Project_AUEB

December 8, 2023

1 Hyperspectral Imaging

1.1 Machine Learning and Computational Statistics, MSc in Data Science (2023/2024)

1.1.1 Project Assignment

Dimitris Tsirmpas MSc in Data Science f3352315 Athens University of Economics and Business

Disclaimer: Most documentation on the written code has been adapted from ChatGPT.

[1]:
```
from time import time

start = time()
```

1.2 Exploratory Analysis

1.2.1 Importing the Data

We begin by importing the data. Our data is split in two files:

- `PaviaU_cube.mat`, which depicts the pixels for each image (300x200 resolution) for each of the 103 spectral bands
- `PaviaU_ground_truth.mat` which contains the class labels (materials) for each pixel in the image.

[2]:
```
import scipy.io as sio
import numpy as np
import scipy.optimize
import matplotlib.pyplot as plt


pavia = sio.loadmat('data/PaviaU_cube.mat')
hsi = pavia['X'] #Pavia HSI : 300x200x103

ends = sio.loadmat('data/PaviaU_endmembers.mat') # Endmember's matrix: 103x9
endmembers = ends["endmembers"]
```

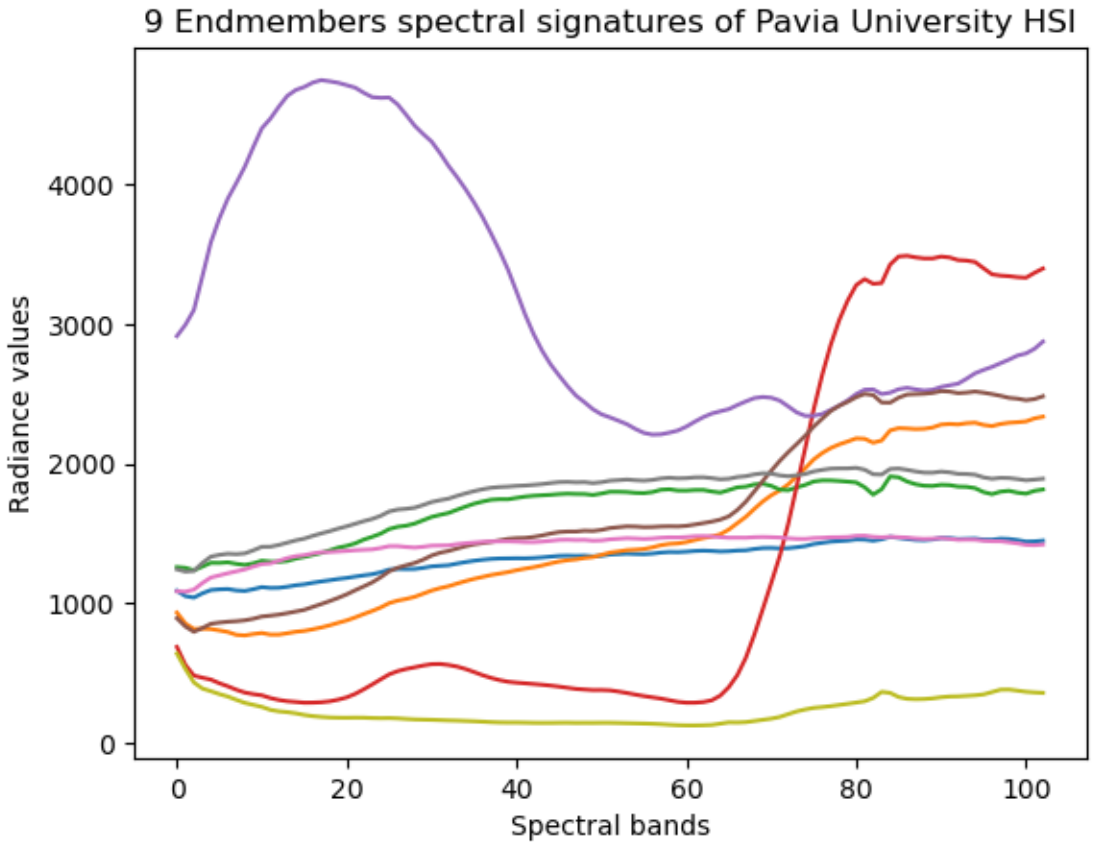
[3]:
```
#Perform unmixing for the pixels corresponding to nonzero labels
ground_truth = sio.loadmat('data/PaviaU_ground_truth.mat')
labels = ground_truth['y']
labels.shape
```

[3]:
```
(300, 200)
```

1.2.2 Data Visualization

Each of the 9 materials contains different values for each spectral band. We can see those values in the plot below:

[4]:
```
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()
```



These 9 endmembers represent 9 classes of materials, which can be seen in the dictionary below:

[5]:
```
class_names = {
    0: "Mixed",
    1: "Water",
    2: "Trees",
    3: "Asphalt",
    4: "Self-Blocking Bricks",
    5: "Bitumen",
    6: "Tiles",
    7: "Shadows",
    8: "Meadows",
    9: "Bare Soil"
}
```

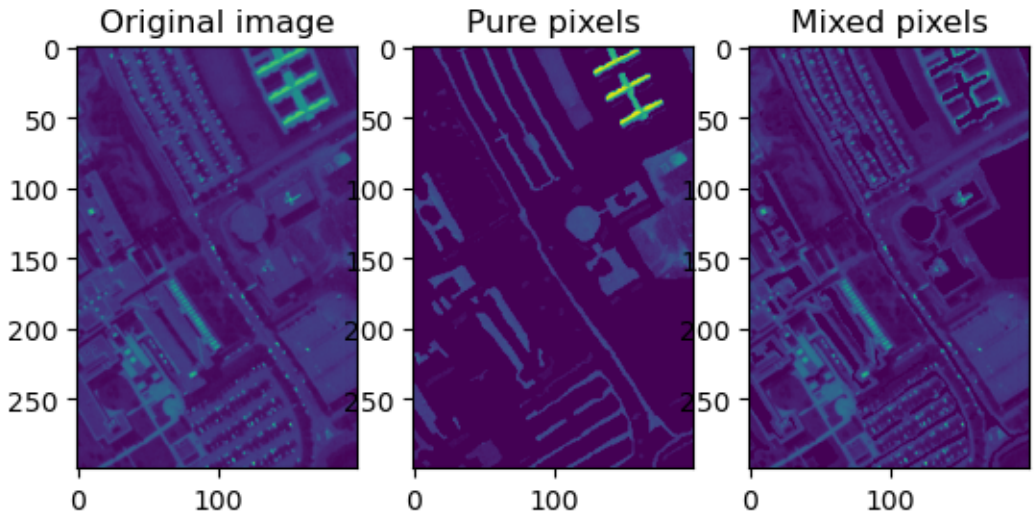
For this analysis, we need to only keep pure pixels, aka pixels that corresponds to geographical areas with a single geological feature. We thus split our dataset into "classless" aka mixed pixels, and pixels with defined class labels (pure pixels).

[6]:
```
mask = labels != 0
pure_hsi = np.where(mask[:,:, None], hsi, 0)
```

[7]:
```
mask = labels == 0
mixed_hsi = np.where(mask[:,:, None] , hsi, 0)
```

Since our data represent an image we can also show the image for a specific spectral band

[8]:
```
fig, axes = plt.subplots(1, 3)

axes[0].imshow(hsi[:,:,10])
axes[0].set_title("Original image")

axes[1].imshow(pure_hsi[:,:,10])
axes[1].set_title('Pure pixels')

axes[2].imshow(mixed_hsi[:,:,10])
axes[2].set_title('Mixed pixels')

fig.suptitle('RGB Visualization of Pavia University (10th spectral band)')
plt.show()
```

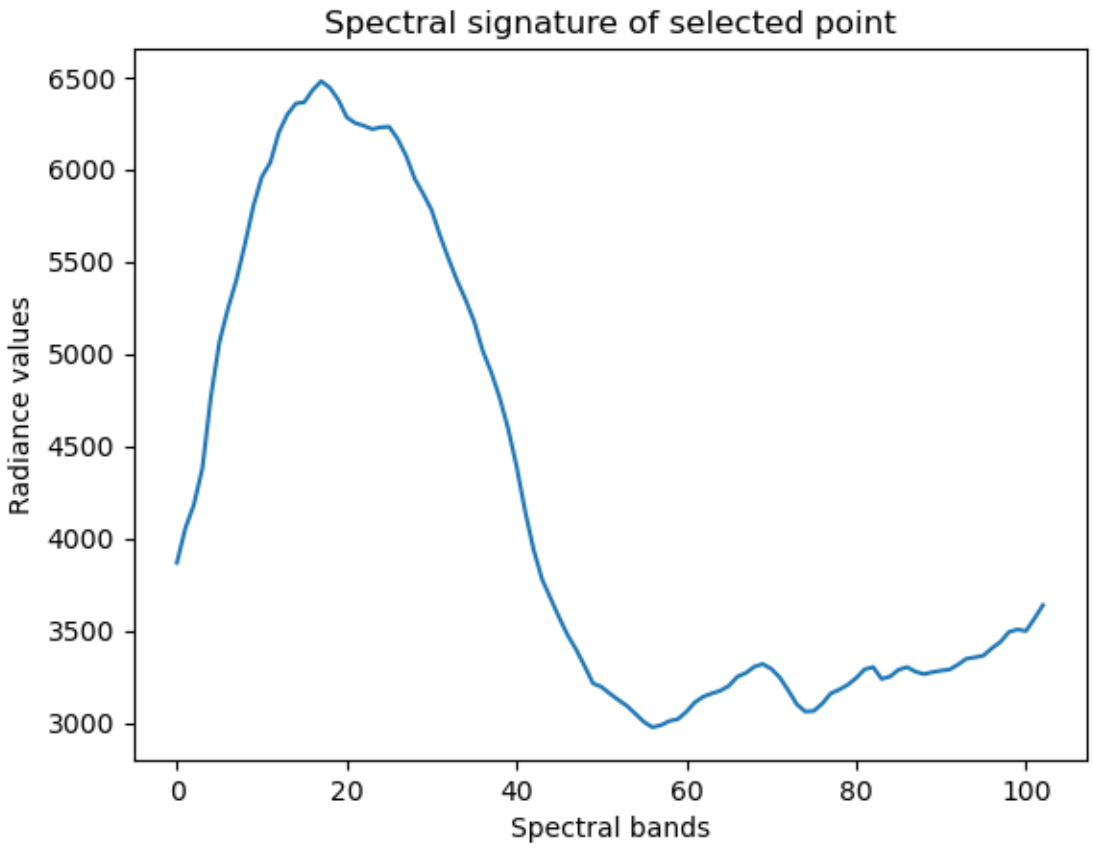


For demonstration purposes, let's select a pure pixel from the photograph. We select one of the pixels from the top right, from what looks like a building. This area of the photograph appears to have many continuous pure pixels, as seen by the Figure above.

[9]:
```
focus_point = (32, 147)
print(f"Focus point is made of {class_names[labels[focus_point]]}")
```

```
Focus point is made of Bitumen
```

[10]:
```
plt.plot(hsi[focus_point])
plt.ylabel('Radiance values')
plt.xlabel('Spectral bands')
plt.title("Spectral signature of selected point")
plt.show()
```



Notice how similar this spectral signature is to the one defined in the graph above for pixels containing bitumen material.

We could also attempt to make a qqnorm plot for each material on the 1st spectral band:

1.2.3 Assuming Normality

One of the most important assumptions in Linear Regression models is the assumption of normality. There are three ways of verifying this assumption:

- Running a normality test (such as Shapiro or Jarque-Bera test) on each class
- Building qqnorm plots to graphically gauge the normality of each class
- Assume with a large degree of certainty that the distributions of our classes are very likely to approximate the Normal Distribution under the Central Limit Theorem
- Run the model anyway and assume the distributions are "normal-like" enough if it gives us good results

The first option is unattractive, since normality tests are normally "strong" tests, meaning that with a sufficiently large sample size, the test will almost always assume the distribution is not normal.

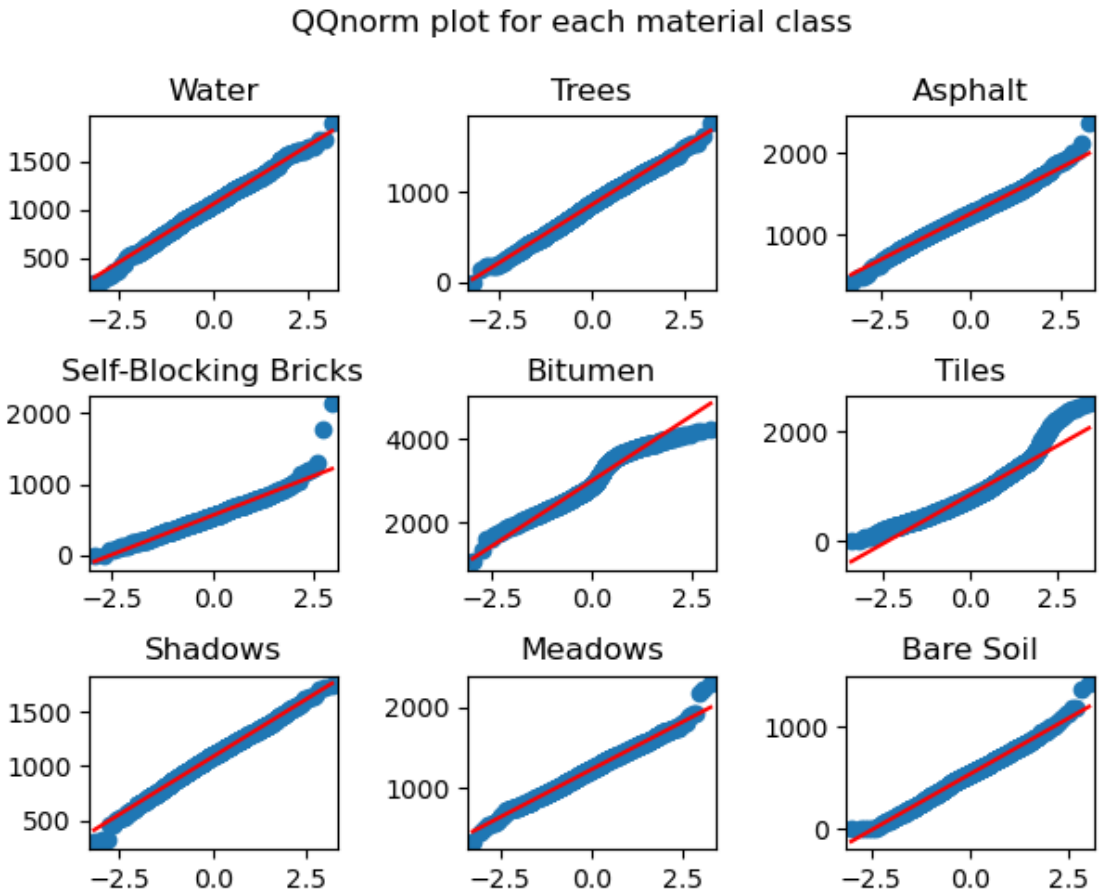
The third option can be defended by the size of our data, since each class features a very large number of points ($n_j >> 30, \forall j \in [1, 9]$) (where n_j represents the count of the jth class):

[11]:
```python
def split_by_label(data, labels):
    split_arrays = {}

    # Iterate over unique labels and split the array
    for label in np.unique(labels):
        mask = labels == label   # Create a boolean mask for the current label
        split_arrays[label] = data[mask]
    return split_arrays


split_classes = split_by_label(pure_hsi, labels)
for label, array in split_classes.items():
    print(f"Class {class_names[label]}, #Points={len(array)} ")
```

```
Class Mixed, #Points=47171
Class Water, #Points=1108
Class Trees, #Points=1377
Class Asphalt, #Points=2099
Class Self-Blocking Bricks, #Points=609
Class Bitumen, #Points=698
Class Tiles, #Points=3042
Class Shadows, #Points=1330
Class Meadows, #Points=1795
Class Bare Soil, #Points=771
```

And a snippet of the 2nd solution can be seen below:

[12]:
```python
from statsmodels.graphics.gofplots import qqplot


fig, axes = plt.subplots(3, 3)
fig.tight_layout(pad=2.0)
fig.suptitle("QQnorm plot for each material class", y=1.05)

for i, (label_idx, array) in enumerate(split_classes.items()):
    if i != 0:
        idx = i - 1
        ax = axes[idx//3, idx%3]
        qqplot(array[:,1], line="s", ax=ax)
        ax.set_title(class_names[label_idx])
        ax.set_ylabel("")
        ax.set_xlabel("")

plt.show()
```



Note that the 1st spectral band for all materials but Bitumen and Tiles seem to follow the normal line almost perfectly.

We could repeat this procedure for each of the 103 spectral bands, but for now, this plot does reassure us that the size of our dataset probably is sufficient to assume normality.

1.3 Spectral Unmixing

According to the *linear spectral unmixing hypothesis*, we can assume that every pixel (y) in our image is a linear combination of the various spectral signatures (X), where each spectral signature contributes a certrain percentage (θ). Since all percentages must sum to one, this would make θ a *probability distribution function* (pdf), a property which would greately help in interpreting our results.

Under this assumption, the generating distribution for each pixel would be $y = X\theta + \eta, \eta \sim N(0, \sigma^2)$, where η represents white noise. This model would be under the following constraints: - No intercept (which would normally indicate a set % of an "other" material for all pixels) - $\theta_i > 0, \forall i$ - $\sum_i \theta_i = 1$

We will be considering 4 models, each one following different constraints. Specifically, these will be: 1. An OLS model with no constraints 2. An OLS model with no intercept 3. An OLS model with

no intercept and positive coefficients 4. An OLS model with no intercept and positive coefficients which sum to one 5. A LASSO model with no intercept and positive coefficients

Notice that out of the 5 models, only #4 satisfies the prerequisitives of a pdf.

[13]:
```
from prettytable import PrettyTable


def arrays_stats(arrays: list[np.ndarray], names: list[str]|None = None,␣
↪decimals: int = 2) -> PrettyTable:
    """
    Generate statistics for a list of NumPy arrays and create a PrettyTable.

    :param arrays: List of NumPy arrays for which statistics are calculated.
    :type arrays: list[np.ndarray]

    :param names: List of names corresponding to each array. If None, default␣
↪names will be generated based on array indices. Default is None.
    :type names: list[str]|None

    :param decimals: Number of decimals to round the statistical values.␣
↪Default is 2.
    :type decimals: int

    :return: Table containing statistics for each array, including mean,␣
↪standard deviation, minimum, maximum, first quartile (Q1), and third␣
↪quartile (Q3).
    :rtype: PrettyTable
    """
    stats_table = PrettyTable(["Array", "Shape", "Mean", "Std", "Min", "Max",␣
↪"Q1", "Q3"])
    stats_table.float_format = "."+ str(decimals)

    if type(decimals) is not int:
        raise ValueError("The 'decimals' argument must be an integer.")

    if names is None:
        names = [i for i in range(len(arrays))]

    if len(names) != len(arrays):
        raise ValueError("Names and arrays must have the same length.")

    for name, array in zip(names, arrays):
        stats_table.add_row([name, array.shape, array.mean(),  array.std(),
                             array.min(), array.max(),
                             np.quantile(array, 0.25), np.quantile(array, 0.
↪75)])
```

```
    return stats_table
```

[14]:
```
# our analysis only concerns pure pixels
mask = labels != 0
y1 = pure_hsi.flatten()[pure_hsi[mask]].T
x1 = endmembers

y1.shape, x1.shape
```

[14]:
```
((103, 12829), (103, 9))
```

[15]:
```
arrays_stats([y1, x1], names=["y1", "x1"])
```

[15]:
```
+-------+--------------+---------+--------+--------+---------+---------+--------
-+
| Array |    Shape     |   Mean  |  Std   |  Min   |   Max   |    Q1   |    Q3
|
+-------+--------------+---------+--------+--------+---------+---------+--------
-+
|   y1  | (103, 12829) |  530.14 | 687.91 |   0    |   2266  |   0.00  | 1369.00
|
|   x1  |   (103, 9)   | 1551.32 | 926.43 | 128.24 | 4745.65 | 1093.66 | 1897.79
|
+-------+--------------+---------+--------+--------+---------+---------+--------
-+
```

We will also define some convinience functions to analyze the results of each model:

[16]:
```
def reconstruction_error(y: np.ndarray, X: np.ndarray,
                         abundance_map: np.ndarray) -> float:
    """
    Get the reconstruction error for the pure pixels in the image.
    Essentially a Mean Squares Error wrapper for the problem's domain.
    """
    theta = abundance_map
    diff = (y - X @ theta.T)
    # mean of l2 norm squared
    return np.mean(diff.T @ diff)
```

We will now define and run our models. We will then view and discuss the results in the next section.

1.3.1 Ordinary Least Squares

As stated in the assignment, we will be using our own implementation of the Least Squares regressor for the no-constraint task:

[17]:
```
class LS_Model:
    """
```

```
    A class representing a linear regression model, with a X data matrix and a␣
↪y value vector.
    """

    # Pretend there is a return type hint here
    # if we declare it we somehow reach a cyclical dependency, where the type␣
↪hint is declared before the class
    @classmethod
    def from_normal_dataset(cls, mean: np.ndarray, cov: np.ndarray, size: int):
        """
        Generate a new LS_Model based on a datast following the Normal␣
↪distribution with a set mean and covariance matrix.
        :param mean: a Mx1 vector of the r.v's mean (where M the number of␣
↪features)
        :param cov: a MxM covariance matrix (where M the number of features)
        :param size: the size of the LS_Model
        :return: a LS_Model with Nx1 explanatory variable vector and a Nx(M+1)␣
↪data vector, where the first row is filled with ones
        """
        data = np.random.multivariate_normal(mean=mean, cov=cov, size=size)
        return LS_Model(y=data[:,0],  x=data[:,1], include_coef_col=True)

    def __init__(this, y: np.ndarray, x: np.ndarray, ridge: float = 0,␣
↪include_coef_col: bool = False):
        """
        Construct a new model with the associated X data matrix and y values.
        :param x: a NxM data matrix
        :param y: a Nx1 value vector
        :param ridge: the ridge regression term, 0 for ordinary least squares␣
↪(default 0)
        :param include_coef_col: whether to add an all-ones column in the X␣
↪matrix to support estimation
        of the intercept, default False
        """
        if include_coef_col:
            this.x = np.column_stack([np.ones(x.shape[0]), x])
        else:
            this.x = x

        this.ridge = ridge
        this.y = y

    def fit(this, lamda: float = 0) -> np.array:
        """
        Perform linear regression using the sum of error squares criterion.
        :return: the theta vector of size Mx1 containing the model parameters
```

```
        """
        X = this.x
        y = this.y

        if this.ridge < 0:
            raise ValueError("Lamda values must be equal to, or larger than 0.")

        # if X is a vector and not matrix, turn it into a matrix as
        # vector @ vector => float, but matrix @ vector => vector
        if X.ndim < 2:
            X = X.reshape(-1, 1)

        inverted = X.T@X + this.ridge * np.identity(X.shape[1]) # ridge␣
↪regression term
        if np.linalg.det(inverted) == 0:
            print("Warning: X^TX not invertible, solution is not unique")

        # mutiply the inverse of X^T with X^T and y
        # the result is our theta vector
        return np.linalg.inv(inverted) @ X.T @ y

    def estimate(this, theta: np.ndarray=None) -> np.ndarray:
        """
        Get the LS estimation for the LS_Model's X matrix based on generated␣
↪weights.
        :param theta: a Mx1 vector of the LS model's parameters (where M the␣
↪number of features)
        None to calculate the parameters automatically. Default: None.
        :return: a Nx1 vector containing the y_hat estimations
        """
        if theta is None:
            theta = this.fit()
        return this.x @ theta

    def mse(this, estimates: np.ndarray=None) -> float:
        """
        Get the MSE for the current model. This is a wrapper method for the MSE␣
↪free function.
        :param estimates: a Nx1 vector containing the y_hat estimations,
        None to calculate estimations automatically. Default: None.
        :return: the MSE of the model
        """
        if estimates is None:
            estimates = this.estimate(this.fit())
        return mse(this.y, estimates)
```

```
def mse(y_hat: np.array, y: np.array) -> float:
    """
    Calculate the Mean Squared Error between the predictions and actual values.
    :param y_hat: A Nx1 prediction vector
    :param y: A Nx1 vector containing the actual values
    :return: the MSE error
    """
    return np.mean((y - y_hat)**2)
```

[18]:
```
base_model = LS_Model(y=y1, x=x1, include_coef_col=True)
base_abundance = base_model.fit()
```

[19]:
```
base_error = reconstruction_error(base_model.y, base_model.x, base_abundance.T)
base_error
```

[19]: 57505.15031205081

This error may seem a consequence of faulty implementation, but `sklearn`'s Linear Regression model with no restrictions yields a similarly large error.

1.4 OLS no intercept

[20]:
```
no_inter_model = LS_Model(y=y1, x=x1, include_coef_col=False)
no_inter_abundance = no_inter_model.fit()
```

[21]:
```
no_inter_error = reconstruction_error(no_inter_model.y, no_inter_model.x,␣
↪no_inter_abundance.T)
no_inter_error
```

[21]: 67062.4384199842

1.4.1 OLS no intercept, positive coefficients

[22]:
```
from sklearn.linear_model import LinearRegression


pos_model = LinearRegression(fit_intercept=False, positive=True).fit(x1, y1)
pos_abundance = pos_model.coef_
```

[23]:
```
pos_error = reconstruction_error(y1, x1, pos_abundance)
pos_error
```

[23]: 2424695.429849765

1.4.2 LASSO, no intercept, positive coefficients

[24]:
```
from sklearn.linear_model import Lasso


lasso = Lasso(fit_intercept=False,
              positive=True,
              tol=1e-3,
              max_iter=int(1e5)).fit(x1, y1)
lasso_abundance = lasso.coef_
```

```
C:\Users\user\anaconda3\envs\manis\Lib\site-
packages\sklearn\linear_model\_coordinate_descent.py:628: ConvergenceWarning:
Objective did not converge. You might want to increase the number of iterations,
check the scale of the features or consider increasing regularisation. Duality
gap: 0.000e+00, tolerance: 0.000e+00
  model = cd_fast.enet_coordinate_descent(
```

[25]:
```
lasso_error = reconstruction_error(y1, x1, lasso_abundance)
lasso_error
```

[25]:
```
2427525.515551848
```

1.4.3 OLS, no intercept, sum-to-one coefficients

In order to enforce the sum-to-one constraint we will build our own function using methods from the `scipy.optimize` module. While there are other solutions, these tend to either be very complicated and hard to modify for our needs (such as the `CVXOPT` package) or to not work because of memory constraints / implementation bugs (in this case the `cvxpy` package).

Of note is that the current implementation needs to run a non-negative least squares and a coordinate descent with constraints routine for each one of our points, as the `nnls` function only supports vector calculations. This isn't as inefficient as it may seem, since the cost of the coordinate descent and non-negative least squares routines is much larger proportionally compared to the overhead of calling them for each point.

Still, the calculation may take some minutes to complete. We could speed it up by assigning each point's calculations to different threads, as the calculation is independent between them. However, we choose to keep the single-threaded code for the sake of clarity and simplicity.

[26]:
```
from tqdm.auto import tqdm
from scipy.optimize import minimize
from scipy.optimize import nnls


def lsq_sum_to_one(A: np.ndarray, b: np.ndarray,
                   bounds :list[list[float|None]],
                   verbose: bool=True) -> np.ndarray:
    """
```

```
    Solve a linear least-squares problem subject to the constraint that the sum of
    the solution vector elements is equal to one.

    :param A: The coefficient matrix in the linear system.
    :type A: np.ndarray

    :param b: The target values in the linear system.
    :type b: np.ndarray

    :param bounds: The bounds for the variables in the optimization problem.
                   Each element is a list [min, max] or None.
    :type bounds: list[list[float|None]]

    :param verbose: If True, display progress using tqdm.
    :type verbose: bool, optional

    :return: The solution vector that minimizes the least-squares problem
             subject to the sum-to-one constraint.
    :rtype: np.ndarray

    :Example:

    .. code-block:: python

        >>> A = np.array([[1, 2], [3, 4]])
        >>> b = np.array([5, 6])
        >>> bounds = [[0, 1], [0, 1]]
        >>> coefficients = lsq_sum_to_one(A, b, bounds)
        >>> print(coefficients)
        [0.545, 0.455]
    """

    coefficients = np.zeros((A.shape[1], b.shape[1]))

    # define minimization function
    min_func = lambda x, A, b: np.linalg.norm(A.dot(x) - b)

    #Define constraints and bounds
    cons = {'type': 'eq', 'fun': lambda x:  np.sum(x)-1}

    iterable = tqdm(range(y1.shape[1])) if verbose else range(y1.shape[1])

    for i in iterable:
        bi = b[:, i]
        #Use nnls to get initial guess
        x0, rnorm = nnls(A, bi)
```

```
        #Call minimisation subject to these values
        minout = minimize(min_func, x0, args=(A, bi), method='SLSQP',
                          bounds=bounds,constraints=cons)
        x = minout.x
        coefficients[:, i] = x

    return coefficients
```

[27]:
```
sum1_coeffs = lsq_sum_to_one(x1, y1, 9*[[None, None]]).T
```

```
  0%|          | 0/12829 [00:00<?, ?it/s]
```

Let's confirm that our results do indeed sum up to one:

[28]:
```
sum1_coeffs.shape, np.isclose(sum1_coeffs.sum(axis=1), 1).sum()
```

[28]:
```
((12829, 9), 12829)
```

[29]:
```
sum1_abundance = sum1_coeffs
sum1_error = reconstruction_error(y1, x1, sum1_abundance)
sum1_error
```

[29]:
```
151083.76969310985
```

1.4.4 OLS, no intercept, positive and sum-to-one coefficients

[30]:
```
pos_sum1_coeffs = lsq_sum_to_one(x1, y1, 9*[[0,None]]).T
```

```
  0%|          | 0/12829 [00:00<?, ?it/s]
```

We can quickly check that our results satisfy the requirements of a pdf:

[31]:
```
pos_sum1_coeffs.shape, np.isclose(pos_sum1_coeffs.sum(axis=1), 1, 1e-2).sum(),␣
↪(pos_sum1_coeffs < 0).sum()
```

[31]:
```
((12829, 9), 9381, 0)
```

There are a few rows which don't exactly add up to 1 (within ± 0.01), which is expected since our algorithm is only approximate to our sum-to-one goal.

[32]:
```
pos_sum1_abundance = pos_sum1_coeffs
pos_sum1_error = reconstruction_error(y1, x1, pos_sum1_abundance)
pos_sum1_error
```

[32]:
```
4161801.393312393
```

1.4.5 Results

A table of the results for each model is presented below. Each row represents a different linear model, and the columns the reconstruction error, as well as the coefficients of the selected pixel we

analyzed during the exploratory analysis.

[33]:
```
def add_row(table, name, error, abundance, focus_point=0):
    res_table.add_row([name, error, abundance[focus_point][0],␣
 ↪abundance[focus_point][1],
                       abundance[focus_point][2], abundance[focus_point][3],␣
 ↪abundance[focus_point][4],
                       abundance[focus_point][5], abundance[focus_point][6],␣
 ↪abundance[focus_point][7],
                       abundance[focus_point][8]])


abundance_dim = 9
res_table = PrettyTable(["Model", "Reconstruction Error"] + [f"x{i}" for i in␣
 ↪range(1, abundance_dim+1)])
res_table.float_format = ".3"

add_row(res_table, "OLS", base_error, base_abundance)
add_row(res_table, "OLS no interc.", no_inter_error, no_inter_abundance)
add_row(res_table, "OLS pos", pos_error, pos_abundance)
add_row(res_table, "LASSO pos", lasso_error, lasso_abundance)
add_row(res_table, "OLS sum-to-1", sum1_error, sum1_abundance)
add_row(res_table, "OLS pos sum-to-1", pos_sum1_error, pos_sum1_abundance)

res_table
```

[33]:
```
+------------------+---------------------+----------+----------+-----------+-
---------+----------+----------+----------+----------+----------+
|      Model       | Reconstruction Error |    x1    |    x2    |     x3    |
x4    |     x5    |     x6    |     x7    |     x8    |     x9    |
+------------------+---------------------+----------+----------+-----------+-
---------+----------+----------+----------+----------+----------+
|       OLS        |      57505.150       | 11370.650 | 1492.548 | -10447.333 |
-3862.049 | -3357.114 | -1974.905 | -6631.420 | -6238.872 | -6609.833 |
|  OLS no interc.  |      67062.438       |  -36.069  |  3.676   |  -18.059   |
-9.075  |  -15.295  |  -46.506  |  -12.394  |  -14.204  |   -0.744  |
|     OLS pos      |     2424695.430      |   0.000   |  0.000   |   0.000    |
0.268   |   0.000   |   0.000   |   0.000   |   0.000   |   0.000   |
|    LASSO pos     |     2427525.516      |   0.000   |  0.000   |   0.000    |
0.268   |   0.000   |   0.000   |   0.000   |   0.000   |   0.000   |
|   OLS sum-to-1   |      151083.770      |  -41.842  |  7.105   |   5.302    |
1.403   |   0.034   |   -8.065  |   14.878  |   13.968  |   8.217   |
| OLS pos sum-to-1 |     4161801.393      |   0.000   |  0.000   |   0.000    |
0.184   |   0.000   |   0.000   |   0.000   |   0.000   |   0.816   |
+------------------+---------------------+----------+----------+-----------+-
---------+----------+----------+----------+----------+----------+
```

A number of observations can be made about our models:

1. The OLS model with no constraints leads to the best estimation by far in regards to the MSE error. This was certain to happen, since an optimization problem with no constraints will always have less or equal error to a constraint one (assuming same estimator and data). However, its solutions are also completely uninterpretable (huge positive and negative values).
2. The MSE error is also relatively low when we force the coefficients to sum to one, but not necessarily be positive.
3. The OLS with positive coefficients has almost identical results with the LASSO estimator, indicating our problem does not need any regularization in the form of sparsity. In fact, our model already becomes very sparse when demanding a positive solution with no other constraints.
4. The OLS modeling a pdf (positive coefficients summing up to one) leads to 20 times (!) the error compared to the one without the positive constraint. However, it's the only one with the possibility of any interpretation: each coefficient represents the % presence of a material in a given pixel. Therefore, it's the only one which carries any explanatory information.

So why is our most preferred solution the one with by far the biggest error? The answer is that by constraining our model so heavily, it can no longer reconstruct the original y values. This would be a catastrophic problem if we used this model for estimating the exact signatures or for prediction. Our use case however is unrelated to both of these, and thus the error isn't as meaningful. Besides, we have little reason to doubt our hypothesis since it very often holds and is widely used in practice [1,2].

[1] Jiaojiao Wei, Xiaofei Wang, "An Overview on Linear Unmixing of Hyperspectral Data", Mathematical Problems in Engineering, vol. 2020, Article ID 3735403, 12 pages, 2020. https://doi.org/10.1155/2020/3735403

[2] Jan G.P.W. Clevers, Raul Zurita-Milla, 3 - Multisensor and multiresolution image fusion using the linear mixing model, Image Fusion, Academic Press, 2008, Pages 67-84, ISBN 9780123725295, https://doi.org/10.1016/B978-0-12-372529-5.00004-4 (https://www.sciencedirect.com/science/article/pii/B9780123725295000044)

1.5 Classification

We will attempt to classify pure pixels in the image.

We will import the pixel labels. These represent the training, test and operational sets. The latter represents data we would normally acquire after having trained and tested our classifiers, and thus will not be used in this project.

[34]:
```
# Trainining set for classification
Pavia_labels = sio.loadmat('data/classification_labels_Pavia.mat')
y_train_set = (np.reshape(Pavia_labels['training_set'],(200,300))).T
y_test_set = (np.reshape(Pavia_labels['test_set'],(200,300))).T
y_oper_set = (np.reshape(Pavia_labels['operational_set'],(200,300))).T

arrays_stats([y_train_set, y_test_set, y_oper_set], names=["Train", "Test",␣
 ↪"Operational"])
```

[34]:
```
+-------------+------------+------+------+-----+-----+------+------+
|    Array    |   Shape    | Mean | Std  | Min | Max |  Q1  |  Q3  |
```

```
+-------------+------------+------+------+-----+-----+------+------+
|    Train    | (300, 200) | 0.54 | 1.76 |  0  |  9  | 0.00 | 0.00 |
|     Test    | (300, 200) | 0.27 | 1.27 |  0  |  9  | 0.00 | 0.00 |
| Operational | (300, 200) | 0.27 | 1.27 |  0  |  9  | 0.00 | 0.00 |
+-------------+------------+------+------+-----+-----+------+------+
```

The label sets also act as masks, each containing only a subset of the original picture. We can use that to obtain the training, test and operation data from our original picture as follows:

[35]:
```
mask = y_train_set != 0
x_train = hsi[mask]
y_train = y_train_set[mask].flatten()

x_train.shape, y_train.shape
```

[35]:
```
((6415, 103), (6415,))
```

[36]:
```
mask = y_test_set != 0
x_test = hsi[mask]
y_test = y_test_set[mask].flatten()

x_test.shape, y_test.shape
```

[36]:
```
((3207, 103), (3207,))
```

[37]:
```
mask = y_oper_set != 0
x_oper = hsi[mask]
y_oper = y_oper_set[mask].flatten()

x_oper.shape, y_oper.shape
```

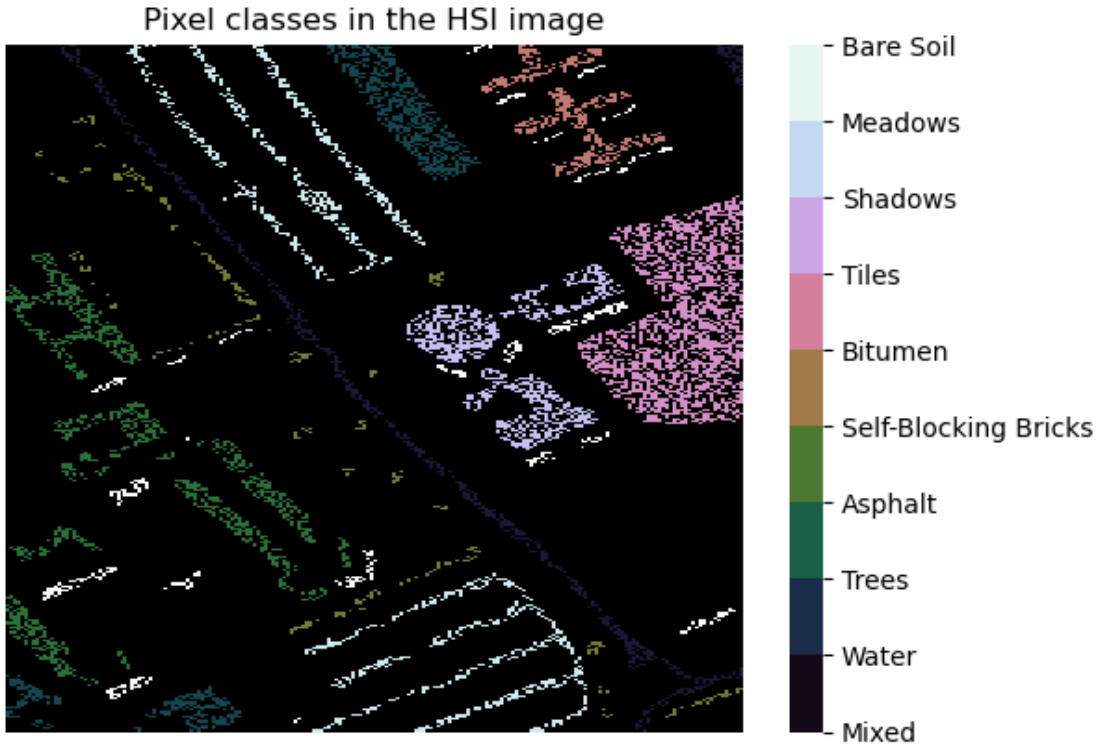
[37]:
```
((3207, 103), (3207,))
```

Notice how there is a 2:1 train to test ratio, which is atypical of most Machine Learning tasks (a 80%-20% split is much more widely accepted).

Since the labels and data still represent image pixels, we can clearly visualize the classes in our image:

[38]:
```
import seaborn as sns


ax = sns.heatmap(y_train_set.astype(int),
                 cmap=sns.color_palette("cubehelix", as_cmap=True),
                 cbar_kws={"boundaries": np.arange(y_train_set.min(),␣
 ↪y_train_set.max()+1, 1)})

ax.set(xticklabels=[], yticklabels=[])
ax.tick_params(left=False, bottom=False)
```

```
cbar = ax.collections[0].colorbar
cbar.set_ticklabels(class_names.values())

plt.title("Pixel classes in the HSI image")
plt.show()
```



We will be considering the following estimators:

- Naive Bayes
- Minimum Euclidean Distance
- Bayesian Classifier
- K-Nearest Neighbours

Each model will be evaluated based on 10-K cross validation. We will then train the classifier with all available data to get the training accuracy and confusion matrix.

[39]:
```
def confusion_matrix(true_labels, predicted_labels):
    """
    Compute a confusion matrix based on true and predicted labels.

    :param true_labels: The true class labels.
    :type true_labels: iterable
```

```
    :param predicted_labels: The predicted class labels.
    :type predicted_labels: iterable

    :return: The confusion matrix, where rows represent true labels and
             columns represent predicted labels.
    :rtype: np.ndarray
    """
    num_classes = max(np.max(true_labels), np.max(predicted_labels)) + 1

    confusion_matrix = np.zeros((num_classes, num_classes), dtype=int)

    for true, pred in zip(true_labels, predicted_labels):
        confusion_matrix[true, pred] += 1

    return confusion_matrix
```

[40]:
```
def get_accuracy(confusion_matrix: np.ndarray) -> float:
    """
    Calculate the accuracy from a confusion matrix.

    :param confusion_matrix: A 2D NumPy array representing a confusion matrix.
    :type confusion_matrix: np.ndarray

    :return: The accuracy calculated from the confusion matrix.
    :rtype: float
    """
    return confusion_matrix.diagonal().sum() / confusion_matrix.sum()
```

[41]:
```
def conf_matrix_heatmap(name, conf_matrix, ax=None):
    """
    Plot a heatmap of a confusion matrix.

    :param name: The title of the heatmap.
    :type name: str

    :param conf_matrix: The confusion matrix to be visualized.
    :type conf_matrix: np.ndarray

    :param ax: The Axes on which to draw the heatmap. If not provided, a new␣
 ↪figure will be created.
    :type ax: matplotlib.axes._axes.Axes, optional

    :return: This function does not return anything.
    """
    sns.heatmap(conf_matrix,
                cmap=sns.color_palette("light:b", as_cmap=True),
                cbar_kws={'label': "Instances Categorized"},
```

```
                annot=True,
                fmt="d",
                linewidths=.5,
                xticklabels=class_names.values(),
                yticklabels=class_names.values(),
                ax=ax)
    plt.title(name)
```

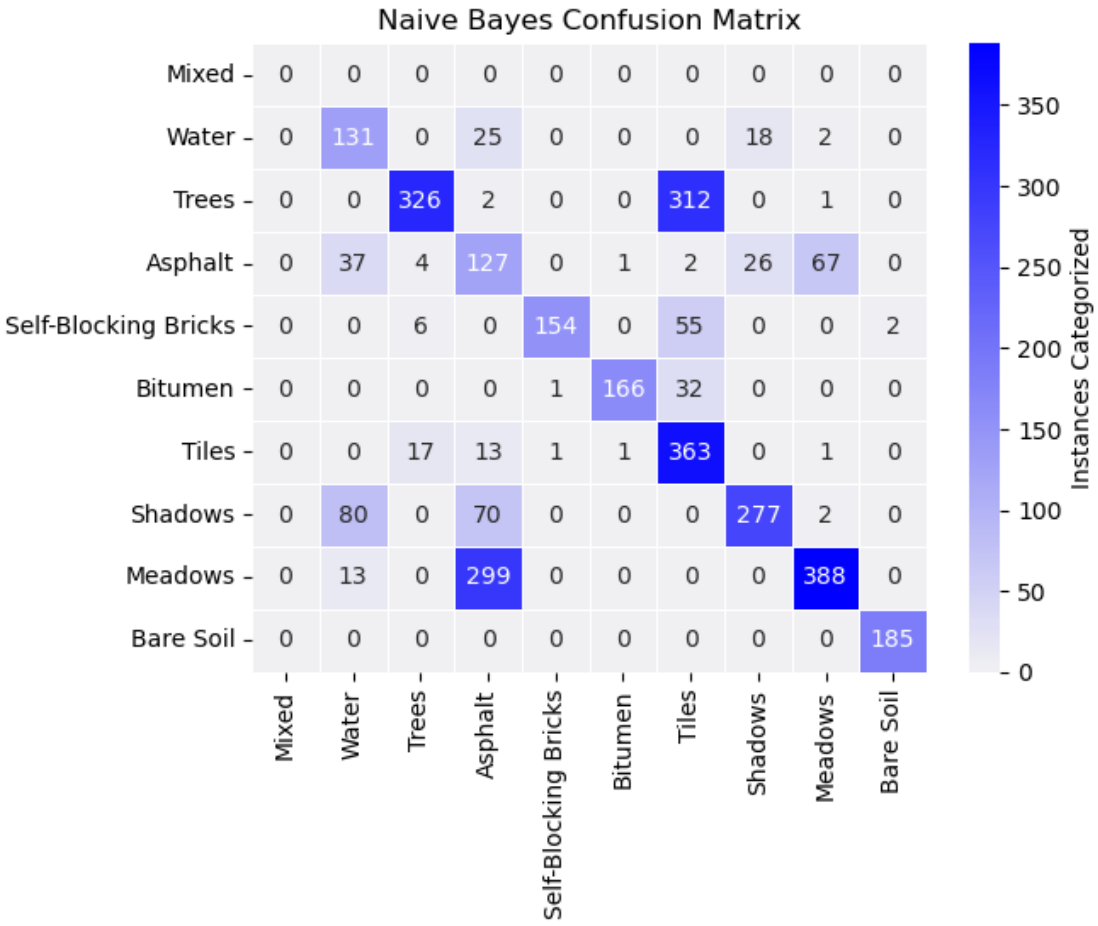
1.5.1 Naive Bayes

[42]:
```
from sklearn.model_selection import cross_val_score


def cross_val_res(model, x, y, scoring=None):
    res = cross_val_score(model, x, y, cv=10, scoring=scoring)
    return res.mean(), res.std()
```

[43]:
```
from sklearn.naive_bayes import GaussianNB


res = cross_val_res(GaussianNB(), x_train, y_train)
print(f"Naive Bayes mean accuracy {res[0]:.4f}, std: {res[1]:.4f}")
```

```
Naive Bayes mean accuracy 0.6447, std: 0.0568
```

[44]:
```
naive_model = GaussianNB().fit(x_train, y_train)
naive_train_res = naive_model.predict(x_train)
naive_test_res = naive_model.predict(x_test)

naive_conf = confusion_matrix(naive_test_res, y_test)
conf_matrix_heatmap("Naive Bayes Confusion Matrix", naive_conf)
```



[45]:
```
naive_conf_train = confusion_matrix(naive_train_res, y_train)
print(f"Naive Bayes \nTraining Accuracy: {get_accuracy(naive_conf_train):.
  ↪3f}\nTesting Accuracy: {get_accuracy(naive_conf):.3f}")
```

```
Naive Bayes
Training Accuracy: 0.660
Testing Accuracy: 0.660
```

1.5.2 Minimum Euclidean Distance

The Minimum Euclidean Distance classifier is a variant of the Bayesian Classifier, where we assume our data have the same variance across all axes, as well as each dimension being statistically independent from each other. Graphically, this would be represented as circles with equal radii, which according to the Figure containing the image classes, does not seem to be the case here.

Under this assumption, the optimal classification for a point x would be the equivalent to finding the minimum euclidean distance between it and the $\mu_j \forall j$, representing the means of each class j.

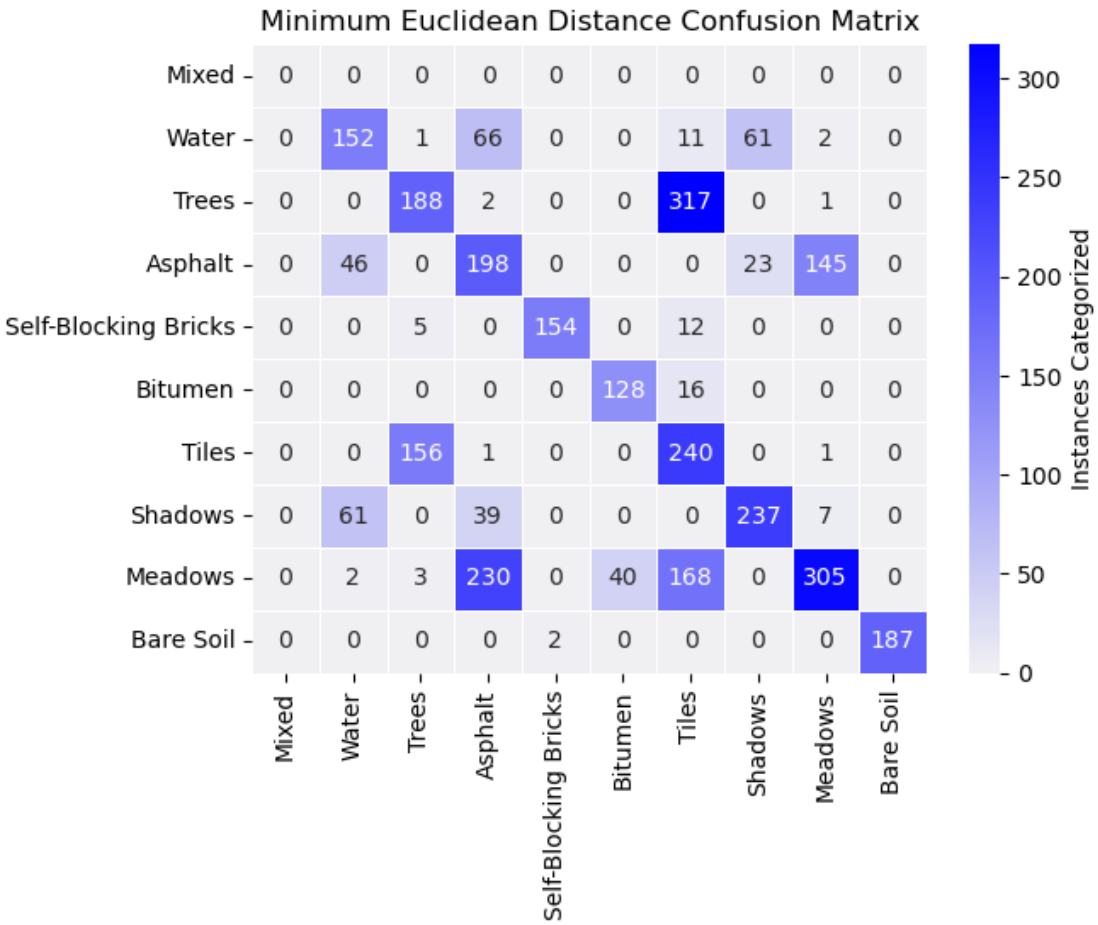
[46]:
```
from sklearn.base import BaseEstimator, RegressorMixin


class EuclideanDistanceClassifier(BaseEstimator, RegressorMixin):
    """
    A Bayesian Classifier which assumes equal covariances across classes,
    (diagonal Covariance matrix, where all elements have the same variance␣
 ↪sigma^2).
    """

    def __init__(self):
        self.means_labels = []
        self.means = []

    def fit(self, x_train, y_train):
        self.means_labels = np.unique(y_train)

        for label in self.means_labels:
            data = x_train[y_train == label]
            self.means.append(data.mean(axis=0))

        return self

    def predict(self, x):
        preds = []

        for point in x:
            distances = []

            for label in self.means_labels:
                distances.append(np.linalg.norm(point - self.means[label-1]))

            min_dist = np.argmin(distances)
            preds.append(self.means_labels[min_dist])

        return np.array(preds)
```

[47]:
```
res = cross_val_res(EuclideanDistanceClassifier(), x_train, y_train,␣
 ↪scoring="f1_weighted")
print(f"Minimum Euclidean Distance Mean accuracy: {res[0]:.4f}, std: {res[1]:.
 ↪4f}")
```

```
Minimum Euclidean Distance Mean accuracy: 0.5966, std: 0.1240
```

[48]:
```
euclid_model = EuclideanDistanceClassifier().fit(x_train, y_train)

euclid_train_res = euclid_model.predict(x_train)
```

```
euclid_test_res = euclid_model.predict(x_test)

euclid_conf = confusion_matrix(euclid_test_res, y_test)
conf_matrix_heatmap("Minimum Euclidean Distance Confusion Matrix", euclid_conf)
```



[49]:
```
euclid_train_conf = confusion_matrix(euclid_train_res, y_train)
print(f"Minimum Euclidean Distance \nTraining Accuracy:␣
  ↪{get_accuracy(euclid_train_conf):.3f}\nTesting Accuracy:␣
  ↪{get_accuracy(euclid_conf):.3f}")
```

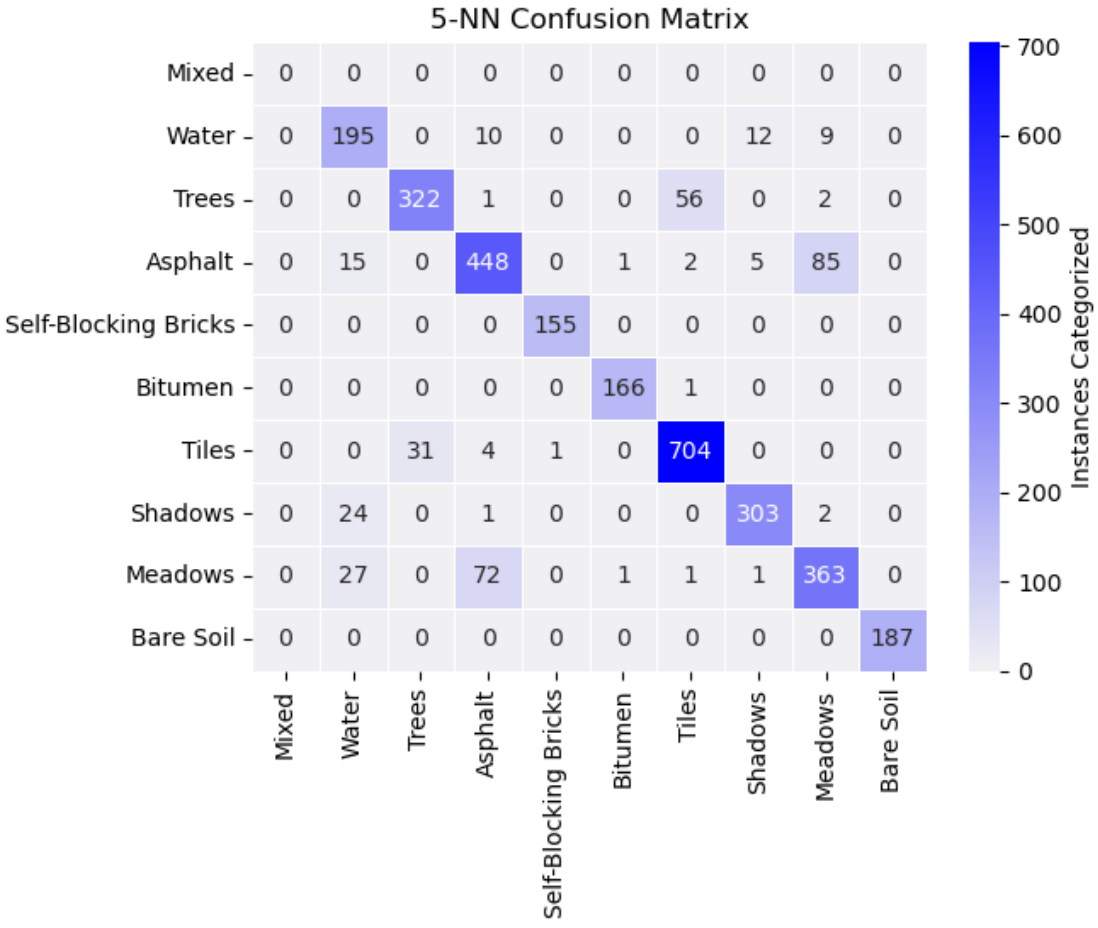
```
Minimum Euclidean Distance
Training Accuracy: 0.568
Testing Accuracy: 0.558
```

1.5.3 K-Nearest Neighbours

[50]:
```
from sklearn.neighbors import KNeighborsClassifier


k = 5
res = cross_val_res(KNeighborsClassifier(n_neighbors=k), x_train, y_train)
print(f"{k}-NN mean accuracy {res[0]:.4f}, std: {res[1]:.4f}")
```

```
5-NN mean accuracy 0.8480, std: 0.0541
```

[51]:
```
knn_model = KNeighborsClassifier(n_neighbors=5).fit(x_train, y_train)

knn_train_res = knn_model.predict(x_train)
knn_test_res = knn_model.predict(x_test)
knn_conf = confusion_matrix(knn_test_res, y_test)

conf_matrix_heatmap(f"{k}-NN Confusion Matrix", knn_conf)
```



[52]:
```
knn_conf_train = confusion_matrix(knn_train_res, y_train)
print(f"{k}-Nearest Neighbours \nTraining Accuracy:␣
  ↪{get_accuracy(knn_conf_train):.3f}\nTesting Accuracy:␣
  ↪{get_accuracy(knn_conf):.3f}")
```

```
5-Nearest Neighbours
Training Accuracy: 0.917
Testing Accuracy: 0.886
```

1.5.4 Bayes Classifier

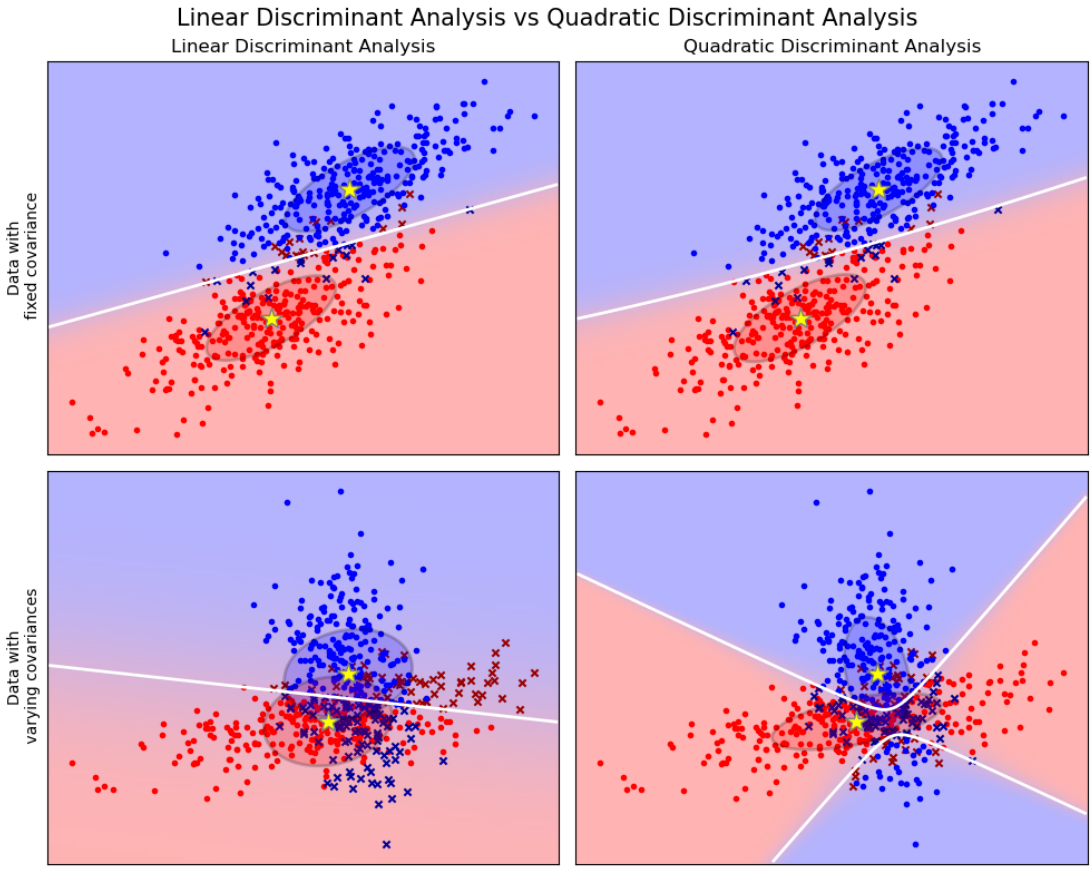
The general Bayes Classifier, as opposed to the Minimum Euclidean Distance Classifier, makes no assumptions in regards to the variances and covariances (statistical dependencies) between the data's dimensions, and thus is a more flexible classifier.

This classifier can be proven to be optimal in regards to the classification error, provided we know the generating probability density functions (pdfs). Since, this is impossible in most real world applications, these need to be estimated, and we thus must make some assumptions about the class distributions.

The most commonly used Bayesian Classifier is the *Mutlinomial Guassian Bayesian Classifier*.

`sklearn`'s equivalent models are the Linear Discriminant Analysis and Quadratic Discriminant Analysis estimators. The Figure below demonstrates the assumed data distributions (the two ellipses), their means (stars) and the decision surfaces for each class (blue and red areas) for both of these models.

- The Linear Discriminant Analysis estimator assumes that both classes follow Gaussian (normal) distributions with different means and same covariances, in which case the optimal decision surface is a line. This is depicted graphically as the two classes having the same shape and being spread in the same direction.
- The Quadratic Discriminant Analysis estimator assumes that both classes follow Gaussian (normal) distributions but with different means AND different covariances, meaning that our classes may feature different normal-like shapes.



Picture taken from https://scikit-learn.org/stable/modules/lda_qda.html Accessed 22/11/2023.

Choosing between the two available Bayesian Classifiers, we pick the Quadratic Discriminant Analysis estimator. As stated above, QDA works best when the classes have different (normal) shapes, and its decision surfaces or graphically can be represented by curves splitting each class from the others. This seems to be the case according to our Figure featuring our own classes in the image, and thus QDA will probably be a good fit.

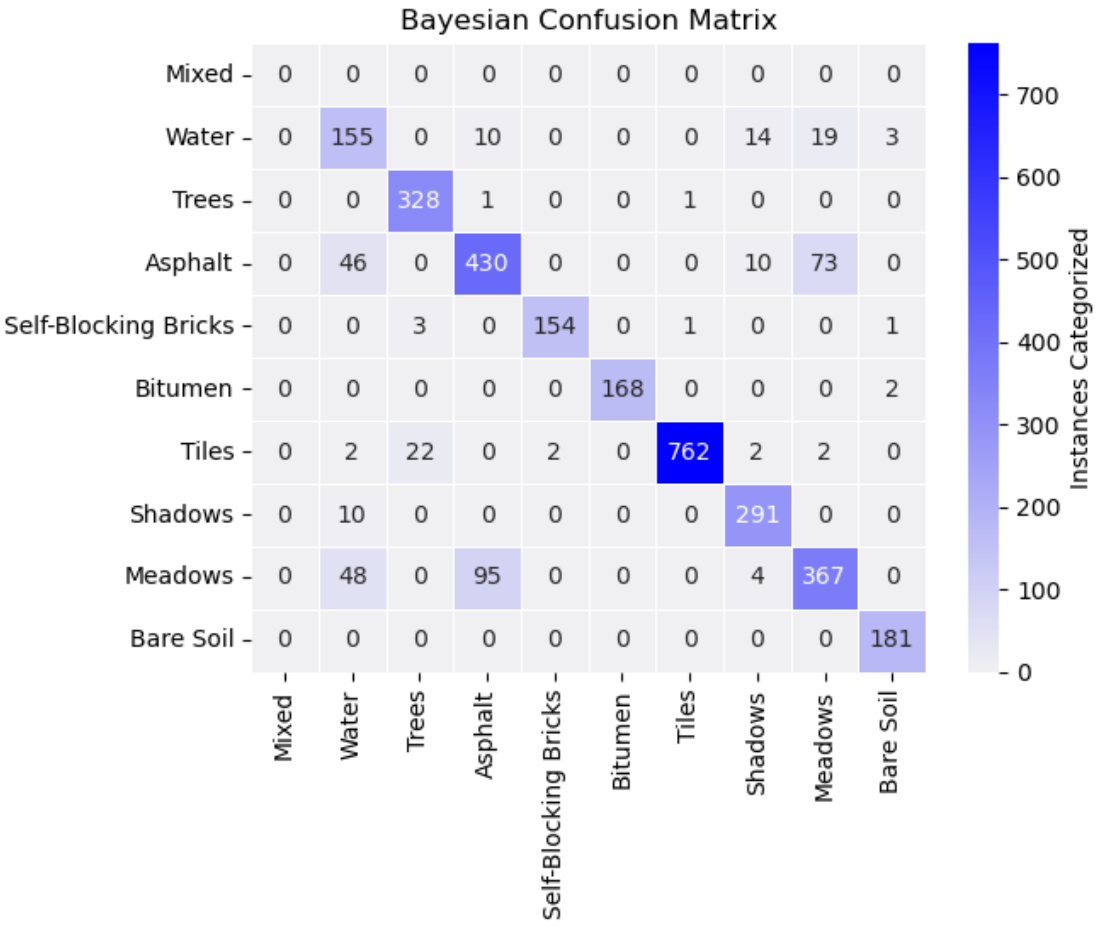
By choosing QDA as our Bayesian Classifier, we make one important assumption towards our data: that each class is, or sufficiently approximates, a normal distribution. We can make this assumption relatively safely, as discussed in the Exploratory Analysis Section.

We thus run our Bayesian Classifier with unequal variances:

[53]:
```
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis


res = cross_val_res(QuadraticDiscriminantAnalysis(), x_train, y_train)
print(f"Bayesian mean accuracy {res[0]:.4f}, std: {res[1]:.4f}")
```

```
Bayesian mean accuracy 0.8561, std: 0.0290
```

[54]:
```
bayes_model = QuadraticDiscriminantAnalysis().fit(x_train, y_train)

bayes_train_res = bayes_model.predict(x_train)
bayes_test_res = bayes_model.predict(x_test)
bayes_conf = confusion_matrix(bayes_test_res, y_test)

conf_matrix_heatmap("Bayesian Confusion Matrix", bayes_conf)
```

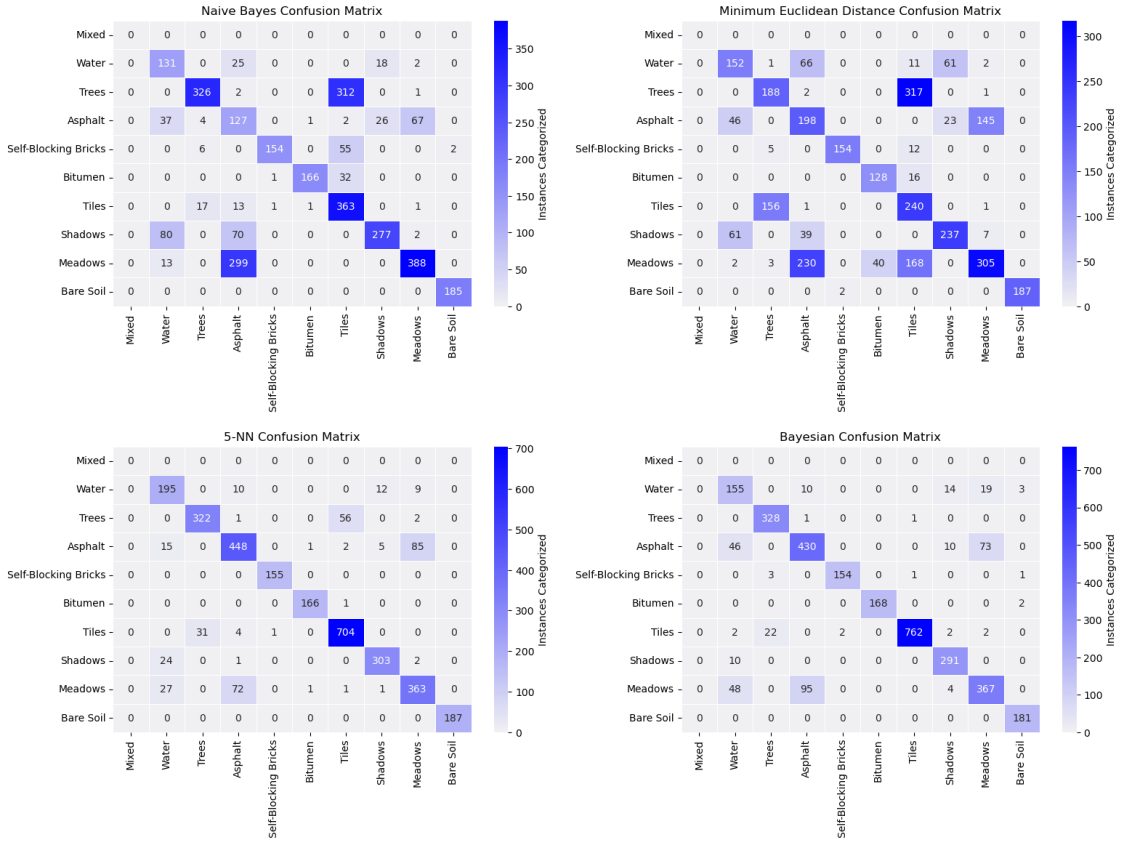


[55]:
```
bayes_conf_train = confusion_matrix(bayes_train_res, y_train)
print(f"Bayesian Classifier \nTraining Accuracy:␣
  ↪{get_accuracy(bayes_conf_train):.3f}\nTesting Accuracy:␣
  ↪{get_accuracy(bayes_conf):.3f}")
```

```
Bayesian Classifier
Training Accuracy: 0.991
Testing Accuracy: 0.884
```

1.5.5 Comparing the classifiers

[56]:
```
fig, axes = plt.subplots(2, 2)

conf_matrix_heatmap("", confusion_matrix(naive_test_res, y_test), ax=axes[0,0])
axes[0,0].set_title("Naive Bayes Confusion Matrix")

conf_matrix_heatmap("", confusion_matrix(euclid_test_res, y_test), ax=axes[0,1])
axes[0,1].set_title("Minimum Euclidean Distance Confusion Matrix")

conf_matrix_heatmap("", confusion_matrix(knn_test_res, y_test), ax=axes[1,0])
axes[1,0].set_title(f"{k}-NN Confusion Matrix")

conf_matrix_heatmap("", confusion_matrix(bayes_test_res, y_test), ax=axes[1,1])
axes[1,1].set_title("Bayesian Confusion Matrix")


fig.set_size_inches(16, 12)
plt.tight_layout(pad=2)
plt.show()
```



By comparing the Confusion matrices between the different classifiers we can make the following observations:

- Naive Bayes has great difficulty distinguishing between Tiles-Trees, as well as Asphalt-Meadows and Water-Shadows.
- The Minimum Euclidean Classifier is generally very easily confused, as we can see major deviations from the matrix diagonal. It does however perform well when classifying Bare Soil and Self Blocking Bricks.
- The 5-NN Classifier performs very well, being very accurate with almost all materials. We see some confusion in regards to Asphalt and Meadows.
- The Bayesian Classifier also performs very well, with innaccuracies being recorded on the same materials as the 5-NN model. This probably indicates that the Asphalt and Meadows are not well separable from their spectral signatures in general.

1.6 Spectral Unmixing and Classification: Possible Correlations

The results obtained from spectral unmixing (SU), which was posed here as a regression problem, and classification in hyperspectral image (HSI) analysis can be correlated, as both approaches provide information about the materials present in a given pixel.

1.6.1 Complementary Information

Spectral Unmixing provides sub-pixel information by estimating the abundance of pure materials within a mixed pixel, while classification simply assigns each pixel to a specific class. The results of spectral unmixing can thus complement the information obtained through classification by indicating the % contribution of different materials within a given pixel instead of simply a discrete label for each pixel.

Thus, should classification produce fuzzy or unreliable results in any parts of the image, Spectral Unmixing may be able to help in interpreting these areas with the extra information it provides.

1.6.2 Handling Mixed Pixels

Spectral Unmixing is specifically designed to handle mixed pixels by decomposing them into their constituent pure materials, while classification may struggle with mixed pixels, as it assigns a single class label to each pixel, potentially overlooking the presence of multiple materials.

In this project we specifically only focused on pure pixels. As we can see from our data however, real world problems most of the time need to handle mixed pixels. The relative abundance of mixed pixels is described in van der Meer (1999) [3] where according to the author:

> Reflected radiation from a pixel as observed in remote sensing imagery, however, has rarely interacted with a volume composed of a single homogenous material because natural surfaces and vegetation composed of a single uniform material do not exist. Most often the electromagnetic radiation observed as pixel reflectance values results from the spectral mixture of a number of ground spectral classes present at the surface sensed.

In these cases, spectral unmixing can help in understanding the content of mixed pixels, and the results can guide the classification process. For example, if a pixel is found to have significant

abundances of multiple materials, it may be more appropriately assigned to a mixed class during classification.

Additionally, the aforementioned mixed classified pixels could be then given to a SU algorithm in order to obtain more information on the materials present in the mix.

1.6.3 Derived end-member selection

Conversly, the classes generated from the classification algorithm can be used as labels in the SU task in order to derive better spectral signatures. This task is called "end-member selection", and is described by van der Meer (1999) [3] as:

> A set of end-members should allow to describe all spectral variability for all pixels, produce unique results, and be of significance to the underlying science objectives. Selection of end-members can be achieved in two ways:
>
> 1. from a spectral (field or laboratory) library, and
> 2. from the purest pixels in the image.
>
> The second method has the advantage that selected end-members were collected under similar atmospheric conditions. End-members resulting through the first option are generally denoted as 'known', while those from the second option results are known as 'derived' end-members

which traditionally was performed using techniques such as PCA.

> [4] van der Meer, F. (1999). Image classification through spectral unmixing. In Spatial Statistics for Remote Sensing (pp. 185-193). Dordrecht: Springer Netherlands.

1.6.4 Summary

In summary, spectral unmixing and classification are complementary techniques that can be used together to obtain a more comprehensive understanding of hyperspectral data. Besides allowing a richer interpretation of the HSI, these techniques can generate data used by the other to collaboratively enhance their results (better interepretation in classification and more accurate end-member selection in SU).

[57]:
```
print(f"Notebook executed in {int((time()-start)// 60)} minutes and␣
↪{(time()-start) % 60:.1f} seconds")
```

```
Notebook executed in 6 minutes and 11.9 seconds
```