draft

March 25, 2021

1 hyperspectral-images: spectral unmixing & classification

Assignee full name: Antonopoulos Ilias email: iantonopoulos@aueb.gr ID: P3352004 course: Machine Learning and Computational Statistics (M36104P) program: MSc in Data Science (PT)

2 Reproducibility

To easily reproduce the results of this Jupyter notebook, in a clean & efficient manner, do read the following:

Assuming that a Python (v3.6.x or greater) is installed in your system:

- you could (optionally) upgrade `pip`:

```
python -m pip install --upgrade pip
```

- you could install all the necessary dependencies:

note: the usage of a virtual environment for this is highly advised, in order to keep your system-wide Python interpreter clean of unnecessary dependencies such as `scikit-learn` etc.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import scipy.io as sio
import scipy.optimize

from scipy.spatial import distance
from scipy.stats import multivariate_normal, norm
from sklearn import linear_model
from sklearn.base import BaseEstimator
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.metrics import confusion_matrix, make_scorer
from sklearn.model_selection import cross_val_score, KFold
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
```

2.1 Exploratory Data Analysis

We'll begin by loading all the available data, trying to develop an initial understanding of the problem at hand.

[2]:
```python
salinas = sio.loadmat('data/Salinas_cube.mat')

salinas
```

[2]:
```
{'__header__': b'MATLAB 5.0 MAT-file, Platform: PCWIN64, Created on: Mon Mar  1
14:46:31 2021',
 '__version__': '1.0',
 '__globals__': [],
 'salinas_cube': array([[[369, 579, 866, …,  31,   9,  15],
         [369, 495, 735, …,  33,  13,  15],
         [369, 495, 866, …,  33,  11,  19],
         …,
         [373, 398, 725, …,  12,   4,   2],
         [373, 398, 659, …,   8,   4,   0],
         [373, 482, 594, …,   8,   0,   5]],

        [[441, 558, 787, …,  26,  11,  16],
         [441, 558, 787, …,  32,   7,  12],
         [441, 474, 787, …,  26,   9,  16],
         …,
         [447, 393, 590, …,   3,   0,   9],
         [376, 393, 655, …,  11,   0,   6],
         [376, 393, 590, …,   3,   5,  -3]],

        [[444, 566, 790, …,  30,  10,  15],
         [373, 566, 790, …,  30,  12,  21],
         [373, 398, 790, …,  32,  16,  13],
         …,
         [305, 468, 534, …,   6,   3,  -1],
         [376, 384, 664, …,   6,   1,  -3],
         [376, 384, 599, …,   0,   0,   8]],


        …,


        [[381, 568, 799, …,  76,  25,  37],
         [381, 568, 799, …,  34,  15,  23],
         [381, 401, 799, …,  10,   3,   0],
         …,
         [369, 466, 599, …,  30,  13,  11],
         [369, 466, 730, …,  34,  11,  23],
         [227, 383, 599, …,  40,  13,  15]],

        [[369, 466, 730, …,  72,  19,  37],
```

```
 [369, 466, 664, …,  62,  27,  39],
 [369, 550, 795, …,  40,  11,  19],
 …,
 [444, 477, 609, …,  34,  15,  18],
 [301, 477, 609, …,  34,  15,  22],
 [301, 477, 675, …,  36,  13,  24]],

[[369, 466, 730, …,  72,  19,  37],
 [369, 466, 664, …,  62,  27,  39],
 [369, 550, 795, …,  40,  11,  19],
 …,
 [368, 485, 610, …,  32,  13,  19],
 [368, 568, 676, …,  42,  20,  21],
 [297, 568, 610, …,  38,  13,  21]]], dtype=int16)}
```

[3]:
```
hsi = salinas['salinas_cube']

hsi.shape
```

[3]:
```
(220, 120, 204)
```

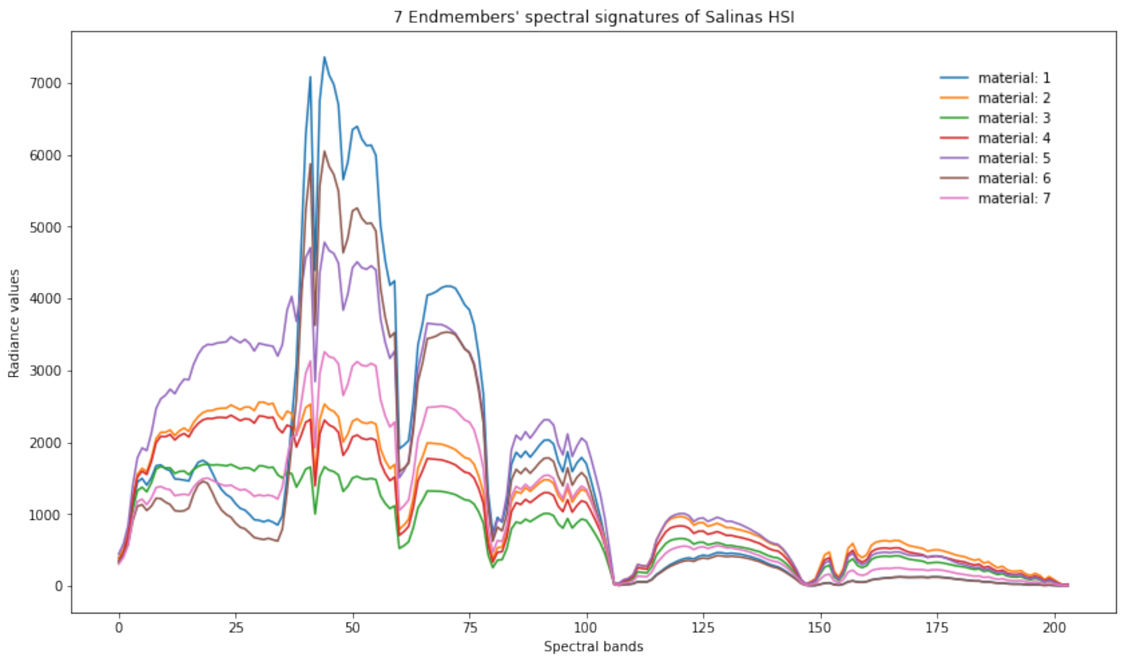
[4]:
```
ends = sio.loadmat('data/Salinas_endmembers.mat')

ends
```

[4]:
```
{'__header__': b'MATLAB 5.0 MAT-file, Platform: PCWIN64, Created on: Tue Mar 23
15:44:08 2021',
 '__version__': '1.0',
 '__globals__': [],
 'salinas_endmembers': array([[392.98079561, 388.55390904, 325.42702051, …,
446.79332153,
         345.42833194, 306.5428824 ],
        [496.35070873, 504.777721  , 418.4185766 , …, 585.49152542,
         430.39340598, 398.5928382 ],
        [702.8079561 , 735.25140521, 630.0747889 , …, 838.19251202,
         574.78066499, 566.7683466 ],
        …,
        [  4.96799268,  48.75217169,  28.2907117 , …,  31.76979509,
           4.73959206,  15.29619805],
        [  1.9304984 ,  17.03628002,   9.76839566, …,  11.04528206,
           1.76809165,   5.29045093],
        [  2.83539095,  27.07307103,  15.36670688, …,  17.61801164,
           2.63593182,   8.59195402]])}
```

[5]:
```
endmembers = ends['salinas_endmembers']

endmembers.shape
```

[5]: (204, 7)

[6]:
```
fig = plt.figure(figsize=(14, 8))

plt.plot(endmembers)
plt.legend(['material: {}'.format(i + 1) for i in range(7)], bbox_to_anchor=(0.
↪95, 0.95), framealpha=0)
plt.ylabel('Radiance values')
plt.xlabel('Spectral bands')
plt.title("7 Endmembers' spectral signatures of Salinas HSI")

plt.show()
```



[7]:
```
ground_truth = sio.loadmat('data/Salinas_gt.mat')

ground_truth
```

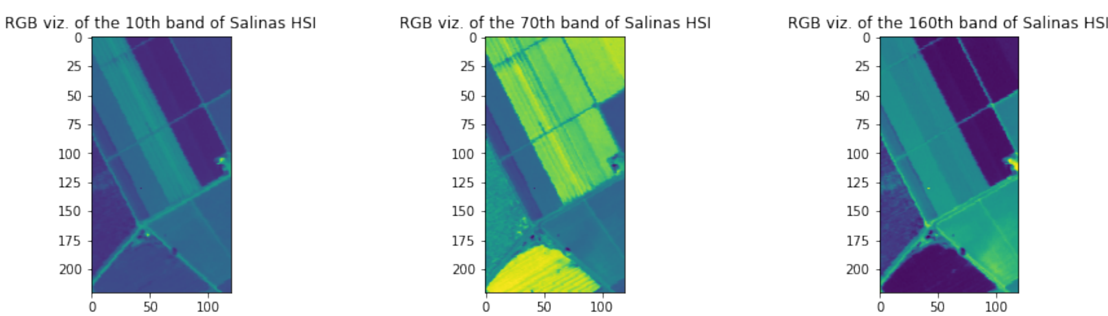
[7]:
```
{'__header__': b'MATLAB 5.0 MAT-file, Platform: PCWIN64, Created on: Mon Mar  1
23:21:46 2021',
 '__version__': '1.0',
 '__globals__': [],
 'salinas_gt': array([[0, 0, 0, …, 0, 0, 0],
        [6, 6, 6, …, 0, 0, 0],
        [6, 6, 6, …, 0, 0, 0],
        …,
        [0, 0, 0, …, 0, 0, 0],
```

```
      [0, 0, 0, …, 0, 0, 0],
      [0, 0, 0, …, 0, 0, 0]], dtype=uint8)}
```

[8]:
```
labels = ground_truth['salinas_gt']

labels.shape
```

[8]: (220, 120)

[9]:
```
fig = plt.figure(figsize=(20, 8))

ax = fig.add_subplot(2,2,1)

ax.imshow(hsi[:, :, 10])
ax.set_title('RGB viz. of the 10th band of Salinas HSI')

ax = fig.add_subplot(2,3,2)

ax.imshow(hsi[:, :, 70])
ax.set_title('RGB viz. of the 70th band of Salinas HSI')

ax = fig.add_subplot(2,2,2)

ax.imshow(hsi[:, :, 160])
ax.set_title('RGB viz. of the 160th band of Salinas HSI')

plt.show()
```



We can definitely see an area with landfields, a partial road network etc.

According to the dataset, it is an area of the Salinas valley in California, USA.

[10]:
```
salinas_labels = sio.loadmat('data/classification_labels_Salinas.mat')

salinas_labels
```

```
[10]: {'__header__': b'MATLAB 5.0 MAT-file, Platform: PCWIN64, Created on: Mon Mar  1
16:49:08 2021',
 '__version__': '1.0',
 '__globals__': [],
 'operational_set': array([[0],
        [0],
        [0],
        …,
        [0],
        [0],
        [0]], dtype=uint8),
 'test_set': array([[0],
        [0],
        [0],
        …,
        [0],
        [0],
        [0]], dtype=uint8),
 'training_set': array([[0],
        [6],
        [6],
        …,
        [0],
        [0],
        [0]], dtype=uint8)}
```

3 Spectral Unmixing

In this first part, our aim is to perform spectral unmixing on each one of the pixels in the image with nonzero label, with respect to the $m = 7$ endmembers.

We adopt the **linear spectral unmixing hypothesis**:

$$y = X\theta + \eta$$

where:

- y is the L-dimensional spectral signature of the pixel under study
- X is composed by the spectral signatures $x_1, \ldots, x_m$ of the pure pixels (i.e. pure materials) in the image - they are also L-dimensional columns
- θ is the m-dimensional abundance vector of the pixel
- η is the L-dimensional i.i.d., zero-mean Gaussian noise vector

We also define the reconstruction error as follows:

$$\frac{1}{N}\sum_{n=1}^{N}\|y_i - X\theta_i\|^2$$

note: in our particular problem, N designates the total number of pixels in the image with non-zero label.

[11]:
```python
xi, yi = np.nonzero(labels)

nonzero_hsi = hsi[xi, yi, :]

nonzero_hsi.shape
```

[11]: (16929, 204)

3.0.1 (a) Least Squares, with no constraints

We will firstly approach this task via the unconstrained Least Squares method.

That is, we will solve the problem:

$$\text{argmin}_{\theta} J(\theta), \;\; \text{where} \;\; J(\theta) = \|y - X\theta\|^2$$

It can be shown that:

$$\hat{\theta} = (X^T X)^{-1} X^T y$$

[12]:
```python
def unconstrained_least_squares_solver(image, endmembers):
    """Implements a Least Squares solver, assuming no constraints.

    Args:
        image: an (x, l) array, that contains non-zero pixels.
        endmembers: an (l, 7) array.

    Returns:
        The Least Squares solution, as an (7, x) array, containing the unmixing␣
↪estimates.
    """
    inverse = np.linalg.inv(np.dot(endmembers.T, endmembers))

    return inverse.dot(endmembers.T).dot(image.T)
```

[13]:
```python
def abundance_maps(estimates, xi, yi):
    """Plots the abundance maps for the 7 materials.

    Args:
        estimates: an (7, x) array, containing the unmixing estimates.
        xi: a (x,) array with the non-zero x positions of pixels.
        yi: a (y,) array with the non-zero y positions of pixels

    Returns:
        The abundance maps.
    """
```

```
    fig, axs = plt.subplots(3, 3, figsize=(20, 20), facecolor='w',␣
↪edgecolor='k')
    axs = axs.ravel()

    abundance_maps = np.zeros((225, 130, 9))

    for i in range(7):

        abundance_maps[xi, yi, i] = estimates[i, :]

        axs[i].imshow(abundance_maps[:, :, i])
        axs[i].set_title('material: {}'.format(i + 1))
        axs[i].grid(False)

    fig.tight_layout()

    # remove 8th and 9th subplot entry of the 3x3 grid
    fig.delaxes(axs[-1])
    fig.delaxes(axs[-2])
```

```
[14]: def reconstruction_error(image, endmembers, labels, estimates):
    """Implements the reconstruction error metric.

    Args:
        image: an (x, l) array, that contains non-zero pixels.
        endmembers: an (l, 7) array.
        labels: an (m, n) array.
        estimates: an (7, x) array, containing the unmixing estimates.

    Returns (float):
        The reconstruction error.
    """
    n = np.count_nonzero(labels)

    return np.linalg.norm(image.T - np.dot(endmembers, estimates)) ** 2 / n
```

Let's proceed with the calculations:

```
[15]: estimation = unconstrained_least_squares_solver(nonzero_hsi, endmembers)

estimation.shape
```

```
[15]: (7, 16929)
```

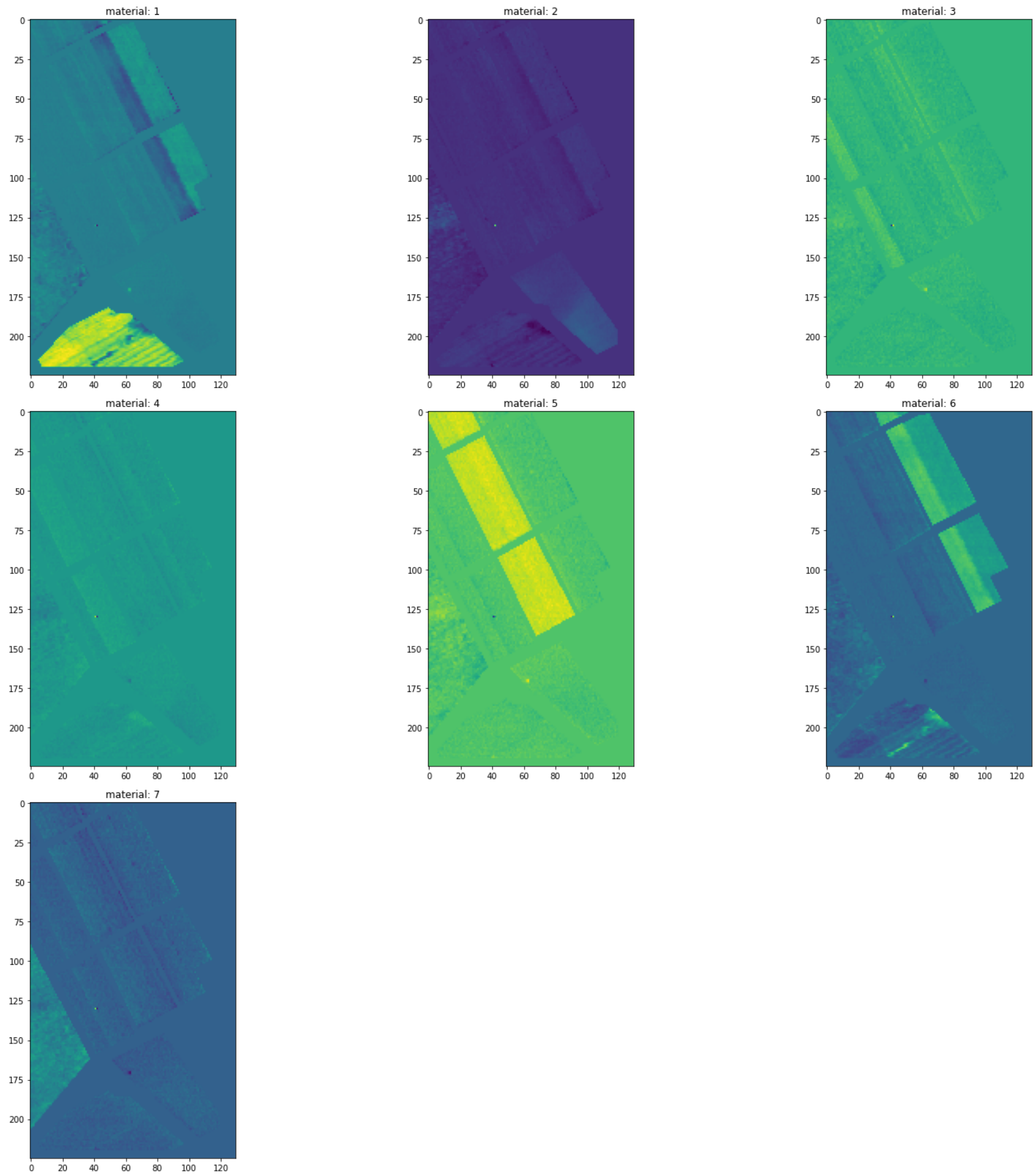
```
[16]: error1 = reconstruction_error(nonzero_hsi, endmembers, labels, estimation)

print('method: unconstrained Least Squares')
```

```python
print(f'reconstruction error: {error1}\n')
```

```
method: unconstrained Least Squares
reconstruction error: 35058.88066277267
```

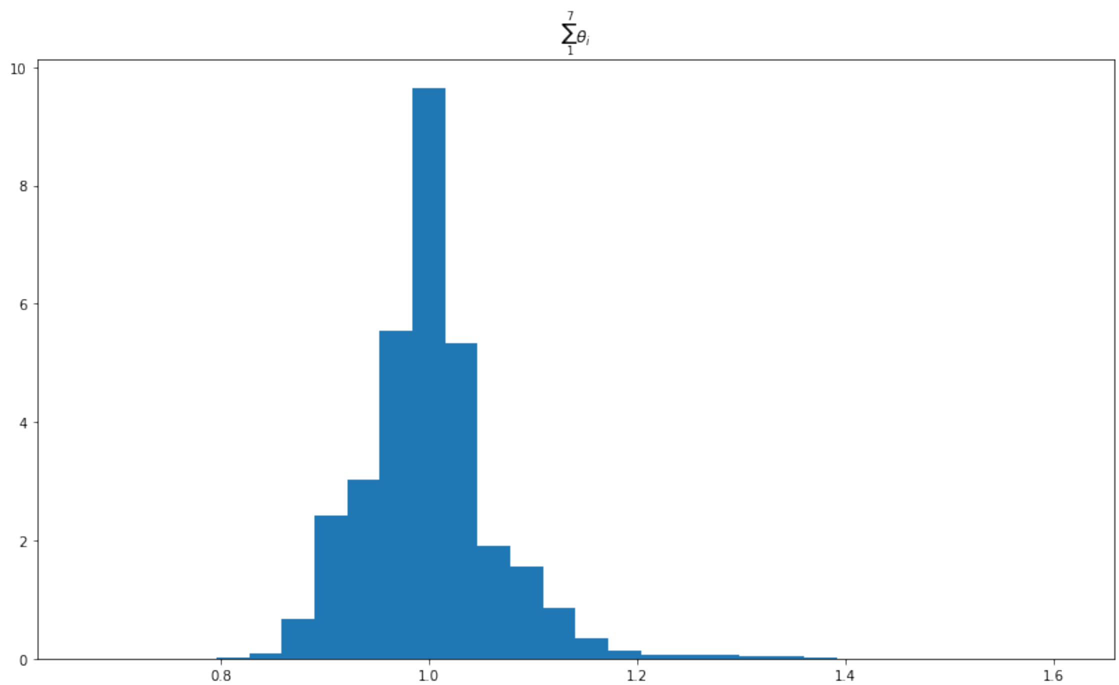
[17]:
```python
print('abundance map per endmember/material:')

abundance_maps(estimation, xi, yi)
```

```
abundance map per endmember/material:
```



[18]:
```
def visualize_estimates(estimates):
    """Visualizes the estimates across materials.

    Args:
        estimates: an (7, x) array, containing the unmixing estimates.

    Returns:
        The visualization.
    """
    fig = plt.figure(figsize=(14, 8))

    ax = fig.add_subplot(111)

    ax.hist(np.sum(estimation, axis=0),
            density=True,
            bins=30)

    ax.set_title('$\sum_1^7 \\theta_i$')

    plt.show()
```

[19]:
```
visualize_estimates(estimation)
```



3.0.2 (b) Least Squares, with a sum-to-one constraint

We will now include a sum-to-one constraint to our Least Squares problem:

That is, we will solve the problem:

$$\text{argmin}_\theta J(\theta), \text{ where } J(\theta) = \|y - X\theta\|^2, \text{ subject to} \sum_{i=1}^{7} \theta_i = 1$$

There are a couple of ways to subject the problem to the sum-to-one constraint. Namely: - solve the unconstrained problem and perform suitable post-transformation to recover the under-constraint solution - introduce constraint as an extra problem equation, along with a weighting policy in favor of this equation, so as to "force" solution to uphold constraint. - etc.

We will utilize the scipy.optimize.minimize function.

[20]:
```
def sum_to_one_squares_solver(image, endmembers, labels):
    """Implements a Least Squares solver, assuming the sum-to-one constraint.

    Args:
        image: an (x, l) array, that contains non-zero pixels.
        endmembers: an (l, 7) array.
        labels: an (m, n) array.

    Returns:
        The Least Squares solution, as an (7, x) array, containing the unmixing␣
↪estimates.
    """
    # define objective function
    def obj_func(x, a, b):
        return np.linalg.norm(a.dot(x) - b) ** 2

    # define constraint(s)
    constraints = {'type': 'eq', 'fun': lambda y: np.sum(y) - 1}

    # define minimization strategy
    def minimizer(c):

        inits = np.zeros((1, 7))

        for i in range(c):

            res = scipy.optimize.minimize(
                obj_func,
                inits,
                args=(endmembers, image[i, :]),
                method='SLSQP',
                tol='1e-6',
```

```python
                constraints=constraints,
            )

            yield res.x

    n = np.count_nonzero(labels)

    return np.array([*minimizer(n)]).T
```

[21]:
```python
estimation = sum_to_one_squares_solver(nonzero_hsi, endmembers, labels)

estimation.shape
```

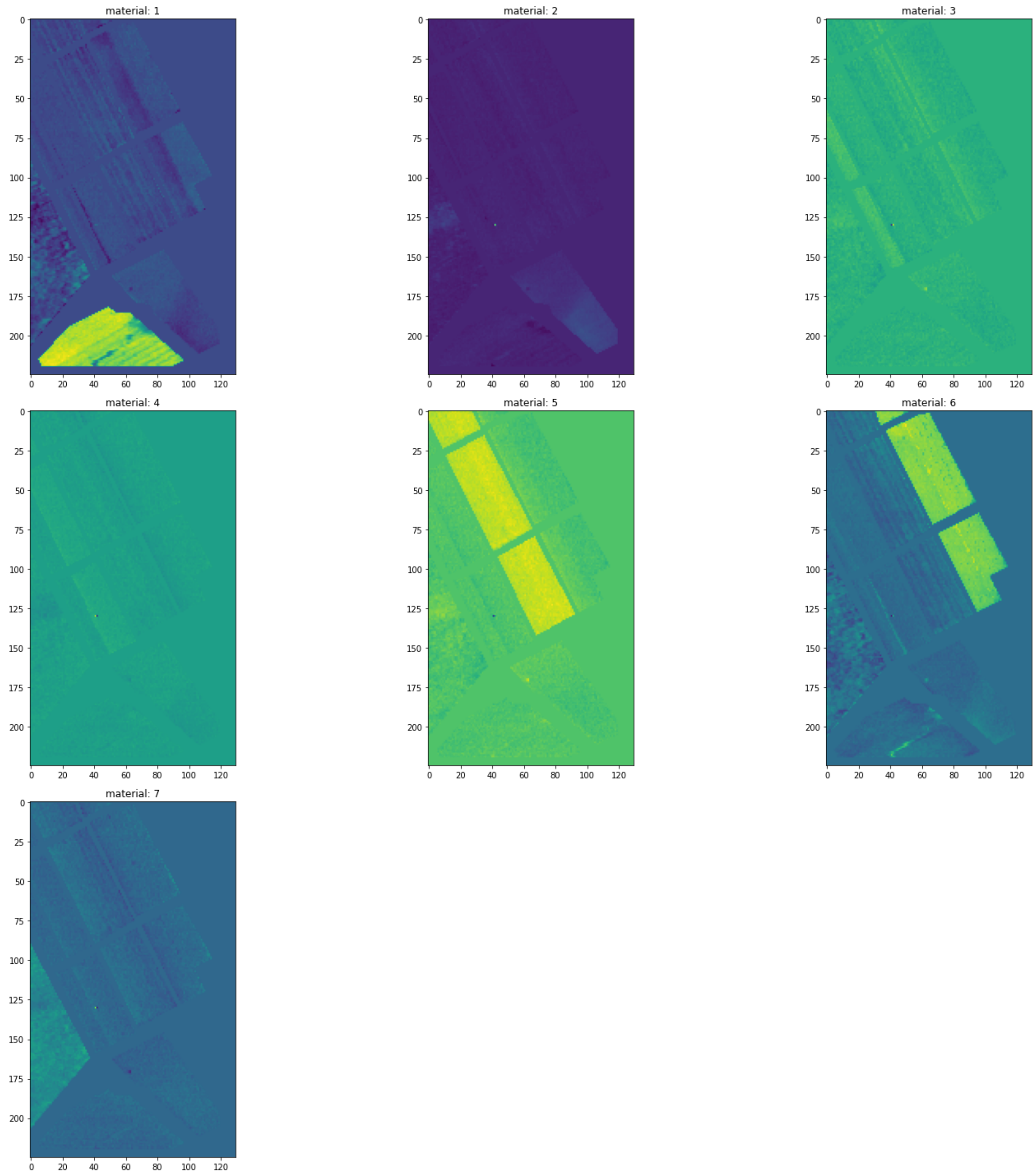
[21]: (7, 16929)

[22]:
```python
error2 = reconstruction_error(nonzero_hsi, endmembers, labels, estimation)

print('method: Least Squares with sum-to-one constraint')
print(f'reconstruction error: {error2}\n')
```

```
method: Least Squares with sum-to-one constraint
reconstruction error: 43082.576302782494
```

[23]:
```python
print('abundance map per endmember/material:')

abundance_maps(estimation, xi, yi)
```

```
abundance map per endmember/material:
```



[24]:
```
np.sum(estimation, axis=0)
```

[24]: `array([1., 1., 1., …, 1., 1., 1.])`

[25]:
```
np.sum(np.sum(estimation, axis=0))
```

[25]: `16929.0`

The estimated params indeed respect the sum-to-one constraint.

[26]:
```
# TODO: add also my WLS approach.
```

3.0.3 (c) Least Squares, with a non-negativity constraint

We will now attempt to solve the LS problem, by introducing a non-negativity constraint:

$$\text{argmin}_{\theta} J(\theta), \;\text{ where }\; J(\theta) = \|y - X\theta\|^2, \;\text{ subject to }\; \theta \geq 0$$

A key observation here is that we cannot use a direct approach as in **(a)** but rather an iterative algorithm is required.

Since this is a straightforward restriction (enforcing bounds on the values a parameter can get) we will utilize the respective open-source implementation: scipy.optimize.nnls

[27]:
```
def nonnegative_least_squares_solver(image, endmembers, labels):
    """Implements a Least Squares solver, assuming the non-negativity␣
↪constraint.

    Args:
        image: an (x, l) array, that contains non-zero pixels.
        endmembers: an (l, 7) array.
        labels: an (m, n) array.

    Returns:
        The Least Squares solution, as an (7, x) array, containing the unmixing␣
↪estimates.
    """
    def optimizer(c):

        for i in range(c):

            theta, _ = scipy.optimize.nnls(endmembers, image[i, :])

            yield theta

    n = np.count_nonzero(labels)

    return np.array([*optimizer(n)]).T
```

[28]:
```
estimation = nonnegative_least_squares_solver(nonzero_hsi, endmembers, labels)

estimation.shape
```

[28]: (7, 16929)

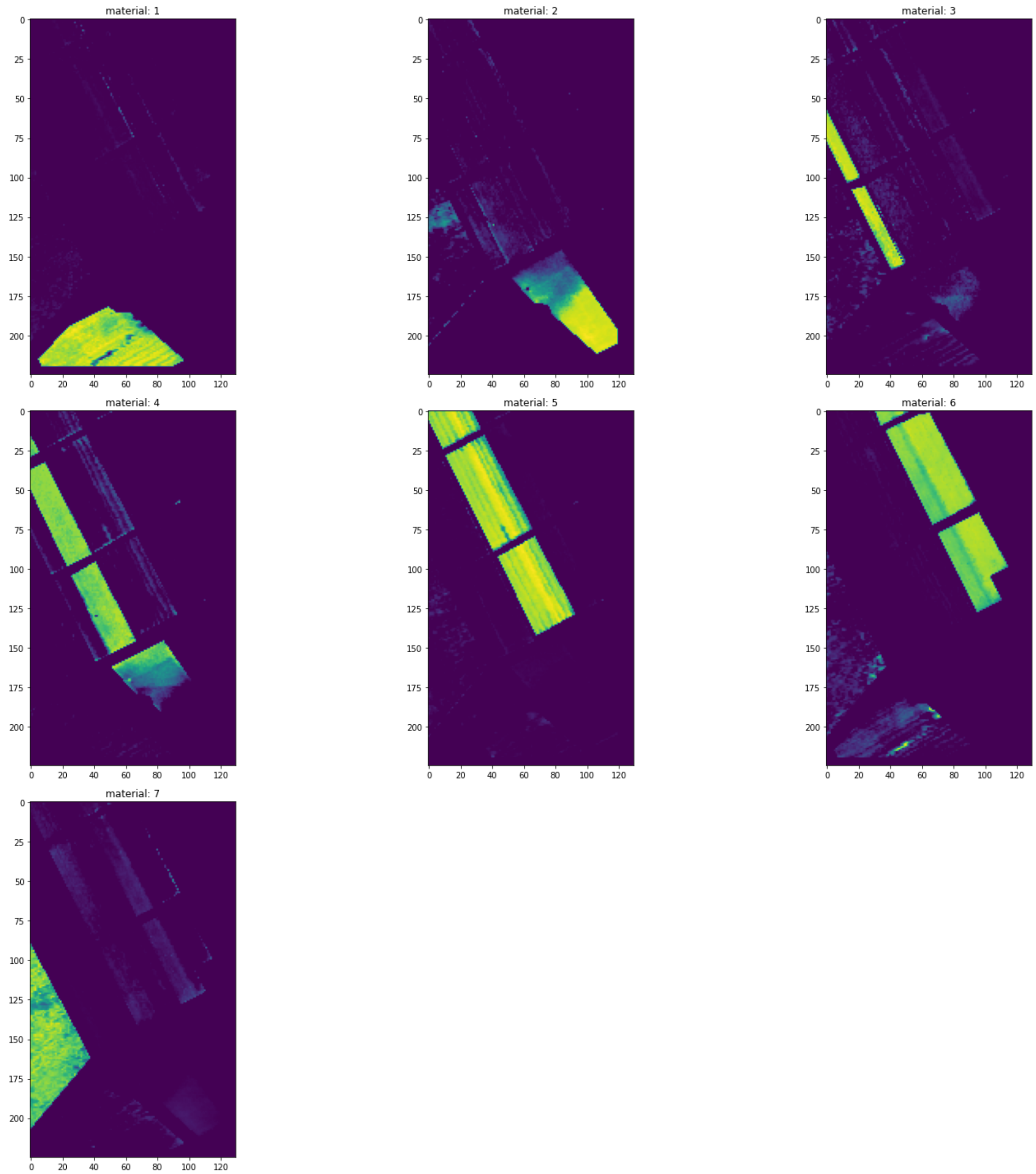
[29]:
```
error3 = reconstruction_error(nonzero_hsi, endmembers, labels, estimation)
```

```python
print('method: Least Squares with non-negativity constraint')
print(f'reconstruction error: {error3}\n')
```

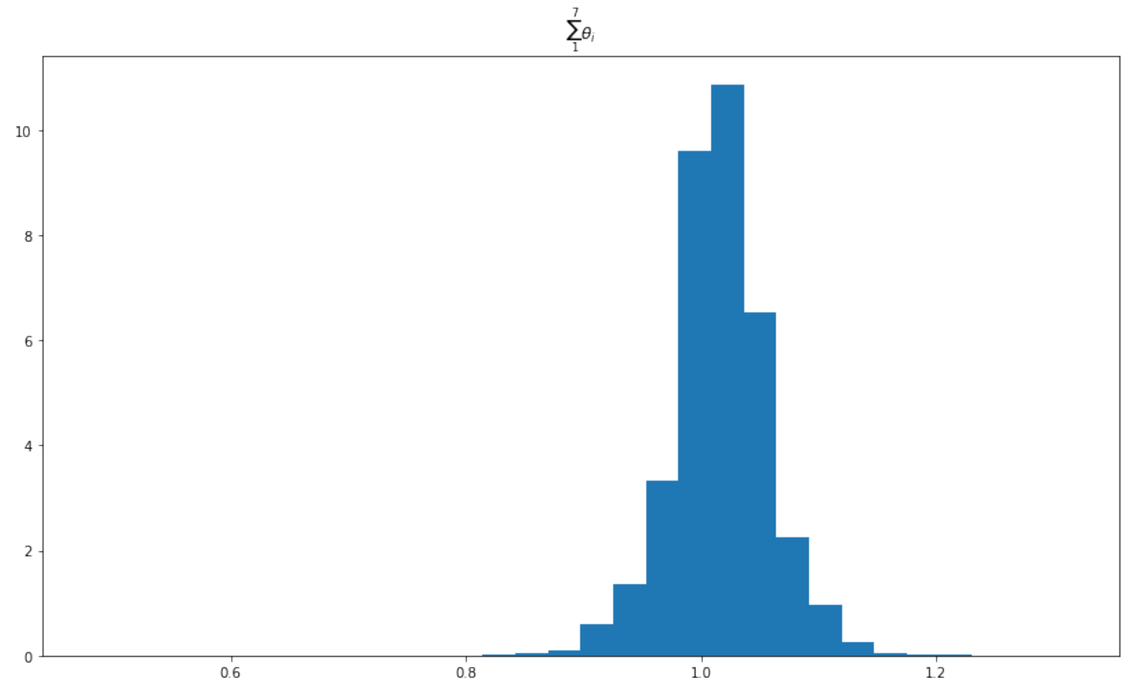
```
method: Least Squares with non-negativity constraint
reconstruction error: 156104.18220644674
```

[30]:
```python
print('abundance map per endmember/material:')

abundance_maps(estimation, xi, yi)
```

```
abundance map per endmember/material:
```



[31]:
```
visualize_estimates(estimation)
```



3.0.4 (d) Least Squares, with sum-to-one & non-negativity constraints

We will now combine the previous two, by introducing a non-negativity constraint as well as a sum-to-one constraint:

$$\text{argmin}_\theta J(\theta), \;\text{ where }\; J(\theta) = \|y - X\theta\|^2, \;\text{ subject to }\; \theta \geq 0 \;\text{ and }\; \sum_{i=1}^{7} \theta_i = 1$$

Again, an iterative algorithm is required to solve the problem. We will again go with scipy.optimize.minimize function.

Before going into the implementation details, it is worth noting that something like:

$$\theta = [1/7, \ldots, 1/7]^T$$

is a reasonable parameter configuration, since both constraints need to be upheld.

Let's see how it plays out in practice.

[32]:
```
def nn_and_sum_to_one_squares_solver(image, endmembers, labels):
    """Implements a Least Squares solver, assuming the sum-to-one and␣
↪non-negativity constraints.

    Args:
        image: an (x, l) array, that contains non-zero pixels.
```

```python
        endmembers: an (l, 7) array.
        labels: an (m, n) array.

    Returns:
        The Least Squares solution, as an (7, x) array, containing the unmixing␣
↪estimates.
    """
    # define objective function
    def obj_func(x, a, b):
        return np.linalg.norm(a.dot(x) - b) ** 2

    # define constraint(s)
    constraints = {'type': 'eq', 'fun': lambda y: np.sum(y) - 1}
    bounds = [[0, None]] * endmembers.shape[1]

    # define minimization strategy
    def minimizer(c):

        inits = np.zeros((1, 7))

        for i in range(c):

            res = scipy.optimize.minimize(
                obj_func,
                inits,
                args=(endmembers, image[i, :]),
                bounds=bounds,
                method='SLSQP',
                tol='1e-6',
                constraints=constraints,
            )

            yield res.x

    n = np.count_nonzero(labels)

    return np.array([*minimizer(n)]).T
```

[33]:
```python
estimation = nn_and_sum_to_one_squares_solver(nonzero_hsi, endmembers, labels)

estimation.shape
```

[33]: (7, 16929)

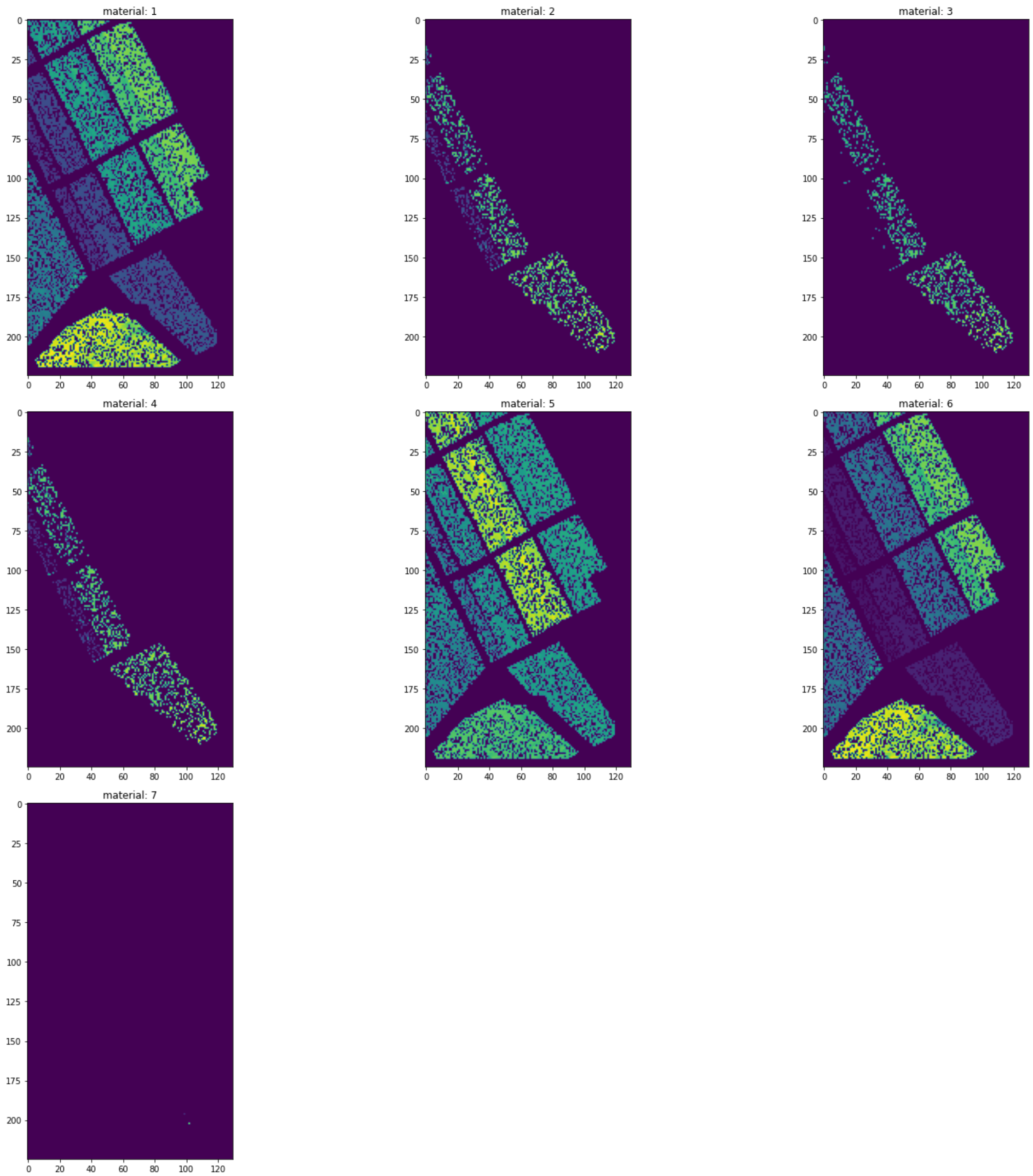
[34]:
```python
error4 = reconstruction_error(nonzero_hsi, endmembers, labels, estimation)

print('method: Least Squares with sum-to-one + non-negativity constraints')
```

```python
print(f'reconstruction error: {error4}\n')
```

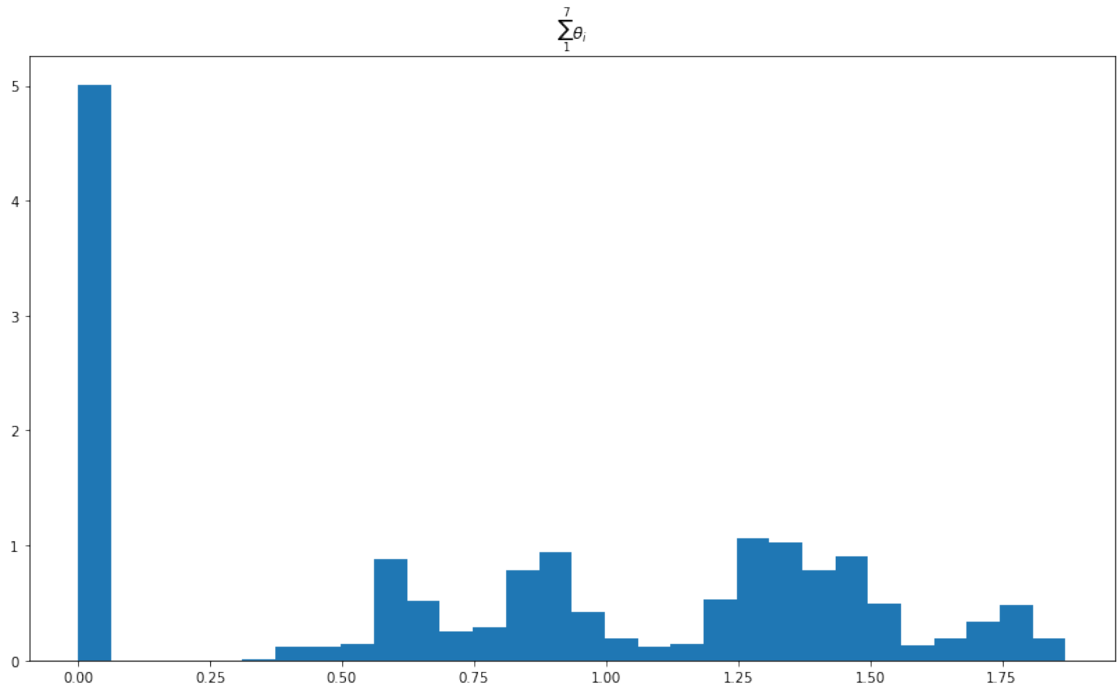
```
method: Least Squares with sum-to-one + non-negativity constraints
reconstruction error: 339088712.2633131
```

[35]:
```python
print('abundance map per endmember/material:')

abundance_maps(estimation, xi, yi)
```

```
abundance map per endmember/material:
```



```
[36]: visualize_estimates(estimation)
```



3.0.5 (e) LASSO

We would now try to impose a sparsity on θ, via LASSO and L_1 norm minimization.

That is, we would like to solve the following regularized Least Squares problem:

$$\text{argmin}_\theta J(\theta), \text{ where } J(\theta) = \|y - X\theta\|^2, \text{ subject to } \|\theta\|_1 \leq \rho$$

We will utilize the scikit-learn-provided LASSO i.e. a linear model trained with L_1 prior as regularizer.

A low reconstruction error can be yielded by a Lagrangian of `37`.

```
[37]: def lasso_least_squares_solver(image, endmembers, labels):
          """Implements a Least Squares solver, with a LASSO regularization scheme.

          Args:
              image: an (x, l) array, that contains non-zero pixels.
              endmembers: an (l, 7) array.
              labels: an (m, n) array.
```

```
    Returns:
        The Least Squares solution, as an (7, x) array, containing the unmixing␣
↪estimates.
    """
    clf = linear_model.Lasso(alpha=37, positive=True, fit_intercept=False,␣
↪max_iter=1e7)

    def optimizer(c):

        for i in range(c):

            clf.fit(endmembers, image[i, :])

            yield clf.coef_

    n = np.count_nonzero(labels)

    return np.array([*optimizer(n)]).T
```

[38]:
```
estimation = lasso_least_squares_solver(nonzero_hsi, endmembers, labels)

estimation.shape
```

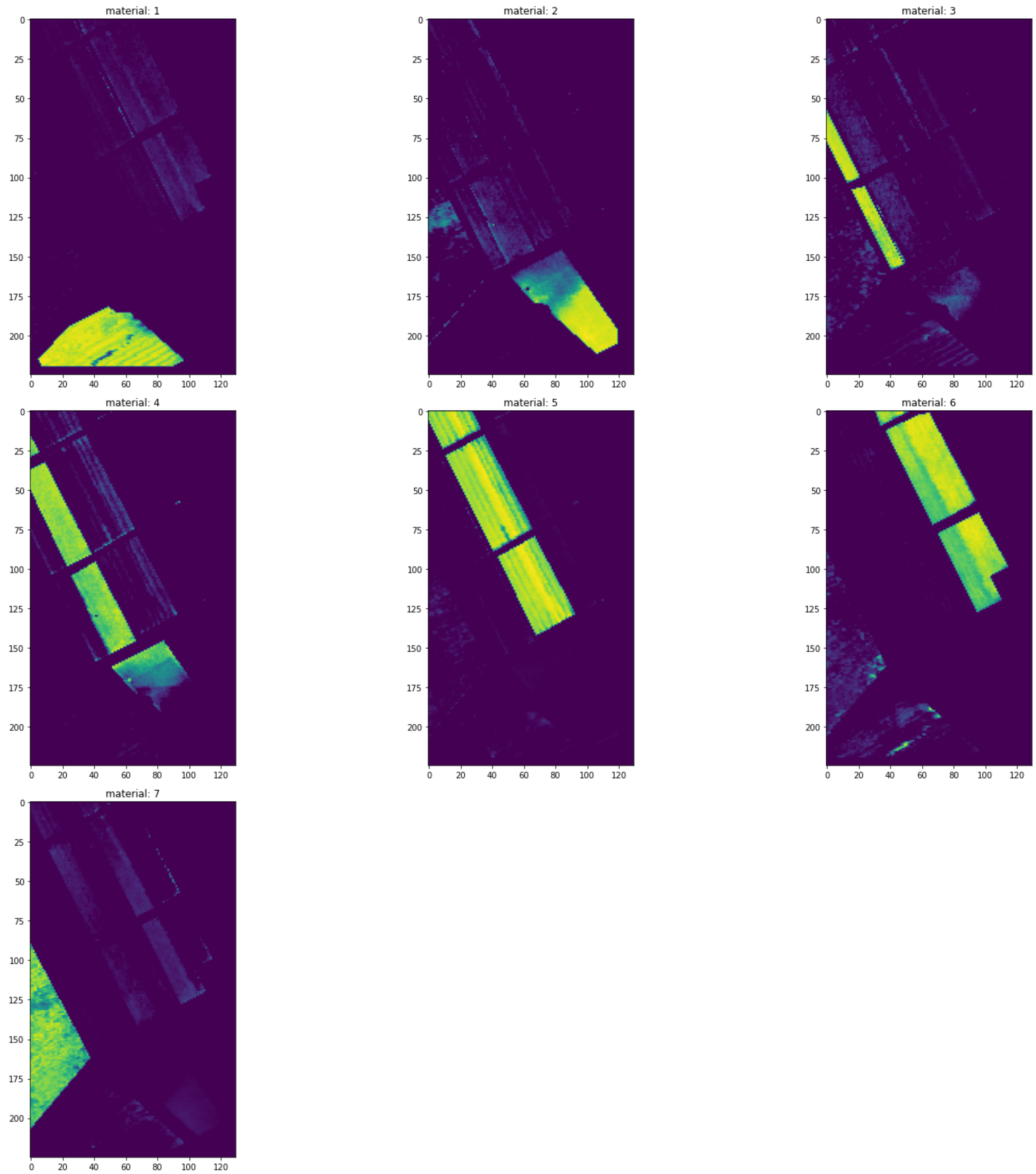
[38]: (7, 16929)

[39]:
```
error5 = reconstruction_error(nonzero_hsi, endmembers, labels, estimation)

print('method: LASSO')
print(f'reconstruction error: {error5}\n')
```

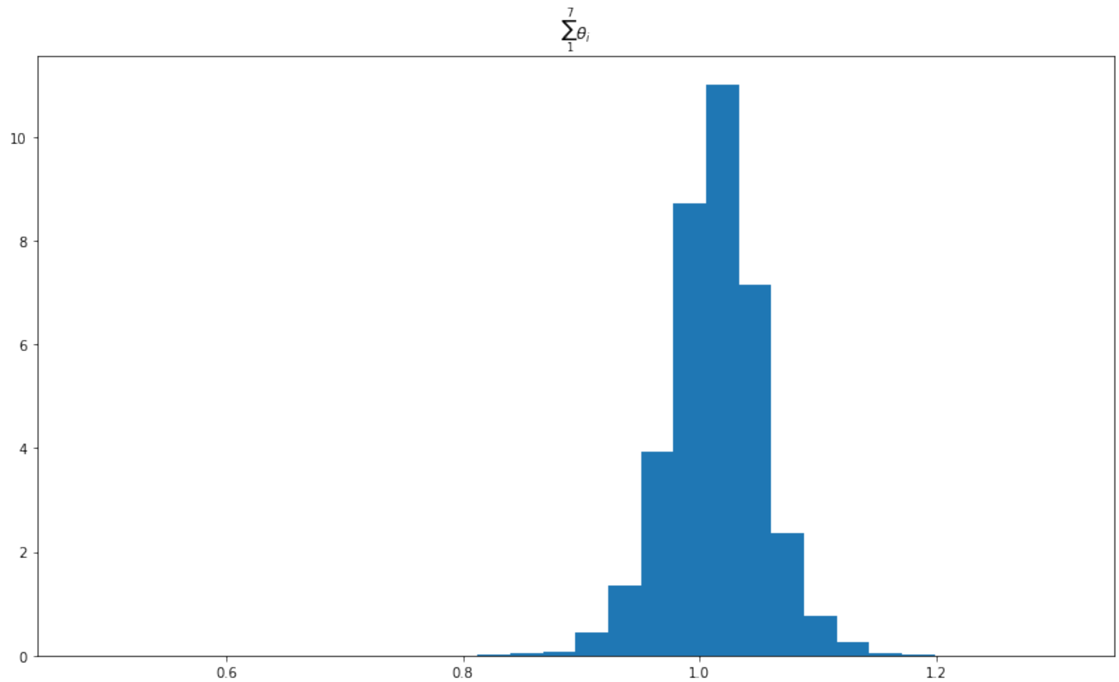
```
method: LASSO
reconstruction error: 158097.38670329918
```

[40]:
```
print('abundance map per endmember/material:')

abundance_maps(estimation, xi, yi)
```

```
abundance map per endmember/material:
```



[41]:
```
visualize_estimates(estimation)
```



3.0.6 Spectral Unmixing - Comparison and Remarks

[42]:
```
# TODO: write remarks about the comparison
```

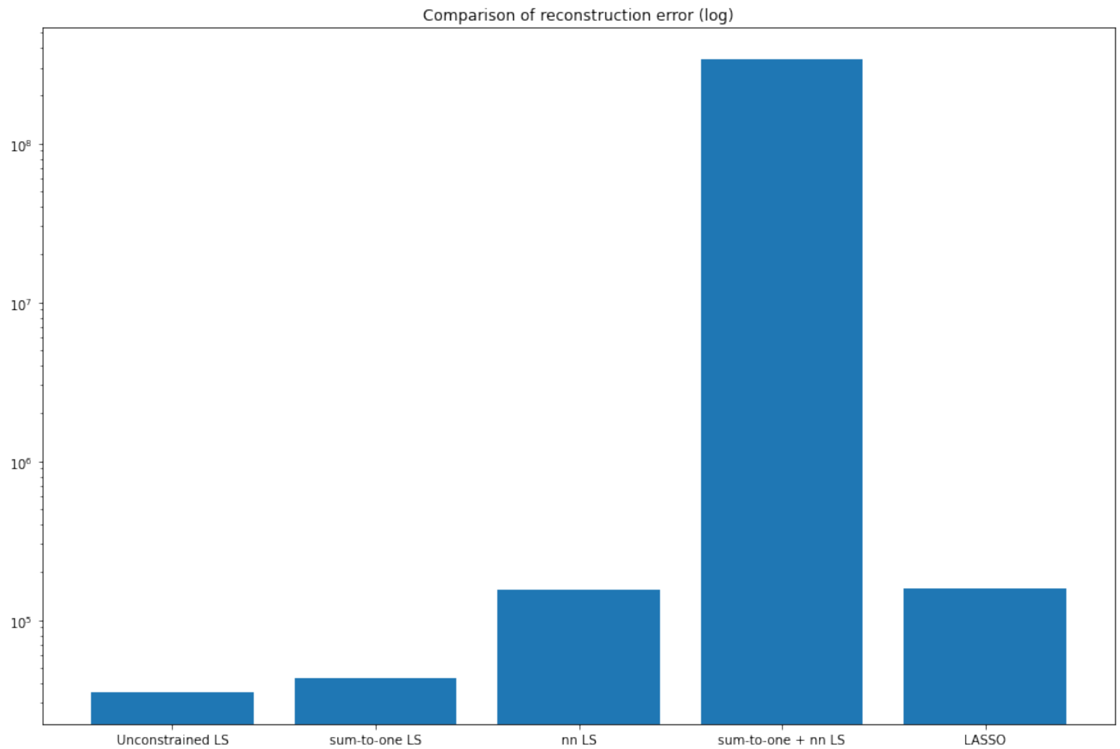
[43]:
```
errors = [error1, error2, error3, error4, error5]

errors
```

[43]:
```
[35058.88066277267,
 43082.576302782494,
 156104.18220644674,
 339088712.2633131,
 158097.38670329918]
```

[44]:
```
method_names = ['Unconstrained LS', 'sum-to-one LS', 'nn LS', 'sum-to-one + nn␣
↪LS', 'LASSO']
```

[45]:
```
fig = plt.figure(figsize=(15,10))

ax = fig.add_subplot(111)

ax.set_title('Comparison of reconstruction error (log)')

ax.bar(method_names, errors, align='center')
```

```
ax.set_yscale('log')

plt.show()
```



4 Classification

We again consider only the image pixels with non-zero class label.

Our goal is to assign each one of them to the most appropriate class, among the **7** known classes.

We will do so, with the following 4 classifiers:

- **Naive Bayes classifier**
- **minimum Euclidean distance classifier**
- **k-nearest neighbor classifier**
- **Bayesian classifier**

[46]:
```
training_set = (np.reshape(salinas_labels['training_set'], (120, 220))).T
test_set = (np.reshape(salinas_labels['test_set'], (120, 220))).T
operational_set = (np.reshape(salinas_labels['operational_set'], (120, 220))).T
```

[47]:
```
training_set.shape
```

[47]: (220, 120)

[48]:
```
test_set.shape
```

[48]: (220, 120)

[49]:
```
operational_set.shape
```

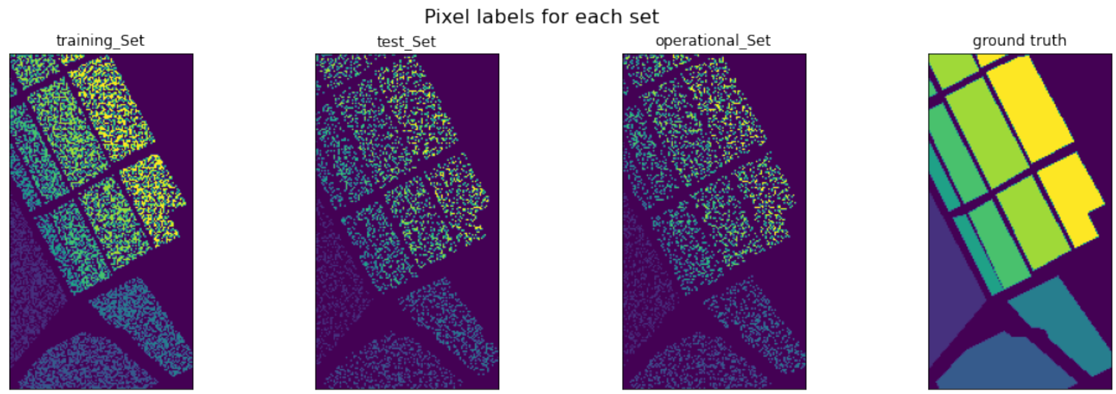
[49]: (220, 120)

[50]:
```
labels.shape
```

[50]: (220, 120)

Let's visualize the datasets, to get a better idea.

[51]:
```
fig, axs = plt.subplots(1, 4, figsize=(17, 5), sharey=True)

fig.suptitle('Pixel labels for each set', fontsize=16)

axs[0].imshow(training_set)
axs[0].set_title('training_Set')
axs[0].set_xticks([])

axs[1].imshow(test_set)
axs[1].set_title('test_Set')
axs[1].set_xticks([])
axs[1].set_yticks([])

axs[2].imshow(operational_set)
axs[2].set_title('operational_Set')
axs[2].set_xticks([])
axs[2].set_yticks([])

axs[3].imshow(labels)
axs[3].set_title('ground truth')
axs[3].set_xticks([])
axs[3].set_yticks([])
plt.show()
```



(i) We will train each classifier on the training set, performing a 10-fold cross-validation. We will report the estimated validation error by averaging over the 10 result sets, along with the respective standard deviation.

(ii) We will then use the whole training set to train the classifier and evaluate each performance on the test set:

- via the confusion matrix
- by computing the success rate

[52]:
```
X_train = hsi[training_set != 0]
y_train = training_set[training_set != 0]

print(X_train.shape)
print(y_train.shape)
```

```
(8465, 204)
(8465,)
```

[53]:
```
X_test = hsi[test_set != 0]
y_test = test_set[test_set != 0]

print(X_test.shape)
print(y_test.shape)
```

```
(4232, 204)
(4232,)
```

[54]:
```
X_op = hsi[operational_set != 0]
y_op = test_set[operational_set != 0]

print(X_op.shape)
print(y_op.shape)
```

```
(4232, 204)
(4232,)
```

4.0.1 Naive Bayes Classifier

The Naive Bayes classifier is a special case of the Bayes classifier that makes the assumption of all the features being statistically independent from each other.

[55]:
```
# TODO: add blah blah
```

(i) performing a 10-fold cross-validation.

[56]:
```
def error(predictions, gold):
    """Implements a mis-classification error.
    Args:
        predictions: The predictions made by a classifier, on a set of␣
↪datapoints.
        gold: The ground truth for this set of datapoints.

    Returns (float):
        The error.
    """
    return 1 - np.sum((predictions == gold)) / len(gold)
```

[57]:
```
nb_scores = cross_val_score(
    GaussianNB(),
    X=X_train,
    y=y_train,
    cv=10,
    scoring=make_scorer(error)
)
```

[58]:
```
print('Naive Bayes - Validation Error (mean): {}'.format(np.mean(nb_scores)))
print('Naive Bayes - Validation Error (stdev): {}'.format(np.std(nb_scores)))
```

```
Naive Bayes - Validation Error (mean): 0.026223969454143535
Naive Bayes - Validation Error (stdev): 0.016023209106526503
```

(ii) training on the whole training set, reporting on test set.

[59]:
```
model = GaussianNB()

model.fit(X_train, y_train)
```

[59]:
```
GaussianNB()
```

[60]:
```
y_pred = model.predict(X_test)
```

```
[61]: cm1 = confusion_matrix(y_test, y_pred)

print(cm1)
```

```
[[545   0   0   0   0   0   3]
 [  5 512   0   0   0   0   0]
 [  0   0 470   0  42   0   0]
 [  0   0   0 210   4   0   0]
 [  0   0  12   4 547   0   0]
 [  1   0   2   0   0 995   0]
 [  6   0   0   0   0   0 874]]
```

```
[62]: def success_rate(cmatrix):
    """Implements the success rate of a classified, via its confusion matrix.

    success rate: the sum of the diagonal elements of the confusion matrix,
                  divided by the sum of all matrix elements.

    Args:
        cmatrix (2-dim array): the confusion matrix of a classifier.

    Returns (float):
        The success rate.
    """
    return np.trace(cmatrix) / np.sum(cmatrix)
```

```
[63]: print('Naive Bayes - success rate: {}'.format(success_rate(cm1)))
```

```
Naive Bayes - success rate: 0.9813327032136105
```

4.0.2 minimum Euclidean distance classifier

```
[64]: # TODO: add blah blah
```

In order to implement a minimum Euclidean distance classifier from scratch, we will subclass `BaseEstimator` of scikit-learn library, to expose the familiar `fit/predict` API.

```
[65]: class MinEuclideanDistanceClassifier(BaseEstimator):
    """Implements a minimum Euclidean distance classifier.

    Attributes:

        fit: given X and y, fits the classifier on the data.
        predict: given X, returns the predictions.
    """
    def __init__(self):
        self.classes_num = None
        self.classes_mean = None
```

```
    def __str__(self):
        return "Bayes classifier"

    @staticmethod
    def euclidean_distance(arr1, arr2):
        """Returns the Euclidean distance between two arrays.
        """
        diff = arr1 - arr2

        return np.dot(diff, diff)

    def fit(self, X, y):

        self.classes_num = len(np.unique(y))

        m, n = X.shape

        self.classes_mean = np.zeros((self.classes_num, n))

        for i in range(self.classes_num):

            self.classes_mean[i] = np.mean(X[y == i + 1], axis=0)

        return self

    def predict(self, X):

        m, _ = X.shape

        y_pred = np.zeros(m)

        if self.classes_num is None:
            raise ValueError("fit() was not called before predict() - aborting.
↪")

        for i in range(m):

            dist = np.zeros(self.classes_num)

            for j in range(self.classes_num):

                dist[j] = self.euclidean_distance(X[i], self.classes_mean[j])

            y_pred[i] = np.argmin(dist) + 1.0

        return y_pred
```

(i) performing a 10-fold cross-validation.

[66]:
```
clf = MinEuclideanDistanceClassifier()

mineucl_scores = cross_val_score(
    MinEuclideanDistanceClassifier(),
    X=X_train,
    y=y_train,
    cv=10,
    scoring=make_scorer(error)
)
```

[67]:
```
print('{} - Validation Error (mean): {}'.format(str(clf), np.
 ↪mean(mineucl_scores)))
print('{} - Validation Error (stdev): {}'.format(str(clf), np.
 ↪std(mineucl_scores)))
```

```
Bayes classifier - Validation Error (mean): 0.05507548544299027
Bayes classifier - Validation Error (stdev): 0.07682360107147099
```

(ii) training on the whole training set, reporting on test set.

[68]:
```
model = MinEuclideanDistanceClassifier()

model.fit(X_train, y_train)
```

[68]:
```
MinEuclideanDistanceClassifier()
```

[69]:
```
y_pred = model.predict(X_test)
```

[70]:
```
cm2 = confusion_matrix(y_test, y_pred)

print(cm2)
```

```
[[536   0   4   0   1   0   7]
 [  2 484   0   0   0   0  31]
 [  0   0 417   0  95   0   0]
 [  0   0   0 212   2   0   0]
 [  0   0  16   4 543   0   0]
 [  0   0   6   0   0 992   0]
 [  5   0   0   0   0   0 875]]
```

[71]:
```
print('{} - success rate: {}'.format(str(model), success_rate(cm2)))
```

```
Bayes classifier - success rate: 0.9591209829867675
```

4.0.3 k-nearest neighbor classifier

[72]:
```
# TODO: add blah blah
```

(i) performing a 10-fold cross-validation.

We will try different values of `k` when performing the cross-validation.

[73]:
```
for k in range(10):

    clf = KNeighborsClassifier(n_neighbors=k + 1)  # default is: 5 neighbors

    knn_scores = cross_val_score(
        clf,
        X=X_train,
        y=y_train,
        cv=10,
        scoring=make_scorer(error),
    )

    print('{} - Validation Error (mean): {}'.format(str(clf), np.
↪mean(knn_scores)))
    print('{} - Validation Error (stdev): {}'.format(str(clf), np.
↪std(knn_scores)))
    print('----------------------------------------------')
```

```
KNeighborsClassifier(n_neighbors=1) - Validation Error (mean):
0.00850784719256672
KNeighborsClassifier(n_neighbors=1) - Validation Error (stdev):
0.012978550223365852
----------------------------------------------
KNeighborsClassifier(n_neighbors=2) - Validation Error (mean):
0.008508824079423693
KNeighborsClassifier(n_neighbors=2) - Validation Error (stdev):
0.01385401606523653
----------------------------------------------
KNeighborsClassifier(n_neighbors=3) - Validation Error (mean):
0.008862178011114174
KNeighborsClassifier(n_neighbors=3) - Validation Error (stdev):
0.012961687747303887
----------------------------------------------
KNeighborsClassifier(n_neighbors=4) - Validation Error (mean):
0.009926147353613501
KNeighborsClassifier(n_neighbors=4) - Validation Error (stdev):
0.014727956905239871
----------------------------------------------
KNeighborsClassifier() - Validation Error (mean): 0.010162135307202980
KNeighborsClassifier() - Validation Error (stdev): 0.014536383096264979
```

```
----------------------------------------------
KNeighborsClassifier(n_neighbors=6) - Validation Error (mean):
0.010871215610093743
KNeighborsClassifier(n_neighbors=6) - Validation Error (stdev):
0.014657312744675029
----------------------------------------------
KNeighborsClassifier(n_neighbors=7) - Validation Error (mean):
0.010161995751937714
KNeighborsClassifier(n_neighbors=7) - Validation Error (stdev):
0.014136851235651886
----------------------------------------------
KNeighborsClassifier(n_neighbors=8) - Validation Error (mean):
0.010989000253990577
KNeighborsClassifier(n_neighbors=8) - Validation Error (stdev):
0.014065848184995604
----------------------------------------------
KNeighborsClassifier(n_neighbors=9) - Validation Error (mean):
0.011815865200778153
KNeighborsClassifier(n_neighbors=9) - Validation Error (stdev):
0.014223096096097213
----------------------------------------------
KNeighborsClassifier(n_neighbors=10) - Validation Error (mean):
0.012288538884283561
KNeighborsClassifier(n_neighbors=10) - Validation Error (stdev):
0.01443978405725648
----------------------------------------------
```

(ii) training on the whole training set, reporting on test set.

[74]:
```
model = KNeighborsClassifier(n_neighbors=1)

model.fit(X_train, y_train)
```

[74]:
```
KNeighborsClassifier(n_neighbors=1)
```

[75]:
```
y_pred = model.predict(X_test)
```

[76]:
```
cm3 = confusion_matrix(y_test, y_pred)

print(cm3)
```

```
[[548   0   0   0   0   0   0]
 [  0 516   0   0   0   0   1]
 [  0   0 510   0   2   0   0]
 [  0   0   0 214   0   0   0]
 [  0   0   4   1 555   3   0]
 [  0   0   0   0   0 998   0]
 [  0   0   0   0   0   0 880]]
```

[77]:
```
print('{} - success rate: {}'.format(str(model), success_rate(cm3)))
```

```
KNeighborsClassifier(n_neighbors=1) - success rate: 0.9974007561436673
```

4.0.4 Bayesian classifier

[78]:
```
# TODO
```

[79]:
```
class BayesClassifier(BaseEstimator):
    """Implements a Bayes classifier.

    Attributes:

        fit: given X and y, fits the classifier on the data.
        predict: given X, returns the predictions.
    """
    def __init__(self):
        self.classes_num = None
        self.classes_mean = None
        self.classes_cov = None
        self.priors = None

    def __str__(self):
        return "minimum Euclidean distance classifier"

    @staticmethod
    def euclidean_distance(arr1, arr2):
        """Returns the Euclidean distance between two arrays.
        """
        diff = arr1 - arr2

        return np.dot(diff, diff)

    def fit(self, X, y):

        self.classes_num = len(np.unique(y))

        m, n = X.shape

        self.classes_mean = np.zeros((self.classes_num, n))
        self.classes_cov = np.zeros((self.classes_num, n))

        for i in range(self.classes_num):

            self.classes_mean[i] = np.mean(X[y == i + 1], axis=0)
            self.classes_cov[i] = np.cov(X[y == i + 1], axis=0)
```

```
        return self

    def predict(self, X):

        m, _ = X.shape

        y_pred = np.zeros(m)

        if self.classes_num is None:
            raise ValueError("fit() was not called before predict() - aborting.
↪")

        for i in range(m):

            dist = np.zeros(self.classes_num)

            for j in range(self.classes_num):

                dist[j] = self.euclidean_distance(X[i], self.classes_mean[j])

            y_pred[i] = np.argmin(dist) + 1.0

        return y_pred
```

[80]:
```
def means(X, y):

    for i in range(7):

        temp = X[y]
        yield np.mean(temp[y == i + 1], axis=0)
```

[81]:
```
def covs(X, y):

    covs = np.empty((204, 204, 7))

    for i in range(7):

        covs[:, :, i] = np.cov(np.array(X[y == i + 1]).T)

    return covs
```

(i) performing a 10-fold cross-validation.

[82]:
```
def cross_validate_bayes(X, y):
    """Implements a custom-made cross validation scheme for Bayes classifier.

    Args:
```

```
        X: the design matrix.
        y: the class labels.

    Returns:
        Iterator, containing cross-validation errors.
    """
    classes_num = len(np.unique(y))

    for train_idx, test_idx in KFold(n_splits=10).split(X, y):

        classes_mean = [*means(X[train_idx], y[train_idx])]
        classes_cov = covs(X[train_idx], y[train_idx])
        priors = np.zeros((classes_num, 1))

        scores = []
        for i in range(classes_num):

            priors[i] = np.sum(y[train_idx] == i + 1)

            d = multivariate_normal(classes_mean[i], classes_cov[:, :, i])

            scores.append(priors[i] * np.array(d.pdf(X_train[test_idx])) /␣
↪len(y[train_idx]))

        y_pred = np.argmax(np.array(scores), axis=0) + 1.0

        yield error(y_pred, y_train[test_idx])
```

[83]:
```
bayes_scores = [*cross_validate_bayes(X_train, y_train)]
```

[84]:
```
print('Bayes classifier - Validation Error (mean): {}'.format(np.
 ↪mean(bayes_scores)))
print('Bayes classifier - Validation Error (stdev): {}'.format(np.
 ↪std(bayes_scores)))
```

```
Bayes classifier - Validation Error (mean): 0.7937029873200085
Bayes classifier - Validation Error (stdev): 0.1407160567615001
```

(ii) training on the whole training set, reporting on test set.

[85]:
```
classes_num = len(np.unique(y_train))
classes_mean = [*means(X_train, y_train)]
classes_cov = covs(X_train, y_train)
priors = np.zeros((classes_num, 1))

scores = []
for i in range(classes_num):
```

```
    idx = (y_train == i + 1)
    priors[i] = np.sum(y_train == i + 1)

    d = multivariate_normal(classes_mean[i], classes_cov[:, :, i])
    scores.append(priors[i] * np.array(d.pdf(X_test)) / len(y_train))

y_pred = np.argmax(np.array(scores), axis=0) + 1.0

bayes_error = error(y_pred, y_test)

cm4 = confusion_matrix(y_test, y_pred)
```

[86]:
```
bayes_error
```

[86]:
```
0.791351606805293
```

[87]:
```
cm4
```

[87]:
```
array([[548,   0,   0,   0,   0,   0,   0],
       [517,   0,   0,   0,   0,   0,   0],
       [512,   0,   0,   0,   0,   0,   0],
       [214,   0,   0,   0,   0,   0,   0],
       [563,   0,   0,   0,   0,   0,   0],
       [663,   0,   0,   0,   0, 335,   0],
       [880,   0,   0,   0,   0,   0,   0]])
```

[88]:
```
print('Bayes classifier - success rate: {}'.format(success_rate(cm4)))
```

```
Bayes classifier - success rate: 0.208648393194707
```

We will also use the `QuadraticDiscriminantAnalysis` model, which essentially does the same job.

[89]:
```
bayes_scores = cross_val_score(
    QuadraticDiscriminantAnalysis(),
    X=X_train,
    y=y_train,
    cv=10,
    scoring=make_scorer(error)
)
```

[90]:
```
print('Bayes classifier - Validation Error (mean): {}'.format(np.
 ↪mean(bayes_scores)))
print('Bayes classifier - Validation Error (stdev): {}'.format(np.
 ↪std(bayes_scores)))
```

```
Bayes classifier - Validation Error (mean): 0.03426123629218406
Bayes classifier - Validation Error (stdev): 0.005850919532443715
```

[91]:
```python
model = QuadraticDiscriminantAnalysis()

model.fit(X_train, y_train)
```

[91]:
```
QuadraticDiscriminantAnalysis()
```

[92]:
```python
y_pred = model.predict(X_test)
```

[93]:
```python
cm4 = confusion_matrix(y_test, y_pred)

print(cm4)
```

```
[[548   0   0   0   0   0   0]
 [  0 517   0   0   0   0   0]
 [  0   0 512   0   0   0   0]
 [  0   0   0 125  89   0   0]
 [  0   0   3   0 558   2   0]
 [  0   0   0   0   0 998   0]
 [  0   0   0   0   0   0 880]]
```

[94]:
```python
print('Bayes classifier - success rate: {}'.format(success_rate(cm4)))
```

```
Bayes classifier - success rate: 0.9777882797731569
```

4.0.5 Classification - Comparison and Remarks

[95]:
```python
# TODO
```

5 Combination

[96]:
```python
# TODO
```