INF250 Exam

December 2020

1 Exercise 1

The image **drone_blurr.tif** turned out to be a bit blurry. To make the image sharper I used two different techniques; unsharpen mask (USM) and a laplace filter. I started by importing the image, and converted it to gray.

```python
import matplotlib.pyplot as plt
from skimage.color import rgb2gray
from skimage import io
drone1 = io.imread('drone_blurr.tif')
drone = rgb2gray(drone1)
plt.imshow(drone, 'gray')
```

1.1 Unsharpen mask

USM is a technique to increase the sharpness using edge detection. This technique is constructed by a combination of the original image and a smoothed filter. It can be any kind of smoothing filter, but the Gaussian filter is the most common. To increase the edges you first put a smoothing filter on the original image, then you subtract the smoothed image from the original image. Now you just have the edges left, and multiplies these on the original image until you're satisfied with the result, as I did in figure 1

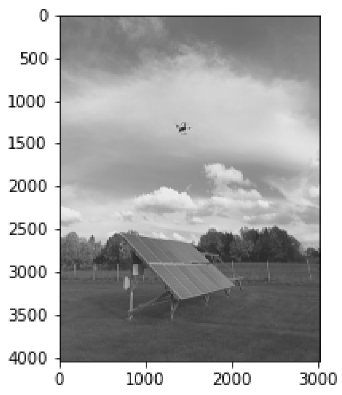

Figure 1: The shrapened image using unsharpen mask

```
from skimage.filters import gaussian
amount = 3
gaussdrone = gaussian(drone, sigma=5)
usm_drone = drone + amount*(drone - gaussdrone)
plt.imshow(usm_drone, 'gray')
```

The laplace filter is the second derivative of the image, and used to increase the contrast. Whenever there is an edge, there is also a positive(white) and a negative bump(black) so the edges comes out more clear. Then we subtract the laplace image from the original image, now the white and black edges has switched place and the image is sharpened, as shown in figure 2. As we did on the USM, laplace can also multiplies several times if necessary.

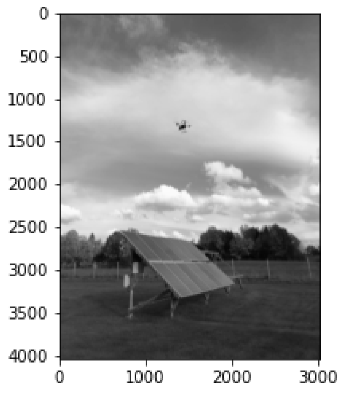

Figure 2: The sharpened image using laplace edge detection

```
from skimage.filters import laplace
drone_lap = laplace(drone)
drone_sharp = drone-amount*drone_lap
plt.imshow(drone_sharp, 'gray')
plt.savefig('lap_blurr.png')
```

2 Exercise 2

2.1 String

Started with plotting the histogram for the image string.jpg to find the errors. As we can see from figure 3 there is a big bump in the intensity level around 150, and a small bump from 150-200, so the defect must be around there. Then I used otsu to check the result and got the threshold value 167, which fits well with the histogram.

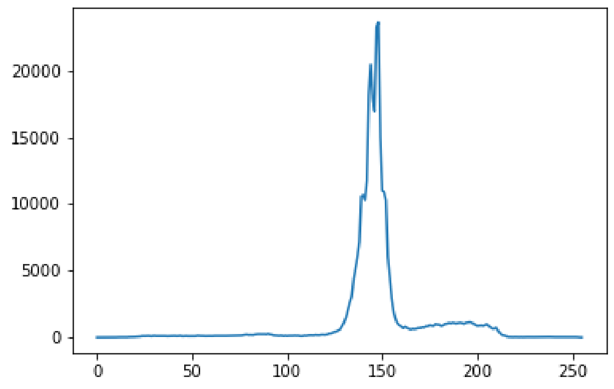

Figure 3: Histogram for the string fault with a birght line

Using the threshold value from otsu I computed a binarized image where the errors is shown really well, as shown in figure 4

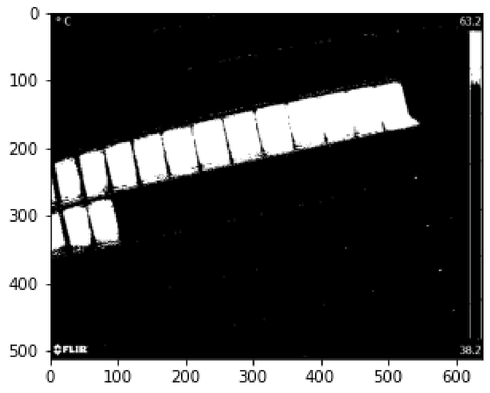

Figure 4: A binarized image of a defect line in a solar panel

2.2 Module

Used the same procedure for the module.tif image, but as figure 5 shows, there is just one big bump, so it is hard to choose the threshold value. I therefor split the channels, and analyze the three pictures separately

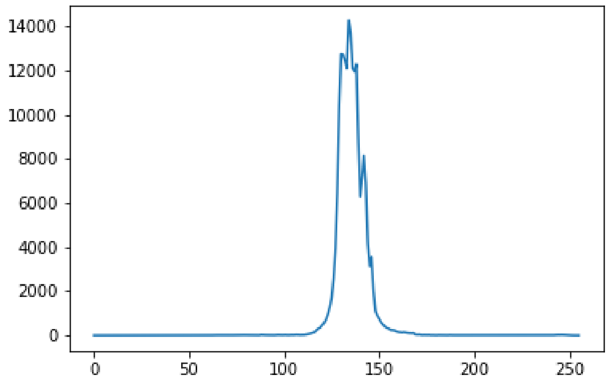

Figure 5: Histogram for the module fault with individual spots

These were the red, green and blue images extracted from the original image. As we can see figure 8 is the one with the clearest spots. I then looked at the histogram for the blue image.

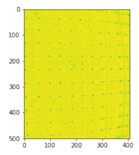

Figure 6: Red channel

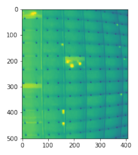

Figure 7: Green channel

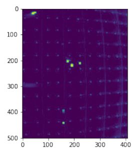

Figure 8: Blue channel

Figure 9 clearly shows to bumps, one around 100 and the other around 200, so the threshold value should be between those two. Ran the otsu function to check, and got the threshold value 147, and used that to run the threshold function. Figure 10 which is now a binarized image, shows the point errors a lot clearer then the original module image.

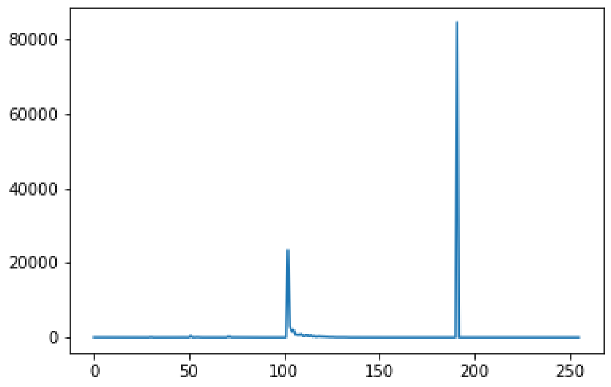

Figure 9: The histogram made from the blue channel

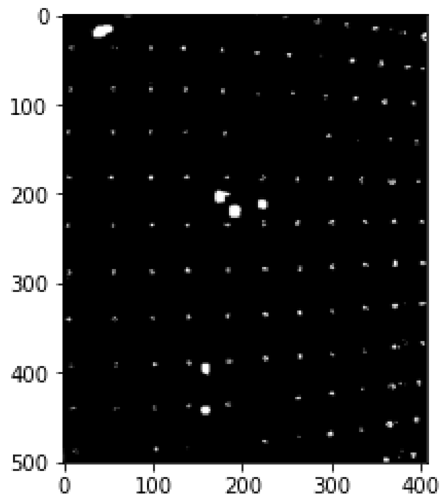

Figure 10: The binarized image

```
import numpy as np
def histogram(image):
    histogram = np.zeros(256)
    shape = np.shape(image)
    if len(shape)==3:
        image = image.mean(axis=2)
    for i in range(shape[0]):
        for j in range(shape[1]):
            pixval = int(image[i,j])
            histogram[pixval] += 1
    return histogram

def threshold(image, th=None):
    shape = np.shape(image)
    binarised = np.zeros([shape[0], shape[1]], dtype=np.uint8)
    if len(shape) == 3:
        image = image.mean(axis=2)
    if th is None:
        th = otsu(image)
        print(th)
    for i in range(shape[0]):
        for j in range(shape[1]):
            pixval = int(image[i,j])
            if pixval > th:
                binarised[i,j] = 255
            else:
                binarised[i,j] = 0
    plt.imshow(binarised, cmap = 'gray')

def otsu(image):
    hist = np.array(histogram(image))
    shape = np.shape(image)
    if len(shape) == 3:
        image = image.mean(axis=2)
    total = sum(hist)
    sumB = 0
    wB = 0
    maximum = 0.0
    vec = np.arange(256)
    sum1 = vec @ hist
    for i in vec:
        wF = total - wB
        if (wB > 0 and wF > 0):
            mF = (sum1- sumB)/ wF
            val = wB * wF * ((sumB/wB) - mF)*((sumB/wB)-mF)
            if val >= maximum:
                th = i
                maximum = val
        wB += hist[i]
        sumB += i*hist[i]
    return th
```

3 Exercise 3

The UVfluor.tif image has a low contrast, the histogram in figure 11 shows a lot of pixels with low intensity which is equivalent to dark areas.

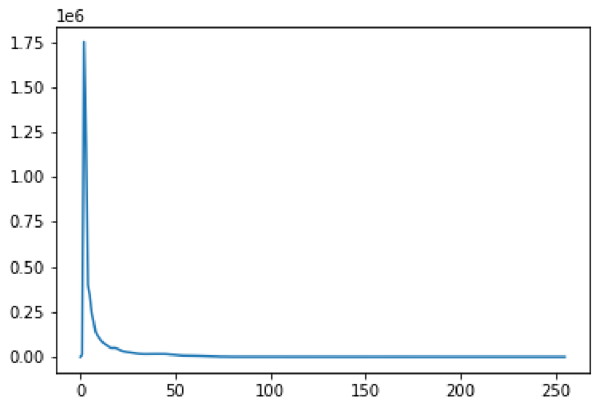

Figure 11: Histogram for image UVfluor.tif

To obtain images that can be used to spot cracks I used histogram equalization. Histogram equalization "spreads out the most frequent intensity values" in an image. This means that the dark areas will be lighter so the cracks comes out more clear. To compute a histogram equalization you first need a cumulative histogram. The cumulative histogram shows us where the variation is zero.

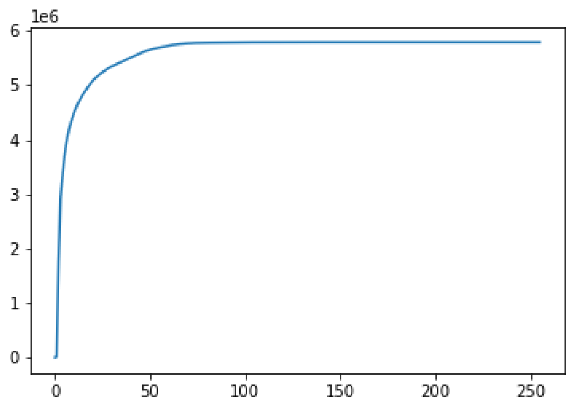

Figure 12: The cumulative histogram for UVfluor.tif

```
# Cumulative histogram
cumhist = np.zeros(256)
cumhist[0] = histogram[0]
for i in range(255):
    cumhist[i+1] = cumhist[i] + histogram[i+1]

plt.figure()
plt.plot(cumhist)
```

The equalization histogram is plotted from the cumulative histogram. Now figure 11 shows us a rather spread histogram.

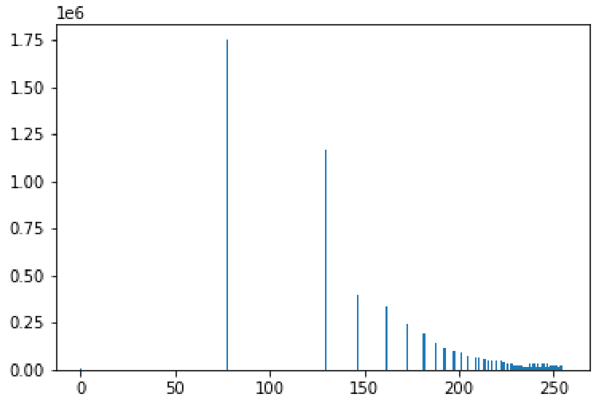

Figure 13: Equalization histogram

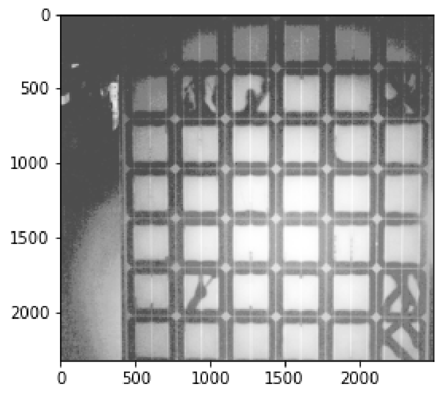

Figure 14: The equalized image

```
# histogram equalisation
K = 256
M = shape[0]
N = shape[1]
imagemean = UV.mean(axis=2)
for i in range(shape[0]):
    for j in range(shape[1]):
        a = int(imagemean[i,j])
        b = cumhist[a]*(K-1)/(M*N)
        imagemean[i,j] = b
plt.imshow(imagemean,'gray')
plt.figure()
plt.hist(imagemean.ravel(),256,[0,256])
plt.show()
```

To obtain a binarized image I ran the threshold function with a threshold value of 190. The otsu did not work on this image, so I used the trial and error method to find the threshold value that made the best image.

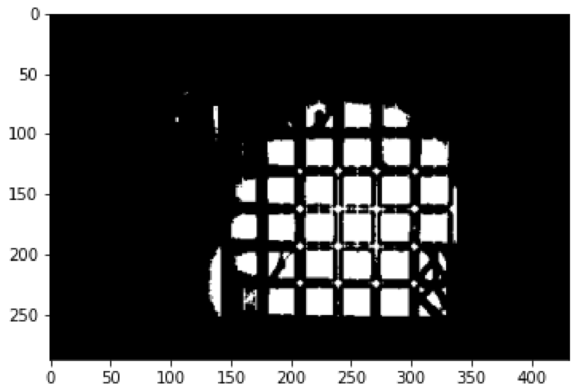

Figure 15: Binarized image of UVfluor.tif

4 Exercise 4

Started by importing the image and the packages that's needed. Then located the wavebands for red, green and blue and displayed it, as we can see figure 16 shows an RGB image with waveband number; 14, 50, 70.

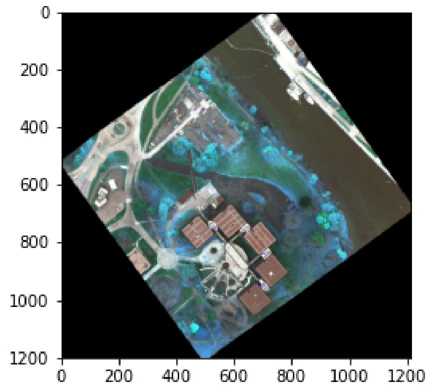

Figure 16: RGB image of the district Sandvika

```
img = envi.open('hyperspectral_vnir.hdr', 'hyperspectral_vnir.bsq')
wavelenght = envi.read_envi_header('hyperspectral_vnir.hdr')['wavelength']
ww = [float(i) for i in wavelenght]

view = imshow(img,(14, 50, 70), stretch=((0.02,0.98),(0.02,0.98),(0.02,0.98)))
```

4.1 Spectrum for all materials

The spectrum for the different materials shown in figure 17 where computed by the code below. Takes out a pixel from the given coordinates, and for all wavelengths, then plots it against the intensity of the pixel. From the image we can see that green line fits well to be vegetation since it has a little bump in the green wavelength area, and a beg peak in the near infrared (NIR) area. We can also see that the blue line is water since it has most of its pixels in the blue area.

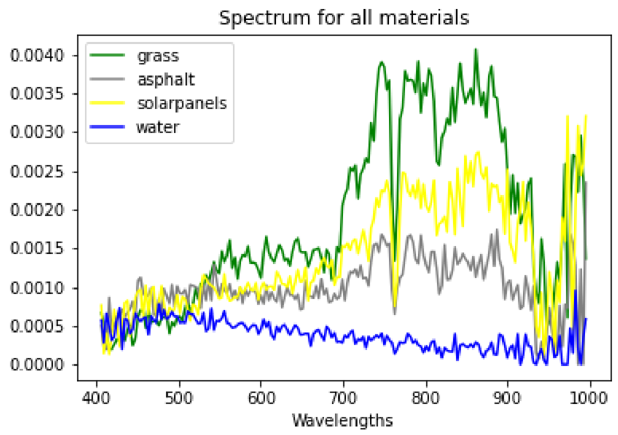

Figure 17: Spectrum for all materials

```
grass = np.array(img[500,700,:].reshape(-1,1))
asphalt = np.array(img[425,190,:].reshape(-1,1))
solarpanels = np.array(img[1000,780,:].reshape(-1,1))
water = np.array(img[650,1050,:].reshape(-1,1))
fig, ax = plt.subplots()
ax.plot(ww,grass, 'g', label = 'grass')
ax.plot(ww,asphalt, 'gray', label = 'asphalt')
ax.plot(ww,solarpanels, 'yellow', label = 'solarpanels')
ax.plot(ww, water, 'b', label = 'water')
plt.title('Spectrum for all materials')
plt.xlabel('Wavelengths')
leg = ax.legend();
plt.show()
```

4.2 Mean spectrum for all materials

When looking at the mean spectrum of 20x20 pixels shown in figure 18 we can see that there is less vegetation in the near infrared area. Also, the solar panels has a lot of pixels in the infrared area. We can see that the water has highest intensity for the wavelengths in the blue area, so that fits well with our expectations.

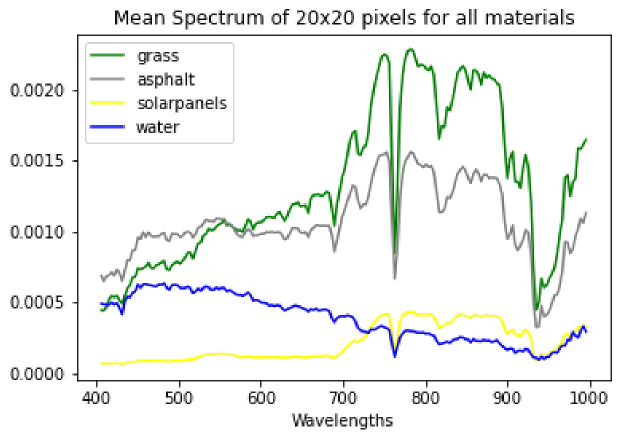

Figure 18: Mean spectrum of 20x20 pixels for all materials

```
grass = img[400:420,600:620]
g1 = grass.mean(axis=0)
g2 = g1.mean(axis=0).reshape(-1,1)

asphalt = img[400:420,180:200]
a1 = asphalt.mean(axis=0)
a2 = a1.mean(axis=0).reshape(-1,1)

water = img[650:670,1020:1040]
w1 = water.mean(axis=0)
w2 = w1.mean(axis=0).reshape(-1,1)

solar = img[1000:1020,730:750]
s1 = solar.mean(axis=0)
s2 = s1.mean(axis=0).reshape(-1,1)

fig, ax = plt.subplots()
ax.plot(ww,g2, 'g', label = 'grass')
ax.plot(ww,a2, 'gray', label = 'asphalt')
ax.plot(ww,s2, 'yellow', label = 'solarpanels')
ax.plot(ww, w2, 'b', label = 'water')
plt.title('Mean Spectrum of 20x20 pixels for all materials')
plt.xlabel('Wavelengths')
leg = ax.legend();
plt.show()
```

4.3 Normalized difference vegetation index

Normalized difference vegetation index shows us whether the image we are analyzing contains vegetation or not. This is done by taking a wavelength from the NIR then subtracting by a wavelength from the red area and dividing by the sum of it. The image that is displayed in figure 19 shows us the healthy vegetation area (light yellow)

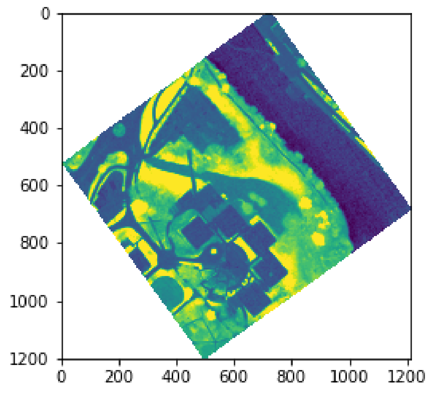

Figure 19: NDVI image, shows us healthy vegetation

```
ndvi_ima = (img[:,:,120]-img[:,:,80])/(img[:,:,120]+img[:,:,80])
plt.imshow(ndvi_ima, vmin=-0.3, vmax=0.6)
```

4.4 Principal component analysis

A PCA extract the bands that gives the largest variation in the data set. All the wavelengths are now variables and a new set of coordinates is computed which gives an image of the scores. In the first image, with the first principal component (figure 20) the darker areas with water and dark road contributes to variation. The solar panels can be identified in the second principal component (figure 21) and something that may be metal components either on roofs or cars. This is also were the vegetation contributes to variation. I don't think the third component (figure 22) gives out that much of information.

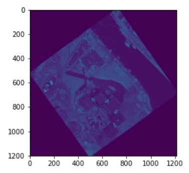

Figure 20: Principal component 1

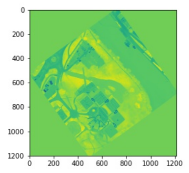

Figure 21: Principal component 2

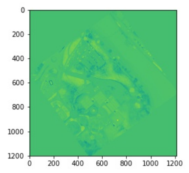

Figure 22: Principal component 3

```
# pca
hyperim = img[:,:,:]
pc = principal_components(hyperim)
pc_0990 = pc.reduce(fraction=0.999)

# score images
img_pc = pc_0990.transform(hyperim)
plt.figure()
plt.imshow(img_pc[:,:,0])
plt.figure()
plt.imshow(img_pc[:,:,1])
plt.figure()
plt.imshow(img_pc[:,:,2])
```

The loading plot to each principal component is computed by the eigenvectors, and is used to see what kind of variables that has the most effect on the components. From figure 23 we can see that there is a shift in the NIR which shows that the vegetation has an effect on the principal components, this is a different assumption then when we just looked at the score image. This shows as that it is not straight forward to analyze images, and we often need both the plot and the score image to get a good result. 21 also has a bump in the same are, but the bump is bigger compared to the rest, so vegetation is even more effective here. In figure 22 we can see a deep gorge which tells us that something in that spectral area is contributing to variance in principal component 3.

```
loadings = pc_0990.eigenvectors
plt.figure()
plt.plot(loadings[:,[0,1,2]])
```

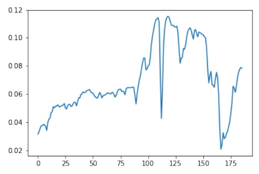
Figure 23: Loading plot for Principal component 1

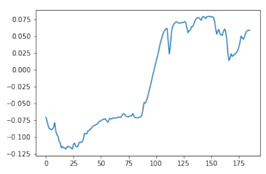
Figure 24: Loading plot for Principal component 2

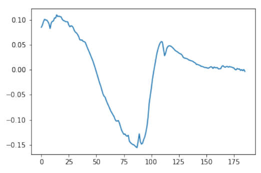
Figure 25: Loading plot for Principal component 3

K-means clustering is an unsupervised classification that separates n observation in to k clusters. Unsupervised classification means that its the program it self that sets the different groups. This is done by finding the closest mean (centroid) for each observation, in such way that objects in the same group are more similar compared to objects in other groups. The image in figure 26 shows the clusters displayed in an image, each class has a different color.

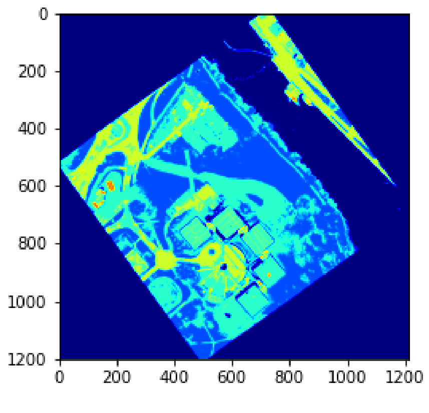
Figure 26: Cluster image, each cluster has a different color

```
(m,c) = kmeans(hyperim, 7, 30)
```

```
plt.imshow(m,'jet')
plt.figure()
```

The Gaussian Maximum Likelihood classifier assigns predefined labels to objects based on their features, because the humans are the one to predefine labels this is a supervised classification. In the code below groundtruth is first a list with only zeros. Then we decide what value some of the coordinates should have (we are making different classes). The image in figure 27 shows all the classes after the GMLC puts the similar objects in the groups we made.

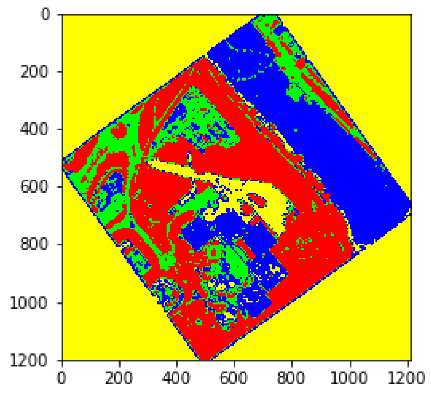

Figure 27: Gaussian Maximum Likelihood classification, each class is chosen by humans

```
shape = hyperim.shape
groundtruth = np.zeros([shape[0], shape[1]])

groundtruth[390:410,290:310] = 1.0 # grass
groundtruth[400:430,600:640] = 1.0 # grass
groundtruth[450:470,150:180] = 2.0 # asphalt
groundtruth[590:610,60:90] = 2.0 # asphalt
groundtruth[600:700,1000:1100] = 3.0 # water
groundtruth[400:450,870:900] = 3.0 # water
groundtruth[650:700,550:600] = 4.0 # Solar panels
groundtruth[775:795,465:485] = 4.0 # Solar panels
groundtruth[590:610,700:730] = 5.0 # Dark road
groundtruth[590:610,590:610] = 5.0 # Dark road
plt.imshow(groundtruth)

classes = create_training_classes(hyperim, groundtruth)
gmlc = GaussianClassifier(classes)
```

```
clmap = gmlc.classify_image(hyperim)
imshow(classes=clmap)
```