Vision2P

CNRS

Mar 04, 2025

This package is designed specifically for spectro-microscopy measurements, providing tools for clustering and unmixing spectral data. It includes a K-means clustering algorithm for segmentation and a modified Non-Negative Matrix Factorization (NMF) algorithm with enhanced features tailored for spectro-microscopy applications.

The K-means clustering algorithm enables automatic segmentation of spectro-microscopy measurements and helps to identify distinct regions or features within a dataset.

A regularization term has been added to the NMF algorithm to improve the results from spectro-microscopy with linearly polarized light. It is also possible to constrain some components, using simulated or experimental components, to obtain more accurate results. It is useful for spectral unmixing, to identify pure spectral components within a complex mixture.

It is intended to be compatible with data formats used in some of the most popular Python machine learning packages such as scikit-learn. It retains the same option names to simplify its usage.

`Vision2P.kmeans.`**`kmeans`**(*data*, *k*, *max_iters=100*, *tol=0.0001*, *random_state=None*)

K-means clustering algorithm

Inputs:

X

array-like, shape (n_samples, n_features) The input data matrix.

k

int The number of clusters.

max_iter

int, default=50 Maximum number of iterations to run the algorithm.

tol

float, default=1e-4 Tolerance to declare convergence based on centroid movement

Returns:

centroids

array-like, shape (k, n_features) K-means centroids.

labels

array-like, shape (n_samples,) The cluster index for each data point.

`Vision2P.shgnmf.`**`nmf`**(*X*, *W=None*, *H=None*, *n_components=4*, *init='random'*, *update_W=True*, *update_H=True*, *solver='mu'*, *tol=0.0001*, *max_iter=50*, *random_state=None*, *polarimetry_reg=0*, *constrained_components=None*)

Perform Non-negative Matrix Factorization (NMF) on the input matrix X.
The cost function is:

$$X = \|X - WH\|^2 + \alpha\|H - FH\|^2 \tag{1}$$

with α corresponding to the parameter *polarimetry_reg* and:

$$F = \begin{bmatrix} 0 & I_n \\ I_n & 0 \end{bmatrix} \tag{2}$$

with n being the number of features divided by two, the number of features should be even if the parameter *polarimetry_reg* is not 0.

Parameters:

X
array-like, shape (n_samples, n_features) The input data matrix to be factorized.

W
array-like, shape (n_samples, n_components), optional Initial guess for the matrix W. If None, it will be initialized based on *init*.

H
array-like, shape (n_components, n_features), optional Initial guess for matrix H. If None, it will be initialized based on *init*.

n_components
int, default=4 The number of components (latent features) to extract.

init
{'random', 'nndsvd', 'nndsvda', 'nndsvdar'}, default='random'
Initialization method for W and H. Options include:
- 'random': Random initialization.
- 'nndsvd': Nonnegative Double Singular Value Decomposition.
- 'nndsvda': NNDSVD with zeros filled with the average of X.
- 'nndsvdar': NNDSVD with zeros filled with small random values.

update_W
bool, default=True Whether to update matrix W during the optimization process.

update_H
bool, default=True Whether to update matrix H during the optimization process.

solver
{'mu', 'cd'}, default='mu' The solver to use for optimization:

- ‘mu’: Multiplicative update rule.
- ‘cd’: Coordinate descent.

tol
: float, default=1e-4 Tolerance for stopping condition.

max_iter
: int, default=50 Maximum number of iterations to run the algorithm.

random_state
: int, RandomState instance, or None, default=None Seed or random state for reproducibility.

polarimetry_reg
: float, default=0 Regularization parameter for polarimetry with linearly polarized light.

constrained_components
: array-like or None, default=None List of the components of the H matrix that will not be updated.

Output:

H
: array-like, shape (n_components, n_features) The learned coefficient matrix.

W
: array-like, shape (n_samples, n_components) The learned basis matrix.

reconstruction_err
: float The final reconstruction error.

Code examples

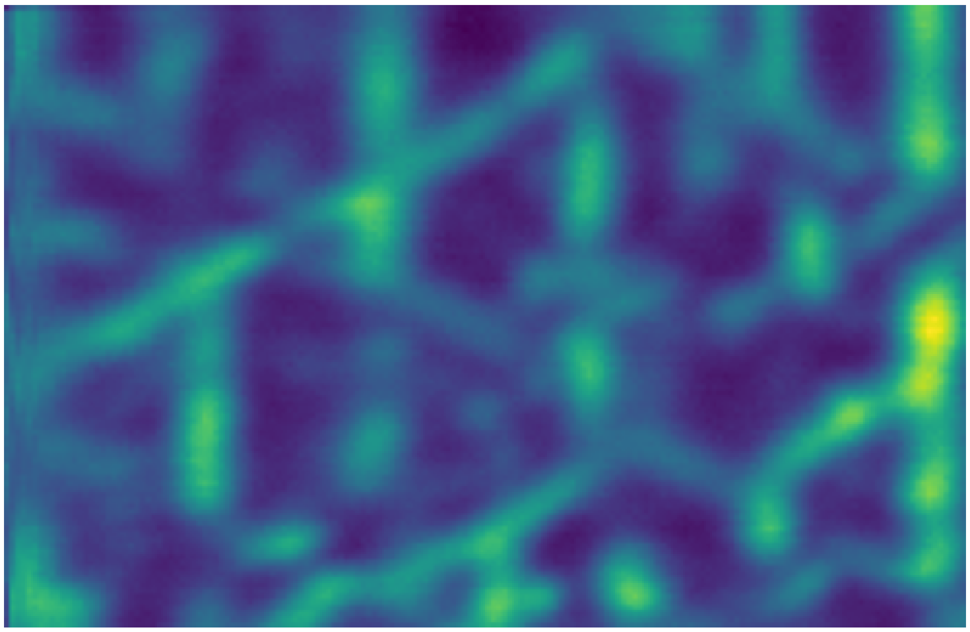
Imports

```
import os
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors
from Vision2P import nmf, kmeans
```

K-means and NMF on germanium telluride dataset

Open an process GeTe test data

```
n_components = 4
path = "14-56-42"

data_path = path + "/DAT/"
data_raw = np.stack([np.loadtxt(data_path + data_file) for data_file in os.
    listdir(data_path)])

plt.imshow(np.sum(data_raw, axis=0))
plt.axis('off')
plt.show()

data = np.reshape(data_raw, (data_raw.shape[0], -1)).T
norms = np.linalg.norm(data, axis=1, keepdims=True)
data = data / norms

parameters = np.loadtxt(path + "/parameters.dat")
angles = np.radians(parameters[0,4] + np.arange(0, parameters[1,4]) *
    parameters[2,4])
```

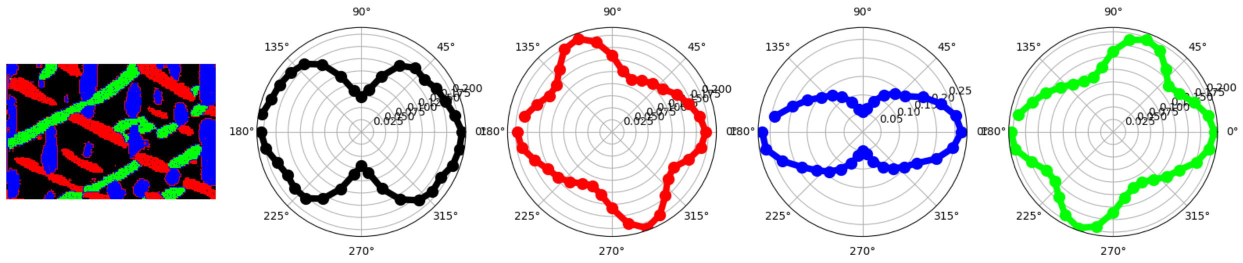


Apply K-means and NMF

```
centroids, labels = kmeans(data, n_components, random_state=4)
W, H = nmf(data, n_components=n_components, init="nndsvdar", solver="cd",
    max_iter=200)
```

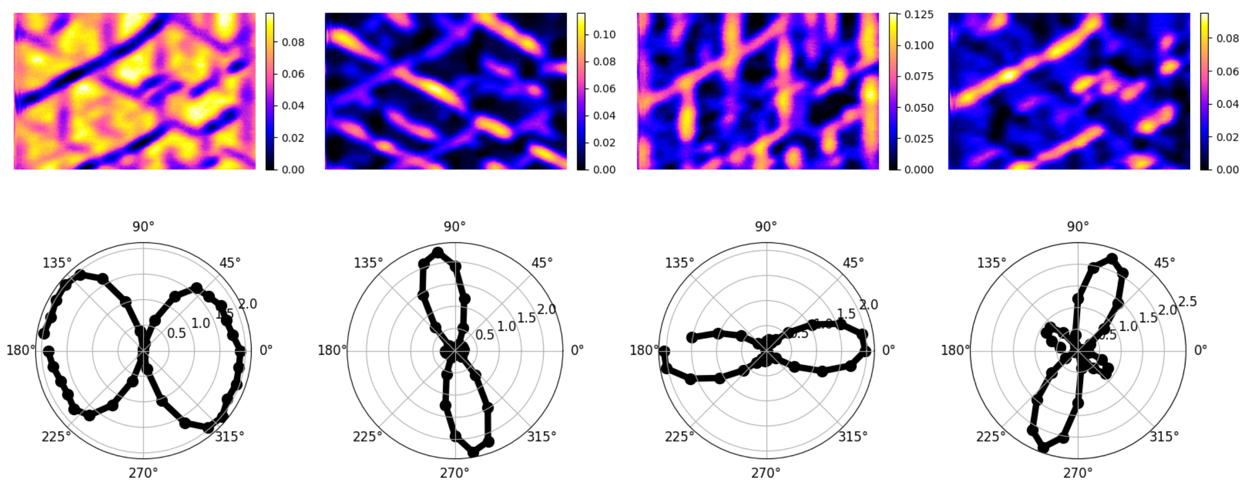
Display K-means results

```
clust_colors = []
clust_colors = [(0,0,0),(1,0,0),(0,0,1),(0,1,0),(1,1,0)] #Black, red, blue,
    green, yellow, cyan
cmap = colors.LinearSegmentedColormap.from_list('randomnameorisitreally',
    clust_colors[:n_components])
norm = colors.BoundaryNorm(np.arange(n_components+1), cmap.N)

fig = plt.figure(figsize=(5*n_components, 5))

plt.subplot(1, n_components+1, 1)
labels = np.reshape(labels, (data_raw.shape[1], data_raw.shape[2]))
plt.imshow(labels, cmap=cmap, norm=norm)
plt.axis("off")

for i in range(n_components):
    plt.subplot(1, n_components+1, i+2, projection='polar')
    plt.plot(angles, centroids[i], '-o', lw=6, ms=10, color=clust_colors[i])
```



Display NMF results

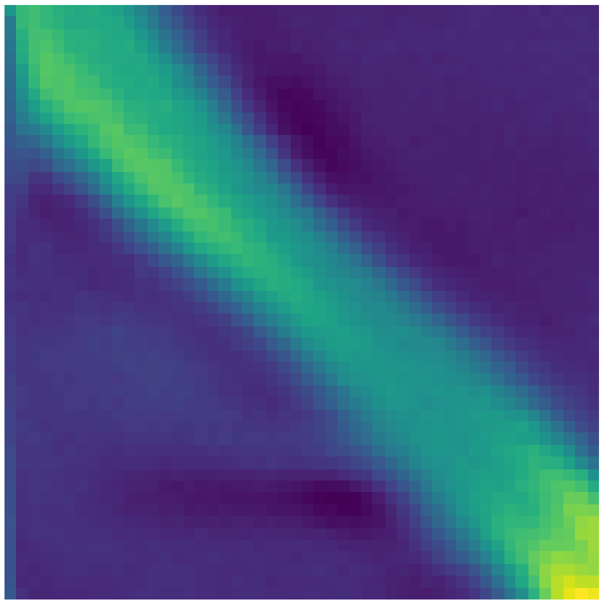
```
plt.figure(figsize=(n_components*5,8))
for i in range(n_components):
        plt.subplot(2, n_components, i+1)
        plt.axis('off')
        plt.imshow(np.reshape(H[:,i], (data_raw.shape[1], data_raw.shape[2])),
            cmap='gnuplot2')
        plt.colorbar(fraction=0.03, pad=0.04)

for i in range(n_components):
    plt.rcParams['font.size'] = 12
    plt.subplot(2, n_components, 1 + n_components + i, projection='polar')
    plt.plot(angles, W[i,:], '-o', lw=6, ms=10, zorder=1, color='black')
```



NMF with regularization: highly mixed signals

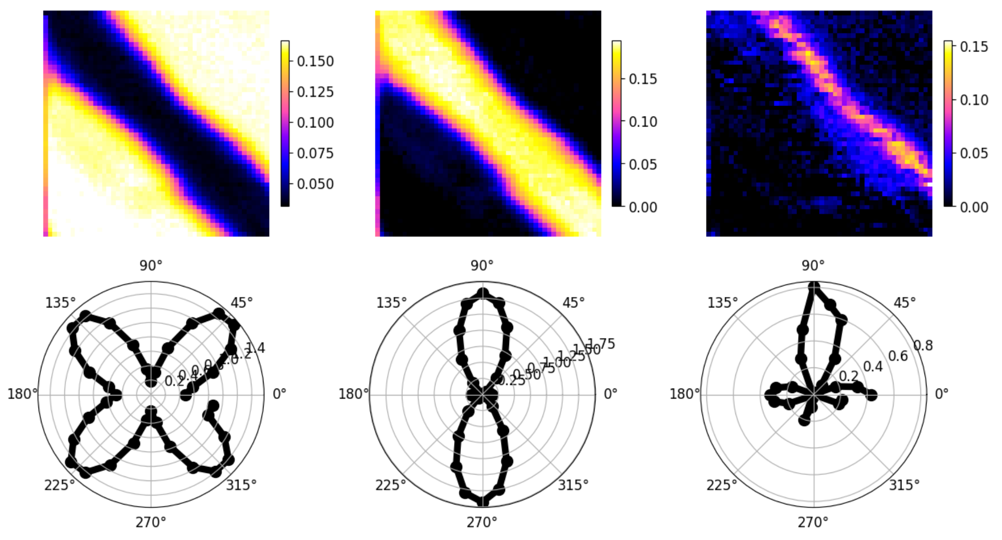
Open an process BTO data

```
n_components = 3
path = "2019-08-30_17-06-08"

data_path = path + "/DAT/"
data_raw = np.stack([np.loadtxt(data_path + data_file) for data_file in os.
    listdir(data_path)])

plt.imshow(np.sum(data_raw, axis=0))
plt.axis('off')
plt.show()

data = np.reshape(data_raw, (data_raw.shape[0], -1)).T
norms = np.linalg.norm(data, axis=1, keepdims=True)
data = data / norms

parameters = np.loadtxt(path + "/parameters.dat")
angles = np.radians(parameters[0,4] + np.arange(0, parameters[1,4]) *
    parameters[2,4])
```



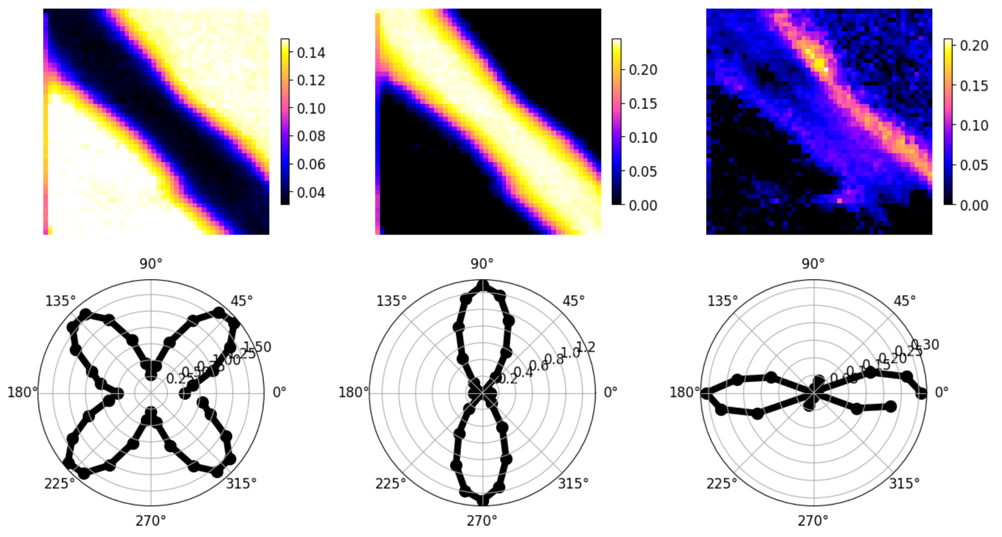
Apply NMF without regularization and display the results

```python
W, H = nmf(data, n_components=n_components, init="nndsvdar", solver="mu",
    max_iter=200)

plt.figure(figsize=(n_components*5,8))
for i in range(n_components):
        plt.subplot(2, n_components, i+1)
        plt.axis('off')
        plt.imshow(np.reshape(H[:,i], (data_raw.shape[1], data_raw.shape[2])),
             cmap='gnuplot2')
        plt.colorbar(fraction=0.03, pad=0.04)

for i in range(n_components):
    plt.rcParams['font.size'] = 12
    plt.subplot(2, n_components, 1 + n_components + i, projection='polar')
    plt.plot(angles, W[i,:], '-o', lw=6, ms=10, zorder=1, color='black')
```



Apply NMF withregularization and display the results

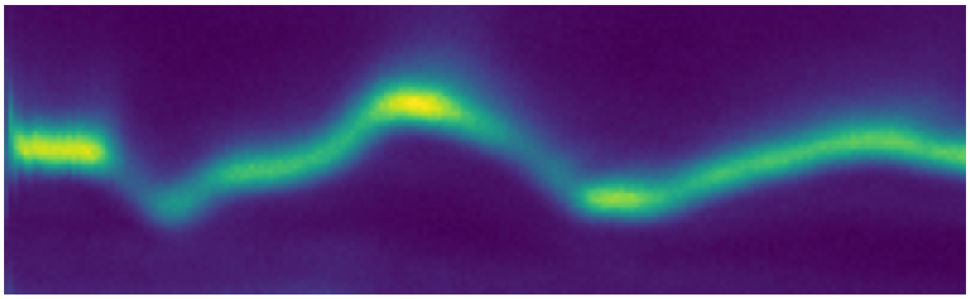
```
W, H = nmf(data, n_components=n_components, init="nndsvdar", solver="mu",
    max_iter=200, polarimetry_reg=500)

plt.figure(figsize=(n_components*5,8))
for i in range(n_components):
        plt.subplot(2, n_components, i+1)
        plt.axis('off')
        plt.imshow(np.reshape(H[:,i], (data_raw.shape[1], data_raw.shape[2])),
            cmap='gnuplot2')
        plt.colorbar(fraction=0.03, pad=0.04)

for i in range(n_components):
    plt.rcParams['font.size'] = 12
    plt.subplot(2, n_components, 1 + n_components + i, projection='polar')
    plt.plot(angles, W[i,:], '-o', lw=6, ms=10, zorder=1, color='black')
```



constrained NMF

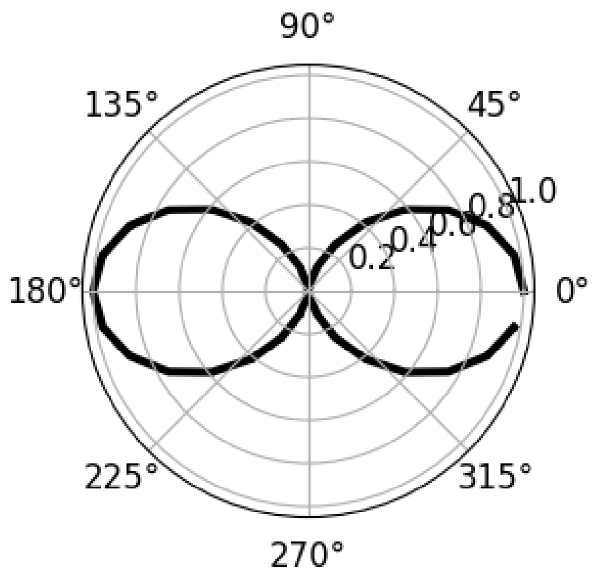
Open an process CLT data

```python
n_components = 2
path = "20230108_09-40-59"

data_path = path + "/DAT/"
data_raw = np.stack([np.loadtxt(data_path + data_file) for data_file in os.
    listdir(data_path)])

plt.imshow(np.sum(data_raw, axis=0))
plt.axis('off')
plt.show()

data = np.reshape(data_raw, (data_raw.shape[0], -1)).T
parameters = np.loadtxt(path + "/parameters.dat")
angles = np.radians(parameters[0,4] + np.arange(0, parameters[1,4]) *
    parameters[2,4])
```



Simulate Bloch and Néel types domain wall polar plots

```python
x = angles
y = np.sin(x) ** 2
y2 = np.cos(x) ** 2

plt.figure(figsize=(8,3))
plt.subplot(1, 2, 2, projection='polar')
plt.plot(x, y2, '-o', lw=3, ms=1, zorder=1, color='black')
plt.show()

Hi = np.stack((y2,y))
```

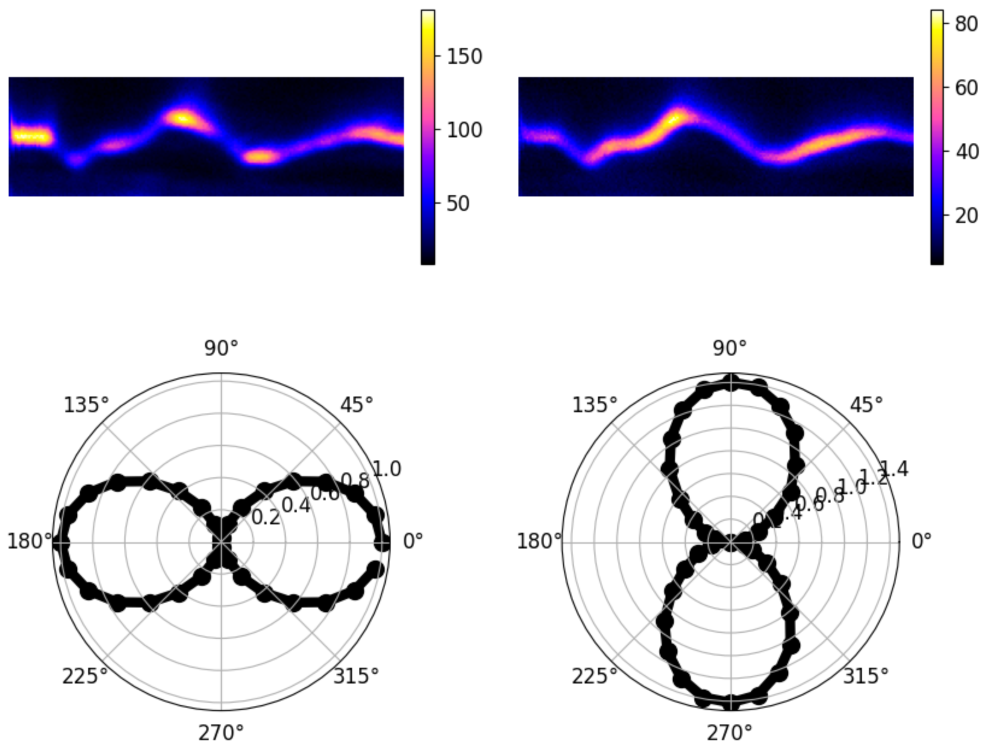


Apply NMF with constraint on the Bloch-type domain wall polar plot

```
W, H = nmf(data, H=Hi, n_components=n_components, init="nndsvda", solver="mu",
    max_iter=200, constrained_components=[0])
```

Display the results

```
plt.figure(figsize=(n_components*5,8))
for i in range(n_components):
        plt.subplot(2, n_components, i+1)
        plt.axis('off')
        plt.imshow(np.reshape(H[:,i], (data_raw.shape[1], data_raw.shape[2])),
            cmap='gnuplot2')
        plt.colorbar(fraction=0.03, pad=0.04)

for i in range(n_components):
    plt.rcParams['font.size'] = 12
    plt.subplot(2, n_components, 1 + n_components + i, projection='polar')
    plt.plot(angles, W[i,:], '-o', lw=6, ms=10, zorder=1, color='black')
```


150
100
50
80
60
40
20
90°
135°
45°
180°
0°
225°
315°
270°
0.2
0.4
0.6
0.8
1.0
1.2
1.4