uniPort

Release 1.2.1

Kai Cao

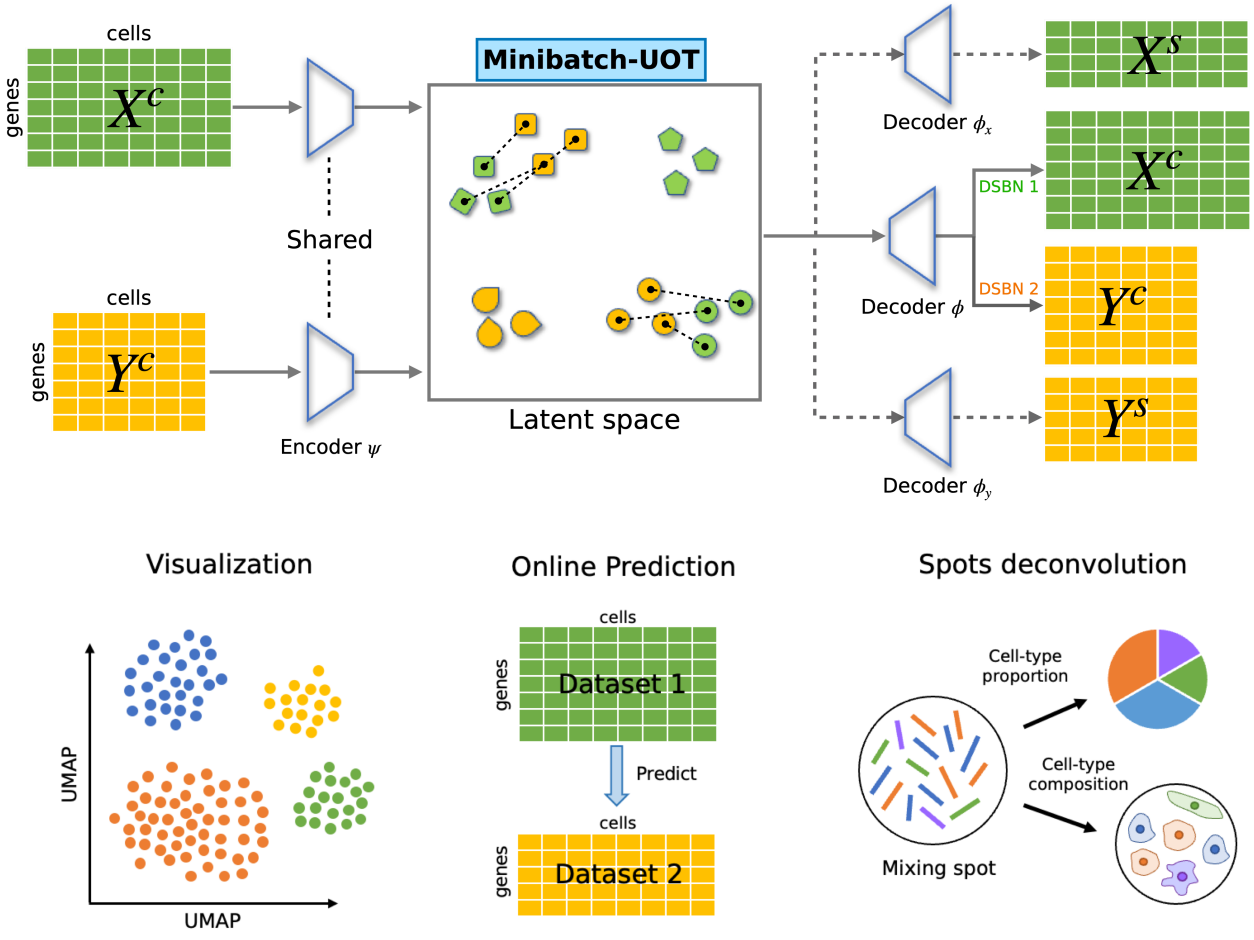
Mar 07, 2023

CONTENTS

CHAPTER

ONE

TUTORIALS



1.1 Installation

To use uniPort, first install it using pip:

```
pip3 install uniport
```

After a correct installation, you should be able to import the module without errors:

```
import uniport as up
```

We highly recommand training the model with Nvidia GPU devices, consider install Pytorch cuda version:

```
pip3 install torch torchvision torchaudio
```

1.2 Examples

1.2.1 Integrate scATAC and scRNA

PBMC data integration

We apply uniPort to integrate transcriptomic and epigenomic data using scATAC (gene activity matrix) and scRNA datasets profiled from peripheral blood mononuclear cells (PBMC), including 11259 paired cells with 19434 genes in scATAC and 26187 genes in scRNA.The PBMC data consists of paired scATAC-seq and scRNA-seq profiles, but we treat them as unpaired.

[12]:
```
import uniport as up
import numpy as np
import pandas as pd
import scanpy as sc
print(up.__version__)
```

```
1.1.1
```

Data preprocessing

Read cell types for both scATAC-seq and scRNA-seq

[13]:
```
labels = pd.read_csv('PBMC/meta.txt', sep='\t')
celltype = labels['cluster'].values
```

Read gene activity matrix and RNA counts into `AnnData` objects using `load_file` fucntion in uniport.

[14]:
```
adata_rna = up.load_file('PBMC/rna.h5ad')
adata_atac = up.load_file('PBMC/atac_meastro.h5ad')
print(adata_rna)
print(adata_atac)
```

```
AnnData object with n_obs × n_vars = 11259 × 11942
    obs: 'cell_type', 'domain_id', 'source', 'n_genes'
    var: 'n_cells'
AnnData object with n_obs × n_vars = 11259 × 28307
```

Add 'cell_type', 'domain_id' and 'source' to the `AnnData`objects.
'cell_type' stores cell label annotations.
'domain_id' identifies the modality using a number category.
'source' identifies the modality using its name.

[15]:
```
adata_atac.obs['cell_type'] = celltype
adata_atac.obs['domain_id'] = 0
adata_atac.obs['domain_id'] = adata_atac.obs['domain_id'].astype('category')
adata_atac.obs['source'] = 'ATAC'

adata_rna.obs['cell_type'] = celltype
adata_rna.obs['domain_id'] = 1
adata_rna.obs['domain_id'] = adata_rna.obs['domain_id'].astype('category')
adata_rna.obs['source'] = 'RNA'
```

Filter cells and features using `filter_data` function in uniport.

[16]:
```
# up.filter_data(adata_atac, min_features=3, min_cells=200)
# up.filter_data(adata_rna, min_features=3, min_cells=200)
# print(adata_atac)
# print(adata_rna)
```

Concatenate scATAC-seq and scRNA-seq with common genes using `AnnData.concatenate`.

[17]:
```
adata_cm = adata_atac.concatenate(adata_rna, join='inner', batch_key='domain_id')
```

Preprocess data with common genes. Select 2,000 highly variable common genes.
Scale data using `batch_scale` function in uniport (modified from SCALEX).

[18]:
```
# sc.pp.highly_variable_genes(adata_cm, n_top_genes=2000, flavor="seurat_v3")
sc.pp.normalize_total(adata_cm)
sc.pp.log1p(adata_cm)
sc.pp.highly_variable_genes(adata_cm, n_top_genes=2000, inplace=False, subset=True)
up.batch_scale(adata_cm)
# sc.pp.scale(adata_cm)
print(adata_cm.obs)
```

```
                     cell_type domain_id source  n_genes
AAACAGCCAAGGAATC.1-0  CD4 Naive         0   ATAC      NaN
AAACAGCCAATCCCTT.1-0   CD4 Tmem         0   ATAC      NaN
AAACAGCCAATGCGCT.1-0  CD4 Naive         0   ATAC      NaN
AAACAGCCACACTAAT.1-0  CD8 Naive         0   ATAC      NaN
AAACAGCCACCAACCG.1-0  CD8 Naive         0   ATAC      NaN
...                         ...       ...    ...      ...
TTTGTTGGTGACATGC.1-1  CD8 Naive         1    RNA   1586.0
TTTGTTGGTGTTAAAC.1-1  CD8 Naive         1    RNA   1525.0
TTTGTTGGTTAGGATT.1-1         NK         1    RNA   2024.0
TTTGTTGGTTGGTTAG.1-1   CD4 Tmem         1    RNA   1620.0
TTTGTTGGTTTGCAGA.1-1   CD8 Tmem         1    RNA   1920.0
```

(continues on next page)

(continued from previous page)

```
[22518 rows x 4 columns]
```

Preprocess scRNA-seq data. Select 2,000 highly variable genes as RNA specific.

[19]:
```
# sc.pp.highly_variable_genes(adata_rna, n_top_genes=2000, flavor="seurat_v3")
sc.pp.normalize_total(adata_rna)
sc.pp.log1p(adata_rna)
sc.pp.highly_variable_genes(adata_rna, n_top_genes=2000, inplace=False, subset=True)
up.batch_scale(adata_rna)
# sc.pp.scale(adata_rna)
```

Preprocess scATAC-seq data. Select 2,000 highly variable genes as ATAC speicifc.

[20]:
```
# sc.pp.highly_variable_genes(adata_atac, n_top_genes=2000, flavor="seurat_v3")
sc.pp.normalize_total(adata_atac)
sc.pp.log1p(adata_atac)
sc.pp.highly_variable_genes(adata_atac, n_top_genes=2000, inplace=False, subset=True)
up.batch_scale(adata_atac)
# sc.pp.scale(adata_atac)
```

Save the preprocessed data for integration.

Integration with specific genes and optimal transport loss

Integrate the scATAC-seq and scRNA-seq data using both common and dataset-specific genes by `Run` function in uniport. The latent representations of data are stored in `adata.obs['latent']`.

[21]:
```
adata = up.Run(adatas=[adata_atac,adata_rna], adata_cm=adata_cm, lambda_s=1.0)
```

```
Device: cuda
Dataset 0: ATAC
AnnData object with n_obs × n_vars = 11259 × 2000
    obs: 'cell_type', 'domain_id', 'source'
    var: 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
Dataset 1: RNA
AnnData object with n_obs × n_vars = 11259 × 2000
    obs: 'cell_type', 'domain_id', 'source', 'n_genes'
    var: 'n_cells', 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
Reference dataset is dataset 1


Data with common HVG
AnnData object with n_obs × n_vars = 22518 × 2000
    obs: 'cell_type', 'domain_id', 'source', 'n_genes'
    var: 'n_cells-1', 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
```

```
Epochs: 100%|| 345/345 [10:09<00:00,  1.77s/it, recloss=1148.08,klloss=9.01,otloss=6.87]
```

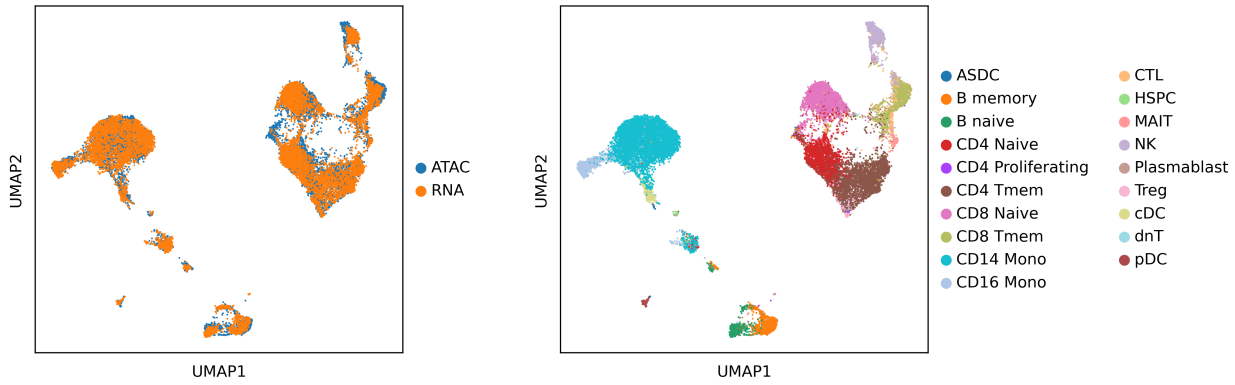
Before integration. Visualize the data using UMAP according to their cell types and sources.

After integration

[22]:
```
sc.set_figure_params(dpi=200, fontsize=10)
sc.pp.neighbors(adata, use_rep='latent')
sc.tl.umap(adata, min_dist=0.1)
sc.pl.umap(adata, color=['source', 'cell_type'], save='uniport-pbmc.pdf', title=['',''],␣
↪wspace=0.3, legend_fontsize=10)
```

```
... storing 'cell_type' as categorical
... storing 'source' as categorical
```

```
WARNING: saving figure to file figures/umapuniport-pbmc.pdf
```



Evaluate the results with various scores

We evaluated the results by F1, ARI, NMI, Batch Entropy and Silhouette scores.

[23]:
```
adata1 = adata[adata.obs['domain_id']=='0']
adata2 = adata[adata.obs['domain_id']=='1']
y_test = up.metrics.label_transfer(adata2, adata1, label='cell_type', rep='X_umap')
from sklearn.metrics import adjusted_rand_score, normalized_mutual_info_score, f1_score
print('F1:', f1_score(adata1.obs['cell_type'], y_test, average='micro'))
print('ARI:', adjusted_rand_score(adata1.obs['cell_type'], y_test))
print('NMI:', normalized_mutual_info_score(adata1.obs['cell_type'], y_test))
print('Batch Entropy:', up.metrics.batch_entropy_mixing_score(adata.obsm['X_umap'],␣
↪adata.obs['domain_id']))
print('Silhouette:', up.metrics.silhouette(adata.obsm['X_umap'], adata.obs['cell_type']))
```

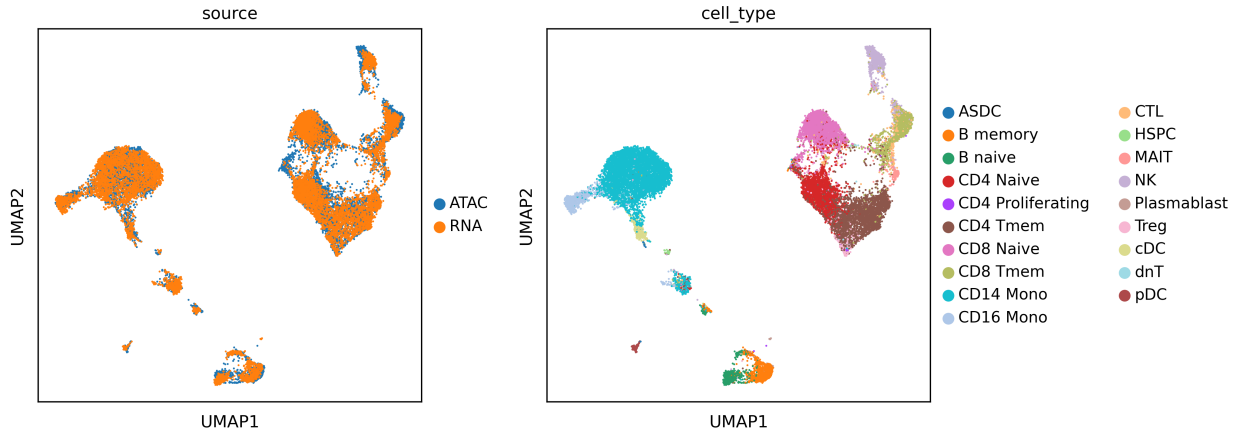
```
F1: 0.8490984989785949
ARI: 0.7657955254910905
NMI: 0.7411449325196692
Batch Entropy: 0.6158191095057443
Silhouette: 0.6382090449333191
```

Project data after training

To project data into the latent space without training, we can set `out='project'`.

[24]:
```
adata = up.Run(adata_cm=adata_cm, out='project')
sc.pp.neighbors(adata, use_rep='project')
sc.tl.umap(adata, min_dist=0.1)
sc.pl.umap(adata, color=['source', 'cell_type'])
```

```
Device: cuda
```



Mouse spleen integration

We'll apply uniPort to integrate transcriptomic and epigenomic data using scATAC (gene activity matrix) and scRNA datasets profiled from mouse spleen, including 3166 cells with 19410 genes in scATAC and 4382 genes with 13575 genes in scRNA.

[1]:
```
import uniport as up
import scanpy as sc
up.__version__
```

[1]:
```
'1.1.1'
```

The ATAC data was generated and annotated in Chen et al., 2018, while the RNA data was generated as part of the MultiMAP and annotated to match.

We downloaded the Anndata from ftp://ngs.sanger.ac.uk/production/teichmann/MultiMAP.

[2]:
```
rna = sc.read('mouse_spleen/rna.h5ad', backup_url='ftp://ngs.sanger.ac.uk/production/
↪teichmann/MultiMAP/rna.h5ad')
atac_genes = sc.read('mouse_spleen/atac-genes.h5ad', backup_url='ftp://ngs.sanger.ac.uk/
↪production/teichmann/MultiMAP/atac-genes.h5ad')
```

[3]:
```
print(atac_genes)
print(rna)
rna_process = rna.copy()
atac_process = atac_genes.copy()
```

```
AnnData object with n_obs × n_vars = 3166 × 19410
    obs: 'cell_type', 'source'
AnnData object with n_obs × n_vars = 4382 × 13575
    obs: 'cell_type', 'source'
```

Add 'domain_id' to the `AnnData` objects. 'domain_id' identifies the modality using a number category.

[4]:
```
atac_process.obs['domain_id'] = 0
atac_process.obs['domain_id'] = atac_process.obs['domain_id'].astype('category')
rna_process.obs['domain_id'] = 1
rna_process.obs['domain_id'] = rna_process.obs['domain_id'].astype('category')
```

Filter cells and features using `filter_data` function in uniport.

[5]:
```
up.filter_data(atac_process, min_features=3, min_cells=200)
up.filter_data(rna_process, min_features=3, min_cells=200)
print(atac_process)
print(rna_process)
```

```
AnnData object with n_obs × n_vars = 3166 × 14331
    obs: 'cell_type', 'source', 'domain_id', 'n_genes'
    var: 'n_cells'
AnnData object with n_obs × n_vars = 4382 × 5931
    obs: 'cell_type', 'source', 'domain_id', 'n_genes'
    var: 'n_cells'
```

Concatenate scATAC-seq and scRNA-seq with common genes using `AnnData.concatenate`.

[6]:
```
adata_cm = atac_process.concatenate(rna_process, join='inner', batch_key='domain_id')
print(adata_cm)
```

```
AnnData object with n_obs × n_vars = 7548 × 4877
    obs: 'cell_type', 'source', 'domain_id', 'n_genes'
    var: 'n_cells-0', 'n_cells-1'
```

Preprocess data with common genes. Select 2,000 highly variable common genes.
Scale data using `batch_scale` function in uniport (modified from SCALEX).

[7]:
```
sc.pp.normalize_total(adata_cm)
sc.pp.log1p(adata_cm)
sc.pp.highly_variable_genes(adata_cm, n_top_genes=2000, batch_key='domain_id',␣
↪inplace=False, subset=True)
up.batch_scale(adata_cm)
print(adata_cm)
```

```
... storing 'cell_type' as categorical
... storing 'source' as categorical
```

```
AnnData object with n_obs × n_vars = 7548 × 2000
    obs: 'cell_type', 'source', 'domain_id', 'n_genes'
    var: 'n_cells-0', 'n_cells-1', 'highly_variable', 'means', 'dispersions',
↪'dispersions_norm', 'highly_variable_nbatches', 'highly_variable_intersection'
    uns: 'log1p', 'hvg'
```

Preprocess scATAC-seq data. Select 2,000 highly variable genes as ATAC speicifc.

[8]:
```
sc.pp.normalize_total(atac_process)
sc.pp.log1p(atac_process)
sc.pp.highly_variable_genes(atac_process, n_top_genes=2000, inplace=False, subset=True)
up.batch_scale(atac_process)
print(atac_process)
```

```
AnnData object with n_obs × n_vars = 3166 × 2000
    obs: 'cell_type', 'source', 'domain_id', 'n_genes'
    var: 'n_cells', 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
```

Preprocess scRNA-seq data. Select 2,000 highly variable genes as ATAC speicifc.

[9]:
```
sc.pp.normalize_total(rna_process)
sc.pp.log1p(rna_process)
sc.pp.highly_variable_genes(rna_process, n_top_genes=2000, inplace=False, subset=True)
up.batch_scale(rna_process)
print(rna_process)
```

```
AnnData object with n_obs × n_vars = 4382 × 2000
    obs: 'cell_type', 'source', 'domain_id', 'n_genes'
    var: 'n_cells', 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
```

Save the preprocessed data for integration.

Visualize the data using UMAP according to their cell types and sources.

Read the preprocessed data. Integrate the scATAC-seq and scRNA-seq data using both common and dataset-specific genes by `Run` function in uniport. The latent representations of data are stored in adata.obs['latent'].

[10]:
```
adatas = [atac_process, rna_process]
adata = up.Run(adatas=adatas, adata_cm=adata_cm, lr=0.001, iteration=10000)
```

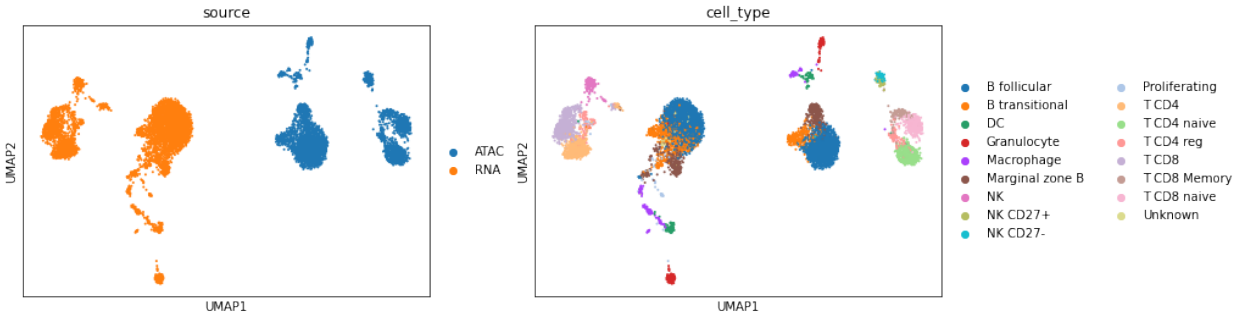
```
Dataset 0: ATAC
AnnData object with n_obs × n_vars = 3166 × 2000
    obs: 'cell_type', 'source', 'domain_id', 'n_genes'
    var: 'n_cells', 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
Dataset 1: RNA
AnnData object with n_obs × n_vars = 4382 × 2000
    obs: 'cell_type', 'source', 'domain_id', 'n_genes'
    var: 'n_cells', 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
Reference dataset is dataset 1


Data with common HVG
AnnData object with n_obs × n_vars = 7548 × 2000
    obs: 'cell_type', 'source', 'domain_id', 'n_genes'
    var: 'n_cells-0', 'n_cells-1', 'highly_variable', 'means', 'dispersions',
↪'dispersions_norm', 'highly_variable_nbatches', 'highly_variable_intersection'
    uns: 'log1p', 'hvg'
```

```
Epochs:  81%|      | 278/345 [04:07<00:59,  1.12it/s, recloss=1374.99,klloss=8.79,
↪otloss=7.70]
EarlyStopping: run 279 epoch
```

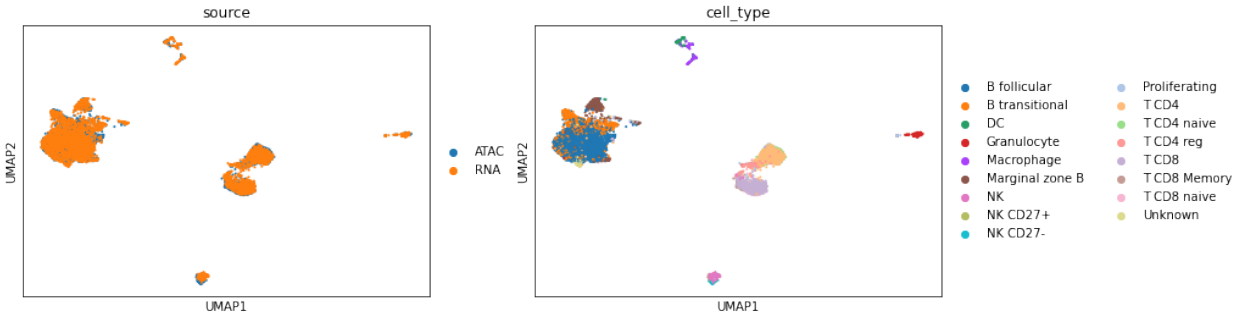
Before integration. Visualize the data using UMAP according to their cell types and sources.

[11]:
```
sc.pp.pca(adata_cm)
sc.pp.neighbors(adata_cm)
sc.tl.umap(adata_cm)
sc.pl.umap(adata_cm, color=['source', 'cell_type'])
```



After integration. The output is an AnnData object with the uniport embedding stored in `.obsm['latent']`. Visualize the latent data using UMAP according to their cell types and sources.

[12]:
```
sc.pp.neighbors(adata, use_rep='latent')
sc.tl.umap(adata, min_dist=0.1)
sc.pl.umap(adata, color=['source', 'cell_type'])
```



[ ]:

1.2.2 Integrate MERFISH and scRNA

MERFISH and scRNA data preprocess

We apply uniPort to integrate high-plex RNA imaging-based spatially resolved MERFISH data with scRNA-seq data. The MERFISH data includes 64,373 cells with 155 genes, and the scRNA-seq data includes 30,370 cells with 21,030 genes from six mice measured with dissociated scRNA-seq (10X).

[1]:
```
import uniport as up
import scanpy as sc
```

(continues on next page)

(continued from previous page)

```
import pandas as pd
import numpy as np
up.__version__
```

[1]:
```
'1.1.1'
```

Read the cell type annotations of MERFISH and scRNA-seq data seperately.

[2]:
```
labels_merfish = pd.read_csv('MERFISH/MERFISH_st_filter_cluster.txt', sep='\t')
celltype_merfish = labels_merfish['cluster_main'].values
labels_rna = pd.read_csv('MERFISH/MERFISH_scRNA_filter_cluster.txt', sep='\t')
celltype_rna = labels_rna['cluster_main'].values
```

Read MERFISH and scRNA-seq into `AnnData` objects using `load_file` fucntion in uniport.

[3]:
```
# adata_merfish = sc.read_h5ad('MERFISH/merfish0.h5ad')
# adata_rna = sc.read_h5ad('MERFISH/rna0.h5ad')
adata_merfish = up.load_file('MERFISH/MERFISH_mouse1.txt')
adata_rna = up.load_file('MERFISH/RNA_count.txt')
```

Add 'cell_type', 'domain_id' and 'source' to the `AnnData`objects.
'cell_type' stores cell label annotations.
'domain_id' identifies the modality using a number category.
'source' identifies the modality using its name.

[4]:
```
adata_merfish.obs['cell_type'] = celltype_merfish
adata_merfish.obs['domain_id'] = 0
adata_merfish.obs['domain_id'] = adata_merfish.obs['domain_id'].astype('category')
adata_merfish.obs['source'] = 'MERFISH'

adata_rna.obs['cell_type'] = celltype_rna
adata_rna.obs['domain_id'] = 1
adata_rna.obs['domain_id'] = adata_rna.obs['domain_id'].astype('category')
adata_rna.obs['source'] = 'RNA'

print(adata_rna.obs)
print(adata_merfish.obs)
```

```
                    cell_type domain_id source
AAACCTGAGATGTGGC-1  Fibroblast         1    RNA
AAACCTGCACACAGAG-1  Excitatory         1    RNA
AAACCTGCACTACAGT-1  Inhibitory         1    RNA
AAACCTGTCAGGATCT-1  Excitatory         1    RNA
AAACCTGTCGCACTCT-1   OD Mature         1    RNA
...                        ...       ...    ...
TTTGGTTGTTATCACG-6  Inhibitory         1    RNA
TTTGGTTGTTATTCTC-6  Inhibitory         1    RNA
TTTGTCAGTTCCGTCT-6  Inhibitory         1    RNA
TTTGTCATCGTGGGAA-6  Inhibitory         1    RNA
TTTGTCATCTTTACAC-6  Excitatory         1    RNA
```

(continues on next page)

(continued from previous page)

```
[30370 rows x 3 columns]
             cell_type domain_id   source
cell1        Astrocyte         0  MERFISH
cell2       Inhibitory         0  MERFISH
cell3       Inhibitory         0  MERFISH
cell4       Inhibitory         0  MERFISH
cell5       Inhibitory         0  MERFISH
...                ...       ...      ...
cell73650    OD Mature         0  MERFISH
cell73651    OD Mature         0  MERFISH
cell73653  OD Immature         0  MERFISH
cell73654    OD Mature         0  MERFISH
cell73655    OD Mature         0  MERFISH

[64373 rows x 3 columns]
```

Concatenate scATAC-seq and scRNA-seq with common genes using `AnnData.concatenate`.

[5]:
```
adata_cm = adata_merfish.concatenate(adata_rna, join='inner', batch_key='domain_id')
print(adata_cm.obs)
```

```
                      cell_type domain_id   source
cell1-0               Astrocyte         0  MERFISH
cell2-0              Inhibitory         0  MERFISH
cell3-0              Inhibitory         0  MERFISH
cell4-0              Inhibitory         0  MERFISH
cell5-0              Inhibitory         0  MERFISH
...                         ...       ...      ...
TTTGGTTGTTATCACG-6-1  Inhibitory         1      RNA
TTTGGTTGTTATTCTC-6-1  Inhibitory         1      RNA
TTTGTCAGTTCCGTCT-6-1  Inhibitory         1      RNA
TTTGTCATCGTGGGAA-6-1  Inhibitory         1      RNA
TTTGTCATCTTTACAC-6-1  Excitatory         1      RNA

[94743 rows x 3 columns]
```

Preprocess data using functions `normalize_total`, `log1p` and `highly_variable_genes` in `scanpy` and `batch_scale` in uniport (modified from SCALEX)

[6]:
```
sc.pp.normalize_total(adata_cm)
sc.pp.log1p(adata_cm)
sc.pp.highly_variable_genes(adata_cm, n_top_genes=2000, batch_key='domain_id',␣
↪inplace=False, subset=True)
up.batch_scale(adata_cm)
```

```
... storing 'cell_type' as categorical
... storing 'source' as categorical
```

[7]:
```
sc.pp.normalize_total(adata_merfish)
sc.pp.log1p(adata_merfish)
sc.pp.highly_variable_genes(adata_merfish, n_top_genes=2000, inplace=False, subset=True)
up.batch_scale(adata_merfish)
```

[8]:
```
sc.pp.normalize_total(adata_rna)
sc.pp.log1p(adata_rna)
sc.pp.highly_variable_genes(adata_rna, n_top_genes=2000, inplace=False, subset=True)
up.batch_scale(adata_rna)
```

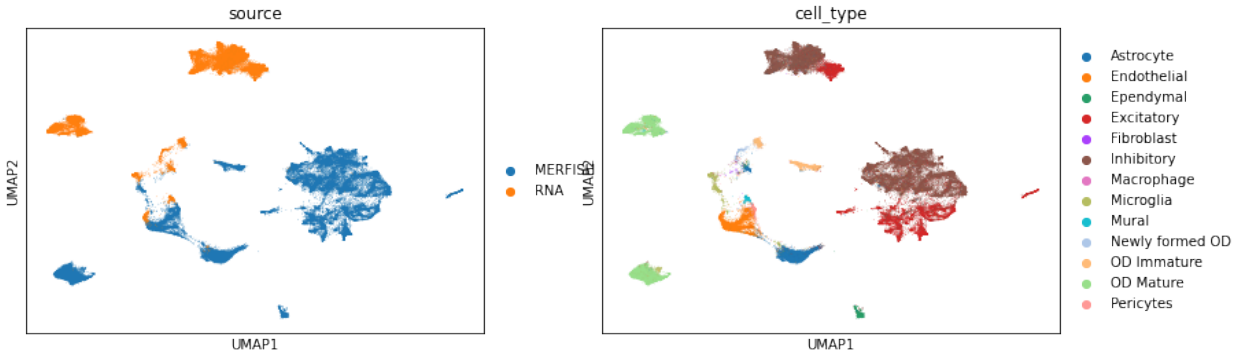
Save the preprocessed data for integration and online prediction.

[9]:
```
adata_merfish.write('MERFISH/MERFISH_processed.h5ad', compression='gzip')
adata_rna.write('MERFISH/RNA_processed.h5ad', compression='gzip')
adata_cm.write('MERFISH/MERFISH_and_RNA.h5ad', compression='gzip')
```

```
... storing 'cell_type' as categorical
... storing 'source' as categorical
... storing 'cell_type' as categorical
... storing 'source' as categorical
```

Visualize the data using UMAP according to their cell types and sources.

[10]:
```
adata_cm_copy = adata_cm.copy()
sc.pp.pca(adata_cm_copy)
sc.pp.neighbors(adata_cm_copy)
sc.tl.umap(adata_cm_copy, min_dist=0.1)
sc.pl.umap(adata_cm_copy, color=['source', 'cell_type'])
```



Integrate the MERFISH and scRNA-seq data using both common and dataset-specific genes by *Run()* function in uniport. The latent representations of data are stored in `adata.obs['latent']`.

MERFISH and scRNA data integration

[11]:
```
adata = up.Run(adatas=[adata_merfish, adata_rna], adata_cm=adata_cm, lambda_kl=5.0)
```

```
Dataset 0: MERFISH
AnnData object with n_obs × n_vars = 64373 × 155
    obs: 'cell_type', 'domain_id', 'source'
    var: 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
Dataset 1: RNA
AnnData object with n_obs × n_vars = 30370 × 2000
    obs: 'cell_type', 'domain_id', 'source'
    var: 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
```

(continues on next page)

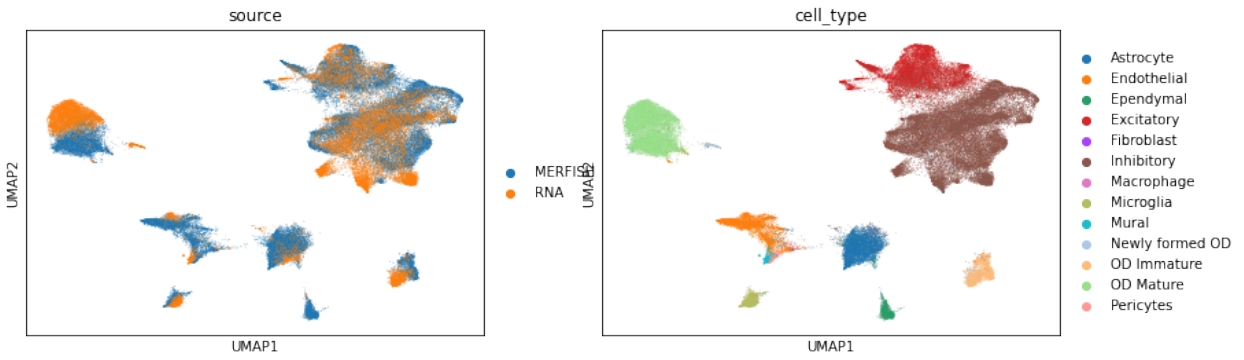
(continued from previous page)

```
Reference dataset is dataset 1


Data with common HVG
AnnData object with n_obs × n_vars = 94743 × 153
    obs: 'cell_type', 'domain_id', 'source'
    var: 'highly_variable', 'means', 'dispersions', 'dispersions_norm', 'highly_variable_
↪nbatches', 'highly_variable_intersection'
    uns: 'log1p', 'hvg'
```

```
Epochs: 100%|| 82/82 [19:21<00:00, 14.16s/it, recloss=795.64,klloss=29.19,otloss=4.08]
```

[12]:
```
sc.pp.neighbors(adata, use_rep='latent')
sc.tl.umap(adata, min_dist=0.1)
sc.pl.umap(adata, color=['source', 'cell_type'])
```



Predict scRNA-seq through MERFISH

uniPort trained an encoder network to project cells with common genes across datasets into a common cell-embedding latent space and a decoder network to reconstruct cells with common and specific genes. Therefore, once the coupled-VAE is trained well, it can be regarded as a reference atlas, in turn allowing uniPort to integrate new single-cell data in an online manner without modal retraining. Most importantly, uniPort can generate both common and specific genes in one dataset through common genes in another dataset by the atlas.

[1]:
```
import uniport as up
import scanpy as sc
import pandas as pd
```

Read MERFISH and scRNA-seq with common genes.

[2]:
```
adata_cm = sc.read_h5ad('MERFISH/MERFISH_and_RNA.h5ad')
adata_merfish1 = adata_cm[adata_cm.obs['source']=='MERFISH'].copy()
```

Read cell type annotations of MERFISH profiled from a new mouse.

[3]:
```
labels_mouse2 = pd.read_csv('MERFISH/MERFISH_mouse2_cluster.txt', sep='\t')
celltype_mouse2 = labels_mouse2['cluster_main'].values
```

[4]:
```
adata_merfish2 = up.load_file('MERFISH/MERFISH_mouse2.txt')
```

Add information to `AnnData`. Identify the new MERFISH as domain '-1'.

[5]:
```
adata_merfish2.obs['cell_type'] = celltype_mouse2
adata_merfish2.obs['domain_id'] = -1
adata_merfish2.obs['domain_id'].astype('category')
adata_merfish2.obs['domain_id'] = adata_merfish2.obs['domain_id'].astype('category')
adata_merfish2.obs['source'] = 'MERFISH2'
```

Select common genes in new MERFISH data to make it consistent with training data.

[6]:
```
adata_cm = adata_merfish2.concatenate(adata_merfish1, join='inner', batch_key='domain_id
↪')
adata_merfish2 = adata_cm[adata_cm.obs['source']=='MERFISH2'].copy()
```

Data normalization.

[7]:
```
sc.pp.normalize_total(adata_merfish2)
sc.pp.log1p(adata_merfish2)
up.batch_scale(adata_merfish2)
```

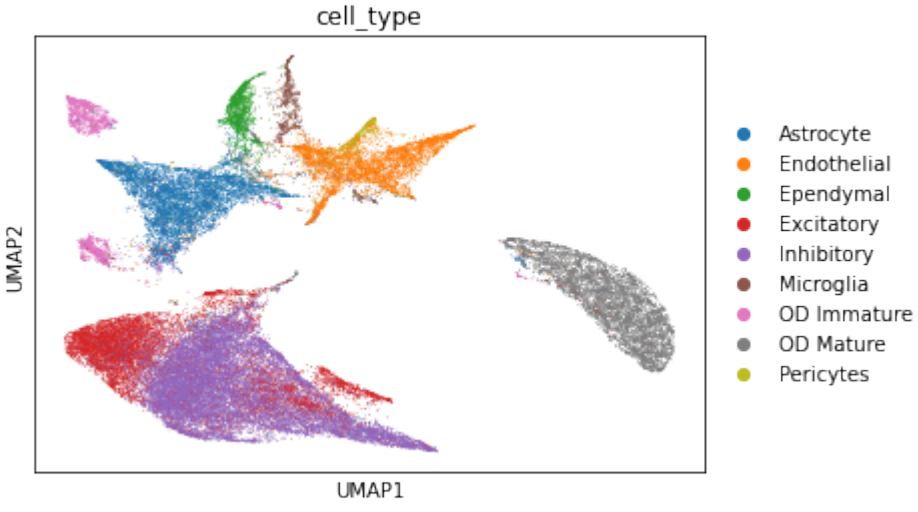
Predict corresponding RNA gene expression through the new MERFISH data. Here pred_id is the doamin_id of the modality we want to predict.

[8]:
```
adata = up.Run(adata_cm=adata_merfish2, out='predict', pred_id=1)
```

Perform UMAP visualization of predict scRNA-seq data.

[9]:
```
sc.pp.neighbors(adata, use_rep='predict')
sc.tl.umap(adata, min_dist=0.1)
sc.pl.umap(adata, color=['cell_type'])
```

```
... storing 'cell_type' as categorical
... storing 'source' as categorical
```



[10]:
```
print(adata.obsm['predict'].shape)
```

```
(59651, 2000)
```

Save the predicted data.

[11]:
```
latent = adata.obsm['predict']
adata_rna = sc.read_h5ad('MERFISH/RNA.h5ad')
predict = pd.DataFrame(latent.T, index=adata_rna.var_names, columns=adata.obs_names)
predict.to_csv('MERFISH/Predict_mouse2_RNA.txt', sep='\t')
```

Impute genes for MERFISH

[1]:
```
import uniport as up
import scanpy as sc
import numpy as np
from scipy.stats import spearmanr, pearsonr
import pandas as pd
from scvi.external import GIMVI
print(up.__version__)

seed = 1
train_size = 0.8
np.random.seed(seed)
```

```
Global seed set to 0
```

```
1.1.2
```

```
/home/kcao/miniconda3/envs/py39/lib/python3.9/site-packages/pytorch_lightning/utilities/
↪warnings.py:53: LightningDeprecationWarning: pytorch_lightning.utilities.warnings.rank_
↪zero_deprecation has been deprecated in v1.6 and will be removed in v1.8. Use the␣
↪equivalent function from the pytorch_lightning.utilities.rank_zero module instead.
  new_rank_zero_deprecation(
/home/kcao/miniconda3/envs/py39/lib/python3.9/site-packages/pytorch_lightning/utilities/
↪warnings.py:58: LightningDeprecationWarning: The `pytorch_lightning.loggers.base.rank_
↪zero_experiment` is deprecated in v1.7 and will be removed in v1.9. Please use␣
↪`pytorch_lightning.loggers.logger.rank_zero_experiment` instead.
  return new_rank_zero_deprecation(*args, **kwargs)
```

Read and process data.

[2]:
```
labels_merfish = pd.read_csv('/home/kcao/uniPort/MERFISH/MERFISH_mouse1_cluster.txt',␣
↪sep='\t')
celltype_merfish = labels_merfish['cluster_main'].values
labels_rna = pd.read_csv('/home/kcao/uniPort/MERFISH/scRNA_cluster.txt', sep='\t')
celltype_rna = labels_rna['cluster_main'].values

spatial_data = sc.read_h5ad('/home/kcao/uniPort/MERFISH/merfish0.h5ad')
seq_data = sc.read_h5ad('/home/kcao/uniPort/MERFISH/rna0.h5ad')

spatial_data.obs['cell_type'] = celltype_merfish
spatial_data.obs['domain_id'] = 0
spatial_data.obs['domain_id'] = spatial_data.obs['domain_id'].astype('category')
spatial_data.obs['source'] = 'MERFISH'
```

(continues on next page)

(continued from previous page)

```
seq_data.obs['cell_type'] = celltype_rna
seq_data.obs['domain_id'] = 1
seq_data.obs['domain_id'] = seq_data.obs['domain_id'].astype('category')
seq_data.obs['source'] = 'RNA'

adata_cm = spatial_data.concatenate(seq_data, join='inner', batch_key='domain_id')
spatial_data = adata_cm[adata_cm.obs['source']=='MERFISH'].copy()
seq_data = adata_cm[adata_cm.obs['source']=='RNA'].copy()
```

Randomly select training and testing genes.

[3]:
```
#only use genes in both datasets
seq_data = seq_data[:, spatial_data.var_names].copy()

seq_gene_names = seq_data.var_names
n_genes = seq_data.n_vars
n_train_genes = int(n_genes*train_size)

#randomly select training_genes
rand_train_gene_idx = np.random.choice(range(n_genes), n_train_genes, replace = False)
rand_test_gene_idx = sorted(set(range(n_genes)) - set(rand_train_gene_idx))
rand_train_genes = seq_gene_names[rand_train_gene_idx]
rand_test_genes = seq_gene_names[rand_test_gene_idx]

#spatial_data_partial has a subset of the genes to train on
spatial_data_partial = spatial_data[:,rand_train_genes].copy()

#remove cells with no counts
sc.pp.filter_cells(spatial_data_partial, min_counts= 1)
sc.pp.filter_cells(seq_data, min_counts = 1)

#setup_anndata for spatial and sequencing data
GIMVI.setup_anndata(spatial_data_partial, labels_key='cell_type', batch_key='source')
GIMVI.setup_anndata(seq_data, labels_key='cell_type')

#spatial_data should use the same cells as our training data
#cells may have been removed by scanpy.pp.filter_cells()
spatial_data = spatial_data[spatial_data_partial.obs_names]

print(spatial_data_partial.var_names)
```

```
Index(['Htr2c', 'Cyp19a1', 'Man1a', 'Tiparp', 'Cspg5', 'Sema4d', 'Pou3f2',
       'Cbln1', 'Gem', 'Fn1',
       ...
       'Trhr', 'Galr1', 'Cenpe', 'Mc4r', 'Amigo2', 'Sst', 'Crhr2', 'Trh',
       'Sema3c', 'Gabrg1'],
      dtype='object', length=122)
```

[4]:
```
adata_cm = spatial_data_partial.concatenate(seq_data, join='inner', batch_key='domain_id
↪')
```

[5]:
```
sc.pp.normalize_total(adata_cm)
sc.pp.log1p(adata_cm)
up.batch_scale(adata_cm)
print(adata_cm)
```

```
AnnData object with n_obs × n_vars = 94741 × 122
    obs: 'cell_type', 'domain_id', 'source', 'n_counts', '_scvi_batch', '_scvi_labels'
    uns: 'log1p'
```

[6]:
```
sc.pp.normalize_total(spatial_data_partial)
sc.pp.log1p(spatial_data_partial)
up.batch_scale(spatial_data_partial)
print(spatial_data_partial)
```

```
AnnData object with n_obs × n_vars = 64373 × 122
    obs: 'cell_type', 'domain_id', 'source', 'n_counts', '_scvi_batch', '_scvi_labels'
    uns: '_scvi_uuid', '_scvi_manager_uuid', 'log1p'
```

[7]:
```
sc.pp.normalize_total(seq_data)
sc.pp.log1p(seq_data)
up.batch_scale(seq_data)
print(seq_data)
```

```
AnnData object with n_obs × n_vars = 30368 × 153
    obs: 'cell_type', 'domain_id', 'source', 'n_counts', '_scvi_batch', '_scvi_labels'
    uns: '_scvi_uuid', '_scvi_manager_uuid', 'log1p'
```

[8]:
```
adatas = [spatial_data_partial, seq_data]
```

Integrate the MERFISH and scRNA-seq data using both common and dataset-specific genes by `Run()` function in uniport. The latent representations of data are stored in `adata.obs['latent']`.

[9]:
```
adata = up.Run(adatas=adatas, adata_cm=adata_cm, lambda_kl=5.0, model_info=True)
```

```
Device: cuda
Dataset 0: MERFISH
AnnData object with n_obs × n_vars = 64373 × 122
    obs: 'cell_type', 'domain_id', 'source', 'n_counts', '_scvi_batch', '_scvi_labels'
    uns: '_scvi_uuid', '_scvi_manager_uuid', 'log1p'
Dataset 1: RNA
AnnData object with n_obs × n_vars = 30368 × 153
    obs: 'cell_type', 'domain_id', 'source', 'n_counts', '_scvi_batch', '_scvi_labels'
    uns: '_scvi_uuid', '_scvi_manager_uuid', 'log1p'
Reference dataset is dataset 1


Data with common HVG
AnnData object with n_obs × n_vars = 94741 × 122
    obs: 'cell_type', 'domain_id', 'source', 'n_counts', '_scvi_batch', '_scvi_labels'
    uns: 'log1p'



INFO:root:model
VAE(
```

(continues on next page)

(continued from previous page)

```
  (encoder): Encoder(
    (enc): ModuleList(
      (0): NN(
        (net): ModuleList(
          (0): Block(
            (fc): Linear(in_features=122, out_features=1024, bias=True)
            (norm): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_
↪running_stats=True)
            (act): ReLU()
          )
        )
      )
    )
    (mu_enc): ModuleList(
      (0): NN(
        (net): ModuleList(
          (0): Block(
            (fc): Linear(in_features=1024, out_features=16, bias=True)
          )
        )
      )
    )
    (var_enc): ModuleList(
      (0): NN(
        (net): ModuleList(
          (0): Block(
            (fc): Linear(in_features=1024, out_features=16, bias=True)
          )
        )
      )
    )
  )
  (decoder): Decoder(
    (dec): ModuleList(
      (0): NN(
        (net): ModuleList(
          (0): Block(
            (fc): Linear(in_features=16, out_features=122, bias=True)
            (norm): DSBatchNorm(
              (bns): ModuleList(
                (0): BatchNorm1d(122, eps=1e-05, momentum=0.1, affine=True, track_
↪running_stats=True)
                (1): BatchNorm1d(122, eps=1e-05, momentum=0.1, affine=True, track_
↪running_stats=True)
              )
            )
            (act): Sigmoid()
          )
        )
      )
      (1): NN(
        (net): ModuleList(
```

(continues on next page)

(continued from previous page)

```
          (0): Block(
            (fc): Linear(in_features=16, out_features=122, bias=True)
            (norm): BatchNorm1d(122, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
            (act): Sigmoid()
          )
        )
      )
      (2): NN(
        (net): ModuleList(
          (0): Block(
            (fc): Linear(in_features=16, out_features=153, bias=True)
            (norm): BatchNorm1d(153, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
            (act): Sigmoid()
          )
        )
      )
    )
  )
)
2022-11-25 22:47:36,419 - root - INFO - model
VAE(
  (encoder): Encoder(
    (enc): ModuleList(
      (0): NN(
        (net): ModuleList(
          (0): Block(
            (fc): Linear(in_features=122, out_features=1024, bias=True)
            (norm): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_
↪running_stats=True)
            (act): ReLU()
          )
        )
      )
    )
    (mu_enc): ModuleList(
      (0): NN(
        (net): ModuleList(
          (0): Block(
            (fc): Linear(in_features=1024, out_features=16, bias=True)
          )
        )
      )
    )
    (var_enc): ModuleList(
      (0): NN(
        (net): ModuleList(
          (0): Block(
            (fc): Linear(in_features=1024, out_features=16, bias=True)
          )
        )
```

(continues on next page)

(continued from previous page)

```
      )
    )
  )
  (decoder): Decoder(
    (dec): ModuleList(
      (0): NN(
        (net): ModuleList(
          (0): Block(
            (fc): Linear(in_features=16, out_features=122, bias=True)
            (norm): DSBatchNorm(
              (bns): ModuleList(
                (0): BatchNorm1d(122, eps=1e-05, momentum=0.1, affine=True, track_
↪running_stats=True)
                (1): BatchNorm1d(122, eps=1e-05, momentum=0.1, affine=True, track_
↪running_stats=True)
              )
            )
            (act): Sigmoid()
          )
        )
      )
      (1): NN(
        (net): ModuleList(
          (0): Block(
            (fc): Linear(in_features=16, out_features=122, bias=True)
            (norm): BatchNorm1d(122, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
            (act): Sigmoid()
          )
        )
      )
      (2): NN(
        (net): ModuleList(
          (0): Block(
            (fc): Linear(in_features=16, out_features=153, bias=True)
            (norm): BatchNorm1d(153, eps=1e-05, momentum=0.1, affine=True, track_running_
↪stats=True)
            (act): Sigmoid()
          )
        )
      )
    )
  )
)
Epochs: 100%|| 82/82 [08:27<00:00,  6.19s/it, recloss=817.68,klloss=42.54,otloss=7.25]
```

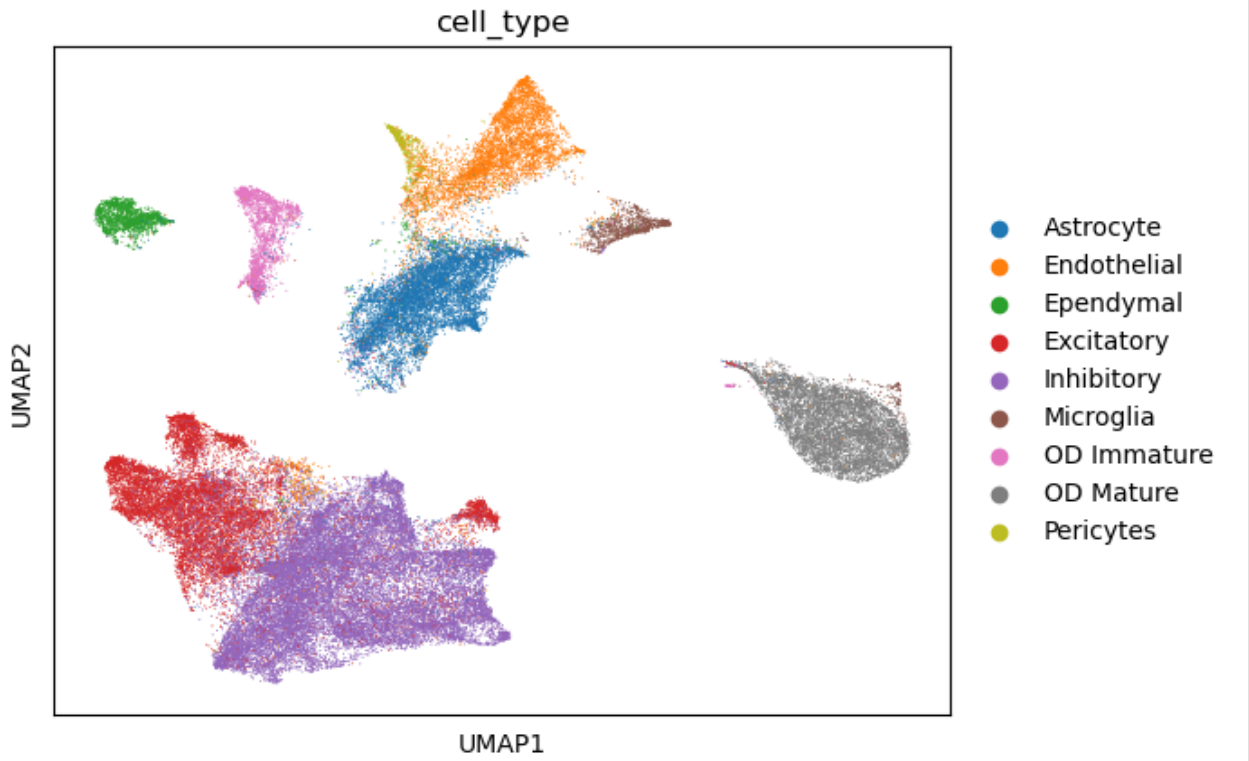
Predict

[10]:
```
adata_predict = up.Run(adata_cm=spatial_data_partial, out='predict', pred_id=1)
print(np.shape(adata_predict.obsm['predict']))
```

```
Device: cuda
(64373, 153)
```

[11]:
```
sc.pp.neighbors(adata_predict, use_rep='predict')
sc.tl.umap(adata_predict, min_dist=0.1)
sc.pl.umap(adata_predict, color=['cell_type'])
```

```
... storing 'cell_type' as categorical
... storing 'source' as categorical
```



Compute average/median Spearman and Pearson Correlation Coefficients.

[12]:
```
def imputation_score(fish_imputation, data_spatial, gene_ids_test, normalized=True):
#     _, fish_imputation = model.get_imputed_values(normalized=normalized)
    original, imputed = (
        data_spatial.X[:, gene_ids_test],
        fish_imputation[:, gene_ids_test],
    )

    if normalized:
        original /= data_spatial.X.sum(axis=1).reshape(-1, 1)

    original = np.array(original)

    spearman_gene = []
    pearsonr_gene = []
    for g in range(imputed.shape[1]):
        if np.all(imputed[:, g] == 0):
            correlation_scc = 0
            correlation_pcc=0
        else:
```

(continues on next page)

(continued from previous page)

```
            correlation_scc = spearmanr(original[:, g], imputed[:, g])[0]
            correlation_pcc = pearsonr(original[:, g], imputed[:, g])[0]
        spearman_gene.append(correlation_scc)
        pearsonr_gene.append(correlation_pcc)
    return np.median(np.array(spearman_gene)), np.mean(np.array(spearman_gene)), np.
↪median(np.array(pearsonr_gene)), np.mean(np.array(pearsonr_gene)),

print(imputation_score(adata_predict.obsm['predict'], spatial_data, rand_test_gene_idx,␣
↪True))
```

```
(0.26249964194121306, 0.2639037051584925, 0.2721606922755887, 0.2920161196211757)
```

1.2.3 Deconvolute pancreatic ductal adenocarcinoma (PDAC)

Integrate PDAC with scRNA

uniPort can output a global optimal transport (OT) plan, i.e., a cell-cell correspondence matrix, that provides flexible transfer learning for deconvolution of spatial heterogeneous data using scRNA data in OT space, instead of embedding latent space.

[1]:
```
import uniport as up
import scanpy as sc
import pandas as pd
```

Read microarray-based ST data of pancreatic ductal adenocarcinoma (PDAC) tissues for integration, the diameter of which stretches for 100 m. Cell-type deconvolution was applied on 428 spots paired with 1926 single cells, measuring 19,736 genes respectively.

[2]:
```
labels_rna = pd.read_csv('PDAC_scRNA_label.txt', sep='\t')
celltype = labels_rna['cell_type'].values
print(celltype)
```

```
['Acinar cells' 'Ductal' 'Ductal' ... 'Ductal' 'pDCs' 'RBCs']
```

[3]:
```
rna = sc.read('PDAC_scRNA.txt').transpose()
spot = up.load_file('PDAC_SPOT.txt')
```

Add *domain_id*, *cell_type* and *source* obs to AnnData.

[4]:
```
spot.obs['domain_id'] = 0
spot.obs['domain_id'] = spot.obs['domain_id'].astype('category')
spot.obs['source'] = 'SPOT'

rna.obs['cell_type'] = celltype
rna.obs['domain_id'] = 1
rna.obs['domain_id'] = rna.obs['domain_id'].astype('category')
rna.obs['source'] = 'RNA'
```

Concatenate SPOT and scRNA-seq with common genes using `AnnData.concatenate`.

[5]:
```
adata_cm = spot.concatenate(rna, join='inner', batch_key='domain_id')
```

Preprocess data with common genes. Select 2,000 highly variable common genes.
Scale data using `batch_scale` function in uniport (modified from SCALEX).

[6]:
```
sc.pp.normalize_total(adata_cm)
sc.pp.log1p(adata_cm)
sc.pp.highly_variable_genes(adata_cm, n_top_genes=2000, batch_key='domain_id',␣
↪inplace=False, subset=True)
up.batch_scale(adata_cm)
```

```
... storing 'source' as categorical
... storing 'cell_type' as categorical
```

Preprocess SPOT data. Select 2,000 highly variable genes as SPOT speicifc.

[7]:
```
sc.pp.normalize_total(spot)
sc.pp.log1p(spot)
sc.pp.highly_variable_genes(spot, n_top_genes=2000, inplace=False, subset=True)
up.batch_scale(spot)
```

Preprocess scRNA-seq data. Select 2,000 highly variable genes as RNA specific.

[8]:
```
sc.pp.normalize_total(rna)
sc.pp.log1p(rna)
sc.pp.highly_variable_genes(rna, n_top_genes=2000, inplace=False, subset=True)
up.batch_scale(rna)
```

Integrate the SPOT and scRNA-seq data using both common and dataset-specific genes by `Run` function in uniport. Set *save_OT=True* and return a OT plan, which is a SPOT by RNA probabilistic matching matrix.

[9]:
```
adata, OT = up.Run(adatas=[spot,rna], adata_cm=adata_cm, save_OT=True)
```

```
Dataset 0: SPOT
AnnData object with n_obs × n_vars = 428 × 2000
    obs: 'domain_id', 'source'
    var: 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
Dataset 1: RNA
AnnData object with n_obs × n_vars = 1926 × 2000
    obs: 'cell_type', 'domain_id', 'source'
    var: 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
Reference dataset is dataset 1


Data with common HVG
AnnData object with n_obs × n_vars = 2354 × 2000
    obs: 'domain_id', 'source', 'cell_type'
    var: 'highly_variable', 'means', 'dispersions', 'dispersions_norm', 'highly_variable_
↪nbatches', 'highly_variable_intersection'
    uns: 'log1p', 'hvg'


Warning! Saving Optimal Transport plan needs extra 0.01 GB memory, please set save_
```

(continues on next page)

(continued from previous page)

```
↪OT=False if no enough memory!
float32
Size of transport plan between datasets 0 and 1: (428, 1926)
```

```
Epochs:  77%|     | 2554/3334 [24:20<07:26,  1.75it/s, recloss=358.15,klloss=6.82,
↪otloss=8.00]
```

```
EarlyStopping: run 2555 epoch
```

Save OT plan for deconvolution.

[10]:
```
name_idx = adata_cm[adata_cm.obs['source']=='SPOT'].obs_names
name_col = adata_cm[adata_cm.obs['source']=='RNA'].obs_names
OT_pd = pd.DataFrame(OT[0], index=name_idx, columns=name_col)
OT_pd.to_csv('/data/PDAC/OT_PDAC.txt', sep='\t')
```

Spatial deconvolution of PDAC with OT plan

Use R packages for plotting temporarily. The python fucntion is in development.

```
library(scatterpie)
library(RColorBrewer)
library(grDevices)
library(Seurat)
library(data.table)
```

```
file_path <- '/data/pdac/'
```

Load spatial transcriptomics of primary pancreatic cancer tissue and reference scRNA. The datasets can be downloaded from GSE111672.

```
# load st expression matrix
dataA = fread(paste0(file_path,"GSM3036911_PDAC-A-ST1-filtered.txt.gz"), header =T,check.
↪names = F)
dataA = as.data.frame(dataA)
dataA = dataA %>% distinct(Genes,.keep_all = T) %>% column_to_rownames("Genes")

# load paired scRNA data
scdataA = fread(paste0(file_path,'GSE111672_PDAC-A-indrop-filtered-expMat.txt.gz'),
↪header = T)
scdataA = as.data.frame(scdataA)
scdataA = scdataA[!duplicated(scdataA$Genes),]
rownames(scdataA) <- scdataA$Genes
scdataA <- scdataA[,-1]

# extract celltype information
names = colnames(scdataA)[1:ncol(scdataA)] %>% as.data.frame() %>% {colnames(.) <- 'raw_
↪type';.}
names$cell = paste0('cell',1:ncol(scdataA))
names$cell_type = names$raw_type
names$cell_type[str_detect(names$cell_type,'Ductal')] = 'Ductal'
names$cell_type[str_detect(names$cell_type,'Acinar cells')] = 'Acinar cells'
```

(continues on next page)

(continued from previous page)

```
names$cell_type[str_detect(names$cell_type,'Cancer clone A')] = 'Cancer clone A'
names$cell_type[str_detect(names$cell_type,'Cancer clone B')] = 'Cancer clone B'
names$cell_type[str_detect(names$cell_type,'mDCs')] = 'mDCs'
names$cell_type[str_detect(names$cell_type,'Tuft cells')] = 'Tuft cells'
names$cell_type[str_detect(names$cell_type,'pDCs')] = 'pDCs'
names$cell_type[str_detect(names$cell_type,'Endocrine cells')] = 'Endocrine cells'
names$cell_type[str_detect(names$cell_type,'Endothelial cells')] = 'Endothelial cells'
names$cell_type[str_detect(names$cell_type,'Macrophages')] = 'Macrophages'
names$cell_type[str_detect(names$cell_type,'Mast cells')] = 'Mast cells'
names$cell_type[str_detect(names$cell_type,'T cells & NK cells')] = 'T & NK cells'
names$cell_type[str_detect(names$cell_type,'Monocytes')] = 'Monocytes'
names$cell_type[str_detect(names$cell_type,'RBCs')] = 'RBCs'
names$cell_type[str_detect(names$cell_type,'Fibroblasts')] = 'Fibroblasts'
rownames(names) <- names$cell
colnames(scdataA) = paste0('cell',1:ncol(scdataA))

# get coordinates of spots from st data
ind <- as.data.frame(t(sapply(
str_split(colnames(dataA), "x"),
function(x){
    x <- as.numeric(x)
    x <- as.vector(x)
}))) %>% {
    names(.) <- c("row_ind", "col_ind")
    rownames(.) <- paste0(.$row_ind,"x",.$col_ind)
    rownames(.) <- paste0('X',rownames(.))
;.}
```

Load plot function. The 'spatial_function.R' is stored here.

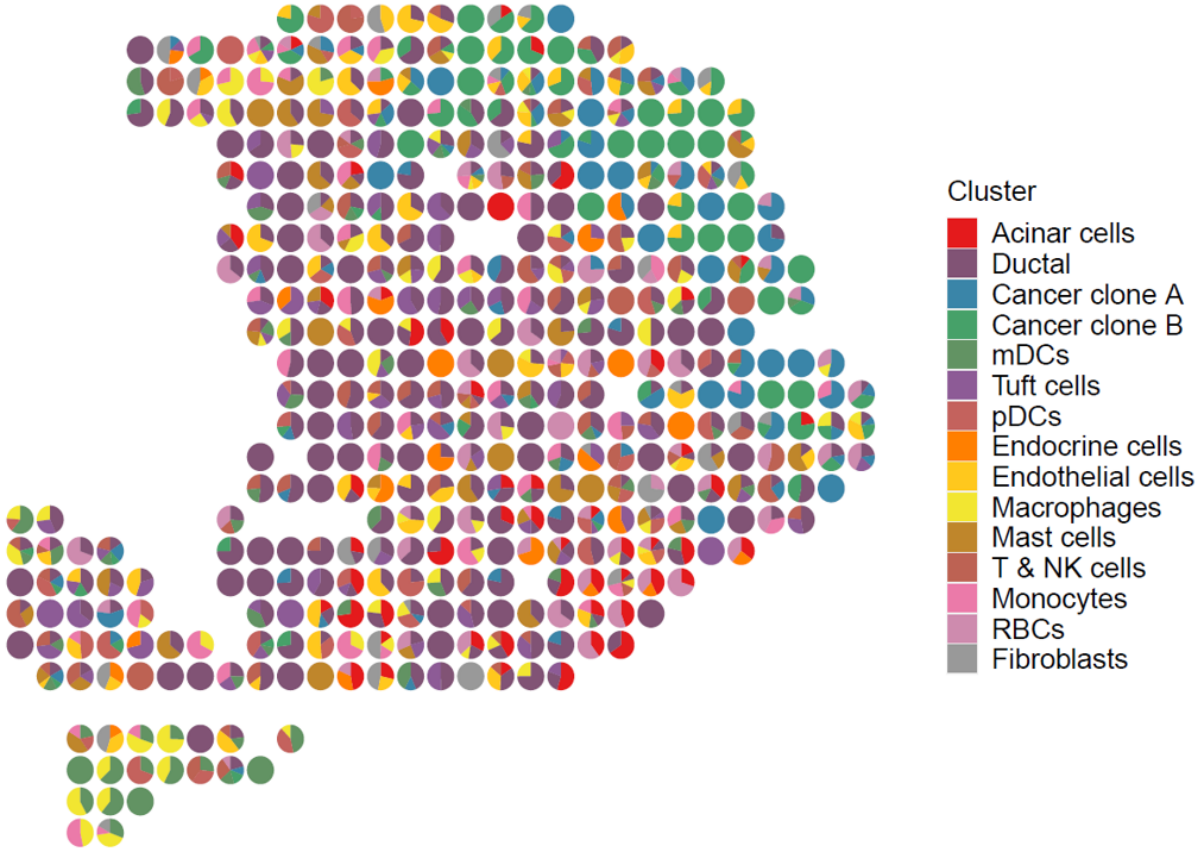
```
source(paste0(file_path,'spatial_function.R'))
```

Load OT matrix from uniPort output.

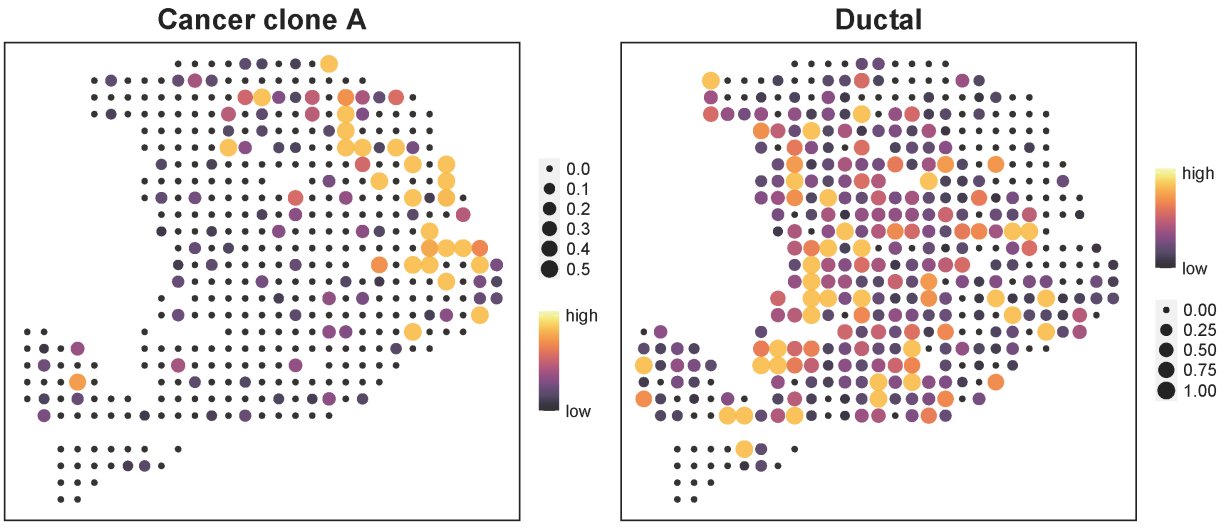
```
ot <- read.table(paste0(file_path,'OT_PDAC.txt'),sep = '\t',header = T, row.names = 1)
ot <- as.data.frame(t(ot))
rownames(ot) <- sapply(strsplit(rownames(ot),'\\.'),function(x)x[[1]])
# We provide balance option for scaling cluster proportion in st data through␣
↪multiplying cluster ratio in scRNA reference.
ot_map <- mapCluster(ot, meta = names, cluster = 'cell_type', min_cut = 0.25, balance =␣
↪T)
```

Visiualization of cluster proportion.

```
p <- stClusterPie(ot_map = ot_map, coord = ind, pie_scale = 0.8)
print(p)
```




```
p1 <- stClusterExp(ot_map, coord = ind, cluster = 'Cancer clone A',cut = 0.25)
p2 <- stClusterExp(ot_map, coord = ind, cluster = 'Ductal',cut = 0.25)
p1+p2
```



1.2.4 Deconvolute HER2-positive breast cancer (BRCA)

Integrate BRCA with scRNA

uniPort can output a global optimal transport (OT) plan, i.e., a cell-cell correspondence matrix, that provides flexible transfer learning for deconvolution of spatial heterogeneous data using scRNA data in OT space, instead of embedding latent space.

[1]:
```
import uniport as up
import scanpy as sc
import pandas as pd
```

Read microarray-based ST data of **HER2-positive breast cancer (BRCA)**, containing diffusely infiltrating cells that make it more diicult to deconvolute spots. In total, 2,518 spots with 17,943 genes and 100,064 cells with 29,733 genes were used for integration.

[2]:
```
labels_rna = pd.read_csv('Whole_miniatlas_meta.csv', sep=',')
celltype = labels_rna['celltype_major'].values
print(celltype)
```

```
['Endothelial' 'Endothelial' 'Endothelial' ... 'Myeloid' 'Myeloid'
 'Myeloid']
```

[3]:
```
# RNA = sc.read_10x_mtx('sc/')
# RNA.write('RNA.h5ad', compression='gzip')
rna = sc.read('RNA.h5ad')
spot = up.load_file('BRCA_SPOT_Count.txt')
```

Add *domain_id*, *cell_type* and *source* obs to AnnData.

[4]:
```
spot.obs['domain_id'] = 0
spot.obs['domain_id'] = spot.obs['domain_id'].astype('category')
spot.obs['source'] = 'SPOT'

rna.obs['cell_type'] = celltype
rna.obs['domain_id'] = 1
rna.obs['domain_id'] = rna.obs['domain_id'].astype('category')
rna.obs['source'] = 'RNA'
```

Concatenate SPOT and scRNA-seq with common genes using `AnnData.concatenate`.

[5]:
```
adata_cm = spot.concatenate(rna, join='inner', batch_key='domain_id')
```

Preprocess data with common genes. Select 2,000 highly variable common genes.
Scale data using `batch_scale` function in uniport (modified from SCALEX).

[6]:
```
sc.pp.normalize_total(adata_cm)
sc.pp.log1p(adata_cm)
sc.pp.highly_variable_genes(adata_cm, n_top_genes=2000, batch_key='domain_id',␣
↪inplace=False, subset=True)
up.batch_scale(adata_cm)
print(adata_cm)
```

```
... storing 'source' as categorical
... storing 'cell_type' as categorical
```

```
AnnData object with n_obs × n_vars = 102582 × 2000
    obs: 'domain_id', 'source', 'cell_type'
    var: 'gene_ids-1', 'feature_types-1', 'highly_variable', 'means', 'dispersions',
↪'dispersions_norm', 'highly_variable_nbatches', 'highly_variable_intersection'
    uns: 'log1p', 'hvg'
```

Preprocess SPOT data. Select 2,000 highly variable genes as SPOT speicifc.

[7]:
```
sc.pp.normalize_total(spot)
sc.pp.log1p(spot)
sc.pp.highly_variable_genes(spot, n_top_genes=2000, inplace=False, subset=True)
up.batch_scale(spot)
print(spot)
```

```
AnnData object with n_obs × n_vars = 2518 × 2000
    obs: 'domain_id', 'source'
    var: 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
```

Preprocess scRNA-seq data. Select 2,000 highly variable genes as RNA specific.

[8]:
```
sc.pp.normalize_total(rna)
sc.pp.log1p(rna)
sc.pp.highly_variable_genes(rna, n_top_genes=2000, inplace=False, subset=True)
up.batch_scale(rna)
print(rna)
```

```
AnnData object with n_obs × n_vars = 100064 × 2000
    obs: 'cell_type', 'domain_id', 'source'
    var: 'gene_ids', 'feature_types', 'highly_variable', 'means', 'dispersions',
↪'dispersions_norm'
    uns: 'log1p', 'hvg'
```

[9]:
```
spot.write('spot_processed.h5ad', compression='gzip')
rna.write('rna_processed.h5ad', compression='gzip')
adata_cm.write('rna_and_spot.h5ad', compression='gzip')
```

```
... storing 'source' as categorical
... storing 'cell_type' as categorical
... storing 'source' as categorical
```

Integrate the SPOT and scRNA-seq data using both common and dataset-specific genes by `Run` function in uniport. Set *save_OT=True* and return a OT plan, which is a SPOT by RNA probabilistic matching matrix.

[10]:
```
adata, OT = up.Run(adatas=[spot,rna], adata_cm=adata_cm, save_OT=True)
```

```
Dataset 0: SPOT
AnnData object with n_obs × n_vars = 2518 × 2000
    obs: 'domain_id', 'source'
    var: 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
Dataset 1: RNA
```

(continues on next page)

(continued from previous page)

```
AnnData object with n_obs × n_vars = 100064 × 2000
    obs: 'cell_type', 'domain_id', 'source'
    var: 'gene_ids', 'feature_types', 'highly_variable', 'means', 'dispersions',
↪'dispersions_norm'
    uns: 'log1p', 'hvg'
Reference dataset is dataset 1


Data with common HVG
AnnData object with n_obs × n_vars = 102582 × 2000
    obs: 'domain_id', 'source', 'cell_type'
    var: 'gene_ids-1', 'feature_types-1', 'highly_variable', 'means', 'dispersions',
↪'dispersions_norm', 'highly_variable_nbatches', 'highly_variable_intersection'
    uns: 'log1p', 'hvg'


Warning! Saving Optimal Transport plan needs extra 4.03 GB memory, please set save_
↪OT=False if no enough memory!
float32
Size of transport plan between datasets 0 and 1: (2518, 100064)
```

```
Epochs: 100%|| 75/75 [12:05<00:00,  9.67s/it, recloss=409.33,klloss=6.93,otloss=10.03]
```

Save OT plan for deconvolution.

[12]:
```
name_idx = adata_cm[adata_cm.obs['source']=='SPOT'].obs_names
name_col = adata_cm[adata_cm.obs['source']=='RNA'].obs_names
OT_pd = pd.DataFrame(OT[0], index=name_idx, columns=name_col)
OT_pd.to_csv('OT_BRCA.txt', sep='\t')
```

Spatial deconvolution of BRCA with OT plan

Use R packages for plotting temporarily. The python fucntion is in development.

```
library(scatterpie)
library(RColorBrewer)
library(grDevices)
library(Seurat)
library(tidyverse)
library(reshape2)
```

```
file_path <- '/data/BRCA/'
```

Load 10x Visium spatial data. The *st* folder contains cellranger outputs, and can be downloaded from 10xGenomics.

```
brca <- Load10X_Spatial(paste0(file_path,'st/'))
brca <- NormalizeData(brca)
brca <- ScaleData(brca)

# load cluster information of reference scRNA data
brca_cluster <- read.csv(paste0(file_path,'sc/Whole_miniatlas_meta.csv'), header = T,row.
↪names = 1) %>% .[-1,]
```

Load plot function. The 'spatial_function.R' is stored here.

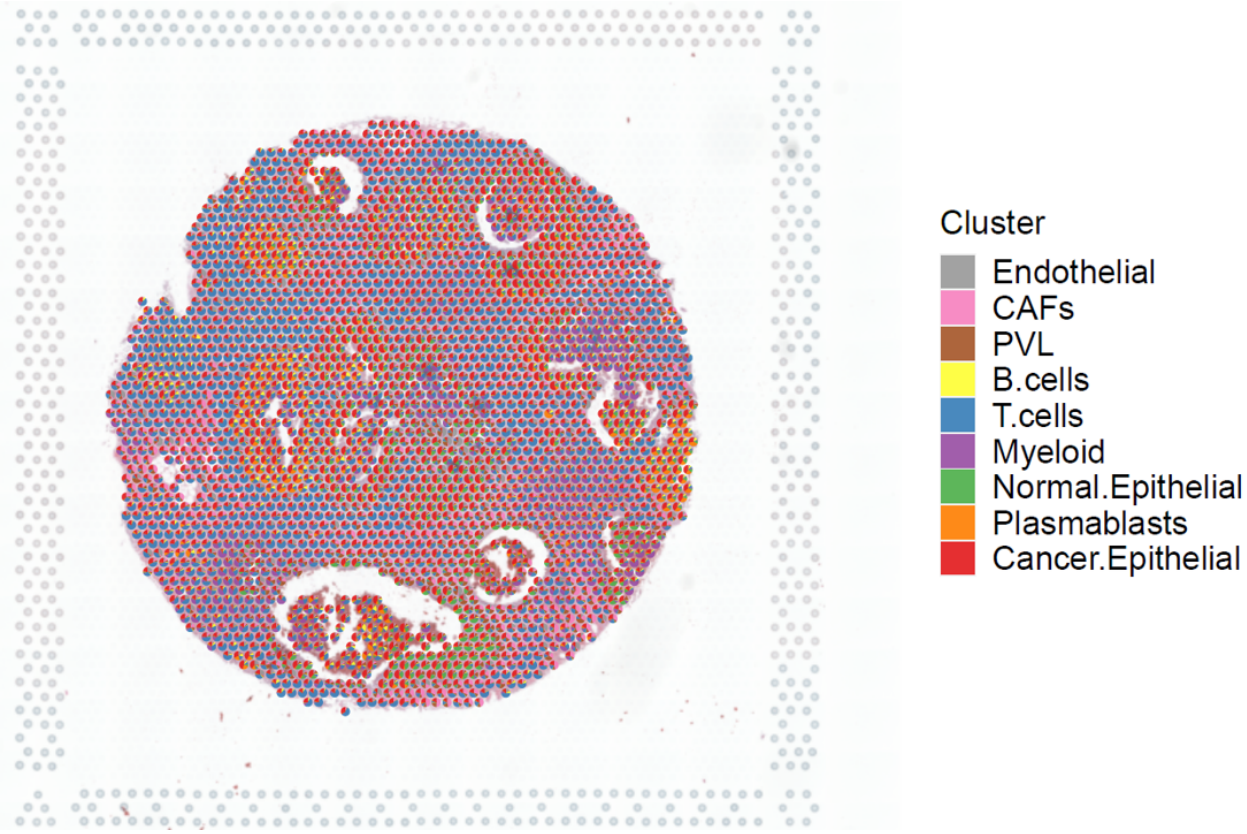
```
source(paste0(file_path,'spatial_function.R'))
```

Load OT plan from uniPort output.

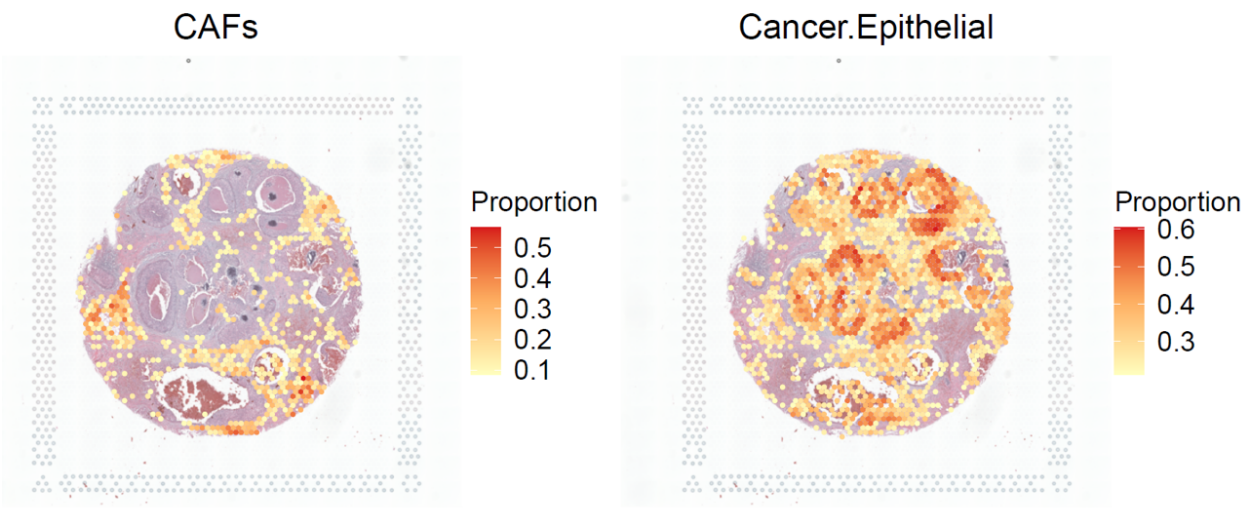
```
ot <- read.table(paste0(file_path,'OT_BRCA.txt'),sep = '\t',header = T,row.names = 1)
ot <- as.data.frame(t(ot))
rownames(ot) <- sapply(strsplit(rownames(ot),'\\.'),function(x)x[[1]])
ot_map <- mapCluster(ot,meta = brca_cluster, cluster = 'celltype_major')
```

Visiualization of cluster proportion.

```
p <- stClusterPie(ot_map = ot_map, st = brca)
print(p)
```




```
p1 <- stClusterExp(ot_map, brca, cluster = 'CAFs',cut = 0.15, point_size = 1.1)
p2 <- stClusterExp(ot_map, brca, cluster = 'Cancer.Epithelial',cut = 0.35, point_size =␣
↪1.1)
p1+p2
```



1.2.5 Vertical integration for paired datasets

uniPort also involves a vertical integration method for paired-cell datasets. Here we use uniPort to integrates paired SNARE-seq CellLineMixture datasets from GSE126074.

[1]:
```
import uniport as up
import numpy as np
import pandas as pd
import scanpy as sc
import episcanpy as epi
from sklearn.preprocessing import MinMaxScaler
```

[2]:
```
labels = pd.read_csv('snare/cell_line_meta.txt', sep='\t')
celltype = labels['cell_line'].values
```

[3]:
```
adata_peaks = up.load_file('snare/GSE126074_CellLineMixture_SNAREseq_chromatin_counts.tsv
↪')
adata_rna = up.load_file('snare/GSE126074_CellLineMixture_SNAREseq_cDNA_counts.tsv')
print(adata_peaks)
print(adata_rna)
```

```
AnnData object with n_obs × n_vars = 1047 × 136771
AnnData object with n_obs × n_vars = 1047 × 18666
```

[4]:
```
adata_peaks.obs['cell_type'] = celltype
adata_peaks.obs['domain_id'] = 0
adata_peaks.obs['domain_id'] = adata_peaks.obs['domain_id'].astype('category')
adata_peaks.obs['source'] = 'ATAC'

adata_rna.obs['cell_type'] = celltype
adata_rna.obs['domain_id'] = 1
adata_rna.obs['domain_id'] = adata_rna.obs['domain_id'].astype('category')
adata_rna.obs['source'] = 'RNA'
```

Preprocess scATAC-seq peaks. Select 2,000 highly variable peaks.

[5]:
```
adata_peaks.X[adata_peaks.X>1] = 1
epi.pp.select_var_feature(adata_peaks, nb_features=2000, show=False, copy=False)
sc.pp.normalize_total(adata_peaks)
up.batch_scale(adata_peaks)
print(adata_peaks)
```

```
AnnData object with n_obs × n_vars = 1047 × 2070
    obs: 'cell_type', 'domain_id', 'source'
    var: 'n_cells', 'prop_shared_cells', 'variability_score'
```

Preprocess scRNA-seq peaks. Select 2,000 highly variable genes.

[6]:
```
sc.pp.normalize_total(adata_rna)
sc.pp.log1p(adata_rna)
sc.pp.highly_variable_genes(adata_rna, n_top_genes=2000, inplace=False, subset=True)
up.batch_scale(adata_rna)
print(adata_rna)
```

```
AnnData object with n_obs × n_vars = 1047 × 2000
    obs: 'cell_type', 'domain_id', 'source'
    var: 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
```

Project scATAC-seq into latent space with the help of scRNA-seq. The latent representations of scATAC-seq are stored at `adata_atac.obs['latent']`

[7]:
```
adata_peaks = up.Run(adatas=[adata_peaks, adata_rna], mode='v', lr=0.001,␣
↪iteration=10000)
```

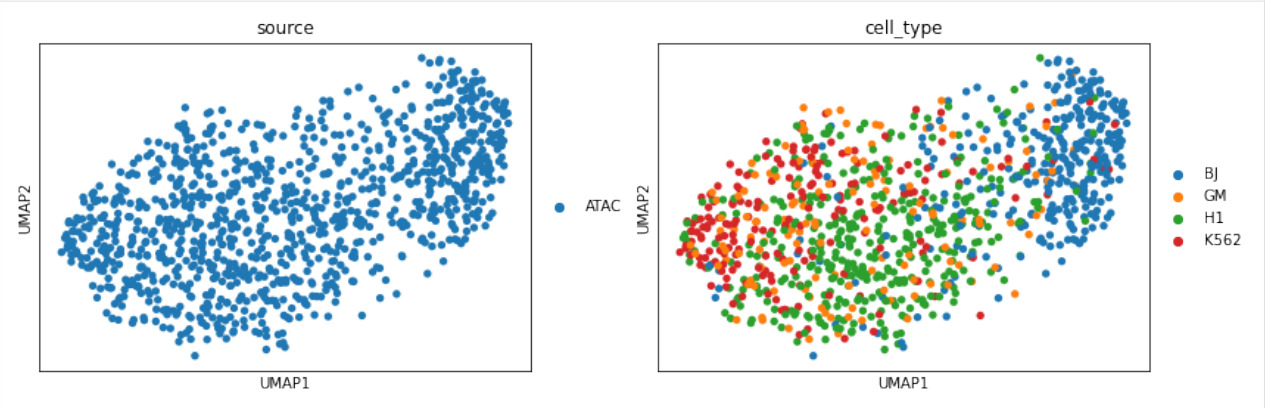
```
Dataset 0: ATAC
AnnData object with n_obs × n_vars = 1047 × 2070
    obs: 'cell_type', 'domain_id', 'source'
    var: 'n_cells', 'prop_shared_cells', 'variability_score'
Dataset 1: RNA
AnnData object with n_obs × n_vars = 1047 × 2000
    obs: 'cell_type', 'domain_id', 'source'
    var: 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
Reference dataset is dataset 1
```

```
Epochs: 100%|| 2500/2500 [16:26<00:00,  2.53it/s, recon_loss=239.627,kl_loss=7.374]
```

Perform UMAP visualization for scATAC before vertical integration.

[8]:
```
sc.pp.pca(adata_peaks)
sc.pp.neighbors(adata_peaks)
sc.tl.umap(adata_peaks, min_dist=0.1)
sc.pl.umap(adata_peaks, color=['source', 'cell_type'])
```

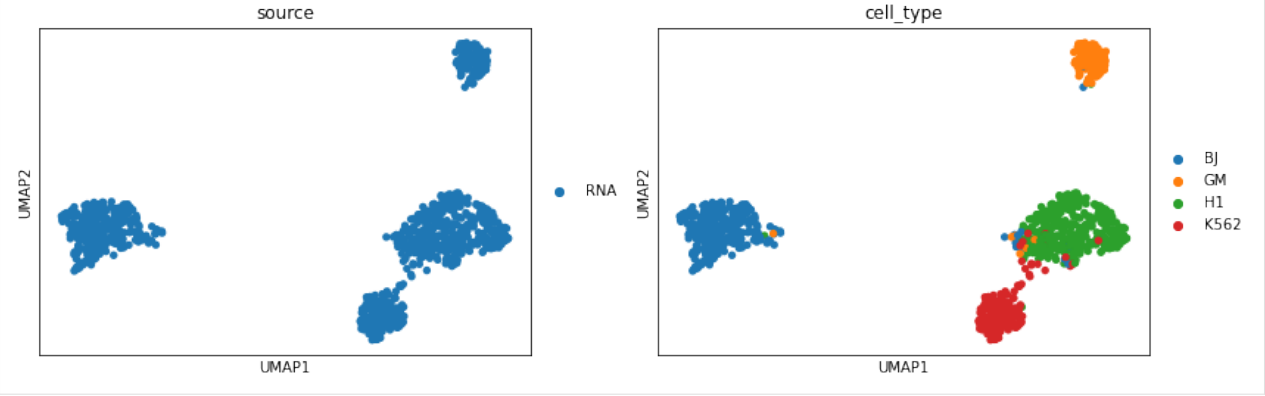
```
... storing 'cell_type' as categorical
... storing 'source' as categorical
```



Perform UMAP visualization for scRNA.

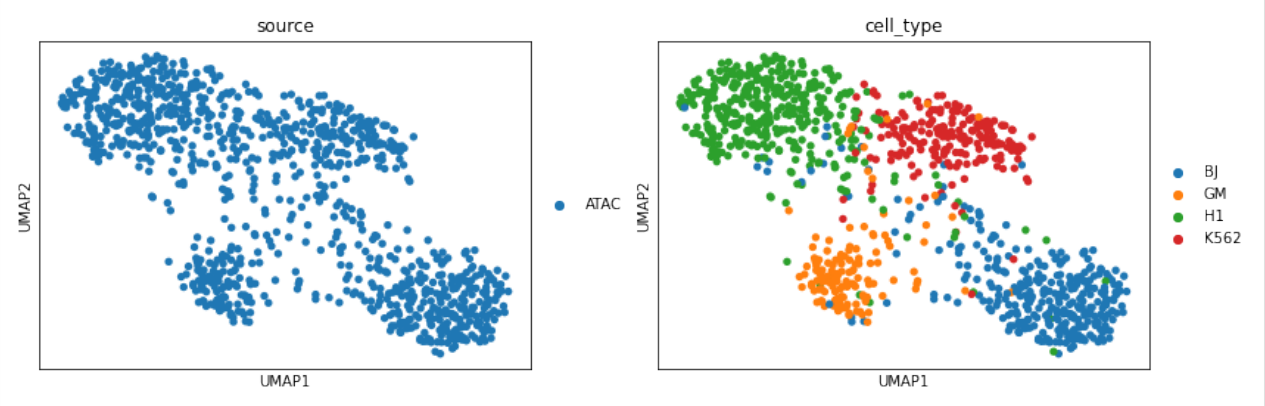
[9]:
```
sc.pp.pca(adata_rna)
sc.pp.neighbors(adata_rna)
sc.tl.umap(adata_rna, min_dist=0.1)
sc.pl.umap(adata_rna, color=['source', 'cell_type'])
```

```
... storing 'cell_type' as categorical
... storing 'source' as categorical
```



Perform UMAP visualization for scATAC latent representations after vertical integration. The cell types of scATAC-seq now are more distinguishable than before.

[10]:
```
sc.pp.neighbors(adata_peaks, use_rep='latent')
sc.tl.umap(adata_peaks, min_dist=0.1)
sc.pl.umap(adata_peaks, color=['source', 'cell_type'])
```



[ ]:

1.2.6 Diagonal integration with contrastive learning

A integration task is called diagonal integration if different modalities share no correspondence information, either among cells or features, e.g., scATAC peaks and scRNA genes, we can use label annotations as prior information to improve the performance.

[1]:
```
import uniport as up
import scanpy as sc
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
up.__version__
```

[1]:
```
'1.1.1'
```

Load PBMC scATAC peaks and scRNA counts.

[2]:
```
adata_peaks = sc.read_h5ad('PBMC/atac_peaks.h5ad')
# adata_peaks = up.load_file('PBMC/pbmc_signac_peaks.tsv.gz')
adata_rna = up.load_file('PBMC/RNA_count.txt')
```

[3]:
```
labels = pd.read_csv('PBMC/meta.txt', sep='\t')
celltype = labels['cluster'].values
```

[4]:
```
adata_peaks.obs['cell_type'] = celltype
adata_peaks.obs['domain_id'] = 0
adata_peaks.obs['domain_id'] = adata_peaks.obs['domain_id'].astype('category')
adata_peaks.obs['source'] = 'ATAC-peaks'

adata_rna.obs['cell_type'] = celltype
adata_rna.obs['domain_id'] = 1
adata_rna.obs['domain_id'] = adata_rna.obs['domain_id'].astype('category')
adata_rna.obs['source'] = 'RNA'

print(adata_rna.obs)
print(adata_peaks.obs)
```

```
                    cell_type domain_id source
AAACAGCCAAGGAATC.1  CD4 Naive         1    RNA
AAACAGCCAATCCCTT.1   CD4 Tmem         1    RNA
AAACAGCCAATGCGCT.1  CD4 Naive         1    RNA
AAACAGCCACACTAAT.1  CD8 Naive         1    RNA
AAACAGCCACCAACCG.1  CD8 Naive         1    RNA
...                       ...       ...    ...
TTTGTTGGTGACATGC.1  CD8 Naive         1    RNA
TTTGTTGGTGTTAAAC.1  CD8 Naive         1    RNA
TTTGTTGGTTAGGATT.1         NK         1    RNA
TTTGTTGGTTGGTTAG.1   CD4 Tmem         1    RNA
TTTGTTGGTTTGCAGA.1   CD8 Tmem         1    RNA

[11259 rows x 3 columns]
                    cell_type domain_id      source
AAACAGCCAAGGAATC-1  CD4 Naive         0  ATAC-peaks
AAACAGCCAATCCCTT-1   CD4 Tmem         0  ATAC-peaks
AAACAGCCAATGCGCT-1  CD4 Naive         0  ATAC-peaks
AAACAGCCACACTAAT-1  CD8 Naive         0  ATAC-peaks
AAACAGCCACCAACCG-1  CD8 Naive         0  ATAC-peaks
...                       ...       ...         ...
TTTGTTGGTGACATGC-1  CD8 Naive         0  ATAC-peaks
TTTGTTGGTGTTAAAC-1  CD8 Naive         0  ATAC-peaks
TTTGTTGGTTAGGATT-1         NK         0  ATAC-peaks
TTTGTTGGTTGGTTAG-1   CD4 Tmem         0  ATAC-peaks
TTTGTTGGTTTGCAGA-1   CD8 Tmem         0  ATAC-peaks

[11259 rows x 3 columns]
```

Preprocess scATAC peaks using `up.TFIDF_LSI`.

[5]:
```
adata_peaks.X[adata_peaks.X>1] = 1
sc.pp.normalize_total(adata_peaks)
up.TFIDF_LSI(adata_peaks)
scaler = MinMaxScaler()
adata_peaks.obsm['X_lsi'] = scaler.fit_transform(adata_peaks.obsm['X_lsi'])
```

[6]:
```
sc.pp.normalize_total(adata_rna)
sc.pp.log1p(adata_rna)
sc.pp.highly_variable_genes(adata_rna, n_top_genes=2000, inplace=False, subset=True)
up.batch_scale(adata_rna)
```

Diagonal integration without prior information.

[7]:
```
adata1 = up.Run(adatas=[adata_peaks, adata_rna], use_rep=['X_lsi','X'], mode='d')
```

```
Dataset 0: ATAC-peaks
AnnData object with n_obs × n_vars = 11259 × 131364
    obs: 'cell_type', 'domain_id', 'source'
    obsm: 'X_lsi'
Dataset 1: RNA
AnnData object with n_obs × n_vars = 11259 × 2000
    obs: 'cell_type', 'domain_id', 'source'
```

(continues on next page)

(continued from previous page)

```
    var: 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
Reference dataset is dataset 1
```

```
Epochs: 100%|| 345/345 [15:29<00:00,  2.69s/it, recon_loss=1505.593,kl_loss=8.341,ot_
↪loss=4.676]
```

Construct prior correspondence information with label annotation.

[8]:
```
alpha=2
prior = up.get_prior(adata_peaks.obs['cell_type'].values, adata_rna.obs['cell_type'].
↪values, alpha=alpha)
```

Diagonal integration with correspondence information.

[9]:
```
adata2 = up.Run(adatas=[adata_peaks, adata_rna], use_rep=['X_lsi','X'], prior=[prior],␣
↪mode='d', lambda_ot=5)
```

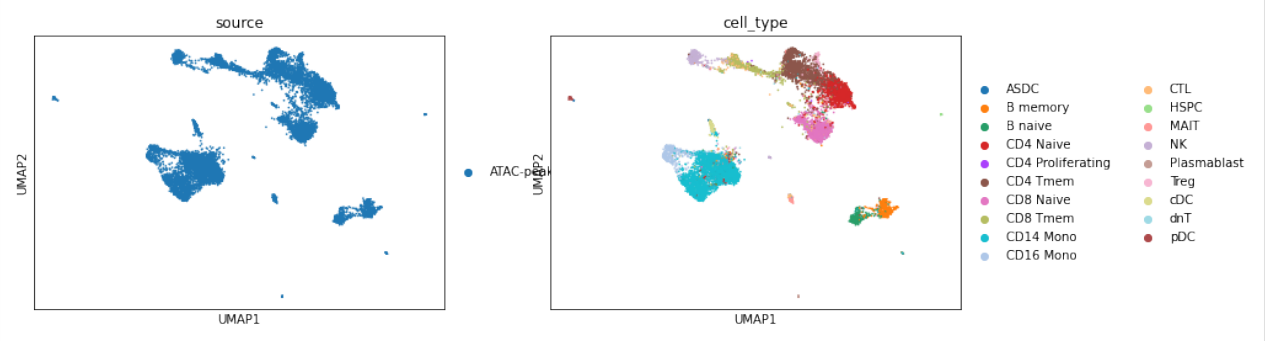
```
Dataset 0: ATAC-peaks
AnnData object with n_obs × n_vars = 11259 × 131364
    obs: 'cell_type', 'domain_id', 'source'
    obsm: 'X_lsi', 'latent'
Dataset 1: RNA
AnnData object with n_obs × n_vars = 11259 × 2000
    obs: 'cell_type', 'domain_id', 'source'
    var: 'highly_variable', 'means', 'dispersions', 'dispersions_norm'
    uns: 'log1p', 'hvg'
    obsm: 'latent'
Reference dataset is dataset 1
```

```
Epochs: 100%|| 345/345 [15:59<00:00,  2.78s/it, recon_loss=1510.078,kl_loss=6.985,ot_
↪loss=19.150]
```

UMAP visualization of scATAC peaks.

[10]:
```
sc.pp.neighbors(adata_peaks, use_rep='X_lsi')
sc.tl.umap(adata_peaks, min_dist=0.1)
sc.pl.umap(adata_peaks, color=['source', 'cell_type'])
```

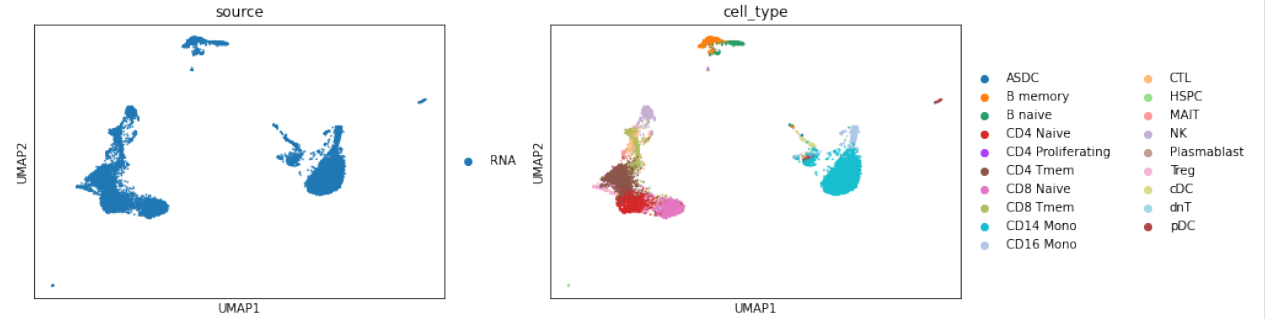
```
... storing 'cell_type' as categorical
... storing 'source' as categorical
```



UMAP visualization of scRNA genes.

[11]:
```
sc.pp.pca(adata_rna)
sc.pp.neighbors(adata_rna)
sc.tl.umap(adata_rna, min_dist=0.1)
sc.pl.umap(adata_rna, color=['source', 'cell_type'])
```

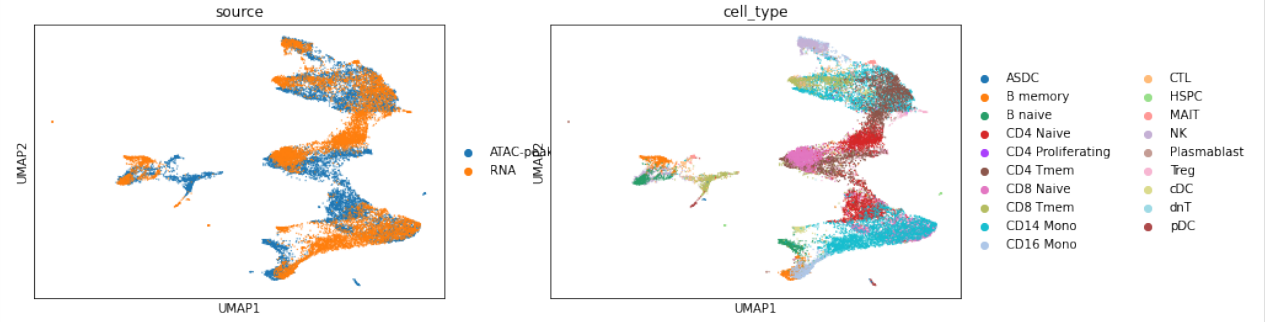
```
... storing 'cell_type' as categorical
... storing 'source' as categorical
```



UMAP visualization after uniPort diagonal integration.

[12]:
```
sc.pp.neighbors(adata1, use_rep='latent')
sc.tl.umap(adata1, min_dist=0.1)
sc.pl.umap(adata1, color=['source', 'cell_type'])
```

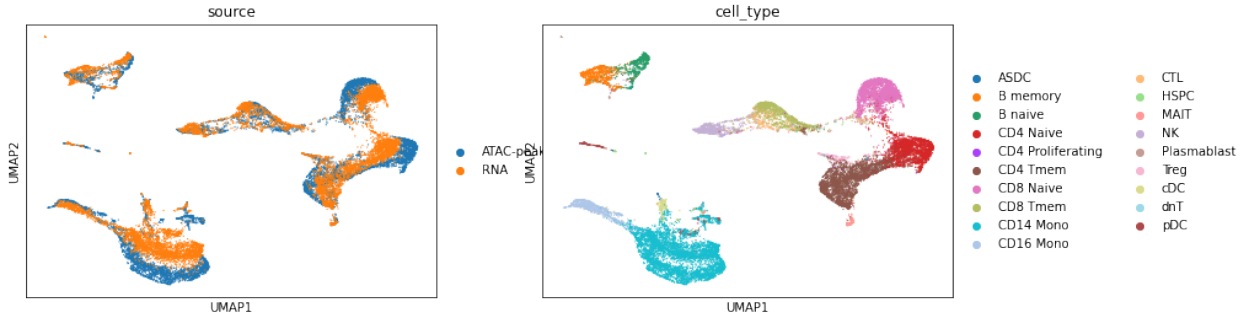
```
... storing 'cell_type' as categorical
... storing 'source' as categorical
```



UMAP visualization after uniPort diagonal integration with **contrastive learning**.

[13]:
```
sc.pp.neighbors(adata2, use_rep='latent')
sc.tl.umap(adata2, min_dist=0.1)
sc.pl.umap(adata2, color=['source', 'cell_type'])
```

```
... storing 'cell_type' as categorical
... storing 'source' as categorical
```



1.3 API and modules

1.3.1 Function

Run([adatas, adata_cm, mode, lambda_s, ...])	Run data integration
label_reweight(celltype)	Reweight labels to make all cell types share the same total weight
load_file(path)	Load single cell dataset from file
filter_data(adata[, min_features, ...])	Filter cells and genes
batch_scale(adata[, use_rep, chunk_size])	Batch-specific scale data

uniport.Run

uniport.**Run**(*adatas=None*, *adata_cm=None*, *mode='h'*, *lambda_s=0.5*, *lambda_recon=1.0*, *lambda_kl=0.5*, *lambda_ot=1.0*, *iteration=30000*, *ref_id=None*, *save_OT=False*, *use_rep=['X', 'X']*, *out='latent'*, *label_weight=None*, *reg=0.1*, *reg_m=1.0*, *batch_size=256*, *lr=0.0002*, *enc=None*, *gpu=0*, *prior=None*, *loss_type='BCE'*, *outdir='output/'*, *input_id=0*, *pred_id=1*, *seed=124*, *num_workers=4*, *patience=30*, *batch_key='domain_id'*, *source_name='source'*, *model_info=False*, *verbose=False*)

Run data integration

Parameters

- **adatas** – List of AnnData matrices, e.g. [adata1, adata2].
- **adata_cm** – AnnData matrices containing common genes.
- **mode** – Choose from [‘h’, ‘v’, ‘d’] If ‘h’, integrate data with common genes (Horizontal integration) If ‘v’, integrate data profiled from the same cells (Vertical integration) If ‘d’, inetrgate data without common genes (Diagonal integration) Default: ‘h’.
- **lambda_s** – Balanced parameter for common and specific genes. Default: 0.5
- **lambda_recon** – Balanced parameter for reconstruct term. Default: 1.0
- **lambda_kl** – Balanced parameter for KL divergence. Default: 0.5

- **lambda_ot** – Balanced parameter for OT. Default: 1.0
- **iteration** – Max iterations for training. Training one batch_size samples is one iteration. Default: 30000
- **ref_id** – Id of reference dataset. Default: None
- **save_OT** – If True, output a global OT plan. Need more memory. Default: False
- **use_rep** – Use '.X' or '.obsm'. For mode='d' only. If use_rep=['X','X'], use 'adatas[0].X' and 'adatas[1].X' for integration. If use_rep=['X','X_lsi'], use 'adatas[0].X' and 'adatas[1].obsm['X_lsi']' for integration. If use_rep=['X_pca', 'X_lsi'], use 'adatas[0].obsm['X_pca']' and 'adatas[1].obsm['X_lsi']' for integration. Default: ['X','X']
- **out** – Output of uniPort. Choose from ['latent', 'project', 'predict']. If out=='latent', train the network and output cell embeddings. If out=='project', project data into the latent space and output cell embeddings. If out=='predict', project data into the latent space and output cell embeddings through a specified decoder. Default: 'latent'.
- **label_weight** – Prior-guided weighted vectors. Default: None
- **reg** – Entropy regularization parameter in OT. Default: 0.1
- **reg_m** – Unbalanced OT parameter. Larger values means more balanced OT. Default: 1.0
- **batch_size** – Number of samples per batch to load. Default: 256
- **lr** – Learning rate. Default: 2e-4
- **enc** – Structure of encoder
- **gpu** – Index of GPU to use if GPU is available. Default: 0
- **prior** – Prior correspondence matrix. Default: None
- **loss_type** – type of loss. 'BCE', 'MSE' or 'L1'. Default: 'BCE'
- **outdir** – Output directory. Default: 'output/'
- **input_id** – Only used when mode=='d' and out=='predict' to choose a encoder to project data. Default: 0
- **pred_id** – Only used when out=='predict' to choose a decoder to predict data. Default: 1
- **seed** – Random seed for torch and numpy. Default: 124
- **patience** – early stopping patience. Default: 10
- **batch_key** – Name of batch in AnnData. Default: domain_id
- **source_name** – Name of source in AnnData. Default: source
- **rep_celltype** – Names of cell-type annotation in AnnData. Default: 'cell_type'
- **umap** – If True, perform UMAP for visualization. Default: False
- **model_info** – If True, show structures of encoder and decoders.
- **verbose** – Verbosity, True or False. Default: False
- **assess** – If True, calculate the entropy_batch_mixing score and silhouette score to evaluate integration results. Default: False
- **show** – If True, show the UMAP visualization of latent space. Default: False

Returns

- *adata.h5ad* – The AnnData matrice after integration. The representation of the data is stored at adata.obsm['latent'], adata.obsm['project'] or adata.obsm['predict'].
- *checkpoint* – model.pt contains the variables of the model and config.pt contains the parameters of the model.
- *log.txt* – Records model parameters.
- *umap.pdf* – UMAP plot for visualization if umap=True.

uniport.label_reweight

uniport.**label_reweight**(*celltype*)

Reweight labels to make all cell types share the same total weight

Parameters
celltype – cell labels

Returns
a vector of weights of cells

Return type
torch.tensor

uniport.load_file

uniport.**load_file**(*path*)

Load single cell dataset from file

Parameters
path – the path store the file

Return type
AnnData

uniport.filter_data

uniport.**filter_data**(*adata: AnnData*, *min_features: int = 0*, *min_cells: int = 0*, *log=None*)

Filter cells and genes

Parameters

- **adata** – An AnnData matrice of shape n_obs × n_vars. Rows correspond to cells and columns to genes.
- **min_features** – Filtered out cells that are detected in less than n genes. Default: 0.
- **min_cells** – Filtered out genes that are detected in less than n cells. Default: 0.

uniport.batch_scale

uniport.**batch_scale**(*adata*, *use_rep='X'*, *chunk_size=20000*)

Batch-specific scale data

Parameters

- **adata** – AnnData
- **use_rep** – use '.X' or '.obsm'
- **chunk_size** – chunk large data into small chunks

1.3.2 DataLoader

data_loader.SingleCellDataset(data, batch)	
data_loader.SingleCellDataset_vertical(adatas)	
data_loader.load_data(adatas[, mode, ...])	Load data for training.

uniport.data_loader.SingleCellDataset

class uniport.data_loader.**SingleCellDataset**(*data*, *batch*)

__init__(*data*, *batch*)

Methods

__init__(data, batch)	

uniport.data_loader.SingleCellDataset_vertical

class uniport.data_loader.**SingleCellDataset_vertical**(*adatas*)

__init__(*adatas*)

Methods

__init__(adatas)	

uniport.data_loader.load_data

uniport.data_loader.**load_data**(*adatas*, *mode='h'*, *use_rep=['X', 'X']*, *num_cell=None*, *max_gene=None*, *adata_cm=None*, *use_specific=False*, *domain_name='domain_id'*, *batch_size=256*, *drop_last=True*, *shuffle=True*, *num_workers=4*)

Load data for training.

Parameters

- **adatas** – A list of AnnData matrice.
- **mode** – training mode. Choose between [‘h’, ‘d’, ‘v’].
- **use_rep** – use ‘.X’ or ‘.obsm’.
- **num_cell** – numbers of cells of each adata in adatas.
- **max_gene** – maximum number of genes of each adata in adatas.
- **adata_cm** – adata with common genes of adatas.
- **use_specific** – use dataset-specific genes.
- **domain_name** – domain name of each adata in adatas.
- **batch_size** – size of each mini batch for training.
- **drop_last** – drop the last samples that not up to one batch.
- **shuffle** – shuffle the data
- **num_workers** – number parallel load processes according to cpu cores.

Returns

- *trainloader* – data loader for training
- *testloader* – data loader for testing

1.3.3 Model

model.vae.VAE(enc, dec, ref_id, n_domain, mode)	Variational Autoencoder framework
model.layer.DSBatchNorm(num_features, n_domain)	Domain-specific Batch Normalization
model.layer.Block(input_dim, output_dim[, ...])	Basic block consist of:
model.layer.NN(input_dim, cfg)	Neural network consist of multi Blocks
model.layer.Encoder(input_dim, cfg, mode)	VAE Encoder
model.layer.Decoder(z_dim, cfg)	VAE Decoder
model.loss.kl_div(mu, var[, weight])	
model.loss.distance_matrix(pts_src, pts_dst)	Returns the matrix of ‖x_i-y_j‖_p^p.
model.loss.distance_gmm(mu_src, mu_dst, ...)	Calculate a Wasserstein distance matrix between the gmm distributions with diagonal variances
model.loss.unbalanced_ot(tran, mu1, var1, ...)	Calculate a unbalanced optimal transport matrix between mini batches.
model.utils.onehot(y, n)	Make the input tensor one hot tensors
model.utils.EarlyStopping([patience, ...])	Early stops the training if loss doesn't improve after a given patience.

uniport.model.vae.VAE

class uniport.model.vae.**VAE**(*enc*, *dec*, *ref_id*, *n_domain*, *mode*)

Variational Autoencoder framework

__init__(*enc*, *dec*, *ref_id*, *n_domain*, *mode*)

Parameters

- **enc** – Encoder structure config
- **dec** – Decoder structure config
- **ref_id** – ID of reference dataset
- **n_domain** – The number of different domains
- **mode** – Choose from ['h', 'v', 'd']

Methods

__init__(enc, dec, ref_id, n_domain, mode)	**param enc** Encoder structure config
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.
bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
encodeBatch(dataloader, num_gene[, mode, ...])	Inference
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
fit(dataloader, tran, num_cell, num_gene[, ...])	train VAE
float()	Casts all floating point parameters and buffers to float datatype.
forward(*input)	Defines the computation performed at every call.
get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.

continues on next page

Table 1 – continued from previous page

load_model(path)	Load trained model parameters dictionary.
load_state_dict(state_dict[, strict])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook)	Registers a forward hook on the module.
register_forward_pre_hook(hook)	Registers a forward pre-hook on the module.
register_full_backward_hook(hook)	Registers a backward hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args[, destination, prefix, ...])	Returns a dictionary containing references to the whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device)	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Sets gradients of all model parameters to zero.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
dump_patches	

uniport.model.layer.DSBatchNorm

class uniport.model.layer.**DSBatchNorm**(*num_features*, *n_domain*, *eps=1e-05*, *momentum=0.1*)

Domain-specific Batch Normalization

__init__(*num_features*, *n_domain*, *eps=1e-05*, *momentum=0.1*)

Parameters

- **num_features** – dimension of the features
- **n_domain** – domain number

Methods

__init__(num_features, n_domain[, eps, momentum])	**param num_features** dimension of the features
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.
bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(x, y)	Defines the computation performed at every call.
get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.

continues on next page

Table 2 – continued from previous page

half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook)	Registers a forward hook on the module.
register_forward_pre_hook(hook)	Registers a forward pre-hook on the module.
register_full_backward_hook(hook)	Registers a backward hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
reset_parameters()	
reset_running_stats()	
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args[, destination, prefix, ...])	Returns a dictionary containing references to the whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device)	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Sets gradients of all model parameters to zero.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
dump_patches	

uniport.model.layer.Block

class uniport.model.layer.**Block**(*input_dim*, *output_dim*, *norm=''*, *act=''*, *dropout=0*)

Basic block consist of:
fc -> bn -> act -> dropout

__init__(*input_dim*, *output_dim*, *norm=''*, *act=''*, *dropout=0*)

Parameters

- **input_dim** – dimension of input
- **output_dim** – dimension of output
- **norm** –

  batch normalization,

  - '' represent no batch normalization
  - 1 represent regular batch normalization
  - int>1 represent domain-specific batch normalization of n domain
- **act** –

  activation function,

  - relu -> nn.ReLU
  - rrelu -> nn.RReLU
  - sigmoid -> nn.Sigmoid()
  - leaky_relu -> nn.LeakyReLU()
  - tanh -> nn.Tanh()
  - '' -> None
- **dropout** – dropout rate

Methods

__init__(input_dim, output_dim[, norm, act, ...])	**param input_dim** dimension of input
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.

continues on next page

Table 3 – continued from previous page

bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(x[, y])	Defines the computation performed at every call.
get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook)	Registers a forward hook on the module.
register_forward_pre_hook(hook)	Registers a forward pre-hook on the module.
register_full_backward_hook(hook)	Registers a backward hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.

continues on next page

Table 3 – continued from previous page

set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args[, destination, prefix, ...])	Returns a dictionary containing references to the whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device)	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Sets gradients of all model parameters to zero.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
dump_patches	

uniport.model.layer.NN

class uniport.model.layer.**NN**(*input_dim*, *cfg*)

Neural network consist of multi Blocks

__init__(*input_dim*, *cfg*)

Parameters

- **input_dim** – input dimension
- **cfg** – model structure configuration, 'fc' -> fully connected layer

Example

```
>>> latent_dim = 10
>>> dec_cfg = [['fc', x_dim, n_domain, 'sigmoid']]
>>> decoder = NN(latent_dim, dec_cfg)
```

Methods

__init__(input_dim, cfg)	**param input_dim** input dimension
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.

continues on next page

Table 4 – continued from previous page

bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(x[, y])	Defines the computation performed at every call.
get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook)	Registers a forward hook on the module.
register_forward_pre_hook(hook)	Registers a forward pre-hook on the module.
register_full_backward_hook(hook)	Registers a backward hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.

continues on next page

Table 4 – continued from previous page

set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args[, destination, prefix, ...])	Returns a dictionary containing references to the whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device)	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Sets gradients of all model parameters to zero.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
dump_patches	

uniport.model.layer.Encoder

class uniport.model.layer.**Encoder**(*input_dim*, *cfg*, *mode*)

VAE Encoder

__init__(*input_dim*, *cfg*, *mode*)

Parameters

- **input_dim** – input dimension
- **cfg** – encoder configuration, e.g. enc_cfg = [[‘fc’, 1024, 1, ‘relu’],[‘fc’, 10, ‘’, ‘’]]
- **mode** – training mode. [‘h’, ‘d’, ‘v’]

Methods

__init__(input_dim, cfg, mode)	**param input_dim** input dimension
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.
bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
cpu()	Moves all model parameters and buffers to the CPU.

continues on next page

Table 5 – continued from previous page

cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(x, domain[, y])	
get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook)	Registers a forward hook on the module.
register_forward_pre_hook(hook)	Registers a forward pre-hook on the module.
register_full_backward_hook(hook)	Registers a backward hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
reparameterize(mu, var)	
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()

continues on next page

Table 5 – continued from previous page

state_dict(*args[, destination, prefix, ...])	Returns a dictionary containing references to the whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device)	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Sets gradients of all model parameters to zero.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
dump_patches	

uniport.model.layer.Decoder

class uniport.model.layer.**Decoder**(*z_dim*, *cfg*)

VAE Decoder

__init__(*z_dim*, *cfg*)

Parameters

- **z_dim** – latent dimension
- **cfg** – decoder configuration, e.g. dec_cfg = [['fc', adatas[i].obsm[obsm[i]].shape[1], 1, 'sigmoid']]

Methods

__init__(z_dim, cfg)	**param z_dim** latent dimension
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.
bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.

continues on next page

Table 6 – continued from previous page

extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(z, domain[, y])	
get_buffer(target)	Returns the buffer given by target if it exists, otherwise throws an error.
get_extra_state()	Returns any extra state to include in the module's state_dict.
get_parameter(target)	Returns the parameter given by target if it exists, otherwise throws an error.
get_submodule(target)	Returns the submodule given by target if it exists, otherwise throws an error.
half()	Casts all floating point parameters and buffers to half datatype.
ipu([device])	Moves all model parameters and buffers to the IPU.
load_state_dict(state_dict[, strict])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix, remove_duplicate])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook)	Registers a forward hook on the module.
register_forward_pre_hook(hook)	Registers a forward pre-hook on the module.
register_full_backward_hook(hook)	Registers a backward hook on the module.
register_load_state_dict_post_hook(hook)	Registers a post hook to be run after module's load_state_dict is called.
register_module(name, module)	Alias for add_module().
register_parameter(name, param)	Adds a parameter to the module.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
set_extra_state(state)	This function is called from load_state_dict() to handle any extra state found within the *state_dict*.
share_memory()	See torch.Tensor.share_memory_()
state_dict(*args[, destination, prefix, ...])	Returns a dictionary containing references to the whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
to_empty(*, device)	Moves the parameters and buffers to the specified device without copying storage.
train([mode])	Sets the module in training mode.

continues on next page

Table 6 – continued from previous page

type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Sets gradients of all model parameters to zero.

Attributes

T_destination	alias of TypeVar('T_destination', bound=Dict[str, Any])
dump_patches	

uniport.model.loss.kl_div

uniport.model.loss.**kl_div**(*mu*, *var*, *weight=None*)

uniport.model.loss.distance_matrix

uniport.model.loss.**distance_matrix**(*pts_src: Tensor*, *pts_dst: Tensor*, *p: int = 2*)

Returns the matrix of ||x_i-y_j||_p^p.

Parameters

- **pts_src** – [R, D] matrix
- **pts_dst** – C, D] matrix
- **p** – p-norm

Returns
distance matrix

Return type
[R, C] matrix

uniport.model.loss.distance_gmm

uniport.model.loss.**distance_gmm**(*mu_src: Tensor*, *mu_dst: Tensor*, *var_src: Tensor*, *var_dst: Tensor*)

Calculate a Wasserstein distance matrix between the gmm distributions with diagonal variances

Parameters

- **mu_src** – [R, D] matrix, the means of R Gaussian distributions
- **mu_dst** – [C, D] matrix, the means of C Gaussian distributions
- **logvar_src** – [R, D] matrix, the log(variance) of R Gaussian distributions
- **logvar_dst** – [C, D] matrix, the log(variance) of C Gaussian distributions

Returns
distance matrix

Return type
[R, C] matrix

uniport.model.loss.unbalanced_ot

uniport.model.loss.**unbalanced_ot**(*tran*, *mu1*, *var1*, *mu2*, *var2*, *reg=0.1*, *reg_m=1.0*, *Couple=None*, *device='cpu'*, *idx_q=None*, *idx_r=None*, *query_weight=None*, *ref_weight=None*)

Calculate a unbalanced optimal transport matrix between mini batches.

Parameters

- **tran** – transport matrix between the two batches sampling from the global OT matrix.
- **mu1** – mean vector of batch 1 from the encoder
- **var1** – standard deviation vector of batch 1 from the encoder
- **mu2** – mean vector of batch 2 from the encoder
- **var2** – standard deviation vector of batch 2 from the encoder
- **reg** – Entropy regularization parameter in OT. Default: 0.1
- **reg_m** – Unbalanced OT parameter. Larger values means more balanced OT. Default: 1.0
- **Couple** – prior information about weights between cell correspondence. Default: None
- **device** – training device
- **idx_q** – domain_id of query batch
- **idx_r** – domain_id of reference batch
- **query_weight** – reweighted vectors of query batch
- **ref_weight** – reweighted vectors of reference batch

Returns

- *float* – minibatch unbalanced optimal transport loss
- *matrix* – minibatch unbalanced optimal transport matrix

uniport.model.utils.onehot

uniport.model.utils.**onehot**(*y*, *n*)

Make the input tensor one hot tensors

Parameters

- **y** – input tensors
- **n** – number of classes

Return type
Tensor

uniport.model.utils.EarlyStopping

class uniport.model.utils.**EarlyStopping**(*patience=10*, *verbose=False*, *checkpoint_file=''*)

Early stops the training if loss doesn't improve after a given patience.

__init__(*patience=10*, *verbose=False*, *checkpoint_file=''*)

Parameters

- **patience** – How long to wait after last time loss improved. Default: 30
- **verbose** – If True, prints a message for each loss improvement. Default: False

Methods

__init__([patience, verbose, checkpoint_file])	**param patience** How long to wait after last time loss improved. Default: 30
save_checkpoint(loss, model)	Saves model when loss decrease.

1.3.4 Evaluation

metrics.batch_entropy_mixing_score(data, batches)	Calculate batch entropy mixing score
metrics.silhouette(X, cell_type[, metric, scale])	Wrapper for sklearn silhouette function values range from [-1, 1] with
metrics.label_transfer(ref, query[, rep, label])	Label transfer

uniport.metrics.batch_entropy_mixing_score

uniport.metrics.**batch_entropy_mixing_score**(*data*, *batches*, *n_neighbors=100*, *n_pools=100*, *n_samples_per_pool=100*)

Calculate batch entropy mixing score

Algorithm

- 1. Calculate the regional mixing entropies at the location of 100 randomly chosen cells from all batches
- 2. Define 100 nearest neighbors for each randomly chosen cell
- 3. Calculate the mean mixing entropy as the mean of the regional entropies
- 4. Repeat above procedure for 100 iterations with different randomly chosen cells.

param data
np.array of shape nsamples x nfeatures.

param batches
batch labels of nsamples.

param n_neighbors
The number of nearest neighbors for each randomly chosen cell. By default, n_neighbors=100.

param n_samples_per_pool
The number of randomly chosen cells from all batches per iteration. By default, n_samples_per_pool=100.

param n_pools
The number of iterations with different randomly chosen cells. By default, n_pools=100.

rtype
Batch entropy mixing score

uniport.metrics.silhouette

uniport.metrics.**silhouette**(*X*, *cell_type*, *metric='euclidean'*, *scale=True*)

Wrapper for sklearn silhouette function values range from [-1, 1] with
1 being an ideal fit 0 indicating overlapping clusters and -1 indicating misclassified cells

By default, the score is scaled between 0 and 1. This is controlled *scale=True*

Parameters

- **group_key** – key in adata.obs of cell labels
- **embed** – embedding key in adata.obsm, default: 'X_pca'
- **scale** – default True, scale between 0 (worst) and 1 (best)

uniport.metrics.label_transfer

uniport.metrics.**label_transfer**(*ref*, *query*, *rep='latent'*, *label='cell_type'*)

Label transfer

Parameters

- **ref** – reference containing the projected representations and labels
- **query** – query data to transfer label
- **rep** – representations to train the classifier. Default is *latent*
- **label** – label name. Defautl is *celltype* stored in ref.obs

Return type
transfered label

1.3.5 Logger

logger.create_logger([name, ch, fh, levelname])

uniport.logger.create_logger

uniport.logger.**create_logger**(*name=''*, *ch=True*, *fh=''*, *levelname=20*)

1.4 Release

1.4.1 v1.1.1

- **add model_log parameter to *Run()* function.**
  - If model_log=True, show structures of encoder and decoders. Default: False.
- fix bugs

1.4.2 v1.1.0

- ***Get_label_Prior()* function changes:**
  - **Get_label_Prior()** -> **get_prior()**
  - set **alpha=2** as default
- ***Run()* function parameter changes:**
  - **labmda_recon** -> **lambda_recon**
  - **Prior** -> **prior**
  - **max_iteration** -> **iteration**
  - add **use_rep** for **mode=d**
- add *TFIDF_LSI()* function for scATAC preprecess.
- **remove AnnData returns in *filter_data()* and *batch_scale()* functions.**
  - e.g., change **adata=up.filter_data(adata)** to **up.filter_data(adata)**.

1.4.3 v1.0.5

Parameter fixes.

The original paper: a unified single-cell data integration framework with optimal transport

Website and documentation: https://uniport.readthedocs.io

Source Code (MIT): https://github.com/caokai1073/uniport

All data before and after processing in Examples are available at source data link

Author's Homepage: www.caokai.site

1.5 Installation

The **uniport** package can be installed via pip:

```
pip3 install uniport
```

1.6 Main function

uniport.Run(...)

Key parameters includes:

- **adatas**: List of AnnData matrices for each dataset.
- **adata_cm**: AnnData matrix containing common genes from different datasets.
- **mode**: Choose from ['h', 'v', 'd'] If 'mode=h', integrate data with common genes (Horizontal integration). If 'mode=v', integrate data profiled from the same cells (Vertical integration). If 'mode=d', inetrgate data without common genes (Diagonal integration). Default: 'h'.
- **lambda_s**: balanced parameter for common and specific genes. Default: 0.5
- **lambda_recon**: balanced parameter for reconstruct term. Default: 1.0
- **lambda_kl**: balanced parameter for KL divergence. Default: 0.5
- **lambda_ot**: balanced parameter for OT. Default: 1.0
- **iteration**: max iterations for training. Training one batch_size samples is one iteration. Default: 30000
- **ref_id**: id of reference dataset. Default: The domain_id of last dataset
- **save_OT**: if True, output a global OT plan. Need more memory. Default: False
- **out**: output of uniPort. Choose from ['latent', 'project', 'predict']. If out=='latent', train the network and output cell embeddings. If out=='project', project data into the latent space and output cell embeddings. If out=='predict', project data into the latent space and output cell embeddings through a specified decoder. Default: 'latent'

```
import uniport as up
import scanpy as sc

# HVG: highly variable genes
adata1 = sc.read_h5ad('adata1.h5ad') # preprocessed data with data1 specific HVG
adata2 = sc.read_h5ad('adata2.h5ad') # preprocessed data with data2 specific HVG, as␣
↪reference data
adata_cm = sc.read_h5ad('adata_cm.h5ad') # preprocesssed data with common HVG

# integration with both common and dataset-specific genes
adata = up.Run(adatas=[adata1, adata2], adata_cm=adata_cm)
# save global optimal transport matrix
adata, OT = up.Run(adatas=[adata1, adata2], adata_cm=adata_cm, save_OT=True)
# integration with only common genes
adata = up.Run(adata_cm=adata_cm)

# integration without common genes
adata = up.Run(adatas=[adata1, adata2], mode='d')
```

(continues on next page)

(continued from previous page)

```
# integration with paired datasets
adata = up.Run(adatas=[adata1, adata2], mode='v')
```

1.7 Citation

```
@article{Cao2022.02.14.480323,
    author = {Cao, Kai and Gong, Qiyu and Hong, Yiguang and Wan, Lin},
    title = {uniPort: a unified computational framework for single-cell data integration␣
↪with optimal transport},
    year = {2022},
    doi = {10.1101/2022.02.14.480323},
    publisher = {Cold Spring Harbor Laboratory},
    journal = {bioRxiv}}
```

PYTHON MODULE INDEX

u

INDEX

Symbols

B

C

D

E

F

K

L

M

N

O

R

S

U

V