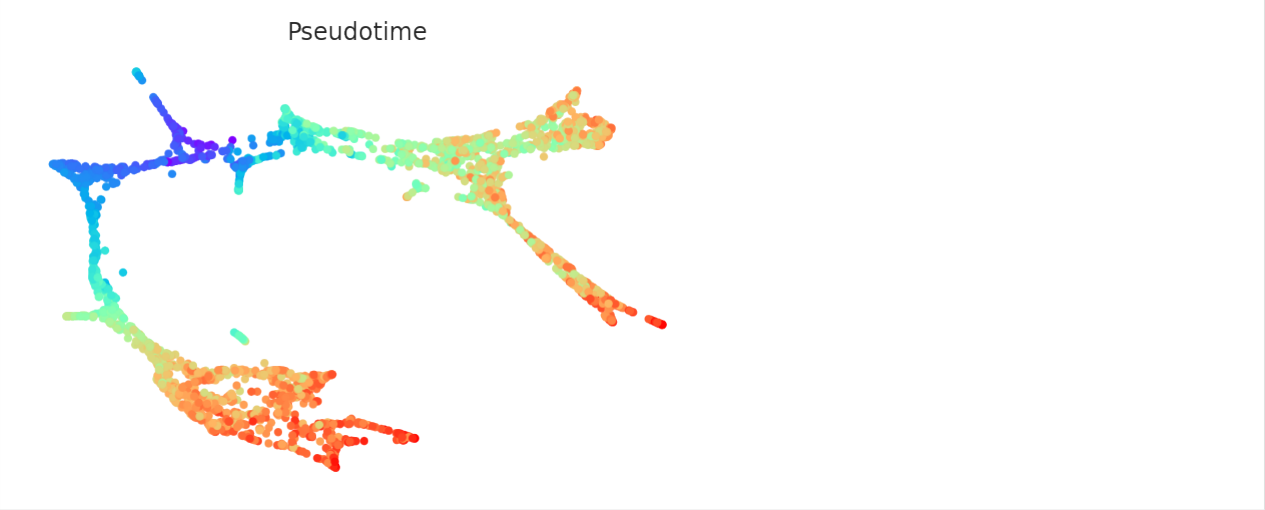
[14]:
```
# sc.pp.neighbors(pt.adata, n_neighbors=30, use_rep=)
# sc.tl.diffmap(pt.adata)
```

2.3.2. Calculate pseudotime

[39]:
```
# Calculate pseudotime
pt.get_pseudotime_per_each_lineage()

# Check results
pt.plot_pseudotime(cmap="rainbow")
```







Pseudotime data is stored in the `pt.adata.obs.Pseudotime`

[41]:
```
# Check result
pt.adata.obs[["Pseudotime"]].head()
```

[41]:
```
       Pseudotime
index
0        0.175565
1        0.712654
2        0.953920
3        0.642302
4        0.951093
```

3. Save data

3.1. If you started calculation with an oracle object

[17]:
```
# Add calculated pseudotime data to the oracle object
oracle.adata.obs = pt.adata.obs

# Save updated oracle object
oracle.to_hdf5(FILE_PATH)
```

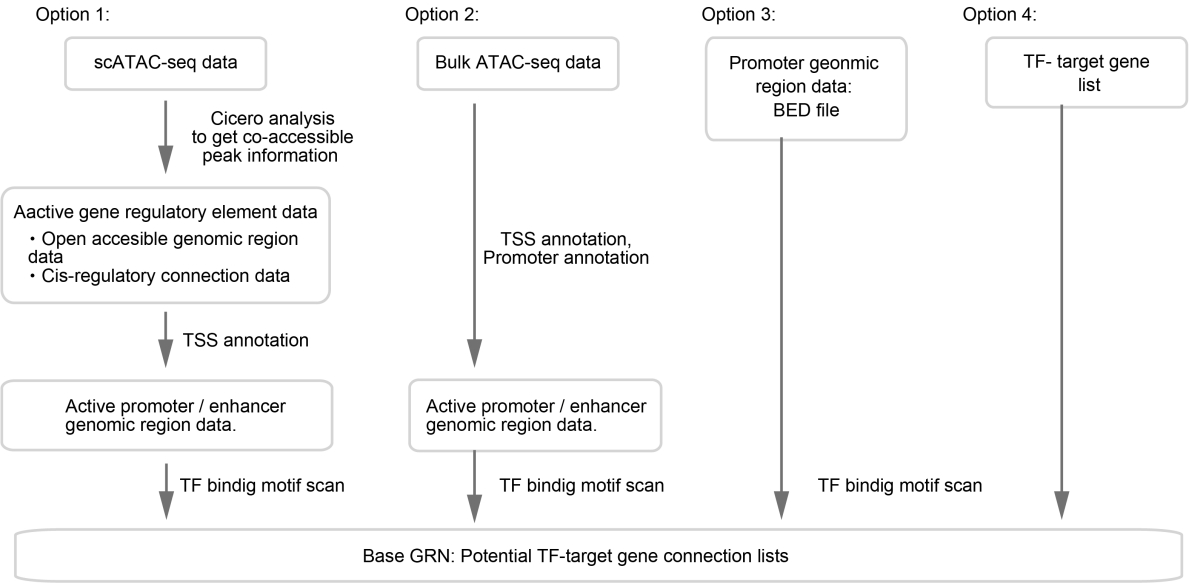
3.2. If you started calculation with anndata

[ ]:
```
# Add calculated pseudotime data to the oracle object
#adata.obs = pt.adata.obs

# Save updated anndata object
#adata.write_h5ad(FILE_PATH)
```

Base GRN input data preparation

Overview

There are several options for CellOracle base-GRN construction. Here is the illustration of base-GRN construction workflow.



- In this documentation, we introduce details of option1 and option2.
- Option3 uses promoter database for the input of base-GRN construction. We provide pre-built promoter base-GRN for 10 species. You can load this base GRN using celloracle data loading function.
- In option4, any TF-target gene list can be used as a base-GRN. Here is an example notebook [link].

Option1. Data preprocessing of scATAC-seq data

If you have scATAC-seq data, you can use scATAC-seq data to obtain the accessible promoter/enhancer DNA sequence. To prepare input data of base-GRN construction, we need to get the accessible promoter/enhancer DNA sequence from scATAC-seq data.

Here, we introduce an example method to extract active promoter / enhancer peaks from scATAC-seq data using Cicero.

Note: Cicero is a R package for scATAC-seq data analysis. It can pick up distal cis-regulatory elements in scATAC-seq data.

Warning:

- Here, we intend to introduce an example of how to prepare input data. **This is not CellOracle analysis. We do NOT use celloracle in this step**.
- This is just an example of data preparation step, you can analyze your data with Cicero in a different way if you are familiar with Cicero. If you have a question about Cicero, please read the documentation of Cicero for the detailed usage.
- If you have a favorite algorithm / software for scATAC-data analysis, you can use totally different software to pick up gene expression regulatory elements.

Step1. scATAC-seq analysis with Cicero

The jupyter notebook file is available here . The R notebook file is available here .

Or click below to see the contents.

Overview

This notebook is an example R script on how to prepare the input data for building a CellOracle-based GRN. We aim to extract cis-regulated connections between scATAC-seq peaks. Here, we will introduce the data preparation method using Cicero.

Notebook file

Notebook file is available at CellOracle GitHub. We have jupyter notebook (with R kernel) and R notebook. The contents are same. Please download and run either one.

- Jupyter notebook: https://github.com/morris-lab/CellOracle/blob/master/docs/notebooks/01_ATAC-seq_data_processing/option1_scATAC-seq_data_analysis_with_cicero/01_atacdata_analysis_with_cicero_and_monocle3.ipynb
- R notebook: https://github.com/morris-lab/CellOracle/blob/master/docs/notebooks/01_ATAC-seq_data_processing/option1_scATAC-seq_data_analysis_with_cicero/01_atacdata_analysis_with_cicero_and_monocle3.Rmd

CAUTION:

- This notebook is intended to explain **how to prepare the input data for CellOracle analysis**. This is **NOT** the CellOracle analysis itself. Also, this notebook does NOT use `celloracle` in this notebook.
- Here, we use `Cicero` to process scATAC-seq data. If you are new to this packages, pelase read the documentation to learn them in advance.
- `Cicero` documentation: https://cole-trapnell-lab.github.io/cicero-release/docs_m3/

0. Import library

[2]:
```
library(cicero)
library(monocle3)
```

1. Download data

This tutorial uses fetal brain acATAC-seq data obtained from a 10x genomics database. If you want to analyze your scATAC-seq data, you do not need to download these data.

You can download the demo file by running the following command: If the file download fails, please manually download and unzip the data. http://cf.10xgenomics.com/samples/cell-atac/1.1.0/atac_v1_E18_brain_fresh_5k/atac_v1_E18_brain_fresh_5k_filtered_peak_bc_matrix.tar.gz

[3]:
```
# Create folder to store data
dir.create("data")

# Download demo dataset from 10x genomics
download.file(url = "http://cf.10xgenomics.com/samples/cell-atac/1.1.0/atac_v1_E18_
↪brain_fresh_5k/atac_v1_E18_brain_fresh_5k_filtered_peak_bc_matrix.tar.gz",
              destfile = "data/matrix.tar.gz")
# Unzip data
system("tar -xvf data/matrix.tar.gz -C data")
```

[4]:
```
# You can substitute the data path below with the data path of your scATAC data.
data_folder <- "data/filtered_peak_bc_matrix"

# Create a folder to save results
output_folder <- "cicero_output"
dir.create(output_folder)
```

2. Load data and make Cell Data Set (CDS) object

[5]:
```
# Read in matrix data using the Matrix package
indata <- Matrix::readMM(paste0(data_folder, "/matrix.mtx"))
# Binarize the matrix
indata@x[indata@x > 0] <- 1

# Format cell info
cellinfo <- read.table(paste0(data_folder, "/barcodes.tsv"))
row.names(cellinfo) <- cellinfo$V1
names(cellinfo) <- "cells"
```

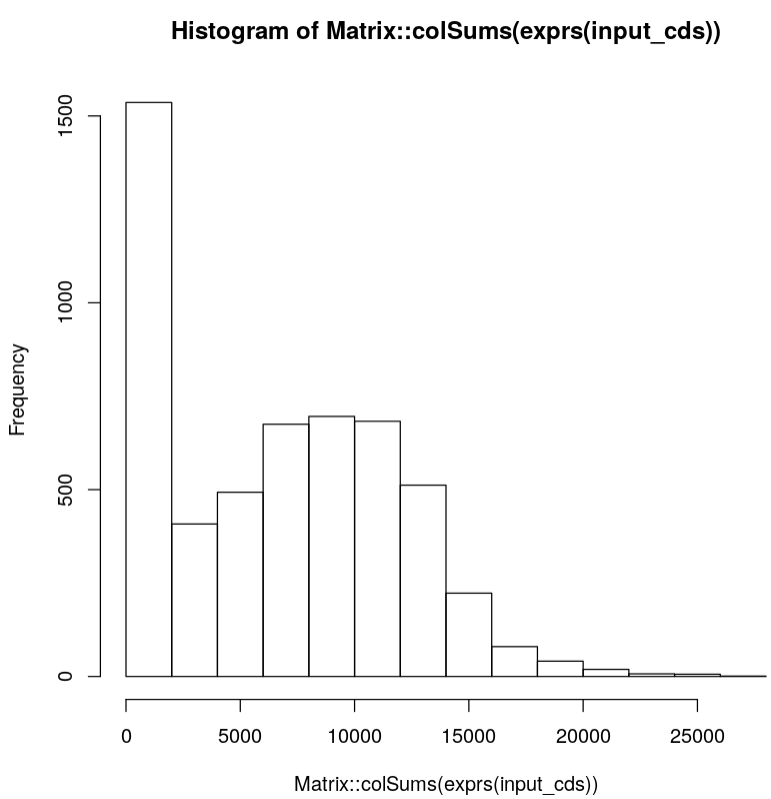
(continues on next page)

(continued from previous page)

```
# Format peak info
peakinfo <- read.table(paste0(data_folder, "/peaks.bed"))
names(peakinfo) <- c("chr", "bp1", "bp2")
peakinfo$site_name <- paste(peakinfo$chr, peakinfo$bp1, peakinfo$bp2, sep="_")
row.names(peakinfo) <- peakinfo$site_name

row.names(indata) <- row.names(peakinfo)
colnames(indata) <- row.names(cellinfo)

# Make CDS
input_cds <-  suppressWarnings(new_cell_data_set(indata,
cell_metadata = cellinfo,
gene_metadata = peakinfo))

input_cds <- monocle3::detect_genes(input_cds)

#Ensure there are no peaks included with zero reads
input_cds <- input_cds[Matrix::rowSums(exprs(input_cds)) != 0,]
```

3. Qauality check and Filtering

[6]:
```
# Visualize peak_count_per_cell
hist(Matrix::colSums(exprs(input_cds)))
```



[7]:
```
# Filter cells by peak_count
# Please set an appropriate threshold values according to your data
max_count <-  15000
min_count <- 2000
input_cds <- input_cds[,Matrix::colSums(exprs(input_cds)) >= min_count]
input_cds <- input_cds[,Matrix::colSums(exprs(input_cds)) <= max_count]
```

4. Process cicero-CDS object

[8]:
```
# Data preprocessing
set.seed(2017)

input_cds <- detect_genes(input_cds)
input_cds <- estimate_size_factors(input_cds)
input_cds <- preprocess_cds(input_cds, method = "LSI")

# Dimensional reduction with umap
input_cds <- reduce_dimension(input_cds, reduction_method = 'UMAP',
                              preprocess_method = "LSI")
umap_coords <- reducedDims(input_cds)$UMAP


cicero_cds <- make_cicero_cds(input_cds, reduced_coordinates = umap_coords)

# Save cds object if you want
#saveRDS(cicero_cds, paste0(output_folder, "/cicero_cds.Rds"))
```

```
Overlap QC metrics:
Cells per bin: 50
Maximum shared cells bin-bin: 44
Mean shared cells bin-bin: 0.84960828849071
Median shared cells bin-bin: 0
```

5. Load reference genome information

To run cicero, you need to get a genomic coordinate file that contains the length of each chromosome. You can download the mm10 genomic information with the following command.

If your scATAC-seq data was generated with another reference genome, you need to get the genome coordinate file for the reference genome you used. See the Cicero documentation for more information.

https://cole-trapnell-lab.github.io/cicero-release/docs_m3/#installing-cicero

[9]:
```
# !!Please make sure that the reference genome information below match the reference␣
↪genome of your scATAC-seq data.

# If your scATAC-seq uses mm10 reference genome, you can read chromosome length file␣
↪with the following command.
download.file(url = "https://raw.githubusercontent.com/morris-lab/CellOracle/master/
↪docs/demo_data/mm10_chromosome_length.txt",
```

(continues on next page)

(continued from previous page)

```
              destfile = "./mm10_chromosome_length.txt")
chromosome_length <- read.table("./mm10_chromosome_length.txt")

# For mm9 genome, you can use the following command.
#data("mouse.mm9.genome")
#chromosome_length <- mouse.mm9.genome

# For hg19 genome, you can use the following command.
#data("human.hg19.genome")
#chromosome_length <- mhuman.hg19.genome
```

6. Run Cicero

[10]:
```
# Run the main function
conns <- run_cicero(cicero_cds, chromosome_length) # Takes a few minutes to run

# Save results if you want
#saveRDS(conns, paste0(output_folder, "/cicero_connections.Rds"))

# Check results
head(conns)
```

A data.frame: 6 × 3

	Peak1 <chr>	Peak2 <fct>	coaccess <dbl>
1	chr10_100006139_100006389	chr10_99774288_99774570	-0.003546179
2	chr10_100006139_100006389	chr10_99825945_99826237	-0.027536333
3	chr10_100006139_100006389	chr10_99830012_99830311	0.009588013
4	chr10_100006139_100006389	chr10_99833211_99833540	-0.008067111
5	chr10_100006139_100006389	chr10_99941805_99941955	0.000000000
7	chr10_100006139_100006389	chr10_100015291_100017830	-0.015018099

7. Save results for the next step

[26]:
```
all_peaks <- row.names(exprs(input_cds))
write.csv(x = all_peaks, file = paste0(output_folder, "/all_peaks.csv"))
write.csv(x = conns, file = paste0(output_folder, "/cicero_connections.csv"))
```

Please go to next step: TSS annotation

https://morris-lab.github.io/CellOracle.documentation/tutorials/base_grn.html#step2-tss-annotation

[ ]:

Step2. TSS annotation

We can get active promoter / enhancer peaks in step1 above. Next, we will make gene annotations for these peaks.

The jupyter notebook file is available here .

Or click below to see the contents.

Overview

In this notebook, we will make TSS annotation in the Cicero coaccessible peak data to get input data of base-GRN construction. - First, we pick up peaks around the transcription starting site (TSS). - Second, we merge cicero data with the peaks around TSS. - Then we remove peaks that have a weak connection to TSS peak so that the final product includes TSS peaks and peaks that have a strong connection with the TSS peaks. We use this information as an active promoter/enhancer elements.

Notebook file

Notebook file is available at CellOracle GitHub. https://github.com/morris-lab/CellOracle/blob/master/docs/notebooks/01_ATAC-seq_data_processing/option1_scATAC-seq_data_analysis_with_cicero/02_preprocess_peak_data.ipynb

0. Import libraries

[2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns


import os, sys, shutil, importlib, glob
from tqdm.notebook import tqdm

from celloracle import motif_analysis as ma
```

[3]:
```
%config InlineBackend.figure_format = 'retina'

plt.rcParams['figure.figsize'] = [6, 4.5]
plt.rcParams["savefig.dpi"] = 300
```

1. Load data made with cicero

In this notebook, we explain how to process Cicero output. Please look at the previous step to know how to get this data yourself. https://morris-lab.github.io/CellOracle.documentation/tutorials/base_grn.html#step1-scatac-seq-analysis-with-cicero

Here, we use preprosessed Cicero data that were made from scATAC-seq data.

You can download the demo file by running the following command: If the file download fails, please manually download and unzip the data.

https://raw.githubusercontent.com/morris-lab/CellOracle/master/docs/demo_data/all_peaks.csv

https://raw.githubusercontent.com/morris-lab/CellOracle/master/docs/demo_data/cicero_connections.csv

[4]:
```
# Download file.
!wget https://raw.githubusercontent.com/morris-lab/CellOracle/master/docs/demo_data/
↪all_peaks.csv
!wget https://raw.githubusercontent.com/morris-lab/CellOracle/master/docs/demo_data/
↪cicero_connections.csv

# If you are using macOS, please try the following command.
#!curl -O https://raw.githubusercontent.com/morris-lab/CellOracle/master/docs/demo_
↪data/all_peaks.csv
#!curl -O https://raw.githubusercontent.com/morris-lab/CellOracle/master/docs/demo_
↪data/cicero_connections.csv
```

```
--2021-07-07 21:42:05--  https://raw.githubusercontent.com/morris-lab/CellOracle/
↪master/docs/demo_data/all_peaks.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.110.133,␣
↪185.199.109.133, 185.199.108.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.110.133|:
↪443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2940392 (2.8M) [text/plain]
Saving to: 'all_peaks.csv'

all_peaks.csv       100%[===================>]   2.80M  --.-KB/s    in 0.05s

2021-07-07 21:42:06 (56.3 MB/s) - 'all_peaks.csv' saved [2940392/2940392]

--2021-07-07 21:42:06--  https://raw.githubusercontent.com/morris-lab/CellOracle/
↪master/docs/demo_data/cicero_connections.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.110.133,␣
↪185.199.109.133, 185.199.108.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.110.133|:
↪443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 22749615 (22M) [text/plain]
Saving to: 'cicero_connections.csv'

cicero_connections. 100%[===================>]  21.70M  78.7MB/s    in 0.3s

2021-07-07 21:42:06 (78.7 MB/s) - 'cicero_connections.csv' saved [22749615/22749615]
```

[5]:
```
# Load scATAC-seq peak list.
peaks = pd.read_csv("all_peaks.csv", index_col=0)
```

(continues on next page)

(continued from previous page)

```
peaks = peaks.x.values
peaks
```

[5]:
```
array(['chr10_100006139_100006389', 'chr10_100015291_100017830',
       'chr10_100018677_100020384', ..., 'chrY_90804622_90805450',
       'chrY_90808626_90809117', 'chrY_90810560_90811167'], dtype=object)
```

[6]:
```
# Load cicero coaccess score.
cicero_connections = pd.read_csv("cicero_connections.csv", index_col=0)
cicero_connections.head()
```

[6]:

	Peak1	Peak2	coaccess
1	chr10_100006139_100006389	chr10_99774288_99774570	-0.003546
2	chr10_100006139_100006389	chr10_99825945_99826237	-0.027536
3	chr10_100006139_100006389	chr10_99830012_99830311	0.009588
4	chr10_100006139_100006389	chr10_99833211_99833540	-0.008067
5	chr10_100006139_100006389	chr10_99941805_99941955	0.000000

2. Make TSS annotation

If your scATAC-seq data was generated with mm10 reference genome, please set `ref_genome="mm10"`.

You can check supported reference genome using `ma.SUPPORTED_REF_GENOME`

If your reference genome is not in the list, please send a request through github issue page.

[7]:
```
ma.SUPPORTED_REF_GENOME
```

[7]:
```
{'Human': ['hg38', 'hg19'],
 'Mouse': ['mm10', 'mm9'],
 'S.cerevisiae': ['sacCer2', 'sacCer3'],
 'Zebrafish': ['danRer7', 'danRer10', 'danRer11'],
 'Xenopus': ['xenTro2', 'xenTro3'],
 'Rat': ['rn4', 'rn5', 'rn6'],
 'Drosophila': ['dm3', 'dm6'],
 'C.elegans': ['ce6', 'ce10'],
 'Arabidopsis': ['tair10'],
 'Chicken': ['galGal4', 'galGal5', 'galGal6']}
```

[8]:
```

tss_annotated = ma.get_tss_info(peak_str_list=peaks, ref_genome="mm10") ##!! Set␣
↪reference genome here


# Check results
tss_annotated.tail()
```

```
que bed peaks: 86935
tss peaks in que: 17238
```

[8]:

	chr	start	end	gene_short_name	strand
17233	chr1	55130650	55132118	Mob4	+
17234	chr6	94499875	94500767	Slc25a26	+
17235	chr19	45659222	45660823	Fbxw4	-
17236	chr12	100898848	100899597	Gpr68	-
17237	chr4	129491262	129492047	Fam229a	-

3. Integrate TSS info and cicero connections

he output file after the integration process has three columns: `["peak_id", "gene_short_name", "coaccess"]`.

- "peak_id" is either the TSS peak or the peaks that have a connection with the TSS peak.
- "gene_short_name" is the gene name that associated with the TSS site.
- "coaccess" is the co-access score between a peak and TSS peak. If the score is 1, it means that the peak is TSS itself.

[9]:
```
integrated = ma.integrate_tss_peak_with_cicero(tss_peak=tss_annotated,
                                               cicero_connections=cicero_connections)
print(integrated.shape)
integrated.head()
```

```
(44309, 3)
```

[9]:

	peak_id	gene_short_name	coaccess
0	chr10_100006139_100006389	Tmtc3	0.017915
1	chr10_100015291_100017830	Kitl	1.000000
2	chr10_100018677_100020384	Kitl	0.146517
3	chr10_100050858_100051762	Kitl	0.069751
4	chr10_100052829_100053395	Kitl	0.202670

4. Filter peaks

Remove peaks that have weak coaccess score.

[10]:
```
peak = integrated[integrated.coaccess >= 0.8]
peak = peak[["peak_id", "gene_short_name"]].reset_index(drop=True)
```

[11]:
```
print(peak.shape)
peak.head()
```

```
(15779, 2)
```

[11]:

	peak_id	gene_short_name
0	chr10_100015291_100017830	Kitl
1	chr10_100486534_100488209	Tmtc3
2	chr10_100588641_100589556	4930430F08Rik
3	chr10_100741247_100742505	Gm35722
4	chr10_101681379_101682124	Mgat4c

5. Save data

Save the promoter/enhancer peak.

[12]:
```
peak.to_csv("processed_peak_file.csv")
```

Please go to next step: Transcriptoin factor motif scan

https://morris-lab.github.io/CellOracle.documentation/tutorials/motifscan.html

[ ]:
```

```

Once you get the input data, please go to the Motif scan section.

Option2. Data preprocessing of bulk ATAC-seq data

Bulk DNA-seq data can be used to get the accessible promoter/enhancer sequences.

The jupyter notebook file is available here .

Or click below to see the contents.

Overview

In this notebook, we will make TSS annotation in the bulk scATAC-seq data to get input data of base-GRN construction.

Notebook file

Notebook file is available here. https://github.com/morris-lab/CellOracle/blob/master/docs/notebooks/01_ATAC-seq_data_processing/option2_Bulk_ATAC-seq_data/01_preprocess_Bulk_ATAC_seq_peak_data.ipynb

0. Import libraries

[7]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns

import os, sys, shutil, importlib, glob
from tqdm import tqdm_notebook as tqdm

%config InlineBackend.figure_format = 'retina'

plt.rcParams['figure.figsize'] = [6, 4.5]
plt.rcParams["savefig.dpi"] = 300
```

[8]:
```
# Import celloracle function
from celloracle import motif_analysis as ma
```

1. Load input data

Import ATAC-seq bed file. This script can also be used with DNase-seq or Chip-seq data.

Here, we use bulk ATAC-seq data. Please prepare bulk ATAC-seq data as a bed file format.

You can download the demo file by running the following command: If the file download fails, please manually download and unzip the data.

https://raw.githubusercontent.com/morris-lab/CellOracle/master/docs/demo_data/bulk_ATAC_seq_peak_data.bed

[3]:
```
# Download file.
!wget https://raw.githubusercontent.com/morris-lab/CellOracle/master/docs/demo_data/
↪bulk_ATAC_seq_peak_data.bed

# If you are using macOS, please try the following command.
#!curl -O https://raw.githubusercontent.com/morris-lab/CellOracle/master/docs/demo_
↪data/bulk_ATAC_seq_peak_data.bed
```

```
--2021-07-07 21:38:59--  https://raw.githubusercontent.com/morris-lab/CellOracle/
↪master/docs/demo_data/bulk_ATAC_seq_peak_data.bed
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.110.133,␣
↪185.199.109.133, 185.199.108.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.110.133|:
↪443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 10446347 (10.0M) [text/plain]
Saving to: 'bulk_ATAC_seq_peak_data.bed'

bulk_ATAC_seq_peak_ 100%[===================>]   9.96M  --.-KB/s    in 0.1s

2021-07-07 21:39:00 (80.3 MB/s) - 'bulk_ATAC_seq_peak_data.bed' saved [10446347/
↪10446347]
```

[9]:
```
# Load bed_file
file_path_of_bed_file = "bulk_ATAC_seq_peak_data.bed"
bed = ma.read_bed(file_path_of_bed_file)
print(bed.shape)
bed.head()
```

```
(436206, 4)
```

[9]:

	chrom	start	end	seqname
0	chr1	3002478	3002968	chr1_3002478_3002968
1	chr1	3084739	3085712	chr1_3084739_3085712
2	chr1	3103576	3104022	chr1_3103576_3104022
3	chr1	3106871	3107210	chr1_3106871_3107210
4	chr1	3108932	3109158	chr1_3108932_3109158

[10]:
```
# Convert bed file into peak name list
peaks = ma.process_bed_file.df_to_list_peakstr(bed)
peaks
```

[10]:
```
array(['chr1_3002478_3002968', 'chr1_3084739_3085712',
       'chr1_3103576_3104022', ..., 'chrY_631222_631480',
       'chrY_795887_796426', 'chrY_2397419_2397628'], dtype=object)
```

2. Make TSS annotation

IMPORTANT: Please make sure that you are setting the correct ref genome!

[11]:
```
tss_annotated = ma.get_tss_info(peak_str_list=peaks, ref_genome="mm9")

# Check results
tss_annotated.tail()
```

```
que bed peaks: 436206
tss peaks in que: 24822
```

[11]:

	chr	start	end	gene_short_name	strand
24817	chr2	60560211	60561602	Itgb6	-
24818	chr15	3975177	3978654	BC037032	-
24819	chr14	67690701	67692101	Ppp2r2a	-
24820	chr17	48455247	48455773	B430306N03Rik	+
24821	chr10	59861192	59861608	Gm17455	+

[12]:
```
# Change format
peak_id_tss = ma.process_bed_file.df_to_list_peakstr(tss_annotated)
tss_annotated = pd.DataFrame({"peak_id": peak_id_tss,
                              "gene_short_name": tss_annotated.gene_short_name.values}
↪)
tss_annotated = tss_annotated.reset_index(drop=True)
print(tss_annotated.shape)
tss_annotated.head()
```

```
(24822, 2)
```

[12]:

	peak_id	gene_short_name
0	chr7_50691730_50692032	Nkg7
1	chr7_50692077_50692785	Nkg7
2	chr13_93564413_93564836	Thbs4
3	chr13_14613429_14615645	Hecw1
4	chr3_99688753_99689665	Spag17

3. Save data

[10]:
```
tss_annotated.to_csv("processed_peak_file.csv")
```

Please go to next step: Transcriptoin factor motif scan

https://morris-lab.github.io/CellOracle.documentation/tutorials/motifscan.html

[ ]:

TF motif scan for base-GRN construction

Transcription factor binding motif scan

In the previous section, we got accessible Promoter/enhancer DNA regions using ATAC-seq data. Next, we will obtain a base-GRN by scanning the regulatory genomic sequences for TF-binding motifs. In the later GRN inference process, this list will be used to define potential regulatory connections.

The jupyter notebook files and data used in this tutorial are available here .

Scan DNA sequences searching for TF binding motifs

Python notebook

Overview

This notebook introduce how to perform TF binding motif scan. Using scATAC-seq peak and Motif information, we generate base-GRN.

Notebook file

Notebook file is available at CellOracle GitHub. https://github.com/morris-lab/CellOracle/blob/master/docs/notebooks/02_motif_scan/02_atac_peaks_to_TFinfo_with_celloracle_20200801.ipynb

0. Import libraries

[10]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


import seaborn as sns

import os, sys, shutil, importlib, glob
from tqdm.notebook import tqdm
```

[11]:
```
from celloracle import motif_analysis as ma
from celloracle.utility import save_as_pickled_object
```

[12]:
```
%config InlineBackend.figure_format = 'retina'
%matplotlib inline

plt.rcParams['figure.figsize'] = (15,7)
plt.rcParams["savefig.dpi"] = 600
```

1. Rerefence genome data preparation

Before starting celloracle analysis, we need to make sure that the reference genome data is correctly installed in your computational environment. If not, please install reference genome first as follows.

[18]:
```
# PLEASE make sure that you are setting correct ref genome.
ref_genome = "mm10"

genome_installation = ma.is_genome_installed(ref_genome=ref_genome)
print(ref_genome, "installation: ", genome_installation)
```

```
mm10 installation:  True
```

[19]:
```
if not genome_installation:
    import genomepy
    genomepy.install_genome(ref_genome, "UCSC")
else:
    print(ref_genome, "is installed.")
```

```
mm10 is installed.
```

2. Load data

In this notebook, we explain how to make base GRN data.

Please look at the previous steps to see an example of input data preparation method.

https://morris-lab.github.io/CellOracle.documentation/tutorials/base_grn.html#step1-scatac-seq-analysis-with-cicero

As a input data, we need scATAC-seq file in the following format.

- Prepare input data as a csv file with tree columns.
- The first column is index.
- The second column is peak_id.
- The third column is gene_short_name.

The csv file should be like this.

```
,peak_id,gene_short_name
0,chr10_100015291_100017830,Kitl
1,chr10_100486534_100488209,Tmtc3
2,chr10_100588641_100589556,4930430F08Rik
3,chr10_100741247_100742505,Gm35722
4,chr10_101681379_101682124,Mgat4c
5,chr10_102158688_102159257,Mgat4c
6,chr10_102511934_102512015,Rassf9
7,chr10_103026814_103029423,Alx1
8,chr10_103235705_103236587,Lrriq1
9,chr10_103366977_103369690,Slc6a15
10,chr10_10472105_10472772,Adgb
11,chr10_105573396_105575735,Gm15663
12,chr10_105573396_105575735,Tmtc2
13,chr10_10557396_10558671,Rab32
14,chr10_105840548_105842058,Ccdc59
15,chr10 105840548 105842058,Mettl25
```

We load this csv file using `pd.read_csv()` to make pandas.DataFrame like this.

	peak_id	gene_short_name
0	chr10_100015291_100017830	Kitl
1	chr10_100486534_100488209	Tmtc3
2	chr10_100588641_100589556	4930430F08Rik
3	chr10_100741247_100742505	Gm35722
4	chr10_101681379_101682124	Mgat4c

You can download the demo file by running the following command: If the file download fails, please manually

download and unzip the data.

https://raw.githubusercontent.com/morris-lab/CellOracle/master/docs/demo_data/processed_peak_file.csv

[7]:
```
# Download file.
!wget https://raw.githubusercontent.com/morris-lab/CellOracle/master/docs/demo_data/
↪processed_peak_file.csv

# If you are using macOS, please try the following command.
#!curl -O https://raw.githubusercontent.com/morris-lab/CellOracle/master/docs/demo_
↪data/processed_peak_file.csv
```

```
--2021-07-07 21:49:27--  https://raw.githubusercontent.com/morris-lab/CellOracle/
↪master/docs/demo_data/processed_peak_file.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.110.133,␣
↪185.199.109.133, 185.199.108.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.110.133|:
↪443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 569448 (556K) [text/plain]
Saving to: 'processed_peak_file.csv'

processed_peak_file 100%[===================>] 556.10K  --.-KB/s    in 0.02s

2021-07-07 21:49:27 (29.2 MB/s) - 'processed_peak_file.csv' saved [569448/569448]
```

[13]:
```
# Load annotated peak data.
peaks = pd.read_csv("processed_peak_file.csv", index_col=0)
peaks.head()
```

[13]:

	peak_id	gene_short_name
0	chr10_100015291_100017830	Kitl
1	chr10_100486534_100488209	Tmtc3
2	chr10_100588641_100589556	4930430F08Rik
3	chr10_100741247_100742505	Gm35722
4	chr10_101681379_101682124	Mgat4c

[16]:
```
# Define function for quality check
def decompose_chrstr(peak_str):
    """
    Args:
        peak_str (str): peak_str. e.g. 'chr1_3094484_3095479'

    Returns:
        tuple: chromosome name, start position, end position
    """

    *chr_, start, end = peak_str.split("_")
    chr_ = "_".join(chr_)
    return chr_, start, end

from genomepy import Genome

def check_peak_foamat(peaks_df, ref_genome):
    """
    Check peak fomat.
     (1) Check chromosome name.
```

(continues on next page)

(continued from previous page)

```
    (2) Check peak size (length) and remove sort DNAs (<5bp)

    """

    df = peaks_df.copy()

    n_peaks_before = df.shape[0]

    # Decompose peaks and make df
    decomposed = [decompose_chrstr(peak_str) for peak_str in df["peak_id"]]
    df_decomposed = pd.DataFrame(np.array(decomposed))
    df_decomposed.columns = ["chr", "start", "end"]
    df_decomposed["start"] = df_decomposed["start"].astype(np.int)
    df_decomposed["end"] = df_decomposed["end"].astype(np.int)

    # Load genome data
    genome_data = Genome(ref_genome)
    all_chr_list = list(genome_data.keys())


    # DNA length check
    lengths = np.abs(df_decomposed["end"] - df_decomposed["start"])


    # Filter peaks with invalid chromosome name
    n_threshold = 5
    df = df[(lengths >= n_threshold) & df_decomposed.chr.isin(all_chr_list)]

    # DNA length check
    lengths = np.abs(df_decomposed["end"] - df_decomposed["start"])

    # Data counting
    n_invalid_length = len(lengths[lengths < n_threshold])
    n_peaks_invalid_chr = n_peaks_before - df_decomposed.chr.isin(all_chr_list).sum()
    n_peaks_after = df.shape[0]

    #
    print("Peaks before filtering: ", n_peaks_before)
    print("Peaks with invalid chr_name: ", n_peaks_invalid_chr)
    print("Peaks with invalid length: ", n_invalid_length)
    print("Peaks after filtering: ", n_peaks_after)

    return df
```

[20]:
```
peaks = check_peak_foamat(peaks, ref_genome)
```

```
Peaks before filtering:  15779
Peaks with invalid chr_name:  0
Peaks with invalid length:  2
Peaks after filtering:  15777
```

You can select TF binding motif data for Celloracle motif analysis. If you have no preference and just want to use a default motif, you don't need to load motif yourself.

If you want to use a non-default motif dataset, we have several options.

- Use custom motifs provided by gimmemotifs >Gimmemotifs is a python package for motif analysis. It provides many motif dataset. https://gimmemotifs.readthedocs.io/en/stable/overview.html#

motif-databases > > Please look at this notebook to see how to load motif data from gimmemotifs database. > https://github.com/morris-lab/CellOracle/blob/master/docs/notebooks/02_motif_scan/motif_data_preparation/01_How_to_load_gimmemotifs_motif_data.ipynb

- Use custom motifs provided by CellOracle.

  Celloracle also provides many motif datasets generated from CisBP. http://cisbp.ccbr.utoronto.ca/

  Please look at this notebook to see how to load CisBP motifs.https://github.com/morris-lab/CellOracle/blob/master/docs/notebooks/02_motif_scan/motif_data_preparation/02_How_to_load_CisBPv2_motif_data.ipynb

- Make a custom motif data by yourself. >You can create custom motif data by yourself. > >Please look at this notebook to see how to create custom motif data.https://github.com/morris-lab/CellOracle/blob/master/docs/notebooks/02_motif_scan/motif_data_preparation/03_How_to_make_custom_motif.ipynb

3. Instantiate TFinfo object and search for TF binding motifs

The motif analysis module has a custom class, `TFinfo`. The TFinfo object will do all steps below.

- Converts a peak data into a DNA sequences.
- Ccans the DNA sequences searching for TF binding motifs.
- Postprocess the results of motif scan.
- Converted data into appropriate format. You can convert data into base-GRN. You can select file format: python dictionary or pandas dataframe. This output data, base-GRN is necessary for GRN model construction in the later step.

[16]:
```
# Instantiate TFinfo object
tfi = ma.TFinfo(peak_data_frame=peaks,
                ref_genome=ref_genome)
```

You can specify TF binding motif data as follows.

`tfi.scan(motifs=motifs)`

If you don't set motifs or set `None`, default motifs will be loaded automatically.

- For mouse and human, "gimme.vertebrate.v5.0." will be used as a default motifs.
- For another species, a species specific TF binding motif data extracted from CisBP ver2.0 will be used.

[ ]:
```
%%time
# Scan motifs. !!CAUTION!! This step may take several hours if you have many peaks!
tfi.scan(fpr=0.02,
         motifs=None,  # If you enter None, default motifs will be loaded.
         verbose=True)

# Save tfinfo object
tfi.to_hdf5(file_path="test1.celloracle.tfinfo")
```

[16]:
```
# Check motif scan results
tfi.scanned_df.head()
```

[16]:
```
                     seqname                      motif_id factors_direct  \
0  chr10_100015291_100017830      GM.5.0.Homeodomain.0001          TGIF1
1  chr10_100015291_100017830            GM.5.0.Mixed.0001
2  chr10_100015291_100017830            GM.5.0.Mixed.0001
3  chr10_100015291_100017830            GM.5.0.Mixed.0001
```

(continues on next page)

(continued from previous page)

```
4  chr10_100015291_100017830  GM.5.0.Nuclear_receptor.0002          NR2C2

         factors_indirect      score   pos  strand
0  ENSG00000234254, TGIF1  10.311002  1003       1
1              SRF, EGR1   7.925873   481       1
2              SRF, EGR1   7.321375   911      -1
3              SRF, EGR1   7.276585   811      -1
4           NR2C2, Nr2c2   9.067331   449      -1
```

We have the score for each sequence and motif_id pair. In the next step we will filter the motifs with low score.

4. Filtering motifs

[15]:
```
# Reset filtering
tfi.reset_filtering()

# Do filtering
tfi.filter_motifs_by_score(threshold=10)

# Do post filtering process. Convert results into several file format.
tfi.make_TFinfo_dataframe_and_dictionary(verbose=True)
```

```
HBox(children=(FloatProgress(value=0.0, max=14142.0), HTML(value='')))
```

```
HBox(children=(FloatProgress(value=0.0, max=15006.0), HTML(value='')))
```

```
HBox(children=(FloatProgress(value=0.0, max=1090.0), HTML(value='')))
```

5. Get Final results

[17]:
```
df = tfi.to_dataframe()
df.head()
```

[17]:
```
                    peak_id gene_short_name  9430076c15rik  Ac002126.6  \
0  chr10_100015291_100017830           Kitl            0.0         0.0
1  chr10_100486534_100488209          Tmtc3            0.0         0.0
2  chr10_100588641_100589556  4930430F08Rik            0.0         0.0
3  chr10_100741247_100742505        Gm35722            0.0         0.0
4  chr10_101681379_101682124         Mgat4c            0.0         0.0

   Ac012531.1  Ac226150.2  Afp  Ahr  Ahrr  Aire  ...  Znf784  Znf8  Znf816  \
0         0.0         0.0  0.0  1.0   1.0   0.0  ...     0.0   0.0     0.0
1         0.0         0.0  0.0  0.0   0.0   0.0  ...     1.0   0.0     0.0
2         1.0         0.0  0.0  1.0   1.0   0.0  ...     0.0   0.0     0.0
3         0.0         0.0  0.0  0.0   0.0   0.0  ...     0.0   0.0     0.0
4         0.0         0.0  0.0  0.0   0.0   0.0  ...     0.0   0.0     0.0

   Znf85  Zscan10  Zscan16  Zscan22  Zscan26  Zscan31  Zscan4
0    0.0      0.0      0.0      0.0      0.0      1.0     0.0
1    0.0      0.0      0.0      1.0      0.0      0.0     0.0
2    0.0      0.0      0.0      0.0      0.0      0.0     0.0
3    0.0      0.0      0.0      0.0      0.0      0.0     0.0
```

(continues on next page)

(continued from previous page)

```
4    0.0      0.0      0.0      0.0      0.0      0.0      1.0

[5 rows x 1092 columns]
```

6. Save result

We'll use this information when making the GRNs. Save the results.

[19]:
```
# Save result as a dataframe
df = tfi.to_dataframe()
df.to_parquet(os.path.join(folder, "base_GRN_dataframe.parquet"))

# If you want, you can save the result as a dictionary as follows.
#td = tfi.to_dictionary(dictionary_type="targetgene2TFs")
#save_as_pickled_object(td, os.path.join(folder, "TFinfo_targetgene2TFs.pickled"))
```

We will use this base-GRN data in the GRN construction section.

https://morris-lab.github.io/CellOracle.documentation/tutorials/networkanalysis.html

[ ]:
```
```

How to use different motif data

Celloracle provides several dafault motifs. If you don't enter motif data, celloracle automatically load default motifs for your species. In most case, you don't need to prepare TF binding motifs yourself.

But you can use another motif data.

gimmemotifs motif data

Here is the notebook describing how to load a motif data from gimmemotifs database. https://github.com/morris-lab/CellOracle/blob/master/docs/notebooks/02_motif_scan/motif_data_preparation/01_How_to_load_gimmemotifs_motif_data.ipynb

CellOracle motif dataset generated from CisBP version2 database

Here is the notebook describing how to load a motif data from CisBP version2 database. https://github.com/morris-lab/CellOracle/blob/master/docs/notebooks/02_motif_scan/motif_data_preparation/02_How_to_load_CisBPv2_motif_data.ipynb

How to create custom motif data

We can creaet motif data by ourself. Here is an example code. https://github.com/morris-lab/CellOracle/blob/master/docs/notebooks/02_motif_scan/motif_data_preparation/03_How_to_make_custom_motif.ipynb

2.3 API

2.3.1 Command Line API

CellOracle has a command line API. This command can be used to convert scRNA-seq data. If you have a scRNA-seq data which was processed with Seurat and saved as Rds file, you can use the following command to make anndata from Seurat object. The anndata object produced by this command can be used for input of celloracle.

```
seuratToAnndata YOUR_SEURAT_OBJECT.Rds OUTPUT_PATH
```

2.3.2 Python API

Custom class in celloracle

We define some custom classes in celloracle.

class celloracle.**Links**(*name*, *links_dict={}*)

Bases: object

This is a class for the processing and visualization of GRNs. Links object stores cluster-specific GRNs and metadata. Please use "get_links" function in Oracle object to generate Links object.

links_dict
Dictionary that store unprocessed network data.

Type dictionary

filtered_links
Dictionary that store filtered network data.

Type dictionary

merged_score
Network scores.

Type pandas.dataframe

cluster
List of cluster name.

Type list of str

name
Name of clustering unit.

Type str

palette
DataFrame that store color information.

Type pandas.dataframe

filter_links(*p=0.001*, *weight='coef_abs'*, *threshold_number=10000*, *genelist_source=None*, *genelist_target=None*, *thread_number=None*)

Filter network edges. In most cases, inferred GRN has non-significant random edges. We have to remove these edges before analyzing the network structure. You can do the filtering in any of the following ways.

(1) Filter based on the p-value of the network edge. Please enter p-value for thresholding.

(2) Filter based on network edge number. If you set the number, network edges will be filtered based on the order of a network score. The top n-th network edges with network weight will remain, and the other edges will be removed. The network data has several types of network weight, so you have to select which network weight do you want to use.

(3) Filter based on an arbitrary gene list. You can set a gene list for source nodes or target nodes.

Parameters

- **p** (*float*) – threshold for p-value of the network edge.
- **weight** (*str*) – Please select network weight name for the filtering
- **genelist_source** (*list of str*) – gene list to remain in regulatory gene nodes. Default is None.
- **genelist_target** (*list of str*) – gene list to remain in target gene nodes. Default is None.

get_network_entropy(*value='coef_abs'*)

Calculate network entropy scores.

Parameters **value** (*str*) – Default is "coef_abs".

get_score(*test_mode=False*)

Get several network sores using R libraries. Make sure all dependent R libraries are installed in your environment before running this function. You can check the installation for the R libraries by running test_installation() in network_analysis module.

plot_cartography_scatter_per_cluster(*gois=None*, *clusters=None*, *scatter=True*, *kde=False*, *auto_gene_annot=False*, *percentile=98*, *args_dot={'n_levels': 105}*, *args_line={'c': 'gray'}*, *args_annot={}*, *save=None*)

Make a gene network cartography plot. Please read the original paper describing gene network cartography for more information. https://www.nature.com/articles/nature03288

Parameters

- **links** (Links) – See network_analysis.Links class for detail.
- **gois** (*list of srt*) – List of gene name to highlight.
- **clusters** (*list of str*) – List of cluster name to analyze. If None, all clusters in Links object will be analyzed.
- **scatter** (*bool*) – Whether to make a scatter plot.
- **auto_gene_annot** (*bool*) – Whether to pick up genes to make an annotation.
- **percentile** (*float*) – Genes with a network score above the percentile will be shown with annotation. Default is 98.
- **args_dot** (*dictionary*) – Arguments for scatter plot.
- **args_line** (*dictionary*) – Arguments for lines in cartography plot.

- **args_annot** (*dictionary*) – Arguments for annotation in plots.
- **save** (*str*) – Folder path to save plots. If the folder does not exist in the path, the function creates the folder. Plots will not be saved if [save=None]. Default is None.

plot_cartography_term(*goi*, *save=None*, *plt_show=True*)

Plot the gene network cartography term like a heatmap. Please read the original paper of gene network cartography for the principle of gene network cartography. https://www.nature.com/articles/nature03288

Parameters

- **links** (Links) – See network_analysis.Links class for detail.
- **gois** (*list of srt*) – List of gene name to highlight.
- **save** (*str*) – Folder path to save plots. If the folder does not exist in the path, the function creates the folder. Plots will not be saved if [save=None]. Default is None.

plot_degree_distributions(*plot_model=False*, *save=None*)

Plot the network degree distributions (the number of edge per gene). The network degree will be visualized in both linear scale and log scale.

Parameters

- **links** (Links) – See network_analysis.Links class for detail.
- **plot_model** (*bool*) – Whether to plot linear approximation line.
- **save** (*str*) – Folder path to save plots. If the folder does not exist in the path, the function creates the folder. Plots will not be saved if [save=None]. Default is None.

plot_network_entropy_distributions(*update_network_entropy=False*, *save=None*)

Plot the distribution for network entropy. See the CellOracle paper for more detail.

Parameters

- **links** (*Links object*) – See network_analysis.Links class for detail.
- **values** (*list of str*) – The list of score to visualize. If it is None, all network score (listed above) will be used.
- **update_network_entropy** (*bool*) – Whether to recalculate network entropy.
- **save** (*str*) – Folder path to save plots. If the folder does not exist in the path, the function creates the folder. Plots will not be saved if [save=None]. Default is None.

plot_score_comparison_2D(*value*, *cluster1*, *cluster2*, *percentile=99*, *annot_shifts=None*, *save=None*, *plt_show=True*, *interactive=False*)

Make a scatter plot that compares specific network scores in two groups.

Parameters

- **links** (Links) – See network_analysis.Links class for detail.
- **value** (*srt*) – The network score type.
- **cluster1** (*str*) – Cluster name. Network scores in cluster1 will be visualized in the x-axis.
- **cluster2** (*str*) – Cluster name. Network scores in cluster2 will be visualized in the y-axis.
- **percentile** (*float*) – Genes with a network score above the percentile will be shown with annotation. Default is 99.
- **annot_shifts** (*(float, float)*) – Annotation visualization setting.

- **save** (*str*) – Folder path to save plots. If the folder does not exist in the path, the function creates the folder. Plots will not be saved if [save=None]. Default is None.

plot_score_discributions(*values=None*, *method='boxplot'*, *save=None*)
Plot the distribution of network scores. An individual data point is a network edge (gene).

Parameters

- **links** (Links) – See Links class for details.
- **values** (*list of str*) – The list of score to visualize. If it is None, all of the network score will be used.
- **method** (*str*) – Plotting method. Select either "boxplot" or "barplot".
- **save** (*str*) – Folder path to save plots. If the folder does not exist in the path, the function creates the folder. Plots will not be saved if [save=None]. Default is None.

plot_score_per_cluster(*goi*, *save=None*, *plt_show=True*)
Plot network score for a gene. This function visualizes the network score for a specific gene between clusters to get an insight into the dynamics of the gene.

Parameters

- **links** (Links) – See network_analysis.Links class for detail.
- **goi** (*srt*) – Gene name.
- **save** (*str*) – Folder path to save plots. If the folder does not exist in the path, the function creates the folder. Plots will not be saved if [save=None]. Default is None.

plot_scores_as_rank(*cluster*, *n_gene=50*, *save=None*)
Pick up top n-th genes wich high-network scores and make plots.

Parameters

- **links** (Links) – See network_analysis.Links class for detail.
- **cluster** (*str*) – Cluster name to analyze.
- **n_gene** (*int*) – Number of genes to plot. Default is 50.
- **save** (*str*) – Folder path to save plots. If the folder does not exist in the path, the function creates the folder. Plots will not be saved if [save=None]. Default is None.

to_hdf5(*file_path*)
Save object as hdf5.

Parameters **file_path** (*str*) – file path to save file. Filename needs to end with '.celloracle.links'

class celloracle.**Net**(*gene_expression_matrix*, *gem_standerdized=None*, *TFinfo_matrix=None*, *cell-state=None*, *TFinfo_dic=None*, *annotation=None*, *verbose=True*)
Bases: object

Net is a custom class for inferring sample-specific GRN from scRNA-seq data. This class is used inside the Oracle class for GRN inference. This class requires two types of information below.

(1) Single-cell RNA-seq data: The Net class needs processed scRNA-seq data. Gene and cell filtering, quality check, normalization, log-transformation (but not scaling and centering) have to be done before starting the GRN calculation with this class. You can also use any arbitrary metadata (i.e., mRNA count, cell-cycle phase) for GRN input.

(2) Potential regulatory connection (or base GRN): This method uses the list of potential regulatory TFs as input. This information can be calculated from ATAC-seq data using the motif-analysis module. If sample-specific ATAC-seq data is not available, you can use general TF-binding info derived from public ATAC-seq dataset of various tissue/cell type.

linkList
The results of the GRN inference.

Type pandas.DataFrame

all_genes
An array of all genes that exist in the input gene expression matrix

Type numpy.array

embedding_name
The key name name in adata.obsm containing dimensional reduction coordinates

Type str

annotation
Annotation. you can add custom annotation.

Type dictionary

coefs_dict
Coefs of linear regression.

Type dictionary

stats_dict
Statistic values about coefs.

Type dictionary

fitted_genes
List of genes where the regression model was successfully calculated.

Type list of str

failed_genes
List of genes that were not assigned coefs

Type list of str

cellstate
A metadata for GRN input

Type pandas.DataFrame

TFinfo
Information about potential regulatory TFs.

Type pandas.DataFrame

gem
Merged matrix made with gene_expression_matrix and cellstate matrix.

Type pandas.DataFrame

gem_standerdized
Almost the same as gem, but the gene_expression_matrix was standardized.

Type pandas.DataFrame

library_last_update_date
Last update date of this code. This info is for code development. It can be deprecated in the future

Type str

object_initiation_date
The date when this object was made.

Type str

addAnnotation(*annotation_dictionary*)
Add a new annotation.

Parameters **annotation_dictionary** (*dictionary*) – e.g. {"sample_name": "NIH 3T3 cell"}

addTFinfo_dictionary(*TFdict*)
Add a new TF info to pre-exiting TFdict.

Parameters **TFdict** (*dictionary*) – python dictionary of TF info.

addTFinfo_matrix(*TFinfo_matrix*)
Load TF info dataframe.

Parameters **TFinfo** (*pandas.DataFrame*) – information about potential regulatory TFs.

copy()
Deepcopy itself

fit_All_genes(*bagging_number=200*, *scaling=True*, *model_method='bagging_ridge'*, *command_line_mode=False*, *log=None*, *alpha=1*, *verbose=True*)
Make ML models for all genes. The calculation will be performed in parallel using scikit-learn bagging function. You can select a modeling method (bagging_ridge or bayesian_ridge). This calculation usually takes a long time.

Parameters

- **bagging_number** (*int*) – The number of estimators for bagging.
- **scaling** (*bool*) – Whether or not to scale regulatory gene expression values.
- **model_method** (*str*) – ML model name. Please select either "bagging_ridge" or "bayesian_ridge"
- **command_line_mode** (*bool*) – Please select False if the calculation is performed on jupyter notebook.
- **log** (*logging object*) – log object to output log
- **alpha** (*int*) – Strength of regularization.
- **verbose** (*bool*) – Whether or not to show a progress bar.

fit_All_genes_parallel(*bagging_number=200*, *scaling=True*, *log=None*, *verbose=10*)
IMPORTANT: this function being debugged and is currently unavailable.

Make ML models for all genes. The calculation will be performed in parallel using joblib parallel module.

Parameters

- **bagging_number** (*int*) – The number of estimators for bagging.
- **scaling** (*bool*) – Whether or not to scale regulatory gene expression values.
- **log** (*logging object*) – log object to output log
- **verbose** (*int*) – verbose for joblib parallel

fit_genes(*target_genes*, *bagging_number=200*, *scaling=True*, *model_method='bagging_ridge'*, *save_coefs=False*, *command_line_mode=False*, *log=None*, *alpha=1*, *verbose=True*)

Make ML models for genes of interest. This calculation will be performed in parallel using scikit-learn's bagging function. You can select a modeling method; Please chose either bagging_ridge or bayesian_ridge.

Parameters

- **target_genes** (*list of str*) – gene list
- **bagging_number** (*int*) – The number of estimators for bagging.
- **scaling** (*bool*) – Whether or not to scale regulatory gene expression values.
- **model_method** (*str*) – ML model name. Please select either "bagging_ridge" or "bayesian_ridge"
- **save_coefs** (*bool*) – Whether or not to store details of coef values in bagging model.
- **command_line_mode** (*bool*) – Please select False if the calculation is performed on jupyter notebook.
- **log** (*logging object*) – log object to output log
- **alpha** (*int*) – Strength of regularization.
- **verbose** (*bool*) – Whether or not to show a progress bar.

plotCoefs(*target_gene*, *sort=True*, *threshold_p=None*)

Plot the distribution of Coef values (network edge weights).

Parameters

- **target_gene** (*str*) – gene name
- **sort** (*bool*) – Whether or not to sort genes by its strength
- **bagging_number** (*int*) – The number of estimators for bagging.
- **threshold_p** (*float*) – the threshold for p-values. TFs will be filtered based on the p-value. if None, no filtering is applied.

to_hdf5(*file_path*)

Save object as hdf5.

Parameters **file_path** (*str*) – file path to save file. Filename needs to end with '.celloracle.net'

updateLinkList(*verbose=True*)

Update LinkList. LinkList is a data frame that store information about inferred GRNs.

Parameters **verbose** (*bool*) – Whether or not to show a progress bar

updateTFinfo_dictionary(*TFdict*)

Update TF info matrix

Parameters **TFdict** (*dictionary*) – A python dictionary in which a key is Target gene, value are potential regulatory genes for the target gene.

class celloracle.**Oracle**

Bases: celloracle.trajectory.modified_VelocytoLoom_class.modified_VelocytoLoom, celloracle.visualizations.oracle_object_visualization.Oracle_visualization

Oracle is the main class in CellOracle. Oracle object imports scRNA-seq data (anndata) and TF information to infer cluster-specific GRNs. It can predict the future gene expression patterns and cell state transitions in

response to the perturbation of TFs. Please see the CellOracle paper for details. The code of the Oracle class was made of the three components below.

(1) Anndata: Gene expression matrix and metadata from single-cell RNA-seq are stored in the anndata object. Processed values, such as normalized counts and simulated values, are stored as layers of anndata. Metadata (i.e., Cluster info) are saved in anndata.obs. Refer to scanpy/anndata documentation for detail.

(2) Net: Net is a custom class in celloracle. Net object processes several data to infer GRN. See the Net class documentation for details.

(3) VelycytoLoom: Calculation of transition probability and visualization of directed trajectory graph will be performed in the same way as velocytoloom. VelocytoLoom is class from Velocyto, a python library for RNA-velocity analysis. In celloracle, we use some functions in velocytoloom for the visualization.

adata
Imported anndata object

Type anndata

cluster_column_name
The column name in adata.obs containing cluster info

Type str

embedding_name
The key name in adata.obsm containing dimensional reduction cordinates

Type str

addTFinfo_dictionary(*TFdict*)
Add new TF info to pre-existing TFdict. Values in the old TF dictionary will remain.

Parameters **TFdict** (*dictionary*) – Python dictionary of TF info.

calculate_mass_filter(*min_mass=0.01*, *plot=False*)

calculate_p_mass(*smooth=0.8*, *n_grid=40*, *n_neighbors=200*, *n_jobs=- 1*)

change_cluster_unit(*new_cluster_column_name*)
Change clustering unit. If you change cluster, previous GRN data and simulation data will be delated. Please re-calculate GRNs.

copy()
Deepcopy itself.

count_cells_in_mc_resutls(*cluster_use*, *end=- 1*, *order=None*)
Count the simulated cell by the cluster.

Parameters

- **cluster_use** (*str*) – cluster information name in anndata.obs. You can use any cluster information in anndata.obs.
- **end** (*int*) – The end point of Sankey-diagram. Please select a step in the Markov simulation. if you set [end=-1], the final step of Markov simulation will be used.

Returns Number of cells before / after simulation

Return type pandas.DataFrame

extract_active_gene_lists(*return_as=None*, *verbose=False*)

Parameters

- **return_as** (*str*) – If not None, it returns dictionary or list. Chose either "individual_dict" or "unified_list".

- **verbose** (*bool*) – Whether to show progress bar.

Returns The format depends on the argument, "return_as".

Return type dictionary or list

fit_GRN_for_simulation(*GRN_unit='cluster'*, *alpha=1*, *use_cluster_specific_TFdict=False*)
Do GRN inference. Please see the paper of CellOracle paper for details.

GRN can be constructed for the entire population or each clusters. If you want to infer cluster-specific GRN, please set [GRN_unit="cluster"]. You can select cluster information when you import data.

If you set [GRN_unit="whole"], GRN will be made using all cells.

Parameters

- **GRN_unit** (*str*) – Select "cluster" or "whole"
- **alpha** (*float or int*) – The strength of regularization. If you set a lower value, the sensitivity increases, and you can detect weaker network connections. However, there may be more noise. If you select a higher value, it will reduce the chance of overfitting.

get_cluster_specific_TFdict_from_Links(*links_object*, *ignore_warning=False*)
Extract TF and its target gene information from Links object. This function can be used to reconstruct GRNs based on pre-existing GRNs saved in Links object.

Parameters **links_object** (Links) – Please see the explanation of Links class.

get_links(*cluster_name_for_GRN_unit=None*, *alpha=10*, *bagging_number=20*, *verbose_level=1*, *test_mode=False*, *model_method='bagging_ridge'*)
Makes GRN for each cluster and returns results as a Links object. Several preprocessing should be done before using this function.

Parameters

- **cluster_name_for_GRN_unit** (*str*) – Cluster name for GRN calculation. The cluster information should be stored in Oracle.adata.obs.
- **alpha** (*float or int*) – The strength of regularization. If you set a lower value, the sensitivity increases, and you can detect weaker network connections. However, there may be more noise. If you select a higher value, it will reduce the chance of overfitting.
- **bagging_number** (*int*) – The number used in bagging calculation.
- **verbose_level** (*int*) – if [verbose_level>1], most detailed progress information will be shown. if [verbose_level > 0], one progress bar will be shown. if [verbose_level == 0], no progress bar will be shown.
- **test_mode** (*bool*) – If test_mode is True, GRN calculation will be done for only one cluster rather than all clusters.
- **model_method** (*str*) – Chose modeling algorithm. "bagging_ridge" or "bayesian_ridge"

get_mcmc_cell_transition_table(*cluster_column_name=None*, *end=- 1*)
Return cell count in the initial state and final state after mcmc. Cell counts are grouped by the cluster of interest. Result will be returned as 2D matrix.

import_TF_data(*TF_info_matrix=None*, *TF_info_matrix_path=None*, *TFdict=None*)
Load data about potential-regulatory TFs. You can import either TF_info_matrix or TFdict. For more information on how to make these files, please see the motif analysis module within the celloracle tutorial.

Parameters

- **TF_info_matrix** (*pandas.DataFrame*) – TF_info_matrix.

- **TF_info_matrix_path** (*str*) – File path for TF_info_matrix (pandas.DataFrame).
- **TFdict** (*dictionary*) – Python dictionary of TF info.

import_anndata_as_normalized_count (*adata*, *cluster_column_name=None*, *embedding_name=None*, *test_mode=False*)

Load scRNA-seq data. scRNA-seq data should be prepared as an anndata object. Preprocessing (cell and gene filtering, dimensional reduction, clustering, etc.) should be done before loading data. The method will import NORMALIZED and LOG TRANSFORMED data but NOT SCALED and NOT CENTERED data. See the tutorial for more details on how to process scRNA-seq data.

Parameters

- **adata** (*anndata*) – anndata object containing scRNA-seq data.
- **cluster_column_name** (*str*) – the name of column containing cluster information in anndata.obs. Clustering data should be in anndata.obs.
- **embedding_name** (*str*) – the key name for dimensional reduction information in anndata.obsm. Dimensional reduction (or 2D trajectory graph) should be in anndata.obsm.
- **transform** (*str*) – The method for log-transformation. Chose one from "natural_log" or "log2".

import_anndata_as_raw_count (*adata*, *cluster_column_name=None*, *embedding_name=None*, *transform='natural_log'*)

Load scRNA-seq data. scRNA-seq data should be prepared as an anndata object. Preprocessing (cell and gene filtering, dimensional reduction, clustering, etc.) should be done before loading data. The method imports RAW GENE COUNTS because unscaled and uncentered gene expression data are required for the GRN inference and simulation. See tutorial notebook for the details about how to process scRNA-seq data.

Parameters

- **adata** (*anndata*) – anndata object that stores scRNA-seq data.
- **cluster_column_name** (*str*) – the name of column containing cluster information in anndata.obs. Clustering data should be in anndata.obs.
- **embedding_name** (*str*) – the key name for dimensional reduction information in anndata.obsm. Dimensional reduction (or 2D trajectory graph) should be in anndata.obsm.
- **transform** (*str*) – The method for log-transformation. Chose one from "natural_log" or "log2".

plot_mc_result_as_kde (*n_time*, *args={}*)

Pick up one timepoint in the cell state-transition simulation and plot as a kde plot.

Parameters

- **n_time** (*int*) – the number in Markov simulation
- **args** (*dictionary*) – An argument for seaborn.kdeplot. See seaborn documentation for details (https://seaborn.pydata.org/generated/seaborn.kdeplot.html#seaborn.kdeplot).

plot_mc_result_as_trajectory (*cell_name*, *time_range*, *args={}*)

Pick up several timepoints in the cell state-transition simulation and plot as a line plot. This function can be used to visualize how cell-state changes after perturbation focusing on a specific cell.

Parameters

- **cell_name** (*str*) – cell name. chose from adata.obs.index
- **time_range** (*list of int*) – the list of index in Markov simulation

- **args** (*dictionary*) – dictionary for the arguments for matplotlib.pyplit.plot. See matplotlib documentation for details (https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot).

plot_mc_resutls_as_sankey(*cluster_use*, *start=0*, *end=- 1*, *order=None*, *font_size=10*)

Plot the simulated cell state-transition as a Sankey-diagram after groping by the cluster.

Parameters

- **cluster_use** (*str*) – cluster information name in anndata.obs. You can use any cluster information in anndata.obs.
- **start** (*int*) – The starting point of Sankey-diagram. Please select a step in the Markov simulation.
- **end** (*int*) – The end point of Sankey-diagram. Please select a step in the Markov simulation. if you set [end=-1], the final step of Markov simulation will be used.
- **order** (*list of str*) – The order of cluster name in the Sankey-diagram.
- **font_size** (*int*) – Font size for cluster name label in the Sankey diagram.

prepare_markov_simulation(*verbose=False*)

Pick up cells for Markov simulation.

Parameters **verbose** (*bool*) – If True, it plots selected cells.

run_markov_chain_simulation(*n_steps=500*, *n_duplication=5*, *seed=123*, *calculate_randomized=True*)

Do Markov simlations to predict cell transition after perturbation. The transition probability between cells has been calculated based on simulated gene expression values in the signal propagation process. The cell state transition will be simulated based on the probability. You can simulate the process multiple times to get a robust outcome.

Parameters

- **n_steps** (*int*) – steps for Markov simulation. This value is equivalent to the amount of time after perturbation.
- **n_duplication** (*int*) – the number for multiple calculations.

simulate_shift(*perturb_condition=None*, *GRN_unit=None*, *n_propagation=3*, *ignore_warning=False*)

Simulate signal propagation with GRNs. Please see the CellOracle paper for details. This function simulates a gene expression pattern in the near future. Simulated values will be stored in anndata.layers: ["simulated_count"]

The simulation use three types of data. (1) GRN inference results (coef_matrix). (2) Perturb_condition: You can set arbitrary perturbation condition. (3) Gene expression matrix: The simulation starts from imputed gene expression data.

Parameters

- **perturb_condition** (*dictionary*) – condition for perturbation. if you want to simulate knockout for GeneX, please set [perturb_condition={"GeneX": 0.0}] Although you can set any non-negative values for the gene condition, avoid setting biologically infeasible values for the perturb condition. It is strongly recommended to check gene expression values in your data before selecting the perturb condition.
- **GRN_unit** (*str*) – GRN type. Please select either "whole" or "cluster". See the documentation of "fit_GRN_for_simulation" for the detailed explanation.
- **n_propagation** (*int*) – Calculation will be performed iteratively to simulate signal propagation in GRN. You can set the number of steps for this calculation. With a higher

number, the results may recapitulate signal propagation for many genes. However, a higher number of propagation may cause more error/noise.

suggest_mass_thresholds (*n_suggestion=12*, *s=1*, *n_col=4*)

summarize_mc_results_by_cluster (*cluster_use*, *random=False*)
This function summarizes the simulated cell state-transition by groping the results into each cluster. It returns sumarized results as a pandas.DataFrame.

Parameters **cluster_use** (*str*) – cluster information name in anndata.obs. You can use any arbitrary cluster information in anndata.obs.

to_hdf5 (*file_path*)
Save object as hdf5.

Parameters **file_path** (*str*) – file path to save file. Filename needs to end with '.celloracle.oracle'

updateTFinfo_dictionary (*TFdict={}*)
Update a TF dictionary. If a key in the new TF dictionary already exists in the old TF dictionary, old values will be replaced with a new one.

Parameters **TFdict** (*dictionary*) – Python dictionary of TF info.

celloracle.**check_python_requirements** (*return_detail=True*, *print_warning=True*)
Check installation status and requirements of dependant libraries.

celloracle.**load_hdf5** (*file_path*, *object_class_name=None*)
Load an object of celloracle's custom class that was saved as hdf5.

Parameters

- **file_path** (*str*) – file_path.
- **object_class_name** (*str*) – Types of object. If it is None, object class will be identified from the extension of file_name. Default is None.

celloracle.**test_R_libraries_installation** (*show_all_stdout=False*)
CellOracle.network_analysis use several R libraries for network analysis. This is a test function to check for instalation of the necessary R libraries.

Modules for ATAC-seq analysis

celloracle.motif_analysis module

The *motif_analysis* module implements transcription factor motif scan.

Genomic activity information (peak of ATAC-seq or Chip-seq) is extracted first. Then the peak DNA sequence will be subjected to TF motif scan. Finally we will get list of TFs that potentially binds to a specific gene.

class celloracle.motif_analysis.**TFinfo** (*peak_data_frame*, *ref_genome*)
Bases: object

This is a custom class for motif analysis in celloracle. TFinfo object performs motif scan using the TF motif database in gimmemotifs and several functions of genomepy. Analysis results can be exported as a python dictionary or dataframe. These files; python dictionary of dataframe of TF binding information, are needed during GRN inference.

peak_df
dataframe about DNA peak and target gene data.

Type pandas.dataframe

all_target_gene
target genes.
Type array of str

ref_genome
reference genome name that was used in DNA peak generation.
Type str

scanned_df
Results of motif scan. Key is a peak name. Value is a dataframe of motif scan.
Type dictionary

dic_targetgene2TFs
Final product of motif scan. Key is a target gene. Value is a list of regulatory candidate genes.
Type dictionary

dic_peak2Targetgene
Dictionary. Key is a peak name. Value is a list of the target gene.
Type dictionary

dic_TF2targetgenes
Final product of motif scan. Key is a TF. Value is a list of potential target genes of the TF.
Type dictionary

copy()
Deepcoty itself.

filter_motifs_by_score(*threshold*, *method='cumlative_score'*)
Remove motifs with low binding scores.
Parameters **method** (*str*) – thresholding method. Select either of ["individual_score", "cumlative_score"]

filter_peaks(*peaks_to_be_remained*)
Filter peaks.
Parameters **peaks_to_be_remained** (*array of str*) – list of peaks. Peaks that are NOT in the list will be removed.

make_TFinfo_dataframe_and_dictionary(*verbose=True*)
This is the final step of motif_analysis. Convert scanned results into a data frame and dictionaries.
Parameters **verbose** (*bool*) – Whether to show a progress bar.

reset_dictionary_and_df()
Reset TF dictionary and TF dataframe. The following attributes will be erased: TF_onehot, dic_targetgene2TFs, dic_peak2Targetgene, dic_TF2targetgenes.

reset_filtering()
Reset filtering information. You can use this function to stat over the filtering step with new conditions. The following attributes will be erased: TF_onehot, dic_targetgene2TFs, dic_peak2Targetgene, dic_TF2targetgenes.

save_as_parquet(*folder_path=None*)
Save itself. Some attributes are saved as parquet file.
Parameters **folder_path** (*str*) – folder path

scan(*background_length=200*, *fpr=0.02*, *n_cpus=- 1*, *verbose=True*, *motifs=None*, *TF_evidence_level='direct_and_indirect'*, *TF_formatting='auto'*)
Scan DNA sequences searching for TF binding motifs.
Parameters

- **background_length** (*int*) – background length. This is used for the calculation of the binding score.
- **fpr** (*float*) – False positive rate for motif identification.
- **n_cpus** (*int*) – number of CPUs for parallel calculation.
- **verbose** (*bool*) – Whether to show a progress bar.
- **motifs** (*list*) – a list of gimmemotifs motifs, will revert to default_motifs() if None
- **TF_evidence_level** (*str*) – Please select one from ["direct", "direct_and_indirect"]. If "direct" is selected, TFs that have a binding evidence were used. If "direct_and_indirect" is selected, TFs with binding evidence and inferred TFs are used. For more information, please read explanation of Motif class in gimmemotifs documentation (https://gimmemotifs.readthedocs.io/en/master/index.html)

to_dataframe(*verbose=True*)

Return results as a dataframe. Rows are peak_id, and columns are TFs.

Parameters **verbose** (*bool*) – Whether to show a progress bar.

Returns TFinfo matrix.

Return type pandas.dataframe

to_dictionary(*dictionary_type='targetgene2TFs'*, *verbose=True*)

Return TF information as a python dictionary.

Parameters **dictionary_type** (*str*) – Type of dictionary. Select from ["targetgene2TFs", "TF2targetgenes"]. If you chose "targetgene2TFs", it returns a dictionary in which a key is a target gene, and a value is a list of regulatory candidate genes (TFs) of the target. If you chose "TF2targetgenes", it returns a dictionary in which a key is a TF and a value is a list of potential target genes of the TF.

Returns dictionary.

Return type dictionary

to_hdf5(*file_path*)

Save object as hdf5.

Parameters **file_path** (*str*) – file path to save file. Filename needs to end with '.celloracle.tfinfo'

celloracle.motif_analysis.**get_tss_info**(*peak_str_list*, *ref_genome*, *verbose=True*)

Get annotation about Transcription Starting Site (TSS).

Parameters

- **peak_str_list** (*list of str*) – list of peak_id. e.g., ["chr5_0930303_9499409", "chr11_123445555_123445577"]
- **ref_genome** (*str*) – reference genome name.
- **verbose** (*bool*) – verbosity.

celloracle.motif_analysis.**integrate_tss_peak_with_cicero**(*tss_peak*, *cicero_connections*)

Process output of cicero data and returns DNA peak information for motif analysis in celloracle. Please see the celloracle tutorial for more information.

Parameters

- **tss_peak** (*pandas.dataframe*) – dataframe about TSS information. Please use the function, "get_tss_info" to get this dataframe.
- **cicero_connections** (*dataframe*) – dataframe that stores the results of cicero analysis.

Returns DNA peak about promoter/enhancer and its annotation about target gene.

Return type pandas.dataframe

celloracle.motif_analysis.**is_genome_installed**(*ref_genome*)
Celloracle motif_analysis module uses gimmemotifs and genomepy internally. Reference genome files should be installed in the PC to use gimmemotifs and genomepy. This function checks the installation status of the reference genome.

Parameters **ref_genome** (*str*) – names of reference genome. i.e., "mm10", "hg19"

celloracle.motif_analysis.**load_TFinfo**(*file_path*)
Load TFinfo object which was saved as hdf5 file.

Parameters **file_path** (*str*) – file path.

Returns Loaded TFinfo object.

Return type *TFinfo*

celloracle.motif_analysis.**load_TFinfo_from_parquets**(*folder_path*)
Load TFinfo object which was saved with the function; "save_as_parquet".

Parameters **folder_path** (*str*) – folder path

Returns Loaded TFinfo object.

Return type *TFinfo*

celloracle.motif_analysis.**load_motifs**(*motifs_name*)
Load motifs from celloracle motif database

Parameters **motifs_name** (*str*) – Name of motifs.

Returns List of gimmemotifs.motif object.

Return type list

celloracle.motif_analysis.**make_TFinfo_from_scanned_file**(*path_to_raw_bed*, *path_to_scanned_result_bed*, *ref_genome*)
This function is currently an available.

celloracle.motif_analysis.**peak2fasta**(*peak_ids*, *ref_genome*)
Convert peak_id into fasta object.

Parameters

- **peak_id** (*str or list of str*) – Peak_id. e.g. "chr5_0930303_9499409" or it can be a list of peak_id. e.g. ["chr5_0930303_9499409", "chr11_123445555_123445577"]
- **ref_genome** (*str*) – Reference genome name. e.g. "mm9", "mm10", "hg19" etc

Returns DNA sequence in fasta format

Return type gimmemotifs fasta object

celloracle.motif_analysis.**read_bed**(*bed_path*)
Load bed file and return as dataframe.

Parameters **bed_path** (*str*) – File path.

Returns bed file in dataframe.

Return type pandas.dataframe

celloracle.motif_analysis.**remove_zero_seq**(*fasta_object*)
Remove DNA sequence with zero length

celloracle.motif_analysis.**scan_dna_for_motifs**(*scanner_object*, *motifs_object*, *sequence_object*, *verbose=True*)
This is a wrapper function to scan DNA sequences searchig for Gene motifs.

Parameters

- **scanner_object** (*gimmemotifs.scanner*) – Object that do motif scan.
- **motifs_object** (*gimmemotifs.motifs*) – Object that stores motif data.
- **sequence_object** (*gimmemotifs.fasta*) – Object that stores sequence data.

Returns scan results is stored in data frame.

Return type pandas.dataframe

Modules for Network analysis

celloracle.network_analysis module

The *network_analysis* module implements Network analysis.

class celloracle.network_analysis.**Links**(*name*, *links_dict={}*)
Bases: object

This is a class for the processing and visualization of GRNs. Links object stores cluster-specific GRNs and metadata. Please use "get_links" function in Oracle object to generate Links object.

links_dict
Dictionary that store unprocessed network data.
Type dictionary

filtered_links
Dictionary that store filtered network data.
Type dictionary

merged_score
Network scores.
Type pandas.dataframe

cluster
List of cluster name.
Type list of str

name
Name of clustering unit.
Type str

palette
DataFrame that store color information.
Type pandas.dataframe

filter_links(*p=0.001*, *weight='coef_abs'*, *threshold_number=10000*, *genelist_source=None*, *genelist_target=None*, *thread_number=None*)
Filter network edges. In most cases, inferred GRN has non-significant random edges. We have

to remove these edges before analyzing the network structure. You can do the filtering in any of the following ways.
(1) Filter based on the p-value of the network edge. Please enter p-value for thresholding.
(2) Filter based on network edge number. If you set the number, network edges will be filtered based on the order of a network score. The top n-th network edges with network weight will remain, and the other edges will be removed. The network data has several types of network weight, so you have to select which network weight do you want to use.
(3) Filter based on an arbitrary gene list. You can set a gene list for source nodes or target nodes.

Parameters
- **p** (*float*) – threshold for p-value of the network edge.
- **weight** (*str*) – Please select network weight name for the filtering
- **genelist_source** (*list of str*) – gene list to remain in regulatory gene nodes. Default is None.
- **genelist_target** (*list of str*) – gene list to remain in target gene nodes. Default is None.

get_network_entropy(*value='coef_abs'*)
Calculate network entropy scores.
Parameters **value** (*str*) – Default is "coef_abs".

get_score(*test_mode=False*)
Get several network sores using R libraries. Make sure all dependent R libraries are installed in your environment before running this function. You can check the installation for the R libraries by running test_installation() in network_analysis module.

plot_cartography_scatter_per_cluster(*gois=None*, *clusters=None*, *scatter=True*, *kde=False*, *auto_gene_annot=False*, *percentile=98*, *args_dot={'n_levels': 105}*, *args_line={'c': 'gray'}*, *args_annot={}*, *save=None*)
Make a gene network cartography plot. Please read the original paper describing gene network cartography for more information. https://www.nature.com/articles/nature03288
Parameters
- **links** (Links) – See network_analysis.Links class for detail.
- **gois** (*list of srt*) – List of gene name to highlight.
- **clusters** (*list of str*) – List of cluster name to analyze. If None, all clusters in Links object will be analyzed.
- **scatter** (*bool*) – Whether to make a scatter plot.
- **auto_gene_annot** (*bool*) – Whether to pick up genes to make an annotation.
- **percentile** (*float*) – Genes with a network score above the percentile will be shown with annotation. Default is 98.
- **args_dot** (*dictionary*) – Arguments for scatter plot.
- **args_line** (*dictionary*) – Arguments for lines in cartography plot.
- **args_annot** (*dictionary*) – Arguments for annotation in plots.
- **save** (*str*) – Folder path to save plots. If the folder does not exist in the path, the function creates the folder. Plots will not be saved if [save=None]. Default is None.

plot_cartography_term(*goi*, *save=None*, *plt_show=True*)
Plot the gene network cartography term like a heatmap. Please read the original paper of gene network cartography for the principle of gene network cartography. https://www.nature.com/articles/nature03288
Parameters
- **links** (Links) – See network_analysis.Links class for detail.

- **gois** (*list of srt*) – List of gene name to highlight.
- **save** (*str*) – Folder path to save plots. If the folder does not exist in the path, the function creates the folder. Plots will not be saved if [save=None]. Default is None.

plot_degree_distributions(*plot_model=False*, *save=None*)

Plot the network degree distributions (the number of edge per gene). The network degree will be visualized in both linear scale and log scale.

Parameters

- **links** (Links) – See network_analysis.Links class for detail.
- **plot_model** (*bool*) – Whether to plot linear approximation line.
- **save** (*str*) – Folder path to save plots. If the folder does not exist in the path, the function creates the folder. Plots will not be saved if [save=None]. Default is None.

plot_network_entropy_distributions(*update_network_entropy=False*, *save=None*)

Plot the distribution for network entropy. See the CellOracle paper for more detail.

Parameters

- **links** (*Links object*) – See network_analysis.Links class for detail.
- **values** (*list of str*) – The list of score to visualize. If it is None, all network score (listed above) will be used.
- **update_network_entropy** (*bool*) – Whether to recalculate network entropy.
- **save** (*str*) – Folder path to save plots. If the folder does not exist in the path, the function creates the folder. Plots will not be saved if [save=None]. Default is None.

plot_score_comparison_2D(*value*, *cluster1*, *cluster2*, *percentile=99*, *annot_shifts=None*, *save=None*, *plt_show=True*, *interactive=False*)

Make a scatter plot that compares specific network scores in two groups.

Parameters

- **links** (Links) – See network_analysis.Links class for detail.
- **value** (*srt*) – The network score type.
- **cluster1** (*str*) – Cluster name. Network scores in cluster1 will be visualized in the x-axis.
- **cluster2** (*str*) – Cluster name. Network scores in cluster2 will be visualized in the y-axis.
- **percentile** (*float*) – Genes with a network score above the percentile will be shown with annotation. Default is 99.
- **annot_shifts** (*(float, float)*) – Annotation visualization setting.
- **save** (*str*) – Folder path to save plots. If the folder does not exist in the path, the function creates the folder. Plots will not be saved if [save=None]. Default is None.

plot_score_discributions(*values=None*, *method='boxplot'*, *save=None*)

Plot the distribution of network scores. An individual data point is a network edge (gene).

Parameters

- **links** (Links) – See Links class for details.
- **values** (*list of str*) – The list of score to visualize. If it is None, all of the network score will be used.
- **method** (*str*) – Plotting method. Select either "boxplot" or "barplot".
- **save** (*str*) – Folder path to save plots. If the folder does not exist in the path, the function creates the folder. Plots will not be saved if [save=None]. Default is None.

plot_score_per_cluster(*goi*, *save=None*, *plt_show=True*)
Plot network score for a gene. This function visualizes the network score for a specific gene between clusters to get an insight into the dynamics of the gene.

Parameters
- **links** (Links) – See network_analysis.Links class for detail.
- **goi** (*str*) – Gene name.
- **save** (*str*) – Folder path to save plots. If the folder does not exist in the path, the function creates the folder. Plots will not be saved if [save=None]. Default is None.

plot_scores_as_rank(*cluster*, *n_gene=50*, *save=None*)
Pick up top n-th genes wich high-network scores and make plots.

Parameters
- **links** (Links) – See network_analysis.Links class for detail.
- **cluster** (*str*) – Cluster name to analyze.
- **n_gene** (*int*) – Number of genes to plot. Default is 50.
- **save** (*str*) – Folder path to save plots. If the folder does not exist in the path, the function creates the folder. Plots will not be saved if [save=None]. Default is None.

to_hdf5(*file_path*)
Save object as hdf5.

Parameters **file_path** (*str*) – file path to save file. Filename needs to end with '.celloracle.links'

celloracle.network_analysis.**draw_network**(*linkList*, *return_graph=False*)
Plot network graph.

Parameters
- **linkList** (*pandas.DataFrame*) – GRN saved as linkList.
- **return_graph** (*bool*) – Whether to return graph object.

Returns Network X graph objenct.

Return type Graph object

celloracle.network_analysis.**get_R_path**()

celloracle.network_analysis.**get_links**(*oracle_object*, *cluster_name_for_GRN_unit=None*, *alpha=10*, *bagging_number=20*, *verbose_level=1*, *test_mode=False*, *model_method='bagging_ridge'*)
Make GRN for each cluster and returns results as a Links object. Several preprocessing should be done before using this function.

Parameters
- **oracle_object** (Oracle) – See Oracle module for detail.
- **cluster_name_for_GRN_unit** (*str*) – Cluster name for GRN calculation. The cluster information should be stored in Oracle.adata.obs.
- **alpha** (*float or int*) – The strength of regularization. If you set a lower value, the sensitivity increases, and you can detect weaker network connections. However, there may be more noise. If you select a higher value, it will reduce the chance of overfitting.
- **bagging_number** (*int*) – The number used in bagging calculation.

- **verbose_level** (*int*) – if [verbose_level>1], most detailed progress information will be shown. if [verbose_level > 0], one progress bar will be shown. if [verbose_level == 0], no progress bar will be shown.
- **test_mode** (*bool*) – If test_mode is True, GRN calculation will be done for only one cluster rather than all clusters.
- **model_method** (*str*) – Chose modeling algorithm. "bagging_ridge" or "bayesian_ridge"

celloracle.network_analysis.**linkList_to_networkgraph**(*filteredlinkList*)
Convert linkList into Graph object in NetworkX.

Parameters **filteredlinkList** (*pandas.DataFrame*) – GRN saved as linkList.

Returns Network X graph objenct.

Return type Graph object

celloracle.network_analysis.**load_links**(*file_path*)
Load links object saved as a hdf5 file.

Parameters **file_path** (*str*) – file path.

Returns loaded links object.

Return type *Links*

celloracle.network_analysis.**set_R_path**(*R_path*)

celloracle.network_analysis.**test_R_libraries_installation**(*show_all_stdout=False*)
CellOracle.network_analysis use several R libraries for network analysis. This is a test function to check for instalation of the necessary R libraries.

celloracle.network_analysis.**transfer_scores_from_links_to_adata**(*adata*, *links*, *method='median'*)
Transfer the summary of network scores (median or mean) per group from Links object into adata.

Parameters

- **adata** (*anndata*) – anndata
- **links** (Links) – Likns object
- **method** (*str*) – The method to summarize data.

Other modules

celloracle.go_analysis module

The *go_analysis* module implements Gene Ontology analysis. This module use goatools internally.

celloracle.go_analysis.**geneID2Symbol**(*IDs*, *species='mouse'*)
Convert Entrez gene id into gene symbol.

Parameters

- **IDs** (*array of str*) – Entrez gene id.
- **species** (*str*) – Select species. Either "mouse" or "human".

Returns Gene symbol

Return type list of str

celloracle.go_analysis.**geneSymbol2ID**(*symbols*, *species='mouse'*)

Convert gene symbol into Entrez gene id.

Parameters

- **symbols** (*array of str*) – gene symbol
- **species** (*str*) – Select species. Either "mouse" or "human"

Returns Entrez gene id

Return type list of str

celloracle.go_analysis.**get_GO**(*gene_query*, *species='mouse'*)

Get Gene Ontologies (GOs).

Parameters

- **gene_query** (*array of str*) – gene list.
- **species** (*str*) – Select species. Either "mouse" or "human"

Returns GO analysis results as dataframe.

Return type pandas.dataframe

celloracle.utility module

The *utility* module has several functions that support celloracle.

celloracle.utility.**exec_process**(*commands*, *message=True*, *wait_finished=True*, *return_process=True*)

Excute a command. This is a wrapper of "subprocess.Popen"

Parameters

- **commands** (*str*) – command.
- **message** (*bool*) – Whether to return a message or not.
- **wait_finished** (*bool*) – Whether or not to wait for the process to finish. If false, the process will be perfomed in background and the function will finish immediately
- **return_process** (*bool*) – Whether to return "process".

celloracle.utility.**intersect**(*list1*, *list2*)

Intersect two list and get components that exists in both list.

Parameters

- **list1** (*list*) – input list.
- **list2** (*list*) – input list.

Returns intersected list.

Return type list

celloracle.utility.**knn_data_transferer**(*adata_ref*, *adata_que*, *n_neighbors=20*, *cluster_name=None*, *embedding_name=None*, *adata_true=None*, *transfer_color=True*, *n_PCA=30*, *use_PCA_in_adata=False*)

Extract categorical information from adata.obs or embedding information from adata.obsm and transfer these information into query anndata. In the calculation, KNN is used after PCA.

Parameters

- **adata_ref** (*anndata*) – reference anndata
- **adata_que** (*anndata*) – query anndata
- **cluster_name** (*str or list of str*) – cluster name(s) to be transfered. If you want to transfer multiple data, you can set the cluster

names as a list.
- **embedding_name** (*str or list of str*) – embedding name(s) to be transfered. If you want to transfer multiple data, you can set the embedding names as a list.
- **adata_true** (*str*) – This argument can be used for the validataion of this algorithm. If you have true answer, you can set it. If you set true answer, the function will return some metrics for benchmarking.
- **transfer_color** (*bool*) – Whether or not to transfer color data in addition to cluster information.
- **n_PCA** (*int*) – Number of PCs that will be used for the input of KNN algorithm.

celloracle.utility.**load_hdf5**(*file_path*, *object_class_name=None*)
Load an object of celloracle's custom class that was saved as hdf5.
Parameters
- **file_path** (*str*) – file_path.
- **object_class_name** (*str*) – Types of object. If it is None, object class will be identified from the extension of file_name. Default is None.

celloracle.utility.**load_pickled_object**(*filepath*)
Load pickled object.
Parameters **filepath** (*str*) – file path.
Returns loaded object.
Return type python object

class celloracle.utility.**makelog**(*file_name=None*, *directory=None*)
Bases: object

This is a class for making log.

info(*comment*)
Add comment into the log file.
Parameters **comment** (*str*) – comment.

celloracle.utility.**save_as_pickled_object**(*obj*, *filepath*)
Save any object using pickle.
Parameters
- **obj** (*any python object*) – python object.
- **filepath** (*str*) – file path.

celloracle.utility.**standard**(*df*)
Standardize value.
Parameters **df** (*padas.dataframe*) – dataframe.
Returns Data after standardization.
Return type pandas.dataframe

celloracle.utility.**transfer_all_colors_between_anndata**(*adata_ref*, *adata_que*)
Extract all color information from reference anndata and transfer the color into query anndata.
Parameters
- **adata_ref** (*anndata*) – reference anndata
- **adata_que** (*anndata*) – query anndata

celloracle.utility.**transfer_color_between_anndata**(*adata_ref*, *adata_que*, *cluster_name*)
Extract color information from reference anndata and transfer the color into query anndata.

Parameters
- **adata_ref** (*anndata*) – reference anndata
- **adata_que** (*anndata*) – query anndata
- **cluster_name** (*str*) – cluster name. This information should exist in the anndata.obs.

celloracle.utility.**update_adata**(*adata*)

celloracle.data module

The *data* module implements data download and loading.

celloracle.data.**load_Celegans_promoter_base_GRN**(*version='ce10_CisBPv2_fpr2'*)
Load Base GRN made from promoter DNA sequence and motif scan.

Args:
Returns Base GRN as a matrix.
Return type pandas.dataframe

celloracle.data.**load_Scerevisiae_promoter_base_GRN**(*version='sacCer3_CisBPv2_fpr2'*)
Load Base GRN made from promoter DNA sequence and motif scan.

Args:
Returns Base GRN as a matrix.
Return type pandas.dataframe

celloracle.data.**load_TFinfo_df_mm9_mouse_atac_atlas**()
Load Transcription factor binding information made from mouse scATAC-seq atlas dataset. mm9 genome was used for the reference genome.

Args:
Returns TF binding info.
Return type pandas.dataframe

celloracle.data.**load_arabidopsis_promoter_base_GRN**(*version='TAIR10_CisBPv2_fpr2'*)
Load Base GRN made from promoter DNA sequence and motif scan.

Args:
Returns Base GRN as a matrix.
Return type pandas.dataframe

celloracle.data.**load_chicken_promoter_base_GRN**(*version='galGal6_CisBPv2_fpr2'*)
Load Base GRN made from promoter DNA sequence and motif scan.

Args:
Returns Base GRN as a matrix.
Return type pandas.dataframe

celloracle.data.**load_drosophila_promoter_base_GRN**(*version='dm6_CisBPv2_fpr2'*)
Load Base GRN made from promoter DNA sequence and motif scan.

Args:
Returns Base GRN as a matrix.
Return type pandas.dataframe

celloracle.data.**load_human_promoter_base_GRN**(*version='hg19_gimmemotifsv5_fpr2'*)
Load Base GRN made from promoter DNA sequence and motif scan.

Args:
Returns Base GRN as a matrix.

Return type pandas.dataframe

celloracle.data.**load_mouse_promoter_base_GRN**(*version='mm10_gimmemotifsv5_fpr2'*)
Load Base GRN made from promoter DNA sequence and motif scan.

Args:
Returns Base GRN as a matrix.
Return type pandas.dataframe

celloracle.data.**load_rat_promoter_base_GRN**(*version='rn6_gimmemotifsv5_fpr2'*)
Load Base GRN made from promoter DNA sequence and motif scan.

Args:
Returns Base GRN as a matrix.
Return type pandas.dataframe

celloracle.data.**load_tutorial_links_object**()

celloracle.data.**load_tutorial_oracle_object**()

celloracle.data.**load_xenopus_tropicalis_promoter_base_GRN**(*version='xenTro3_CisBPv2_fpr2'*)
Load Base GRN made from promoter DNA sequence and motif scan.

Args:
Returns Base GRN as a matrix.
Return type pandas.dataframe

celloracle.data.**load_zebrafish_promoter_base_GRN**(*version='danRer11_CisBPv2_fpr2'*)
Load Base GRN made from promoter DNA sequence and motif scan.

Args:
Returns Base GRN as a matrix.
Return type pandas.dataframe

celloracle.data_conversion module

The *data_conversion* module implements data conversion between different platform.

celloracle.data_conversion.**seurat_object_to_anndata**(*file_path_seurat_object*, *delete_tmp_file=True*)
Convert seurat object into anndata.
Parameters
- **file_path_seurat_object** (*str*) – File path of seurat object. Seurat object should be saved as Rds format.
- **delete_tmp_file** (*bool*) – Whether to delete temporary file.

Returns anndata object.
Return type anndata

2.4 Changelog

- *0.5.1 <2020-08-4>*

  - Add new promoter-TSS reference data for several reference genomes; (1)"Xenopus": ["xenTro2", "xenTro3"],n(2)"Rat": ["rn4", "rn5", "rn6"], (3)"Drosophila": ["dm3", "dm6"], (4)"C.elegans": ["ce6", "ce10"], (5)"Arabidopsis": ["tair10"].

  - Add new motif data for several species: "Xenopus", "Rat", "Drosophila", "C.elegans" and "Arabidopsis".

- *0.5.0 <2020-08-3>*

  - Add now functions for custom motifs. You can select motifs from several options. Also, we updated our web tutorial to introduce how to load / make a different motif data.

  - Change default motifs for S.cerevisiae and Zebrafish.

  - Change requirements for dependent package: gimmemotifs and geomepy. Celloracle codes were updated to support new version of gimmemotifs (0.14.4) and genomepy (0.8.4).

- *0.4.2 <2020-07-14>*

  - Add promoter-TSS information for Zebrafish reference genome (danRer7, danRer10 and danRer11).

- *0.4.1 <2020-07-02>*

  - Add promoter-TSS information for S.cerevisiae reference genome (sacCer2 and sacCer3).

- *0.4.0 <2020-06-28>*

  - Change requirements.

  - From this version, pandas version 1.0.3 or later is required.

  - From this version, scanpy version 1.5.3 or later is required.

- *0.3.7 <2020-06-12>*

  - Delete GO function from r-script

  - Update some functions for network visualization

- *0.3.6 <2020-06-08>*

  - Fix a bug on the transition probability calculation in Markov simulation

  - Add new function "count_cells_in_mc_results" to oracle class

- *0.3.5 <2020-05-09>*

  - Fix a bug on the function for gene cortography visualization

  - Change some settings for installation

  - Update data conversion module

- *0.3.4 <2020-04-29>*

  - Change pandas version restriction

  - Fix a bug on the function for gene cortography visualization

  - Add new functions for R-path configuration

- *0.3.3 <2020-04-24>*

  - Add promoter-TSS information for hg19 and hg38 reference genome

- *0.3.1 <2020-03-23>*

  - Fix an error when try to save file larger than 4GB file

- *0.3.0 <2020-2-17>*

  - Release beta version

2.5 License

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential

damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

2.6 Authors and citations

2.6.1 Cite celloracle

If you use celloracle please cite our bioarxiv preprint CellOracle: Dissecting cell identity via network inference and in silico gene perturbation.

2.6.2 celloracle software development

celloracle is developed and maintained by Kenji Kamimoto and members of Samantha Morris Lab. Please post troubles or questions on the Github repository.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

N

O

P

R

S

T

U