Abundance_Matrix

December 12, 2022

[3]:
```python
import os
import cv2
import numpy as np
from time import time
import pymatreader as pymat
import matplotlib.pyplot as plt
from pysptools import material_count, eea, abundance_maps, distance
```

☐ **Task 1**

0.0.1

Old Method

[11]:
```python
path_str = os.getcwd() + '/'
```

[12]:
```python
file_name_corr_str = 'Indian_pines_corrected.mat'
file_name_gt_str = 'Indian_pines_gt.mat'
```

[13]:
```python
data_gt_dict = pymat.read_mat(path_str + file_name_gt_str)
data_corr_dict = pymat.read_mat(path_str + file_name_corr_str)

data_gt_ndarray = data_gt_dict['indian_pines_gt']
data_corr_ndarray = data_corr_dict['indian_pines_corrected']
```

[14]:
```python
m, n, d = data_corr_ndarray.shape
data_corr_2d = np.reshape(data_corr_ndarray, newshape=(m*n, d))
data_gt_1d = data_gt_ndarray.flatten()
print(f'\nNew Dimensions of Data: {data_corr_2d.shape}\n')
print(f'New Dimensions of Label: {data_gt_1d.shape}\n')
```

```
New Dimensions of Data: (21025, 200)
New Dimensions of Label: (21025,)
```

[15]:
```
def standard_normalization(data_2d: np.ndarray) -> np.ndarray:

    data = data_2d.copy()
    mean_value = data.mean()
    std_value = data.std()
    new_data = (data - mean_value) / (std_value)
    return new_data
```

[16]:
```
def synthetic_noisy_data_generator(data_2d: np.ndarray, SNR_db: float=0) -> np.
 ↪ndarray:

    data = data_2d.copy()
    m, n = data.shape

    mu, std = 0, 1
    N = np.random.normal(mu, std, size=(m, n))

    SNR = 10 ** (SNR_db / 10)
    E_x2 = (data ** 2).mean()
    A_coeff = (SNR / E_x2) ** (0.5)
    noisy_data = (A_coeff * data) + N
    return noisy_data, N
```

[17]:
```
normal_data = standard_normalization(data_corr_2d)
synthetic_data, N = synthetic_noisy_data_generator(normal_data)
```

[18]:
```
# Test:
expected_std_of_synthetic_data = (normal_data.std() + N.std() +␣
 ↪2*(N*normal_data).mean())**(0.5)
print(f'\nExpected std for synthetic data: {expected_std_of_synthetic_data}\n')
print(f'Actual std of synthetic data: {synthetic_data.std()}\n')
```

```

Expected std for synthetic data: 1.4146508857228337

Actual std of synthetic data: 1.4147488629846336
```

[19]:
```
# Asigning the input data
input_data = synthetic_data
```

[20]:
```
# Rn: Corr
RN = np.identity(input_data.shape[1])
print(f'\nRN is Correlation Matrix of noise that in this example is \
Identity Matrix and shape of this is: {RN.shape}\n')
```

```
RN is Correlation Matrix of noise that in this example is Identity Matrix and
shape of this is: (200, 200)
```

[21]:
```python
# q: No. Endmembers
q = (material_count.vd.hysime(y=input_data, n=N, Rn=RN))[0]
print(f'\nNo. Endmembers(q): {q}\n')
```

```
No. Endmembers(q): 3
```

[22]:
```python
# eea: Endmeber Extraction Algorithm
nfindr = eea.NFINDR()
```

[23]:
```python
# U: Endmeber Matrix
input_data_HSI_cube = input_data.reshape(m, n, d)
U = nfindr.extract(M=input_data_HSI_cube, q=int(q))
print(f'\nShape of the U(Endmembers): {U.shape}\n')
```

```
Shape of the U(Endmembers): (3, 200)
```

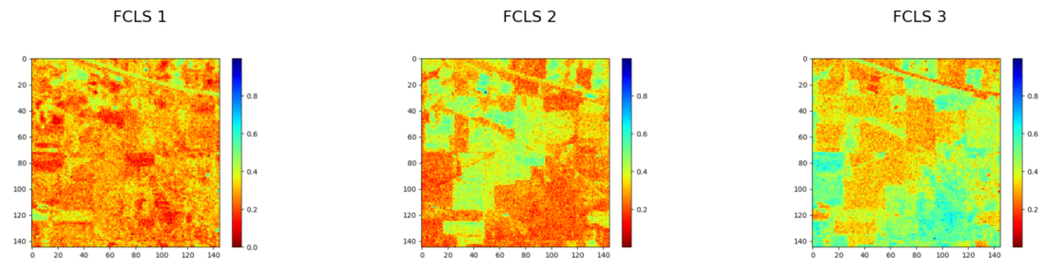
[24]:
```python
tic = time()
X = abundance_maps.amaps.FCLS(M=input_data, U=U)
toc = time()
run_time = toc - tic

print(f'\nShape of Abundance Matrix(X): {X.shape}\n')
print(f'Run time for this method: {run_time}(s)\n')
```

```
Shape of Abundance Matrix(X): (21025, 3)

Run time for this method: 9.057004451751709(s)
```

[25]:
```python
# Exploring the Non-Negativity Constraint(NNC)
b = X[X < 0]
print(b)
```

```
[-1.0180324e-08]
```

```
[26]: # Exploring the Sum-to-one Constraint
      np.sum(X[1000])

[26]: 1.0

[27]: abundance_map = abundance_maps.FCLS()

[28]: U2 = abundance_map.map(M=input_data_HSI_cube, U=U)

[29]: abundance_map.plot(path=path_str)

      <Figure size 640x480 with 0 Axes>

[30]: img1, img2, img3 = cv2.imread(path_str + 'FCLS_1.png'), cv2.imread(path_str +␣
       ↪'FCLS_2.png'), \
      cv2.imread(path_str + 'FCLS_3.png')

      plt.figure(num=1, figsize=(15, 15))
      plt.subplot(1, 3, 1), plt.imshow(img1), plt.axis('off'), plt.title('FCLS 1')
      plt.subplot(1, 3, 2), plt.imshow(img2), plt.axis('off'), plt.title('FCLS 2')
      plt.subplot(1, 3, 3), plt.imshow(img3), plt.axis('off'), plt.title('FCLS 3')
      plt.show()
```



☒ **Task 1**

☐ **Task 2**

0.0.2

SUNSAL Algorithm

[31]:
```
import sunsal
```

[32]:
```
U3, y3 = U.reshape((U.shape[1], U.shape[0])), input_data.reshape((input_data.
 ↪shape[1], input_data.shape[0]))
```

[33]:
```
tic = time()
X_sunsal, _, _, _ = sunsal.sunsal(M=U3, y=y3, positivity=True, addone=True)
toc = time()
sunsal_run_time = toc - tic
print(f'\nRun time for SUNSAL Method: {sunsal_run_time}(s)\n')
```

```

Run time for SUNSAL Method: 0.2668609619140625(s)

```

[34]:
```
X_sunsal.shape
```

[34]:
```
(3, 21025)
```

[35]:
```
# Exploring the Sum-to-one Constraint
np.sum(X_sunsal[:, 1000])
```

[35]:
```
1.0000000198482073
```

[36]:
```
# Exploring the Non-Negativity Constraint(NNC)
b2 = X_sunsal[X_sunsal < 0]
print(b2)
```

```
[]
```

- ☒ **Task 2**
- ☐ **Task 3**

0.0.3

Spectral Angle Mapper(SAM)

[46]:
```
a = X[2, :]
a.shape
```

[46]:
```
(3,)
```

[47]:
```
b = X_sunsal[:, 2]
b.shape
```

[47]:
```
(3,)
```

```
[48]: distance.SAM(a, b)
```

```
[48]: 0.413118177134017
```

☒ **Task 3**

0.0.4 References:

- Old Method
- SUNSAL Algorithm
- Spectral Angle Mapper(SAM)