
celloracle

Release 0.1.0

Sep 26, 2019

CONTENTS

1	Contents	3
1.1	Installation	3
1.2	Tutorial	5
1.3	API	81
1.4	Changelog	101
1.5	License	101
1.6	Authors and citations	101
2	Indices and tables	103
	Python Module Index	105
	Index	107

CellOracle is a python library for the analysis of Gene Regulatory Network with single cell data.

Source code and tutorials are available at [celloracle git hub repository](#)

Note:

For colleagues in Morris Lab

Thank you for your kind help!!

Documentation is also available as a word and pdf file.

[pdf version](#)

[microsoft word version](#)

Could you please add comments on the file and send it back to Kenji?

I really appreciate your support!

Thanks,

Kenji

(This message will be deleted when we launch celloracle.)

Warning: CellOracle is still under development. It is alpha version and functions in this package may change in the future release.

**CHAPTER
ONE**

CONTENTS

1.1 Installation

celloracle uses several python libraries and R library. Please follow this guide below to install the dependent software of celloracle.

1.1.1 Python Requirements

- celloracle was developed with python 3.6. We do not support python 2.7x or python <=3.5.
- celloracle was developed in Linux and macOS. We do not guarantee that celloracle works in Windows OS.
- We highly recommend using anaconda to setup python environment.
- Please install all dependent libraries before installing celloracle according to the instructions below.
- celloracle is still beta version and it is not available through PyPI or anaconda distribution yet. Please install celloracle from GitHub repository according to the instruction below.

0. (Optional) Make a new environment

This step is optional. Please make a new python environment for celloracle and install dependent libraries in it if you get some software conflicts.

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

1. Add conda channels

Installation of some libraries below 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 velocyo

Please install velocyo with the following commands or [the author's instruction](#). On Mac OS, you may have a compile error during velocyo installation. I recommend you installing Xcode in that case.

```
conda install numpy scipy cython numba matplotlib scikit-learn h5py click
```

Then

```
pip install velocyto
```

3. Install scanpy

Please install scanpy with the following commands or [the author's instruction](#) .

```
conda install seaborn scikit-learn statsmodels numba pytables python-igraph louvain
```

Then

```
pip install scanpy
```

4. Install gimmermotifs

Please install gimmermotifs with the following commands or [the author's instruction](#) .

```
conda install gimmermotifs genomepy=0.5.5
```

5. Install other python libraries

Please install other python libraries below with the following commands.

```
conda install goatools pyarrow tqdm joblib jupyter
```

6. install celloracle from github

```
pip install git+https://github.com/KenjiKamimoto-wustl122/CellOracle
```

1.1.2 R requirements

celloracle use R library for the network analysis and scATAC-seq analysis. Please install [R \(>=3.5\)](#) and R libraries below according to the author's instruction.

Seurat

Please install Seurat with the following r-script or [the author's instruction](#) . celloracle is compatible with both Seurat V2 and V3. If you use only scanpy for the scRNA-seq preprocessing and do not use Seurat , you can skip installation of Seurat.

in R console,

```
install.packages('Seurat')
```

Cicero

Please install Cicero with the following r-script or [the author's instruction](#) . If you have no plan for scATAC-seq analysis and just want to use celloracle with a default TF information which was supplied with celloracle, you can skip installation of Cicero.

in R console,

```
if (!requireNamespace("BiocManager", quietly = TRUE))
  install.packages("BiocManager")
BiocManager::install("cicero", version = "3.8")
```

igraph

Please install `igraph` with the following r-script or [the author's instruction](#).

in R console,

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

linkcomm

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

in R console,

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

rnetcarto

`rnetcarto` installation has to be done with several steps. Please install `rnetcarto` with [the author's instruction](#). You need to install [the GNU Scientific Libraries](#) before installing `rnetcarto`. Detailed instruction can be found [here](#).

Check installation

These R libraries above are necessary for the network analysis in `celloracle`. You can check installation using `celloracle`'s function.

in python console,

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

Please make sure that all R libraries are installed. The following message will be shown when all R libraries are appropriately installed.

```
checking R library installation: gProfileR -> OK
checking R library installation: igraph -> OK
checking R library installation: linkcomm -> OK
checking R library installation: rnetcarto -> OK
```

1.2 Tutorial

Analysis of `celloracle` proceeds through multiple steps below. Please run the notebooks following the order. If you do not have ATAC-seq data and want to use default genome and TF binding information, you can skip the first and second step and start from the third step. Please refer to the `celloracle` paper for detailed information.

You can download all notebooks from [here](#).

1.2.1 ATAC-seq data preprocessing

Process scATAC-seq data (or bulk ATAC-seq data) to get open accessible promoter/enhancer DNA sequence. If you have a scATAC-seq data, proximal and distal cis-regulatory elements from TSS site can be picked up using [Cicero](#). If you have bulk ATAC-seq data, open accessible DNA elements around TSS site will be picked up.

A. Extract TF binding information from scATAC-seq data

If you want to use scATAC-seq data, you can start from Cicero analysis to get information of distal cis-regulatory elements. This step use Cicero and do not use celloracle. Please refer to [the documentation of Cicero](#) for the detailed usage.

R notebook

0. Import library

```
[2]: # install cicero before running this script
rm(list=ls())
library(cicero)
```

1. Prepare data

In this tutorial we use scAtac-seq data in 10x genomics database. You do not need to download these data when you analyze your scATAC data.

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

# download demo dataset from 10x genomics
system("wget -O data/matrix.tar.gz 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
        ↪")

# unzip data
system("tar -xvf data/matrix.tar.gz -C data")
```

```
[6]: # Just replace the data path below with the data path of your scATAC data if you have
      ↪your scATAC data.

data_folder <- "data/filtered_peak_bc_matrix"
output_folder <- "cicero_output"
dir.create(output_folder)
```

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

2.1. Process data to make CDS object

```
[7]: # 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"

# 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="_")
```

(continues on next page)

(continued from previous page)

```

row.names(peakinfo) <- peakinfo$site_name

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

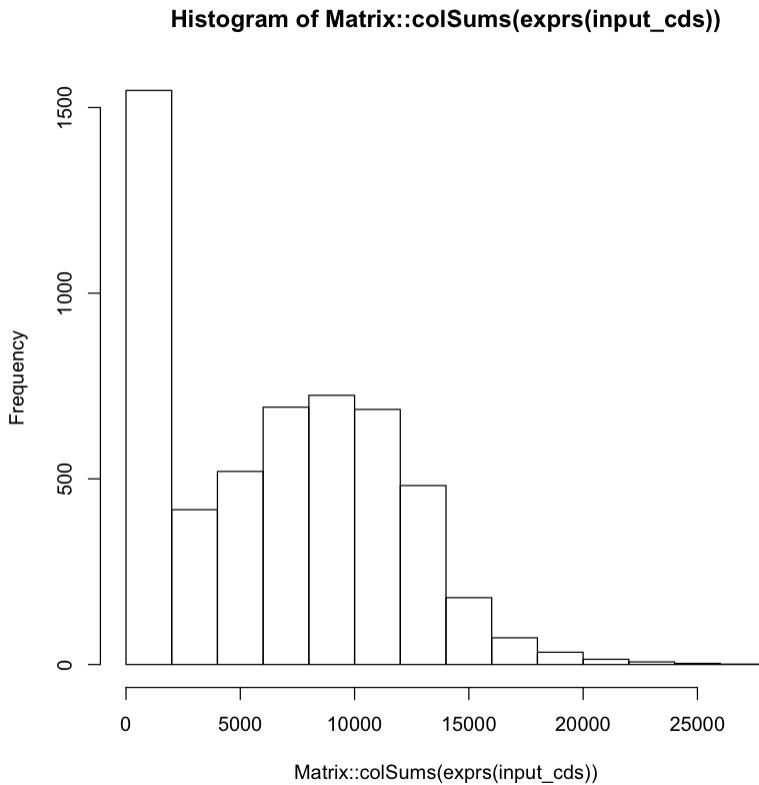
# make CDS
input_cds <- suppressWarnings(newCellDataSet(indata,
                                              phenoData = methods::new("AnnotatedDataFrame", data =_
                                              ↪cellinfo),
                                              featureData = methods::new("AnnotatedDataFrame", data =_
                                              ↪peakinfo),
                                              expressionFamily=VGAM::binomialff(),
                                              lowerDetectionLimit=0))
input_cds@expressionFamily@vfamily <- "binomialff"
input_cds <- monocle::detectGenes(input_cds)

#Ensure there are no peaks included with zero reads
input_cds <- input_cds[Matrix::rowSums(exprs(input_cds)) >= 100,]

```

3. Quality check and Filtering

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



[9]: # filter cells by peak_count
`max_count <- 15000`

(continues on next page)

(continued from previous page)

```
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

```
[10]: # Run cicero to get cis-regulatory networks
set.seed(2017)
input_cds <- detectGenes(input_cds)
input_cds <- estimateSizeFactors(input_cds)

input_cds <- reduceDimension(input_cds, max_components = 2, verbose=T, scaling = FALSE,
                                relative_expr=FALSE,
                                reduction_method = 'tSNE', norm_method = "none")

tsne_coords <- t(reducedDimA(input_cds))
row.names(tsne_coords) <- row.names(pData(input_cds))
cicero_cds <- make_cicero_cds(input_cds, reduced_coordinates = tsne_coords)

saveRDS(cicero_cds, paste0(output_folder, "/cicero_cds.Rds"))

Remove noise by PCA ...
Reduce dimension by tSNE ...
Overlap QC metrics:
Cells per bin: 50
Maximum shared cells bin-bin: 44
Mean shared cells bin-bin: 0.76256263875674
Median shared cells bin-bin: 0
```

5. Run cicero to get cis-regulatory connection scores

```
[11]: # import genome length, which is needed for the function, run_cicero
mm10_chromosome_length <- read.table("./mm10_chromosome_length.txt")

# run the main function
conns <- run_cicero(cicero_cds, mm10_chromosome_length) # Takes a few minutes to run

# save results
saveRDS(conns, paste0(output_folder, "/cicero_connections.Rds"))

# check results
head(conns)

[1] "Starting Cicero"
[1] "Calculating distance_parameter value"
[1] "Running models"
[1] "Assembling connections"
[1] "Done"
```

	Peak1 <fct>	Peak2 <fct>	coaccess <dbl>
A data.frame: 6 x 3	2 chr1_3094484_3095479	chr1_3113499_3113979	-0.316289004
	3 chr1_3094484_3095479	chr1_3119478_3121690	-0.419240532
	4 chr1_3094484_3095479	chr1_3399730_3400368	-0.050867246
	5 chr1_3113499_3113979	chr1_3094484_3095479	-0.316289004
	7 chr1_3113499_3113979	chr1_3119478_3121690	0.370342744
	8 chr1_3113499_3113979	chr1_3399730_3400368	-0.009276026

6. Save results for next step

```
[ ]: 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"))
```

Next, the results of Cicero analysis will be processed with celloracle to make TSS annotations.

Python notebook

In this notebook, we use the results of cicero analysis to get active promoter/enhancer DNA peaks. Cicero provides us information about the connection between peaks. First, we pick up peaks around 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. The final product includes TSS peaks and peaks that have a strong connection with TSS peaks. We use this information as an active promoter/enhancer elements.

0. Import libraries

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

import seaborn as sns
#import time

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

%config InlineBackend.figure_format = 'retina'

plt.rcParams['figure.figsize'] = (15, 7)
plt.rcParams["savefig.dpi"] = 200
```

```
[2]: from celloracle import motif_analysis as ma
```

1. Load data made with cicero

```
[3]: # load all peaks
peaks = pd.read_csv("cicero_output/all_peaks.csv", index_col=0)
peaks = peaks.x.values
peaks
```

```
[3]: array(['chr1_3094484_3095479', 'chr1_3113499_3113979',
       'chr1_3119478_3121690', ..., 'chrY_90804622_90805450',
       'chrY_90808626_90809117', 'chrY_90810560_90811167'], dtype=object)
```

```
[4]: # load cicero results
cicero_connections = pd.read_csv("cicero_output/cicero_connections.csv", index_col=0)
cicero_connections.head()

/home/k/anaconda3/envs/test/lib/python3.6/site-packages/numpy/lib/arraysetops.py:568:_
→FutureWarning: elementwise comparison failed; returning scalar instead, but in the_
→future will perform elementwise comparison
    mask |= (ar1 == a)
```

```
[4]:      Peak1          Peak2  coaccess
2  chr1_3094484_3095479  chr1_3113499_3113979 -0.316289
3  chr1_3094484_3095479  chr1_3119478_3121690 -0.419241
4  chr1_3094484_3095479  chr1_3399730_3400368 -0.050867
5  chr1_3113499_3113979  chr1_3094484_3095479 -0.316289
7  chr1_3113499_3113979  chr1_3119478_3121690  0.370343
```

2. Make TSS annotation

!! Please make sure that you are setting correct ref genomes.

```
[5]: tss_annotated = ma.get_tss_info(peak_str_list=peaks, ref_genome="mm10")
```

```
# check results
tss_annotated.tail()

que bed peaks: 72402
tss peaks in que: 16987
```

```
[5]:      chr      start        end gene_short_name strand
16982  chr1    55130650  55132118        Mob4      +
16983  chr6    94499875  94500767        Slc25a26      +
16984  chr19   45659222  45660823        Fbxw4      -
16985  chr12   100898848 100899597        Gpr68      -
16986  chr4    129491262 129492047        Fam229a     -
```

3. Integrate TSS info and cicero connections

The output file after integrartion process have three columns; peak_id, gene_short_name, coaccess. “peak_id” is either of TSS peak or peaks that have cicero connection between TSS peak. “gene_short_name” is gene name of TSS site. “coaccess” is a coacces score between TSS peak. If the score is 1, it means that the peak_id is TSS itself.

```
[8]: integrated = ma.integrate_tss_peak_with_cicero(tss_peak=tss_annotated,
                                                 cicero_connections=cicero_connections)
```

```
print(integrated.shape)
```

```
integrated.head()
```

```
(263279, 3)
```

```
[8]:      peak_id gene_short_name  coaccess
0  chr10_100015291_100017830        Kitl  1.000000
1  chr10_100018677_100020384        Kitl  0.086299
2  chr10_100050858_100051762        Kitl  0.034558
3  chr10_100052829_100053395        Kitl  0.167188
4  chr10_100128086_100128882      Tmtc3  0.022341
```

4. Filter peaks

Remove peaks that have weak coaccess score.

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

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

(15680, 2)
```

```
[10]:          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.

```
[11]: peak.to_parquet("peak_file.parquet")
```

-> go to next notebook

B. Extract TF binding information from bulk ATAC-seq data or Chip-seq data

Instead of scATAC-seq data bulk DNA-seq data can be used.

Python notebook

0. Import libraries

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

import seaborn as sns
#import time

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

%config InlineBackend.figure_format = 'retina'

plt.rcParams['figure.figsize'] = (15,7)
plt.rcParams["savefig.dpi"] = 200
```

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

1. Load bed file

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

```
[3]: file_path_of_bed_file = "data/all_peaks.bed"
```

```
[4]: # load bed_file
bed = ma.read_bed(file_path_of_bed_file)
print(bed.shape)
bed.head()

(436206, 4)

[4]:   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
```

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

```
[6]: 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

!! Please make sure that you are setting correct ref genomes.

```
[7]: 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
```

```
[7]:      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      +
```

```
[9]: # 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)
```

```
[9]:          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_parquet("peak_file.parquet")
```

-> go to next notebook

1.2.2 Transcription factor binding motif scan

Promoter/enhancer DNA sequence will be subjected to TF-binding motif scan to get the list of potential regulatory genes for each target gene.

Python notebook

0. Import libraries

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

import seaborn as sns
#import time

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

%config InlineBackend.figure_format = 'retina'

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

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

1. Load data

load annotated peak data.

```
[4]: peaks = pd.read_parquet("../01_ATAC-seq_data_processing/option1_scATAC-seq_data_
                           _analysis_with_cicero/peak_file.parquet")
peaks.head()
```

	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

2. Check data

```
[5]: # check data
print(f"number of peak: {len(peaks.peak_id.unique())}")
mean_ = len(peaks.groupby(["gene_short_name", "peak_id"]).count()) / len(peaks.gene_
short_name.unique())
print(f"mean peaks per gene: {mean_}")

def getLength(x):
    a, b, c = x["peak_id"].split("_")
    return int(c) - int(b)

#
df = peaks.apply(lambda x: getLength(x), axis=1)
print(f"mean peak length: {df.values.mean()}")

number of peak: 13919
mean peaks per gene: 1.0591731964333964
mean peak length: 1756.1744260204082
```

2.1. Remove short peaks

Short peak cannot be used for motif scan. Remove short DNA fragment.

```
[6]: peaks = peaks[df>=5]
```

3. Instantiate TFInfo object and search for TF binding motifs

In celloracle motif analysis, we use custom class; TFinfo. This class convert peak data into get DNA sequence and scan the DNA sequence searching for TF binding motifs. Then the results of motif scan will be filtered and converted into several files, such as python dictionary or dataframe. These TF information is necessary for celloracle's GRN inference.

3.1 check reference genome installation

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

ma.is_genome_installed(ref_genome=ref_genome)

genome mm10 is not installed in this environment.
Please install genome using genomepy.
e.g.
    >> import genomepy
    >> genomepy.install_genome("mm9", "UCSC")
```

[7]: False

3.2. Install reference genome (if refgenome is not installed)

```
[9]: import genomepy
genomepy.install_genome(ref_genome, "UCSC")

downloading from http://hgdownload.soe.ucsc.edu/goldenPath/mm10/bigZips/chromFa.tar.
→gz...
done...
name: mm10
```

(continues on next page)

(continued from previous page)

```
local name: mm10
fasta: /home/k/.local/share/genomes/mm10/mm10.fa
```

```
[9]: # check again
ma.is_genome_installed(ref_genome=ref_genome)
```

```
[9]: True
```

```
[14]: # Instantiate TFinfo object
tfi = ma.TFinfo(peak_data_frame=peaks, # peak info calculated from ATAC-seq data
                 ref_genome=ref_genome)
```

4. Scan motifs and save object

This step requires computational resource and may take long time

```
[15]: %%time
# Scan motifs
tfi.scan(fpr=0.02, verbose=True)

# save tfinfo object
tfi.to_hdf5(file_path="test.celloracle.tfinfo")

initiating scanner ...
2019-09-22 23:00:18,604 - INFO - Using background: genome mm10 with length 200
2019-09-22 23:00:18,986 - INFO - Determining FPR-based threshold

getting DNA sequences ...
scanning motifs ...

HBox(children=(IntProgress(value=1, bar_style='info', max=1), HTML(value='')))

CPU times: user 52min 23s, sys: 36.8 s, total: 53min
Wall time: 52min 58s
```

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

	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		
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.

5. Filtering motifs

```
[17]: # reset filtering
tfi.reset_filtering()

# do filtering
tfi.filter_motifs_by_score(threshold=10.5)

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

peaks were filtered: 12934005 -> 2285279
1. converting scanned results into one-hot encoded dataframe.

HBox(children=(IntProgress(value=0, max=13919), HTML(value='')))

2. converting results into dictionaries.
converting scan results into dictionaries...

HBox(children=(IntProgress(value=0, max=14804), HTML(value='')))

HBox(children=(IntProgress(value=0, max=1090), HTML(value='')))
```

6. Get Final results

6.1. Get results as a dictionary

```
[18]: td = tfi.to_dictionary(dictionary_type="targetgene2TFs")
```

6.2. Get results as a dataframe

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

[20]:
peak_id gene_short_name 9430076c15rik Ac002126.6 \
0 chr10_100015291_100017830 Kitl 0 0
1 chr10_100486534_100488209 Tmtc3 0 0
2 chr10_100588641_100589556 4930430F08Rik 0 0
3 chr10_100741247_100742505 Gm35722 0 0
4 chr10_101681379_101682124 Mgat4c 0 0

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

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

(continues on next page)

(continued from previous page)

```
[5 rows x 1092 columns]
```

7. Save TFinfo as dictionary and dataframe

we'll use this information to make GRN. Save this results.

```
[21]: folder = "TFinfo_outputs"
os.makedirs(folder, exist_ok=True)

# save TFinfo as a dictionary
td = tfi.to_dictionary(dictionary_type="targetgene2TFs")
save_as_pickled_object(td, os.path.join(folder, "TFinfo_targetgene2TFs.pickled"))

# save TFinfo as a dataframe
df = tfi.to_dataframe()
df.to_parquet(os.path.join(folder, "TFinfo_dataframe.parquet"))
```

```
[ ]:
```

1.2.3 Single-cell RNA-seq data preprocessing

Network analysis and simulation in celloracle require processed scRNA-seq data. We recommend processing scRNA-seq data using either scanpy or Seurat. If you want to know general idea of scRNA-seq data processing, please go to [the documentation of scanpy](#) and [the documentation of Seurat](#). This step needs to include data quality check, filtering, normalization, dimensional reduction, clustering and cluster annotation. If you have preprocessed scRNA-seq data, such as anndata or seurat object, you can skip this part.

Note: several tips

A. scRNA-seq data preprocessing with scanpy

scanpy is a python library for the analysis of scRNA-seq data.

In this tutorial, we use the almost same code as [one of scanpy tutorials](#) with some modification.

Python notebook

0. Import libraries

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

[2]: %matplotlib inline
%config InlineBackend.figure_format = 'retina'
plt.rcParams["savefig.dpi"] = 300
plt.rcParams["figure.figsize"] = [12, 6]
```

1. Load data

In this notebook, we show an example of scRNA-seq preprocessing 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>). This data can be download using a scanpy's function.

Please change the code below if you want to use your 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. Variable gene detection

In celloracle we will make GRN and do simulation using many genes. Removing weakly-variable gene genes will reduce calculation time. We recomend using top 2000~3000 variable genes.

```
[6]: # select top2000 highly-variable genes
filter_result = sc.pp.filter_genes_dispersion(adata.X,
                                              flavor='cell_ranger',
                                              n_top_genes=2000,
                                              log=False)

# 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

Log transformation and scaling are necessary for many single cell analysis. But we still use non-transformed gene expression data in the celloracle analysis. We keep raw count in anndata with following command.

```
[7]: # keep raw cont data before log transformation
adata.raw = adata
```

(continues on next page)

(continued from previous page)

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

[8]: # (optional) Regressing out some values help reduce batch effects.

#sc.pp.regress_out(adata, ['n_counts', 'percent_mito'])
#sc.pp.regress_out(adata, ['n_counts'])
```

6. Dimensional reduction

Dimensional reduction is one of the most important parts in the scRNA-seq analysis. In celloracle, we use dimensional reduction embeddings for the simulation of cell transition. We need to select an algorithm for dimensional reduction so that cell-cell relationship is properly represented. We recommend using one of the dimensional reduction algorithms (or trajectory inference algorithms); UMAP, tSNE, diffusion map, Force-directed graph drawing, PAGA.

In this example, we use a combination of diffusion, force-directed graph drawing, and PAGA.

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

[10]: # diffusion map
sc.pp.neighbors(adata, n_neighbors=4, n_pcs=20)

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

7. Clustering

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

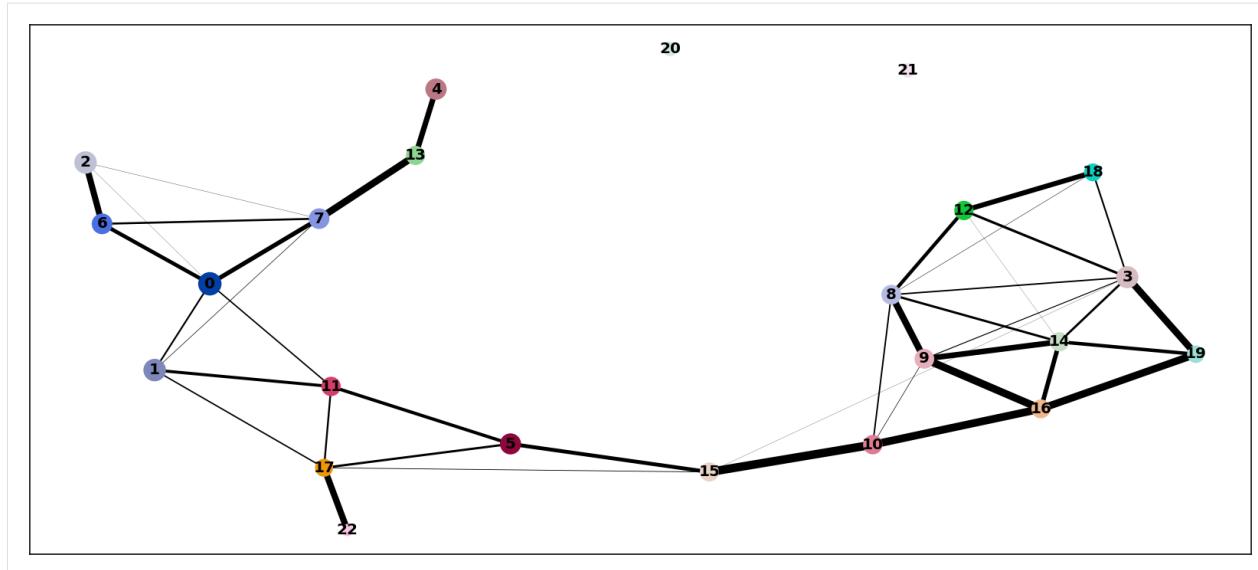
(Optional) Re-calculate Dimensional reduction graph

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

[13]: # check current cluster name
cluster_list = adata.obs.louvain.unique()
cluster_list

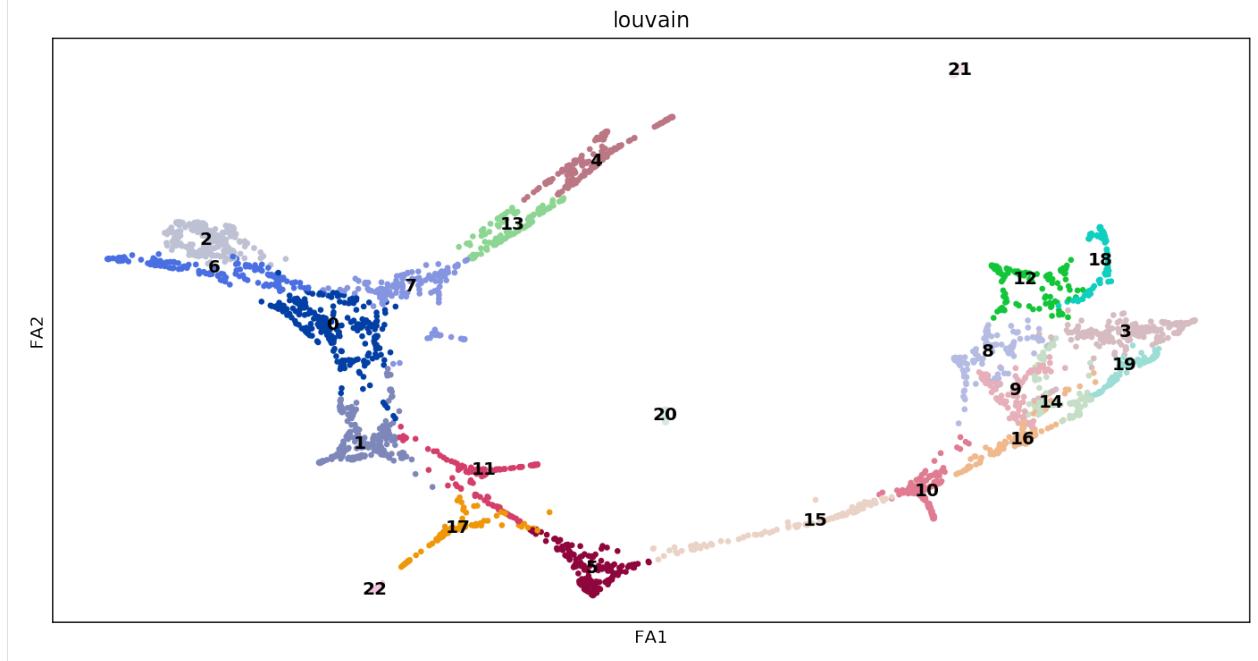
[13]: [5, 2, 12, 13, 0, ..., 6, 20, 14, 15, 21]
Length: 23
Categories (23, object): [5, 2, 12, 13, ..., 20, 14, 15, 21]

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



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

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



8. Check data

8.1. Visualize marker gene expression

```
[17]: markers = {"Erythroids": ["Gata1", "Klf1", "Gypa", "Hba-a2"],
               "Megakaryocytes": ["Itga2b", "Pbx1", "Sdpr", "Vwf"],
               "Granulocytes": ["Elane", "Cebpe", "Ctsg", "Mpo", "Gfil1"],
```

(continues on next page)

(continued from previous page)

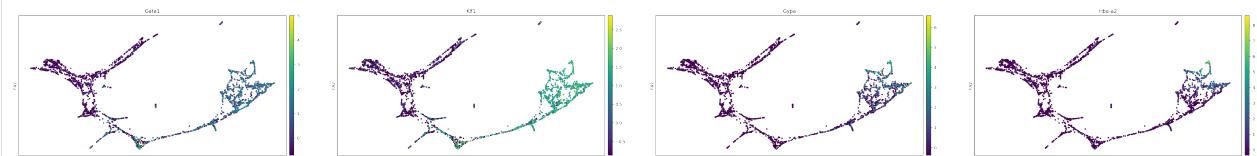
```

    "Monocytes": ["Irf8", "Csf1r", "Ctsg", "Mpo"],
    "Mast_cells": ["Cma1", "Gzmb", "Kit"],
    "Basophils": ["Mcpt8", "Prss34"]
}

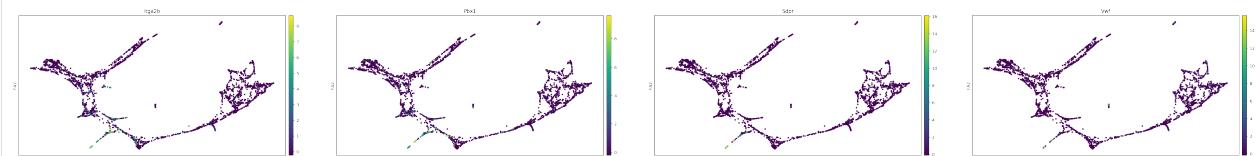
for cell_type, genes in markers.items():
    print(f"marker gene of {cell_type}")
    sc.pl.draw_graph(adata, color=genes, use_raw=False)
    plt.show()

```

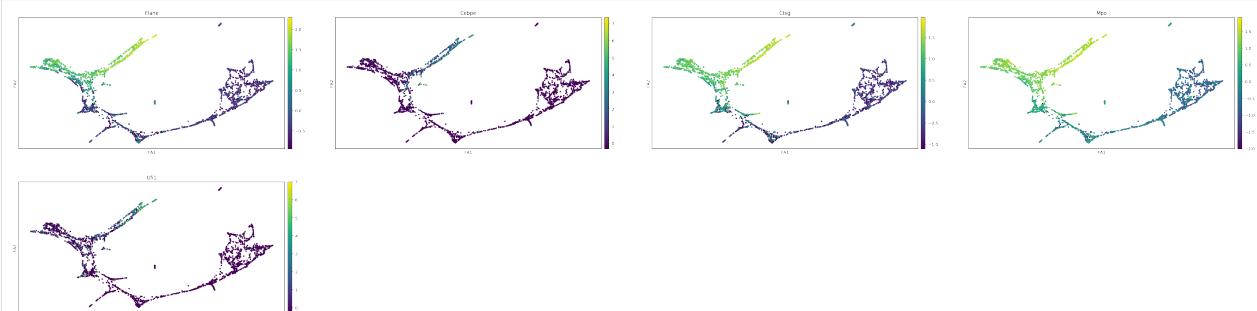
marker gene of Erythroids



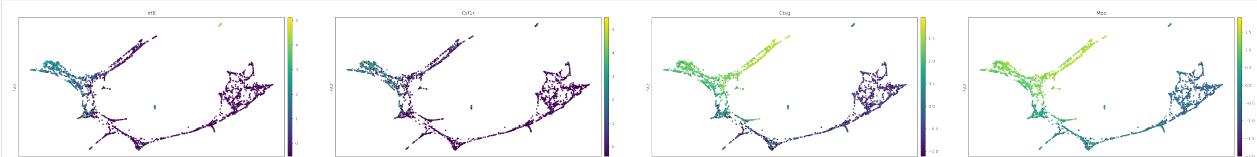
marker gene of Megakaryocytes



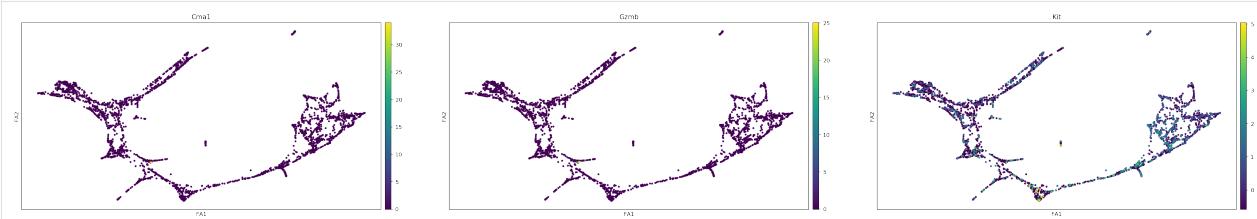
marker gene of Granulocytes



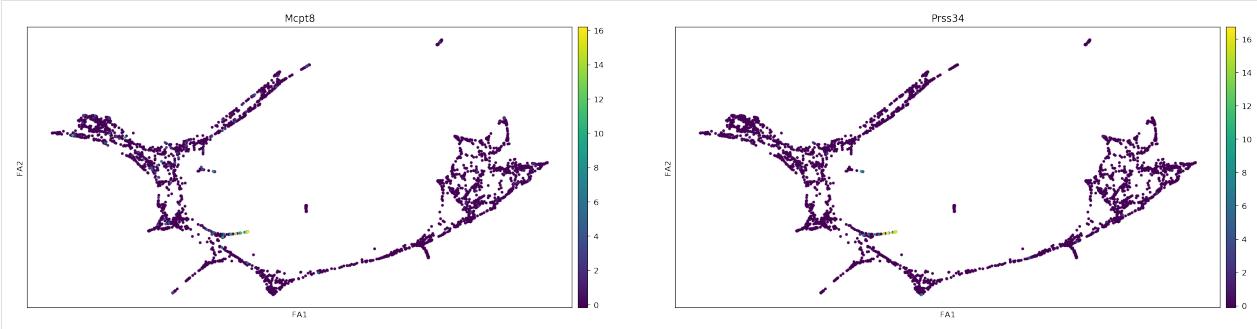
marker gene of Monocytes



marker gene of Mast_cells



marker gene of Basophils

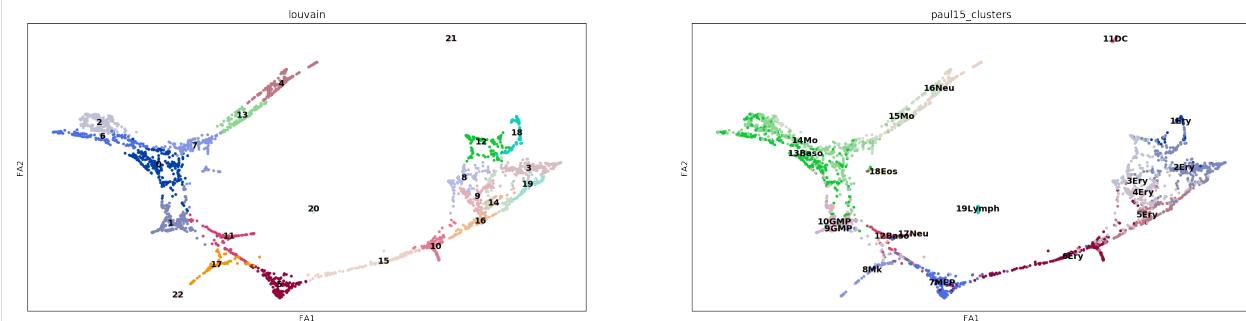


8. Make annotation for cluster

Based on the marker gene expression and previous report, we will manually make annotation for each cluster.

8.1. Make annotation (1)

```
[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 annotation 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:
```

(continues on next page)

(continued from previous page)

```

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'}

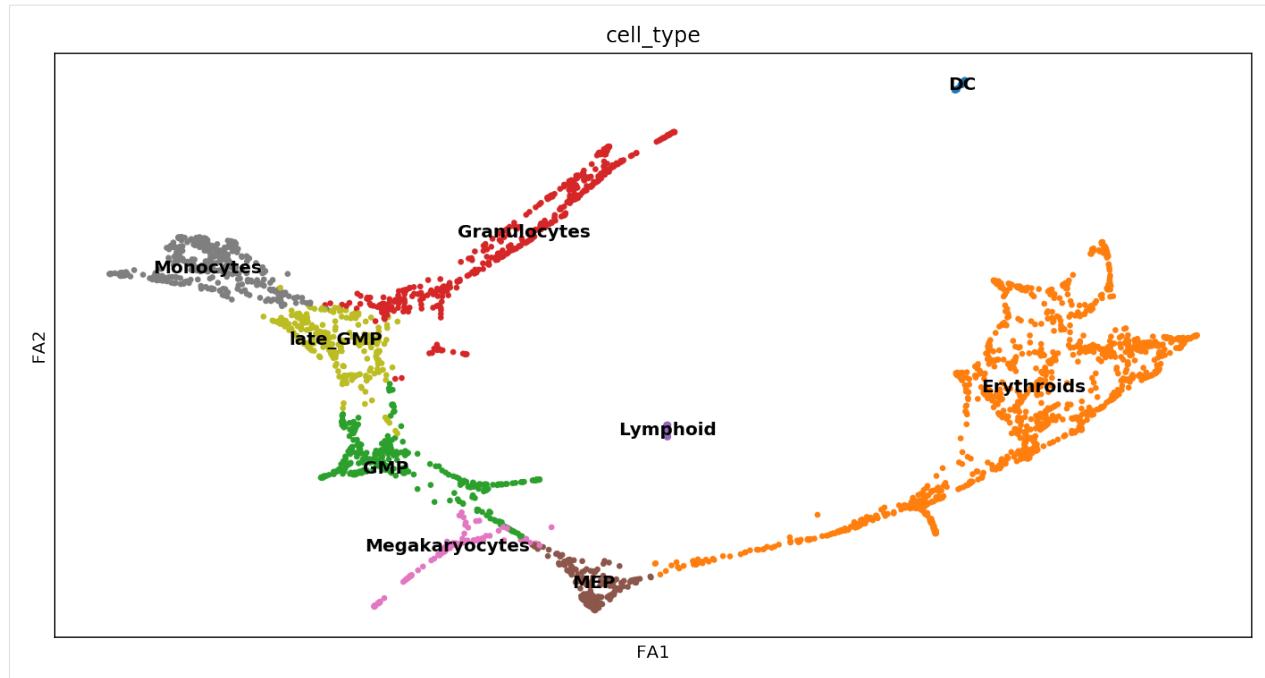
```

```
[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'],
                 legend_loc='on data')
... storing 'cell_type' as categorical

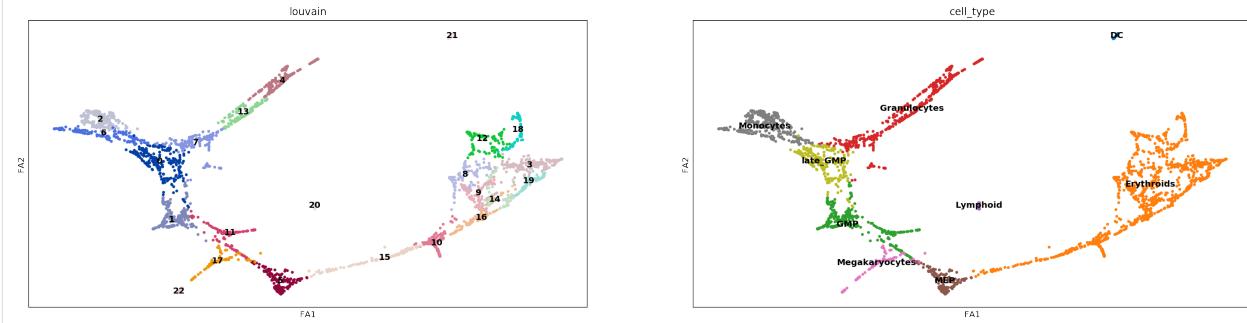
```



8.2. Make annotation (2)

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',}
```

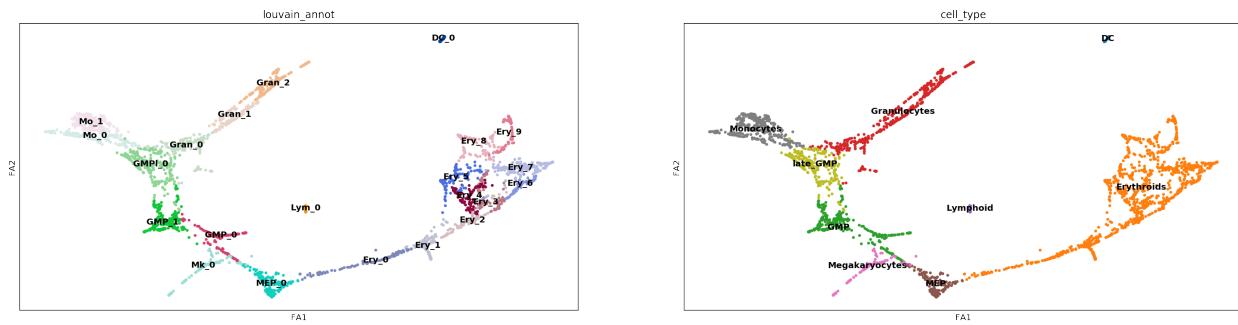
(continues on next page)

(continued from previous page)

```
'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')
... storing 'louvain_annot' as categorical



We've done several scRNA-preprocessing steps; filtering, normalization, clustering, dimensional reduction. In the next step, we'll do GRN inference, network analysis, and in silico simulation based on these information.

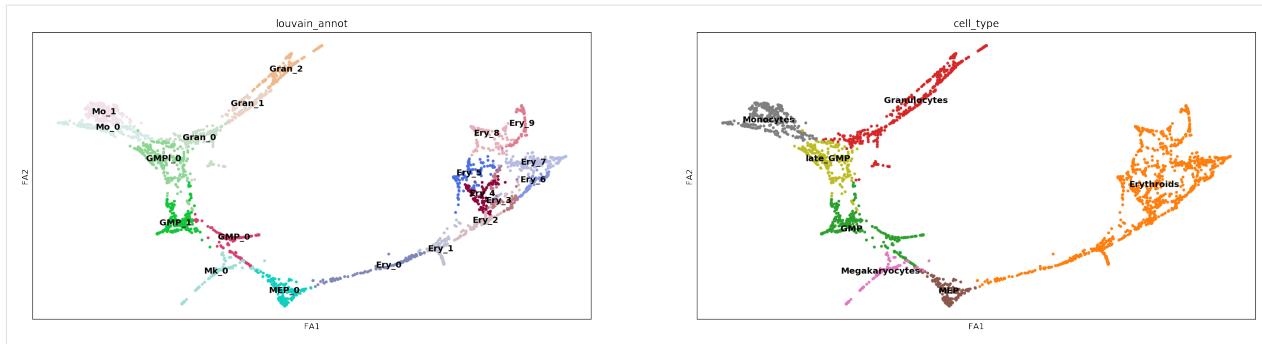
9. (Option) Select cell cluster

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 remove non-myeloid cell cluster, DC and Lymphoid now.

[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')



9. Save data

```
[32]: adata.write_h5ad("data/Paul_et al_15.h5ad")
```

```
[ ]:
```

B. scRNA-seq data preprocessing with Seurat

R notebook ... comming soon

1.2.4 Network analysis

celloracle reconstructs cluster-specific GRNs using scRNA-seq dataset and TF binding information which was made in the previous steps. Then several network analysis methods will be applied to the cluster specific GRNs. This step includes network structure analysis, network score analysis, and key regulatory gene identification.

Python notebook

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
from tqdm import tqdm_notebook as tqdm
```

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

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

plt.rcParams["figure.figsize"] = [9, 6]
plt.rcParams["savefig.dpi"] = 600
```

0.1. Check installation

Celloracle use some R libraries in network analysis. Check installation.

```
[4]: co.network_analysis.test_R_libraries_installation()
checking R library installation: igraph -> OK
checking R library installation: linkcomm -> OK
checking R library installation: rnetcarto -> OK
```

0.2. Make a folder to save graph

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

1. Load data

1.1. Load processed gene expression data (anndata)

Please refer to the previous notebook in the tutorial for an example of how to process scRNA-seq data.

```
[12]: # load data. !!Replace the data path below when you use another data.
adata = sc.read_h5ad("../03_scRNA-seq_data_preprocessing/data/Paul_etal_15.h5ad")
```

1.2. Load TF data.

celloracle need TF information, which contains lists of regulatory candidate gene. There are several way to make such TF information. TF information can be made from scATAC-seq data or Bulk ATAC-seq data. Please refer to the first step of the tutorial.

If you do not have your scATAC-seq data, you can use some builtin data in celloracle. For example, celloracle have a TFinfo which was made with various kind of tissue/cell-types in mouse ATAC-seq atlas dataset (<http://atlas.gs.washington.edu/mouse-atac/>).

This TFinfo is ready. You can load and use it with following command.

```
[13]: # Load TF info which was made from mouse cell atlas dataset.
TFinfo_df = co.data.load_TFinfo_df_mm9_mouse_atac_atlas()

# check data
TFinfo_df.head()
```

	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				

(continues on next page)

(continued from previous page)

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

Basically all information will be stored and processed inside Oracle object. Processed gene expression data (anndata) and TFinfo are necessary to initiate Oracle object.

```
[14]: # make Oracle object
oracle = co.Oracle()
```

2.1. load gene expression data into oracle object.

```
[15]: # The anndata shoud include (1) gene expression count, (2) clustering information,
#       (3) trajectory (dimensional reduction embeddings) data.
# Please refer to another notebook for the detail of anndata preprocessing.

# In this notebook, we use raw mRNA count as an input of Oracle object.
adata.X = adata.raw.X.copy()

# Instantiate Oracle object. All calculation will be done in this object.
oracle.import_anndata_as_raw_count(adata=adata,
                                    cluster_column_name="louvain_annot",
                                    embedding_name="X_draw_graph_fa",
                                    transform="natural_log")
```

2.2. load TFinfo into oracle object

```
[16]: # If you can load TF info dataframe with the following code.
oracle.import_TF_data(TF_info_matrix=TFinfo_df)

# Instead of the way above, you can use TF "dictionary" with the following code.
# oracle.import_TF_data(TFdict=TFinfo_dictionary)
```

2.3. (Optional) Add TF info manually

We mainly use TF info data made from scATAC-seq data. Besides we can also add any information about TF-target gene pair manually.

For example if there is a study or database about TF-target pair, you can use such information by the following way.

2.3.1. Make TF info dictionary manually

Here, we introduce most simple way to add TF binding information using TF binding data in the supplemental table 4 in (<http://doi.org/10.1016/j.cell.2015.11.013>).

```
[17]: # 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
```

	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, Acs16, Add2, Ank...
4	Sfpil	0910001L09Rik, 2310014H01Rik, 4632428N05Rik, A...

```
[18]: # Make dictionary: Key is TF, Value is list of Target gene
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 inverse the dictionary above using a utility function in celloracle.
TG_to_TF_dictionary = co.utility.inverse_dictionary(TF_to_TG_dictionary)
```

HBox(children=(IntProgress(value=0, max=178), HTML(value='')))

2.3.2. Add TF information dictionary into the oracle object

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

3. Knn imputation

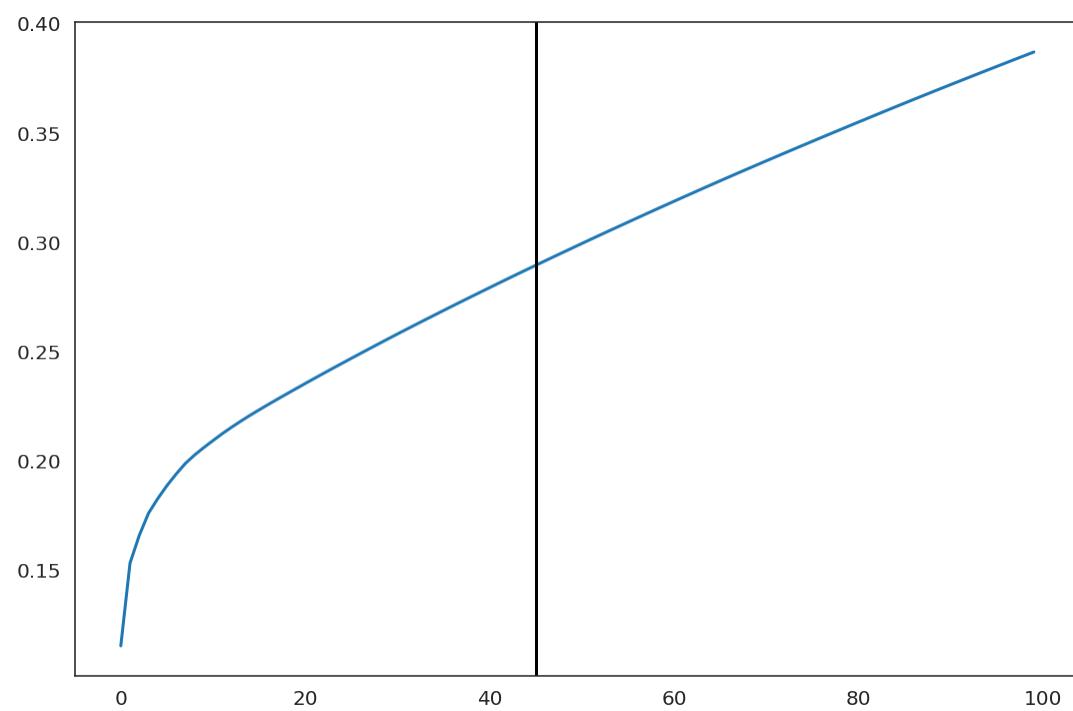
Celloracle uses almost the same strategy as velocity for visualization of trajectory, which requires knn imputation to improves the accuracy of cell transition calculation. Besides, PCA and PC selection are required for knn imputation.

3.1. PCA

```
[85]: # 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.
                           ↪0.002)) [0] [0]
plt.axvline(n_comps, c="k")
print(n_comps)
n_comps = min(n_comps, 50)

45
```



3.2. Knn imputation

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

cell number is :2671
```

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

Auto-selected k is :66
```

```
[23]: 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.

Celloracle has some custom-class; Links, Oracle, TFinfo, etc., You can save such object using “to_hdf5”. The file will be saved as a hdf5 file.

To load saved file, please use “load_hdf5” function.

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

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

4. GRN calculation

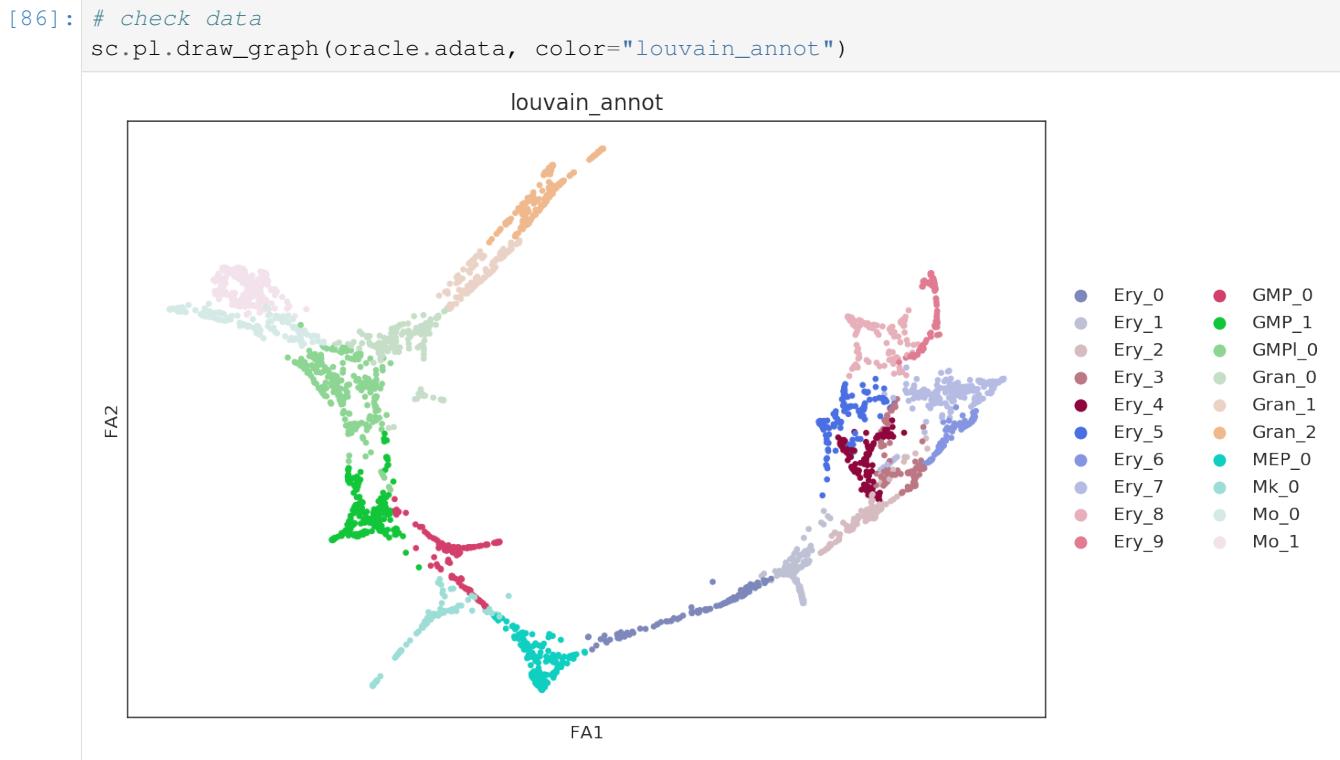
Next step is constructing a cluster-specific GRN for all clusters.

GRN can be calculated at any arbitrary unit.

With “get_links” function, you can calculate cluster-specific GRNs at any unit, and the function returns GRNs as Links object. Links object stores inferred GRNs and its metadata. You can do network analysis with Links object.

In the example below, we will make GRNs based on “louvain_annot” clustering unit.

You can use any clustering unit for GRN calculation as long as clustering information is saved in anndata.



[28]:

```
# calculate GRN for each population in "louvain_annot" clustering unit.
# This step may take long time.
links = oracle.get_links(cluster_name_for_GRN_unit="louvain_annot", alpha=10,
                         verbose_level=2, test_mode=False)
```

HBox(children=(IntProgress(value=0, max=20), HTML(value='')))

inferring GRN for Ery_0...
method: bagging_ridge
alpha: 10

HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))

inferring GRN for Ery_1...
method: bagging_ridge
alpha: 10

HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))

inferring GRN for Ery_2...
method: bagging_ridge
alpha: 10

```
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))
```

```
inferring GRN for Ery_3...
method: bagging_ridge
alpha: 10
```

```
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))
```

```
inferring GRN for Ery_4...
method: bagging_ridge
alpha: 10
```

```
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))
```

```
inferring GRN for Ery_5...
method: bagging_ridge
alpha: 10
```

```
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))
```

```
inferring GRN for Ery_6...
method: bagging_ridge
alpha: 10
```

```
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))
```

```
inferring GRN for Ery_7...
method: bagging_ridge
alpha: 10
```

```
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))
```

```
inferring GRN for Ery_8...
method: bagging_ridge
alpha: 10
```

```
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))
```

```
inferring GRN for Ery_9...
method: bagging_ridge
alpha: 10
```

```
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))
```

```
inferring GRN for GMP_0...
method: bagging_ridge
alpha: 10
```

```
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))
```

```
inferring GRN for GMP_1...
method: bagging_ridge
alpha: 10
```

```
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))
```

```

inferring GRN for GMPl_0...
method: bagging_ridge
alpha: 10
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))

inferring GRN for Gran_0...
method: bagging_ridge
alpha: 10
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))

inferring GRN for Gran_1...
method: bagging_ridge
alpha: 10
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))

inferring GRN for Gran_2...
method: bagging_ridge
alpha: 10
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))

inferring GRN for MEP_0...
method: bagging_ridge
alpha: 10
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))

inferring GRN for Mk_0...
method: bagging_ridge
alpha: 10
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))

inferring GRN for Mo_0...
method: bagging_ridge
alpha: 10
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))

inferring GRN for Mo_1...
method: bagging_ridge
alpha: 10
HBox(children=(IntProgress(value=0, max=1850), HTML(value='')))
```

Links object have some metadata.

```
[33]: links.palette
[33]:      palette
Ery_0   #7D87B9
```

(continues on next page)

(continued from previous page)

Ery_1	#BEC1D4
Ery_2	#D6BCC0
Ery_3	#BB7784
Ery_4	#8E063B
Ery_5	#4A6FE3
Ery_6	#8595E1
Ery_7	#B5BBE3
Ery_8	#E6AFB9
Ery_9	#E07B91
GMP_0	#D33F6A
GMP_1	#11C638
GMP1_0	#8DD593
Gran_0	#C6DEC7
Gran_1	#EAD3C6
Gran_2	#F0B98D
MEP_0	#0FCFC0
Mk_0	#9CDED6
Mo_0	#D5EAE7
Mo_1	#F3E1EB

For the visualization below, we change the order of clusters.

[35]:

```
# For the visualization below, we define and save order of clusters.
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',
         'GMP1_0', 'Mo_0', 'Mo_1', 'Gran_0', 'Gran_1', 'Gran_2']
links_louvain.palette = links_louvain.palette.loc[order]
links_louvain.palette
```

[35]:

palette	
MEP_0	#0FCFC0
Mk_0	#9CDED6
Ery_0	#7D87B9
Ery_1	#BEC1D4
Ery_2	#D6BCC0
Ery_3	#BB7784
Ery_4	#8E063B
Ery_5	#4A6FE3
Ery_6	#8595E1
Ery_7	#B5BBE3
Ery_8	#E6AFB9
Ery_9	#E07B91
GMP_0	#D33F6A
GMP_1	#11C638
GMP1_0	#8DD593
Mo_0	#D5EAE7
Mo_1	#F3E1EB
Gran_0	#C6DEC7
Gran_1	#EAD3C6
Gran_2	#F0B98D

5. Network preprocessing

5.1. Filter network edges

Celloracle leverages bagging ridge or Bayesian ridge regression for network inference. These methods provide a network edge strength as a distribution rather than point value. We can use the distribution of the network edge strength to know the certainness of the connection.

We will do filtering of network edge before starting network analysis.

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

The raw network data is stored as an attribute, “links_dict,” while filtered network data will be stored in “filtered_links.” So the raw network data will not be erased after the filtering. You can come back to the filtering process and do the filtering again if you do not like filtering results.

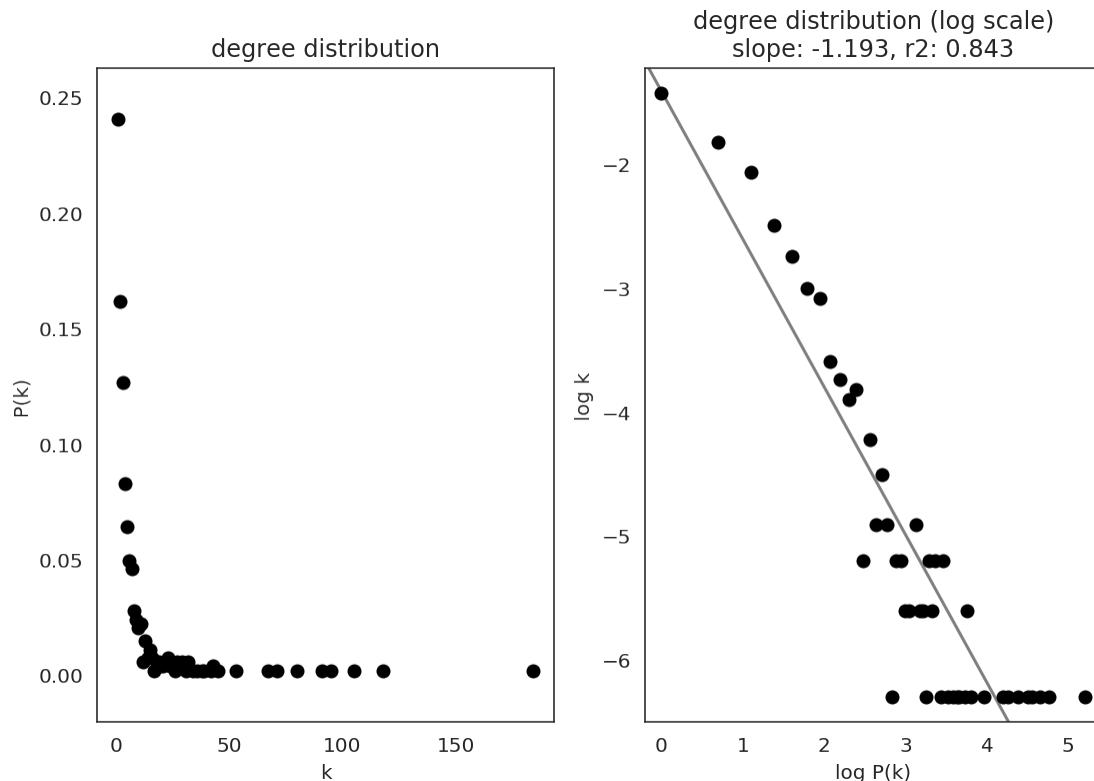
```
[32]: links.filter_links(p=0.001, weight="coef_abs", thread_number=2000)
```

5.2. Degree distribution

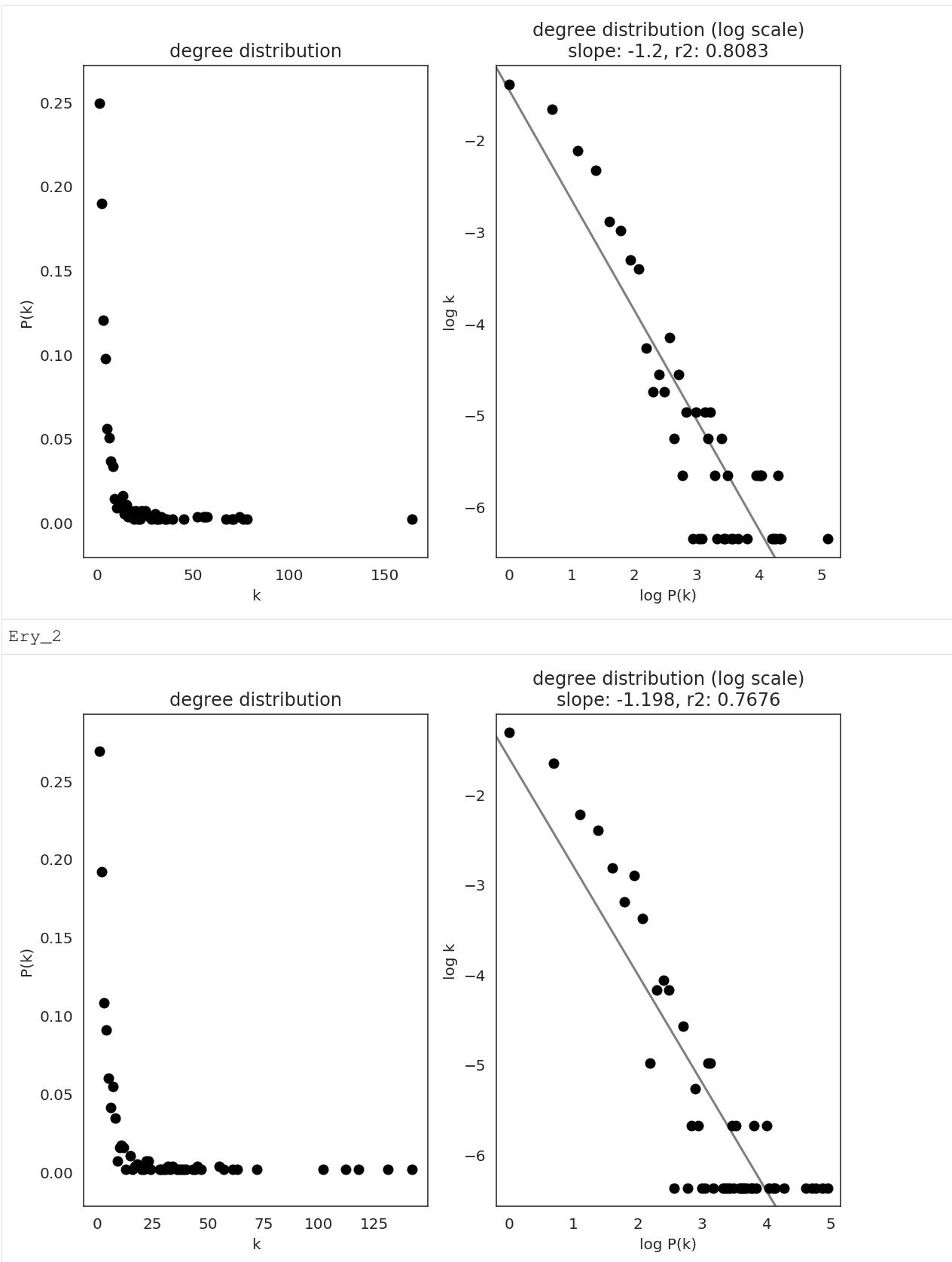
Network degree, which means the number of edges for each gene, is one of the important metrics to know the network analysis. The degree distribution may change depending on the filtering threshold.

```
[87]: links.plot_degree_distributions(plot_model=True, save=f"{save_folder}/degree_distribution/")
```

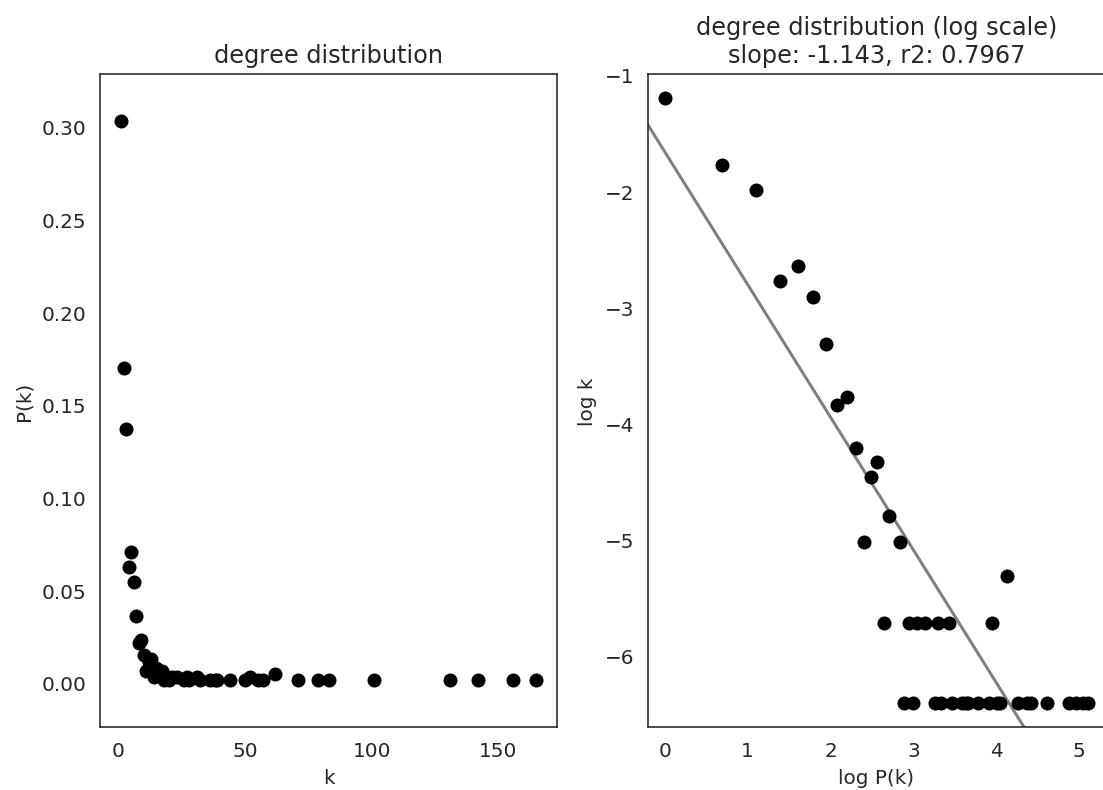
Ery_0



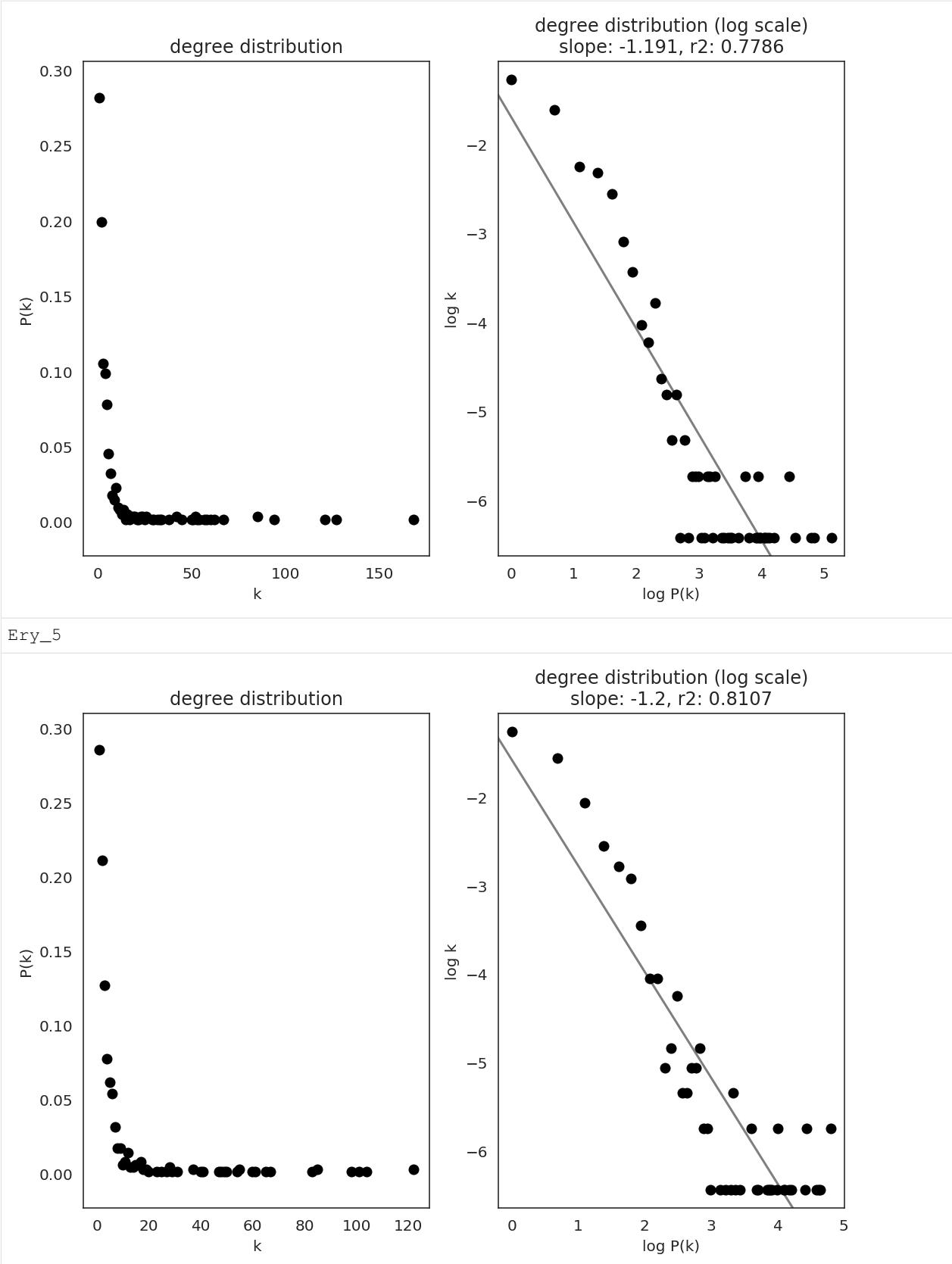
Ery_1



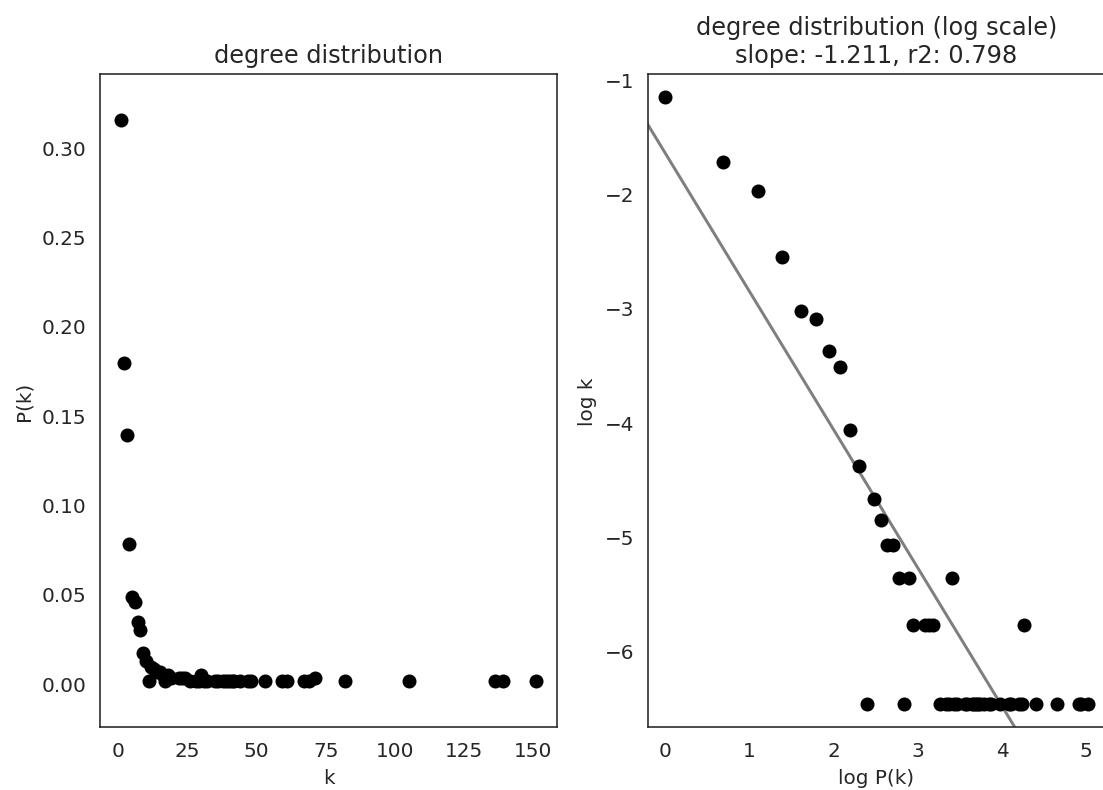
Ery_3



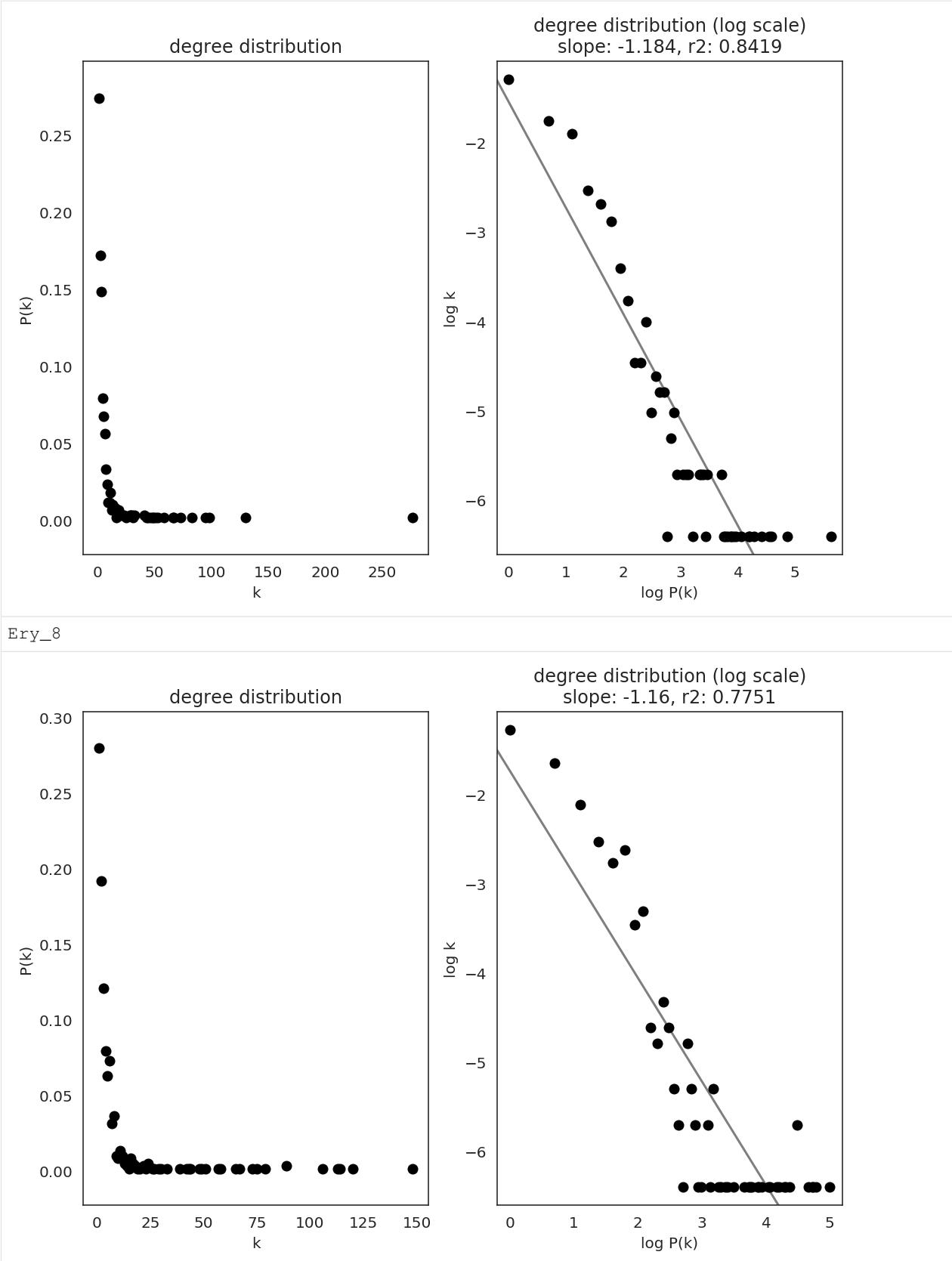
Ery_4



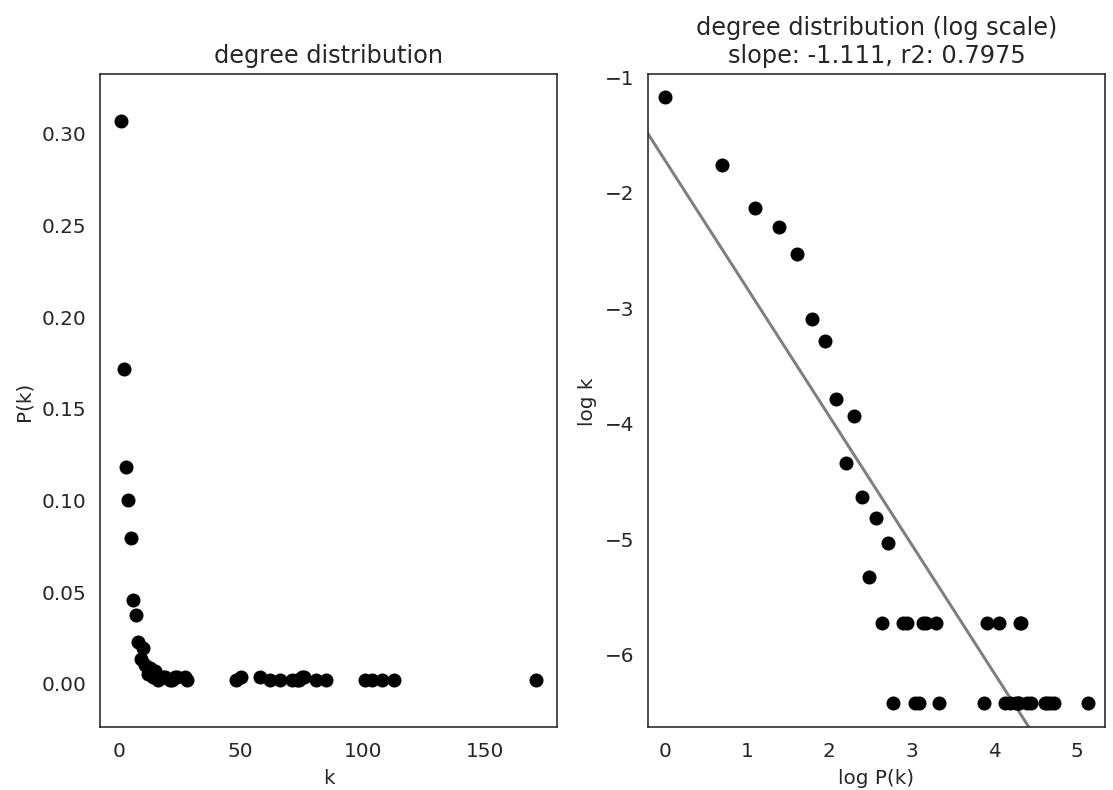
Ery_6

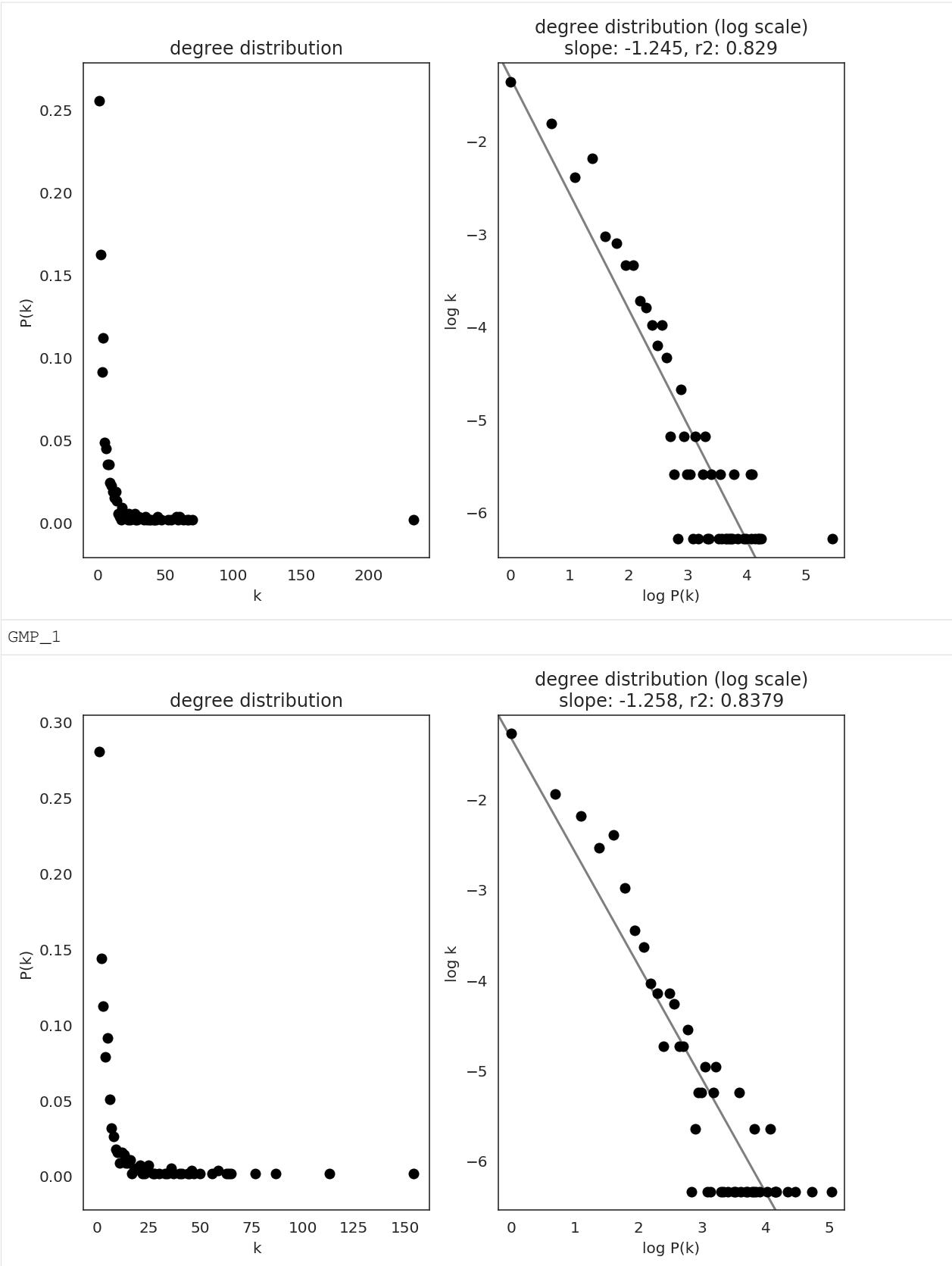


Ery_7

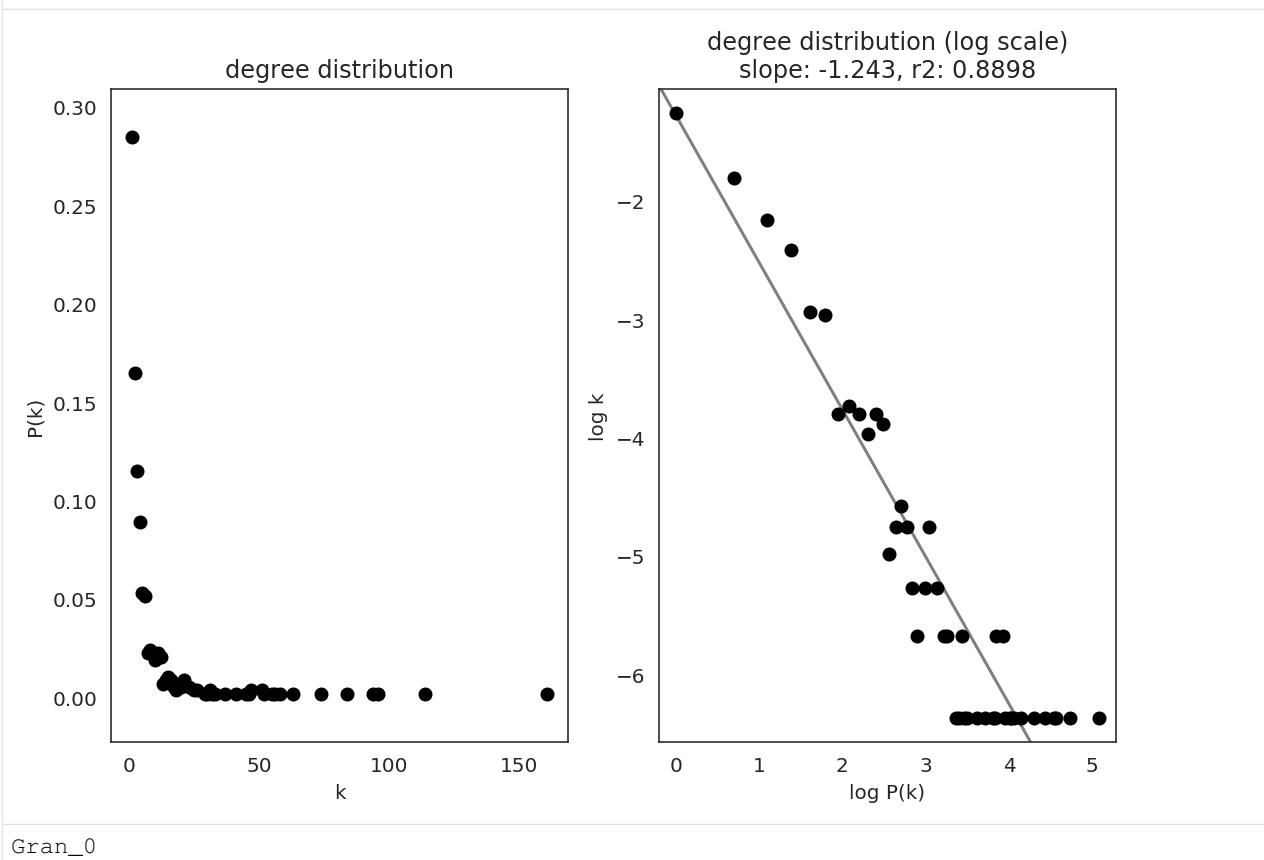


Ery_9

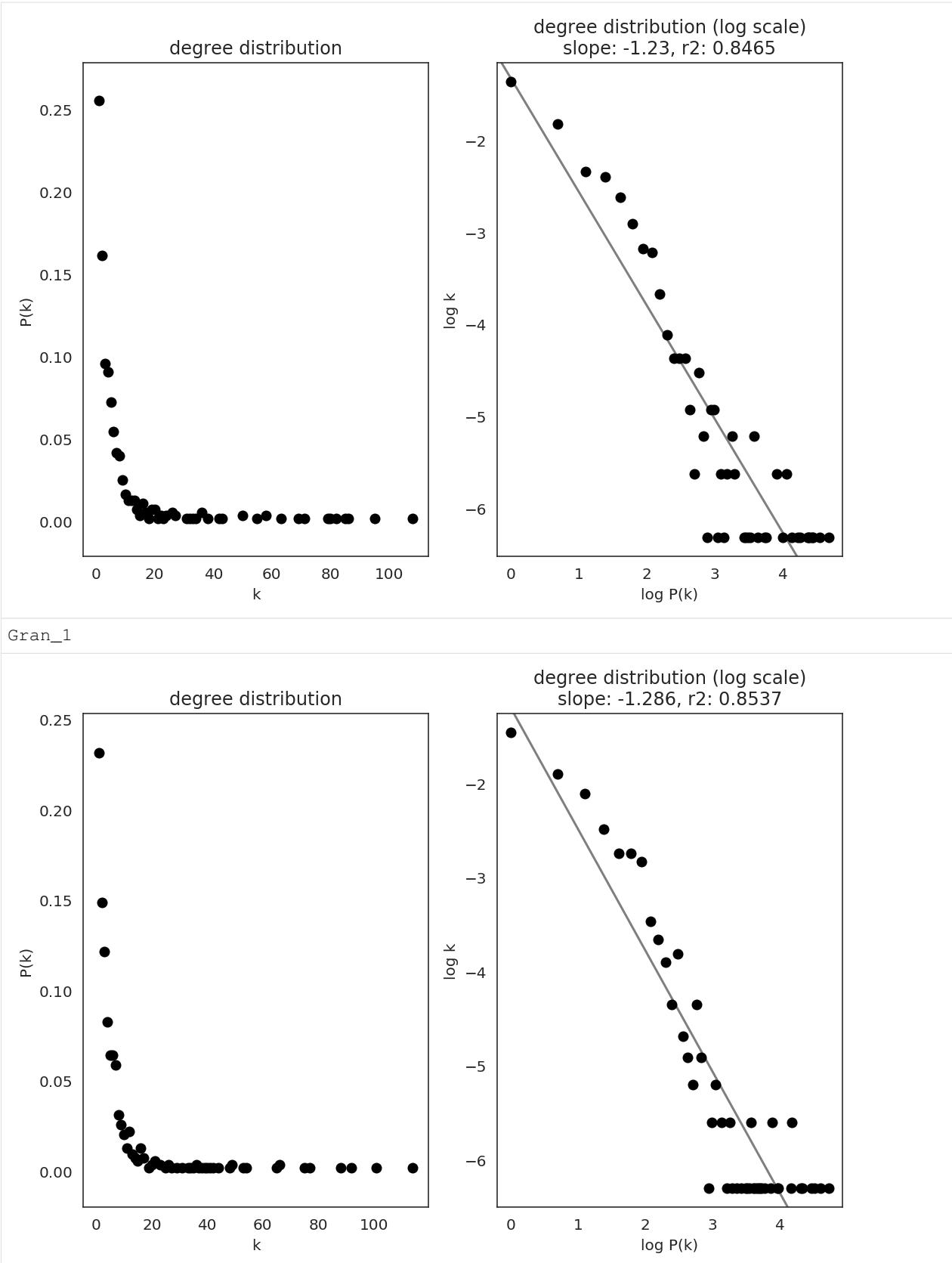




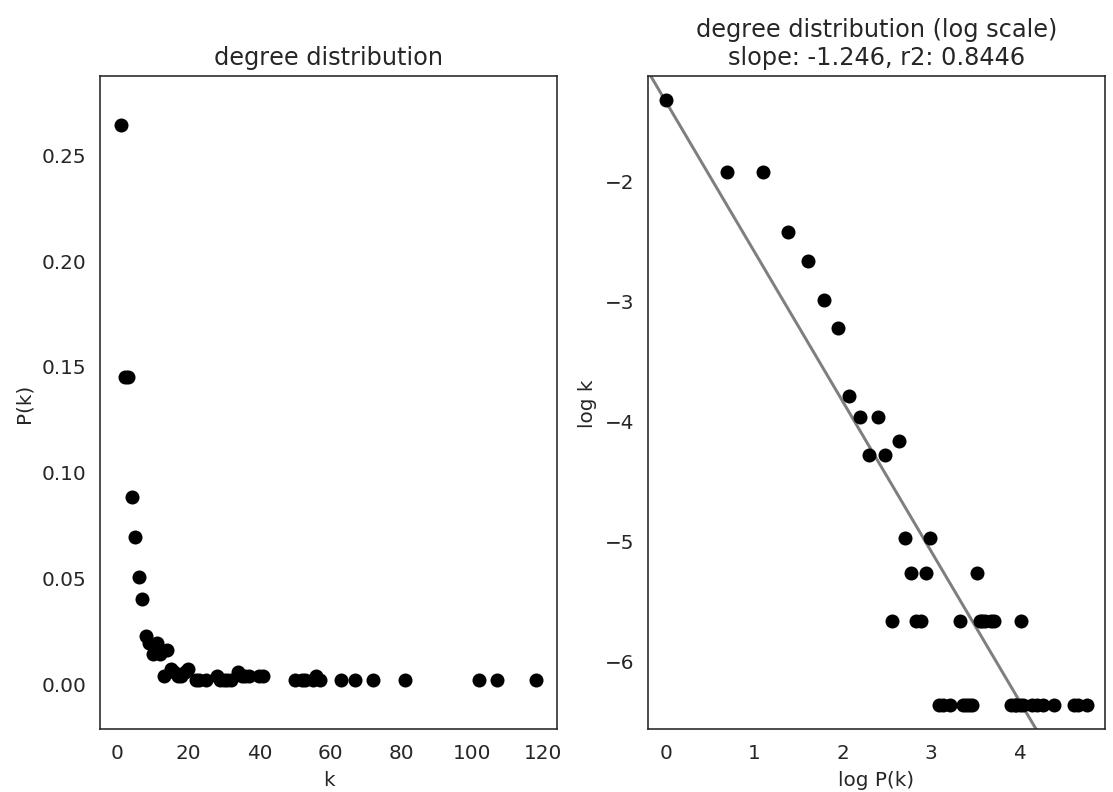
GMP1_0



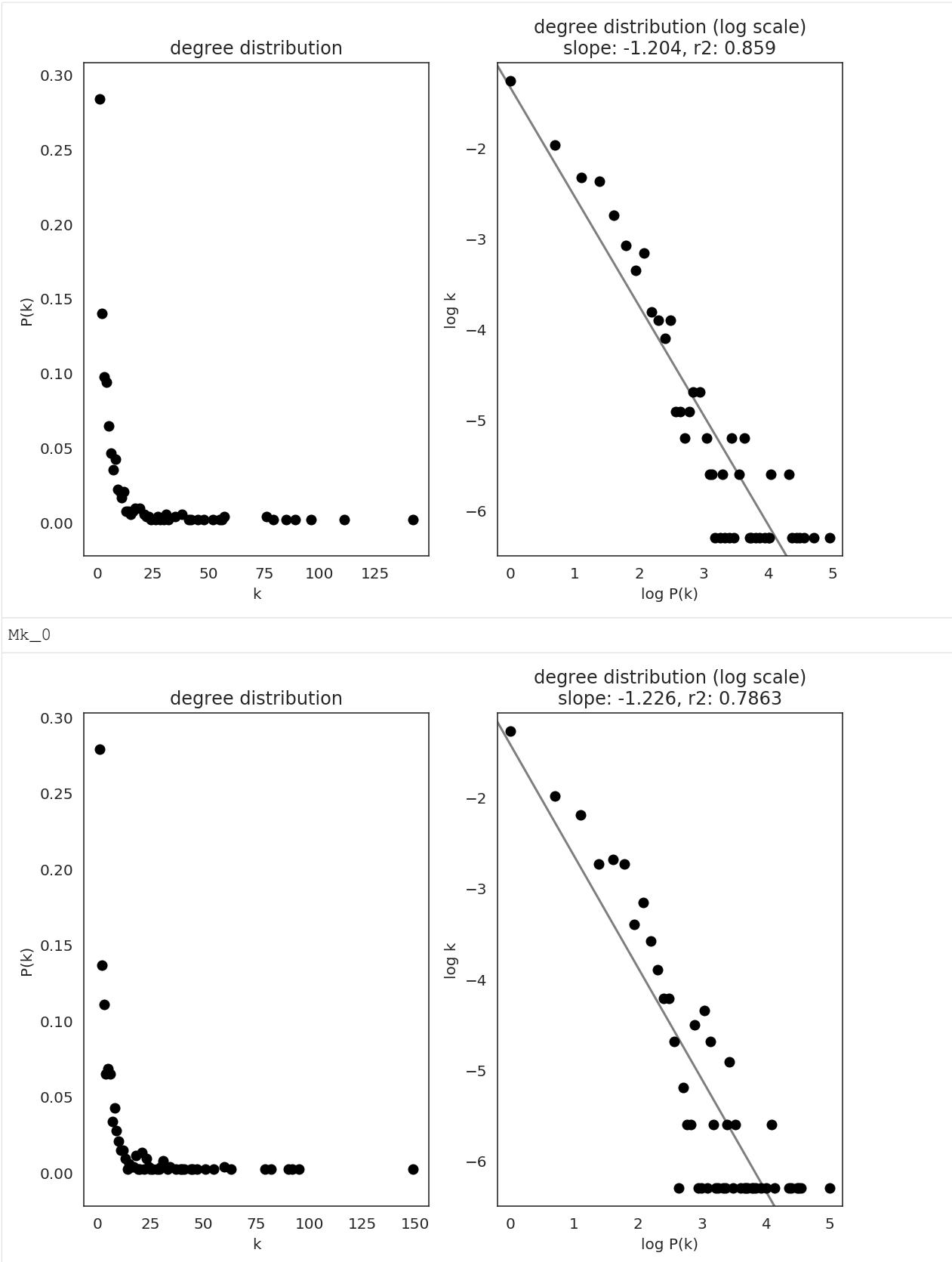
Gran_0



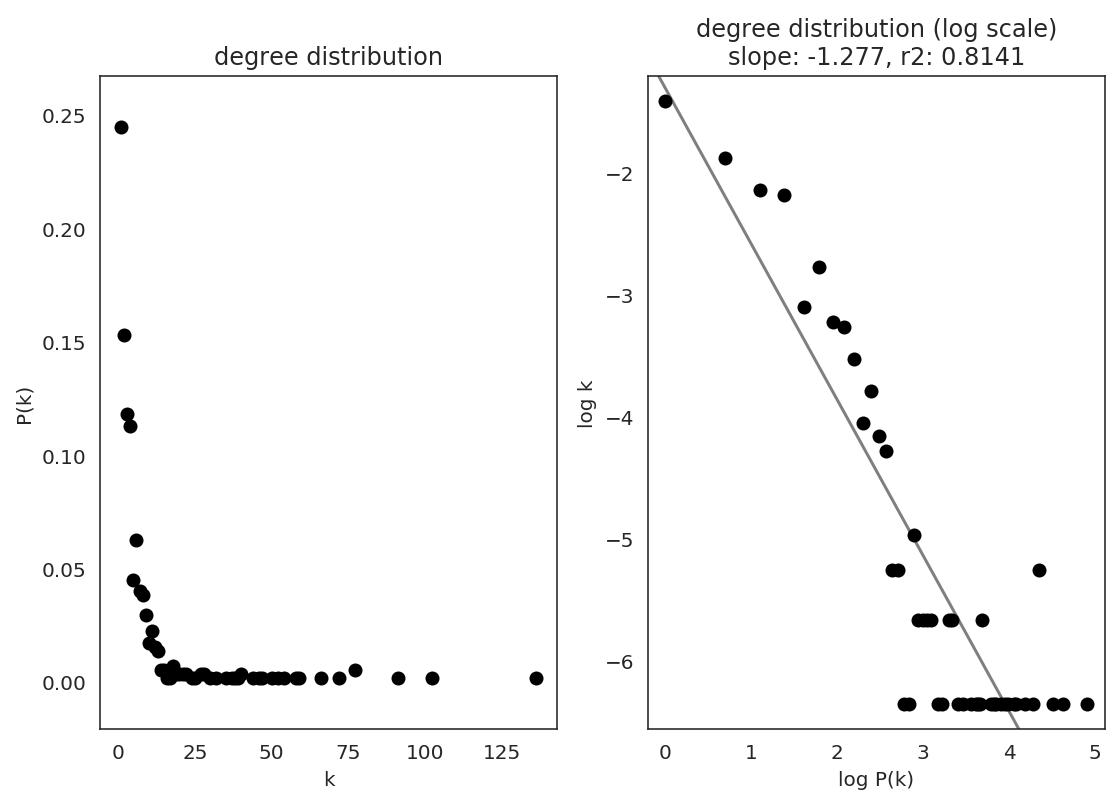
Gran_2



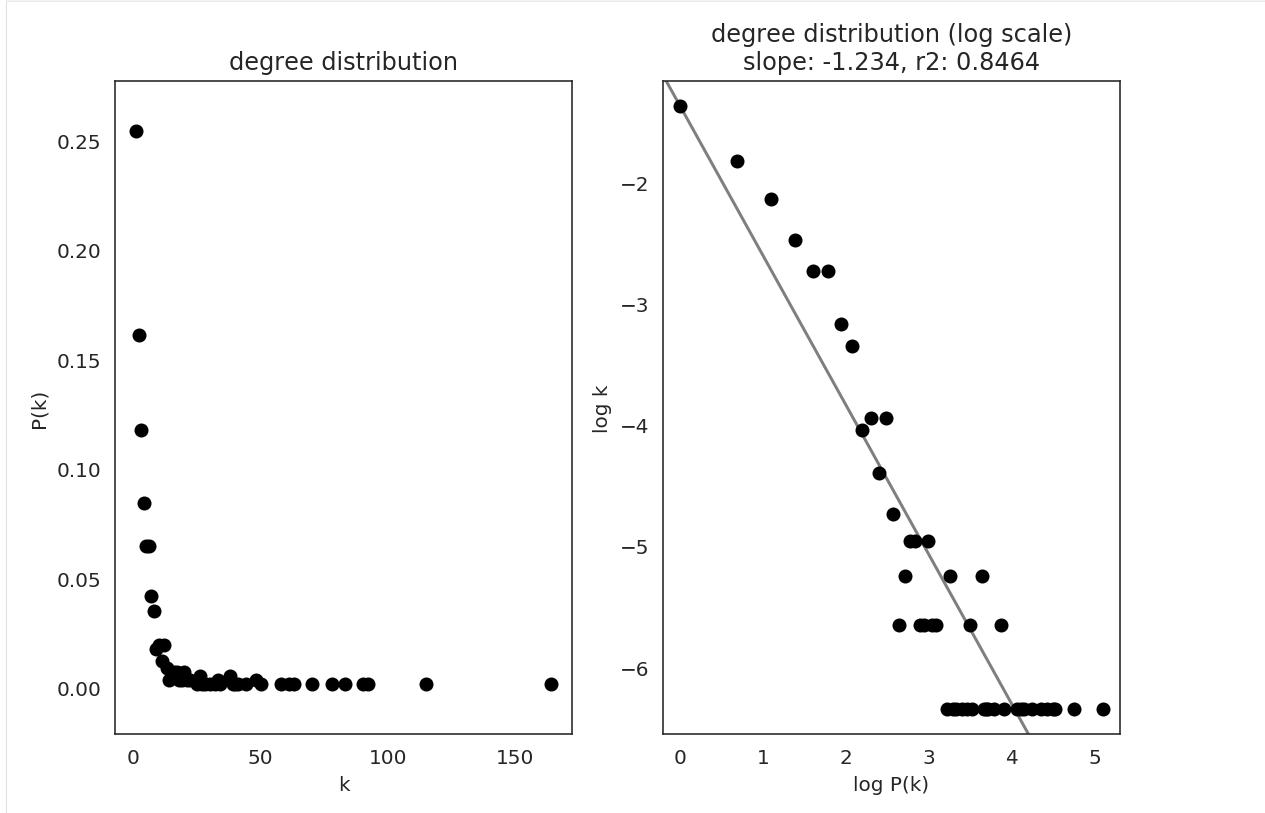
MEP_0



Mo_0



Mo_1



5.3. Calculate netowrk score

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.

```
[39]: # calculate network scores. It takes several minuts.
links.get_score()

processing... batch 1/5
Ery_0: finished.
Ery_1: finished.
Ery_2: finished.
Ery_3: finished.
processing... batch 2/5
Ery_4: finished.
Ery_5: finished.
Ery_6: finished.
Ery_7: finished.
processing... batch 3/5
Ery_8: finished.
Ery_9: finished.
GMP_0: finished.
GMP_1: finished.
processing... batch 4/5
GMP_0: finished.
Gran_0: finished.
Gran_1: finished.
Gran_2: finished.
processing... batch 5/5
```

(continues on next page)

(continued from previous page)

```
MEP_0: finished.
Mk_0: finished.
Mo_0: finished.
Mo_1: finished.
the scores are saved in ./louvain_annot/
```

The score is stored as a attribute “merged_score”, and score will also be saved in a folder.

	links.merged_score.head()
[57]:	<pre>degree_all degree_in degree_out clustering_coefficient \ Stat3 91 0 91 0.019780 MyCN 28 0 28 0.002646 Zbtb1 27 0 27 0.005698 E2f4 186 3 183 0.009474 Ybx1 71 9 62 0.027364 clustering_coefficient_weighted degree_centrality_all \ Stat3 0.020122 0.167279 MyCN 0.001828 0.051471 Zbtb1 0.008576 0.049632 E2f4 0.011623 0.341912 Ybx1 0.027485 0.130515 degree_centrality_in degree_centrality_out betweenness_centrality \ Stat3 0.000000 0.167279 0 MyCN 0.000000 0.051471 0 Zbtb1 0.000000 0.049632 0 E2f4 0.005515 0.336397 2788 Ybx1 0.016544 0.113971 1153 closeness_centrality ... assortative_coefficient \ Stat3 0.000013 ... -0.172207 MyCN 0.000004 ... -0.172207 Zbtb1 0.000011 ... -0.172207 E2f4 0.000010 ... -0.172207 Ybx1 0.000004 ... -0.172207 average_path_length community_edge_betweenness community_random_walk \ Stat3 2.475244 1 1 MyCN 2.475244 2 1 Zbtb1 2.475244 3 1 E2f4 2.475244 4 1 Ybx1 2.475244 5 11 community_eigenvector module connectivity participation \ Stat3 1 0 4.052023 0.632291 MyCN 5 2 2.347146 0.625000 Zbtb1 5 1 2.275431 0.666667 E2f4 3 0 8.496354 0.637760 Ybx1 4 3 5.433857 0.564967 role cluster Stat3 Connector Hub Ery_0 MyCN Connector Ery_0 Zbtb1 Connector Ery_0 E2f4 Connector Hub Ery_0</pre>

(continues on next page)

(continued from previous page)

```
Ybx1    Connector Hub    Ery_0  
[5 rows x 22 columns]
```

5.4. Save

Save processed GRN. The Links file will also be used in the next notebook; “in silico simulation with GRNs”.

```
[42]: # Save Links object.  
links.to_hdf5(file_path="links.celloracle.links")  
  
# You can load files with the following command.  
#links = co.load_hdf5(file_path="links.celloracle.links")
```

If you are not interested in the network score analysis and just want to do simulation, you can skip the analysis below and go to the next notebook.

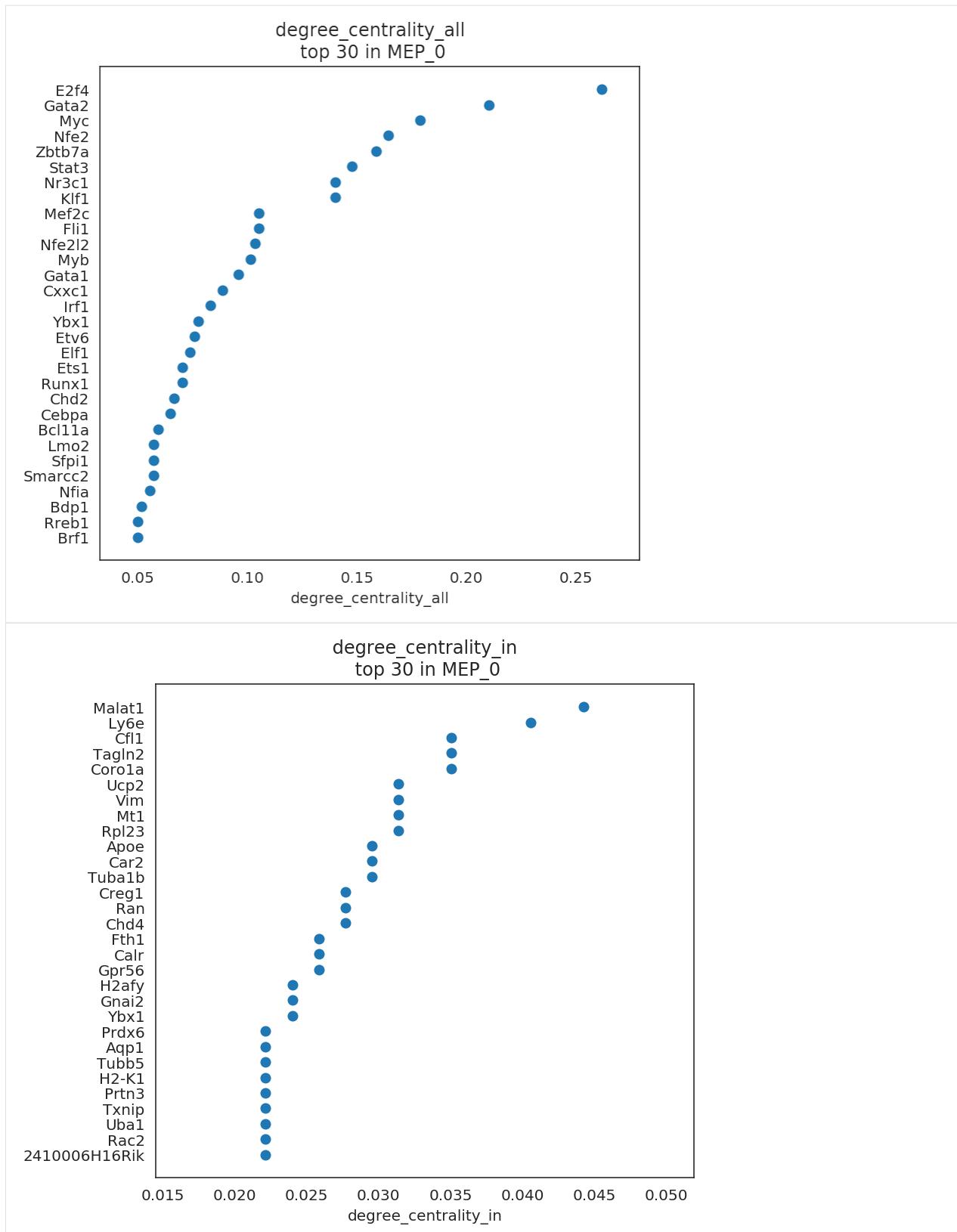
6. Network analysis; Network score for each gene

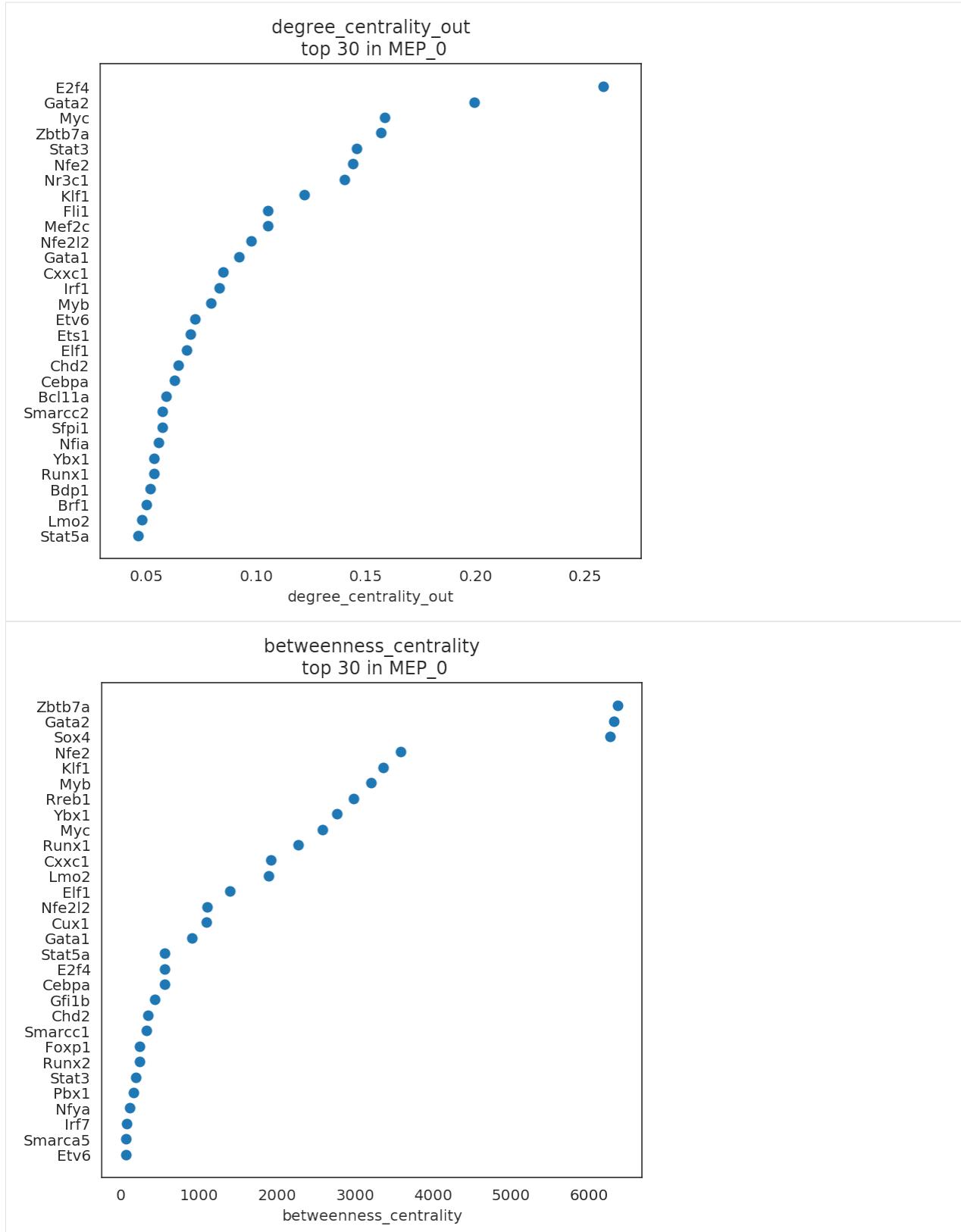
Links class has many functions to visualize network score. See the documentation for the details of the functions.

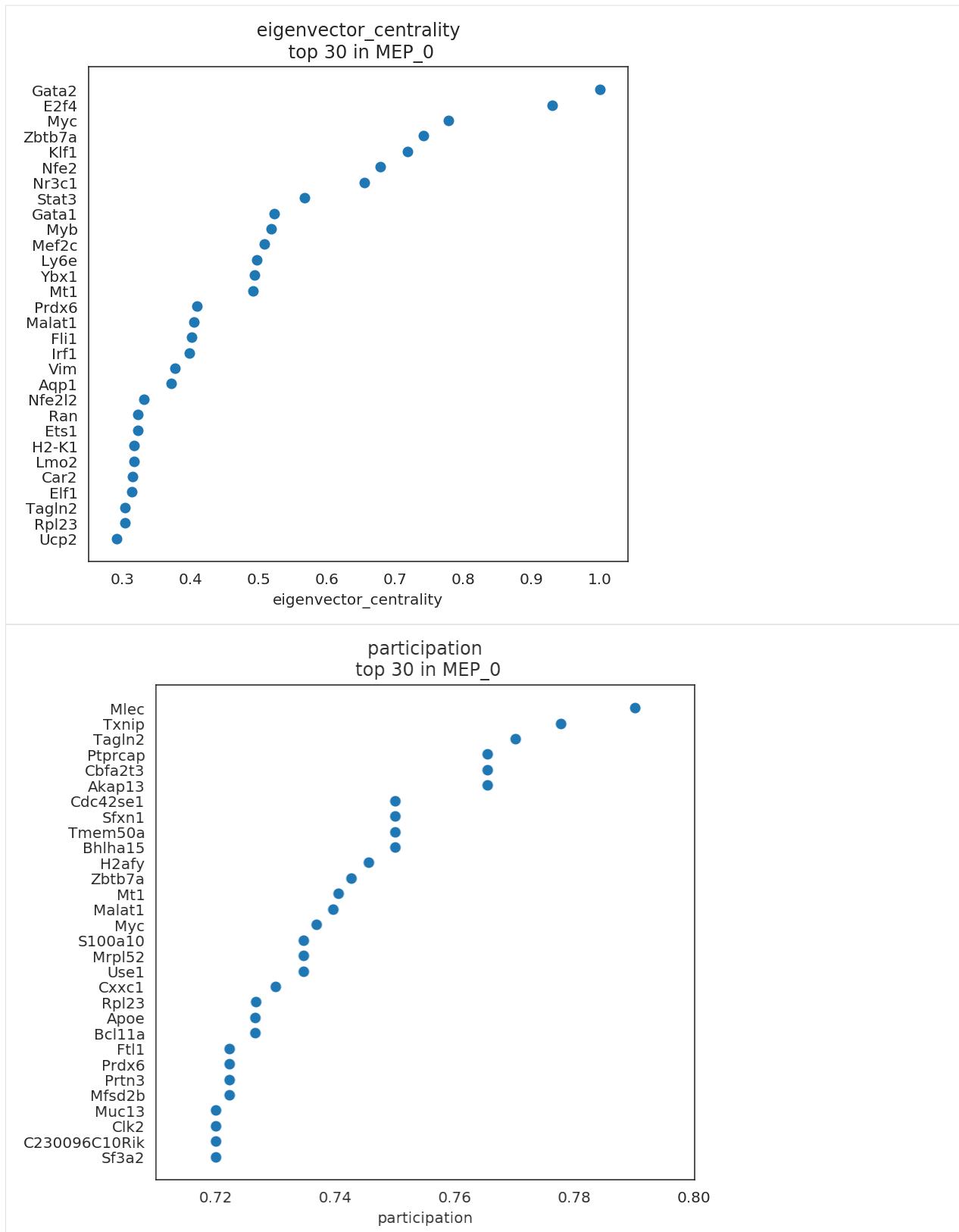
6.1. Network score in each cluster

```
[44]: # check cluster  
links.cluster  
  
[44]: ['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_1_0',  
       'Gran_0',  
       'Gran_1',  
       'Gran_2',  
       'MEP_0',  
       'Mk_0',  
       'Mo_0',  
       'Mo_1']
```

```
[88]: # 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")
```



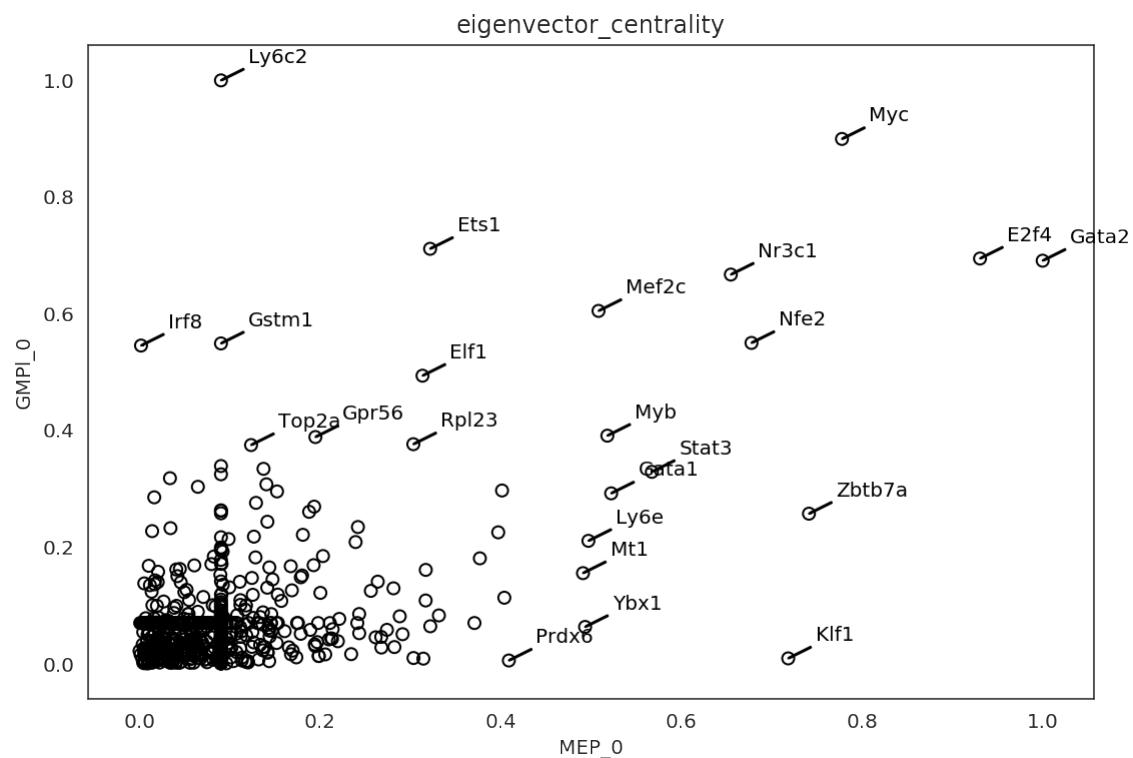




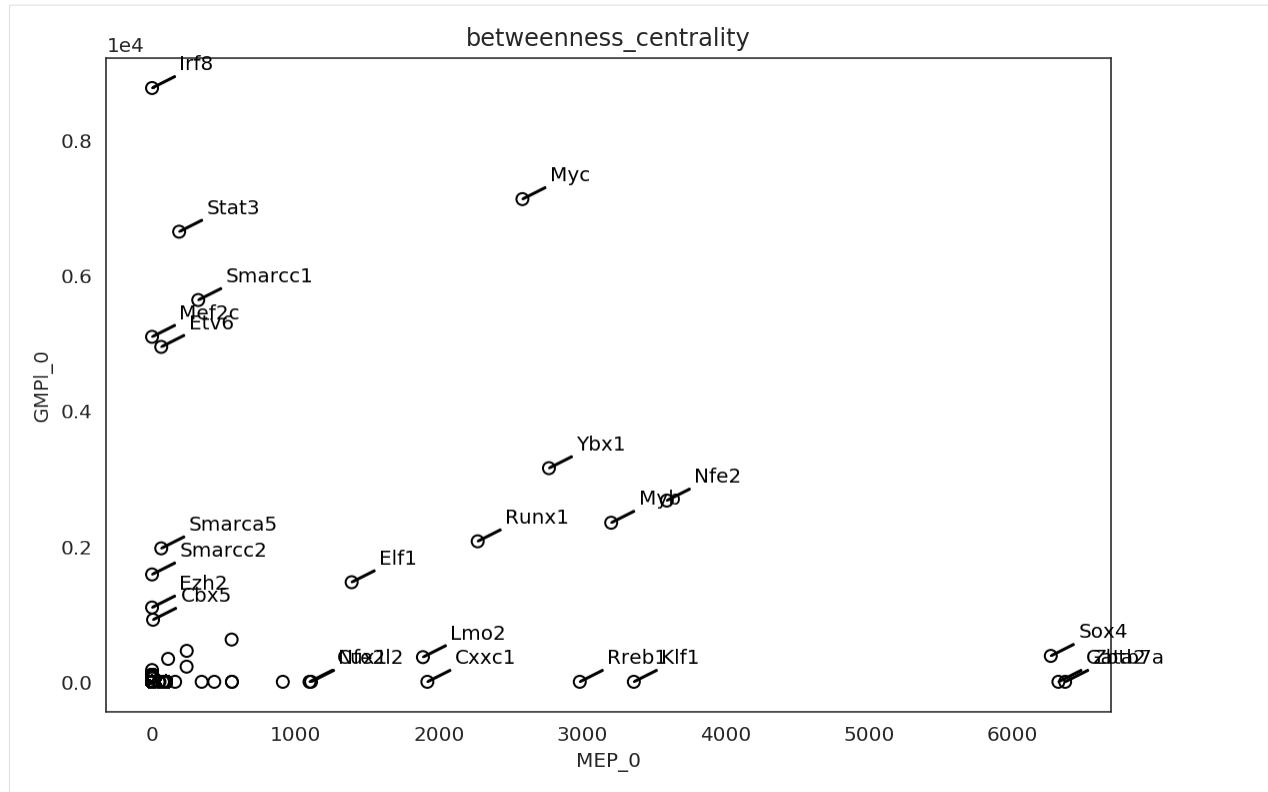
6.2. Network score comparison between two clusters

By comparing network scores between two clusters, we can analyze the difference of GRN structure.

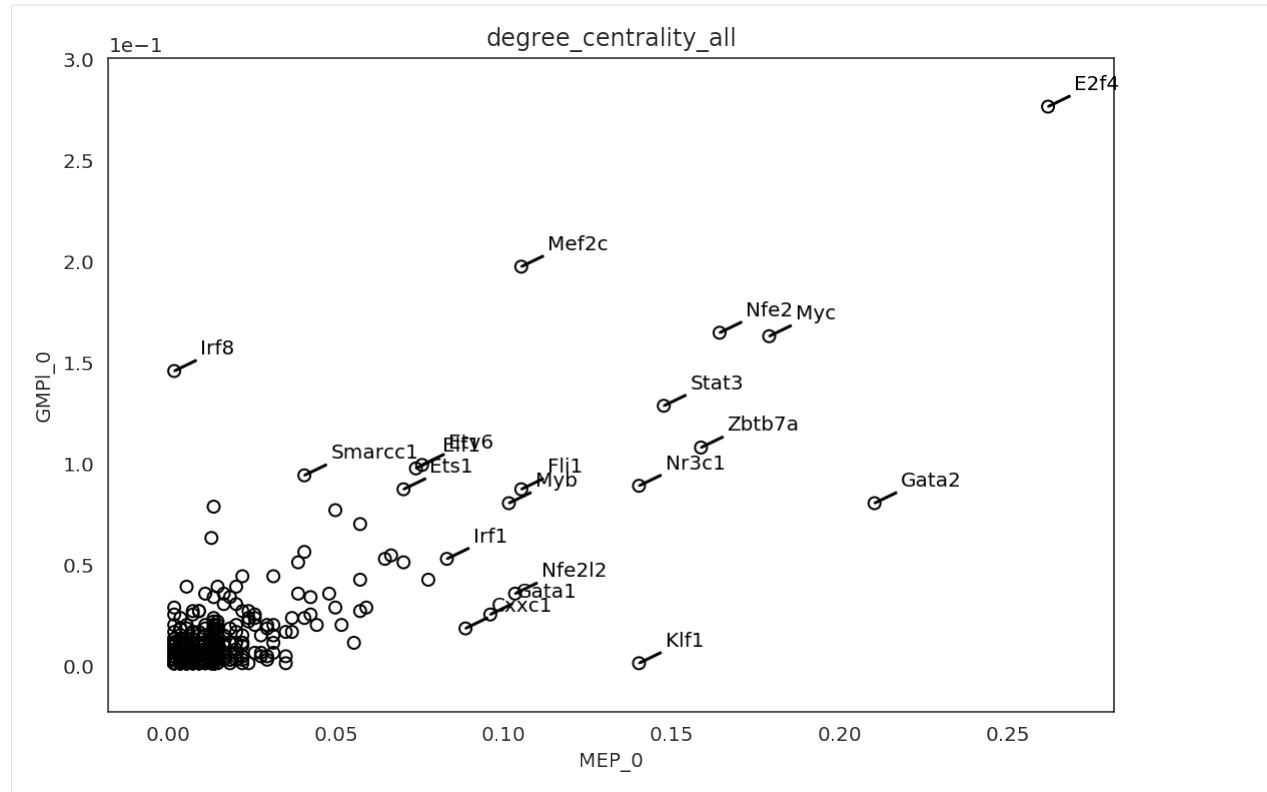
```
[89]: plt.ticklabel_format(style='sci',axis='y',scilimits=(0,0))
links.plot_score_comparison_2D(value="eigenvector_centrality",
                                cluster1="MEP_0", cluster2="GMP1_0",
                                percentile=98, save=f"{save_folder}/score_comparison")
```



```
[90]: 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")
```



```
[91]: 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")
```

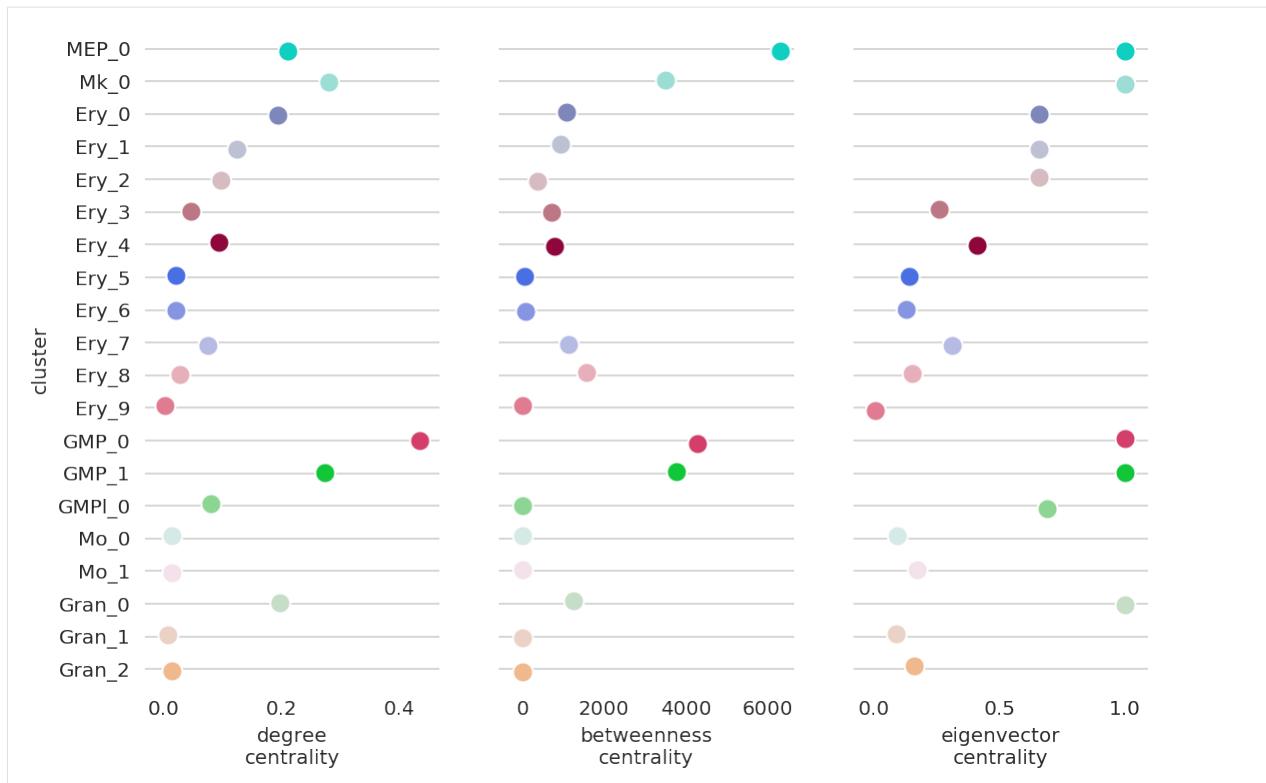


6.3. Network score dynamics

We can analyze how gene function/importance change over the differentiation by the following analysis.

Gata2 is known to play an important role in the early progenitor population; MEP, GMP. The following results can recapitulate the Gata2 feature.

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



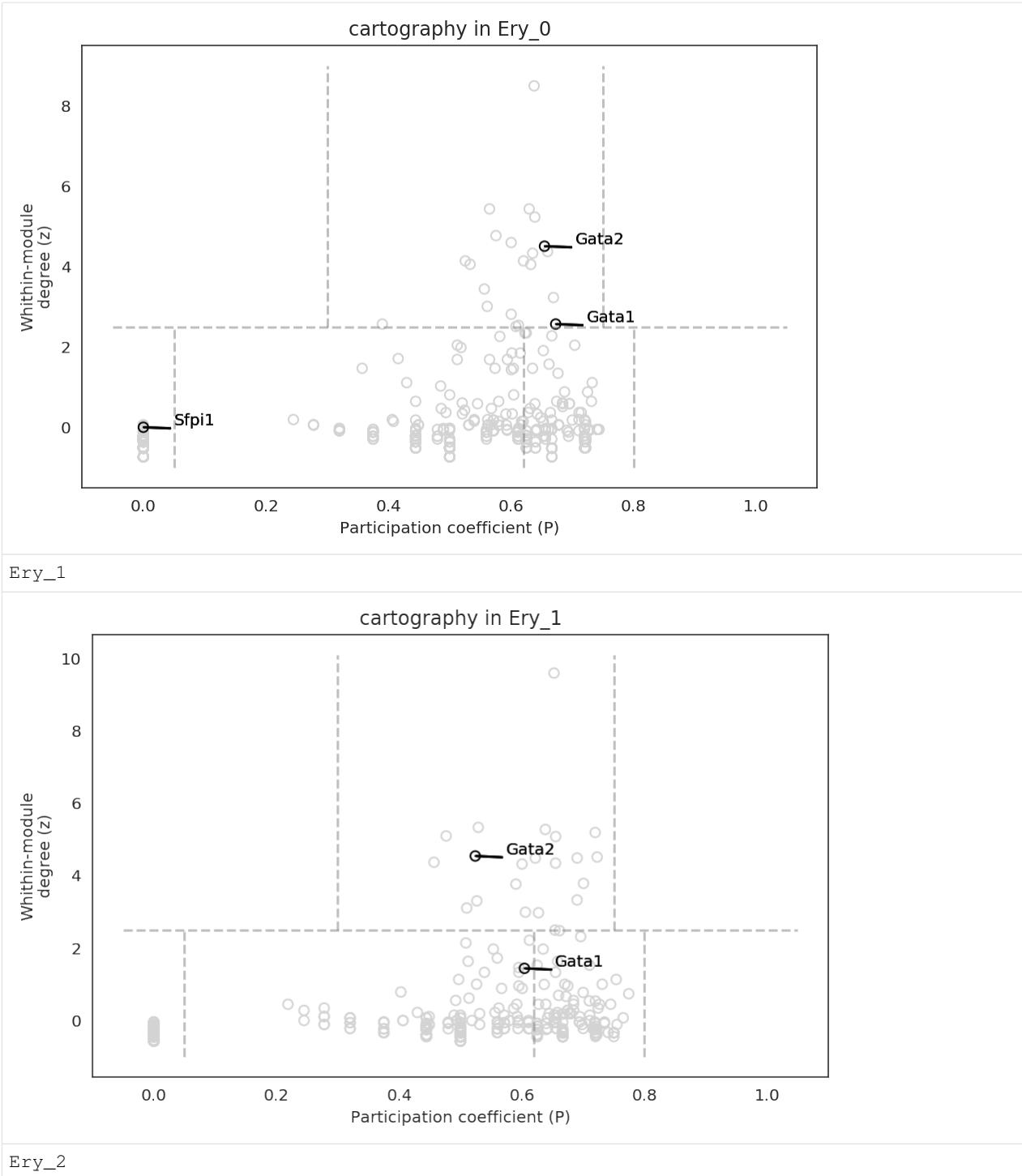
6.4. Gene cartography analysis

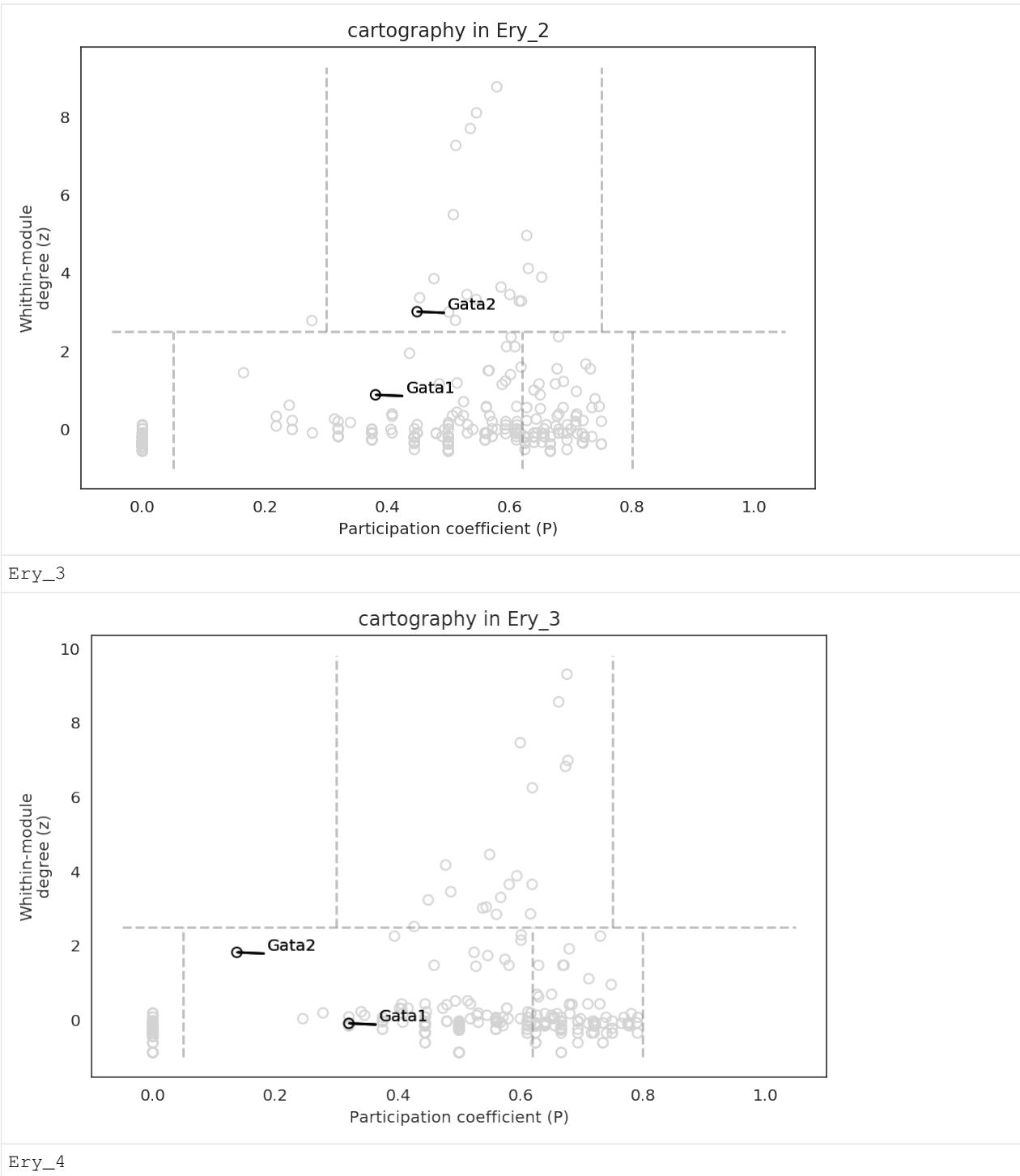
Gene cartography is a method for gene network analysis. The method classifies the gene into several groups using the network module structure and gene connections. It provides us an insight about the regulatory mechanism per each gene. Please read the paper for the details of gene cartography (<https://www.nature.com/articles/nature03288>)

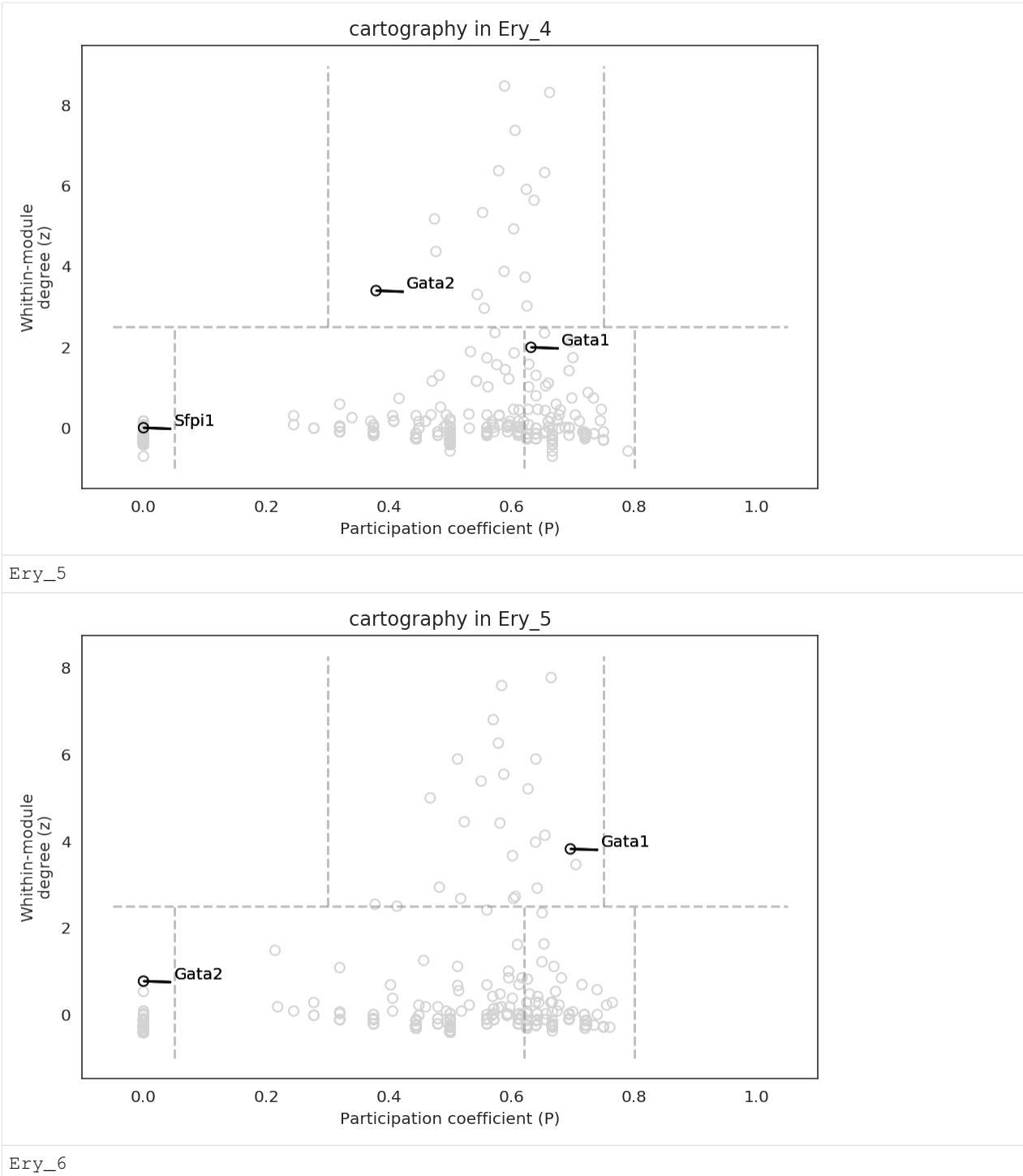
The calculations of gene cartography will be performed for the GRN in each cluster. Thus we can know how gene role change by comparing the cartography in each cluster.

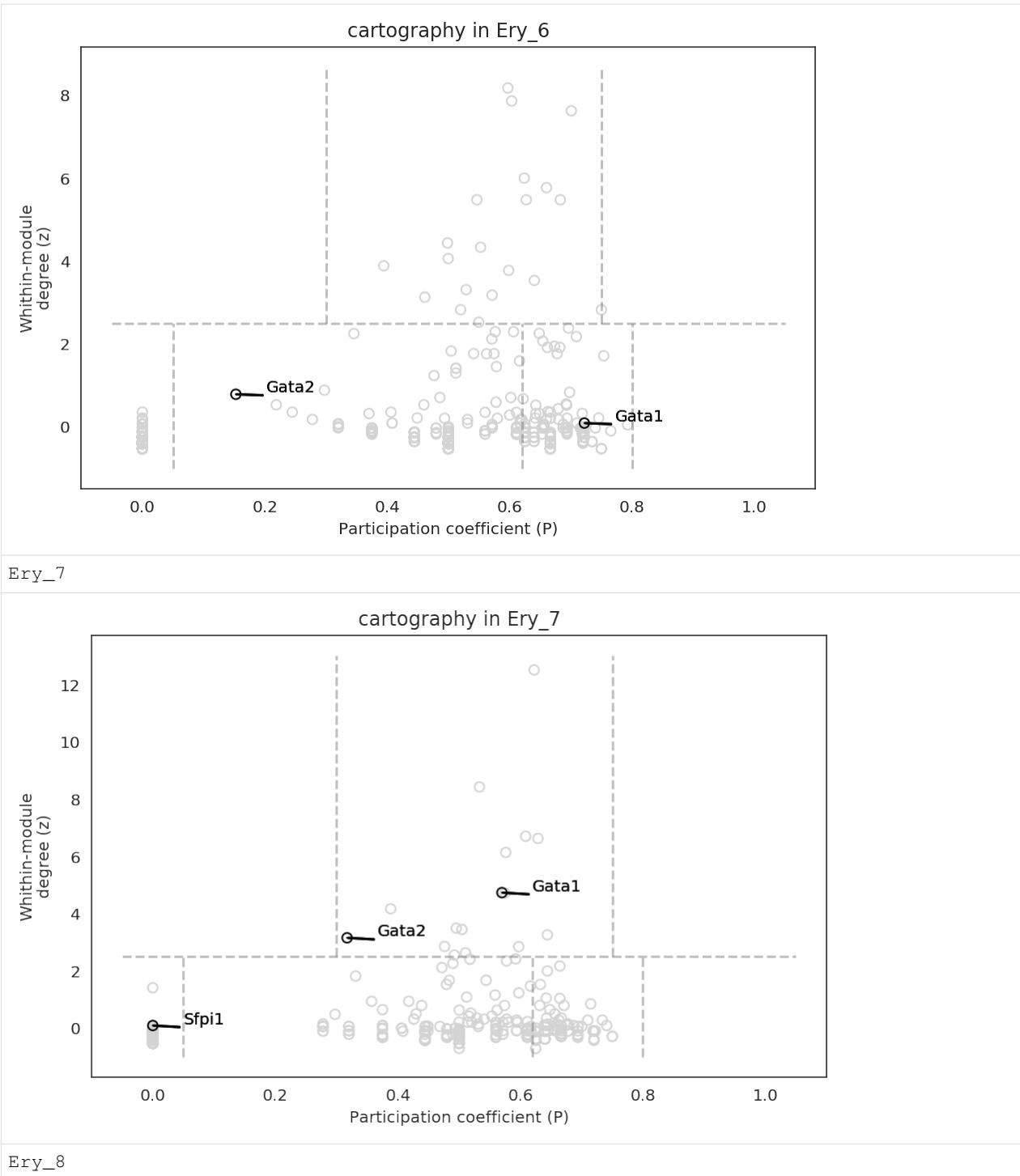
```
[93]: # plot cartography as a scatter plot
links.plot_cartography_scatter_per_cluster(scatter=True,
                                             kde=False,
                                             gois=["Gata1", "Gata2", "Sfpi1"],
                                             auto_gene_annot=False,
                                             percentile=99.5,
                                             args_dot={"n_levels": 105},
                                             args_line={"c": "gray"}, save=f"{{save_
                                             ↪folder}}/cartography")
```

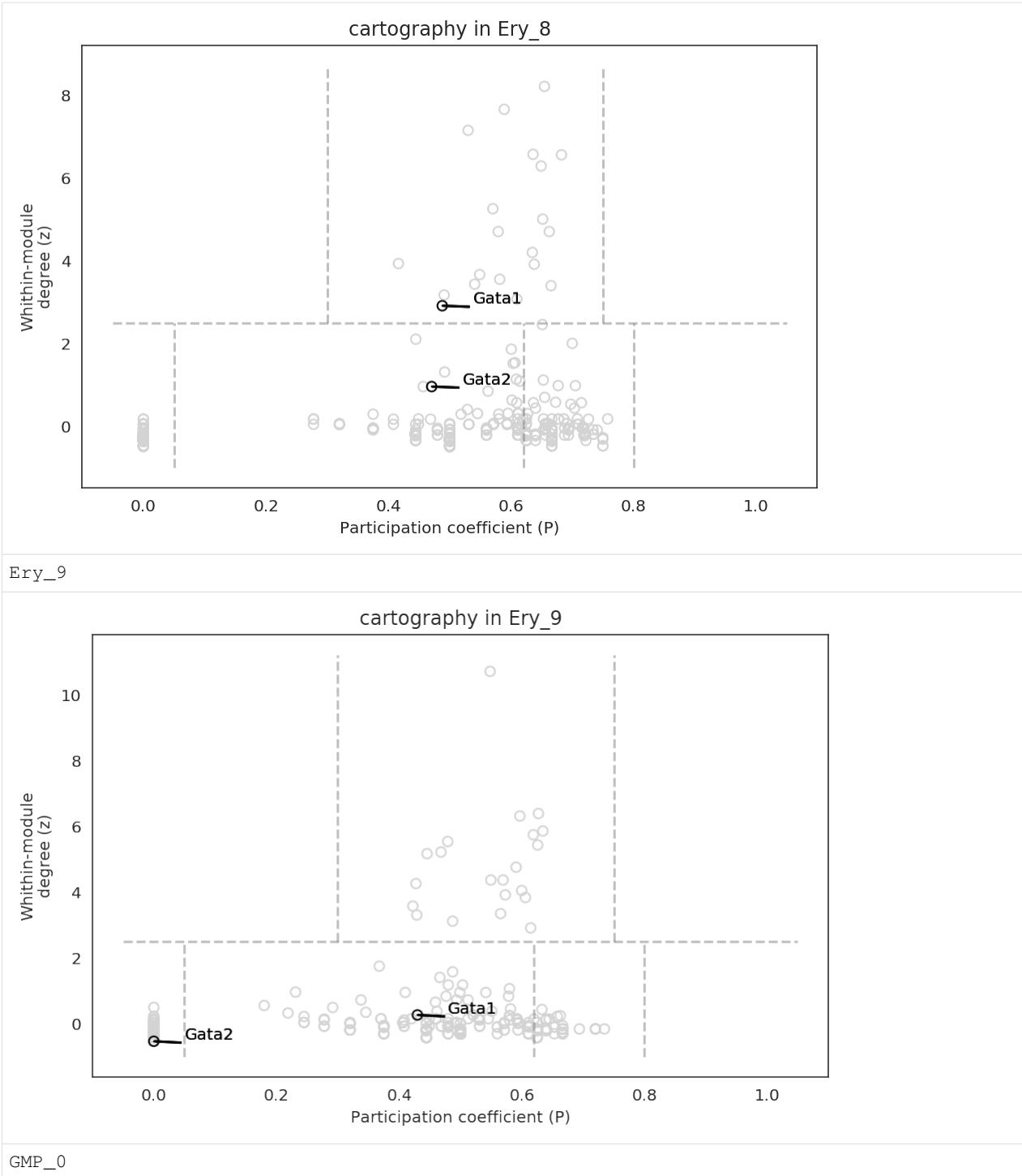
Ery_0

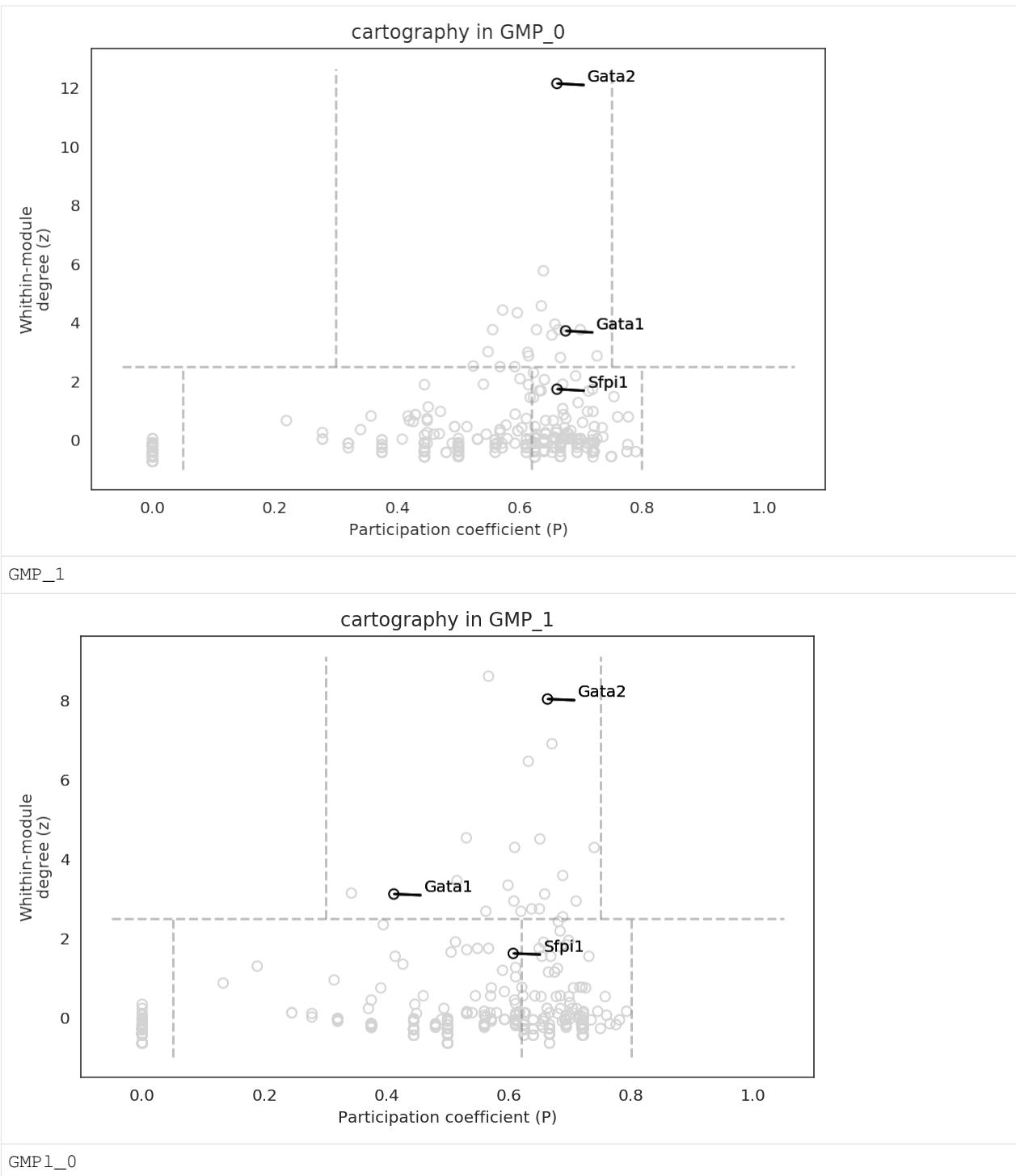


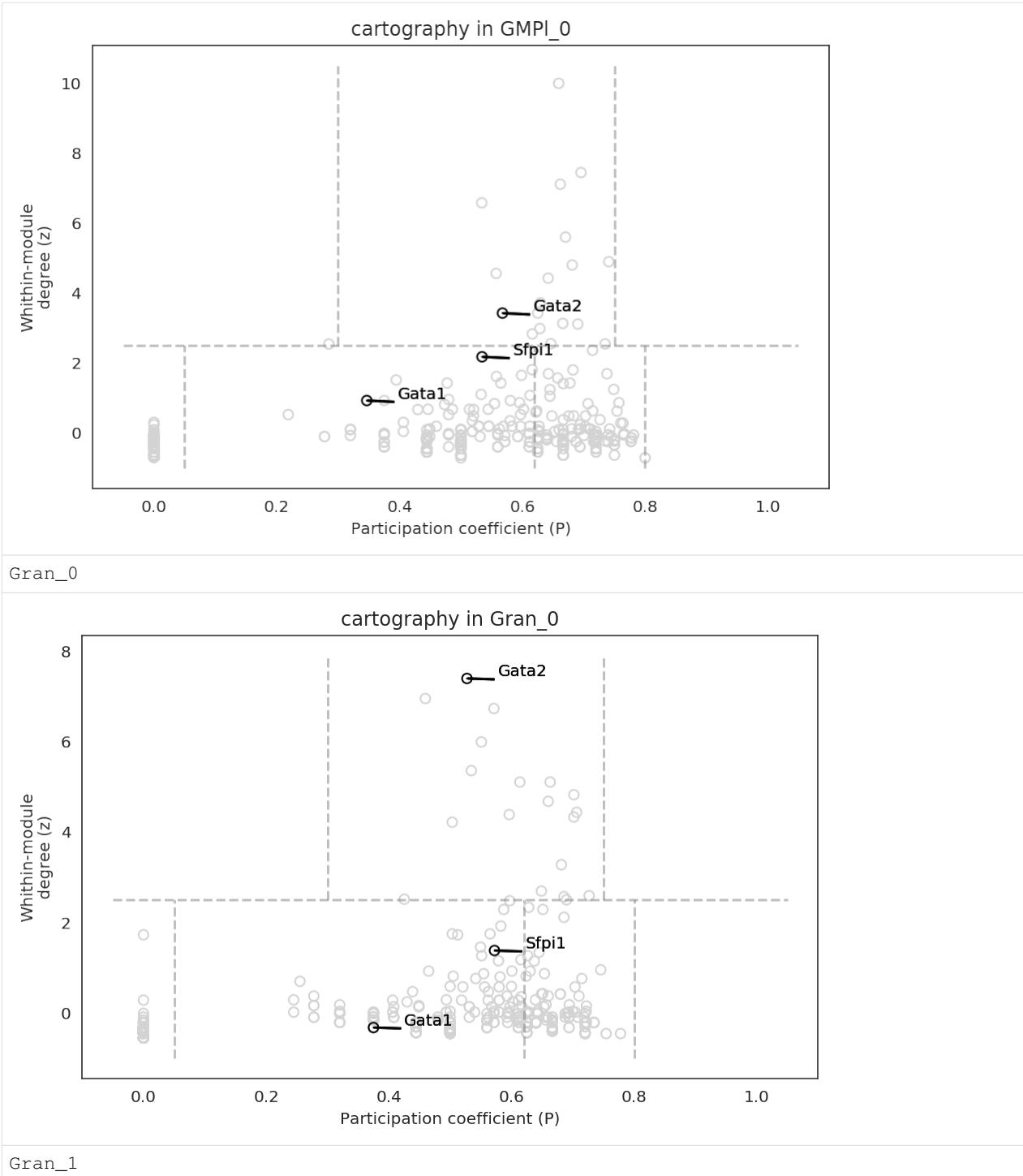


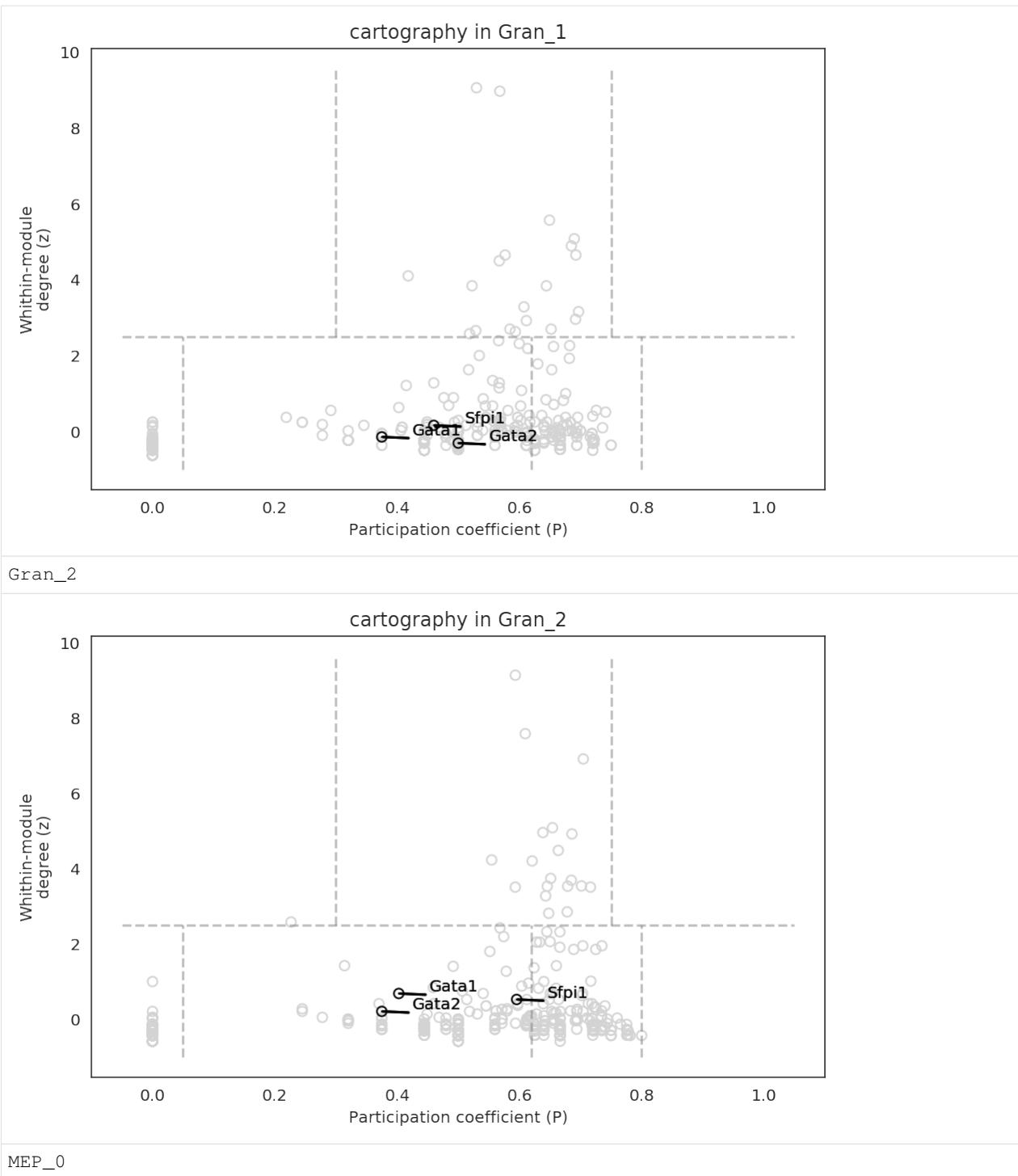


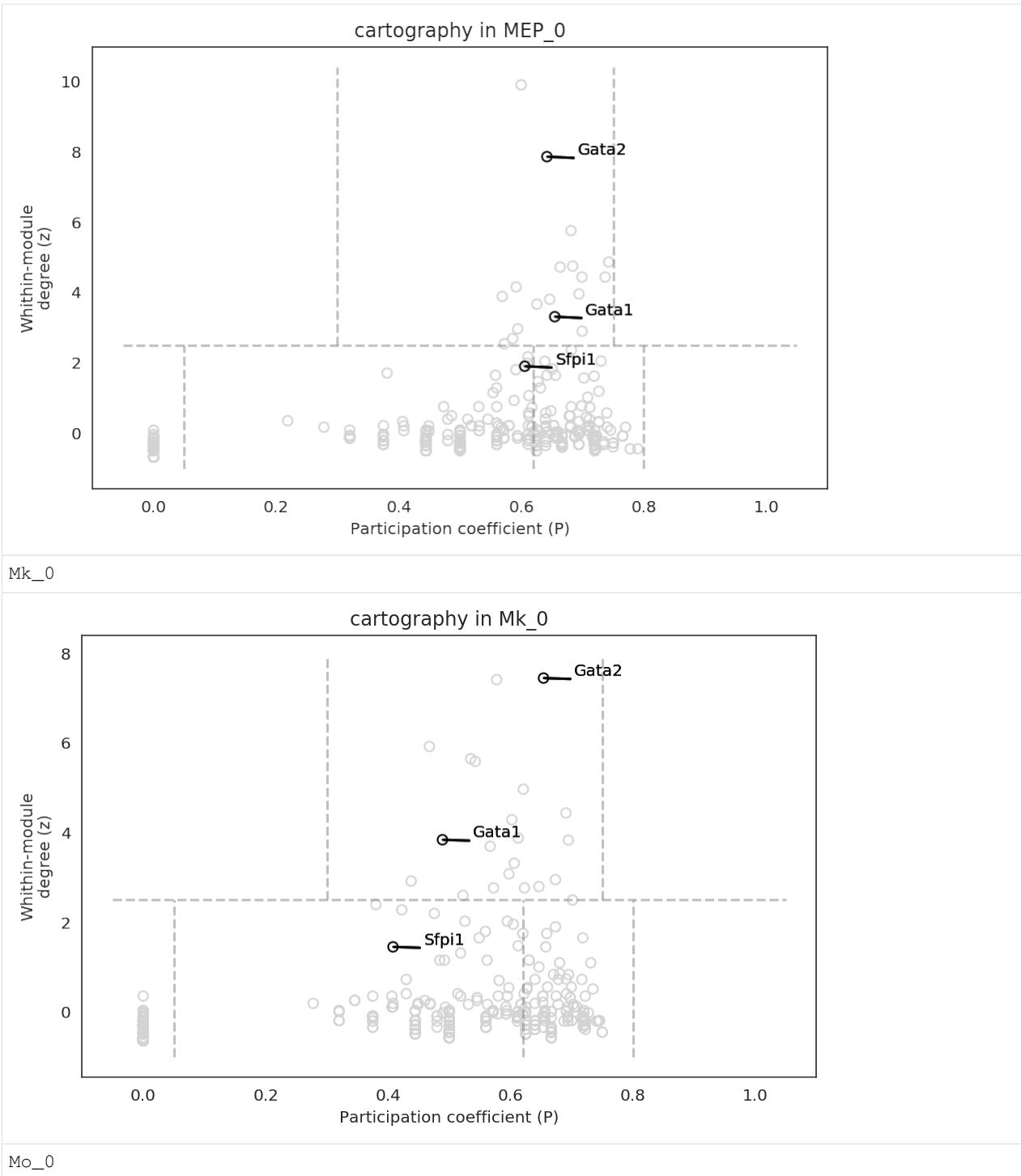


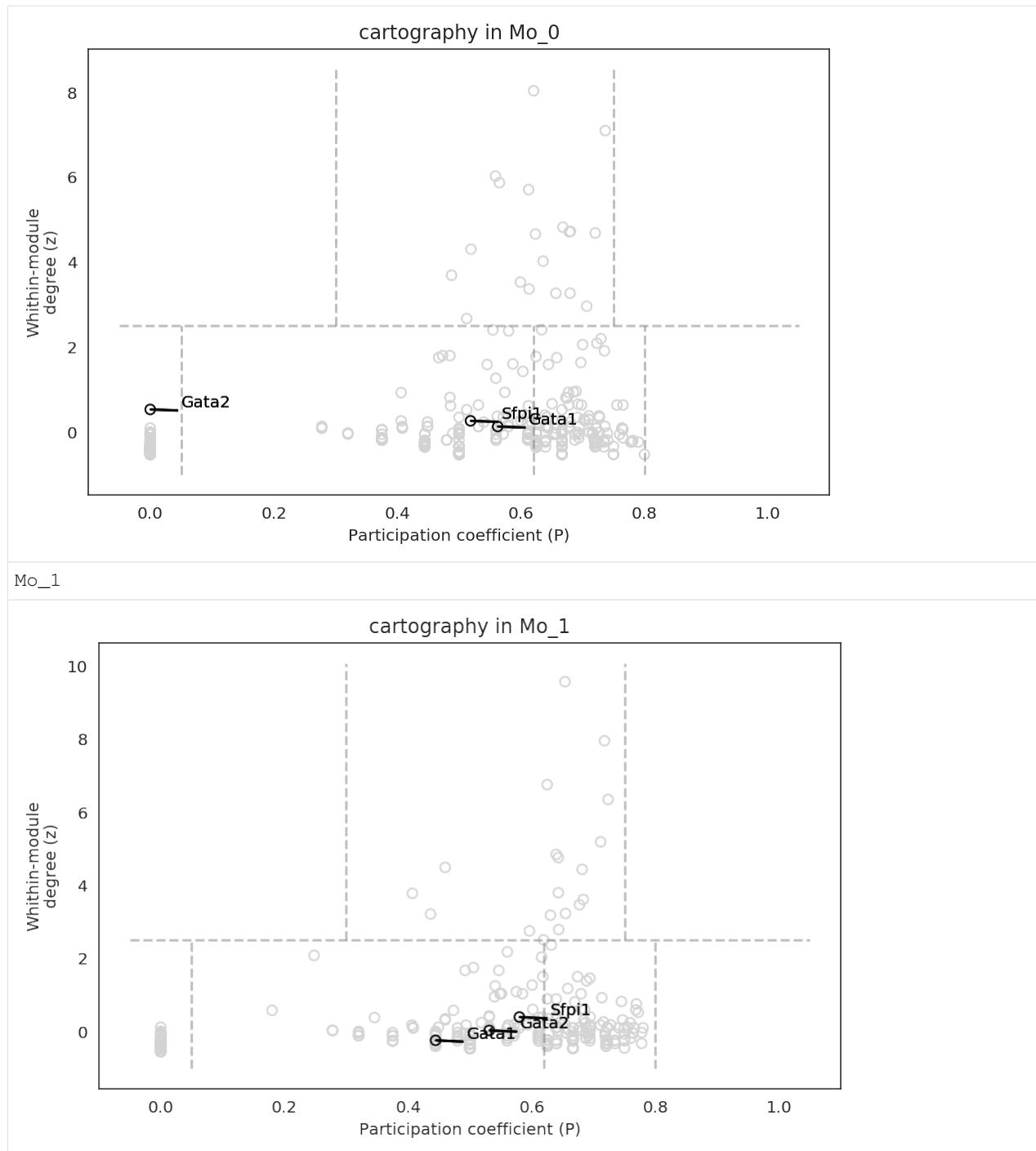












```
[94]: # plot the results of cartography analysis
plt.figure(figsize=[3, 6])
links.plot_cartography_term(goi="Gata2", save=f"{save_folder}/cartography")
Gata2
/home/k/anaconda3/envs/test/lib/python3.6/site-packages/pandas/core/indexing.py:1418: FutureWarning:
  Passing list-like to .loc or [] with any missing label will raise
```

(continues on next page)

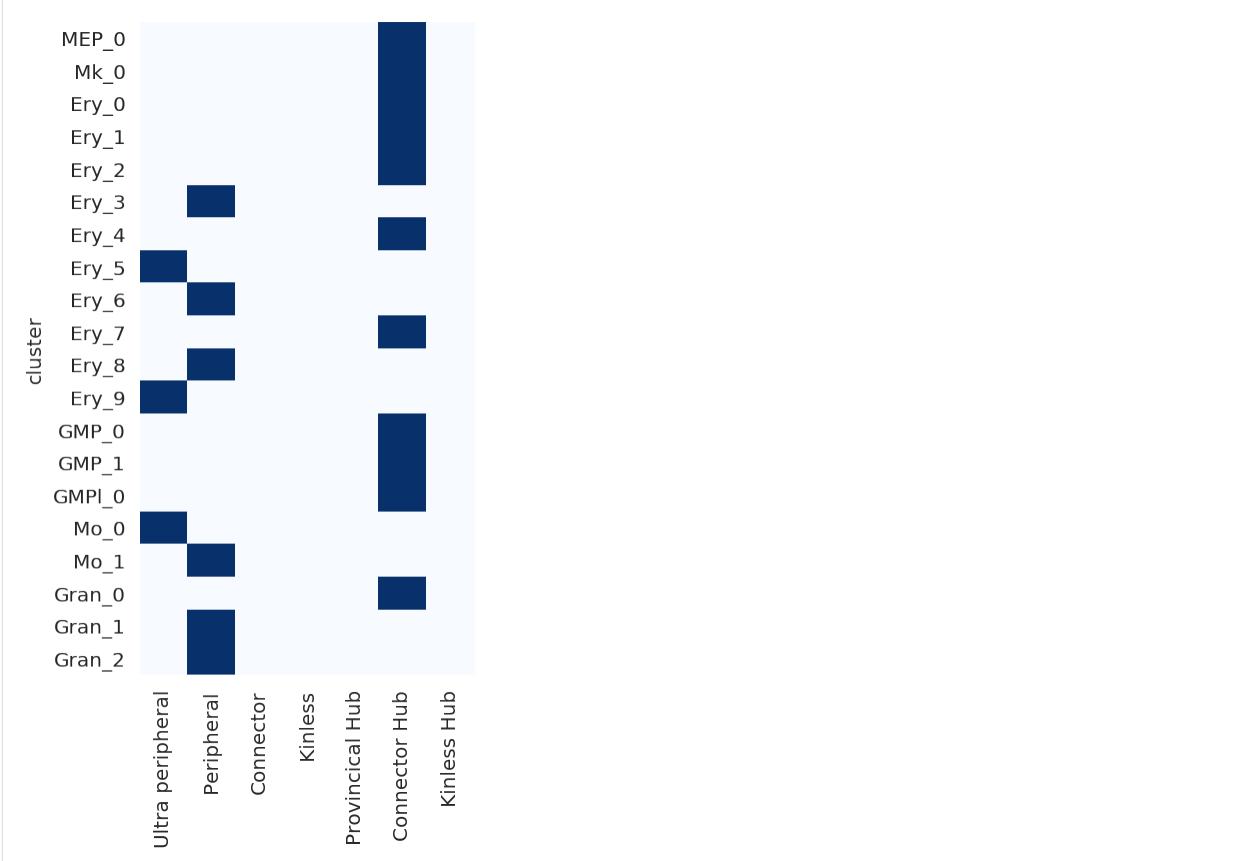
(continued from previous page)

```
KeyError in the future, you can use .reindex() as an alternative.
```

See the documentation here:

https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#deprecate-loc-reindex-listlike

```
    ↪ return self._getitem_tuple(key)
```



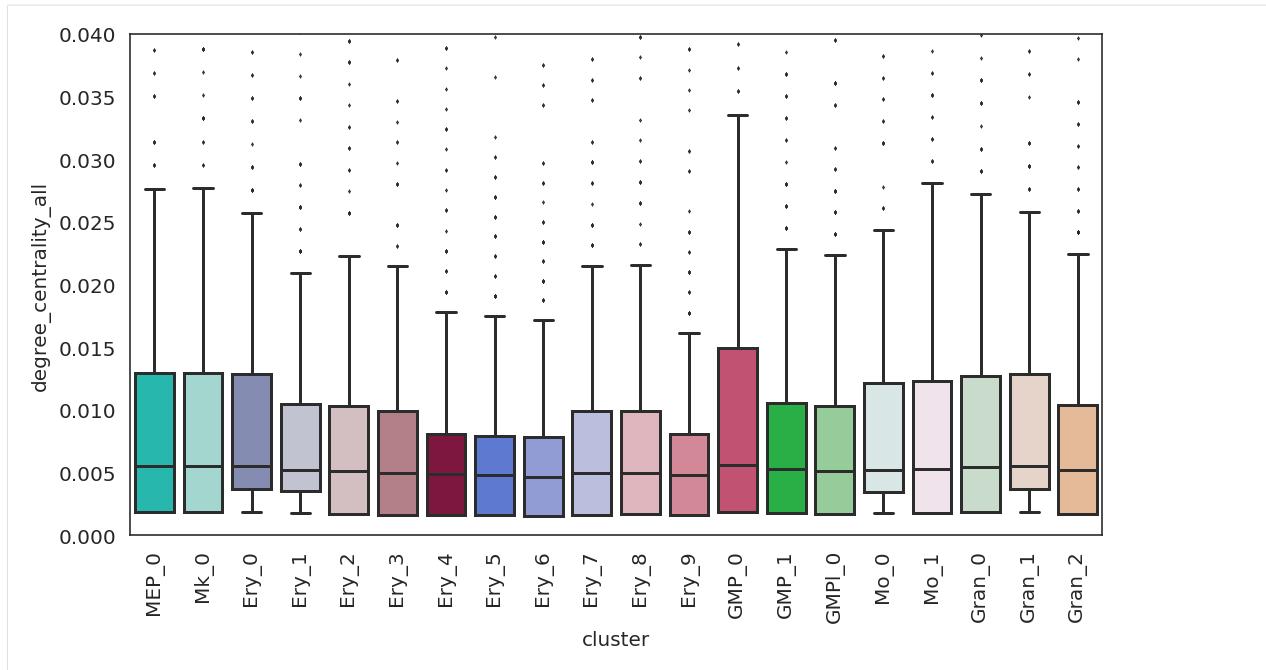
7. Network analysis; Network score distribution

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

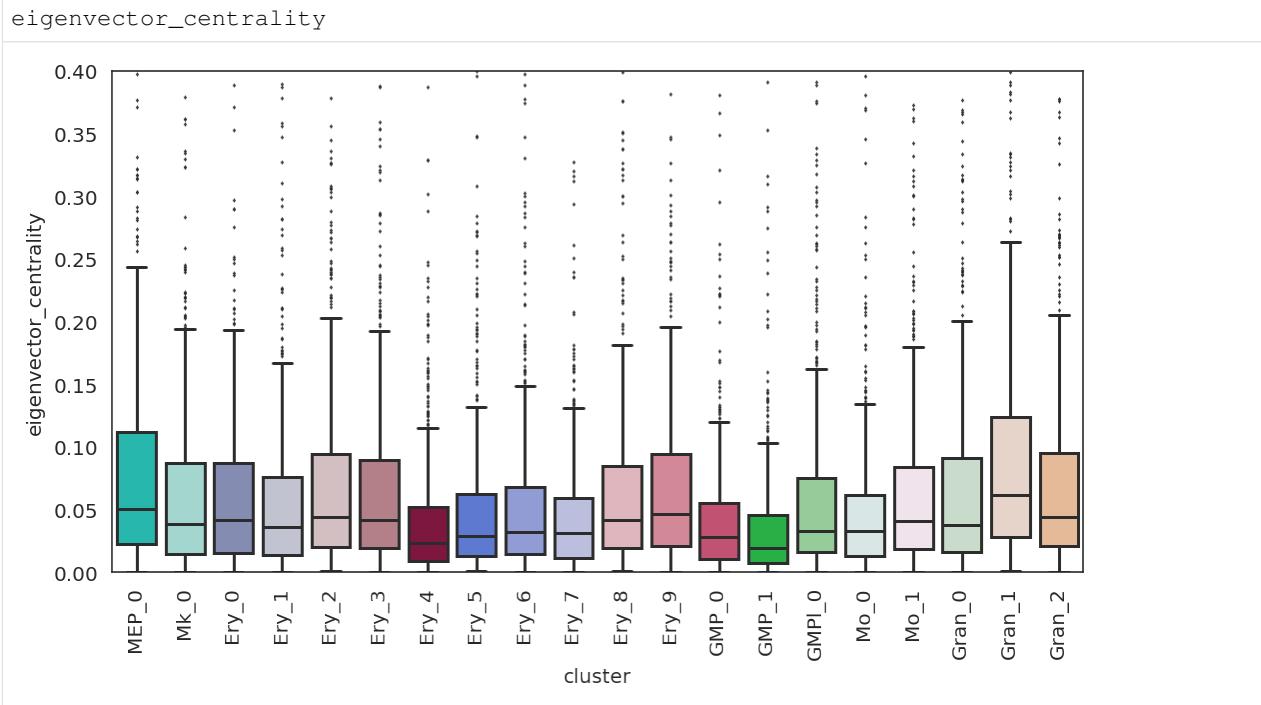
7.1. Distribution of network degree

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

```
degree_centrality_all
```

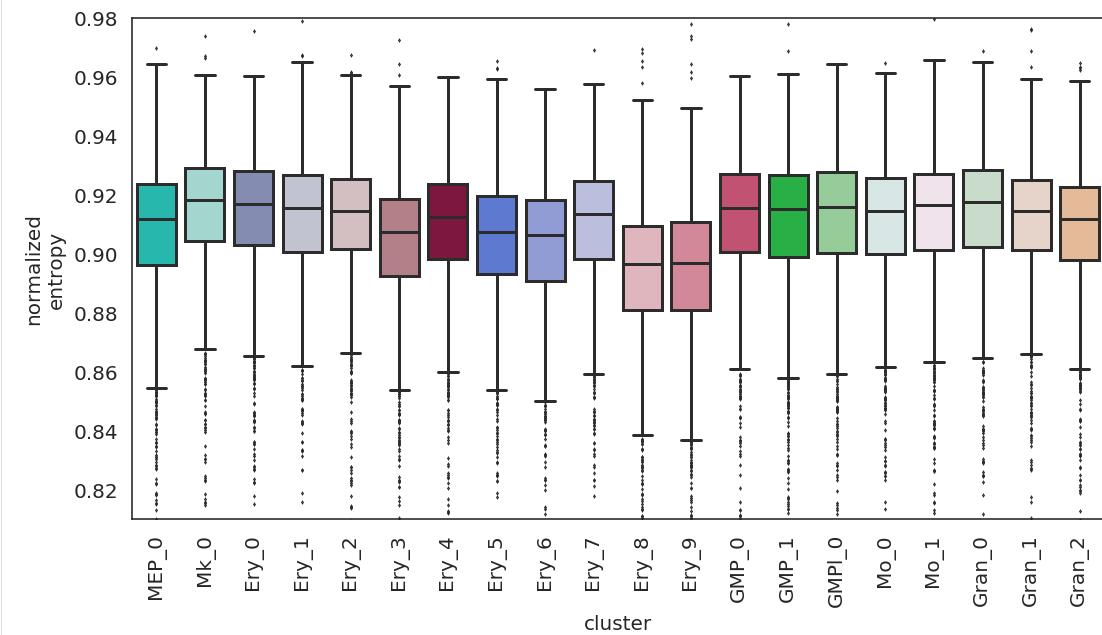


```
[96]: plt.subplots_adjust(left=0.15, bottom=0.3)
plt.ylim([0, 0.40])
links.plot_score_distributions(values=["eigenvector_centrality"], method="boxplot", ↴
save=f"{save_folder}")
```



7.2. Distribution of netowrk entology

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



In the example below, we did several network analysis and showed that several TFs have high network scores. The TFs, for example “Gata2”, “Gata1”, “Klf1”, “E2f1”, are known to play an important role in MEP. Using the network score, we may pick up cluster-specific key TFs.

In this step, however, we cannot know the specific function or relationship between cell fate. We just pick up several candidate genes for Key TFs here, and in the next analysis, we investigate their function and relationship between cell fate by the simulation with GRNs.

1.2.5 Simulation with GRNs

celloracle can simulate an affect of gene perturbation using GRNs. This analysis provides insight into the regulatory mechanism of cel state from a viewpoint of TF and GRN.

Python notebook

0. Import libraries

0.1. Import public 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
```

(continues on next page)

(continued from previous page)

```
import seaborn as sns
```

```
[3]: import celloracle as co
```

```
[15]: #plt.rcParams["font.family"] = "arial"
plt.rcParams["figure.figsize"] = [9, 6]
%config InlineBackend.figure_format = 'retina'
plt.rcParams["savefig.dpi"] = 600

%matplotlib inline
```

0.1. Make a folder to save graph

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

1. Load data

1.1. Load processed oracle object

Load the oracle object. See the previous notebook for the detail of how to prepare oracle object.

```
[7]: oracle = co.load_hdf5("../04_Network_analysis/Paul_15_data.celloracle.oracle")
```

1.2. Load inferred GRNs

In the previous notebook, we already calculated GRNs. We use this GRNs for simulation.

For the simulation with GRNs, we import GRNs which was saved in Links object.

```
[8]: links = co.load_hdf5("../04_Network_analysis/links.celloracle.links")
```

3. Make predictive models for simulation

We will construct regression models again by using the filtered GRN structure. This process take less time than the GRN inference in the previous notebook because we use only significant TFs to predict target gene rather than all regulatory candidate TFs.

```
[12]: 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)

calculating GRN using cluster specific TF dict...
calculating GRN in Ery_0
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))

genes_in_gem: 1999
models made for 1074 genes
calculating GRN in Ery_1
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999  
models made for 1092 genes  
calculating GRN in Ery_2
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999  
models made for 1064 genes  
calculating GRN in Ery_3
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999  
models made for 1105 genes  
calculating GRN in Ery_4
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999  
models made for 1102 genes  
calculating GRN in Ery_5
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999  
models made for 1116 genes  
calculating GRN in Ery_6
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999  
models made for 1097 genes  
calculating GRN in Ery_7
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999  
models made for 1062 genes  
calculating GRN in Ery_8
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999  
models made for 1117 genes  
calculating GRN in Ery_9
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999  
models made for 1121 genes  
calculating GRN in GMP_0
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999
models made for 1107 genes
calculating GRN in GMP_1
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999
models made for 1104 genes
calculating GRN in GMPl_0
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999
models made for 1089 genes
calculating GRN in Gran_0
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999
models made for 1067 genes
calculating GRN in Gran_1
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999
models made for 1076 genes
calculating GRN in Gran_2
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999
models made for 1105 genes
calculating GRN in MEP_0
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999
models made for 1152 genes
calculating GRN in Mk_0
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999
models made for 1114 genes
calculating GRN in Mo_0
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999
models made for 1085 genes
calculating GRN in Mo_1
```

```
HBox(children=(IntProgress(value=0, max=1999), HTML(value='')))
```

```
genes_in_gem: 1999
models made for 1074 genes
```

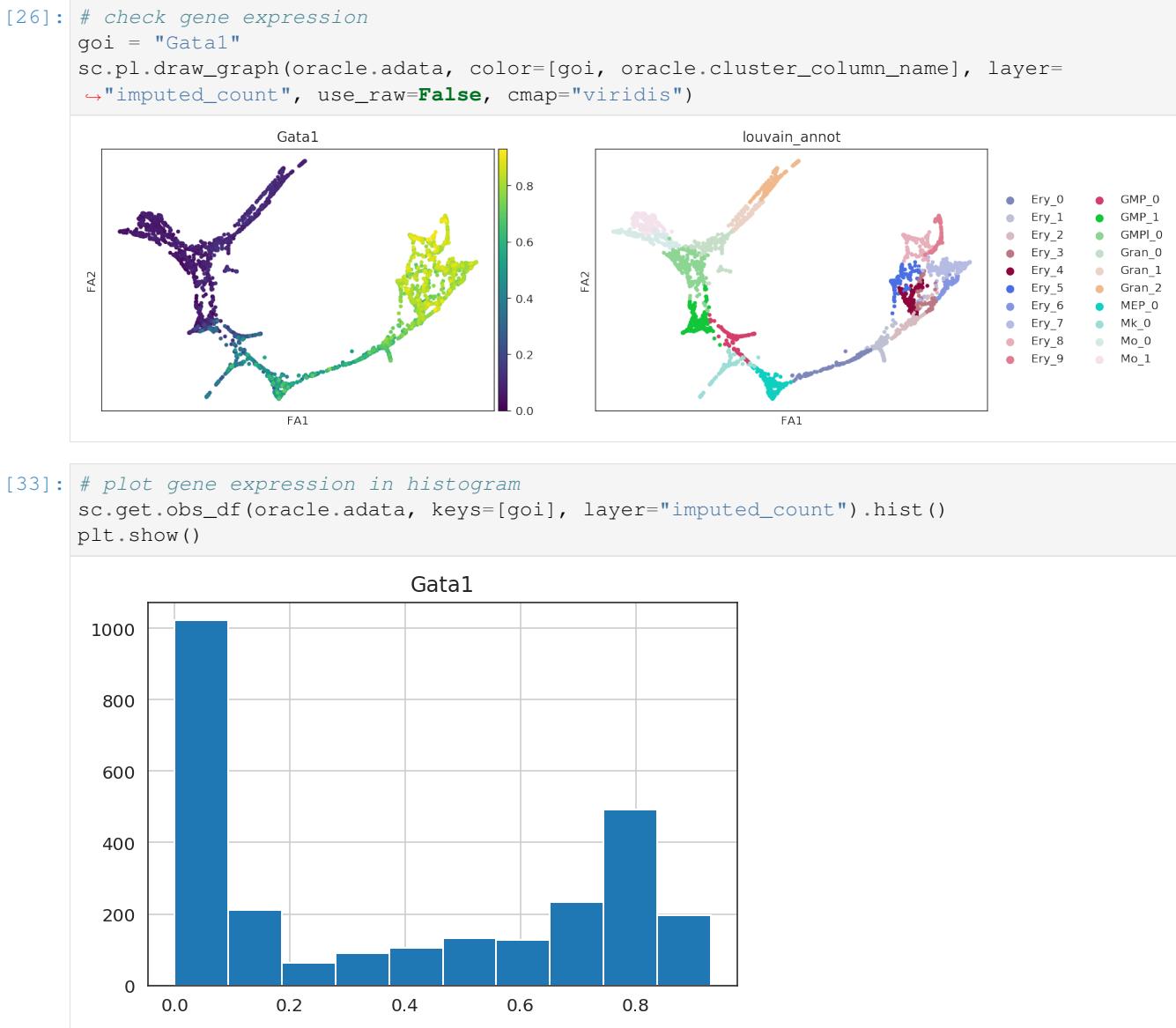
4. in silico Perturbation-simulation

Next, we will do simulation. Using signal propagation algorithm of celloracle, we can predict what happens if we perturb TF(s).

See the celloracle paper for the details.

In this notebook, we'll show an example of how to do in silico perturb simulation; we'll simulate Gata1 KO in the hematopoiesis. Previous study have shown that Gata1 is one of the important TFs that regulate cell fate decision in the myeloid progenitors and Gata1 have positive effect for the erythroid cell differentiation.

4.1. Check gene expression pattern.



4.1. calculate future gene expression after perturbation.

Although you can set any arbitrary gene expression value for perturb condition, we recommend avoiding too high value if you simulate overexpression. In the example below, if you set Gata1 gene expression to 100, for example, it

may lead to biologically infeasible results.

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

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

4.2. calculate transition probability between cells

In the step above, we simulated what happens if we perturb TF(s). This means we simulated future gene expression change after perturbation. This prediction is based on iterative calculations of signal propagations inside the GRN. Here we performed just small number of the calculation, thus the we simulated short future.

Next step, we will calculate the probability of cell state transition based on the simulated data. Using this cell state transition probability between cells, we can predict how cell state changes after perturbation, enabling the analysis of the relationship between genes and cell state determination.

This transition probability will be used in two ways.

1. Visualization of directed trajectory graph.
2. Markov simulation.

```
[35]: oracle.estimate_transition_prob(n_neighbors=200, knn_random=True, sampled_fraction=0.
                                   ↪5)
oracle.calculate_embedding_shift(sigma_corr = 0.05)
oracle.calculate_grid_arrows(smooth=0.8, steps=(40, 40), n_neighbors=300)

/home/k/anaconda3/envs/test/lib/python3.6/site-packages/IPython/core/interactiveshell.
py:3326: FutureWarning: arrays to stack must be passed as a "sequence" type such as
↪list or tuple. Support for non-sequence iterables such as generators is deprecated
↪as of NumPy 1.16 and will raise an error in the future.
exec(code_obj, self.user_global_ns, self.user_ns)
WARNING:root:Nans encountered in corrcoef and corrected to 1s. If not identical cells
↪were present it is probably a small isolated cluster converging after imputation.
```

4.3. Visualization

4.3.1. Detailed directed trajectory graph

```
[36]: plt.figure(None, (6, 6))
quiver_scale = 40

ix_choice = np.random.choice(oracle.adata.shape[0], size=int(oracle.adata.shape[0]/1.
                                                               ↪), replace=False)

embedding = oracle.adata.obsm[oracle.embedding_name]

plt.scatter(embedding[ix_choice, 0], embedding[ix_choice, 1],
            c="0.8", alpha=0.2, s=38, edgecolor=(0,0,0,1), lw=0.3, rasterized=True)

quiver_kw_args=dict(headaxislength=7, headlength=11, headwidth=8,
                     linewidths=0.35, width=0.0045, edgecolors="k",
                     color=oracle.colorandum[ix_choice], alpha=1)
plt.quiver(embedding[ix_choice, 0], embedding[ix_choice, 1],
```

(continues on next page)

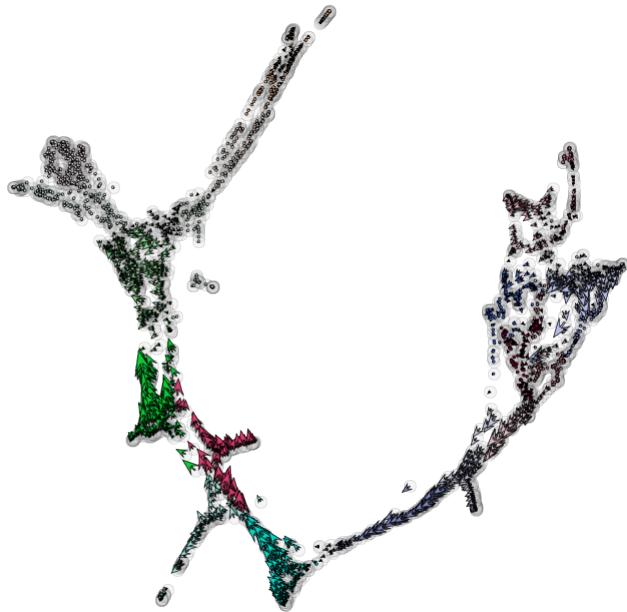
(continued from previous page)

```

        oracle.delta_embedding[ix_choice, 0], oracle.delta_embedding[ix_choice, 1],
        scale=quiver_scale, **quiver_kw_args)

plt.axis("off")
# plt.savefig(f"{save_folder}/full_arrows{goi}.png", transparent=True)
[36]: (-10815.27020913708, 10950.84121716522, -10711.36365432337, 10949.477199695968)

```

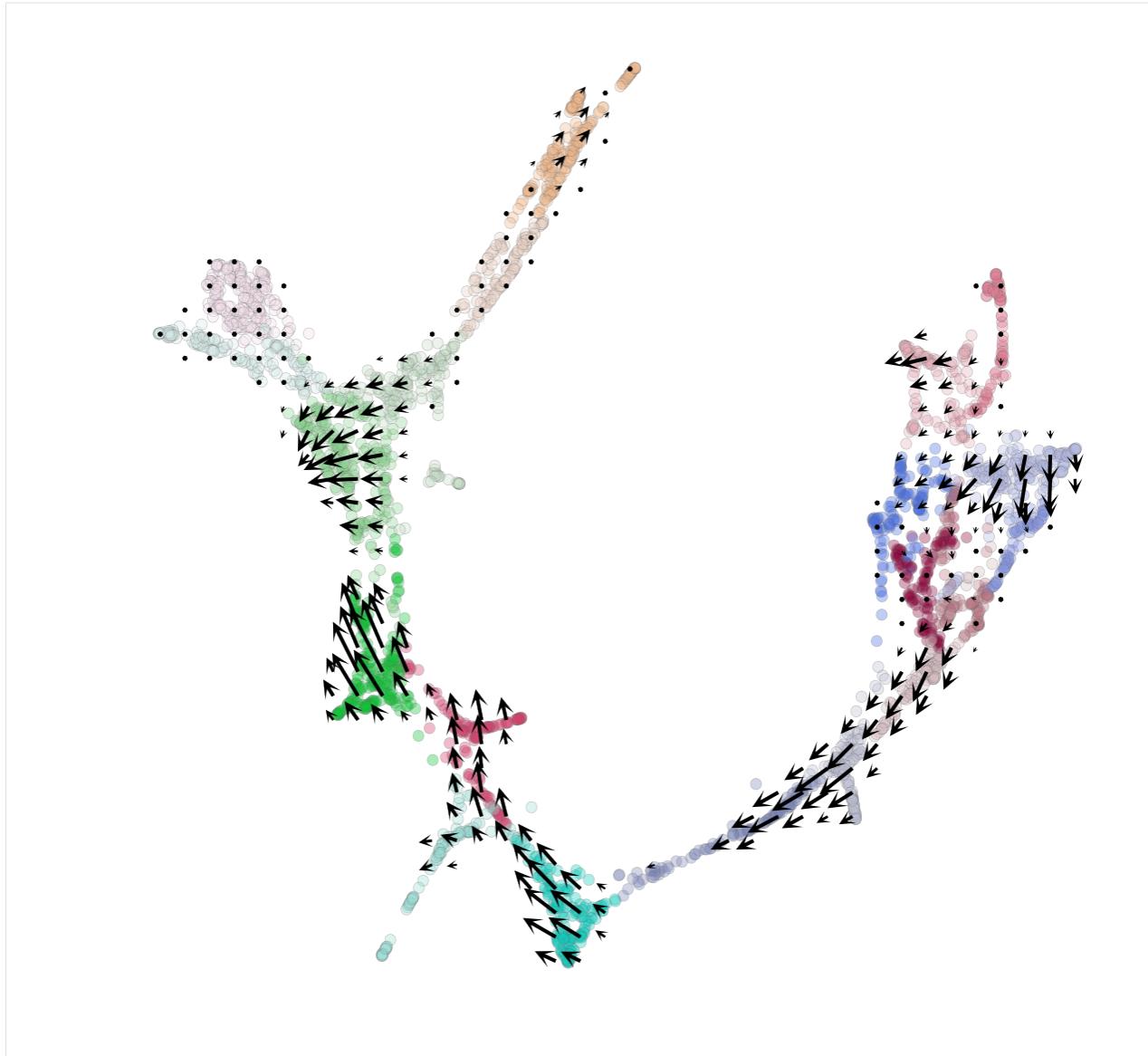


4.3.2. Grid graph

```

[37]: # plot whole graph
plt.figure(None, (10,10))
oracle.plot_grid_arrows(quiver_scale=2.0,
                        scatter_kw_args_dict={"alpha":0.35, "lw":0.35,
                                              "edgecolor":"0.4", "s":38,
                                              "rasterized":True},
                        min_mass=0.015, angles='xy', scale_units='xy',
                        headaxislength=2.75,
                        headlength=5, headwidth=4.8, minlength=1.5,
                        plot_random=False, scale_type="relative")
# plt.savefig(f"{save_folder}/vectorfield_{goi}.png", transparent=True)

```



4.4. Markov simulation to analyze the effects of perturbation on cell fate transition

Based on the transition probability, markov simulation will be performed to simulate cell state transition.

4.4.1. Do simulation

We simulate with a parameter, “n_steps=200” and in the following example. It means that

1. we will do 200 times of iterative simulations to predict how cell changes over time
2. and we will do 5 round of simulations for each cells.

```
[83]: %%time
# n_steps is the number of steps in markov simulation.
# n_duplication is the number of technical duplicate for the simulation
oracle.run_markov_chain_simulation(n_steps=200, n_duplication=5)
```

```
CPU times: user 1.33 s, sys: 0 ns, total: 1.33 s
Wall time: 1.33 s
```

4.4.2. Check the results of simulation focusing on a specific cells

Check the results of simulation by picking some cells and visualize their transition trajectory.

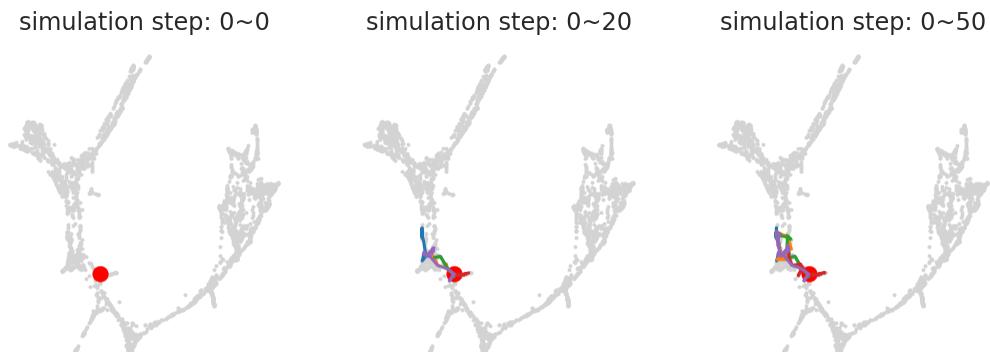
```
[88]: #np.random.seed(12)
# randomly pick up 3 cells
cells = oracle.adata.obs.index.values[np.random.choice(oracle.ixs_mcmc, 3)]

# visualize the simulated results of cell transition after perturbation
for k in cells:
    print(f"cell {k}")
    plt.figure(figsize=[9, 3])
    for j, i in enumerate([0, 20, 50]): # time points
        plt.subplot(1, 3, (j+1))
        oracle.plot_mc_result_as_trajectory(k, range(0, i))
        plt.title(f"simulation step: 0~{i}")
        plt.axis("off")
    plt.show()
```

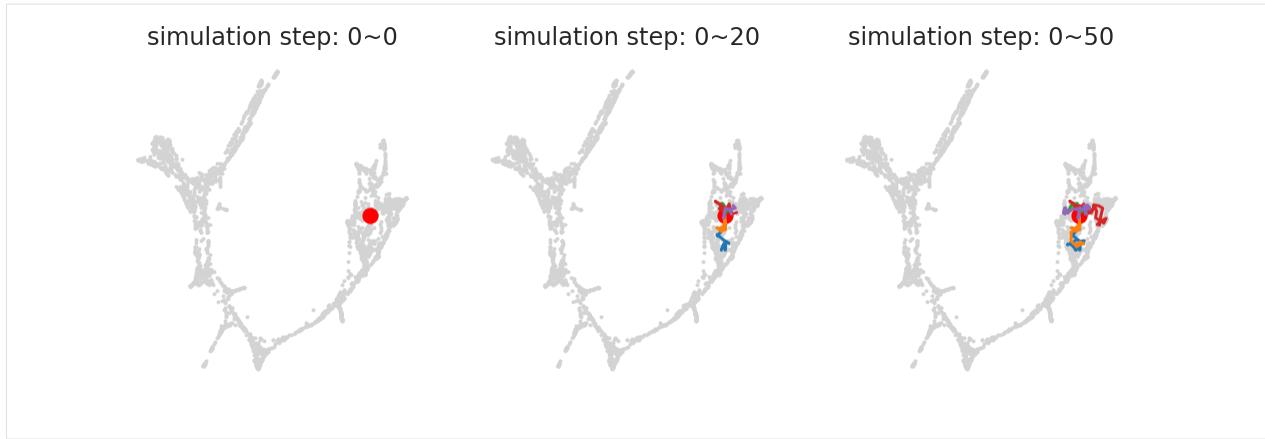
cell 1961



cell 43



cell 1567

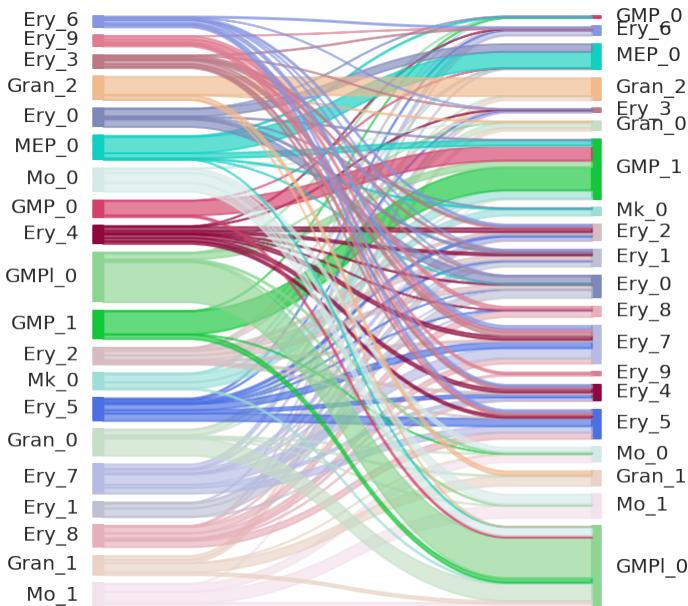


4.4.3. Summarize the results of simulation by plotting sankey diagram

Sankey diagram is useful when you want to visualize cell transition between some groups.

`plot_mc_resutls_as_sankey` function can make a Sankey diagram with any arbitrary cluster information.

```
[89]: # plot sankey diagram
plt.figure(figsize=[5, 6])
cl = "louvain_annot"
oracle.plot_mc_resutls_as_sankey(cluster_use=cl, start=0, end=100)
```



The sankey diagram above looks messy because cluster order is random.

Change cluster order and make plot again

```
[90]:
```

(continues on next page)

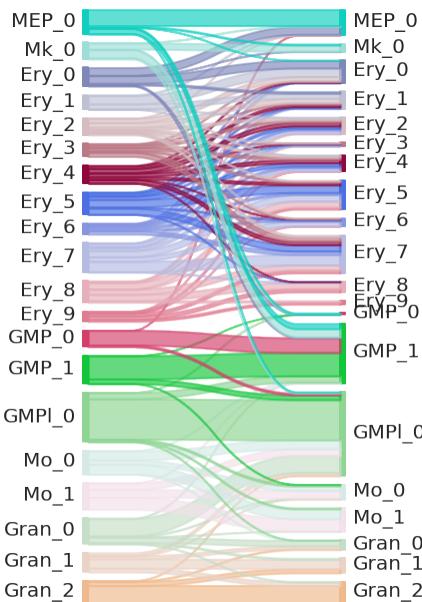
(continued from previous page)

```

cl = "louvain_annotation"
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', 'GMPI_0', 'GMPI_1',
         'Mo_0', 'Mo_1', 'Mo_2', 'Gran_0', 'Gran_1', 'Gran_2', 'Gran_3']

plt.figure(figsize=[5, 6])
plt.subplots_adjust(left=0.3, right=0.7)
oracle.plot_mc_results_as_sankey(cluster_use=cl, start=0, end=100, order=order)
# plt.savefig(f"{save_folder}/mcmc_{cl}.png")

```



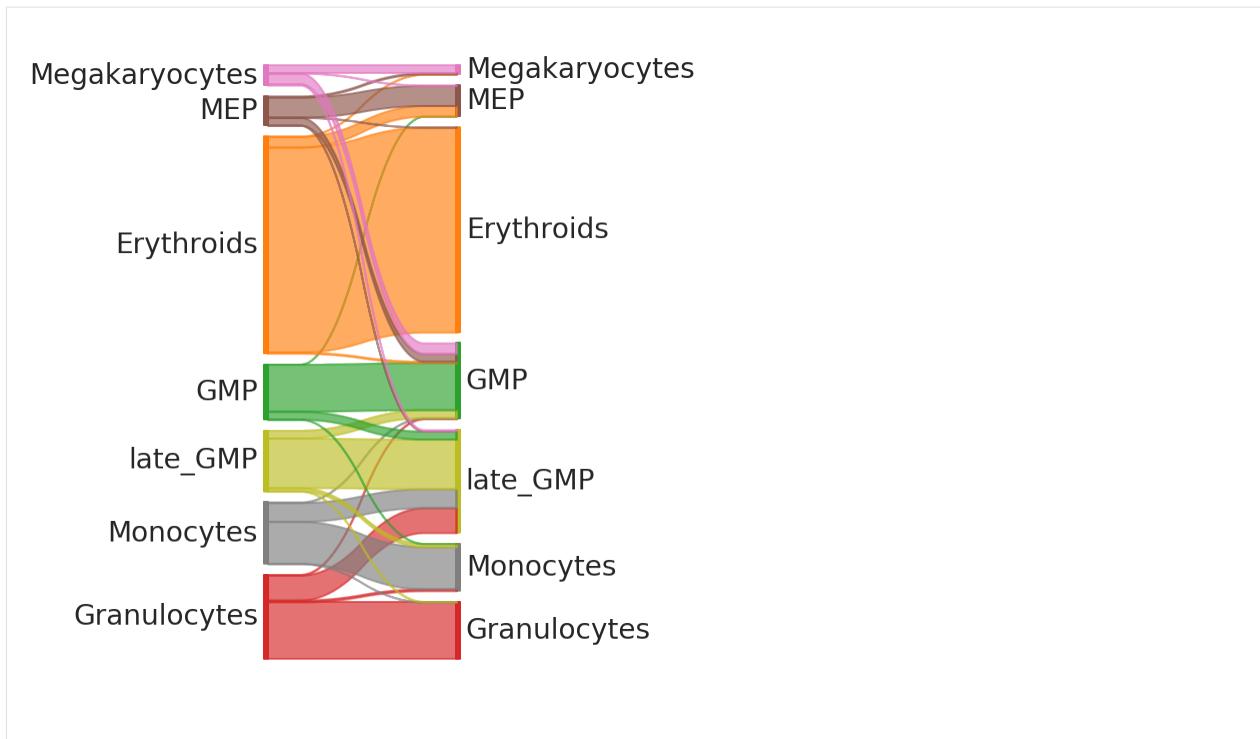
Make another sankey diagram with different cluster unit.

```

[92]: order = ['Megakaryocytes', 'MEP', 'Erythroblasts', 'GMP', 'late_GMP', 'Monocytes',
           'Granulocytes']
cl = "cell_type"

plt.figure(figsize=[5, 6])
plt.subplots_adjust(left=0.35, right=0.65)
oracle.plot_mc_results_as_sankey(cluster_use=cl, start=0, end=100, order=order, font_
size=14)
# plt.savefig(f"{save_folder}/mcmc_{cl}{goi}.png", transparent=True)

```



In the example above, we may conclude several things as follows.

Gata1 KO induces cell state transition from Erythrocytes to MEP, from MEP to GMP. These results suggest that Gata1 may play important roles in the progression of Erythroid differentiation and cell state determination between MEP and GMP lineage.

The data also show that Gata1 KO may induce cell state transition from granulocytes to late GMP, suggesting Gata1's involvement in Granulocytoid differentiation.

These results agree with previous reports about Gata1. We could recapitulate Gata1's cell-type-specific function regarding the cell fate decisions in hematopoiesis.

[]:

1.3 API

1.3.1 Command Line API

CellOracle have one 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 produced by this command can be used for an input of celloracle. Please go to step3 and step4 of the tutorial to see an example.

```
seuratToAnndata YOUR_SEURAT_OBJECT.Rds OUTPUT_PATH
```

1.3.2 Python API

Custom class in celloracle

We define some custom classes in celloracle.

```
class celloracle.Oracle
Bases:                                     celloracle.trajectory.modified_VelocytoLoom_class.
modified_VelocytoLoom
```

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 future gene expression patterns and cell state transition after perturbations of TFs. Please see the paper of CellOracle for details.

The code of the Oracle class was made of three components below.

(1) Anndata: Gene expression matrix and metadata in single-cell RNA-seq are stored in 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 process several data to infer GRN. See the documentation of Net class for detail.

(3) VelocytoLoom: Calculation of transition probability and visualization of directed trajectory graph will be performed in the same way as velocytoloom. VelocytoLoom is class for the Velocyto, which is a python library for RNA-velocity analysis. In the celloracle, we use almost the same functions as velocytoloom for the visualization, but celloracle use simulated gene expression values instead of RNA-velocity data.

Some CellOracle's methods were inspired by velocyto analysis and some codes were made by modifying VelocytoLoom class.

adata

anndata – Imported anndata object

cluster_column_name

str – The column name in adata.obs about cluster info

embedding_name

str – The key name in adata.obsm about dimensional reduction coordinates

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.

copy ()

Deepcopy itself.

fit_GRN_for_simulation (GRN_unit='cluster', alpha=1, use_cluster_specific_TFdict=False)

Do GRN inference. Please see the paper of CellOracle for details.

GRN can be constructed at an arbitrary cell group. If you want to infer cluster-specific GRN, please set [GRN_unit="cluster"]. GRN will be inferred for each cluster. You can select Cluster information when you import data (not when you run this method.).

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 increase, and you can detect a weak network connection, but it might get more noise. With a higher value of alpha may reduce the chance of overfitting.

get_cluster_specific_TFdict_from_Links (links_object)

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)
```

Make 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 increase, and you can detect a weak network connection, but it might get more noise. With a higher value of alpha may reduce the chance of overfitting.
- **bagging_number** (*int*) – The number for 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.

```
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. See the tutorial of celloracle or motif_analysis module for an example to make such files.

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)
```

Load scRNA-seq data. scRNA-seq data should be prepared as an anndata. Preprocessing (cell and gene filtering, calculate DR and cluster, etc.) should be done before loading data. The method will import NORMALIZED and LOG TRANSFORMED but NOT SCALED and NOT CENTERED DATA. See the tutorial for the details for an example of how to process scRNA-seq data.

Parameters

- **adata** (*anndata*) – anndata object that store scRNA-seq data.
- **cluster_column_name** (*str*) – the column name about cluster info in anndata.obs. Clustering data suppose to be in anndata.obs.
- **embedding_name** (*str*) – the key name about a dimensional reduction 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. Preprocessing (cell and gene filtering, calculate DR and cluster, 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 column name about cluster info in anndata.obs. Clustering data suppose to be in anndata.obs.
- **embedding_name** (*str*) – the key name about a dimensional reduction 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 detail (<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 number in markov simulation
- **args** (*dictionary*) – dictionary for the arguments for matplotlib.pyplot.plot. See matplotlib documentation for detail (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 arbitrary 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 sankey-diagram.
- **font_size** (*int*) – Font size for cluster name label in 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*)

Do Markov simlation 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 for multiple times to get a robust outcome.

Parameters

- **n_steps** (*int*) – steps for Markov simulation. This value is equivalent to the time after perturbation.
- **n_duplication** (*int*) – the number for multiple calculations.

simulate_shift (*perturb_condition=None*, *GRN_unit='whole'*, *n_propagation=3*)

Simulate signal propagation with GRNs. Please see the paper of CellOracle for details. This function simulates a gene expression pattern in the near future. Simulated values will be stored in `anndata.layers`: [“simulated_count”]

Three data below are used for the simulation. (1) GRN inference results (`coef_matrix`). (2) `perturb_condition`: You can set arbitrary perturbation condition. (3) gene expression matrix: 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 unfeasible values for the perturb condition. It is strongly recommended to check actual gene expression values in your data before selecting perturb condition.
- **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.

summarize_mc_results_by_cluster (*cluster_use*)

This function summarizes the simulated cell state-transition by groping the results into each cluster. It returns summarized 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 existed in the old TF dictionary, old values will be replaced with a new one.

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

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 – Dictionary that store unprocessed network data.

filtered_links

dictionary – Dictionary that store filtered network data.

merged_score

pandas.dataframe – Network scores.

cluster

list of str – List of cluster name.

name

str – Name of clustering unit.

palette

pandas.DataFrame – DataFrame that store color information.

add_palette (*cluster_name_list*, *color_list*)

Add color information for each cluster.

Parameters

- **cluster_name_list** (*list of str*) – cluster names.
- **color_list** (*list of str*) – list of color for each cluster.

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

Filter network edges. In most case inferred GRN has non-significant random edges. We have to remove such random edges before analyzing network structure. You can do the filtering either of three ways below.

1. Do filtering based on the p-value of the network edge. Please enter p-value for thresholding.
2. Do filtering 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 have several types of network weight, so you have to select which network weight do you want to use.
3. Do filtering 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. Please select a type of network weight for the entropy calculation.

Parameters value (*str*) – Default is “coef_abs”.

get_score (*test_mode=False*)

Get several network scores using R libraries. Make sure all dependant 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=False*, *kde=True*, *auto_gene_annot=False*, *percentile=98*, *args_dot={}*, *args_line={}*, *args_annot={}*, *save=None*)

Make a plot of gene network cartography. 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_analisis.Links` class for detail.
- **gois** (*list of str*) – 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)`

Plot the summary of gene network cartography 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_analisis.Links class for detail.
- **gois** (*list of str*) – 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 distribution of network degree (the number of edge per gene). The network degree will be visualized in both linear scale and log scale.

Parameters

- **links** ([Links](#)) – See network_analisis.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 of network entropy. See the CellOracle paper for the detail.

Parameters

- **links** (*Links object*) – See network_analisis.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)`

Make a scatter plot that shows the relationship of a specific network score in two groups.

Parameters

- **links** ([Links](#)) – See network_analisis.Links class for detail.
- **value** (*srt*) – The network score type.
- **cluster1** (*str*) – Cluster name. Network scores in the cluster1 will be visualized in the x-axis.
- **cluster2** (*str*) – Cluster name. Network scores in the 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_distributions (*values=None, method='boxplot', save=None*)

Plot the distribution of network scores. An individual data point is a network edge (gene) of GRN in each cluster.

Parameters

- **links** ([Links](#)) – See Links class for detail.
- **values** (*list of str*) – The list of score to visualize. If it is None, all network score will be used.
- **method** (*str*) – Plotting method. Select either of “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*)

Plot network score for a gene. This function visualizes the network score of a specific gene between clusters to get an insight into the dynamics of the gene.

Parameters

- **links** ([Links](#)) – See network_analisis.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 with high-network scores and make plots.

Parameters

- **links** ([Links](#)) – See network_analisis.Links class for detail.
- **cluster** (*str*) – Cluster nome 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 Oracle class for GRN inference. This class requires two 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 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 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 made from public ATAC-seq dataset of various tissue/cell type.

linkList

pandas.DataFrame – the result of GRN inference.

all_genes

numpy.array – an array of all genes that exist in input gene expression matrix

embedding_name

str – the key name name in adata.obsm about dimensional reduction coordinates

annotation

dictionary – annotation. you can add an arbitrary annotation.

coefs_dict

dictionary – coefs of linear regression.

stats_dict

dictionary – statistic values about coefs.

fitted_genes

list of str – list of genes with which regression model was successfully calculated.

failed_genes

list of str – list of genes that were not able to get coefs

cellstate

pandas.DataFrame – metadata for GRN input

TFinfo

pandas.DataFrame – information about potential regulatory TFs.

gem

pandas.DataFrame – merged matrix made with gene_expression_matrix and cellstate matrix.

gem_standerdized

pandas.DataFrame – almost the same as gem, but the gene_expression_matrix was standarized.

library_last_update_date

str – last update date of this code (this info is for code development. it will be deleted in the future)

object_initiation_date

str – the date when this object is made.

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-existing 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). The calculation usually takes a long time.

Parameters

- **bagging_number** (*int*) – The number of estimators for bagging.
- **scaling** (*bool*) – Whether to scale regulatory gene expression values.
- **model_method** (*str*) – ML model name. “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 to show a progress bar.

fit_All_genes_parallel (*bagging_number=200, scaling=True, log=None, verbose=10*)

!! This function has some bug now and 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 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. The calculation will be performed in parallel using scikit-learn bagging function. You can select a modeling method (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 to scale regulatory gene expression values.
- **model_method** (*str*) – ML model name. “bagging_ridge” or “bayesian_ridge”
- **save_coefs** (*bool*) – Whether 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 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 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 GRN.

Parameters **verbose** (*bool*) – Whether 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.

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.

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.

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.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.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.make_TFinfo_from_scanned_file` (*path_to_raw_bed*,
path_to_scanned_result_bed,
ref_genome)

This function is currently an available.

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 in GRN inference.

peak_df

pandas.DataFrame – dataframe about DNA peak and target gene data.

all_target_gene

array of str – target genes.

ref_genome

str – reference genome name that was used in DNA peak generation.

scanned_df

dictionary – Results of motif scan. Key is a peak name. Value is a dataframe of motif scan.

dic_targetgene2TFs

dictionary – Final product of Motif scan. Key is a target gene. Value is a list of regulatory candidate genes.

dic_peak2Targetgene
dictionary – Dictionary. Key is a peak name. Value is a list of the target gene.

dic_TF2targetgenes
dictionary – Final product of Motif scan. Key is a TF. Value is a list of potential target genes of the TF.

copy()
 Deepcopy itself.

filter_motifs_by_score (threshold, method='cumulative_score')
 Remove motifs with low binding scores.
Parameters **method** (*str*) – thresholding method. Select either of [“individual_score”, “cumulative_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)
 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.

to_dataframe (verbose=True)
 Return results as a datafram. 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 tutorial of celloracle documentation.

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

Modules for Network analysis

`celloracle.network_analysis module`

The `network_analysis` module implements Network analysis.

`celloracle.network_analysis.get_links (oracle_object, cluster_name_for_GRN_unit=None, alpha=10, bagging_number=20, verbose_level=1, test_mode=False)`

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 increase, and you can detect a weak network connection, but it might get more noise. With a higher value of alpha may reduce the chance of overfitting.

- **bagging_number** (*int*) – The number for 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.

`celloracle.network_analysis.test_R_libraries_installation()`

CellOracle.network_analysis use some R libraries for network analysis. This is a test function to check instalation of necessary R libraries.

`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`

`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 – Dictionary that store unprocessed network data.

filtered_links

dictionary – Dictionary that store filtered network data.

merged_score

pandas.DataFrame – Network scores.

cluster

list of str – List of cluster name.

name

str – Name of clustering unit.

palette

pandas.DataFrame – DataFrame that store color information.

`add_palette(cluster_name_list, color_list)`

Add color information for each cluster.

Parameters

- **cluster_name_list** (*list of str*) – cluster names.
- **color_list** (*list of str*) – list of color for each cluster.

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

Filter network edges. In most case inferred GRN has non-significant random edges. We have to remove such random edges before analyzing network structure. You can do the filtering either of three ways below.

1. Do filtering based on the p-value of the network edge. Please enter p-value for thresholding.
2. Do filtering 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 have several types of network weight, so you have to select which network weight do you want to use.

3. Do filtering 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. Please select a type of network weight for the entropy calculation.

Parameters **value** (*str*) – Default is “coef_abs”.

get_score (*test_mode=False*)

Get several network scores using R libraries. Make sure all dependant 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=False, *kde=True*,
auto_gene_annot=False, *percentile=98*, *args_dot={}*, *args_line={}*,
args_annot={}, *save=None*)

Make a plot of gene network cartography. 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_analisis.Links` class for detail.
- **gois** (*list of str*) – 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*)

Plot the summary of gene network cartography 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_analisis.Links` class for detail.
- **gois** (*list of str*) – 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 distribution of network degree (the number of edge per gene). The network degree will be visualized in both linear scale and log scale.

Parameters

- **links** ([Links](#)) – See network_analisis.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 of network entropy. See the CellOracle paper for the detail.

Parameters

- **links** (*Links object*) – See network_analisis.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,    an-  
                           not_shifts=None, save=None)
```

Make a scatter plot that shows the relationship of a specific network score in two groups.

Parameters

- **links** ([Links](#)) – See network_analisis.Links class for detail.
- **value** (*srt*) – The network score type.
- **cluster1** (*str*) – Cluster name. Network scores in the cluster1 will be visualized in the x-axis.
- **cluster2** (*str*) – Cluster name. Network scores in the 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_distributions(values=None, method='boxplot', save=None)
```

Plot the distribution of network scores. An individual data point is a network edge (gene) of GRN in each cluster.

Parameters

- **links** ([Links](#)) – See Links class for detail.
- **values** (*list of str*) – The list of score to visualize. If it is None, all network score will be used.
- **method** (*str*) – Plotting method. Select either of “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)
```

Plot network score for a gene. This function visualizes the network score of a specific gene between clusters to get an insight into the dynamics of the gene.

Parameters

- **links** ([Links](#)) – See network_analisis.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 with 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.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](#)) – links object
- **method** (*str*) – The method to summarize data.

`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 object.

Return type Graph object

`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 object.

Return type Graph object

Other modules

celloracle.go_analysis module

The [go_analysis](#) module implements Gene Ontology analysis. This module uses goatools internally.

`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 strcelloracle.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 strcelloracle.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.**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.**load_pickled_object** (*filepath*)

Load pickled object.

Parameters **filepath** (*str*) – file path.**Returns** loaded object.**Return type** python objectcelloracle.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** listcelloracle.utility.**exec_process** (*commands, message=True, wait_finished=True, turn_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 to wait for the process finished. If False, the function finish immediately.
- **return_process** (*bool*) – Whether to return “process”.

`celloracle.utility.standard(df)`

Standerdize value.

Parameters **df** (*pandas.DataFrame*) – dataframe.

Returns data after standerdization.

Return type pandas.DataFrame

`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.inverse_dictionary(dictionary, verbose=True, return_value_as_numpy=False)`

Make inversed dictionary. See examples below for detail.

Parameters

- **dictionary** (*dict*) – python dictionary
- **verbose** (*bool*) – Whether to show progress bar.
- **return_value_as_numpy** (*bool*) – Whether to convert values into numpy array.

Returns Python dictionary.

Return type dict

Examples

```
>>> dic = {"a": [1, 2, 3], "b": [2, 3, 4]}
>>> inverse_dictionary(dic)
{1: ['a'], 2: ['a', 'b'], 3: ['a', 'b'], 4: ['b']}
```

```
>>> dic = {"a": [1, 2, 3], "b": [2, 3, 4]}
>>> inverse_dictionary(dic, return_value_as_numpy=True)
{1: array(['a'], dtype='<U1'),
 2: array(['a', 'b'], dtype='<U1'),
 3: array(['a', 'b'], dtype='<U1'),
 4: array(['b'], dtype='<U1')}
```

celloracle.data module

The `data` module implements data download and loading.

`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_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

1.4 Changelog

1.5 License

The software is provided under xxxxxxxx License.

```
License xxxxxxxx
```

```
Copyright (c) 2019 Kenji Kamimoto, Christy Hoffmann, Samantha Morris
```

```
This is a placeholder. License description will be shown here.
```

1.6 Authors and citations

1.6.1 Cite celloracle

If you use velocyto please cite our bioarxiv preprint TITLE.

1.6.2 celloracle software development

celloracle is developed and maintained by the effort of celloracle team in Samantha Morris Lab. Please post troubles or questions on [the Github repository](#)

**CHAPTER
TWO**

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

celloracle, 81
celloracle.data, 100
celloracle.data_conversion, 101
celloracle.go_analysis, 98
celloracle.motif_analysis, 91
celloracle.network_analysis, 94
celloracle.utility, 99

INDEX

A

adata (celloracle.Oracle attribute), 82
add_palette() (celloracle.Links method), 86
add_palette() (celloracle.network_analysis.Links method), 95
addAnnotation() (celloracle.Net method), 89
addTFinfo_dictionary() (celloracle.Net method), 90
addTFinfo_dictionary() (celloracle.Oracle method), 82
addTFinfo_matrix() (celloracle.Net method), 90
all_genes (celloracle.Net attribute), 89
all_target_gene (celloracle.motif_analysis.TFinfo attribute), 92
annotation (celloracle.Net attribute), 89

C

celloracle (module), 81
celloracle.data (module), 100
celloracle.data_conversion (module), 101
celloracle.go_analysis (module), 98
celloracle.motif_analysis (module), 91
celloracle.network_analysis (module), 94
celloracle.utility (module), 99
cellstate (celloracle.Net attribute), 89
cluster (celloracle.Links attribute), 85
cluster (celloracle.network_analysis.Links attribute), 95
cluster_column_name (celloracle.Oracle attribute), 82
coefs_dict (celloracle.Net attribute), 89
copy() (celloracle.motif_analysis.TFinfo method), 93
copy() (celloracle.Net method), 90
copy() (celloracle.Oracle method), 82

D

dic_peak2Targetgene (celloracle.motif_analysis.TFinfo attribute), 92
dic_targetgene2TFs (celloracle.motif_analysis.TFinfo attribute), 92
dic_TF2targetgenes (celloracle.motif_analysis.TFinfo attribute), 93
draw_network() (in module celloracle.network_analysis), 98

E

embedding_name (celloracle.Net attribute), 89
embedding_name (celloracle.Oracle attribute), 82
exec_process() (in module celloracle.utility), 99

F
failed_genes (celloracle.Net attribute), 89
filter_links() (celloracle.Links method), 86
filter_links() (celloracle.network_analysis.Links method), 95
filter_motifs_by_score() (celloracle.motif_analysis.TFinfo method), 93
filter_peaks() (celloracle.motif_analysis.TFinfo method), 93
filtered_links (celloracle.Links attribute), 85
filtered_links (celloracle.network_analysis.Links attribute), 95
fit_All_genes() (celloracle.Net method), 90
fit_All_genes_parallel() (celloracle.Net method), 90
fit_genes() (celloracle.Net method), 90
fit_GRN_for_simulation() (celloracle.Oracle method), 82
fitted_genes (celloracle.Net attribute), 89

G

gem (celloracle.Net attribute), 89
gem_standerdized (celloracle.Net attribute), 89
geneID2Symbol() (in module celloracle.go_analysis), 99
geneSymbol2ID() (in module celloracle.go_analysis), 98
get_cluster_specific_TFdict_from_Links() (celloracle.Oracle method), 82
get_GO() (in module celloracle.go_analysis), 99
get_links() (celloracle.Oracle method), 82
get_links() (in module celloracle.network_analysis), 94
get_network_entropy() (celloracle.Links method), 86
get_network_entropy() (celloracle.network_analysis.Links method), 96
get_score() (celloracle.Links method), 86
get_score() (celloracle.network_analysis.Links method), 96
get_tss_info() (in module celloracle.motif_analysis), 94

I

import_anndata_as_normalized_count() (celloracle.Oracle method), 83
import_anndata_as_raw_count() (celloracle.Oracle method), 83
import_TF_data() (celloracle.Oracle method), 83
info() (celloracle.utility.makelog method), 99
integrate_tss_peak_with_cicero() (in module celloracle.motif_analysis), 94
intersect() (in module celloracle.utility), 99
inverse_dictionary() (in module celloracle.utility), 100
is_genome_installed() (in module celloracle.motif_analysis), 91

L

library_last_update_date (celloracle.Net attribute), 89
linkList (celloracle.Net attribute), 89
linkList_to_networkgraph() (in module celloracle.network_analysis), 98
Links (class in celloracle), 85
Links (class in celloracle.network_analysis), 95
links_dict (celloracle.Links attribute), 85
links_dict (celloracle.network_analysis.Links attribute), 95
load_hdf5() (in module celloracle), 91
load_hdf5() (in module celloracle.utility), 100
load_links() (in module celloracle.network_analysis), 95
load_pickled_object() (in module celloracle.utility), 99
load_TFinfo_df_mm9_mouse_atac_atlas() (in module celloracle.data), 100
load_TFinfo_from_parquets() (in module celloracle.motif_analysis), 92

M

make_TFinfo_dataframe_and_dictionary() (celloracle.motif_analysis.TFinfo method), 93
make_TFinfo_from_scanned_file() (in module celloracle.motif_analysis), 92
makelog (class in celloracle.utility), 99
merged_score (celloracle.Links attribute), 85
merged_score (celloracle.network_analysis.Links attribute), 95

N

name (celloracle.Links attribute), 85
name (celloracle.network_analysis.Links attribute), 95
Net (class in celloracle), 88

O

object_initiation_date (celloracle.Net attribute), 89
Oracle (class in celloracle), 81

P

palette (celloracle.Links attribute), 86

palette (celloracle.network_analysis.Links attribute), 95
peak2fasta() (in module celloracle.motif_analysis), 92
peak_df (celloracle.motif_analysis.TFinfo attribute), 92
plot_cartography_scatter_per_cluster() (celloracle.Links method), 86
plot_cartography_scatter_per_cluster() (celloracle.network_analysis.Links method), 96
plot_cartography_term() (celloracle.Links method), 87
plot_cartography_term() (celloracle.network_analysis.Links method), 96
plot_degree_distributions() (celloracle.Links method), 87
plot_degree_distributions() (celloracle.network_analysis.Links method), 96
plot_mc_result_as_kde() (celloracle.Oracle method), 84
plot_mc_result_as_trajectory() (celloracle.Oracle method), 84
plot_mc_results_as_sankey() (celloracle.Oracle method), 84
plot_network_entropy_distributions() (celloracle.Links method), 87
plot_network_entropy_distributions() (celloracle.network_analysis.Links method), 97
plot_score_comparison_2D() (celloracle.Links method), 87
plot_score_comparison_2D() (celloracle.network_analysis.Links method), 97
plot_score_distributions() (celloracle.Links method), 88
plot_score_distributions() (celloracle.network_analysis.Links method), 97
plot_score_per_cluster() (celloracle.Links method), 88
plot_score_per_cluster() (celloracle.network_analysis.Links method), 97
plot_scores_as_rank() (celloracle.Links method), 88
plot_scores_as_rank() (celloracle.network_analysis.Links method), 98
plotCoefs() (celloracle.Net method), 91
prepare_markov_simulation() (celloracle.Oracle method), 84

R

read_bed() (in module celloracle.motif_analysis), 92
ref_genome (celloracle.motif_analysis.TFinfo attribute), 92
reset_dictionary_and_df() (celloracle.motif_analysis.TFinfo method), 93
reset_filtering() (celloracle.motif_analysis.TFinfo method), 93
run_markov_chain_simulation() (celloracle.Oracle method), 84

S

save_as_parquet() (celloracle.motif_analysis.TFinfo method), 93

save_as_pickled_object() (in module celloracle.utility),
 99
scan() (celloracle.motif_analysis.TFinfo method), 93
scanned_df (celloracle.motif_analysis.TFinfo attribute),
 92
seurat_object_to_anndata() (in module celloracle.
 data_conversion), 101
simulate_shift() (celloracle.Oracle method), 85
standard() (in module celloracle.utility), 100
stats_dict (celloracle.Net attribute), 89
summarize_mc_results_by_cluster() (celloracle.Oracle
 method), 85

T

test_R_libraries_installation() (in module celloracle.
 network_analysis), 95
TFinfo (celloracle.Net attribute), 89
TFinfo (class in celloracle.motif_analysis), 92
to_dataframe() (celloracle.motif_analysis.TFinfo
 method), 93
to_dictionary() (celloracle.motif_analysis.TFinfo
 method), 93
to_hdf5() (celloracle.Links method), 88
to_hdf5() (celloracle.motif_analysis.TFinfo method), 94
to_hdf5() (celloracle.Net method), 91
to_hdf5() (celloracle.network_analysis.Links method), 98
to_hdf5() (celloracle.Oracle method), 85
transfer_scores_from_links_to_adata() (in module celloracle.
 network_analysis), 98

U

updateLinkList() (celloracle.Net method), 91
updateTFinfo_dictionary() (celloracle.Net method), 91
updateTFinfo_dictionary() (celloracle.Oracle method), 85