wsingular

Geert-Jan Huizing, Laura Cantini, Gabriel Peyré

Mar 01, 2023

GETTING STARTED

CHAPTER

ONE

SINKHORN SINGULAR VECTORS

This Jupyter Notebook will walk you through an easy example of Sinkhorn Singular Vectors (SSV), which are the entropic regularization of Wasserstein Singular Vectors (WSV). This example is small enough to be run on CPU.

1.1 Imports

[1]:
```
import wsingular
import torch
import matplotlib.pyplot as plt
```

```
<frozen importlib._bootstrap>:219: RuntimeWarning: scipy._lib.messagestream.
↪MessageStream size changed, may indicate binary incompatibility. Expected 56 from C␣
↪header, got 64 from PyObject
```

1.2 Generate toy data

[2]:
```
# Define the dtype and device to work with.
dtype = torch.double
device = "cpu"
```

[3]:
```
# Define the dimensions of our problem.
n_samples = 20
n_features = 30
```

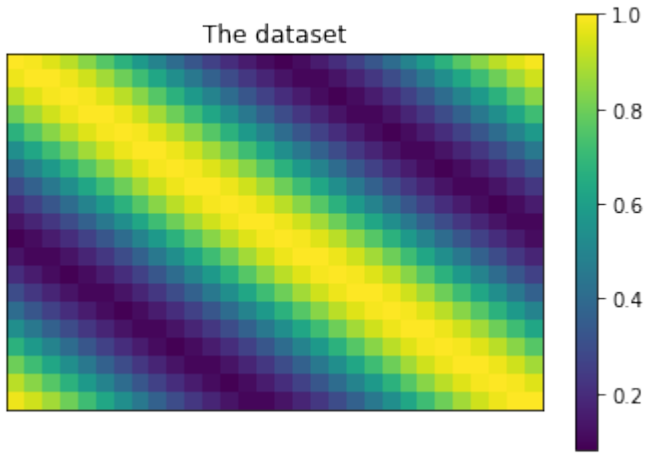
[4]:
```
# Initialize an empty dataset.
dataset = torch.zeros((n_samples, n_features), dtype=dtype)

# Iterate over the features and samples.
for i in range(n_samples):
    for j in range(n_features):

        # Fill the dataset with translated histograms.
        dataset[i, j] = i/n_samples - j/n_features
        dataset[i, j] = torch.abs(dataset[i, j] % 1)

# Take the distance to 0 on the torus.
dataset = torch.min(dataset, 1 - dataset)
```

(continues on next page)

(continued from previous page)

```
# Make it a guassian.
dataset = torch.exp(-(dataset**2) / 0.1)
```

[5]:
```
# Plot the dataset.
plt.title('The dataset')
plt.imshow(dataset)
plt.colorbar()
plt.xticks([])
plt.yticks([])
plt.show()
```



1.3 Compute the SSV

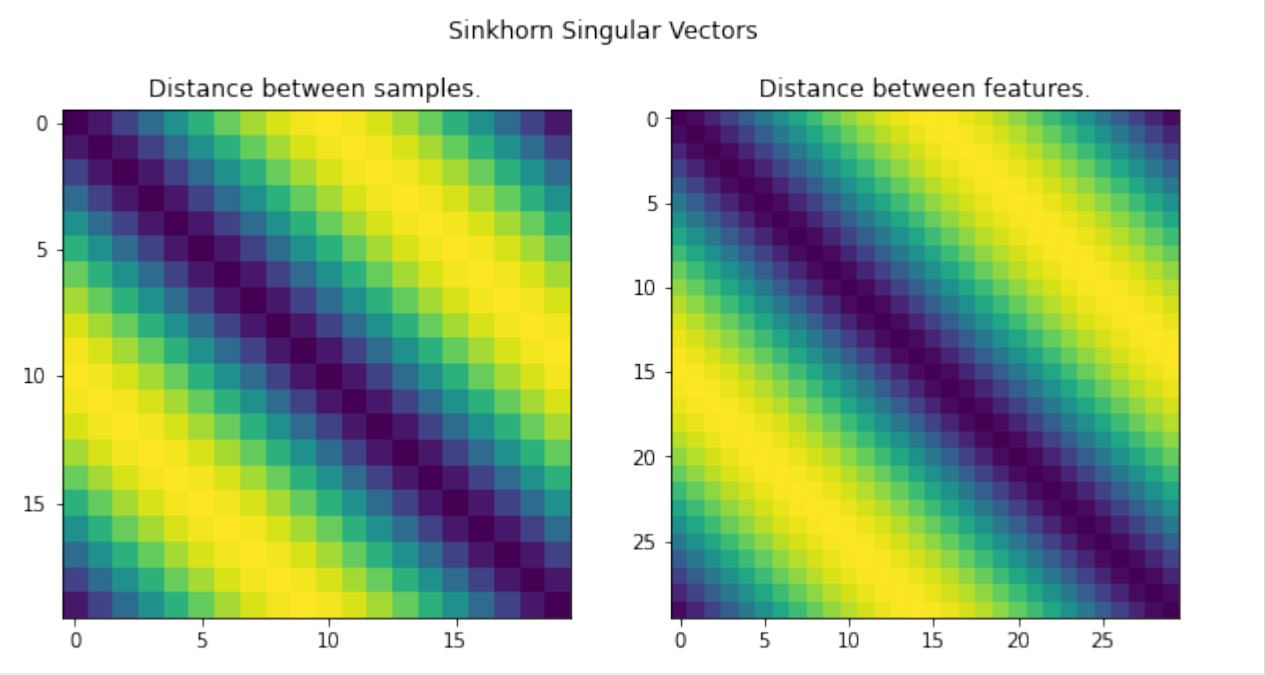
[6]:
```
# Compute the SSV.
C, D = wsingular.sinkhorn_singular_vectors(
    dataset,
    eps=5e-2,
    dtype=dtype,
    device=device,
    n_iter=100,
    progress_bar=True,
)
```

[7]:
```
# Display the SSV.
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
fig.suptitle('Sinkhorn Singular Vectors')

axes[0].set_title('Distance between samples.')
axes[0].imshow(D)
axes[0].set_xticks(range(0, n_samples, 5))
axes[0].set_yticks(range(0, n_samples, 5))
```

(continues on next page)

(continued from previous page)

```
axes[1].set_title('Distance between features.')
axes[1].imshow(C)
axes[1].set_xticks(range(0, n_features, 5))
axes[1].set_yticks(range(0, n_features, 5))

plt.show()
```



CHAPTER

TWO

STOCHASTIC SINKHORN SINGULAR VECTORS

This Jupyter Notebook will walk you through an easy example of stochastic computation of Sinkhorn Singular Vectors. This example is small enough to be run on CPU.

2.1 Imports

[1]:
```
import wsingular
import torch
import matplotlib.pyplot as plt
```

```
<frozen importlib._bootstrap>:219: RuntimeWarning: scipy._lib.messagestream.
↪MessageStream size changed, may indicate binary incompatibility. Expected 56 from C␣
↪header, got 64 from PyObject
```

2.2 Generate toy data

[2]:
```
# Define the dtype and device to work with.
dtype = torch.double
device = "cpu"
```

[3]:
```
# Define the dimensions of our problem.
n_samples = 20
n_features = 30
```

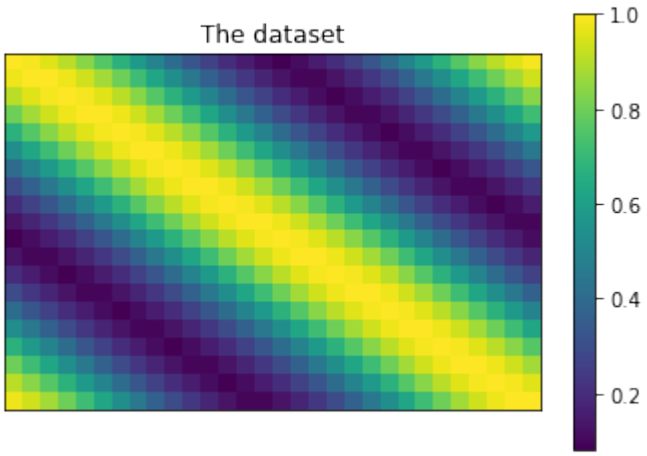
[4]:
```
# Initialize an empty dataset.
dataset = torch.zeros((n_samples, n_features), dtype=dtype)

# Iterate over the features and samples.
for i in range(n_samples):
    for j in range(n_features):

        # Fill the dataset with translated histograms.
        dataset[i, j] = i/n_samples - j/n_features
        dataset[i, j] = torch.abs(dataset[i, j] % 1)

# Take the distance to 0 on the torus.
dataset = torch.min(dataset, 1 - dataset)
```

(continues on next page)

(continued from previous page)

```
# Make it a guassian.
dataset = torch.exp(-(dataset**2) / 0.1)
```

[5]:
```
# Plot the dataset.
plt.title('The dataset')
plt.imshow(dataset)
plt.colorbar()
plt.xticks([])
plt.yticks([])
plt.show()
```



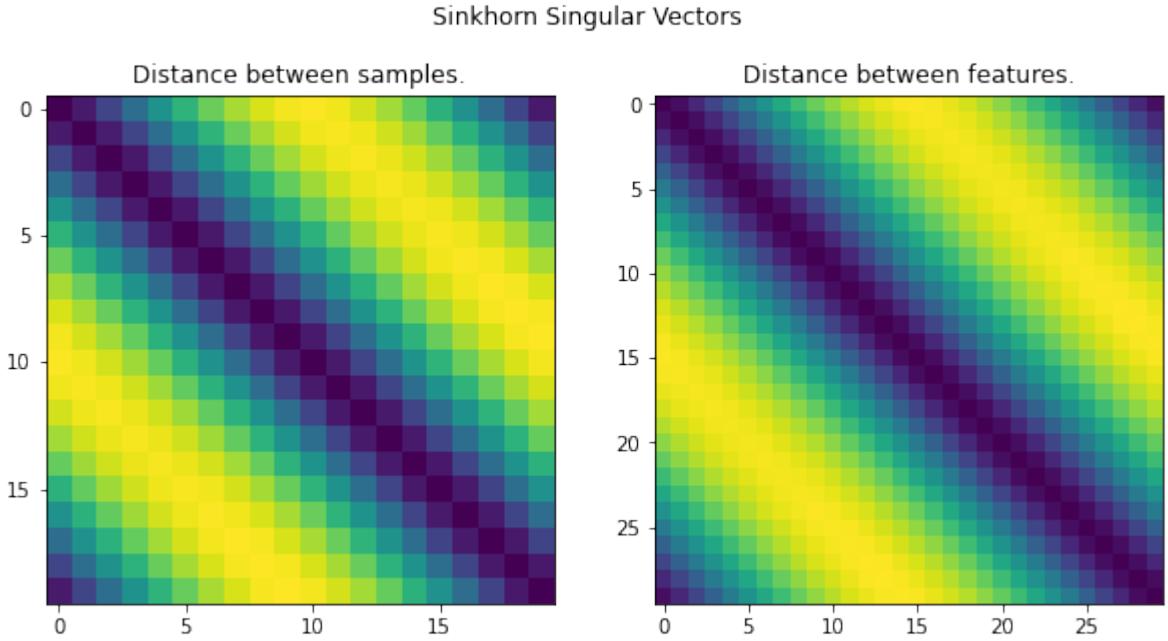
2.3 Compute the SSV

[6]:
```
# Compute the SSV.
C, D = wsingular.stochastic_sinkhorn_singular_vectors(
    dataset,
    dtype=dtype,
    device=device,
    eps=5e-2,
    sample_prop=1e-1,
    p=1,
    n_iter=1_000,
    progress_bar=True,
)
```

```
  0%|          | 0/36 [00:00<?, ?it/s]/users/csb/huizing/anaconda3/lib/python3.8/site-
↪packages/ot/bregman.py:517: UserWarning: Sinkhorn did not converge. You might want to␣
↪increase the number of iterations `numItermax` or the regularization parameter `reg`.
  warnings.warn("Sinkhorn did not converge. You might want to "
```

```
Stopping early after keyboard interrupt!
```

[7]:
```
# Display the SSV.
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
fig.suptitle('Sinkhorn Singular Vectors')

axes[0].set_title('Distance between samples.')
axes[0].imshow(D)
axes[0].set_xticks(range(0, n_samples, 5))
axes[0].set_yticks(range(0, n_samples, 5))

axes[1].set_title('Distance between features.')
axes[1].imshow(C)
axes[1].set_xticks(range(0, n_features, 5))
axes[1].set_yticks(range(0, n_features, 5))

plt.show()
```



CHAPTER

THREE

STOCHASTIC WASSERSTEIN SINGULAR VECTORS

This Jupyter Notebook will walk you through an easy example of stochastic computation of Wasserstein Singular Vectors. This example is small enough to be run on CPU.

3.1 Imports

[1]:
```
import wsingular
import torch
import matplotlib.pyplot as plt
```

```
<frozen importlib._bootstrap>:219: RuntimeWarning: scipy._lib.messagestream.
↪MessageStream size changed, may indicate binary incompatibility. Expected 56 from C␣
↪header, got 64 from PyObject
```

3.2 Generate toy data

[2]:
```
# Define the dtype and device to work with.
dtype = torch.double
device = "cpu"
```

[3]:
```
# Define the dimensions of our problem.
n_samples = 20
n_features = 30
```

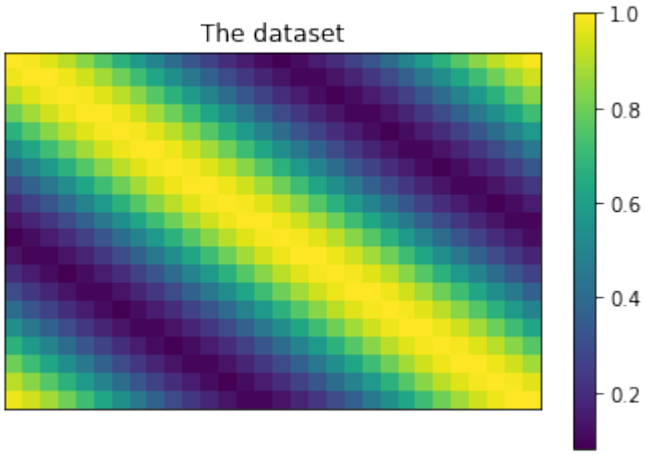
[4]:
```
# Initialize an empty dataset.
dataset = torch.zeros((n_samples, n_features), dtype=dtype)

# Iterate over the features and samples.
for i in range(n_samples):
    for j in range(n_features):

        # Fill the dataset with translated histograms.
        dataset[i, j] = i/n_samples - j/n_features
        dataset[i, j] = torch.abs(dataset[i, j] % 1)

# Take the distance to 0 on the torus.
dataset = torch.min(dataset, 1 - dataset)
```

(continues on next page)

(continued from previous page)

```
# Make it a guassian.
dataset = torch.exp(-(dataset**2) / 0.1)
```

[5]:
```
# Plot the dataset.
plt.title('The dataset')
plt.imshow(dataset)
plt.colorbar()
plt.xticks([])
plt.yticks([])
plt.show()
```



3.3 Compute the WSV

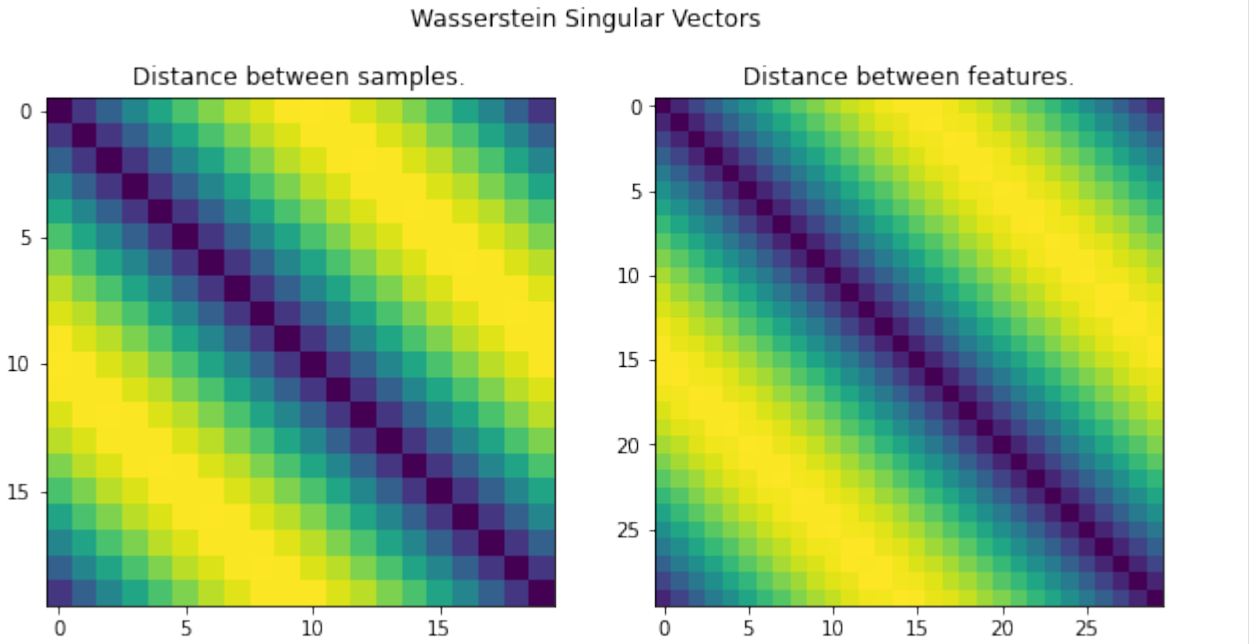
[6]:
```
# Compute the WSV.
C, D = wsingular.stochastic_wasserstein_singular_vectors(
    dataset,
    dtype=dtype,
    device=device,
    n_iter=1_000,
    sample_prop=1e-1,
)
```

[7]:
```
# Display the WSV.
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
fig.suptitle('Wasserstein Singular Vectors')

axes[0].set_title('Distance between samples.')
axes[0].imshow(D)
axes[0].set_xticks(range(0, n_samples, 5))
axes[0].set_yticks(range(0, n_samples, 5))

axes[1].set_title('Distance between features.')
```

(continues on next page)

(continued from previous page)

```
axes[1].imshow(C)
axes[1].set_xticks(range(0, n_features, 5))
axes[1].set_yticks(range(0, n_features, 5))

plt.show()
```



[8]:
```
A, B = wsingular.utils.normalize_dataset(dataset, dtype=dtype, device=device)
wsingular.utils.check_uniqueness(A, B, C, D)
```

[8]:
```
True
```

CHAPTER

FOUR

WASSERSTEIN SINGULAR VECTORS

This Jupyter Notebook will walk you through an easy example of Wasserstein Singular Vectors (WSV). This example is small enough to be run on CPU.

4.1 Imports

[1]:
```
import wsingular
import torch
import matplotlib.pyplot as plt
```

```
<frozen importlib._bootstrap>:219: RuntimeWarning: scipy._lib.messagestream.
↪MessageStream size changed, may indicate binary incompatibility. Expected 56 from C␣
↪header, got 64 from PyObject
```

4.2 Generate toy data

[2]:
```
# Define the dtype and device to work with.
dtype = torch.double
device = "cpu"
```

[3]:
```
# Define the dimensions of our problem.
n_samples = 20
n_features = 30
```

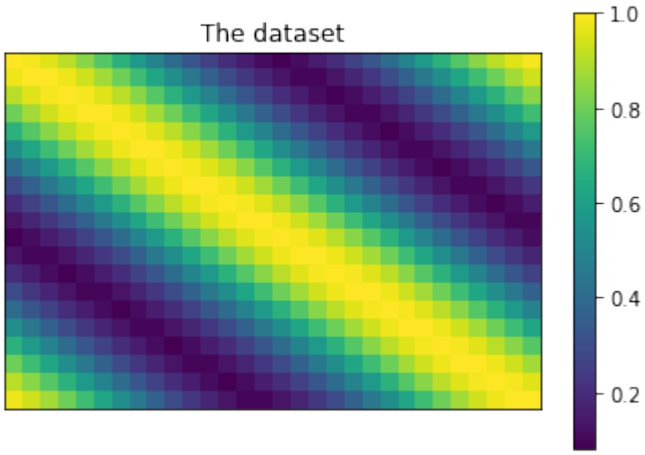
[4]:
```
# Initialize an empty dataset.
dataset = torch.zeros((n_samples, n_features), dtype=dtype)

# Iterate over the features and samples.
for i in range(n_samples):
    for j in range(n_features):

        # Fill the dataset with translated histograms.
        dataset[i, j] = i/n_samples - j/n_features
        dataset[i, j] = torch.abs(dataset[i, j] % 1)

# Take the distance to 0 on the torus.
dataset = torch.min(dataset, 1 - dataset)
```

(continues on next page)

(continued from previous page)

```
# Make it a guassian.
dataset = torch.exp(-(dataset**2) / 0.1)
```

[5]:
```
# Plot the dataset.
plt.title('The dataset')
plt.imshow(dataset)
plt.colorbar()
plt.xticks([])
plt.yticks([])
plt.show()
```



4.3 Compute the WSV

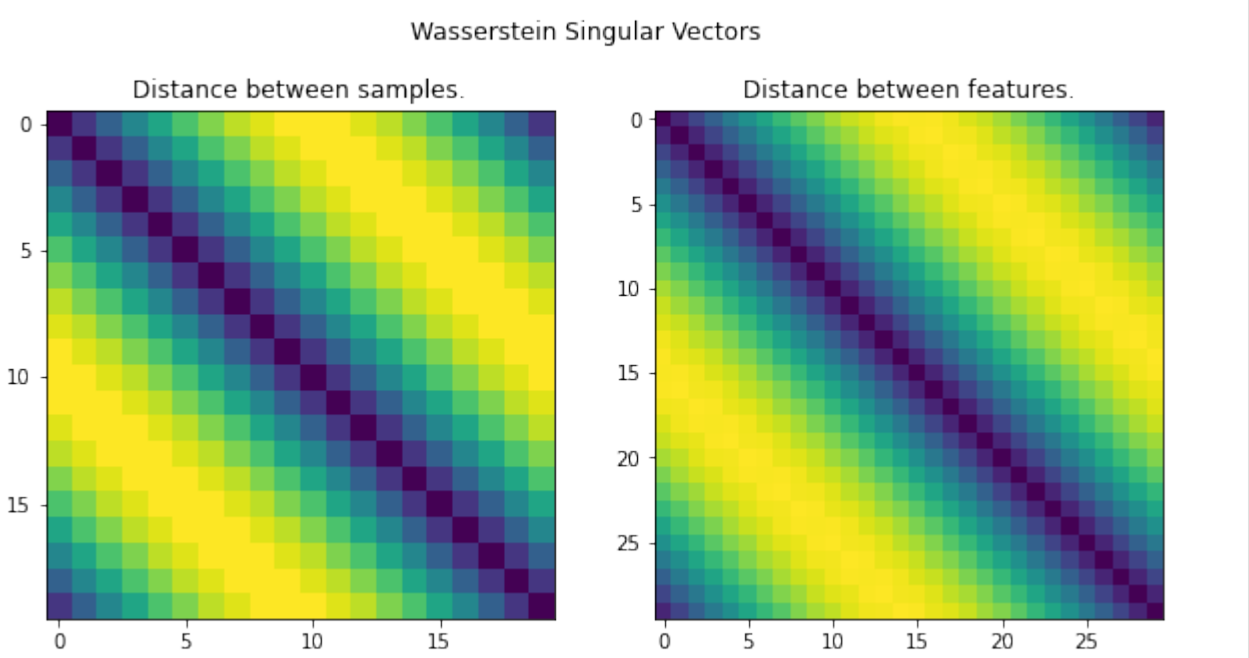
[6]:
```
# Compute the WSV.
C, D = wsingular.wasserstein_singular_vectors(
    dataset,
    n_iter=100,
    dtype=dtype,
    device=device,
)
```

[7]:
```
# Display the WSV.
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
fig.suptitle('Wasserstein Singular Vectors')

axes[0].set_title('Distance between samples.')
axes[0].imshow(D)
axes[0].set_xticks(range(0, n_samples, 5))
axes[0].set_yticks(range(0, n_samples, 5))

axes[1].set_title('Distance between features.')
axes[1].imshow(C)
```

(continues on next page)

(continued from previous page)

```
axes[1].set_xticks(range(0, n_features, 5))
axes[1].set_yticks(range(0, n_features, 5))

plt.show()
```



[8]:
```
A, B = wsingular.utils.normalize_dataset(dataset, dtype=dtype, device=device)
wsingular.utils.check_uniqueness(A, B, C, D)
```

[8]:
```
True
```

CHAPTER

FIVE

WSINGULAR

wsingular.**sinkhorn_singular_vectors**(*dataset: torch.Tensor*, *dtype: str*, *device: str*, *n_iter: int*, *tau: float = 0*, *eps: float = 0.05*, *p: int = 1*, *writer=None*, *small_value: float = 1e-06*, *normalization_steps: int = 1*, *C_ref: Optional[torch.Tensor] = None*, *D_ref: Optional[torch.Tensor] = None*, *log_loss: bool = False*, *progress_bar: bool = False*) → Tuple[torch.Tensor, torch.Tensor]

Performs power iterations and returns Sinkhorn Singular Vectors. Early stopping is possible with Ctrl-C.

Parameters

- **dataset** (*torch.Tensor*) – The input dataset. Alternatively, you can give a tuple of tensors (A, B).
- **dtype** (*str*) – The dtype
- **device** (*str*) – The device
- **n_iter** (*int*) – The number of power iterations.
- **tau** (*float, optional*) – The regularization parameter for the norm R. Defaults to 0.
- **eps** (*float*) – The entropic regularization parameter.
- **p** (*int, optional*) – The order of the norm R. Defaults to 1.
- **writer** (*SummaryWriter, optional*) – If set, the progress will be written to the Tensorboard writer. Defaults to None.
- **small_value** (*float, optional*) – A small value for numerical stability. Defaults to 1e-6.
- **normalization_steps** (*int, optional*) – How many Sinkhorn iterations for the initial normalization of the dataset. Must be > 0. Defaults to 1, which is just regular normalization, along columns for A and along rows for B. For large numbers of steps, A and B are bistochastic.
- **C_ref** (*torch.Tensor, optional*) – If set, Hilbert distances to this reference will be computed. Defaults to None.
- **D_ref** (*torch.Tensor, optional*) – If set, Hilbert distances to this reference will be computed. Defaults to None.
- **log_loss** (*bool, optional*) – Whether to return the loss. Defaults to False.
- **progress_bar** (*bool, optional*) – Whether to display a progress bar for individual matrix computations. Defaults to False.

Returns

Sinkhorn Singular Vectors (C,D). If *log_loss*, it returns (C, D, loss_C, loss_D)

Return type
Tuple[torch.Tensor, torch.Tensor]

wsingular.**stochastic_sinkhorn_singular_vectors**(*dataset: torch.Tensor*, *dtype: torch.dtype*, *device: str*, *n_iter: int*, *tau: float = 0*, *eps: float = 0.05*, *sample_prop: float = 0.1*, *p: int = 1*, *step_fn: ~typing.Callable = <function <lambda>>*, *mult_update: bool = False*, *writer=None*, *small_value: float = 1e-06*, *normalization_steps: int = 1*, *C_ref: ~typing.Optional[torch.Tensor] = None*, *D_ref: ~typing.Optional[torch.Tensor] = None*, *progress_bar: bool = False*) → Tuple[torch.Tensor, torch.Tensor]

Performs stochastic power iterations and returns Sinkhorn Singular Vectors. Early stopping is possible with Ctrl-C.

Parameters

- **dataset** (*torch.Tensor*) – The input dataset. Alternatively, you can give a tuple of tensors (A, B).
- **dtype** (*torch.dtype*) – The dtype
- **device** (*str*) – The device
- **n_iter** (*int*) – The number of power iterations.
- **tau** (*float, optional*) – The regularization parameter for the norm R. Defaults to 0.
- **eps** (*float, optional*) – The entropic regularization parameter. Defaults to 5e-2.
- **sample_prop** (*float, optional*) – The proportion of indices to update at each step. Defaults to 1e-1.
- **p** (*int, optional*) – The order of the norm R. Defaults to 1.
- **step_fn** (*Callable, optional*) – The function that defines step size from the iteration number (which starts at 1). Defaults to lambdak:1/np.sqrt(k).
- **mult_update** (*bool, optional*) – If True, use multiplicative update instead of additive update. Defaults to False.
- **writer** (*SummaryWriter, optional*) – If set, the progress will be written to the Tensorboard writer. Defaults to None.
- **small_value** (*float, optional*) – A small value for numerical stability. Defaults to 1e-6.
- **normalization_steps** (*int, optional*) – How many Sinkhorn iterations for the initial normalization of the dataset. Must be > 0. Defaults to 1, which is just regular normalization, along columns for A and along rows for B. For large numbers of steps, A and B are bistochastic.
- **C_ref** (*torch.tensor, optional*) – If set, Hilbert distances to this reference will be computed. Defaults to None.
- **D_ref** (*torch.Tensor, optional*) – If set, Hilbert distances to this reference will be computed. Defaults to None.
- **progress_bar** (*bool, optional*) – Whether to display a progress bar for individual matrix computations. Defaults to False.

Returns
Sinkhorn Singular Vectors (C,D)

Return type
Tuple[torch.Tensor, torch.Tensor]

wsingular.**stochastic_wasserstein_singular_vectors**(*dataset: torch.Tensor*, *dtype: torch.dtype*, *device: str*, *n_iter: int*, *tau: float = 0*, *sample_prop: float = 0.1*, *p: int = 1*, *step_fn: ~typing.Callable = <function <lambda>>*, *mult_update: bool = False*, *writer=None*, *small_value: float = 1e-06*, *normalization_steps: int = 1*, *C_ref: ~typing.Optional[torch.Tensor] = None*, *D_ref: ~typing.Optional[torch.Tensor] = None*, *progress_bar: bool = False*) → Tuple[torch.Tensor, torch.Tensor]

Performs stochastic power iterations and returns Wasserstein Singular Vectors. Early stopping is possible with Ctrl-C.

Parameters

- **dataset** (*torch.Tensor*) – The input dataset. Alternatively, you can give a tuple of tensors (A, B).
- **dtype** (*torch.dtype*) – The dtype
- **device** (*str*) – The device
- **n_iter** (*int*) – The number of power iterations.
- **tau** (*float, optional*) – The regularization parameter for the norm R. Defaults to 0.
- **sample_prop** (*float, optional*) – The proportion of indices to update at each step. Defaults to 1e-1.
- **p** (*int, optional*) – The order of the norm R. Defaults to 1.
- **step_fn** (*Callable, optional*) – The function that defines step size from the iteration number (which starts at 1). Defaults to lambdak:1/np.sqrt(k).
- **mult_update** (*bool, optional*) – If True, use multiplicative update instead of additive update. Defaults to False.
- **writer** (*SummaryWriter, optional*) – If set, the progress will be written to the Tensorboard writer. Defaults to None.
- **small_value** (*float, optional*) – A small value for numerical stability. Defaults to 1e-6.
- **normalization_steps** (*int, optional*) – How many Sinkhorn iterations for the initial normalization of the dataset. Must be > 0. Defaults to 1, which is just regular normalization, along columns for A and along rows for B. For large numbers of steps, A and B are bistochastic.
- **C_ref** (*torch.tensor, optional*) – If set, Hilbert distances to this reference will be computed. Defaults to None.
- **D_ref** (*torch.Tensor, optional*) – If set, Hilbert distances to this reference will be computed. Defaults to None.
- **progress_bar** (*bool, optional*) – Whether to display a progress bar for individual matrix computations. Defaults to False.

Returns
Wasserstein Singular Vectors (C,D)

Return type
Tuple[torch.Tensor, torch.Tensor]

wsingular.**wasserstein_singular_vectors**(*dataset: torch.Tensor*, *dtype: torch.dtype*, *device: str*, *n_iter: int*, *tau: float = 0*, *p: int = 1*, *writer=None*, *small_value: float = 1e-06*, *normalization_steps: int = 1*, *C_ref: Optional[torch.Tensor] = None*, *D_ref: Optional[torch.Tensor] = None*, *log_loss: bool = False*, *progress_bar: bool = False*) → Tuple[torch.Tensor, torch.Tensor]

Performs power iterations and returns Wasserstein Singular Vectors. Early stopping is possible with Ctrl-C.

Parameters

- **dataset** (*torch.Tensor*) – The input dataset, rows as samples. Alternatively, you can give a tuple of tensors (A, B).
- **dtype** (*str*) – The dtype
- **device** (*str*) – The device
- **n_iter** (*int*) – The number of power iterations.
- **tau** (*float, optional*) – The regularization parameter for the norm R. Defaults to 0.
- **p** (*int, optional*) – The order of the norm R. Defaults to 1.
- **writer** (*SummaryWriter, optional*) – If set, the progress will be written to the Tensorboard writer. Defaults to None.
- **small_value** (*float, optional*) – A small value for numerical stability. Defaults to 1e-6.
- **normalization_steps** (*int, optional*) – How many Sinkhorn iterations for the initial normalization of the dataset. Must be > 0. Defaults to 1, which is just regular normalization, along columns for A and along rows for B. For large numbers of steps, A and B are bistochastic.
- **C_ref** (*torch.Tensor, optional*) – If set, Hilbert distances to this reference will be computed. Defaults to None.
- **D_ref** (*torch.Tensor, optional*) – If set, Hilbert distances to this reference will be computed. Defaults to None.
- **log_loss** (*bool, optional*) – Whether to return the loss. Defaults to False.
- **progress_bar** (*bool, optional*) – Whether to display a progress bar for individual matrix computations. Defaults to False.

Returns
Wasserstein sigular vectors (C,D). If *log_loss*, it returns (C, D, loss_C, loss_D)

Return type
Tuple[torch.Tensor, torch.Tensor]

CHAPTER

SIX

WSINGULAR.DISTANCE

wsingular.distance.**sinkhorn_map**(*A: torch.Tensor*, *C: torch.Tensor*, *eps: float*, *dtype: torch.dtype*, *device: str*, *R: Optional[torch.Tensor] = None*, *tau: float = 0*, *progress_bar: bool = False*, *stop_threshold: float = 1e-05*, *num_iter_max: int = 500*) → torch.Tensor

This function maps a ground cost to the pairwise Sinkhorn divergence matrix on a certain dataset using that ground cost. R is an added regularization.

Parameters

- **A** (*torch.Tensor*) – The input dataset, rows as samples.
- **C** (*torch.Tensor*) – The ground cost.
- **eps** (*float*) – The entropic regularization parameter.
- **dtype** (*torch.dtype*) – The dtype.
- **device** (*str*) – The device.
- **R** (*torch.Tensor*) – The added regularization. Defaults to None.
- **tau** (*float*) – The regularization parameter. Defaults to 0.
- **progress_bar** (*bool*) – Whether to show a progress bar during the computation. Defaults to False.
- **stop_threshold** (*float, optional*) – Stopping threshold for Sinkhorn (please refer to POT). Defaults to 1e-5.
- **num_iter_max** (*int, optional*) – Maximum number of Sinkhorn iterations (please refer to POT). Defaults to 500.

Returns
The pairwise Sinkhorn divergence matrix.

Return type
torch.Tensor

wsingular.distance.**stochastic_sinkhorn_map**(*A: torch.Tensor*, *D: torch.Tensor*, *C: torch.Tensor*, *sample_prop: float*, *gamma: float*, *eps: float*, *R: Optional[torch.Tensor] = None*, *tau: float = 0*, *progress_bar: bool = False*, *return_indices: bool = False*, *batch_size: int = 50*, *stop_threshold: float = 1e-05*, *num_iter_max: int = 100*) → torch.Tensor

Returns the stochastic Sinkhorn divergence map, updating only a random subset of indices and leaving the other ones as they are.

Parameters

- **A** (*torch.Tensor*) – The input dataset.
- **D** (*torch.Tensor*) – The intialization of the distance matrix
- **C** (*torch.Tensor*) – The ground cost
- **sample_prop** (*float*) – The proportion of indices to update
- **gamma** (*float*) – Rescaling parameter. In practice, one should rescale by an approximation of the singular value.
- **eps** (*float*) – The entropic regularization parameter
- **R** (*torch.Tensor*) – The regularization matrix. Defaults to None.
- **tau** (*float*) – The regularization parameter. Defaults to 0.
- **progress_bar** (*bool*) – Whether to show a progress bar during the computation. Defaults to False.
- **return_indices** (*bool*) – Whether to return the updated indices. Defaults to False.
- **batch_size** (*int*) – Batch size, i.e. how many distances to compute at the same time. Depends on your available GPU memory. Defaults to 50.

Returns
The stochastically updated distance matrix.

Return type
torch.Tensor

wsingular.distance.**stochastic_wasserstein_map**(*A: torch.Tensor*, *D: torch.Tensor*, *C: torch.Tensor*, *sample_prop: float*, *gamma: float*, *dtype: torch.dtype*, *device: str*, *R: Optional[torch.Tensor] = None*, *tau: float = 0*, *progress_bar: bool = False*, *return_indices: bool = False*) → torch.Tensor

Returns the stochastic Wasserstein map, updating only a random subset of indices and leaving the other ones as they are.

Parameters

- **A** (*torch.Tensor*) – The input dataset.
- **D** (*torch.Tensor*) – The intialization of the distance matrix
- **C** (*torch.Tensor*) – The ground cost
- **sample_prop** (*float*) – The proportion of indices to update
- **gamma** (*float*) – A scaling factor
- **dtype** (*torch.dtype*) – The dtype
- **device** (*str*) – The device
- **R** (*torch.Tensor*) – The regularization matrix. Defaults to None.
- **tau** (*float*) – The regularization parameter. Defaults to 0.
- **progress_bar** (*bool*) – Whether to show a progress bar during the computation. Defaults to False.
- **return_indices** (*bool*) – Whether to return the updated indices. Defaults to False.
- **stop_threshold** (*float, optional*) – Stopping threshold for Sinkhorn (please refer to POT). Defaults to 1e-5.

- **num_iter_max** (*int, optional*) – Maximum number of Sinkhorn iterations (please refer to POT). Defaults to 500.

Returns
The stochastically updated distance matrix.

Return type
torch.Tensor

wsingular.distance.**wasserstein_map**(*A: torch.Tensor*, *C: torch.Tensor*, *dtype: torch.dtype*, *device: str*, *R: Optional[torch.Tensor] = None*, *tau: float = 0*, *progress_bar: bool = False*) → torch.Tensor

This function maps a ground cost to the Wasserstein distance matrix on a certain dataset using that ground cost. R is an added regularization.

Parameters

- **A** (*torch.Tensor*) – The input dataset, rows as samples.
- **C** (*torch.Tensor*) – the ground cost.
- **dtype** (*torch.dtype*) – The dtype.
- **device** (*str*) – The device.
- **R** (*torch.Tensor*) – The regularization matrix. Defaults to None.
- **tau** (*float*) – The regularization parameter. Defaults to 0.
- **progress_bar** (*bool*) – Whether to show a progress bar during the computation. Defaults to False

Returns
The Wasserstein distance matrix with regularization.

Return type
torch.Tensor

CHAPTER

SEVEN

WSINGULAR.UTILS

wsingular.utils.**check_uniqueness**(*A: torch.Tensor*, *B: torch.Tensor*, *C: torch.Tensor*, *D: torch.Tensor*) → bool

Check uniqueness of singular vectors using the graph connectivity criterion described in the paper.

Parameters

- **A** (*torch.Tensor*) – The samples.
- **B** (*torch.Tensor*) – The features.
- **C** (*torch.Tensor*) – The ground cost.
- **D** (*torch.Tensor*) – The pairwise distance.

Returns
Whether the criterion is verified.

Return type
bool

wsingular.utils.**hilbert_distance**(*D_1: torch.Tensor*, *D_2: torch.Tensor*) → float

Compute the Hilbert distance between two distance-like matrices.

Parameters

- **D_1** (*torch.Tensor*) – The first matrix
- **D_2** (*torch.Tensor*) – The second matrix

Returns
The distance

Return type
float

wsingular.utils.**normalize_dataset**(*dataset: torch.Tensor*, *dtype: str*, *device: str*, *normalization_steps: int = 1*, *small_value: float = 1e-06*) → Tuple[torch.Tensor, torch.Tensor]

Normalize the dataset and return the normalized dataset A and the transposed dataset B.

Parameters

- **dataset** (*torch.Tensor*) – The input dataset, samples as rows.
- **normalization_steps** (*int, optional*) – The number of Sinkhorn normalization steps. For large numbers, we get bistochastic matrices. Defaults to 1 and should be larger or equal to 1.
- **small_value** (*float*) – Small addition to the dataset to avoid numerical errors while computing OT distances. Defaults to 1e-6.

Returns
The normalized matrices A and B.

Return type
Tuple[torch.Tensor, torch.Tensor]

wsingular.utils.**random_distance**(*size: int*, *dtype: str*, *device: str*) → torch.Tensor

Return a random distance-like matrix, i.e. symmetric with zero diagonal. The matrix is also divided by its maximum, so as to have infty norm 1.

Parameters

- **size** (*int*) – Will return a matrix of dimensions size*size
- **dtype** (*str*) – The dtype to be returned
- **device** (*str*) – The device to be returned

Returns
The random distance-like matrix

Return type
torch.Tensor

wsingular.utils.**regularization_matrix**(*A: torch.Tensor*, *p: int*, *dtype: str*, *device: str*) → torch.Tensor

Return the regularization matrix

Parameters

- **A** (*torch.Tensor*) – The dataset, with samples as rows
- **p** (*int*) – order of the norm
- **dtype** (*str*) – The dtype to be returned
- **device** (*str*) – The device to be returned

Returns
The regularization matrix

Return type
torch.Tensor

wsingular.utils.**silhouette**(*D: torch.Tensor*, *labels: Iterable*) → float

Return the average silhouette score, given a distance matrix and labels.

Parameters

- **D** (*torch.Tensor*) – Distance matrix n*n
- **labels** (*Iterable*) – n labels

Returns
The average silhouette score

Return type
float

wsingular.utils.**viz_TSNE**(*D: torch.Tensor*, *labels: Optional[Iterable] = None*) → None

Visualize a distance matrix using a precomputed distance matrix.

Parameters

- **D** (*torch.Tensor*) – Distance matrix
- **labels** (*Iterable, optional*) – The labels, if any. Defaults to None.

`wsingular` is the Python package for the ICML 2022 paper "Unsupervised Ground Metric Learning Using Wasserstein Singular Vectors".

Wasserstein Singular Vectors simultaneously compute a Wasserstein distance between *samples* and a Wasserstein distance between *features* of a dataset. These distance matrices emerge naturally as positive singular vectors of the function mapping ground costs to pairwise Wasserstein distances.

CHAPTER

EIGHT

GET STARTED

Install the package: `pip install wsingular`

Follow the tutorials in this documentation, and if you run into issue, leave an issue on the Github repo.

CHAPTER

NINE

TIPS

- We strongly encourage `torch.double` precision for numerical stability.
- You can easily run the demo notebook in Google Colab! Just use 'open from Github' and add `!pip install wsingular` at the beginning.
- If you want to stop the computation of singular vectors early, just hit `Ctrl-C` and the function will return the result of the latest optimization step.

CHAPTER

TEN

CITING US

The conference proceedings will be out soon. In the meantime you can cite our arXiv preprint.:

```
@article{huizing2021unsupervised,
  title={Unsupervised Ground Metric Learning using Wasserstein Eigenvectors},
  author={Huizing, Geert-Jan and Cantini, Laura and Peyr{\'e}, Gabriel},
  journal={arXiv preprint arXiv:2102.06278},
  year={2021}
}
```

PYTHON MODULE INDEX

W

INDEX