otscomics

Geert-Jan Huizing, Gabriel Peyré, Laura Cantini

Feb 03, 2023

GETTING STARTED

CHAPTER

ONE

OPTIMAL TRANSPORT FOR SINGLE-CELL OMICS

This Jupyter Notebook will walk you trough the code to replicate the experiments from our research on applying OT as a loss function in between single-cell omics data. Bioinformatics paper.

The code is designed to be run on GPU. If you do not have access to a GPU you may want to use a free Google Colab instance to run this notebook.

1.1 Imports

Uncomment this if running on Google Colab

[ ]:
```
# !git clone https://github.com/ComputationalSystemsBiology/OT-scOmics.git
# !pip install scanpy
# !pip install leidenalg
# !pip install otscomics
```

General libraries to load data, do computations and make plots.

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

Function to compute pairwise distance matrices for various functions (e.g. euclidean, cosine ...).

[2]:
```
from scipy.spatial.distance import cdist
```

Clustering models used for evaluation.

[3]:
```
from sklearn.cluster import AgglomerativeClustering, SpectralClustering
```

Metrics used for scoring.

[4]:
```
from sklearn.metrics import silhouette_score
from sklearn.metrics import adjusted_rand_score
from sklearn.metrics import adjusted_mutual_info_score
from sklearn.metrics import normalized_mutual_info_score
```

Library `tqdm` makes pretty progress bars.

[5]:
```
from tqdm import tqdm
```

Our library to compute the OT distance matrix.

[6]:
```
import otscomics
```

Scanpy and AnnData container.

[7]:
```
import scanpy as sc
import anndata as ad
```

1.2 Load preprocessed data

We provide preprocessed data as compressed csv files, with features as rows and cell as columns. The target classes (cell type or cell line) are given in the column name.

[8]:
```
# Load the data.
data = pd.read_csv('OT-scOmics/data/liu_scrna_preprocessed.csv.gz', index_col=0)
```

[9]:
```
# Retrieve the clusters.
clusters = np.array([col.split('_')[-1] for col in data.columns])
idx = np.argsort(clusters) # Sorted indices (for visulization)
```

[10]:
```
# Display unique clusters.
np.unique(clusters)
```

[10]:
```
array(['HCT', 'Hela', 'K562'], dtype='<U4')
```

[11]:
```
data = data.iloc[np.argsort(data.std(1))[::-1][:1_000]]
```

Converting the dataset to an AnnData for the rest of the analysis.

[12]:
```
adata = ad.AnnData(data.T)
adata.obs['cell_line'] = clusters
```

1.3 Compute distance matrix

In this section we will compute distance matrices between cells using both Optimal Transport and the Euclidean distance.

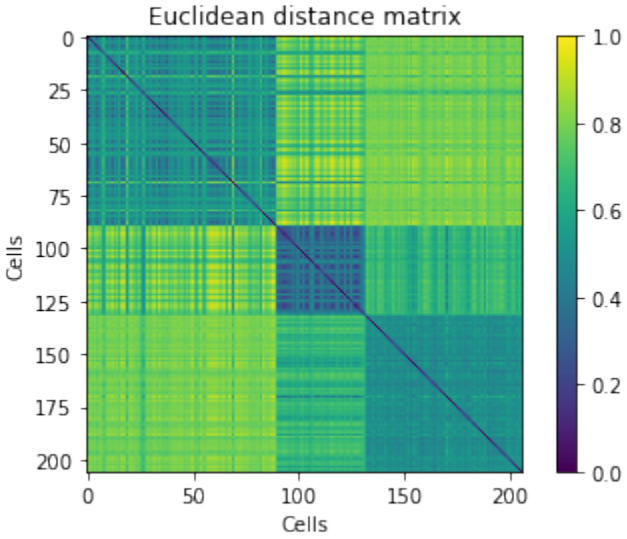
1.3.1 Compute baseline distance matrix

Let us start by computing the euclidean distance matrix. You may change the following line of code using `cityblock`, `cosine`, `correlation` or any Scipy-compatible distance.

[13]:
```
# Replace `euclidean` with any scipy distance metric.
D_eu = cdist(adata.X, adata.X, metric='euclidean')
D_eu /= D_eu.max()
```

Displaying the distance matrix, with cells ordered by class. Clusters should be visible in the form of diagonal blocks of close distances.

[14]:
```
plt.imshow(D_eu[idx][:,idx])
plt.title('Euclidean distance matrix')
plt.xlabel('Cells')
plt.ylabel('Cells')
plt.colorbar()
plt.show()
```



1.3.2 Compute OT distance matrix

Now on the OT distance matrix. We first need to normalize cells (divide columns by their sum). This is because Optimal Transport works on probability distributions. We then compute a cost matrix (you may change `cosine` by another function) and the associated OT distance matrix.

[15]:
```
# Per-cell normalization (mandatory)
data_norm = adata.X.T.astype(np.double)
data_norm /= data_norm.sum(0)
# Add a small value to avoid numerical errors
data_norm += 1e-9
data_norm /= data_norm.sum(0)
```

[16]:
```
# Compute OT distance matrix
C = otscomics.cost_matrix(adata.X.T.astype(np.double), 'cosine')
D_ot, errors = otscomics.OT_distance_matrix(
    data=data_norm, cost=C, eps=.1,
    dtype=torch.double, device='cuda'
)
```

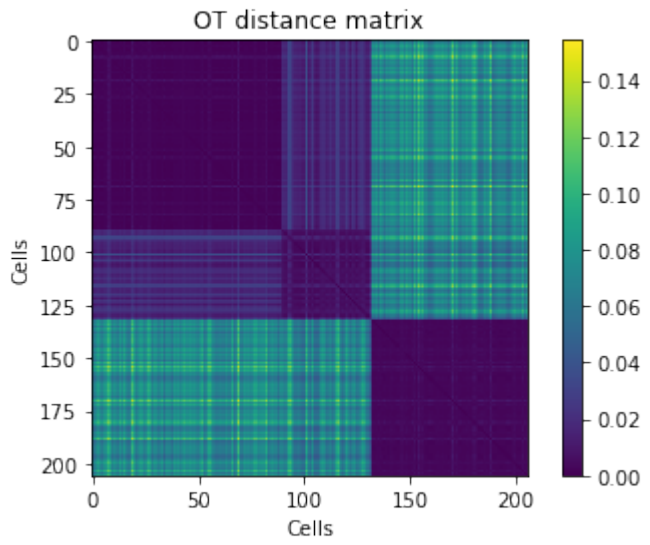
Displaying the OT distance matrix, with cells ordered by class. Clusters should be visible in the form of diagonal blocks of low distances.

[17]:
```
plt.imshow(D_ot[idx][:,idx])
plt.title('OT distance matrix')
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Cells')
plt.ylabel('Cells')
plt.colorbar()
plt.show()
```



1.4 Scoring

Let us compare the score for both distance matrices.

1.4.1 Silhouette score

[18]:
```
print('Euclidean\t',  silhouette_score(D_eu, clusters, metric='precomputed'))
print('OT\t\t',       silhouette_score(D_ot, clusters, metric='precomputed'))
```

```
Euclidean       0.3101426773824726
OT              0.8380411531442389
```

1.4.2 C index

[19]:
```
print('Euclidean\t',  otscomics.C_index(D_eu, clusters))
print('OT\t\t',       otscomics.C_index(D_ot, clusters))
```

```
Euclidean       0.034578119046118724
OT              0.0023526581402160966
```

1.4.3 Hierarchical clustering (fixed number of clusters)

We first cluster cells based on hierarchical clustering, fixed the number of clusters to the number of classes in the ground truth.

[20]:
```
cl = AgglomerativeClustering(affinity='precomputed', n_clusters=len(np.unique(clusters)),
↪ linkage='complete')
```

[21]:
```
print('Euclidean')
cl.set_params().fit(D_eu)
print('ARI', adjusted_rand_score(clusters, cl.labels_))
print('NMI', normalized_mutual_info_score(clusters, cl.labels_))
print('AMI', adjusted_mutual_info_score(clusters, cl.labels_))
```

```
Euclidean
ARI 0.9215689106896331
NMI 0.9084054726687503
AMI 0.9075477330695365
```

[22]:
```
print('OT')
cl.set_params().fit(D_ot)
print('ARI', adjusted_rand_score(clusters, cl.labels_))
print('NMI', normalized_mutual_info_score(clusters, cl.labels_))
print('AMI', adjusted_mutual_info_score(clusters, cl.labels_))
```

```
OT
ARI 1.0
NMI 1.0
AMI 1.0
```

1.4.4 Hiarchical clustering (number of clusters derived from silhouette)

A more realistic setting is to find the "optimal" number of clusters in an unsupervised way. We run the clustering for every number of clusters between 3 and 25, and select the clustering yielding the best silhouette score.

[23]:
```
cl = AgglomerativeClustering(affinity='precomputed', linkage='complete')

range_clusters, sil = range(3, 26), []

# Iterate through numbers of clusters
for n_clusters in range_clusters:
  cl.set_params(n_clusters=n_clusters).fit(D_eu)
  sil.append(silhouette_score(D_eu, cl.labels_, metric='precomputed'))

# Select the number yielding best silhouette
cl.set_params(n_clusters=range_clusters[np.argmax(sil)]).fit(D_eu)

print('Euclidean,', range_clusters[np.argmax(sil)], 'clusters')
print('ARI', adjusted_rand_score(clusters, cl.labels_))
print('NMI', normalized_mutual_info_score(clusters, cl.labels_))
print('AMI', adjusted_mutual_info_score(clusters, cl.labels_))
```

```
Euclidean, 3 clusters
ARI 0.9215689106896331
NMI 0.9084054726687503
AMI 0.9075477330695365
```

[24]:
```
cl = AgglomerativeClustering(affinity='precomputed', linkage='complete')

range_clusters, sil = range(3, 26), []

# Iterate through numbers of clusters
for n_clusters in range_clusters:
  cl.set_params(n_clusters=n_clusters).fit(D_ot)
  sil.append(silhouette_score(D_ot, cl.labels_, metric='precomputed'))

# Select the number yielding best silhouette
cl.set_params(n_clusters=range_clusters[np.argmax(sil)]).fit(D_ot)

print('OT,', range_clusters[np.argmax(sil)], 'clusters')
print('ARI', adjusted_rand_score(clusters, cl.labels_))
print('NMI', normalized_mutual_info_score(clusters, cl.labels_))
print('AMI', adjusted_mutual_info_score(clusters, cl.labels_))
```

```
OT, 3 clusters
ARI 1.0
NMI 1.0
AMI 1.0
```

1.4.5 A typical single-cell clustering based on Leiden

In practice, single-cell clustering is often done by computing a kNN network according to euclidean distance on PCA components, and then performing Leiden clustering on that network. Here, we compare this typical workflow with Leiden clustering on the kNN network computed directly from the OT distance matrix.

[25]:
```
print('PCA + euclidean + kNN + Leiden')

sc.pp.pca(adata)
sc.pp.neighbors(adata)

resolutions = np.linspace(.25, 1.5, 20)
sils, aris, nmis, amis, n_clusters = [], [], [], [], []

# Iterate through resolutions
print('Iterating through resolutions...')
for resolution in tqdm(resolutions):

  sc.tl.leiden(adata, resolution=resolution)

  if len(np.unique(adata.obs['leiden'])) > 1:
    sils.append(silhouette_score(D_ot, adata.obs['leiden'], metric='precomputed'))
    aris.append(adjusted_rand_score(clusters, adata.obs['leiden']))
    nmis.append(normalized_mutual_info_score(clusters, adata.obs['leiden']))
    amis.append(adjusted_mutual_info_score(clusters, adata.obs['leiden']))
```

(continues on next page)

(continued from previous page)

```
    n_clusters.append(len(np.unique(adata.obs['leiden'])))
  else:
    sils.append(-1)
    aris.append(-1)
    amis.append(-1)
    nmis.append(-1)
    n_clusters.append(-1)

# Max silhouette score
print('Resolution selected by silhouette score')
i = np.argmax(sils)

print('Resolution', resolutions[i])
print('ARI', aris[i])
print('NMI', nmis[i])
print('AMI', amis[i])
print('nb clusters', n_clusters[i])
```

```
PCA + euclidean + kNN + Leiden
Iterating through resolutions...
```

```
100%|| 20/20 [00:00<00:00, 46.45it/s]
```

```
Resolution selected by silhouette score
Resolution 0.25
ARI 0.921697966842453
NMI 0.8963498310700881
AMI 0.8953788659034694
nb clusters 3
```

[26]:
```
from scanpy.neighbors import _compute_connectivities_umap as conn_umap
```

[27]:
```
print('OT + kNN + Leiden')

# Not used, just creates objects (some useless computations here...)
sc.pp.pca(adata)
sc.pp.neighbors(adata)

knn_indices, knn_dists, forest = sc.neighbors.compute_neighbors_umap(
  D_ot, n_neighbors=15, metric='precomputed')

adata.obsp['distances'], adata.obsp['connectivities'] = conn_umap(
    knn_indices,
    knn_dists,
    adata.shape[0],
    15,
)

resolutions = np.linspace(.25, 1.5, 20)
sils, aris, nmis, amis, n_clusters = [], [], [], [], []
```

(continues on next page)

(continued from previous page)

```
# Iterate through resolutions
print('Iterating through resolutions...')
for resolution in tqdm(resolutions):
  sc.tl.leiden(adata, resolution=resolution)
  if len(np.unique(adata.obs['leiden'])) > 1:
    sils.append(silhouette_score(D_ot, adata.obs['leiden'], metric='precomputed'))
    aris.append(adjusted_rand_score(clusters, adata.obs['leiden']))
    nmis.append(normalized_mutual_info_score(clusters, adata.obs['leiden']))
    amis.append(adjusted_mutual_info_score(clusters, adata.obs['leiden']))
    n_clusters.append(len(np.unique(adata.obs['leiden'])))
  else:
    sils.append(-1)
    aris.append(-1)
    amis.append(-1)
    nmis.append(-1)
    n_clusters.append(-1)

# Max silhouette score
print('Resolution selected by silhouette score')
i = np.argmax(sils)

print('Resolution', resolutions[i])
print('ARI', aris[i])
print('NMI', nmis[i])
print('AMI', amis[i])
print('nb clusters', n_clusters[i])
```

```
OT + kNN + Leiden
Iterating through resolutions...
```

```
100%|| 20/20 [00:00<00:00, 46.76it/s]
```

```
Resolution selected by silhouette score
Resolution 0.25
ARI 1.0
NMI 1.0
AMI 1.0
nb clusters 3
```

CHAPTER

TWO

OTSCOMICS

otscomics.**C_index**(*D: ndarray*, *clusters: ndarray*) → float

Compute the C index, a measure of how well the pairwise distances reflect ground truth clusters. Implemented here for reference, but the silhouette score (aka Average Silhouette Width) is a more standard metric for this.

Parameters

- **D** (*np.ndarray*) – The pairwise distances.
- **clusters** (*np.ndarray*) – The ground truth clusters.

Returns
The C index.

Return type
float

otscomics.**OT_distance_matrix**(*data: ndarray*, *cost: ndarray*, *eps: float = 0.1*, *dtype: torch.dtype = torch.double*, *device: str = 'cuda'*, *divide_max: bool = False*, *numItermax: int = 500*, *stopThr: float = 1e-05*, *batch_size: int = 200*) → ndarray

Compute the pairwise Optimal Transport distance matrix. We compute Sinkhorn Divergences using POT's implementation of the Sinkhorn algorithm. Computations are done using PyTorch on a specified device. But the result is a numpy array. This allows not saturating the GPU for large matrices.

Parameters

- **data** (*np.ndarray*) – The input data, as a numpy array.
- **cost** (*np.ndarray*) – The ground cost between features.
- **eps** (*float, optional*) – The entropic regularization parameter. Small regularization requires more iterations and double precision. Defaults to .1.
- **dtype** (*torch.dtype, optional*) – The torch dtype used for computations. Double is more precise but takes up more space. Defaults to torch.double.
- **device** (*str, optional*) – The torch device to compute on, typically 'cpu' or 'cuda'. Defaults to 'cuda'.
- **divide_max** (*bool, optional*) – Whether to divide the resulting matrix by its maximum value. This can be useful to compare matrices. Defaults to False.
- **numItermax** (*int, optional*) – Used by POT, maximum number of Sinkhorn iterations. Defaults to 500.
- **stopThr** (*float, optional*) – Used by POT, tolerance for early stopping in the Sinkhorn iterations. Defaults to 1e-5.

- **batch_size** (*int, optional*) – The batch size, i.e. how many distances can be computed at the same time. Should be as large as possible on your hardware. Defaults to 200.

Returns
The pairwise OT distance matrix.

Return type
np.ndarray

otscomics.**cost_matrix**(*data: ndarray*, *cost: str = 'correlation'*, *normalize_features: bool = True*) → ndarray

Compute an empirical ground cost matrix, i.e. a pairwise distance matrix between the rows of the dataset (l1-normalized by default). Accepted distances are the ones compatible with Scipy's *cdist*.

Parameters

- **data** (*np.ndarray*) – The input data, samples as columns and features as rows.
- **cost** (*str*) – The metric use. Defaults to *correlation*.
- **normalize_features** (*bool, optional*) – Whether to divide the rows by their sum before computing distances. Defaults to True.

Returns
The pairwise cost matrix.

Return type
np.ndarray

CHAPTER

THREE

INSTALLATION

```
pip install otscomics
```

PYTHON MODULE INDEX

O

INDEX