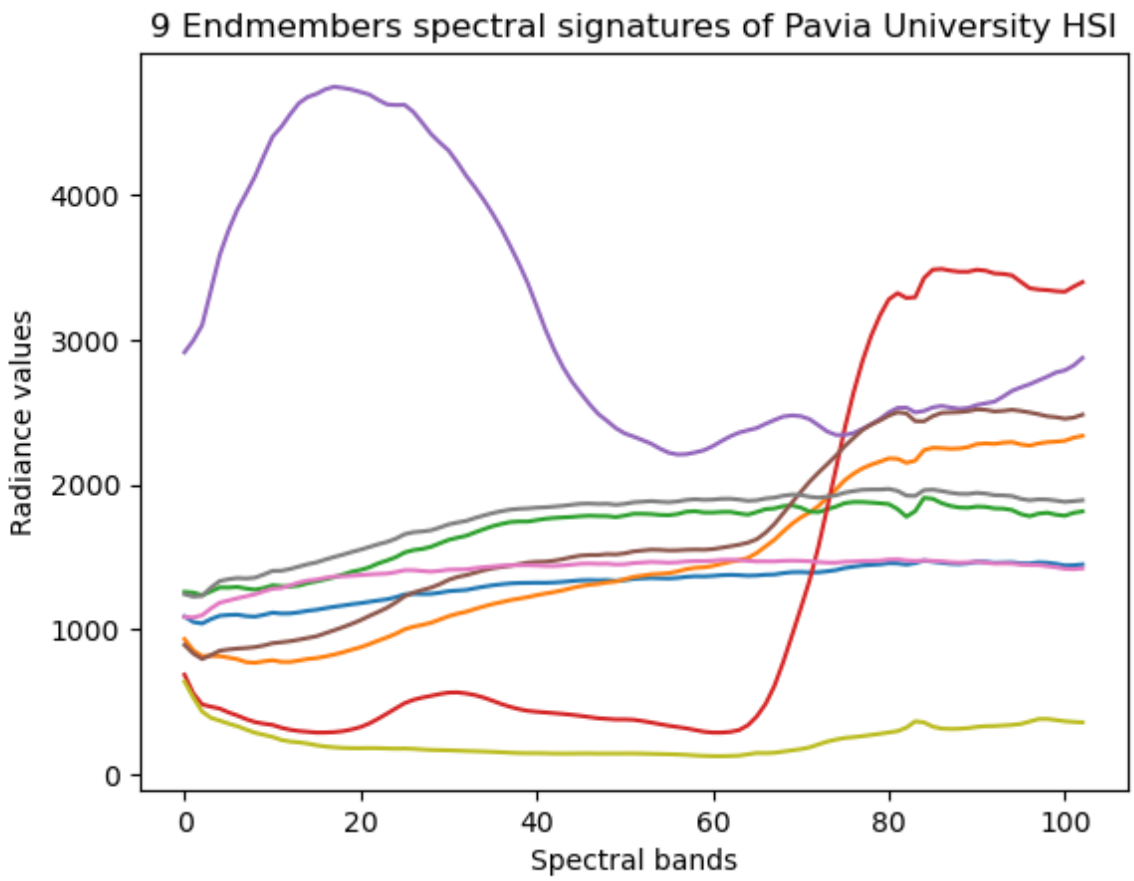
Project - Machine Learning and Computational Statistics

Melina Moniaki - f3352321

In [253…
```
import scipy.io as sio
import pandas as pd
import numpy as np
import scipy.optimize
import matplotlib.pyplot as plt
from scipy.optimize import nnls
from scipy.optimize import minimize
from sklearn.linear_model import Lasso
```

In [254…
```
Pavia = sio.loadmat("C:\\Users\\melin\\Desktop\\Data Science\\Machine Learning a
HSI = Pavia['X'] #Pavia HSI : 300x200x103
ends = sio.loadmat("C:\\Users\\melin\\Desktop\\Data Science\\Machine Learning an
endmembers = ends['endmembers']
fig = plt.figure()
plt.plot(endmembers)
plt.ylabel('Radiance values')
plt.xlabel('Spectral bands')
plt.title('9 Endmembers spectral signatures of Pavia University HSI')
plt.show()
```

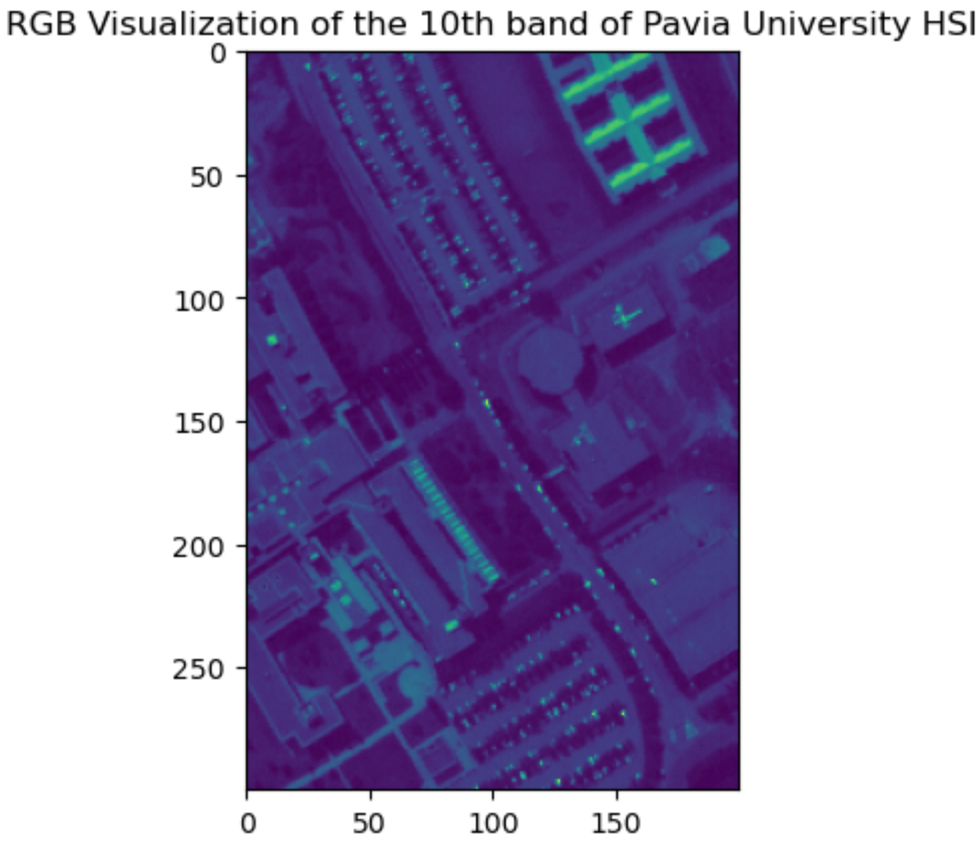


In [255…
```
endmembers.shape
X=np.array(endmembers)
```

In [256…
```
X[:,1].shape
```

Out[256]:
```
(103,)
```

In [257…
```
#Perform unmixing for the pixels corresponding to nonzero labels
ground_truth= sio.loadmat("C:\\Users\\melin\\Desktop\\Data Science\\Machine Lear
labels=ground_truth['y']
fig = plt.figure()
plt.imshow(HSI[:,:,10])
plt.title('RGB Visualization of the 10th band of Pavia University HSI')
plt.show()
# For the non-negative least squares  unmixing algorithm  you can use the nnls f
#https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.optimize.nnls
```



In [ ]:
```
print (HSI)
HSI.shape
```

PART 1 - SPECTRAL UNMIXING

(a) Least squares (as it was presented in the class)

Below, we take only the pixels with nonzero class label and put them on a new array with the name Y.(it consists of 12829 pixels).

In [94]:
```
# In this project we take into consideration only the pixels with nonzero class
y=[]
for i in range (300):
    for j in range (200):
        if labels[i,j] != 0:
            y.append(HSI[i,j])
Y=np.array(y).T
```

In [95]:
```
print(Y)
```

```
[[1351 1377 1462 ...  855 1023 1076]
 [1214 1379 1231 ...  827  887  972]
 [1293 1425 1272 ...  968  774  739]
 ...
 [1477 1461 1556 ... 1135 1122 1210]
 [1512 1466 1572 ... 1129 1106 1166]
 [1499 1439 1587 ... 1136 1113 1157]]
```

In [105…
```
Y.shape
```

Out[105]:
```
(103, 12829)
```

In [106…
```
X.shape
```

Out[106]:
```
(103, 9)
```

we perform least squares to calculate the values of theta and store them in a data frame called theta_est. We also calculate the reconstruction error for this method and then and derive the 9 abundance maps.

In [96]:
```
# Perform least squares unmixing
XTX_inv = np.linalg.inv(np.dot(X.T, X))
theta_est = np.dot(XTX_inv, X.T).dot(Y)

theta_est_df=pd.DataFrame(theta_est)
theta_est_df
```

Out[96]:

	0	1	2	3	4	5	6	7	
0	2.403077	1.712854	-1.874694	-2.755703	-1.815342	-2.160288	-2.639534	2.683418	-1.195
1	-0.197988	-0.068802	0.866352	0.462381	0.239584	0.452331	0.235683	0.076454	0.255
2	1.807542	2.008817	0.172044	0.421226	1.140866	-0.000725	0.010877	0.834710	0.362
3	0.065495	0.057381	0.065997	-0.025464	-0.012214	-0.037040	0.019801	0.085247	-0.016
4	0.000263	-0.034414	-0.014636	-0.004141	0.025568	0.015164	0.028000	-0.006693	-0.002
5	-0.054835	-0.071719	-0.950451	-0.352667	-0.231273	-0.430255	-0.581631	-0.129918	-0.174
6	0.347240	1.080738	1.055433	1.130761	1.022705	0.812925	0.171287	0.572509	0.916
7	-2.769383	-3.106617	1.203806	1.363818	0.109875	1.568176	2.834019	-2.112276	0.741
8	-0.482455	-0.158961	0.983325	1.416612	1.053397	1.535120	1.606377	-1.555396	0.488

9 rows × 12829 columns

In [97]:
```
# Initialize a list to store reconstruction errors for each pixel
reconstruction_errors = []

# Compute reconstruction errors for each pixel
for i in range(12829):
    # Calculate the estimated spectral signature using the estimated abundance
    y_est = np.dot(X, theta_est_df.iloc[:, i])

    # Compute the reconstruction error for the ith pixel
    error_i = np.square(np.linalg.norm(Y[:, i] - y_est, ord=2))  # Euclidean nor

    # Append the error to the list
    reconstruction_errors.append(error_i)

# Calculate the average reconstruction error
average_error = np.mean(reconstruction_errors)

print("Average Reconstruction Error:", average_error)
```

```
Average Reconstruction Error: 118783.18062626586
```

In [98]:
```
labels.shape
```

Out[98]:
```
(300, 200)
```

Below we convert the data frame with the thetas (9x12829) to a 3D array with dimensions 300x200x9 and we create 9 different 2D arrays, each corresponding to a different endmember/material. We do this to derive the corresponding 9 abundance maps (one for each endmember/material).

In [155…
```
# Create a 200x300x9 array filled with zeros
result_array = np.zeros((300, 200, 9))

# Iterate over the pixels and fill in the values from the theta_est_df Data Fram
index = 0
for i in range(300):
    for j in range(200):
        if labels[i,j] != 0:
            result_array[i,j,:] = theta_est_df.iloc[:,index]
            index += 1
```

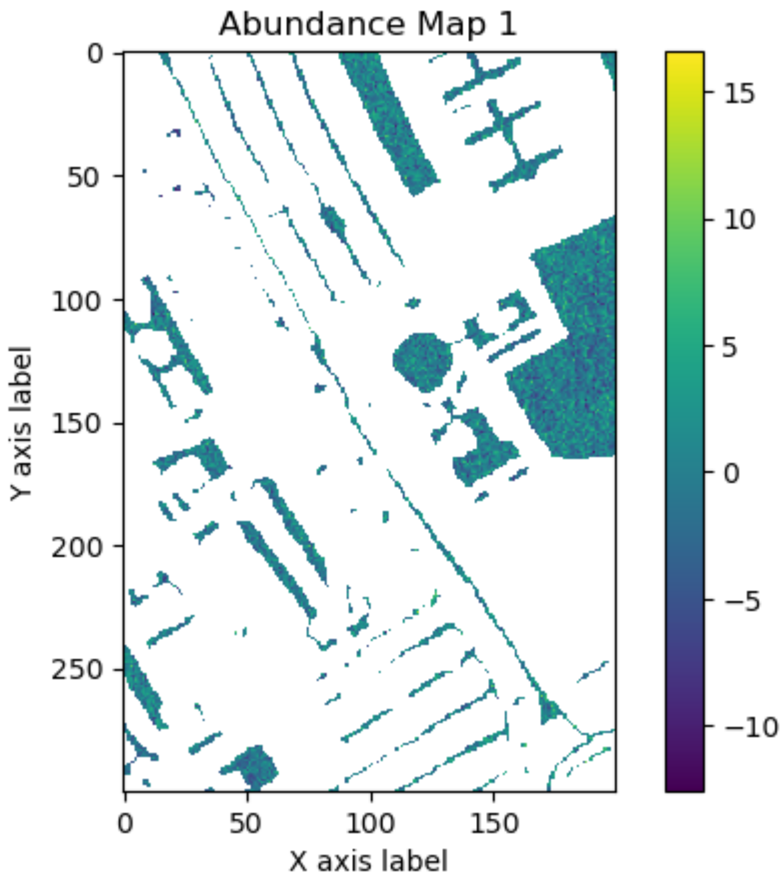
In [156…
```
result_array.shape
```

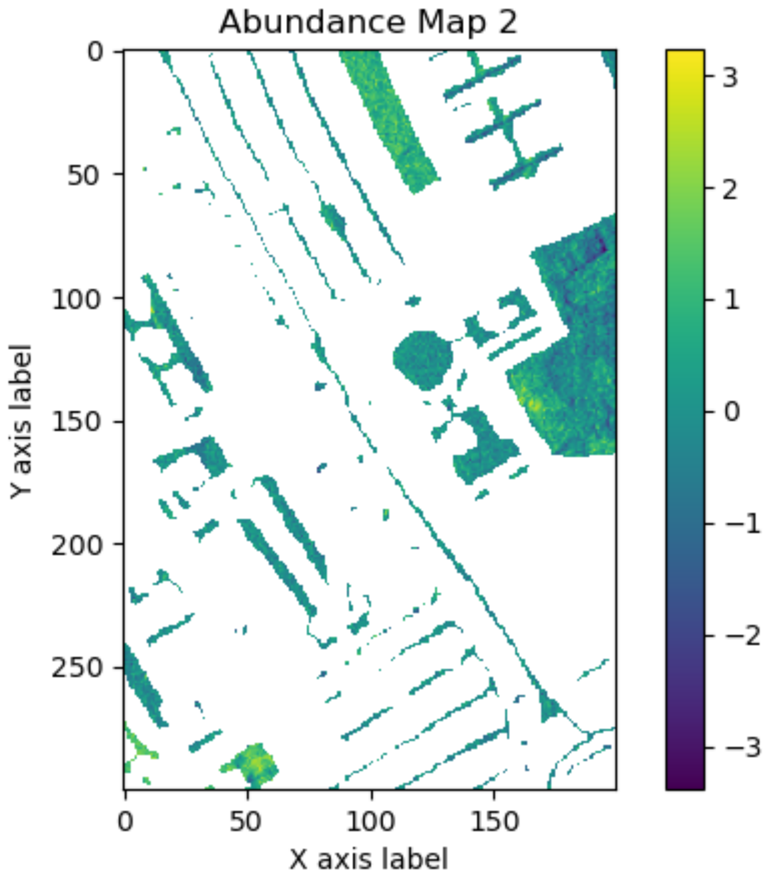
Out[156]:
```
(300, 200, 9)
```

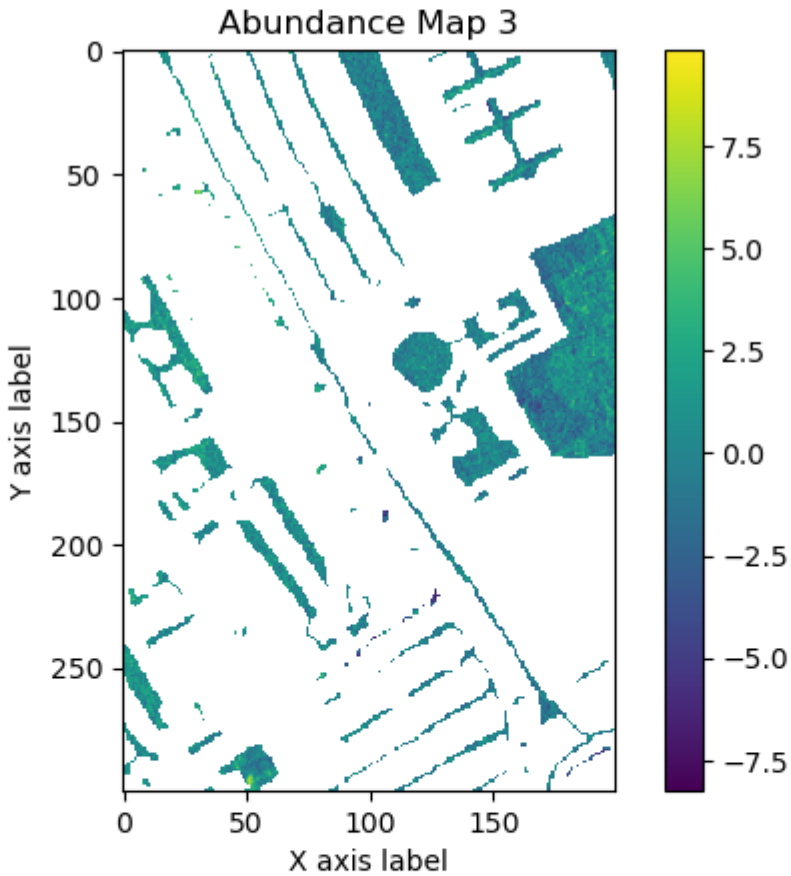
In [157…
```
result_arrays = []

for k in range(9):
    result_arrays.append(result_array[:, :, k])

# Now result_arrays contains 9 2D arrays, each corresponding to a different k va
df_theta = []
for i in range(9):
    df_a = pd.DataFrame(result_arrays[i])
    df_theta.append(df_a)
```

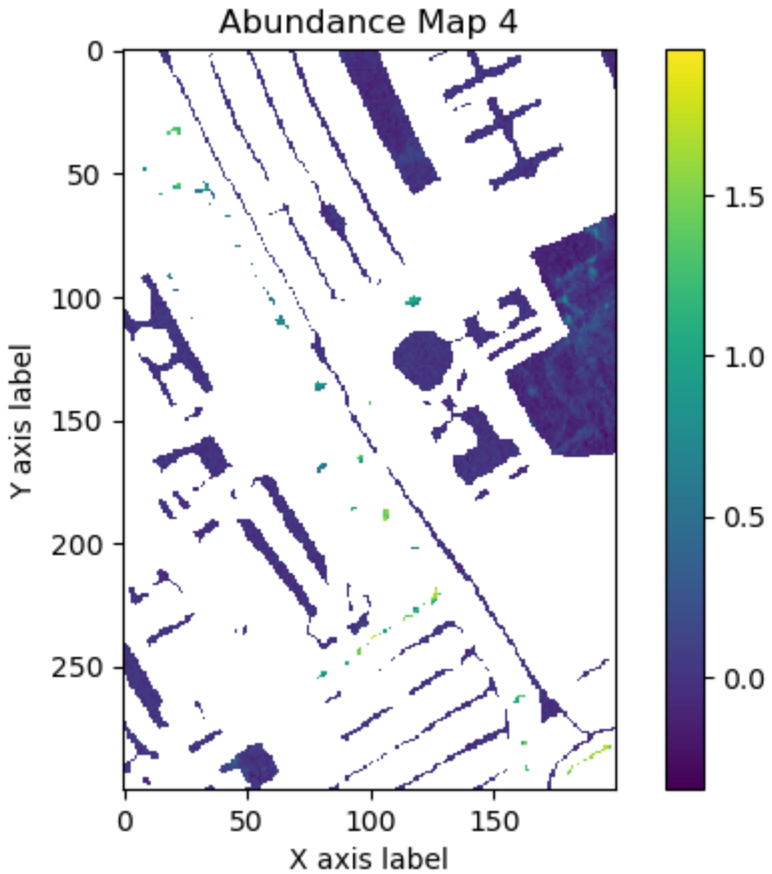
In [159…

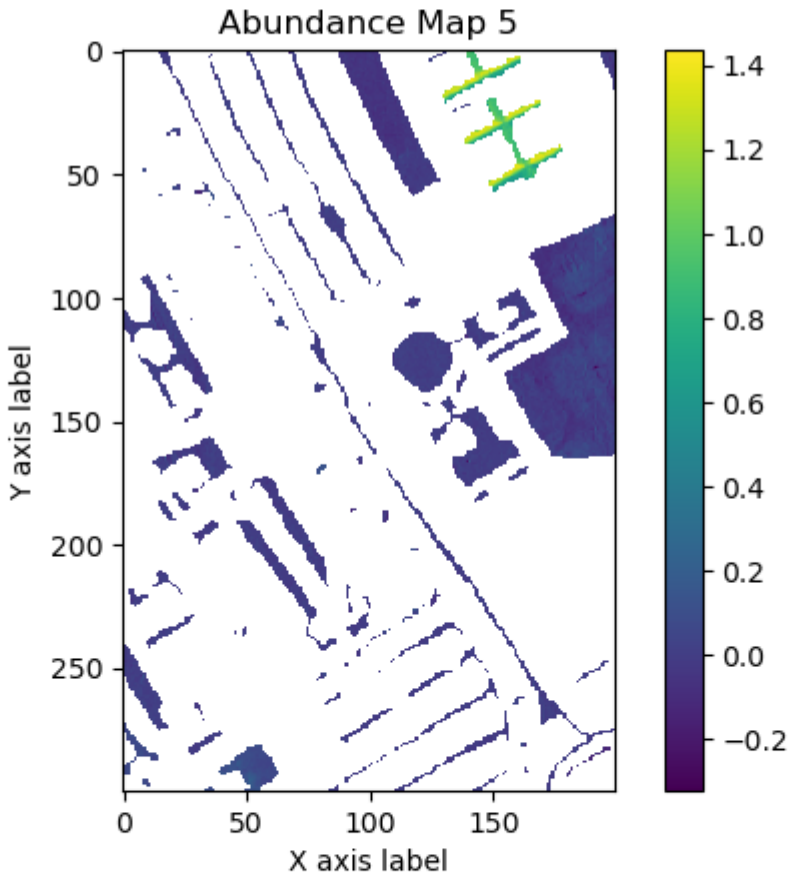
```
for i in range(9):
    # Create the heatmap using Matplotlib's imshow function
    fig, ax = plt.subplots()

    mask = df_theta[i] == 0

    # Set the colormap (cmap) to 'viridis' and set_bad to white
    cmap = plt.get_cmap('viridis')
    cmap.set_bad(color='white')

    # Apply the mask to the data
    im = ax.imshow(np.ma.masked_array(df_theta[i], mask))

    # Add a color bar
    cbar = ax.figure.colorbar(im, ax=ax)

    # Set axis labels
    ax.set_xlabel('X axis label')
    ax.set_ylabel('Y axis label')

    # Add title
    ax.set_title('Abundance Map {}'.format(i + 1))

    # Show the plot
    plt.show()
```

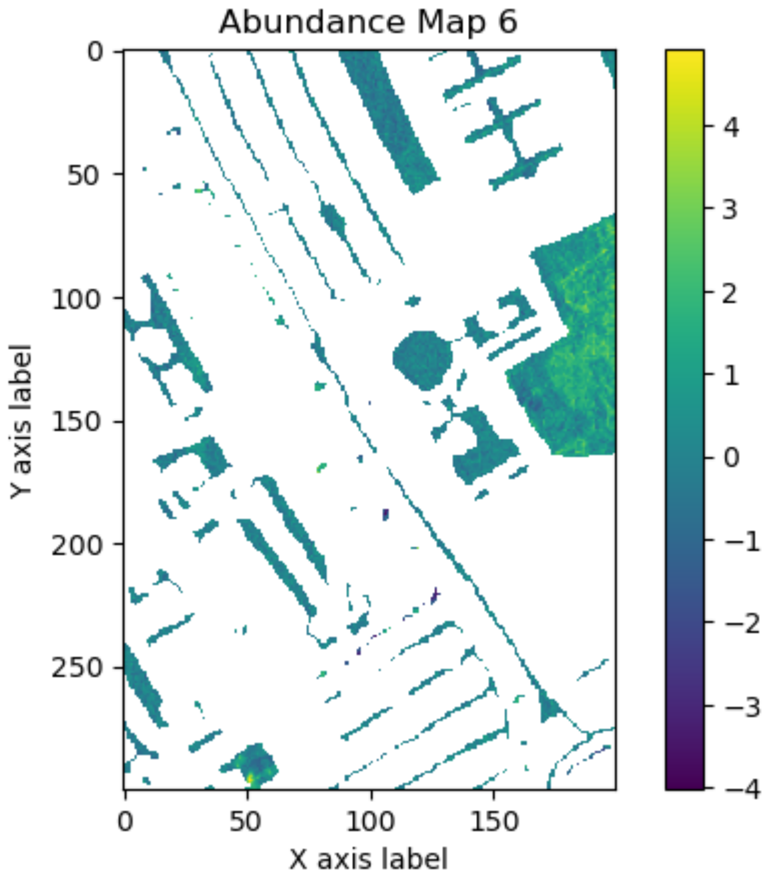


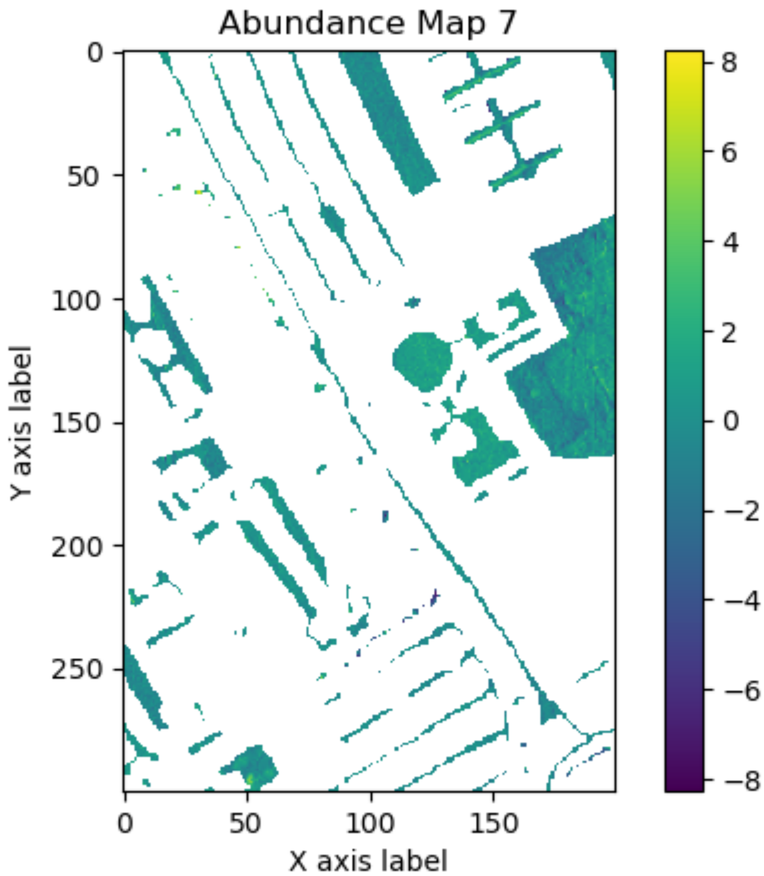

Abundance Map 2
Y axis label
X axis label
0
50
100
150
200
250
3
2
1
0
−1
−2
−3

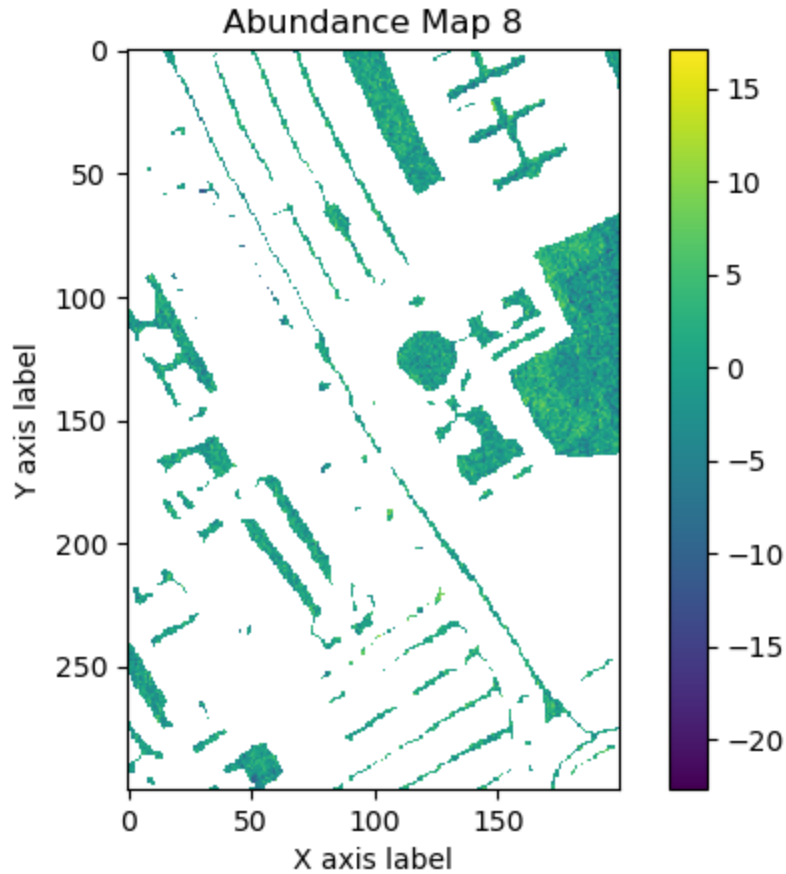

Abundance Map 3
Y axis label
X axis label
0
50
100
150
200
250
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5

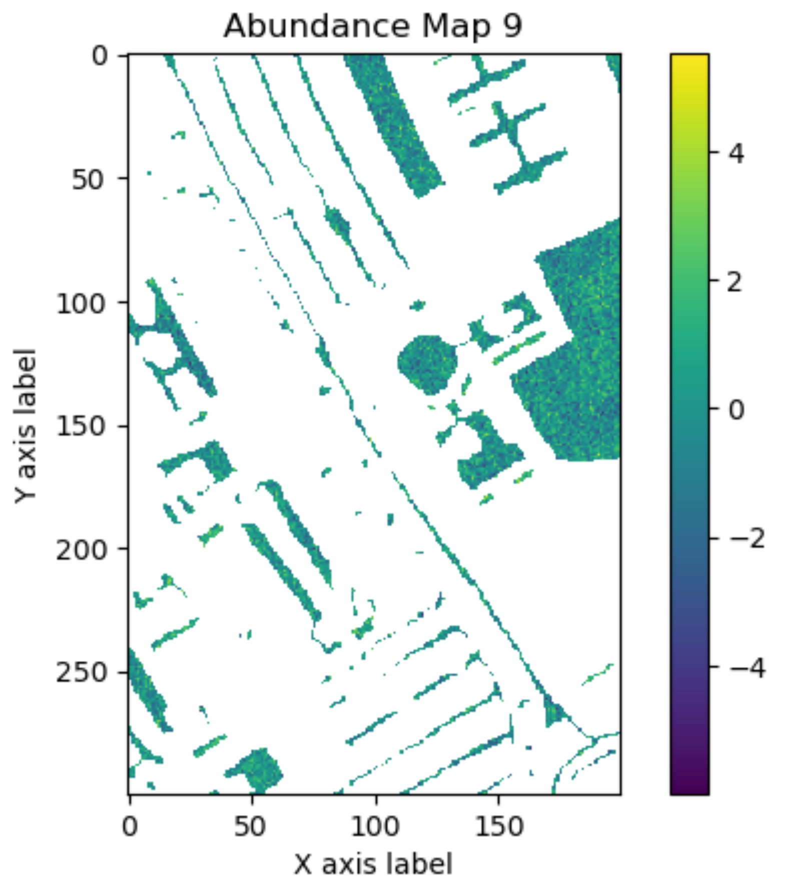

Abundance Map 4
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
1.5
1.0
0.5
0.0


Abundance Map 5
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2


Abundance Map 6
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
4
3
2
1
0
−1
−2
−3
−4


Abundance Map 7
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
8
6
4
2
0
−2
−4
−6
−8





(b) Least squares imposing the sum-to-one constraint

we perform least squares to calculate the values of theta, this time imposing the sum-to-one constraint, and store them in a data frame called theta_est_constrained. We also calculate the reconstruction error for this method and derive the 9 abundance maps.

In [17]:
```
import numpy as np
from scipy.optimize import minimize
from sklearn.metrics import mean_squared_error

# Define the sum-to-one constraint function
def constraint(theta):
    return np.sum(theta) - 1.0

# Define the objective function for minimization (Euclidean norm)
def objective(theta):
    y_est = np.dot(X, theta)
    return np.linalg.norm(Y[:, i] - y_est, ord=2)  # Euclidean norm

# Initialize an array to store the results
theta_est_constrained = np.zeros((9, 12829))

# Iterate over all pixels
for i in range(12829):
    # Initialize the optimization with an equal distribution
    initial_guess = np.ones(9) / 9.0

    # Define the optimization problem with the sum-to-one constraint
    constraint_definition = {'type': 'eq', 'fun': constraint}
    optimization_result = minimize(objective, initial_guess, constraints=constra

    # Store the optimized abundance vector
    theta_est_constrained[:, i] = optimization_result.x

# Calculate the reconstruction error
reconstruction_error_constrained = 0
for i in range(12829):
    y_est_constrained = np.dot(X, theta_est_constrained[:, i])
    reconstruction_error_constrained += np.square(np.linalg.norm(Y[:, i] - y_est

reconstruction_error_constrained /= 12829

print("The parameters θ1, θ2, ..., θ9 with sum-to-one constraint:\n", theta_est_
print("The Reconstruction Error with sum-to-one constraint is:", reconstruction_
```

```
The parameters θ1, θ2, ..., θ9 with sum-to-one constraint:
 [[ 3.17676754  4.439104    1.42306923 ... -0.12403317 -1.79073334
   0.09543534]
 [-0.26290805 -0.29743245  0.58979133 ... -0.03091852  0.38258237
   0.21608521]
 [ 2.0524747   2.87192217  1.21610544 ... -0.72086478 -0.39588229
  -1.58107896]
 ...
 [ 0.15858704  0.41568425  0.2509422  ... -0.12590702  0.46259366
  -1.06768575]
 [-3.37209245 -5.23015282 -1.36489577 ...  1.49014471  1.66348136
   2.8189751 ]
 [-0.80690493 -1.30228891 -0.39968396 ...  0.59137491  0.91230682
   0.80035702]]
The Reconstruction Error with sum-to-one constraint is: 160049.93078108568
```

In [18]:
```
theta_est_constrained_df=pd.DataFrame(theta_est_constrained)
theta_est_constrained_df
```

Out[18]:

	0	1	2	3	4	5	6	7	
0	3.176768	4.439104	1.423069	1.514912	1.651460	2.751531	1.813712	-0.905289	1.243
1	-0.262908	-0.297432	0.589791	0.104253	-0.051184	0.040400	-0.137784	0.377406	0.050
2	2.052475	2.871922	1.216105	1.773249	2.238445	1.554338	1.420769	-0.301461	1.134
3	0.075291	0.091914	0.107772	0.028641	0.031701	0.025178	0.076213	0.039784	0.014
4	0.003024	-0.024666	-0.002843	0.011131	0.037963	0.032727	0.043923	-0.019528	0.006
5	-0.024239	0.035916	-0.820257	-0.184101	-0.094376	-0.236322	-0.405815	-0.271581	-0.077
6	0.158587	0.415684	0.250942	0.088911	0.177011	-0.385291	-0.915055	1.448012	0.321
7	-3.372092	-5.230153	-1.364896	-1.962583	-2.590520	-2.257770	-0.634742	0.683013	-1.158
8	-0.806905	-1.302289	-0.399684	-0.374414	-0.400499	-0.524792	-0.261221	-0.050355	-0.534

9 rows × 12829 columns

In [19]:

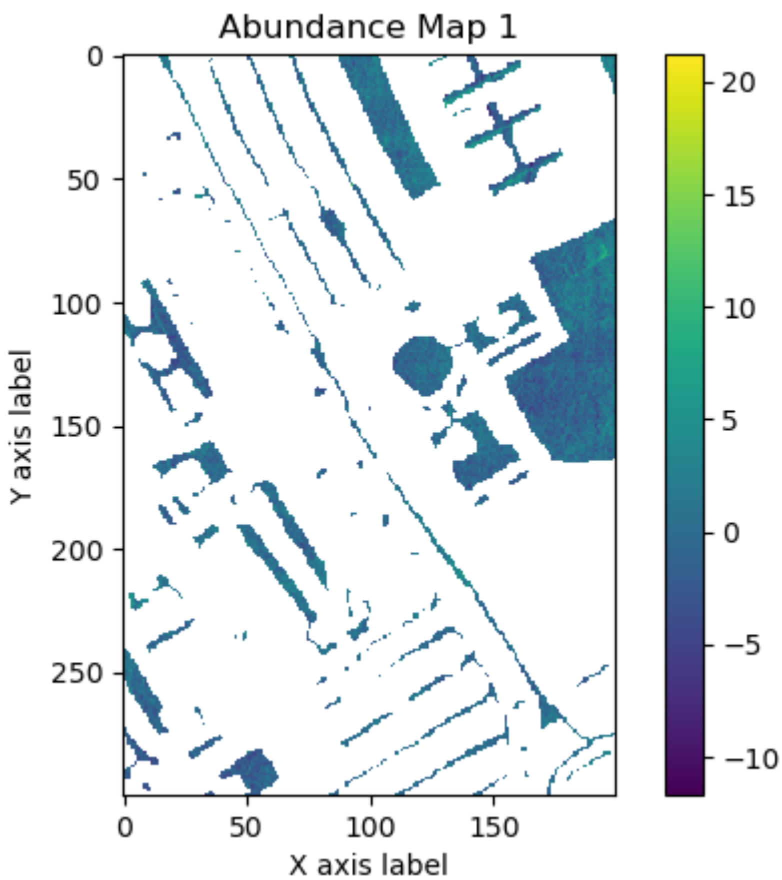
```
# Create a 200x300x9 array filled with zeros
result_array_b = np.zeros((300, 200, 9))

# Iterate over the pixels and fill in the values from the theta_est_constrained
index = 0
for i in range(300):
    for j in range(200):
        if labels[i,j] != 0:
            result_array_b[i,j,:] = theta_est_constrained_df.iloc[:,index]
            index += 1
```

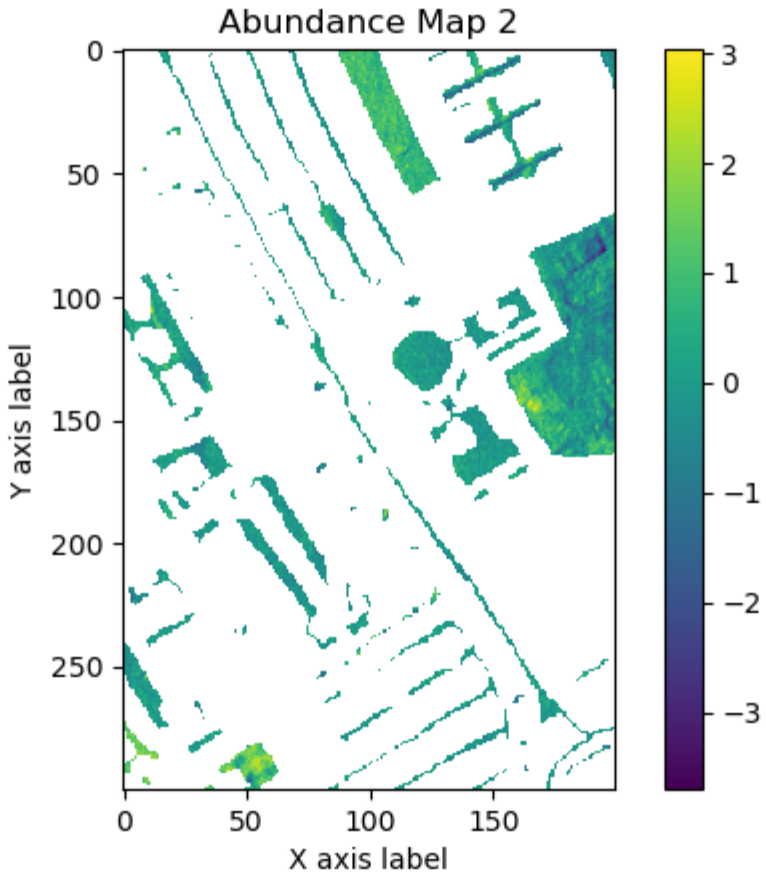
In [20]:

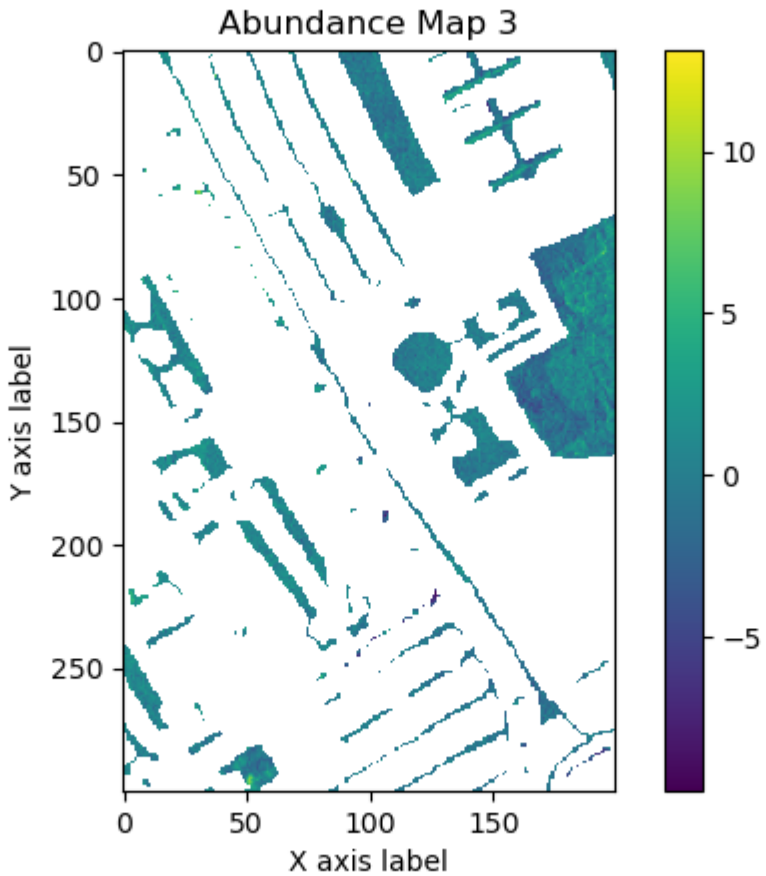
```
result_arrays_b = []

for k in range(9):
    result_arrays_b.append(result_array_b[:, :, k])

# Now result_arrays contains 9 2D arrays, each corresponding to a different k va
df_theta_b = []
for i in range(9):
    df_b = pd.DataFrame(result_arrays_b[i])
    df_theta_b.append(df_b)
```

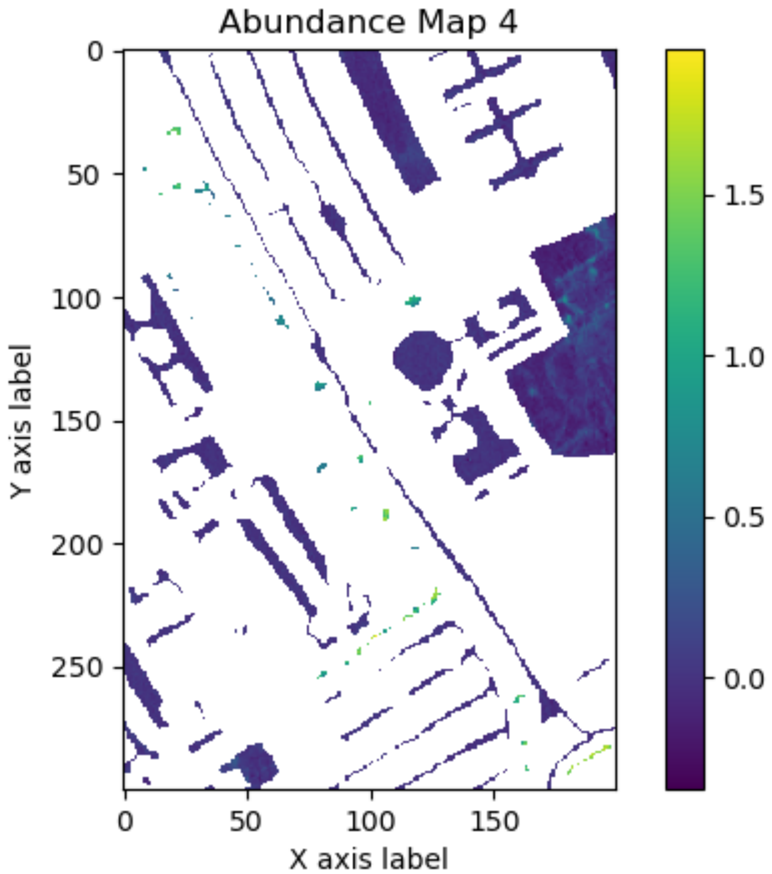
In [160…

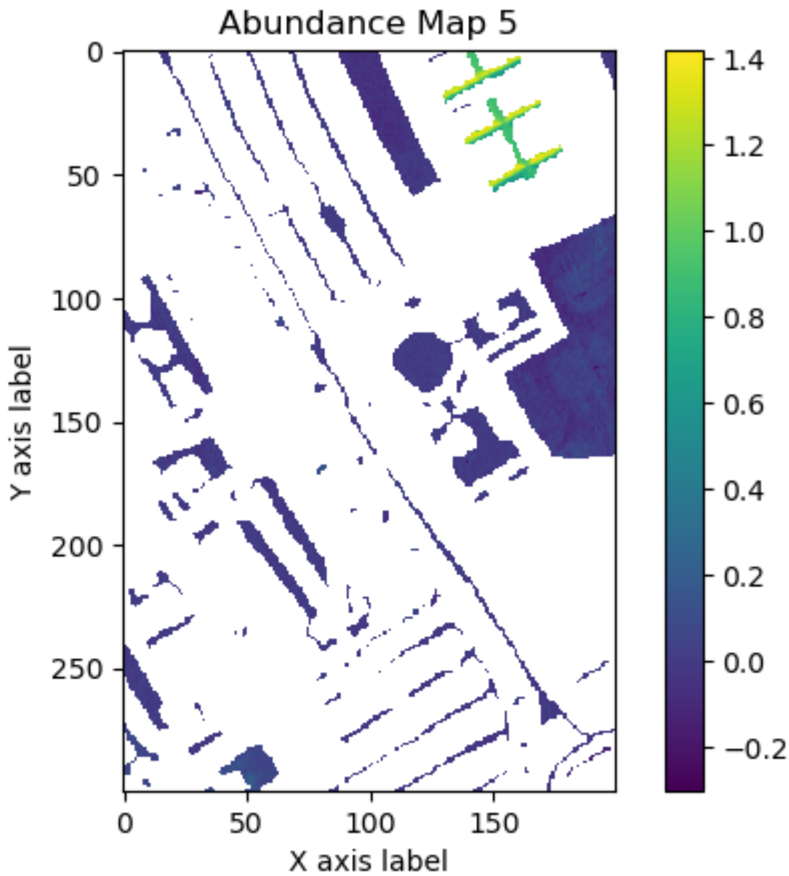
```python
for i in range(9):
    # Create the heatmap using Matplotlib's imshow function
    fig, ax = plt.subplots()


    mask = df_theta[i] == 0

    # Set the colormap (cmap) to 'viridis' and set_bad to white
    cmap = plt.get_cmap('viridis')
    cmap.set_bad(color='white')

    # Apply the mask to the data
    im = ax.imshow(np.ma.masked_array(df_theta_b[i], mask))

    # Add a color bar
    cbar = ax.figure.colorbar(im, ax=ax)

    # Set axis labels
    ax.set_xlabel('X axis label')
    ax.set_ylabel('Y axis label')

    # Add title
    ax.set_title('Abundance Map {}'.format(i + 1))

    # Show the plot
    plt.show()
```

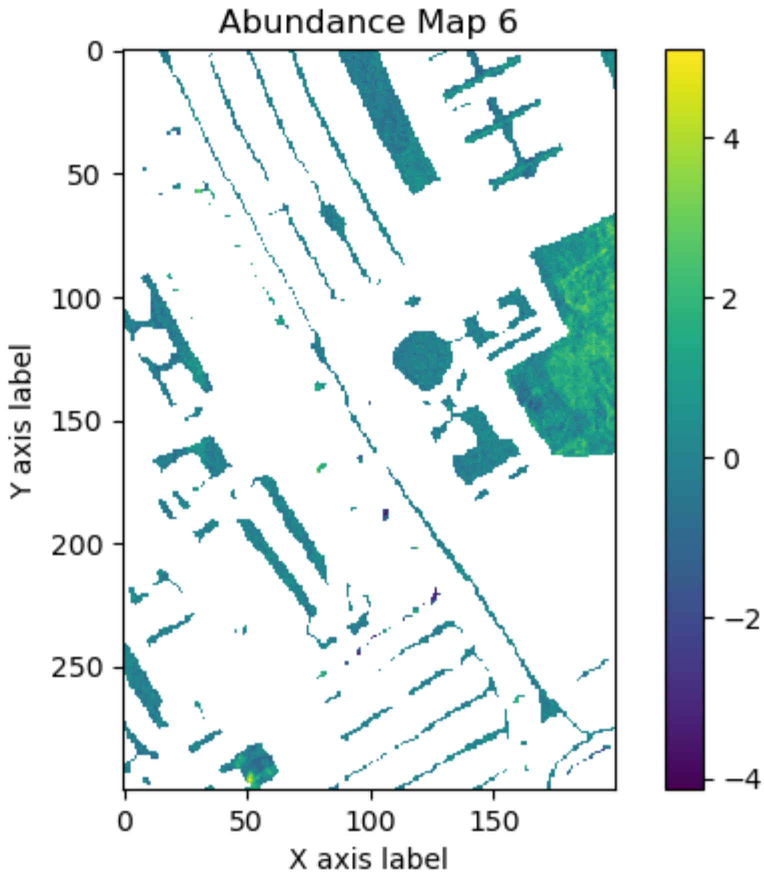


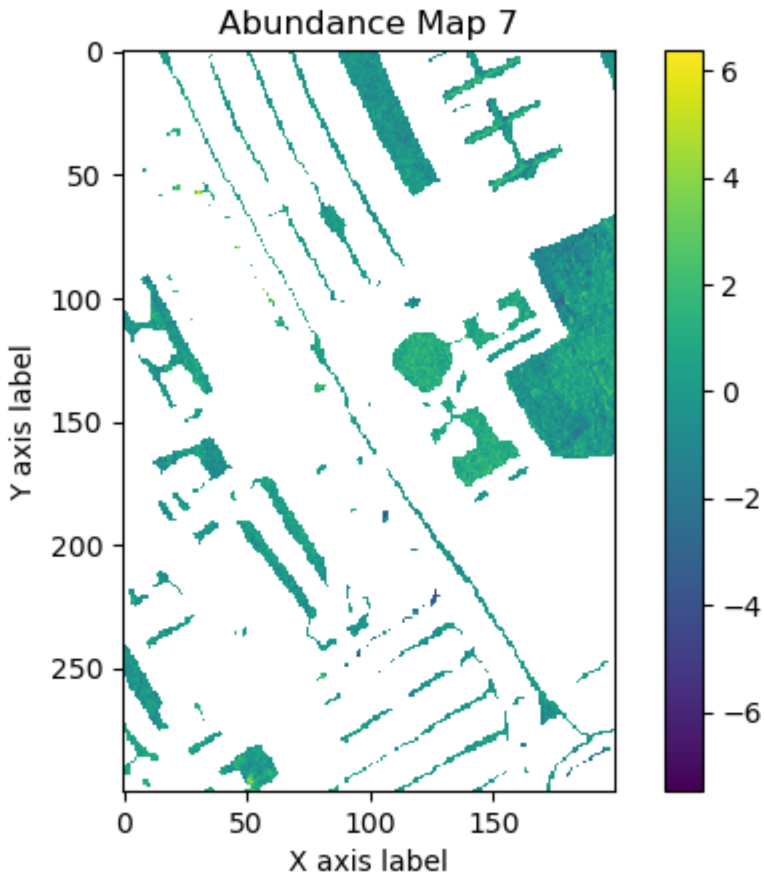

Abundance Map 2
0
50
100
150
200
250
Y axis label
0
50
100
150
X axis label
3
2
1
0
−1
−2
−3

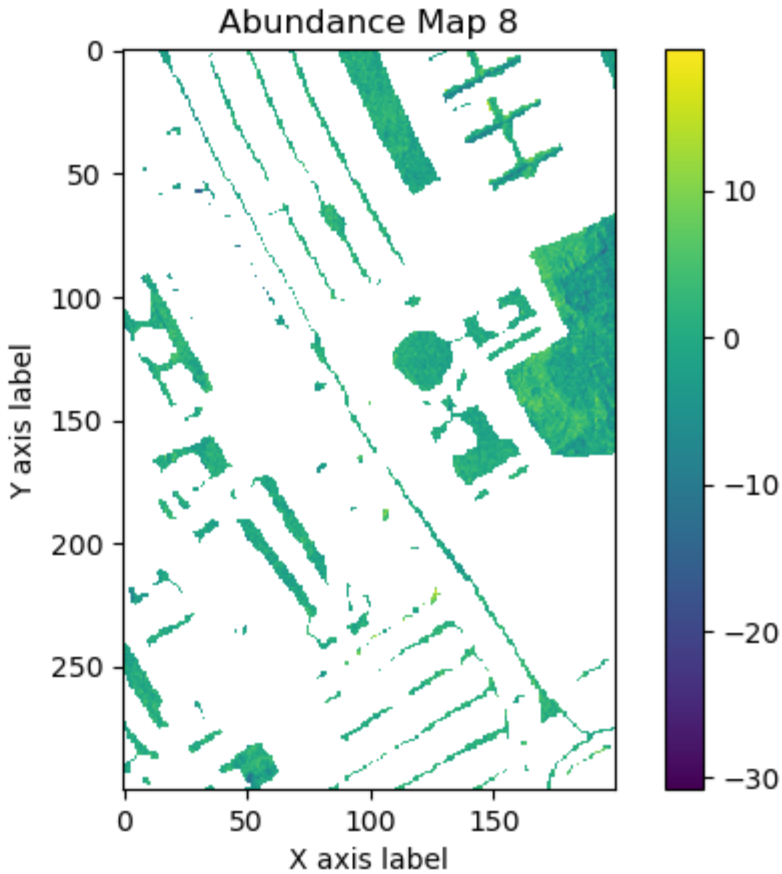

Abundance Map 3
0
50
100
150
200
250
Y axis label
0
50
100
150
X axis label
10
5
0
−5

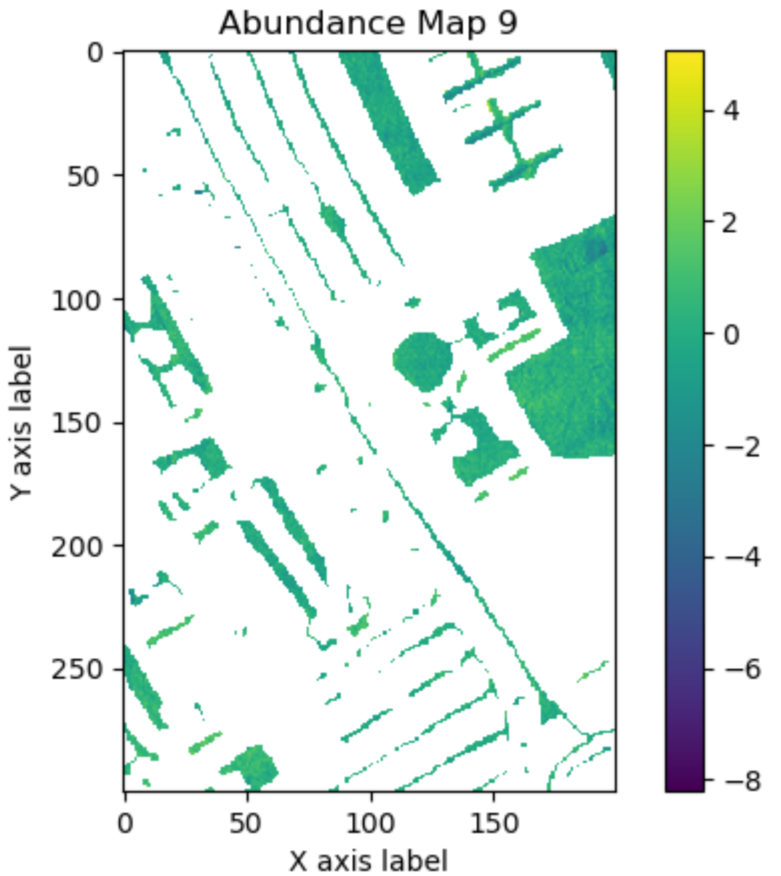

Abundance Map 4
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
1.5
1.0
0.5
0.0


Abundance Map 5
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2


Abundance Map 6
Y axis label
X axis label
0
50
100
150
200
250
4
2
0
−2
−4


Abundance Map 7
Y axis label
X axis label
0
50
100
150
200
250
6
4
2
0
−2
−4
−6





(c) Least squares imposing the non-negativity constraint on the entries of θ

we perform least squares to calculate the values of theta, this time imposing the non-negativity constraint on the entries of θ, and store them in a data frame called theta_est_non_negativity. We also calculate the reconstruction error for this method and derive the 9 abundance maps.

In [22]:
```
# Initialize an array to store the results
theta_est_non_negativity = np.zeros((9, 12829))

# Iterate over all pixels
for i in range(12829):
    # Perform non-negative least squares using nnls
    theta_non_negativity, _ = nnls(X, Y[:, i])

    # Store the non-negative least squares solution
    theta_est_non_negativity[:, i] = theta_non_negativity

# Calculate the reconstruction error
reconstruction_error_non_negativity = 0
for i in range(12829):
    y_est_non_negativity = np.dot(X, theta_est_non_negativity[:, i])
    reconstruction_error_non_negativity += np.square(np.linalg.norm(Y[:, i] - y_

reconstruction_error_non_negativity /= 12829

print("The parameters θ1, θ2, ..., θ9 with non-negativity constraint using nnls:
print("The Reconstruction Error with non-negativity constraint using nnls is:",
```

```
The parameters θ1, θ2, ..., θ9 with non-negativity constraint using nnls:
 [[0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.00724399 0.        ]
 [0.11349154 0.19626121 0.47833974 ... 0.         0.         0.        ]
 ...
 [0.732248   0.60148628 0.31290803 ... 0.14644373 0.22496021 0.        ]
 [0.         0.         0.         ... 0.39332283 0.23581575 0.40309152]
 [0.55945481 0.80799248 0.67606715 ... 0.35829716 0.3958444  0.5383244 ]]
The Reconstruction Error with non-negativity constraint using nnls is: 569339.29
10564177
```

In [161…
```
theta_est_non_negativity_df=pd.DataFrame(theta_est_non_negativity)
theta_est_non_negativity_df
```

Out[161]:

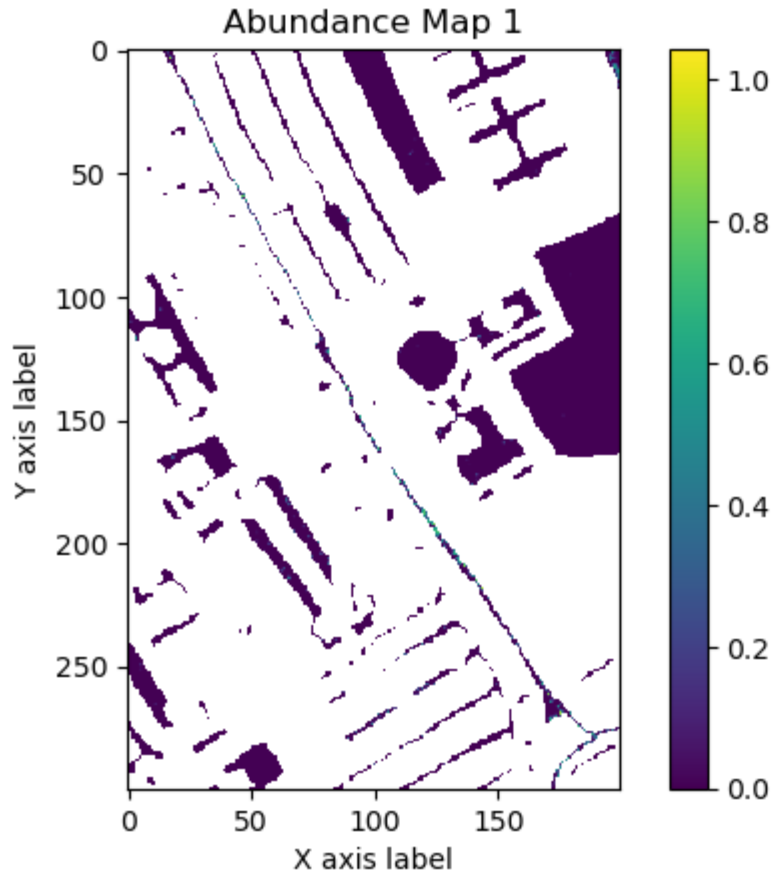
	0	1	2	3	4	5	6	7	8	
0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0
1	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000178	0.000000	0.0
2	0.113492	0.196261	0.478340	0.539566	0.238900	0.082072	0.262692	0.000000	0.618898	0.5
3	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0
4	0.008695	0.000000	0.000000	0.000000	0.013692	0.000000	0.028763	0.000000	0.000000	0.0
5	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.101824	0.000000	0.0
6	0.732248	0.601486	0.312908	0.317650	0.631426	0.789105	0.780756	0.676030	0.532665	0.6
7	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.289456	0.000000	0.0
8	0.559455	0.807992	0.676067	0.458290	0.429644	0.551890	0.102485	0.000000	0.080606	0.0

9 rows × 12829 columns

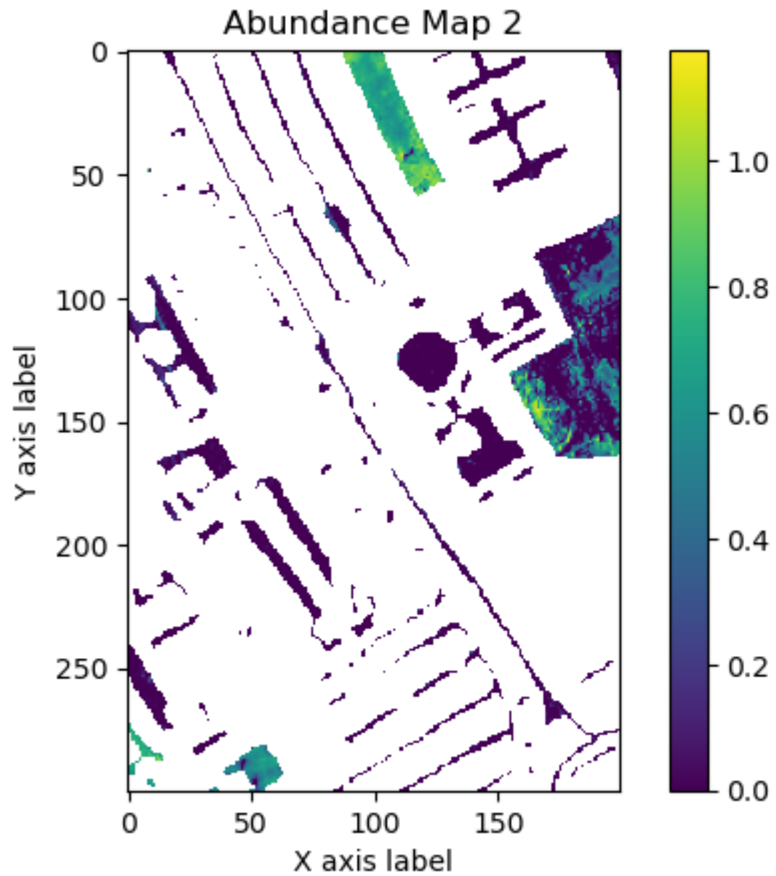
In [24]:
```
# Create a 200x300x9 array filled with zeros
result_array_c = np.zeros((300, 200, 9))

index = 0
for i in range(300):
    for j in range(200):
        if labels[i,j] != 0:
            result_array_c[i,j,:] = theta_est_non_negativity_df.iloc[:,index]
            index += 1
```

In [25]:
```
result_arrays_c = []

for k in range(9):
    result_arrays_c.append(result_array_c[:, :, k])

# Now result_arrays contains 9 2D arrays, each corresponding to a different k va
df_theta_c = []
for i in range(9):
    df_c = pd.DataFrame(result_arrays_c[i])
    df_theta_c.append(df_c)
```

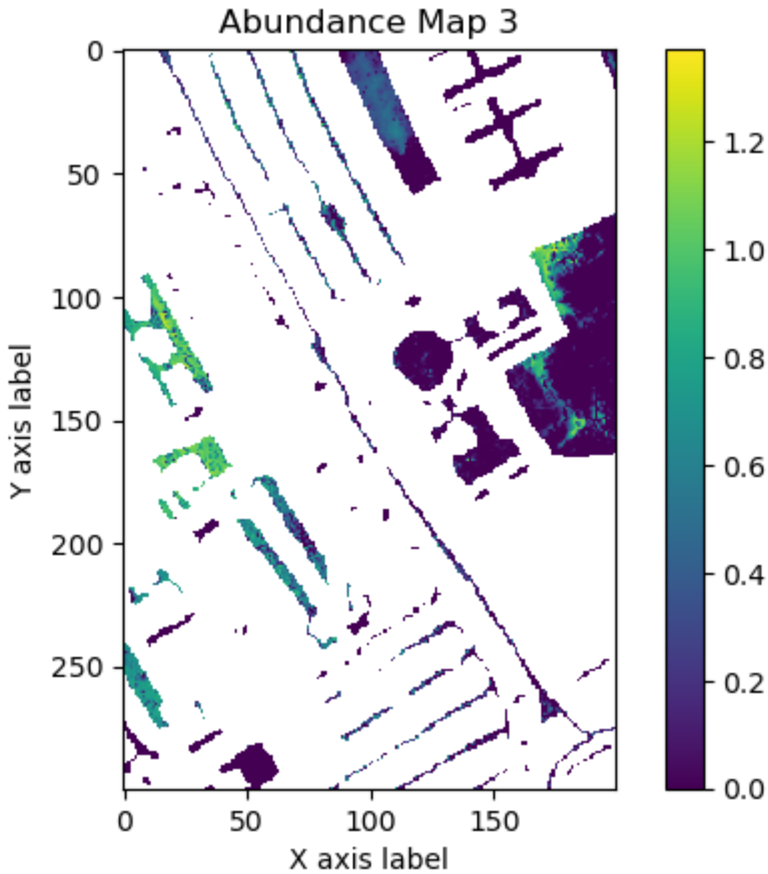
In [162…
```
for i in range(9):
    # Create the heatmap using Matplotlib's imshow function
    fig, ax = plt.subplots()


    mask = df_theta[i] == 0

    # Set the colormap (cmap) to 'viridis' and set_bad to white
    cmap = plt.get_cmap('viridis')
    cmap.set_bad(color='white')

    # Apply the mask to the data
    im = ax.imshow(np.ma.masked_array(df_theta_c[i], mask))

    # Add a color bar
    cbar = ax.figure.colorbar(im, ax=ax)

    # Set axis labels
    ax.set_xlabel('X axis label')
    ax.set_ylabel('Y axis label')

    # Add title
    ax.set_title('Abundance Map {}'.format(i + 1))

    # Show the plot
    plt.show()
```



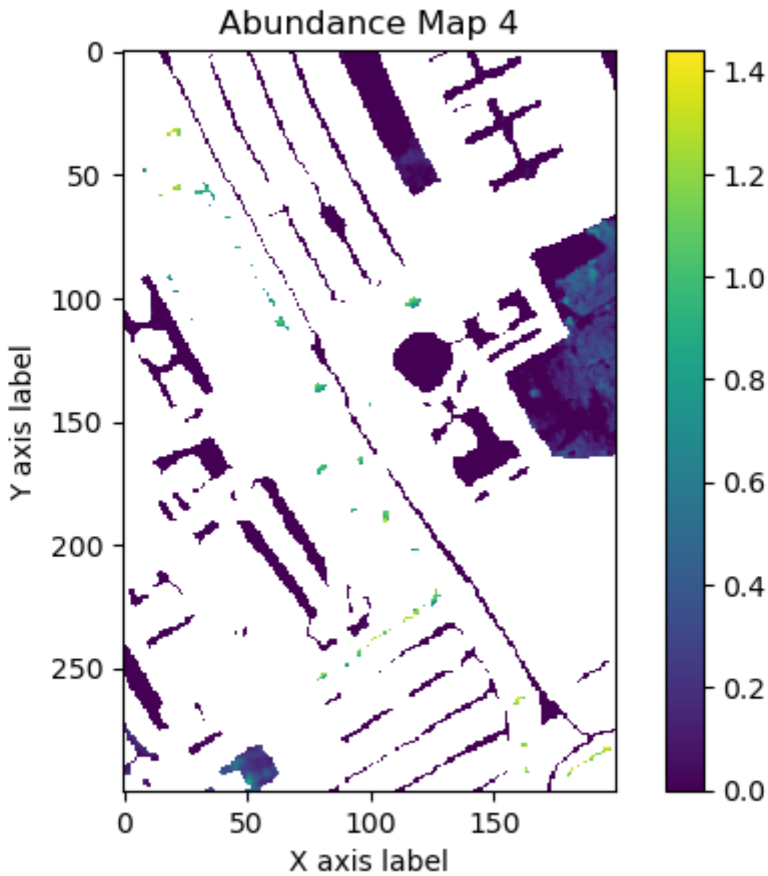
Abundance Map 1
Y axis label
X axis label
0
50
100
150
200
250
0.0
0.2
0.4
0.6
0.8
1.0



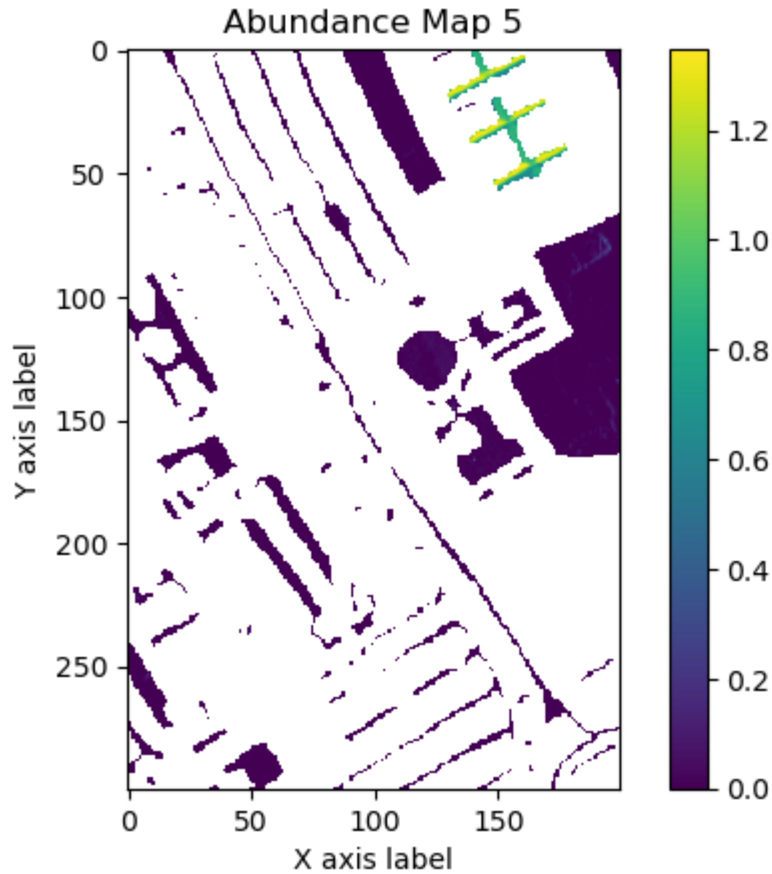
Abundance Map 2
Y axis label
X axis label
0
50
100
150
200
250
0.0
0.2
0.4
0.6
0.8
1.0

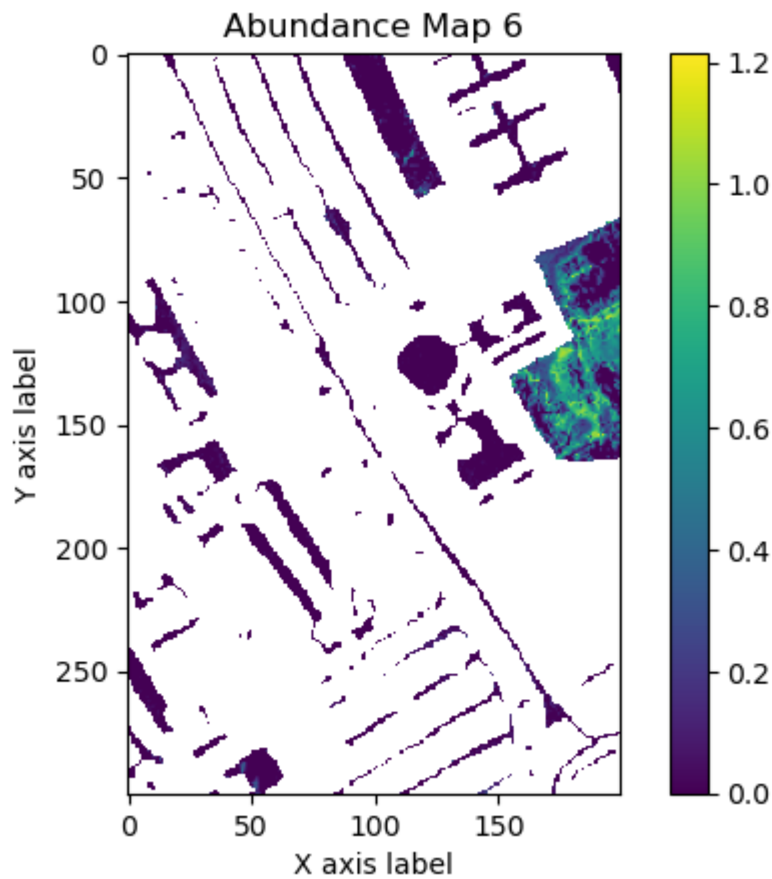


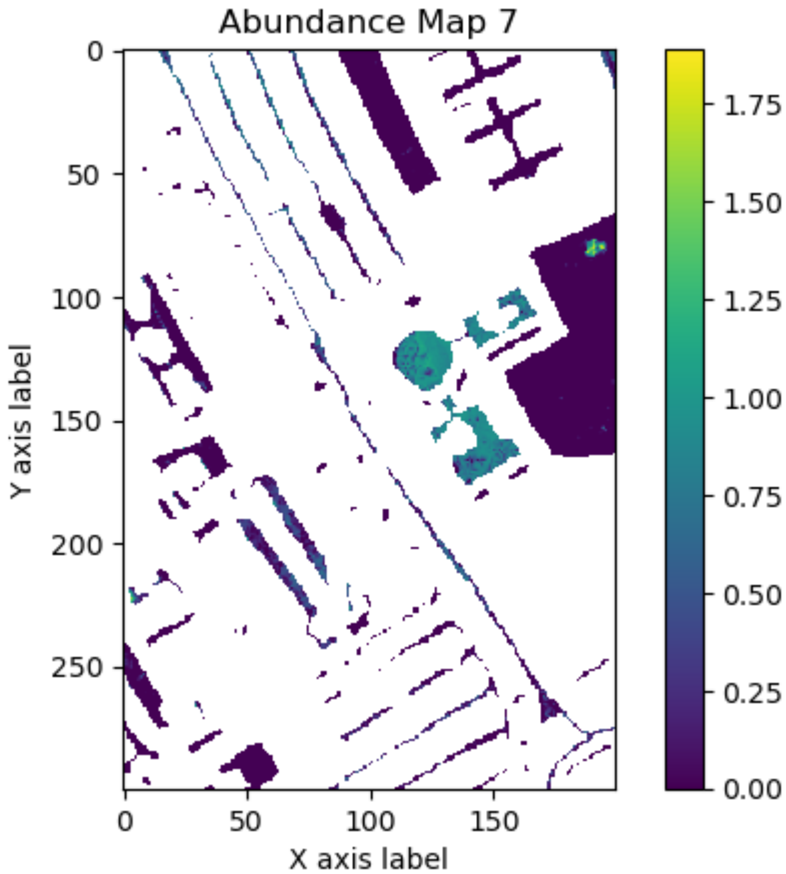
Abundance Map 3
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
0.0
0.2
0.4
0.6
0.8
1.0
1.2

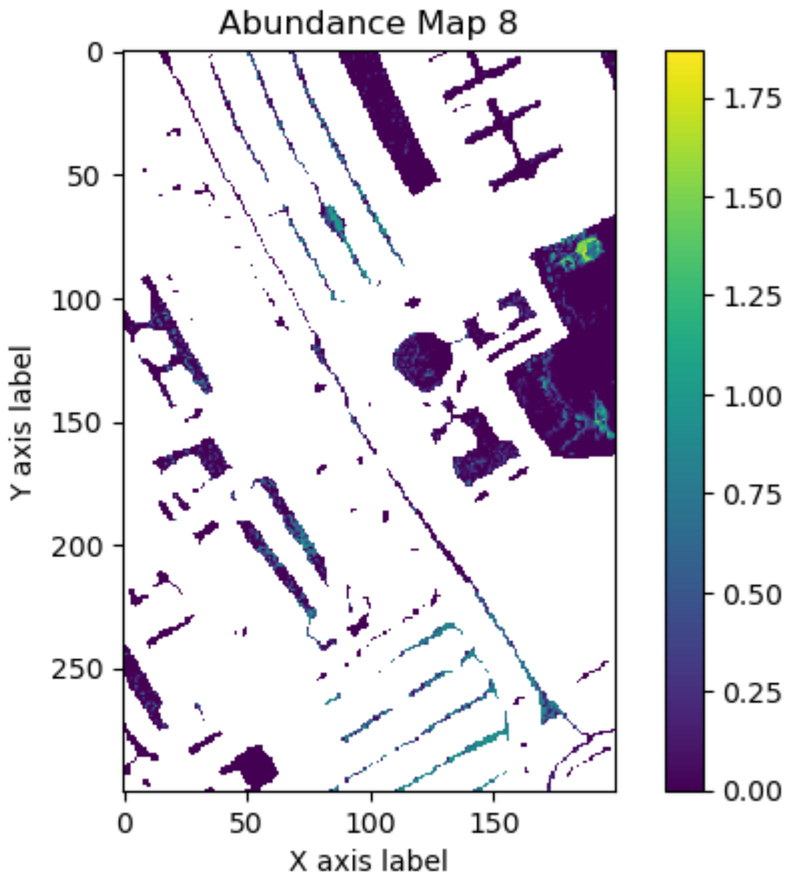


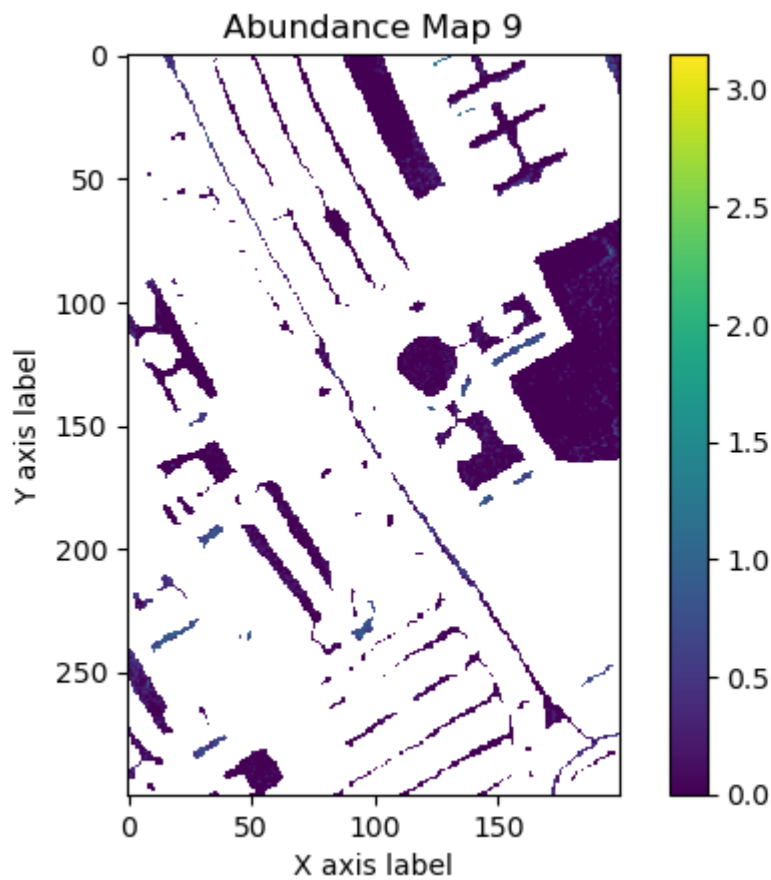
Abundance Map 4
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4


Abundance Map 5
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
0.0
0.2
0.4
0.6
0.8
1.0
1.2


Abundance Map 6
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
0.0
0.2
0.4
0.6
0.8
1.0
1.2


Abundance Map 7
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75


Abundance Map 8
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75



(d) Least squares imposing both the non-negativity and the sum-to-one constraint on the entries of θ.

we perform least squares to calculate the values of theta, this time imposing both the sum-to-one and the non-negativity constraints on the entries of θ, and store them in a data frame called theta_est_combined_constraints. We also calculate the reconstruction error for this method and derive the 9 abundance maps.

In [27]:
```
# Define the sum-to-one constraint function
def sum_to_one_constraint(theta):
    return np.sum(theta) - 1.0

# Define the objective function for minimization (Euclidean norm)
def objective(theta):
    y_est = np.dot(X, theta)
    return np.linalg.norm(Y[:, i] - y_est, ord=2)  # Euclidean norm

# Initialize an array to store the results
theta_est_combined_constraints = np.zeros((9, 12829))

# Iterate over all pixels
for i in range(12829):
    # Perform non-negative least squares using nnls as an initial guess
    theta_non_negativity, _ = nnls(X, Y[:, i])

    # Define the optimization problem with both sum-to-one and non-negativity co
    constraint_definitions = [{'type': 'eq', 'fun': sum_to_one_constraint}]
    bounds = [(0, None)] * len(theta_non_negativity)  # Non-negativity constrain

    # Perform the optimization with both constraints
    optimization_result = minimize(objective, theta_non_negativity, constraints=

    # Store the optimized abundance vector
    theta_est_combined_constraints[:, i] = optimization_result.x

# Calculate the reconstruction error
reconstruction_error_combined_constraints = 0
for i in range(12829):
    y_est_combined_constraints = np.dot(X, theta_est_combined_constraints[:, i])
    reconstruction_error_combined_constraints += np.square(np.linalg.norm(Y[:, i

reconstruction_error_combined_constraints /= 12829

print("The parameters θ1, θ2, ..., θ9 with both constraints:\n", theta_est_combi
print("The Reconstruction Error with both constraints is:", reconstruction_error
```

```
The parameters θ1, θ2, ..., θ9 with both constraints:
 [[6.49668618e-10 2.92916545e-01 5.90886604e-01 ... 0.00000000e+00
  2.03204361e-13 4.18413839e-02]
 [8.06886821e-11 0.00000000e+00 3.76054811e-02 ... 0.00000000e+00
  9.29445454e-13 9.92119905e-13]
 [6.42218420e-01 4.45404047e-01 2.80496002e-01 ... 0.00000000e+00
  1.71536408e-13 0.00000000e+00]
 ...
 [1.68747234e-09 0.00000000e+00 4.91961294e-08 ... 1.40592300e-01
  3.06521067e-01 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 1.37956899e-09 ... 3.63288975e-01
  1.81366034e-01 3.68186172e-01]
 [2.82826452e-01 1.99475066e-01 6.69764511e-02 ... 4.60201661e-01
  4.29442723e-01 4.85443543e-01]]
The Reconstruction Error with both constraints is: 3143837.0783262127
```

In [28]:
```
theta_est_combined_constraints_df=pd.DataFrame(theta_est_combined_constraints)
theta_est_combined_constraints_df
```

Out[28]:

	0	1	2	3	4	5	6	
0	6.496686e-10	0.292917	5.908866e-01	3.268878e-12	0.000000	0.000000e+00	0.000000	0.000(
1	8.068868e-11	0.000000	3.760548e-02	2.905149e-12	0.000000	0.000000e+00	0.000000	0.057
2	6.422184e-01	0.445404	2.804960e-01	7.856002e-01	0.238900	6.137144e-01	0.262692	0.000(
3	1.224179e-09	0.007790	3.160535e-09	3.894963e-12	0.000000	0.000000e+00	0.000000	0.000(
4	7.495513e-02	0.054414	2.403540e-02	3.070195e-02	0.013692	6.532667e-02	0.028763	0.018
5	0.000000e+00	0.000000	8.261689e-09	3.155720e-09	0.000000	4.154508e-08	0.000000	0.034
6	1.687472e-09	0.000000	4.919613e-08	1.262417e-08	0.631426	5.716027e-02	0.780756	0.386
7	0.000000e+00	0.000000	1.379569e-09	7.034891e-10	0.000000	9.386992e-09	0.000000	0.503
8	2.828265e-01	0.199475	6.697645e-02	1.836979e-01	0.429644	2.637986e-01	0.102485	0.000(

9 rows × 12829 columns

In [29]:

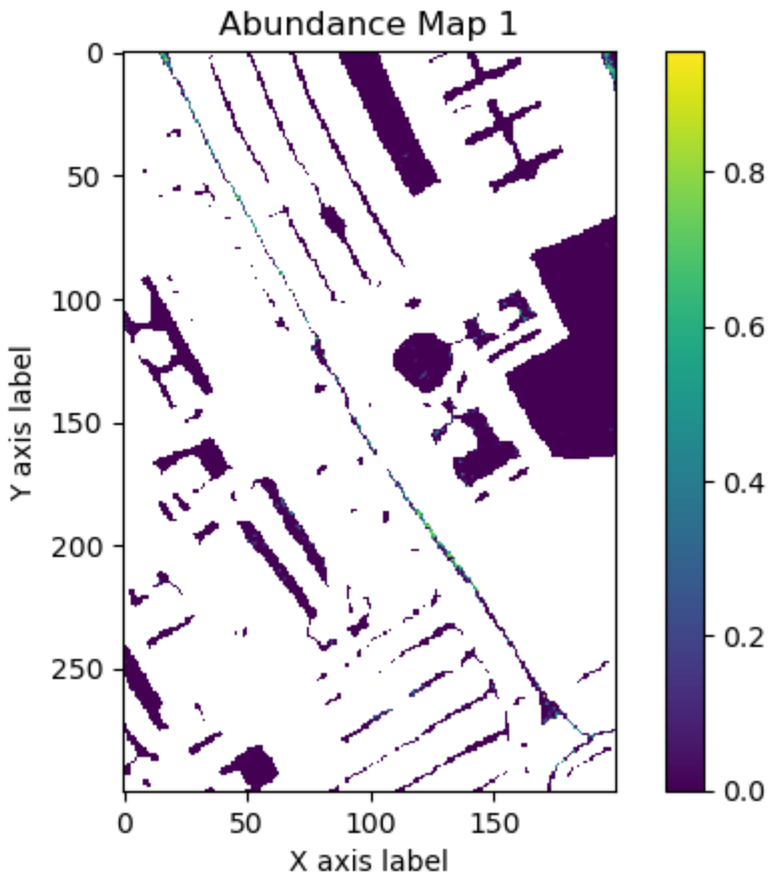
```
# Create a 200x300x9 array filled with zeros
result_array_d = np.zeros((300, 200, 9))

# Iterate over the pixels and fill in the values from the theta_est array
index = 0
for i in range(300):
    for j in range(200):
        if labels[i,j] != 0:
            result_array_d[i,j,:] = theta_est_combined_constraints_df.iloc[:,ind
            index += 1
```

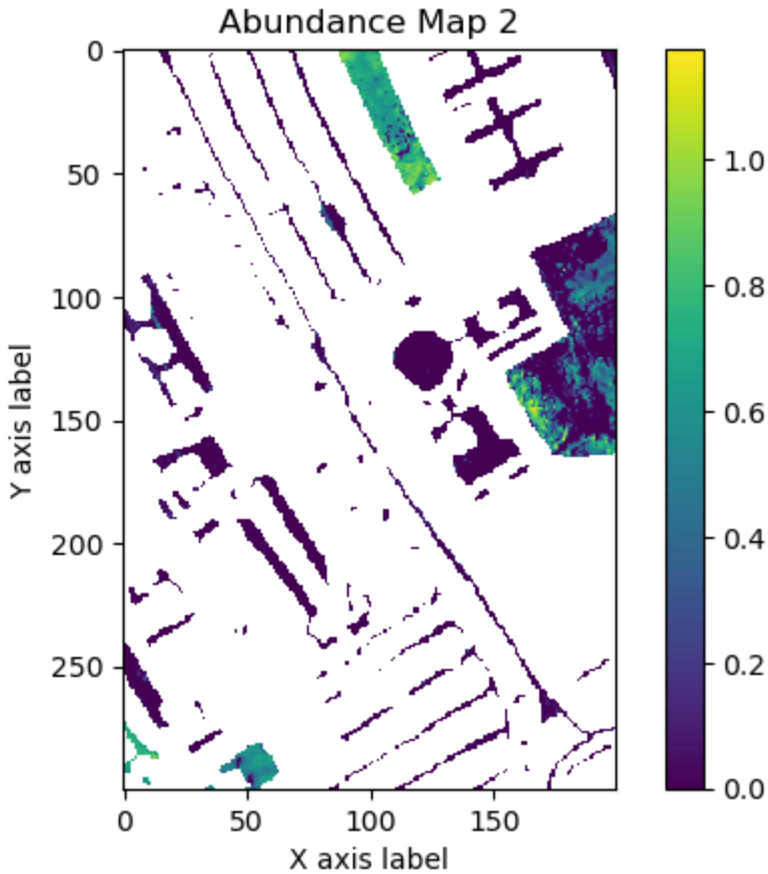
In [30]:

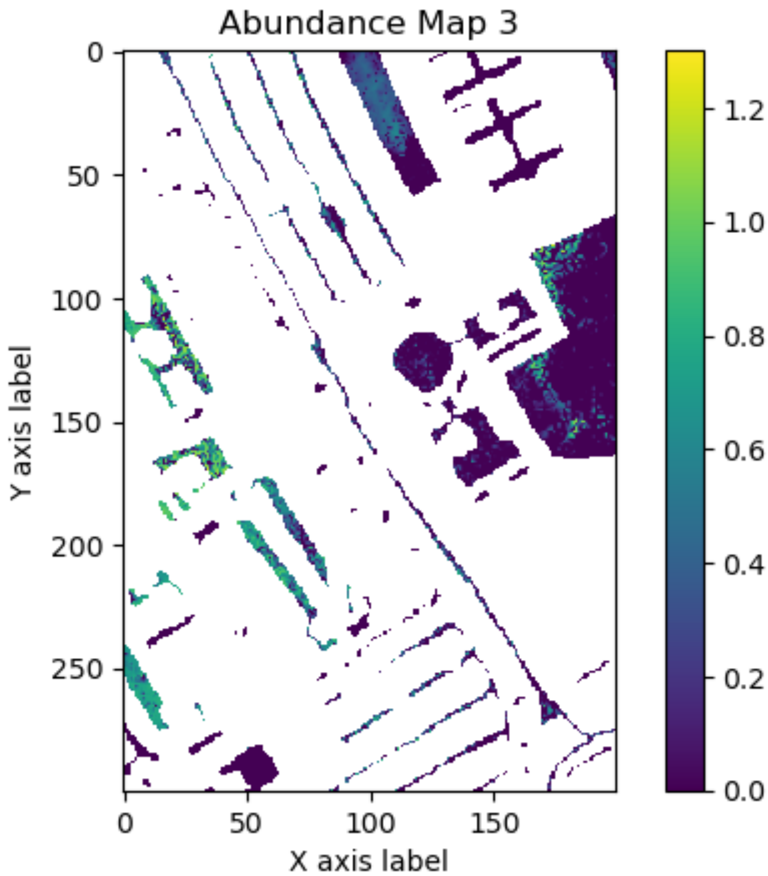
```
result_arrays_d = []

for k in range(9):
    result_arrays_d.append(result_array_d[:, :, k])

# Now result_arrays contains 9 2D arrays, each corresponding to a different k va
df_theta_d = []
for i in range(9):
    df_d = pd.DataFrame(result_arrays_d[i])
    df_theta_d.append(df_d)
```

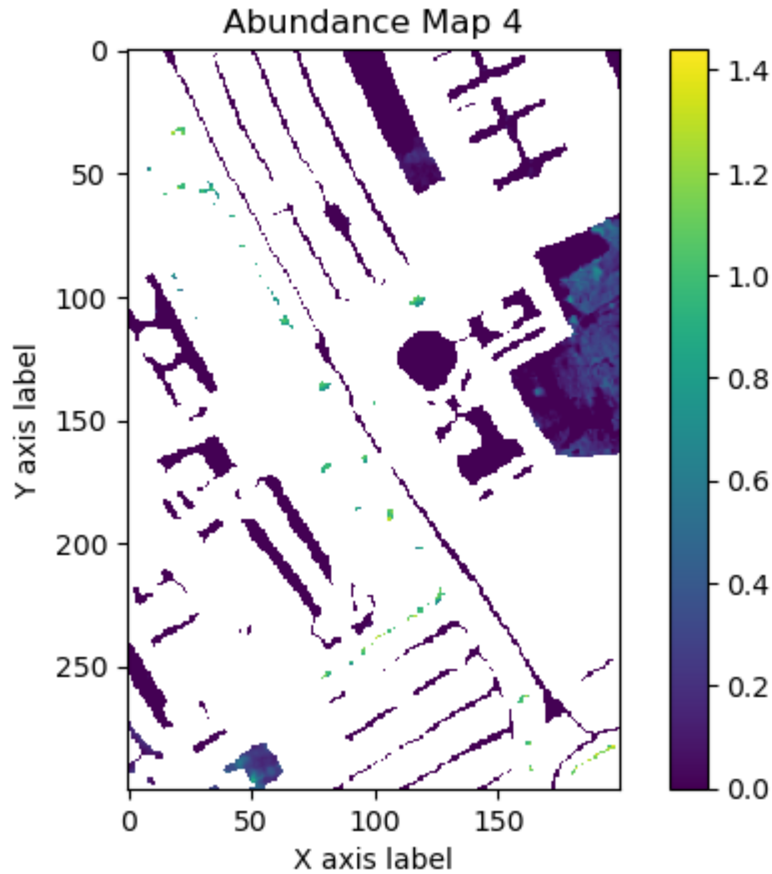
In [235…

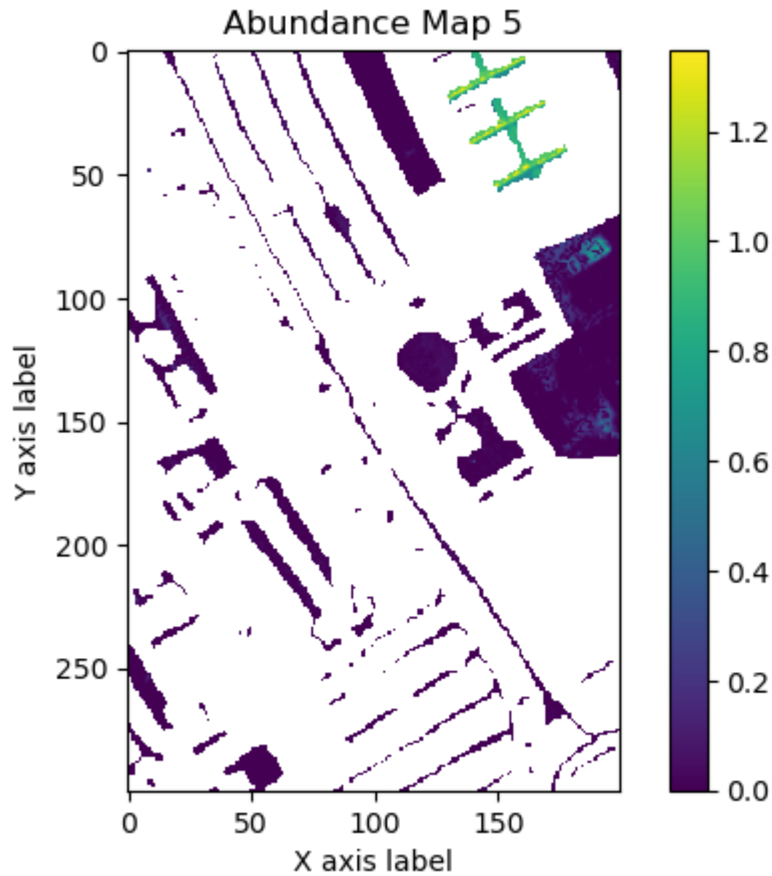
```
for i in range(9):
    # Create the heatmap using Matplotlib's imshow function
    fig, ax = plt.subplots()

    mask = df_theta[i] == 0

    # Set the colormap (cmap) to 'viridis' and set_bad to white
    cmap = plt.get_cmap('viridis')
    cmap.set_bad(color='white')

    # Apply the mask to the data
    im = ax.imshow(np.ma.masked_array(df_theta_d[i], mask))

    # Add a color bar
    cbar = ax.figure.colorbar(im, ax=ax)

    # Set axis labels
    ax.set_xlabel('X axis label')
    ax.set_ylabel('Y axis label')

    # Add title
    ax.set_title('Abundance Map {}'.format(i + 1))

    # Show the plot
    plt.show()
```

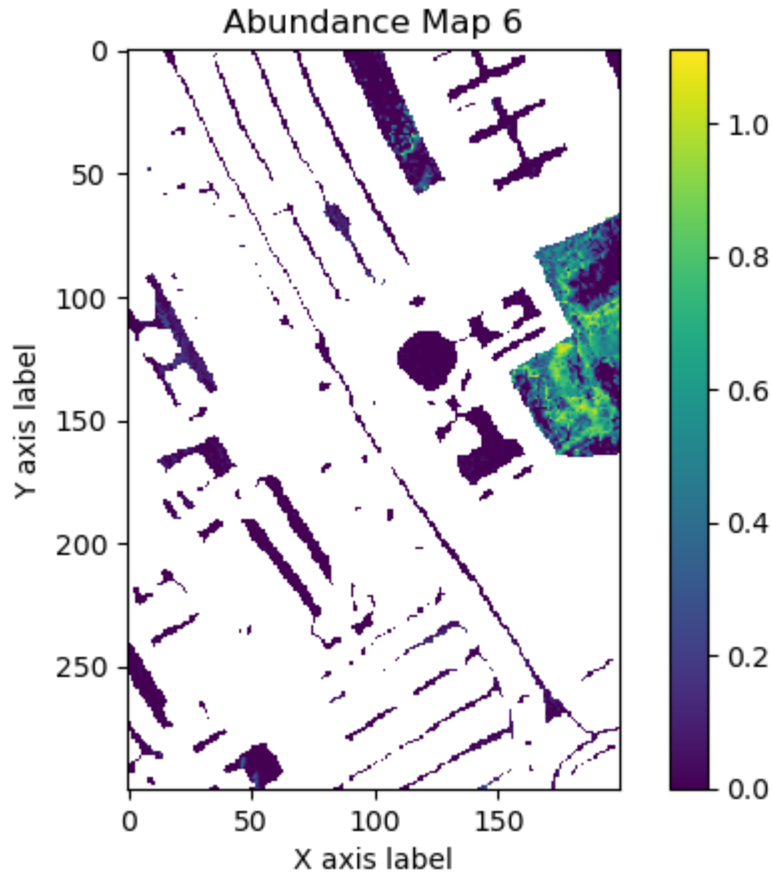


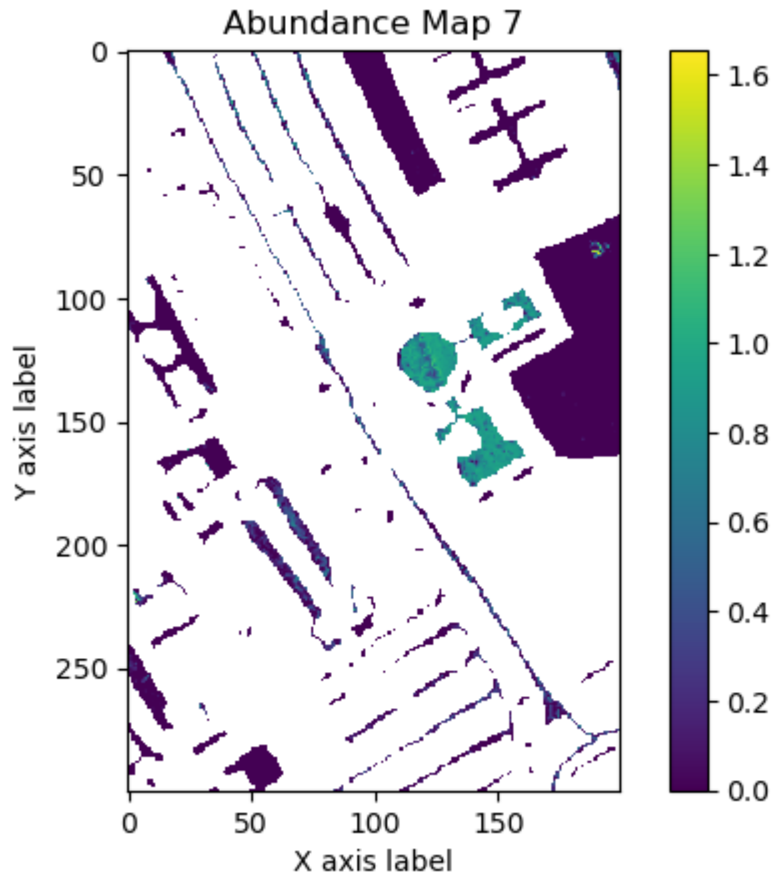

Abundance Map 2
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
0.0
0.2
0.4
0.6
0.8
1.0

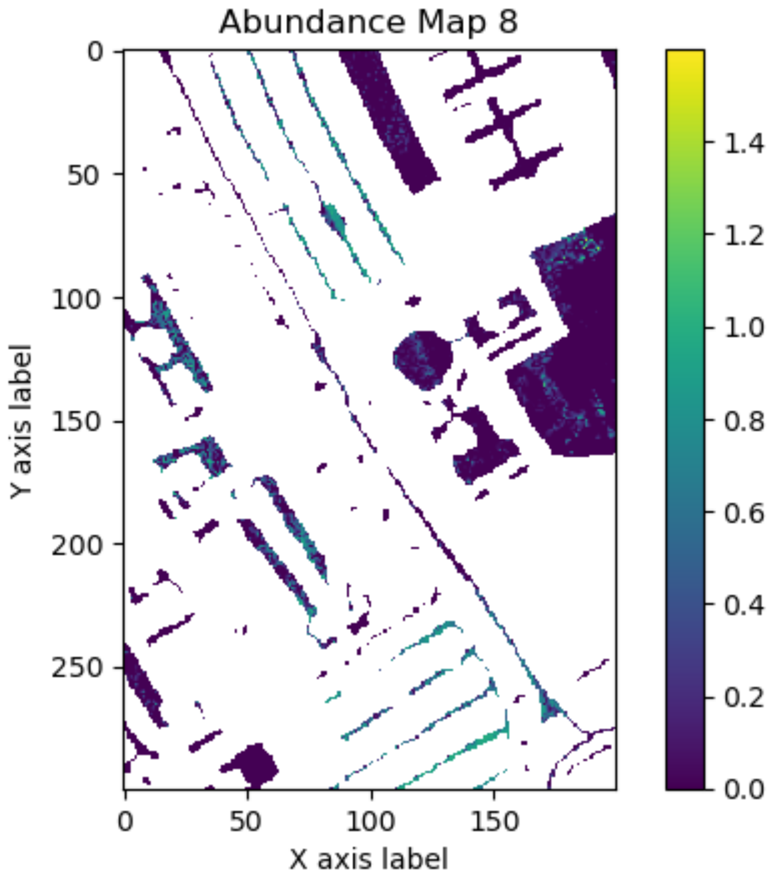

Abundance Map 3
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
0.0
0.2
0.4
0.6
0.8
1.0
1.2

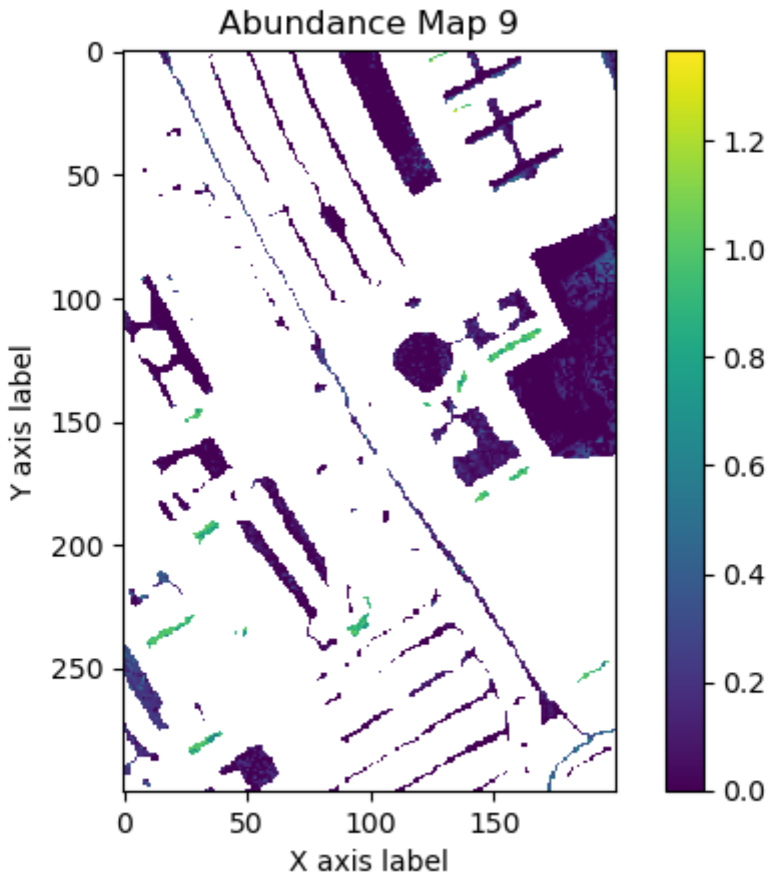

Abundance Map 4
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4


Abundance Map 5
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
0.0
0.2
0.4
0.6
0.8
1.0
1.2


Abundance Map 6
Y axis label
X axis label
0
50
100
150
200
250
0.0
0.2
0.4
0.6
0.8
1.0


Abundance Map 7
Y axis label
X axis label
0
50
100
150
200
250
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6





(e) LASSO, i.e., impose sparsity on θ via $l1$ norm minimization.

We did lasso for two values of alpha. alpha=0.1 and alpha = 1

alpha = 0.1

In [266…

```
from sklearn.linear_model import Lasso

# Initialize an array to store the results
theta_est_lasso = np.zeros((9, 12829))

# Set the regularization strength
alpha = 0.1

# Iterate over all pixels
for i in range(12829):

   # Perform LASSO regularization with increased max_iter
    lasso = Lasso(alpha=alpha, max_iter=100000)
    lasso.fit(X, Y[:, i])

    # Store the optimized abundance vector
    theta_est_lasso[:, i] = lasso.coef_

# Calculate the reconstruction error
reconstruction_error_lasso = 0

for i in range(12829):
    y_est_lasso = np.dot(X, theta_est_lasso[:, i])
    reconstruction_error_lasso += np.square(np.linalg.norm(Y[:, i] - y_est_lasso

reconstruction_error_lasso /= 12829

print("The parameters θ1, θ2, ..., θ9 with LASSO:\n", theta_est_lasso)
print("The Reconstruction Error with LASSO is:", reconstruction_error_lasso)
```

```
The parameters θ1, θ2, ..., θ9 with LASSO:
 [[ 0.17008725 -0.10907687 -1.97208287 ... -0.50191106 -1.65822509
  -0.0740156 ]
 [ 0.24684469  0.29465553  0.88611428 ...  0.08507101  0.20041563
  -0.01496685]
 [ 0.17688752  0.67359657  0.09209181 ... -1.14870277  0.25308743
  -0.75803897]
 ...
 [-1.17261894 -0.16373197  0.97340953 ... -0.72360771  1.78886369
   0.93058527]
 [ 0.66417905 -0.29767859  1.36670292 ...  2.34747606  0.44431747
   1.34181619]
 [-0.45252154 -0.13450002  0.9791593  ...  0.44669519  1.45388957
   1.7299286 ]]
The Reconstruction Error with LASSO is: 115208329.49775025
```

In [267…

```
theta_est_lasso_df=pd.DataFrame(theta_est_lasso)
theta_est_lasso_df
```

Out[267]:

	0	1	2	3	4	5	6	7	
0	0.170087	-0.109077	-1.972083	-1.502636	-1.818170	-2.681121	-2.184596	2.441509	-1.757
1	0.246845	0.294656	0.886114	0.212873	0.240493	0.556430	0.145247	0.124550	0.367
2	0.176888	0.673597	0.092092	1.331989	1.130136	-0.389806	0.335181	0.660861	-0.057
3	0.114800	0.097735	0.068307	-0.053085	-0.011983	-0.025338	0.009875	0.090484	-0.004
4	0.001448	-0.033451	-0.014619	-0.004823	0.025530	0.015404	0.027718	-0.006529	-0.001
5	-0.500557	-0.436482	-0.969975	-0.102854	-0.231943	-0.534079	-0.490633	-0.178663	-0.286
6	-1.172619	-0.163732	0.973410	1.978877	1.005363	0.440278	0.466526	0.416658	0.515
7	0.664179	-0.297679	1.366703	-0.556629	0.127121	2.382239	2.146075	-1.744315	1.620
8	-0.452522	-0.134500	0.979159	1.399263	1.048034	1.534912	1.595103	-1.547334	0.488

9 rows × 12829 columns

In [268…

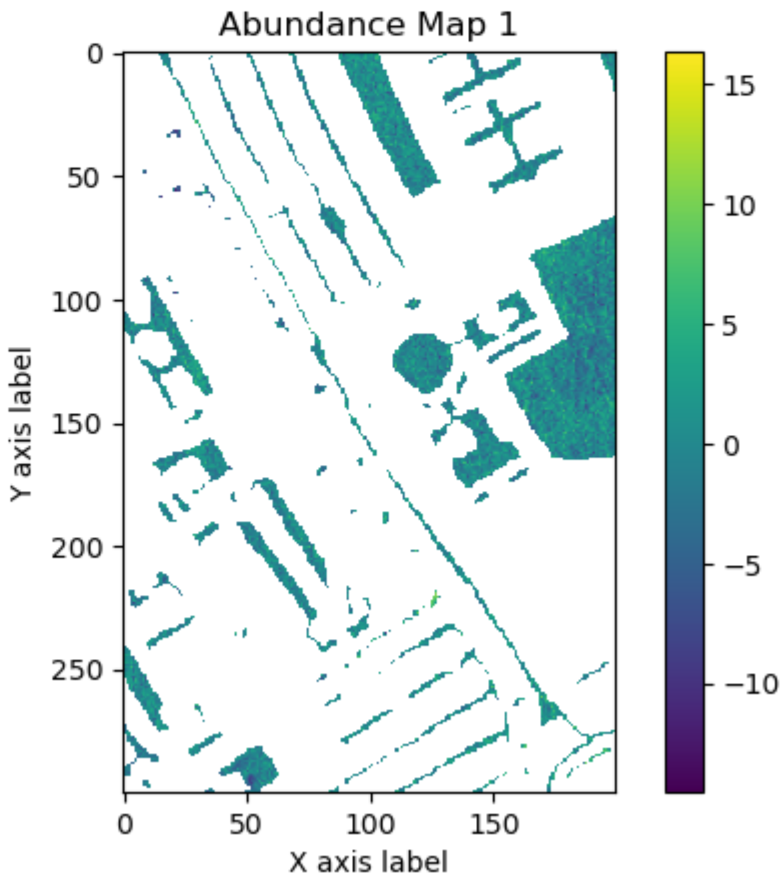
```
# Create a 200x300x9 array filled with zeros
result_array_e = np.zeros((300, 200, 9))

# Iterate over the pixels and fill in the values from the theta_est_lasso array
index = 0
for i in range(300):
    for j in range(200):
        if labels[i,j] != 0:
            result_array_e[i,j,:] = theta_est_lasso_df.iloc[:,index]
            index += 1
```

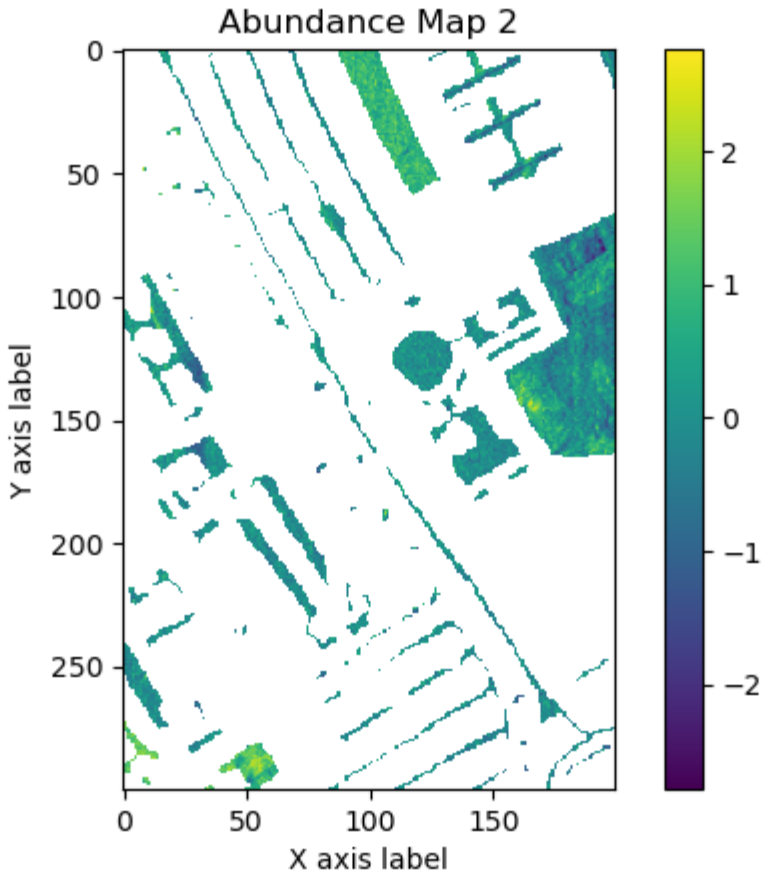
In [269…

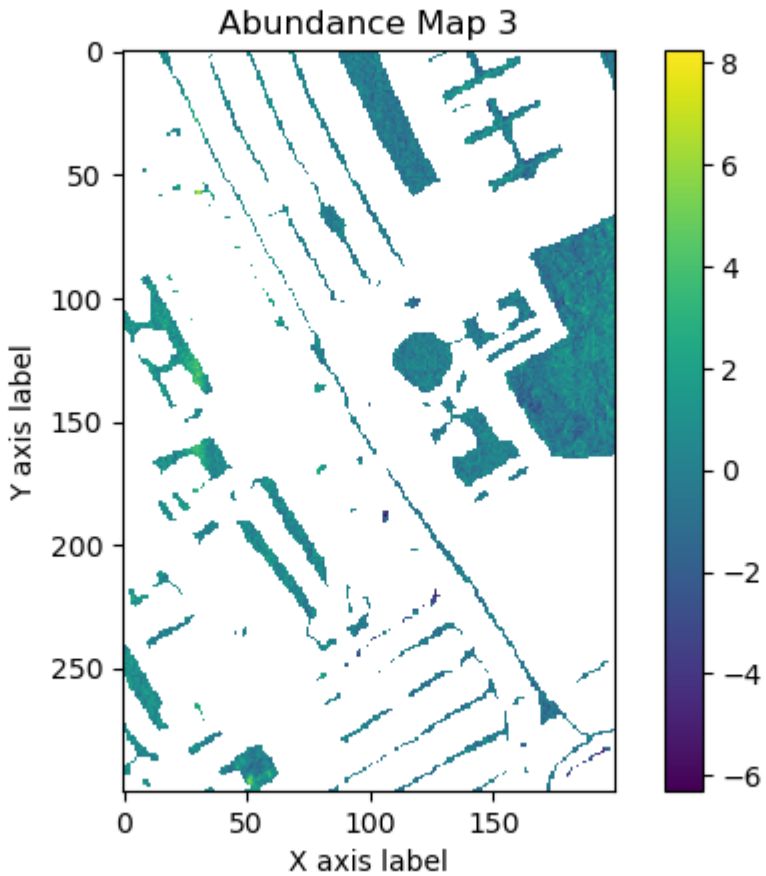
```
result_arrays_e = []

for k in range(9):
    result_arrays_e.append(result_array_e[:, :, k])

# Now result_arrays contains 9 2D arrays, each corresponding to a different k va
df_theta_e = []
for i in range(9):
    df_e = pd.DataFrame(result_arrays_e[i])
    df_theta_e.append(df_e)
```

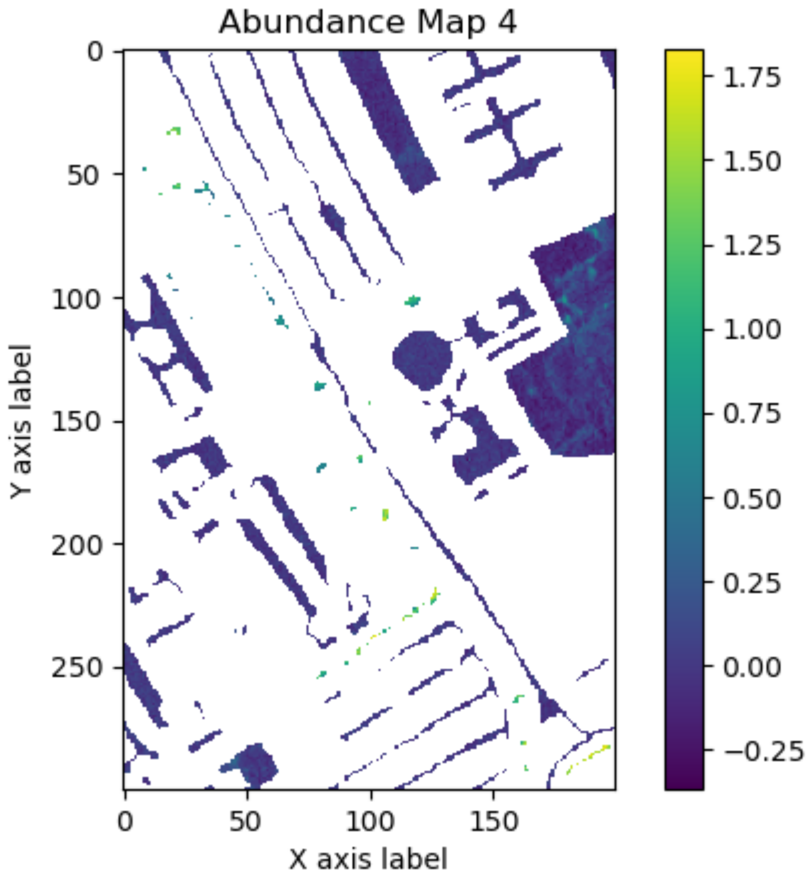
In [270…

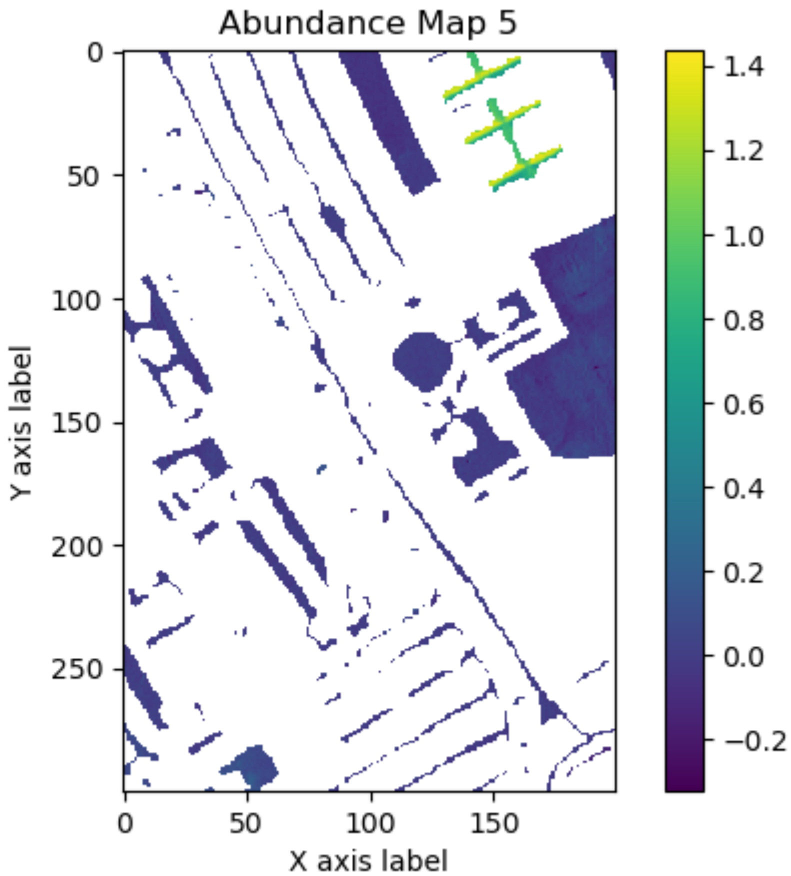
```
for i in range(9):
    # Create the heatmap using Matplotlib's imshow function
    fig, ax = plt.subplots()


    mask = df_theta[i] == 0

    # Set the colormap (cmap) to 'viridis' and set_bad to white
    cmap = plt.get_cmap('viridis')
    cmap.set_bad(color='white')

    # Apply the mask to the data
    im = ax.imshow(np.ma.masked_array(df_theta_e[i], mask))

    # Add a color bar
    cbar = ax.figure.colorbar(im, ax=ax)

    # Set axis labels
    ax.set_xlabel('X axis label')
    ax.set_ylabel('Y axis label')

    # Add title
    ax.set_title('Abundance Map {}'.format(i + 1))

    # Show the plot
    plt.show()
```

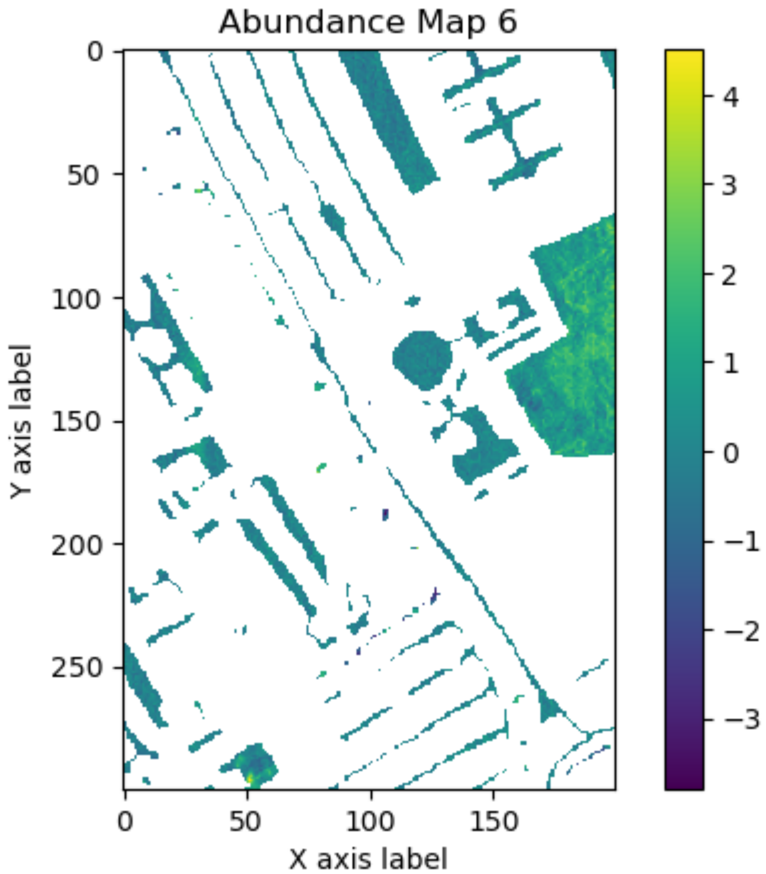


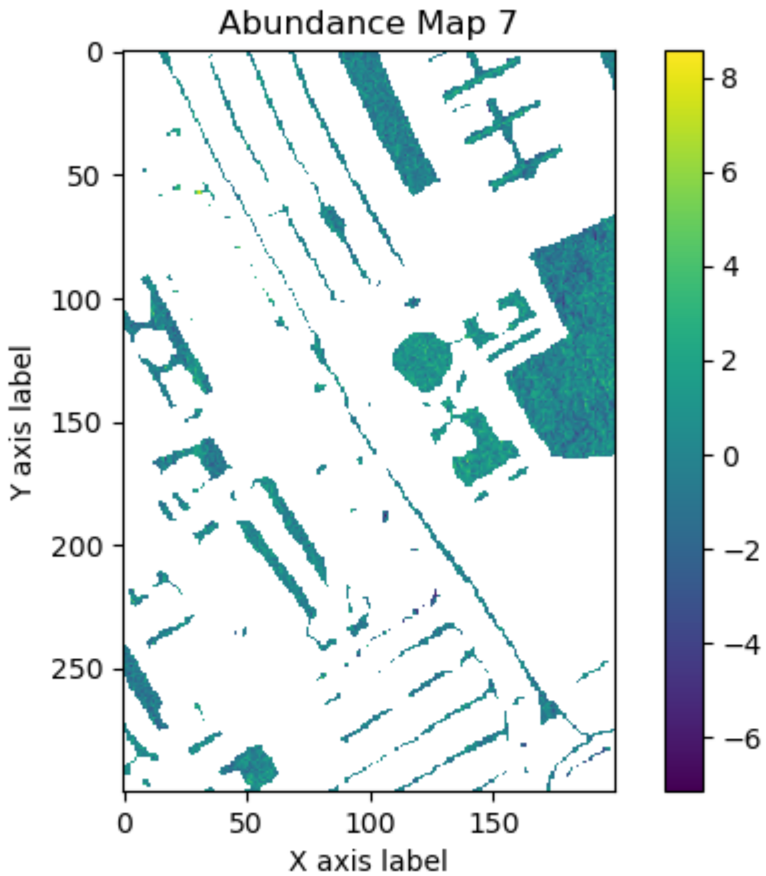

Abundance Map 2
0
50
100
150
200
250
Y axis label
0
50
100
150
X axis label
2
1
0
−1
−2

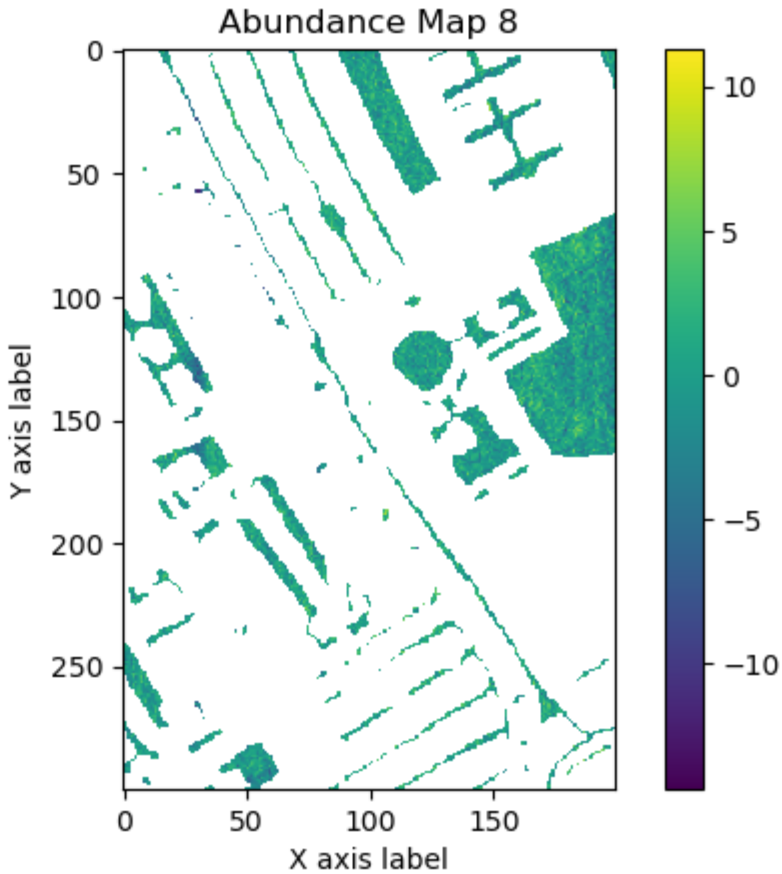

Abundance Map 3
0
50
100
150
200
250
Y axis label
0
50
100
150
X axis label
8
6
4
2
0
−2
−4
−6

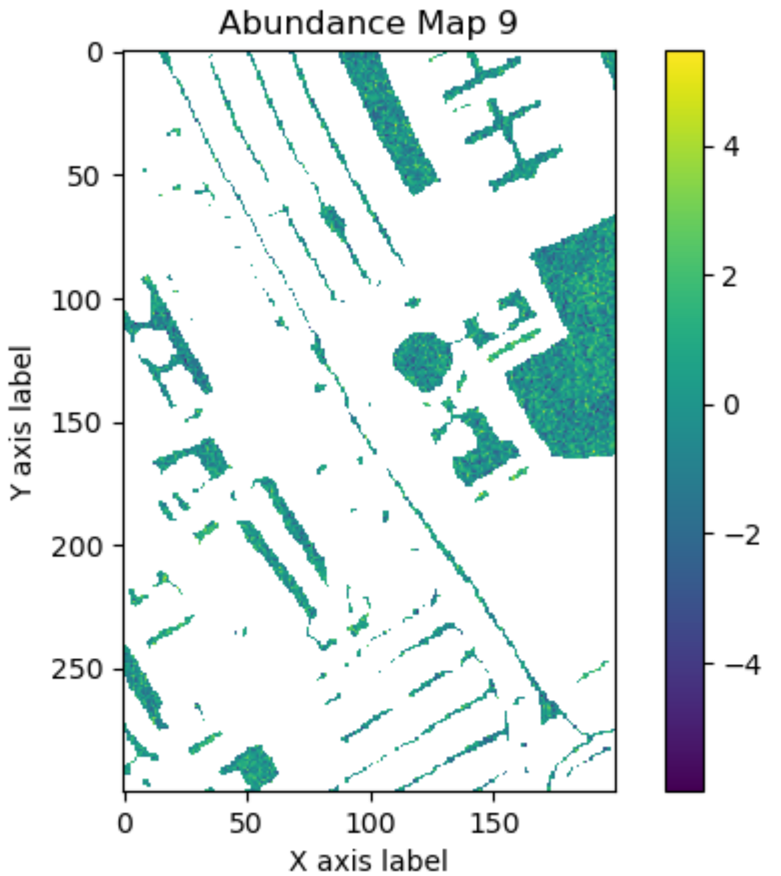

Abundance Map 4
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25


Abundance Map 5
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2


Abundance Map 6
0
50
100
150
200
250
Y axis label
0
50
100
150
X axis label
4
3
2
1
0
−1
−2
−3


Abundance Map 7
0
50
100
150
200
250
Y axis label
0
50
100
150
X axis label
8
6
4
2
0
−2
−4
−6


Abundance Map 8
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
10
5
0
−5
−10


Abundance Map 9
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
4
2
0
−2
−4

alpha = 1

In [32]:
```
# Initialize an array to store the results
theta_est_lasso = np.zeros((9, 12829))

# Set the regularization strength
alpha = 1

# Iterate over all pixels
for i in range(12829):

   # Perform LASSO regularization with increased max_iter
    lasso = Lasso(alpha=alpha, max_iter=100000)
    lasso.fit(X, Y[:, i])

    # Store the optimized abundance vector
    theta_est_lasso[:, i] = lasso.coef_

# Calculate the reconstruction error
reconstruction_error_lasso = 0

for i in range(12829):
    y_est_lasso = np.dot(X, theta_est_lasso[:, i])
    reconstruction_error_lasso += np.square(np.linalg.norm(Y[:, i] - y_est_lasso

reconstruction_error_lasso /= 12829

print("The parameters θ1, θ2, ..., θ9 with LASSO:\n", theta_est_lasso)
print("The Reconstruction Error with LASSO is:", reconstruction_error_lasso)
```

```
The parameters θ1, θ2, ..., θ9 with LASSO:
 [[ 0.14837029 -0.12470699 -1.77628676 ... -0.24616959 -1.49812124
   0.        ]
 [ 0.24518842  0.29748335  0.85005191 ...  0.03260899  0.17349004
  -0.02851849]
 [ 0.19516652  0.64730547  0.15742452 ... -1.00627117  0.29955037
  -0.71044302]
 ...
 [-1.13925943 -0.17473983  0.96769572 ... -0.66724841  1.77692891
   0.94380768]
 [ 0.64024276 -0.25341087  1.18086421 ...  2.01606946  0.30539174
   1.23949054]
 [-0.44136891 -0.1251457   0.92791502 ...  0.38599119  1.41042613
   1.70674547]]
The Reconstruction Error with LASSO is: 117150413.7670268
```

In [33]:
```
theta_est_lasso_df=pd.DataFrame(theta_est_combined_constraints)
theta_est_lasso_df
```

Out[33]:

	0	1	2	3	4	5	6	
0	6.496686e-10	0.292917	5.908866e-01	3.268878e-12	0.000000	0.000000e+00	0.000000	0.000(
1	8.068868e-11	0.000000	3.760548e-02	2.905149e-12	0.000000	0.000000e+00	0.000000	0.057
2	6.422184e-01	0.445404	2.804960e-01	7.856002e-01	0.238900	6.137144e-01	0.262692	0.000(
3	1.224179e-09	0.007790	3.160535e-09	3.894963e-12	0.000000	0.000000e+00	0.000000	0.000(
4	7.495513e-02	0.054414	2.403540e-02	3.070195e-02	0.013692	6.532667e-02	0.028763	0.018
5	0.000000e+00	0.000000	8.261689e-09	3.155720e-09	0.000000	4.154508e-08	0.000000	0.034
6	1.687472e-09	0.000000	4.919613e-08	1.262417e-08	0.631426	5.716027e-02	0.780756	0.386
7	0.000000e+00	0.000000	1.379569e-09	7.034891e-10	0.000000	9.386992e-09	0.000000	0.503
8	2.828265e-01	0.199475	6.697645e-02	1.836979e-01	0.429644	2.637986e-01	0.102485	0.000(

9 rows × 12829 columns

In [34]:

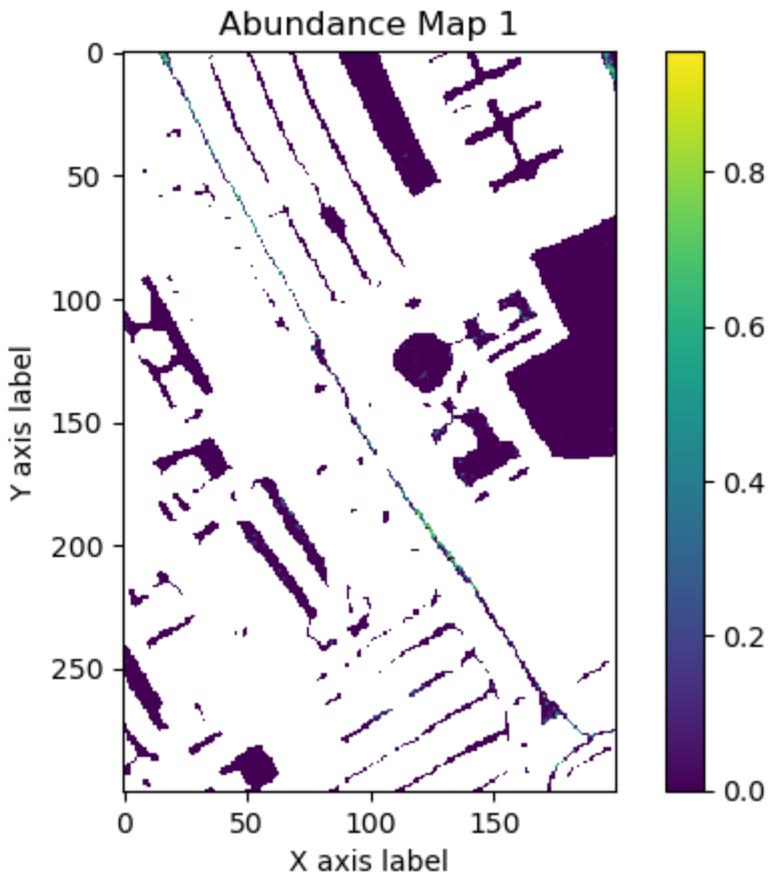
```
# Create a 200x300x9 array filled with zeros
result_array_e = np.zeros((300, 200, 9))

# Iterate over the pixels and fill in the values from the theta_est array
index = 0
for i in range(300):
    for j in range(200):
        if labels[i,j] != 0:
            result_array_e[i,j,:] = theta_est_lasso_df.iloc[:,index]
            index += 1
```

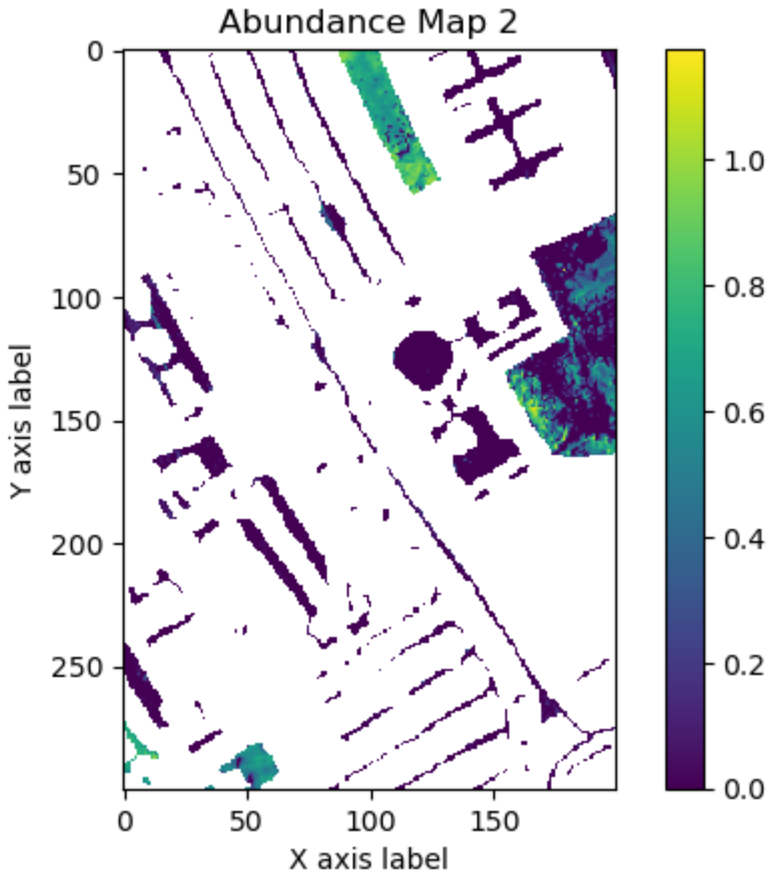
In [35]:

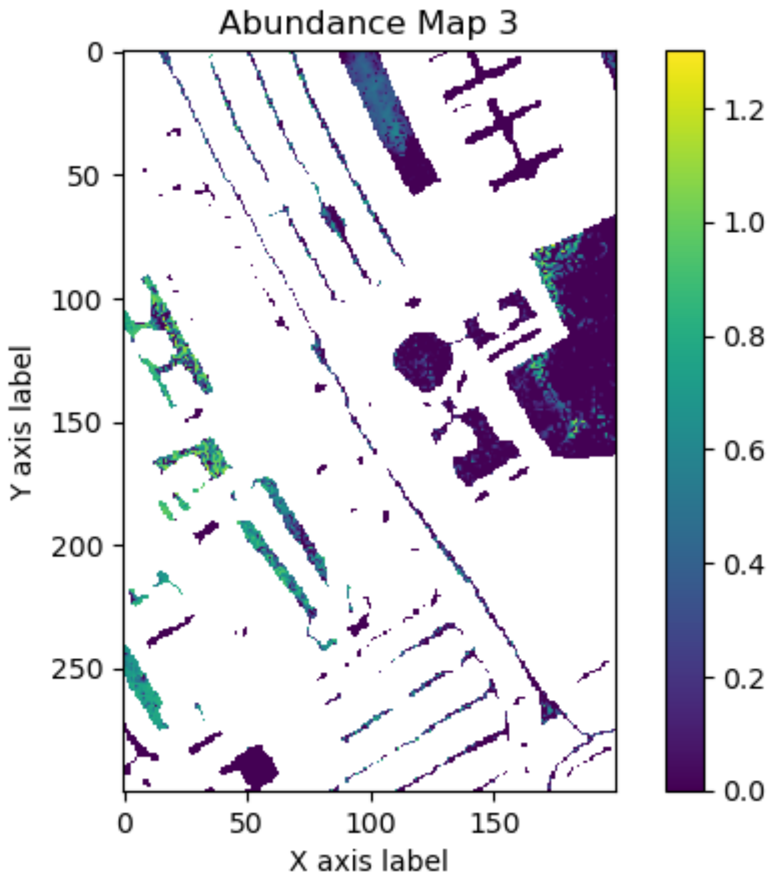
```
result_arrays_e = []

for k in range(9):
    result_arrays_e.append(result_array_e[:, :, k])

# Now result_arrays contains 9 2D arrays, each corresponding to a different k va
df_theta_e = []
for i in range(9):
    df_e = pd.DataFrame(result_arrays_e[i])
    df_theta_e.append(df_e)
```

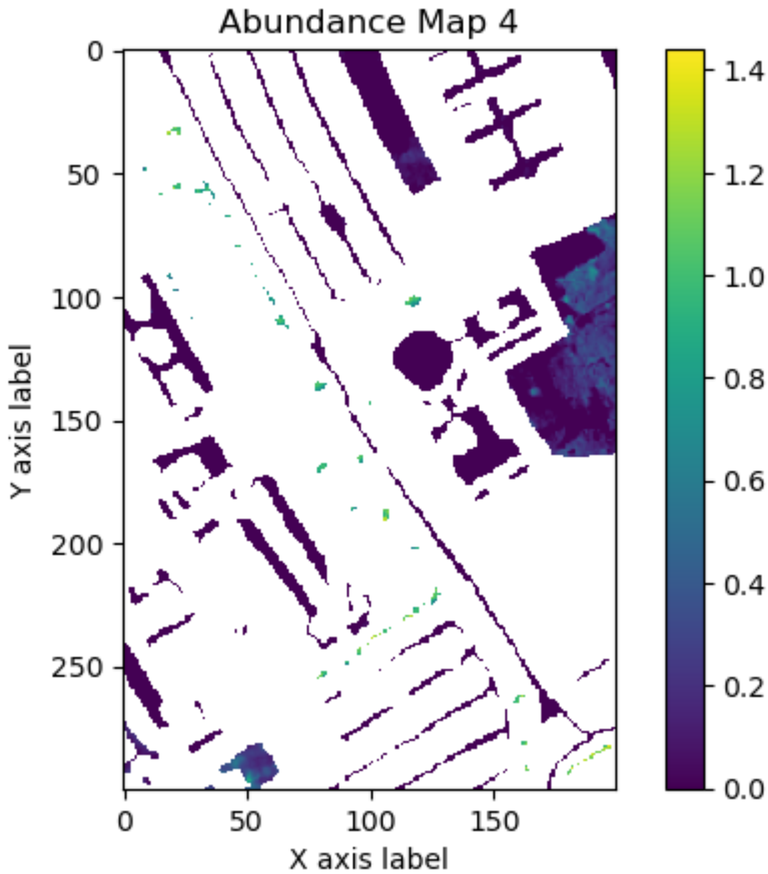
In [231…

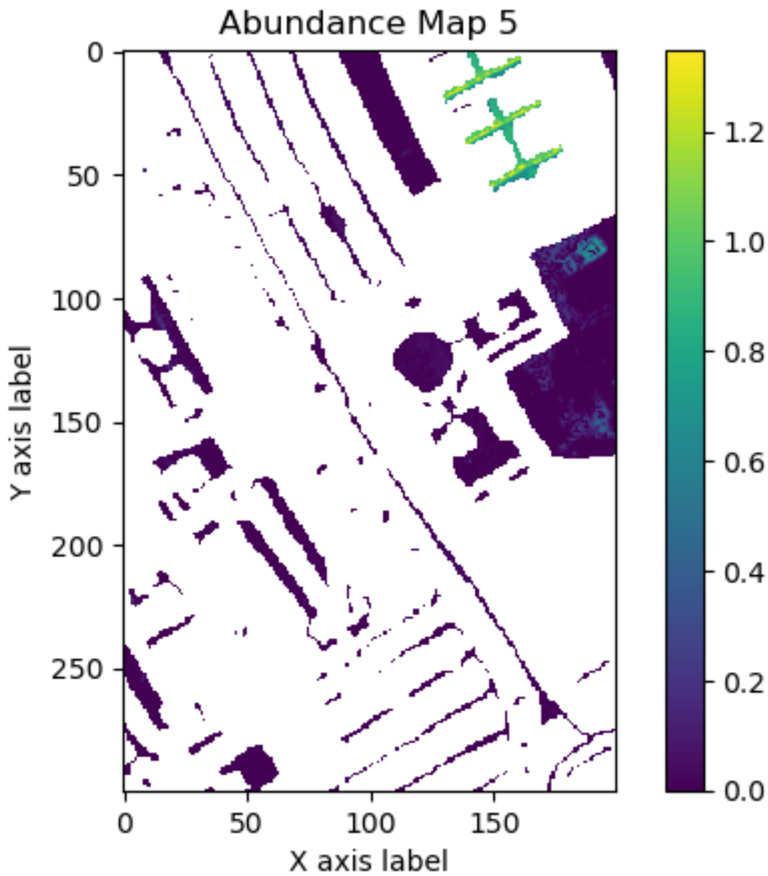
```
for i in range(9):
    # Create the heatmap using Matplotlib's imshow function
    fig, ax = plt.subplots()

    mask = df_theta[i] == 0

    # Set the colormap (cmap) to 'viridis' and set_bad to white
    cmap = plt.get_cmap('viridis')
    cmap.set_bad(color='white')

    # Apply the mask to the data
    im = ax.imshow(np.ma.masked_array(df_theta_e[i], mask))

    # Add a color bar
    cbar = ax.figure.colorbar(im, ax=ax)

    # Set axis labels
    ax.set_xlabel('X axis label')
    ax.set_ylabel('Y axis label')

    # Add title
    ax.set_title('Abundance Map {}'.format(i + 1))

    # Show the plot
    plt.show()
```

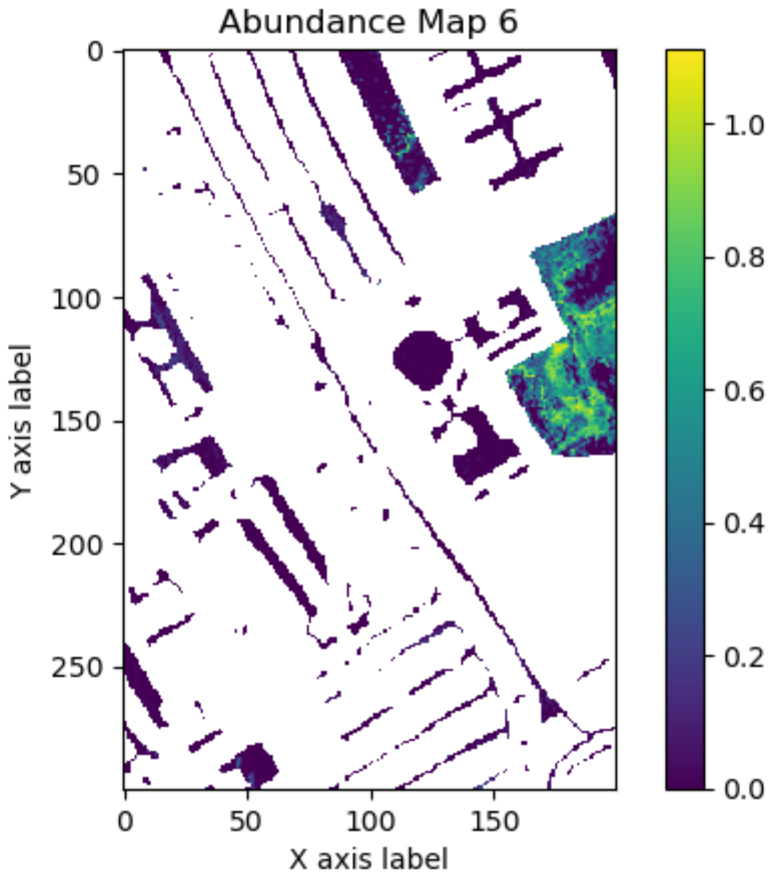


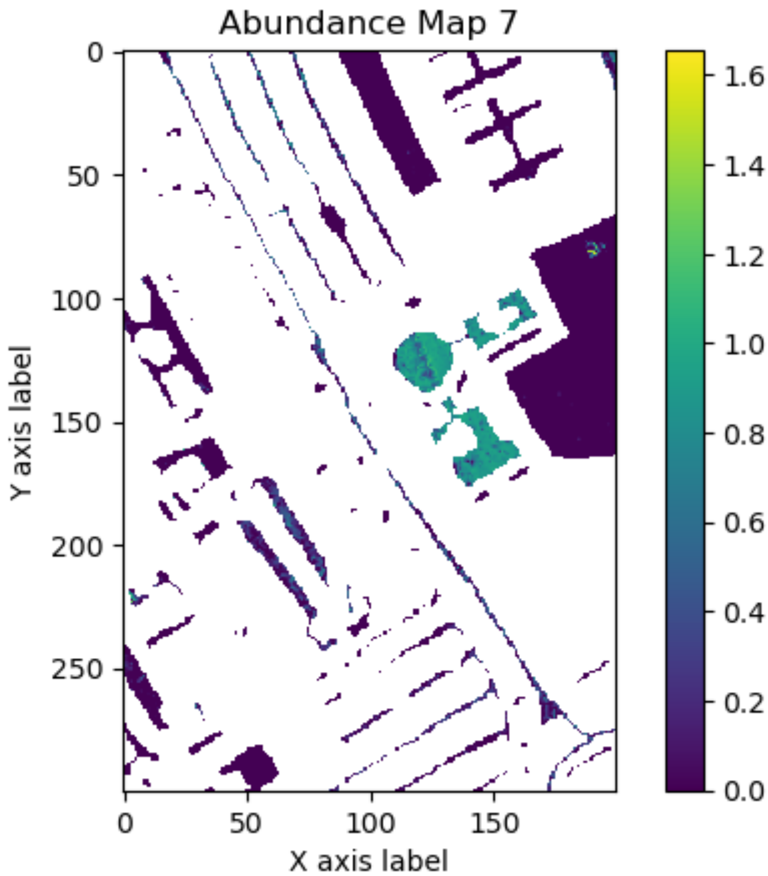

Abundance Map 2
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
1.0
0.8
0.6
0.4
0.2
0.0

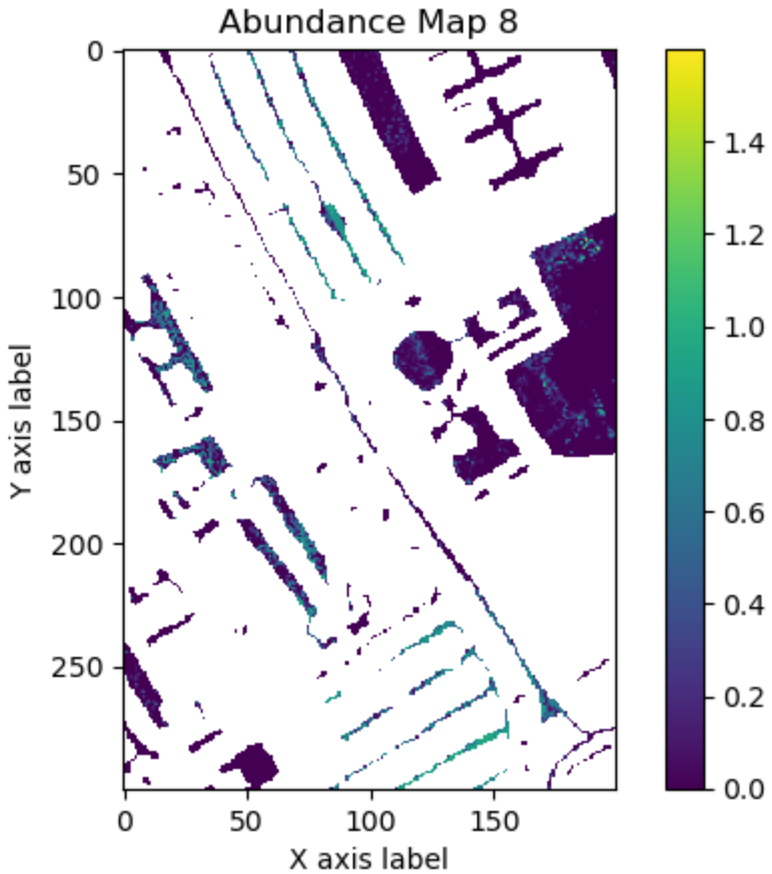

Abundance Map 3
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
1.2
1.0
0.8
0.6
0.4
0.2
0.0

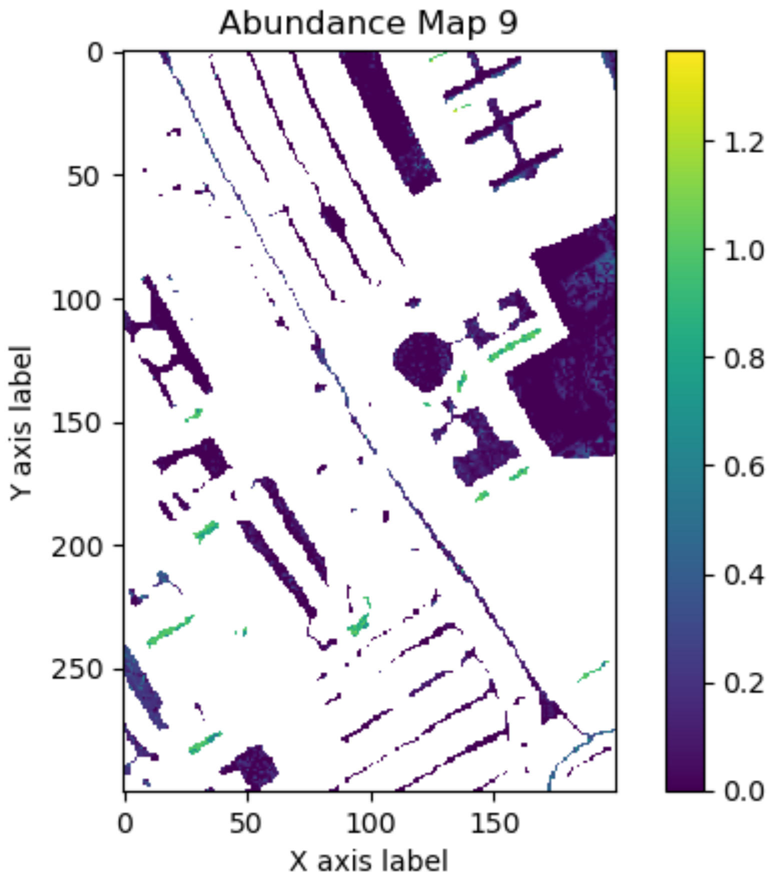

Abundance Map 4
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4


Abundance Map 5
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
0.0
0.2
0.4
0.6
0.8
1.0
1.2


Abundance Map 6
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
0.0
0.2
0.4
0.6
0.8
1.0


Abundance Map 7
Y axis label
X axis label
0
50
100
150
200
250
0
50
100
150
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6



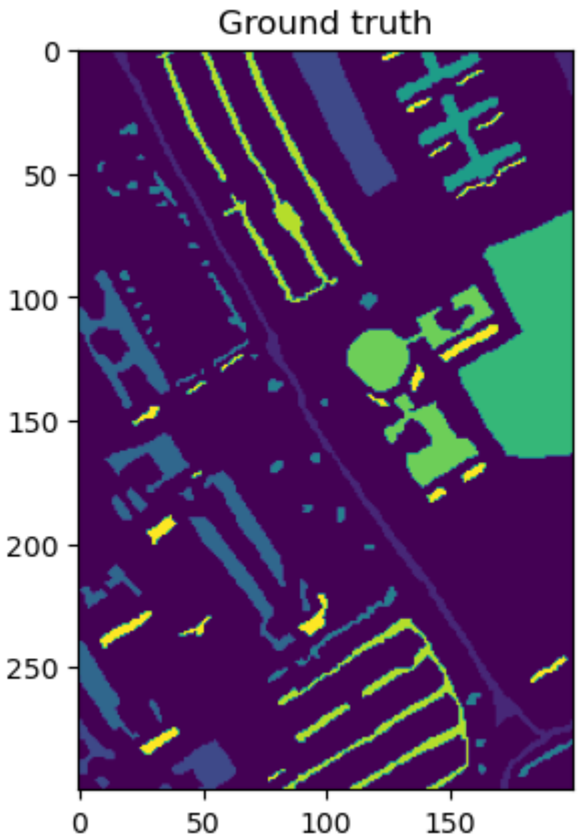


For lasso with alpha = 1 we obtain better results in the abundance maps (more distinct classes).

(B) Compare the results obtained from the above five methods (focusing on the abundance maps and the reconstruction error) and comment briefly on them (utilize the class information given in “PaviaU_ground_truth.mat”).

In [225…

```
#Plot the ground truth map
labels=ground_truth['y']
fig = plt.figure()
plt.imshow(labels)
plt.title('Ground truth')
plt.show()
```



Based on the reference map, we can observe 9 distinct classes to which each pixel is assigned. Our goal was to determine the contribution percentage (abundance) of each pure material in the formation of a given pixel. This was achieved by calculating θi values for each pixel using different methods. In the resulting abundance maps, pixels with higher θi values appear greener, signifying a greater probability that a specific material is present in that pixel.

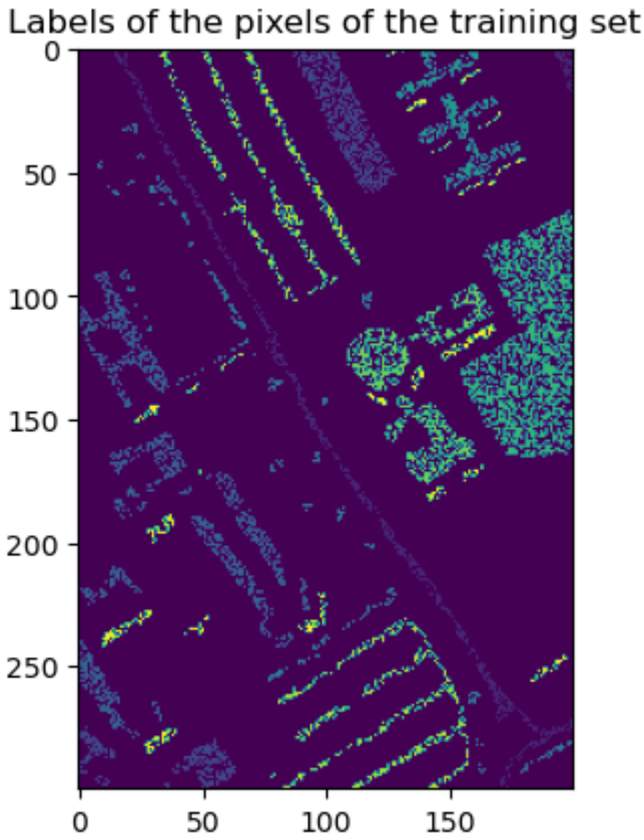
Among the five methods employed, the "Least Squares with both constraints" and "Least Squares imposing the non-negativity constraint," along with the "Lasso method for a =1 ", exhibit more pronounced regions. In these methods, pixels with green coloration are clearly defined and align well with the ground truth map. However, it's worth noting that these methods also show a higher reconstruction error compared to others, with the Lasso method having the highest reconstruction error value.

In the case of the first two methods, "Least Squares" and "Least Squares with sum to 1 constraint," many regions overlap, leading to a lack of discrete regions. This contrasts with the methods mentioned earlier, where distinct regions are more apparent.

PART 2 - CLASSIFICATION

In [38]:
```
from sklearn.preprocessing import normalize
import numpy as np
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics as m
from sklearn.model_selection import cross_val_score
import seaborn as sns
```

In [39]:
```
 # Trainining set for classification
Pavia_labels = sio.loadmat("C:\\Users\\melin\\Desktop\\Data Science\\Machine Lea
Training_Set = (np.reshape(Pavia_labels['training_set'],(200,300))).T
Test_Set = (np.reshape(Pavia_labels['test_set'],(200,300))).T
Operational_Set = (np.reshape(Pavia_labels['operational_set'],(200,300))).T
fig = plt.figure()
plt.imshow(Training_Set)
plt.title('Labels of the pixels of the training set')
plt.show()
```



In [ ]:
```
train_df=pd.DataFrame(Training_Set)
train_df
```

In [ ]:
```
test_df=pd.DataFrame(Test_Set)
test_df
```

In [ ]:
```
Operational_Set_df = pd.DataFrame(Operational_Set)
Operational_Set_df
```

In [278…
```
#Creation of X_train and y_train
X_train = []
y_train = []
for i in range(0,300):
    for j in range (0,200):
        if Training_Set[i,j] != 0 :
            X_train.append(HSI[i,j])
            y_train.append(Training_Set[i,j])
X_train=np.array(X_train)
y_train=np.array(y_train)
X_train.shape
y_train.shape
```

Out[278]:
```
(6415,)
```

In [279…

```
#Creation of X_test, y_test
X_test = []
y_test = []
for i in range(0,300):
    for j in range (0,200):
        if Test_Set[i,j] != 0 :
            X_test.append(HSI[i,j])
            y_test.append(Test_Set[i,j])
X_test=np.array(X_test)
y_test=np.array(y_test)
# X_test
y_test
```

Out[279]:

```
array([1, 1, 8, ..., 2, 8, 8], dtype=uint8)
```

Naive Bayes Classifier

In [58]:

```
from sklearn import metrics as m
from sklearn.naive_bayes import GaussianNB

from sklearn.model_selection import cross_val_score
# Build a Gaussian Classifier
model = GaussianNB()

# (i) Train it based on the training set performing 10-fold cross-validation
scores = cross_val_score(model, X_train, y_train, cv=10, scoring='accuracy')
mean_validation_error_NB = np.mean(1 - scores)  # error is 1 - accuracy
std_validation_error_NB = np.std(1 - scores)

# Print the results
print(f"Estimated Validation Error (Mean): {mean_validation_error_NB:.2f}")
print(f"Estimated Validation Error (Std): { std_validation_error_NB:.2f}")
```

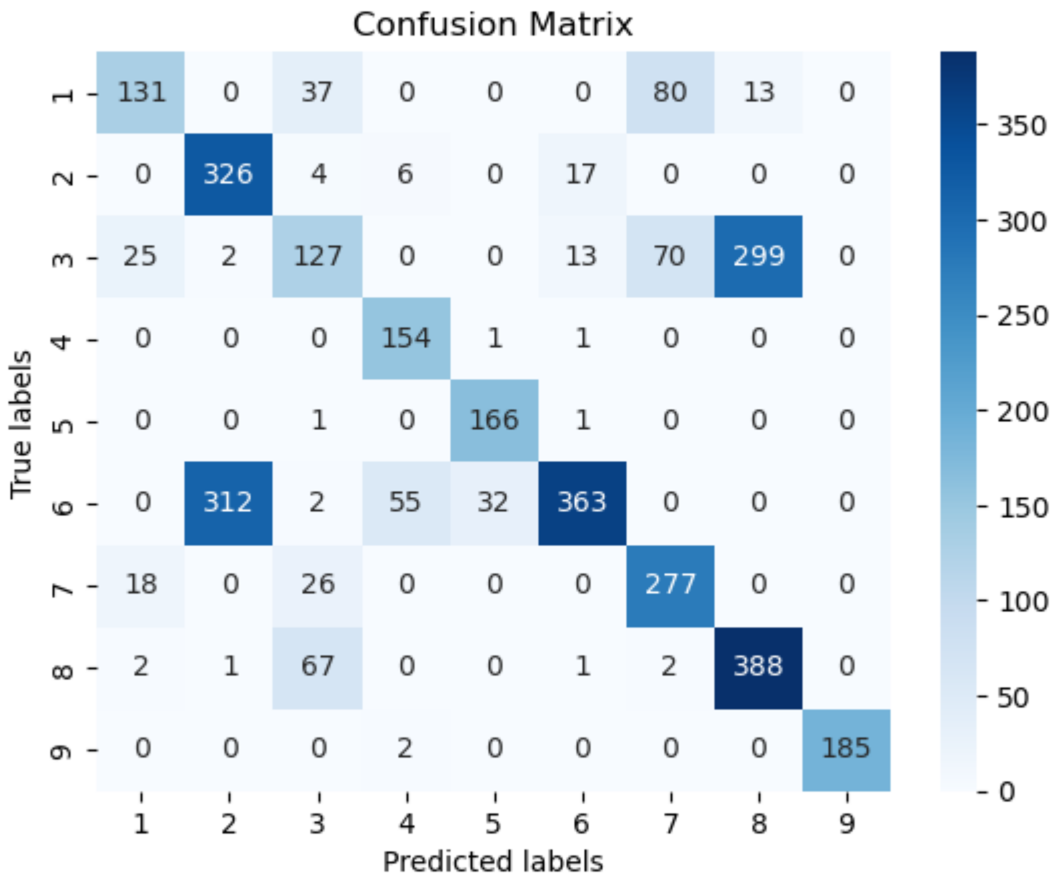
```
Estimated Validation Error (Mean): 0.36
Estimated Validation Error (Std): 0.06
```

In [59]:

```
# Model training
model.fit(X_train, y_train)
# Predict Output
y_pred_Bayes = model.predict(X_test)
res=m.classification_report(y_test, y_pred_Bayes)
print(res)
```

```
              precision    recall  f1-score   support

           1       0.74      0.50      0.60       261
           2       0.51      0.92      0.66       353
           3       0.48      0.24      0.32       536
           4       0.71      0.99      0.83       156
           5       0.83      0.99      0.90       168
           6       0.92      0.48      0.63       764
           7       0.65      0.86      0.74       321
           8       0.55      0.84      0.67       461
           9       1.00      0.99      0.99       187

    accuracy                           0.66      3207
   macro avg       0.71      0.76      0.70      3207
weighted avg       0.70      0.66      0.64      3207
```

In [71]:
```
from sklearn.metrics import confusion_matrix
confusion_mat_NB = confusion_matrix(y_test, y_pred_Bayes)

# Plot the confusion matrix
ax = plt.subplot()
sns.heatmap(confusion_mat_NB, annot=True, cmap='Blues', fmt='d', ax=ax)
labels = ['1', '2', '3', '4', '5','6','7','8','9']
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Confusion Matrix')
ax.xaxis.set_ticklabels(labels)
ax.yaxis.set_ticklabels(labels)
plt.show()
```



In [193…
```
# Identify classes that are not well separated (if any)
poorly_separated_classes_NB = [i for i in range(confusion_mat_NB.shape[0]) if co
print("Poorly Separated Classes:", poorly_separated_classes_NB)

# Compute success rate
success_rate_NB = np.sum(np.diag(confusion_mat_NB)) / np.sum(confusion_mat_NB)
print(f"Success Rate: {success_rate_NB:.2f}")
```

```
Poorly Separated Classes: []
Success Rate: 0.66
```

K-nearest Neighbor Classifier

Firstly, we are going to find the best value of k

In [249…
```
k_values = [i for i in range (1,20)]
scores = []
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    score = cross_val_score(knn, X_train, y_train, cv=10)
    scores.append(np.mean(score))

#Find the k value with the highest accuracy
best_k = k_values[np.argmax(scores)]
best_accuracy = scores[np.argmax(scores)]

print(f"The best k value is {best_k} with an accuracy of {best_accuracy:.4f}")
```

```
The best k value is 9 with an accuracy of 0.8523
```

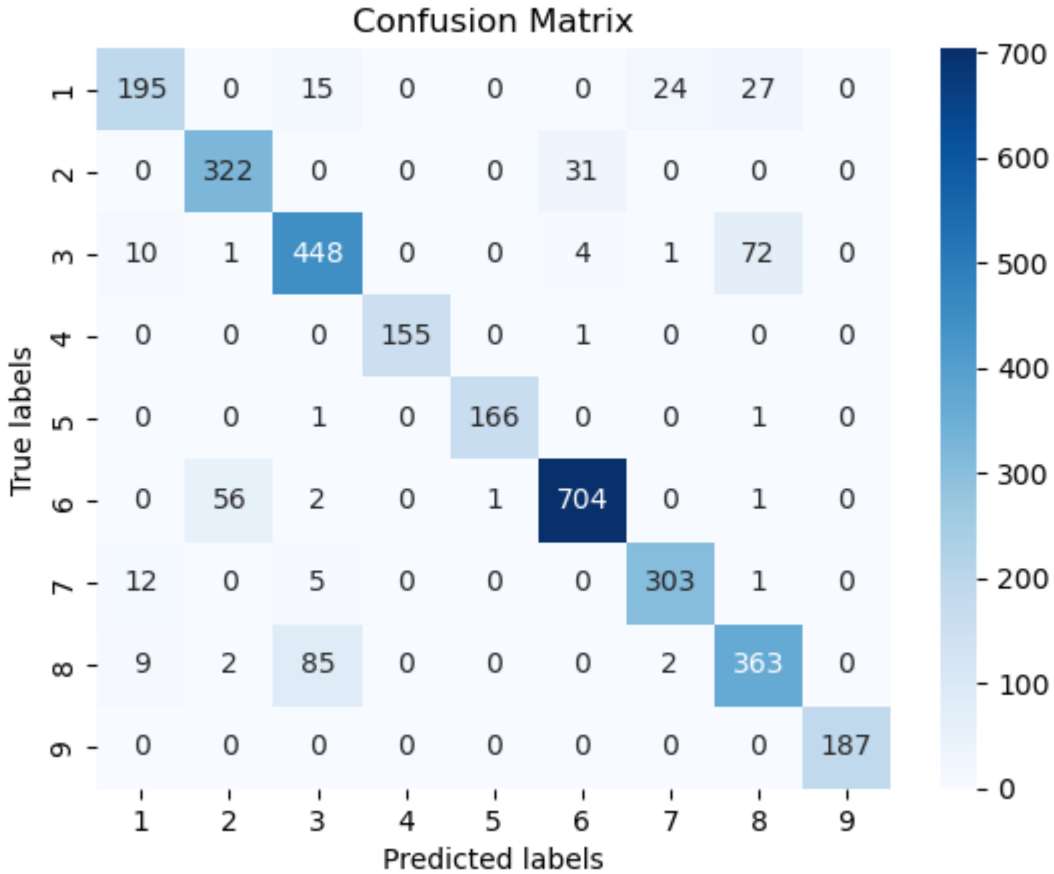
Now we take the k value with the highest accuracy (k=9)

In [245…
```
from sklearn.metrics import  accuracy_score

# Create a k-Nearest Neighbors Classifier (with the best k-value which is 9)
knn_classifier = KNeighborsClassifier(n_neighbors=9)

# (i) Train it based on the training set performing 10-fold cross-validation
scores = cross_val_score(knn_classifier, X_train, y_train, cv=10, scoring='accur
mean_validation_error_knn = np.mean(1 - scores)  # error is 1 - accuracy
std_validation_error_knn = np.std(1 - scores)

# Print the results
print(f"Estimated Validation Error (Mean): {mean_validation_error_knn:.2f}")
print(f"Estimated Validation Error (Std): {std_validation_error_knn:.2f}")

# (ii) Train on the whole training set and evaluate on the test set
knn_classifier.fit(X_train, y_train)

# Predict on the test set
y_pred_test = knn_classifier.predict(X_test)

# Predict Output
res=m.classification_report(y_test, y_pred_test)
print(res)
```

```
Estimated Validation Error (Mean): 0.15
Estimated Validation Error (Std): 0.05
              precision    recall  f1-score   support

           1       0.86      0.75      0.80       261
           2       0.85      0.91      0.88       353
           3       0.81      0.84      0.82       536
           4       1.00      0.99      1.00       156
           5       0.99      0.99      0.99       168
           6       0.95      0.92      0.94       764
           7       0.92      0.94      0.93       321
           8       0.78      0.79      0.78       461
           9       1.00      1.00      1.00       187

    accuracy                           0.89      3207
   macro avg       0.91      0.90      0.90      3207
weighted avg       0.89      0.89      0.89      3207
```

In [246…

```
confusion_mat_KNN = confusion_matrix(y_test, y_pred_test)

# Plot the confusion matrix
ax = plt.subplot()
sns.heatmap(confusion_mat_KNN, annot=True, cmap='Blues', fmt='d', ax=ax)
labels = ['1', '2', '3', '4', '5','6','7','8','9']
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Confusion Matrix')
ax.xaxis.set_ticklabels(labels)
ax.yaxis.set_ticklabels(labels)
plt.show()
```



In [247…

```
# Identify classes that are not well separated (if any)
poorly_separated_classes_knn = [i for i in range(confusion_mat_KNN.shape[0]) if

# Compute success rate
success_rate_knn = np.sum(np.diag(confusion_mat_KNN)) / np.sum(confusion_mat_KNN

# Print the results
print("Poorly Separated Classes:", poorly_separated_classes_knn)
print(f"Success Rate: {success_rate_knn:.2f}")
```

```
Poorly Separated Classes: []
Success Rate: 0.89
```

Minimum Euclidean Distance Classifier

In [271…

```
# First we divide the data into 10 folds
folds_X = np.array_split(X_train, 10)
folds_Y = np.array_split(y_train, 10)
```

In [272…

```
# Implementing 10-fold cross-validation
error_list = []
accuracy_list = []

for i in range(10):
    # Using the current fold as the test set and the rest as the training set
    test_X_fold = folds_X[i]
    test_Y_fold = folds_Y[i]

    train_X_fold = np.concatenate(folds_X[:i] + folds_X[i+1:])
    train_Y_fold = np.concatenate(folds_Y[:i] + folds_Y[i+1:])

    # Calculating the mean of each class in the training set
    class_means = {}
    for label in np.unique(train_Y_fold):
        class_samples = train_X_fold[train_Y_fold == label]
        class_means[label] = np.mean(class_samples, axis=0)

    # Classifying the test set using the minimum Euclidean distance
    predictions = []
    for sample in test_X_fold:
        distances = []
        for label, mean in class_means.items():
            distance = np.linalg.norm(sample - mean)
            distances.append((distance, label))
        prediction = min(distances)[1]
        predictions.append(prediction)

    # Calculating misclassification error and accuracy on the test set
    error = np.mean(predictions != test_Y_fold)
    error_list.append(error)

    accuracy = np.mean(predictions == test_Y_fold)
    accuracy_list.append(accuracy)

# print results
print("Average Accuracy: %0.2f with standard deviation: %0.2f" % (np.mean(accura
print("Average Error: %0.2f with standard deviation: %0.2f" % (np.mean(error_lis
```

```
Average Accuracy: 0.53 with standard deviation: 0.11
Average Error: 0.47 with standard deviation: 0.11
```

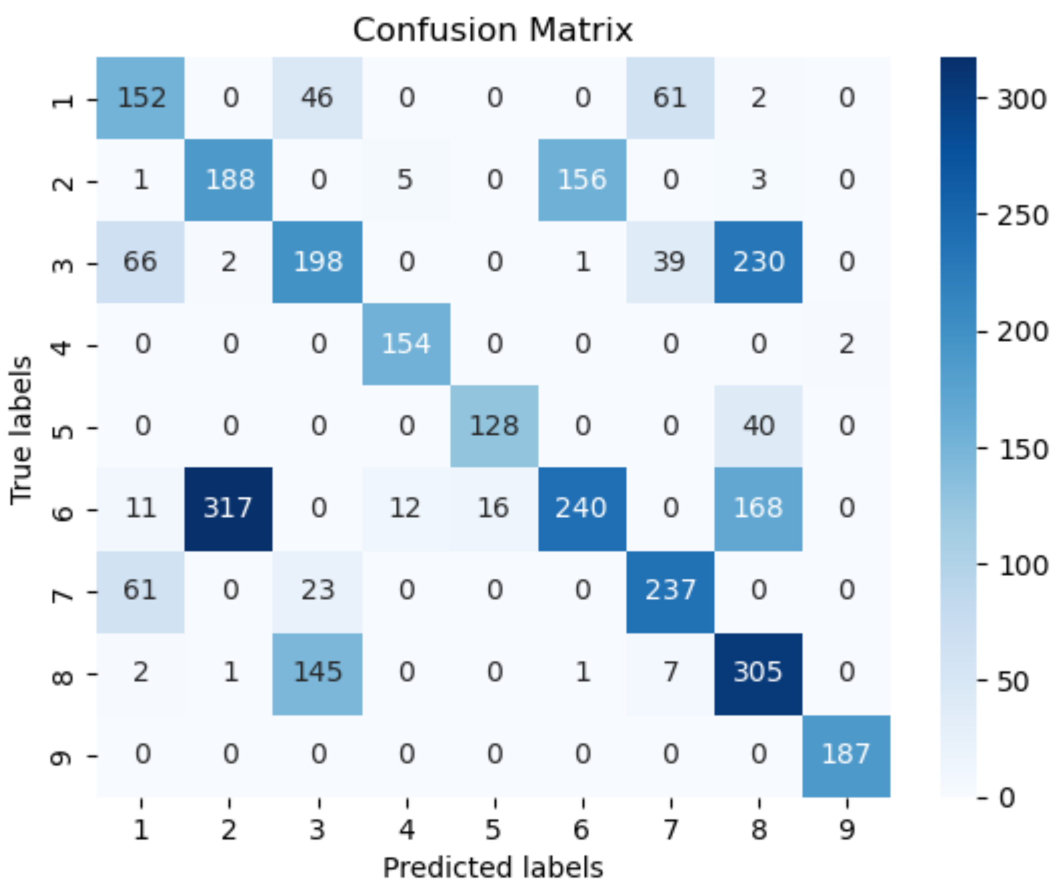
In [280…

```
# Calculating the mean of each class in the training set
class_means_dict = {}

for index, class_label in enumerate(np.unique(y_train)):
    class_samples_train = X_train[y_train == class_label]
    class_means_dict[class_label] = np.mean(class_samples_train, axis=0)

# For each test sample, compute its Euclidean distance to each class mean
predictions_list = []

for sample_index in range(len(X_test)):
    distances_list = []

    for class_label, mean_vector in class_means_dict.items():
        distance_to_mean = np.linalg.norm(X_test[sample_index] - mean_vector)
        distances_list.append((distance_to_mean, class_label))

    # Assign the test sample to the class with the smallest distance
    predicted_class = min(distances_list)[1]
    predictions_list.append(predicted_class)
```

In [282…
```
res=m.classification_report(y_test, predictions_list)
print(res)
```

```
              precision    recall  f1-score   support

           1       0.52      0.58      0.55       261
           2       0.37      0.53      0.44       353
           3       0.48      0.37      0.42       536
           4       0.90      0.99      0.94       156
           5       0.89      0.76      0.82       168
           6       0.60      0.31      0.41       764
           7       0.69      0.74      0.71       321
           8       0.41      0.66      0.50       461
           9       0.99      1.00      0.99       187

    accuracy                           0.56      3207
   macro avg       0.65      0.66      0.64      3207
weighted avg       0.58      0.56      0.55      3207
```

In [283…
```
confusion_mat_EMD = confusion_matrix(y_test, predictions_list)

# Plot the confusion matrix
ax = plt.subplot()
sns.heatmap(confusion_mat_EMD, annot=True, cmap='Blues', fmt='d', ax=ax)
labels = ['1', '2', '3', '4', '5','6','7','8','9']
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Confusion Matrix')
ax.xaxis.set_ticklabels(labels)
ax.yaxis.set_ticklabels(labels)
plt.show()
```



In [285…

```
# (i) Identify classes that are not well separated (if any)
poorly_separated_classes_EMD = [i for i in range(confusion_mat_EMD.shape[0]) if

# (ii) Compute success rate
success_rate_EMD = np.sum(np.diag(confusion_mat_EMD)) / np.sum(confusion_mat_EMD

print("Poorly Separated Classes (EMD):", poorly_separated_classes_EMD)
print(f"Success Rate (EMD): {success_rate_EMD:.2f}")
```

```
Poorly Separated Classes (EMD): []
Success Rate (EMD): 0.56
```

Bayesian classifier

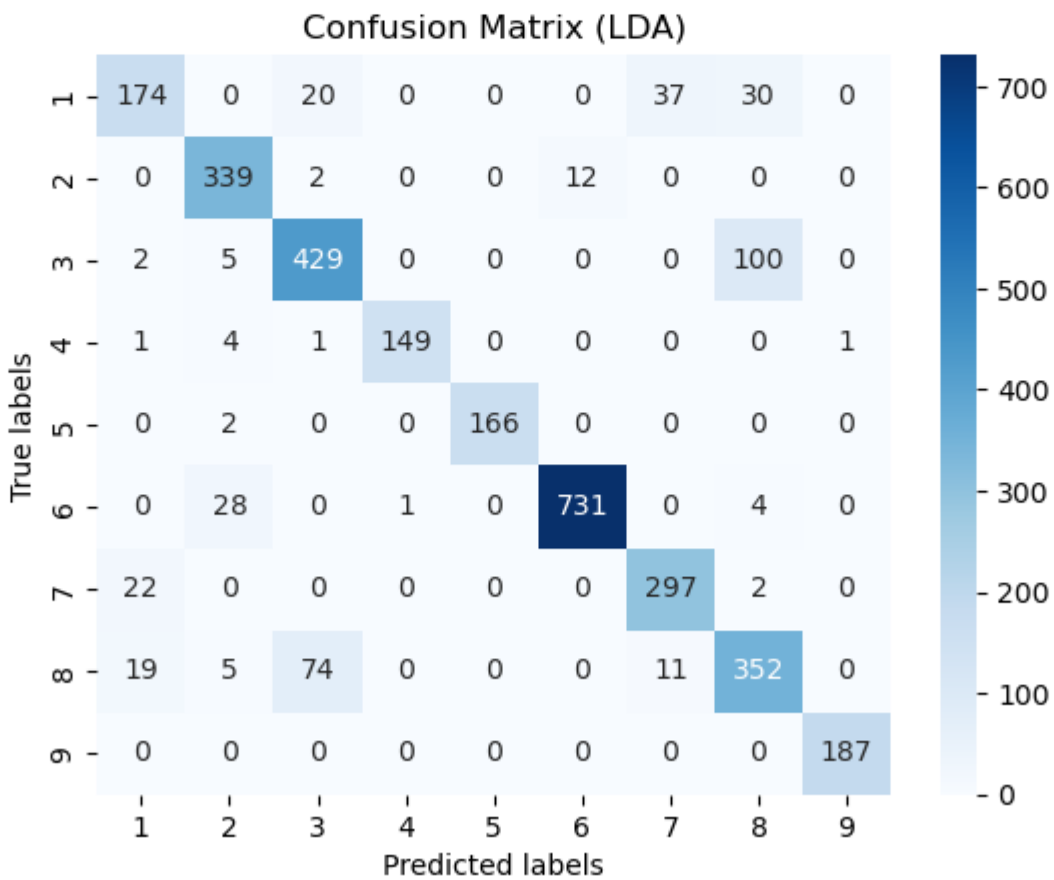
In [214…

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.model_selection import cross_val_score, KFold
from sklearn.metrics import classification_report

# Create a Linear Discriminant Analysis Classifier
lda_classifier = LinearDiscriminantAnalysis()

# (i) Train it based on the training set performing 10-fold cross-validation
k = 10
kf = KFold(n_splits=k, shuffle=True, random_state=42)

# Use cross_val_score to perform k-fold cross-validation and obtain accuracy sco
scores = cross_val_score(lda_classifier, X_train, y_train, cv=kf, scoring='accur
mean_validation_error_lda = 1 - np.mean(scores)  # error is 1 - accuracy
std_validation_error_lda = np.std(1 - scores)

# Print the results
print(f"Estimated Validation Error (Mean): {mean_validation_error_lda:.2f}")
print(f"Estimated Validation Error (Std): {std_validation_error_lda:.2f}")

# (ii) Train on the whole training set and evaluate on the test set
lda_classifier.fit(X_train, y_train)

# Predict on the test set
y_pred_test_lda = lda_classifier.predict(X_test)

# Evaluate the classifier on the test set
print("Classification Report on Test Set:")
print(classification_report(y_test, y_pred_test_lda))
```

```
Estimated Validation Error (Mean): 0.12
Estimated Validation Error (Std): 0.01
Classification Report on Test Set:
              precision    recall  f1-score   support

           1       0.80      0.67      0.73       261
           2       0.89      0.96      0.92       353
           3       0.82      0.80      0.81       536
           4       0.99      0.96      0.97       156
           5       1.00      0.99      0.99       168
           6       0.98      0.96      0.97       764
           7       0.86      0.93      0.89       321
           8       0.72      0.76      0.74       461
           9       0.99      1.00      1.00       187

    accuracy                           0.88      3207
   macro avg       0.89      0.89      0.89      3207
weighted avg       0.88      0.88      0.88      3207
```

In [215…

```python
# Create the confusion matrix
confusion_mat_lda = confusion_matrix(y_test, y_pred_test_lda)

# Plot the confusion matrix
ax = plt.subplot()
sns.heatmap(confusion_mat_lda, annot=True, cmap='Blues', fmt='d', ax=ax)
labels = ['1', '2', '3', '4', '5', '6', '7', '8', '9']
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Confusion Matrix (LDA)')
ax.xaxis.set_ticklabels(labels)
ax.yaxis.set_ticklabels(labels)
plt.show()
```



In [216...

```
# (i) Identify classes that are not well separated (if any) for LDA
poorly_separated_classes_lda = [i for i in range(confusion_mat_lda.shape[0]) if

# (ii) Compute success rate for LDA
success_rate_lda = np.sum(np.diag(confusion_mat_lda)) / np.sum(confusion_mat_lda

# (iii) Print the results for LDA
print("Poorly Separated Classes (LDA):", poorly_separated_classes_lda)
print(f"Success Rate (LDA): {success_rate_lda:.2f}")
```

```
Poorly Separated Classes (LDA): []
Success Rate (LDA): 0.88
```

Compare the results of the classifiers and comment on them

From the success rate of each classifier we can deduce that the best classifier is the KNN classifier (89% success rate) , followed by the Bayes classifier (88%) , then the Naive Bayes classifier (66%) and lastly the minimum euclidean distance classifier (56%). From the confusion matrices of our two best classifiers (knn and Bayes classifiers) we can see that there is a confusion between classes 3 and 8. Based on the Bayes classifier 100 labels are missclasified as 8 (true label = 3) from total of 536. So the missclasification error for class 3 is 18.66%. Also, 74 labels are misslasified as 3 (true label=8) from total of 461. So the missclasification error for class 8 is 16%. Based on knn classifier, 72 labels are missclasified as 8 from total of 536(true label=3). So the misclassification error for class 3 is 13.43%. Also, 85 labels are misclassified as 3 (true label= 8) from total of 461. So the missclasification error for class 8 is : 18.44 %. Furthermore, there are instances of confusion among additional classes, although the degree of confusion is not as pronounced as that observed between classes 3 and 8.

PART 3 - COMBINATION

Comment briefly on the possible correlation of the results obtained from the spectral unmixing procedure with those obtained from classification.

We observe a noticeable correlation between the outcomes derived from the spectral unmixing procedure and those obtained through classification. Specifically, employing the best unmixing methods, such as Least Squares with both constraints and Non-Negativity Constraints, reveals distinctly separated classes, indicated by a prominent green coloration. Similarly, our top-performing classifiers, namely k-Nearest Neighbors (knn) and Bayes, exhibit nearly diagonal confusion matrices, implying accurate assignment of the majority of pixels to their true classes. In spectral unmixing, higher values of θ_i signify a greater likelihood of assigning a specific pixel to class i, mirroring the approach taken in the classification method. Both methods demonstrate effective performance. Additionally, our classification method reveals confusion between classes 3 and 8, a phenomenon corroborated by the abundance maps. Notably, some pixels with true label 3 are colored green in abundance map 8, and vice versa, underscoring the intricacies of class distinctions in both methodologies.