DSAI Assignment on Robotics Data

Subhasis Ray[*]

<2022-10-06 Thu>

- This assignment uses data collected by an agricultural robot developed at the University of Bonn. The full dataset with description is available here: `http://www.ipb.uni-bonn.de/data/sugarbeets2016/` and the associated article is (Chebrolu, Nived and Lottes, Philipp and Schaefer, Alexander and Winterhalter, Wera and Burgard, Wolfram and Stachniss, Cyrill, 2017).
- This robot was fitted with a camera and RGB-D (Red, Green, Blue, and Depth) sensor to image the plants on the field (and many other sensors to track its position).
- The camera recorded both RGB images and near infrared (NIR) emission.
- The complete dataset is very large. We will work with imaging data from only one session of the robot's trips.
- Save your work in a file named `DSAI_robotics.py` and submit it to codePost.

1 Startup

- For this assignment we need some image processing algorithms in addition to regular numpy operations and plotting. We shall use the `scikit-image` module for this. Please install it if you don't have it already: `pip install scikit-image`.
- Next we need to import the required libraries:

[*]subhasis.ray@plaksha.edu.in

```
#%% imports
import os
import numpy as np
# there is an io module in Python standard library, so rename
↪  skimage.io to skio
from skimage import io as skio
from skimage import filters, measure, morphology
from sklearn import cluster
import matplotlib.pyplot as plt
```

2 Load data

- The image data for this assignment has been put in a zip file which you should extract in a convenient location on your system. Inside it there should be two directories: `nir` and `rgb`, containing near-infrared and RGB images respectively.
- The file names are `nir_{ddddd}.png` and `rgb_{ddddd}.png` respectively, where `{ddddd}` is the five digit index of the image.
- Lookup the `listdir` function in the `os` module to obtain the list of files in the two directories.

  ```
  # On my system the image folders are under
  #
  ↪  'C:/Users/raysu/analysis/pybonirob/bonirob_2016-05-23-10-47-22_2_im'
  basedir =
  ↪  'C:/Users/raysu/analysis/pybonirob/bonirob_2016-05-23-10-47-22_2_im'
  # Create OS-specific paths for the image folders
  rgb_dir = os.path.join(basedir, 'rgb')
  nir_dir = os.path.join(basedir, 'nir')
  ```

- Use `os.listdir` for getting the filenames (basename, without the full path) into two variables: `rgb_files` and `nir_files`.
- Now load the corresponding images into two lists, `rgb_list` and `nir_list` using `skio.imread` function.

3 Visualize the data

- Take a look at a few randomly selected images. An rgb image is a 3D array with dimensions (height, width, color), and the way `skio.imread()` reads it, the colors are in the order R-G-B. These are also called channels (red-channel, green-channel, and blue-channel). The `nir` image has a single channel, and is thus a 2D array (height, width). Each entry in the array represents the intensity of light in that channel, at that pixel position. Thus if your RGB image array is `rgb`, then `rgb[10, 20, 1]` is the green intensity of the pixel at the 11-th row and the 21-st column. Note that the Y-axis is inverted compared to your regular data plotting convention.

- First check the shapes of the rgb and the nir images:

  ```
  #%% Print shapes of the image arrays
  print('Shapes: RGB:', rgb_list[5].shape, 'NIR:',
  ↪  nir_list[5].shape)
  ```

  ```
  Shapes: RGB: (966, 1296, 3) NIR: (966, 1296)
  ```

  As you can see, the images have the same size.

- Now display them using matplotlib's `imshow` function.

  - The RGB image

    ```
    #%% Display RGB image
    fig, ax = plt.subplots()
    ax.imshow(rgb_list[5])
    fig.suptitle('RGB image', fontdict={'color': 'blue'})
    fname = 'rgb_image.png'
    fig.savefig(fname)
    fname
    ```

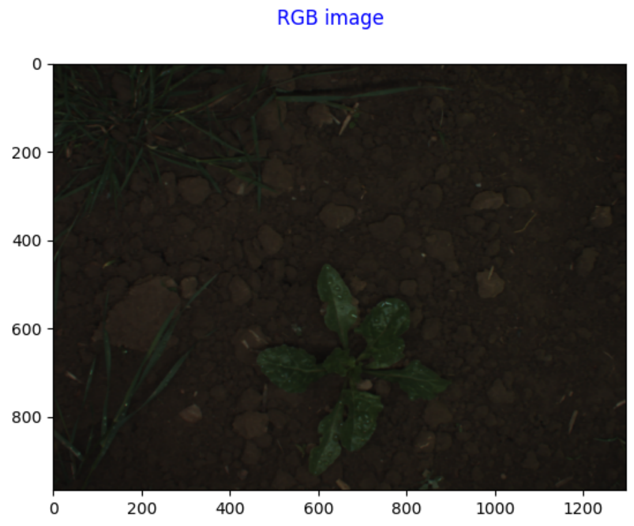

Figure 1: A sample RGB image captured by the robot

– NIR image (note that setting `cmap='gray'` maps low values to darker shades of gray). You can move the mouse pointer over a pixel in interactive mode of matplotlib to see its value along with the coordinates.

```
#%% Display NIR image by mapping the values to shades
↪  of gray
fig, ax = plt.subplots()
ax.imshow(nir_list[5], cmap='gray')
fig.suptitle('NIR image', fontdict={'color': 'blue'})
fname = 'nir_image.png'
fig.savefig(fname)
fname
```

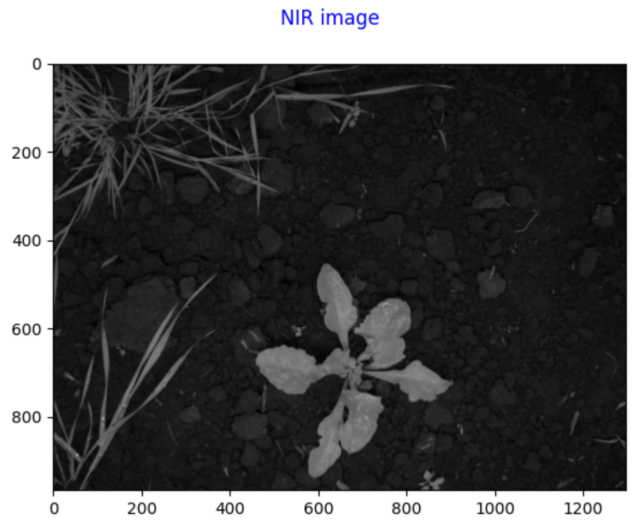

Figure 2: A sample NIR image captured by the robot

– Combined figure with separated channels

```
#%% Display the channels
fig, axes = plt.subplots(nrows=3, ncols=2,
↪  sharex='all', sharey='all')
for jj, color in enumerate(('Red', 'Green', 'Blue')):
  axes[jj, 0].imshow(rgb_list[5][:, :, jj],
  ↪  cmap='gray')
  axes[jj, 0].set_title(color)
axes[0, 1].imshow(rgb_list[5])
axes[0, 1].set_title('RGB')
axes[1, 1].imshow(nir_list[5], cmap='gray')
axes[1, 1].set_title('NIR')
axes[2, 1].remove()
fname = 'all_channels.png'
fig.set_size_inches(6, 8)
for ax in axes.flat:
  ax.set_axis_off()
fig.tight_layout()
fig.savefig(fname)
fname
```

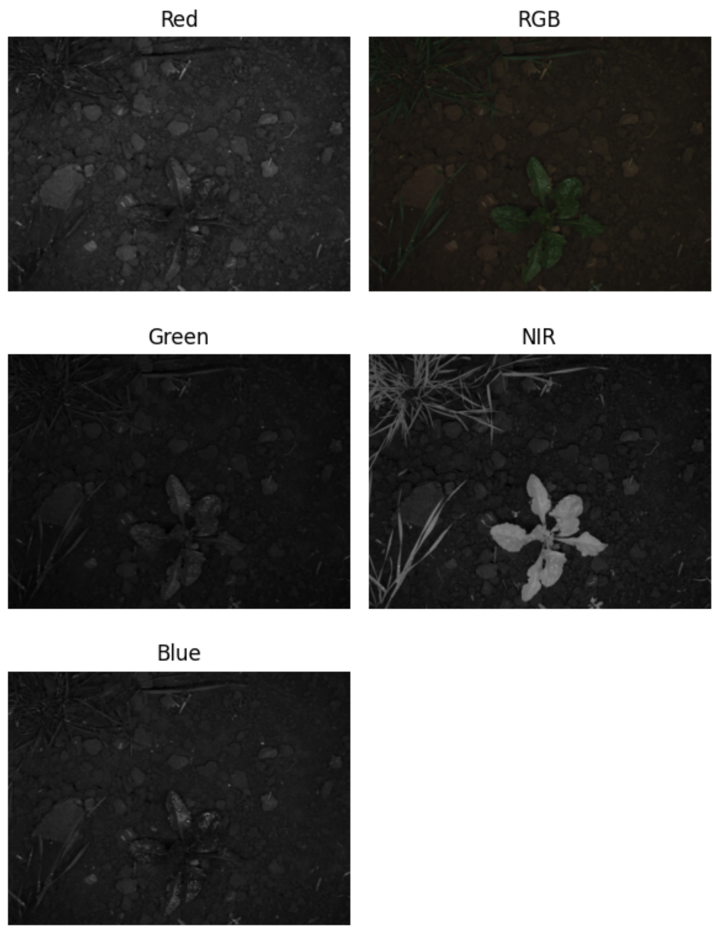

Figure 3: Different channels of RGB image along with the NIR channel

– Conclusion: Leaves are very prominent in NIR spectrum.

4 Histogram for manual threshold detection

As you can see, the leaves appear bright (i.e. higher intensity values) in the NIR image. Now plot a histogram of the pixel values of some of the NIR images. Note that this being a 2D array, you want to flatten it when passing to the histogram function. Since most of the image is background, the histogram would have tall bars for those pixels, and lower bars for the leaves. Choose a threshold value by inspecting the histogram such that pixels with value above it are likely to be leaves.

Store this in the variable: `thresh`.

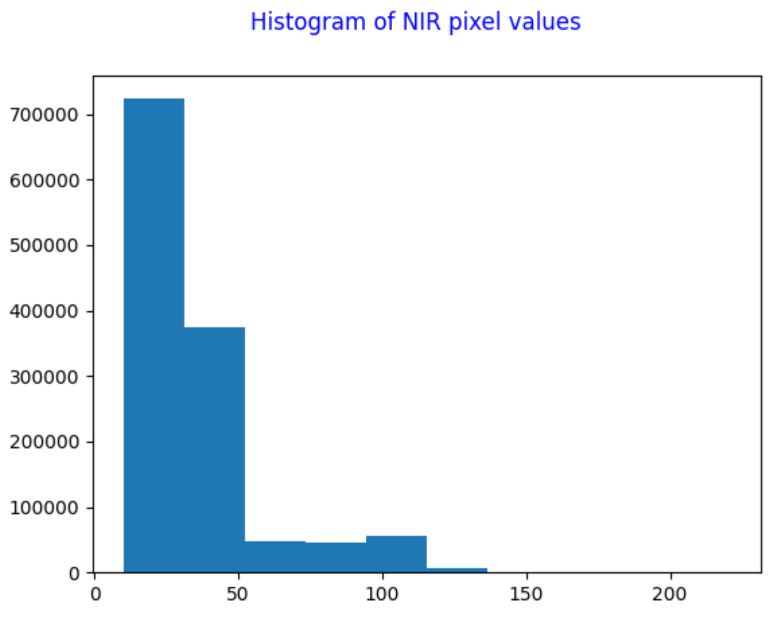

Figure 4: Histogram of pixel values in NIR image

5 Thresholding the image

Now that you have selected a threshold value, create a binary mask for each of the NIR images. Here a binary mask is a 2D array with the same shape as the original image, but with `1` (or `True`) when the pixel at the corresponding position in the original image is above threshold, and 0 otherwise. You can use regular comparison operators on numpy arrays to obtain such masks. If you picked your threshold correctly, the mask image should look like this:

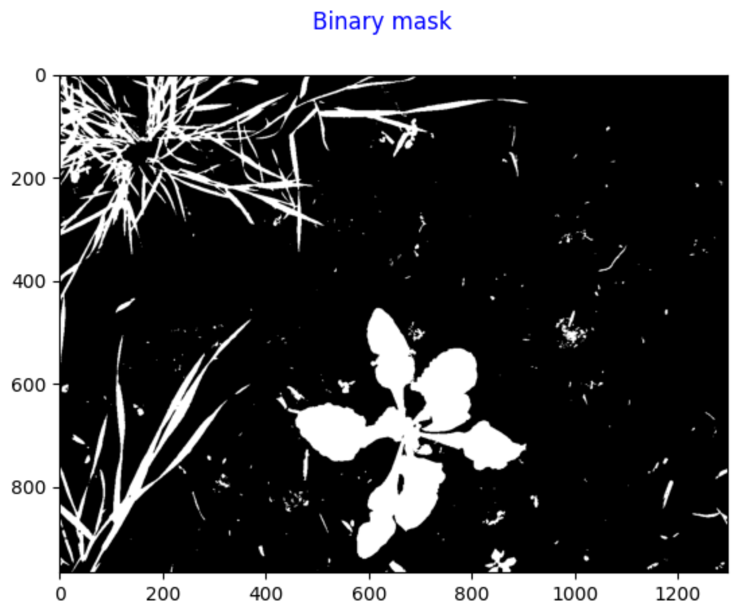

Figure 5: Binary mask from thresholding

You can clearly see the (inverse) silhouette of the sugar-beet plant along with some grass and other garbage.

6 Morphology computation

Now we want to measure the leaf sizes of the sugar beet plants from this binary image. A problem is that in a given plant, all of them are connected via a thin stem. In some other cases the stem is dark and below threshold, producing disconnected leaf shapes.

Mathematical morphology (`https://en.wikipedia.org/wiki/Mathematical_morphology`) is a classical technique for image processing that can help here. The `opening` operation can remove small objects from the foreground. It ‘opens up‘ gaps between bright features. You can use this operation to refine the binary image while keeping the size of the leaves somewhat intact.

Use `http://scipy-lectures.org/packages/scikit-image/#mathematical-morphology` as a guide. Use `morphology.binary_opening` function with a foorprint array `b = morphology.diamond(5)` (`morphology` is the submodule of `skimage` imported at the start). You may want to print `b` to see what it actually contains.

Store the result in the variable: `morphed`.

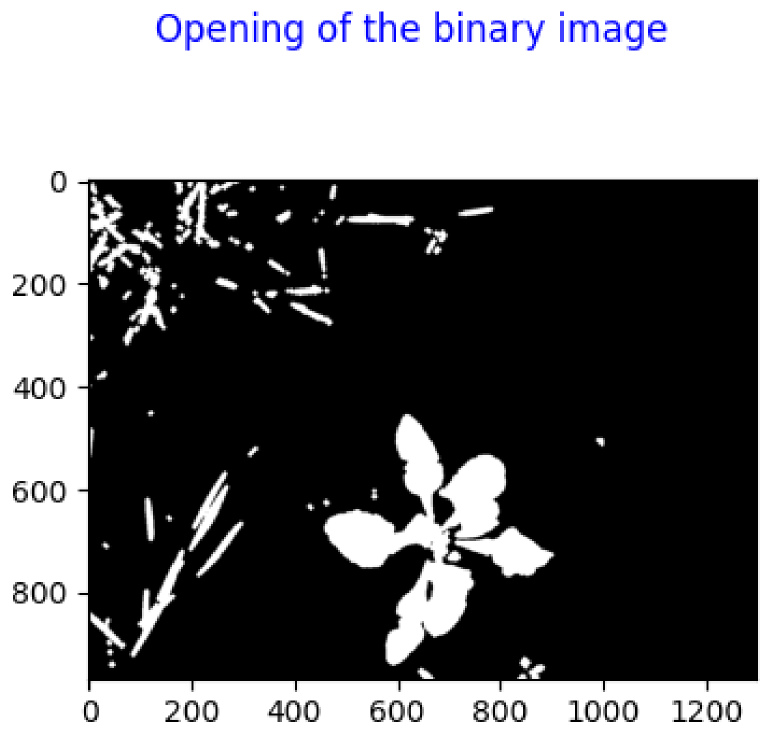

Figure 6: Opening of the binary mask image

7 Label the contiguous pixels

To identify contiguous pixels in the binary image (after opening), use `measure.label` function with the keyword argument `background=0`. This produces an array with identical shape as the image, but the white pixels are now filled with a number indicating the contiguous blob they belong to.

```
#%% Create the label image
label_im = measure.label(morphed, background=0)
fig, axes = plt.subplots(nrows=1, ncols=2, sharex='all',
↪  sharey='all')
```

```
axes[0].set_title('Color-mapped Labels')
axes[0].imshow(label_im, cmap='jet')
axes[1].imshow(nir_list[5], cmap='gray')
axes[1].set_title('Original NIR image')
fname = 'labeled_image.png'
fig.set_size_inches(8, 4)
fig.savefig(fname)
fname
```

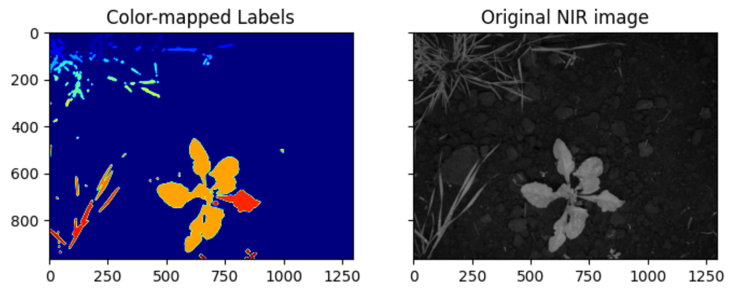

Figure 7: Labeled image (each label represented by a different color/shade)

As you can see, the beet plant has been marked in yellow/orange, the grass blades have been segmented into multiple leaves marked in various colors. In interactive matploltib window you can put the mouse pointer over a pixel and see the label value.

8 Extract region properties

Now you can extract some geometric properties of these labeled regions using `measure.regionprops`. In the code snippet below I am displaying the half-major and half-minor axes of the best fitting ellipse detected by skimage. Note that here `y` comes before `x` in the coordinates as `skimage` treats the image as a 2D array where `row` (`y`) comes before `column` (`x`).

```
#%% Extract and display region properties
fig, ax = plt.subplots()
ax.imshow(label_im)
```

```
props = measure.regionprops(label_im)
for prop in props:
    y0, x0, y1, x1 = prop.bbox
    plt.plot((x0, x1, x1, x0, x0), (y0, y0, y1, y1, y0), 'k-')
    y0, x0 = prop.centroid
    x1 = x0 + np.cos(prop.orientation) * 0.5 *
     ↪  prop.minor_axis_length
    y1 = y0 - np.sin(prop.orientation) * 0.5 *
     ↪  prop.minor_axis_length
    x2 = x0 - np.sin(prop.orientation) * 0.5 *
     ↪  prop.major_axis_length
    y2 = y0 - np.cos(prop.orientation) * 0.5 *
     ↪  prop.major_axis_length
    ax.plot((x0, x1), (y0, y1), '-m')
    ax.plot((x0, x2), (y0, y2), '-r')
fname = 'labeled_image_geom.png'
fig.savefig(fname)
fname
```

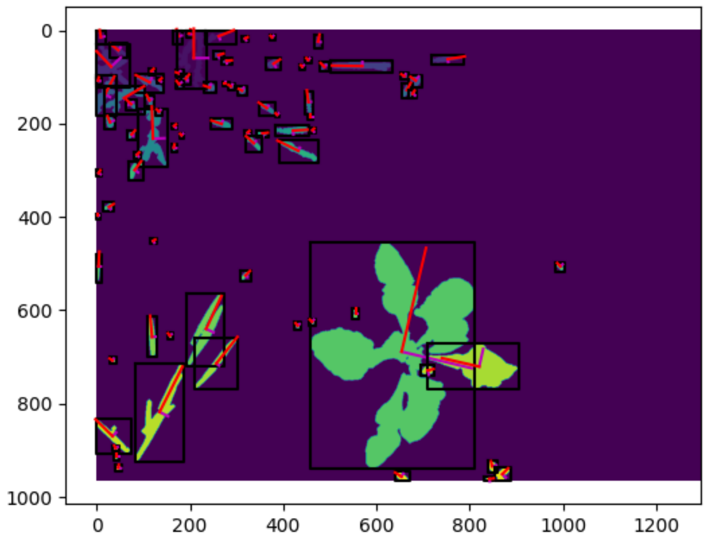

Figure 8: Some geometric properties on detected contiguous blobs

9 Cluster the data

As you can see in the image, grass blades are fit by ellipses with long major axis but short minor axis. In contrast, beet plants, when detected as a whole, have almost circular shape. Thus their major and minor axes are about the same. Therefore we can cluster the major and minor axis pairs to separate grass from beet plants, and possibly from individual leaves (when the image has only partial plant, or when the stem is too thin or dark). These are the "features" to use for clustering. Convert the major and minor axis data into an array:

```python
#%% Extract features for clustering
props = measure.regionprops(label_im)
axlength = [(prop.major_axis_length, prop.minor_axis_length)
↪  for prop in props]
axlength = np.array(axlength)
fig, ax = plt.subplots()
ax.scatter(axlength[:, 0], axlength[:, 1])
ax.set_xlabel('Major axis length')
ax.set_ylabel('Minor axis length')
fname = 'scatterplot_axlength.png'
fig.savefig(fname)
fname
```

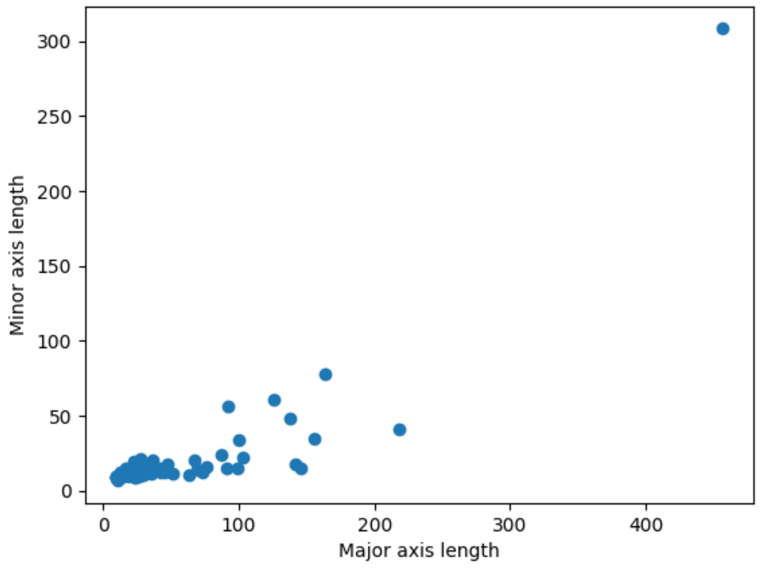

Figure 9: Scatter plot of major and minor axis lengths of detected blobs in a single image

So far I showed you the process for a single image. **Save this in a file named `robo_single_image.py`.** Upload it on codepost.

9.1 Process all the images

Do the image processing steps after thresholding in a loop for all the NIR images in the dataset and concatenate the major and minor axis pairs into a single 2d array.

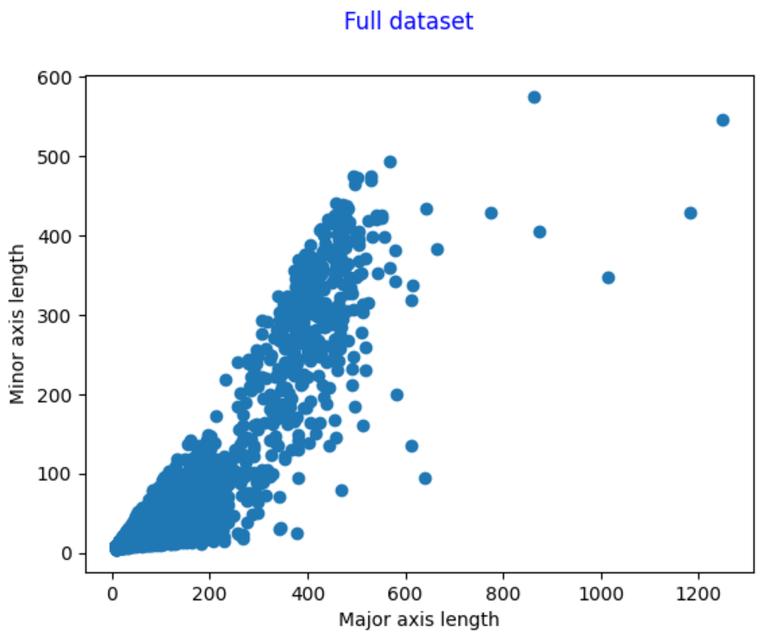

Figure 10: Scatter plot of major and minor axes of blobs detected across all images

9.2 Cluster the data

Now run k-means clustering on this full dataset. Looking at the clusters, the boundaries are not very intuitive. You may realize that just major and minor axes are not good enough for identifying the plants. In case the leaves could not be separated properly by the morphology operation, these can be misleading.

```
#%% Clustering
nc = 4
kmeans = cluster.KMeans(init='k-means++', n_clusters=nc,
↪  n_init=3, random_state=0)
kmeans.fit(feature_matrix[:, :2])
labels = kmeans.predict(feature_matrix[:, :2])
fig, ax = plt.subplots()
for label in range(nc):
    ax.scatter(feature_matrix[:, 0][labels == label],
    ↪  feature_matrix[:, 1][labels == label])
```

```
fig.suptitle('Clustered data (minor and major axes)',
↪  fontdict={'color': 'blue'})
fname = 'clusters_1.png'
fig.savefig(fname)
fname
```

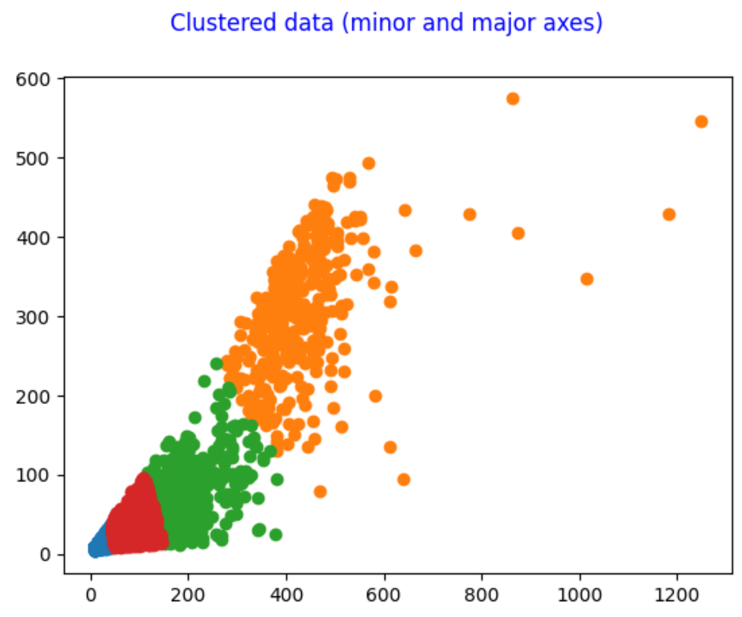

Figure 11: Initial clustering using only major and minor axes

9.3 Include area in the features

If you add the area of the contiguous regions, the clusters look more intuitive:

```
#%% Clustering with area
nc = 4
kmeans = cluster.KMeans(init='k-means++', n_clusters=nc,
↪  n_init=3, random_state=0)
kmeans.fit(feature_matrix)
labels = kmeans.predict(feature_matrix)
fig, ax = plt.subplots()
for label in range(nc):
    ax.scatter(feature_matrix[:, 0][labels == label],
     ↪   feature_matrix[:, 1][labels == label])
fig.suptitle('Clustered data (minor and major axes and area)',
 ↪   fontdict={'color': 'blue'})
fname = 'clusters_2.png'
fig.savefig(fname)
fname
```

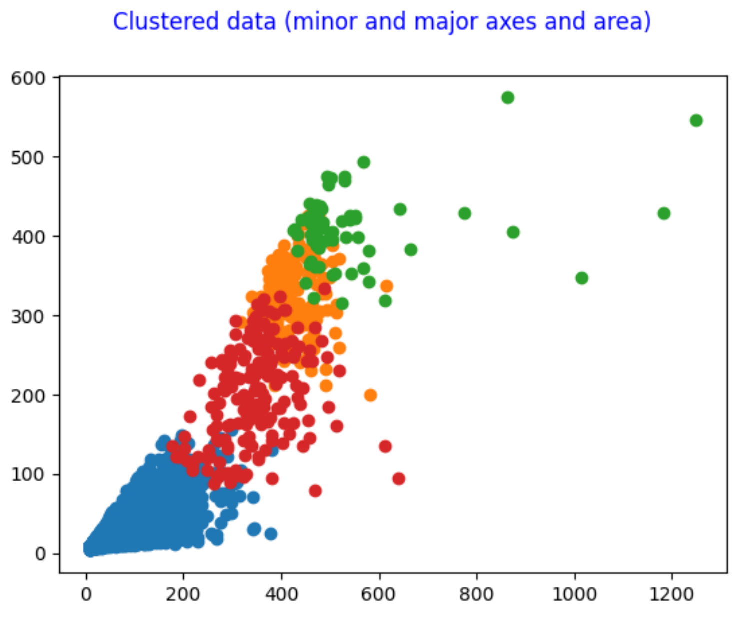

Figure 12: Clusters with area of blob included

10 Verify the clusters

Now verify the clusters in some random images. To associate the clusters with the original images you have to keep track of the image index for each row in the feature matrix as well as the measured properties. Assuming I stored the image-index of each row of the feature matrix in a list `im_idx`, and the properties in a list `props_list`, this is how I would go about checking my clusters (remember: the labels would be 0, . . . , nc -1 where nc is the number of clusters you specified):

Below I am overlaying the bounding box and half-major and half-minor axis in cluster-specific color on the NIR image.

```
#%% Check clusters in some random images
# Turning these into numpy arrays makes indexing simpler
im_idx = np.array(im_idx)
props_list = np.array(props_list)
idx = np.random.choice(len(nir_list))  # pick a random index
nir = nir_list[idx]  # image to check
# feature matrix rows corresponding to image with index idx
fidx = np.flatnonzero(im_idx == idx)
fig, ax = plt.subplots()
ax.imshow(nir, cmap='gray')
# 3 colors for 3 clusters
colors = ['orange', 'magenta', 'blue', 'yellow']
# Remember this from before??
for index in fidx:
    prop = props_list[index]
    label = labels[index]
    y0, x0, y1, x1 = prop.bbox
    plt.plot((x0, x1, x1, x0, x0), (y0, y0, y1, y1, y0), '-',
     ↪  color=colors[label])
    y0, x0 = prop.centroid
    x1 = x0 + np.cos(prop.orientation) * 0.5 *
     ↪  prop.minor_axis_length
    y1 = y0 - np.sin(prop.orientation) * 0.5 *
     ↪  prop.minor_axis_length
    x2 = x0 - np.sin(prop.orientation) * 0.5 *
     ↪  prop.major_axis_length
    y2 = y0 - np.cos(prop.orientation) * 0.5 *
     ↪  prop.major_axis_length
    ax.plot((x0, x1), (y0, y1), '-', color=colors[label])
    ax.plot((x0, x2), (y0, y2), '-', color=colors[label])
fig.suptitle('Labeled data', fontdict={'color': 'blue'})
fname = 'clusters_check.png'
fig.savefig(fname)
fname
```

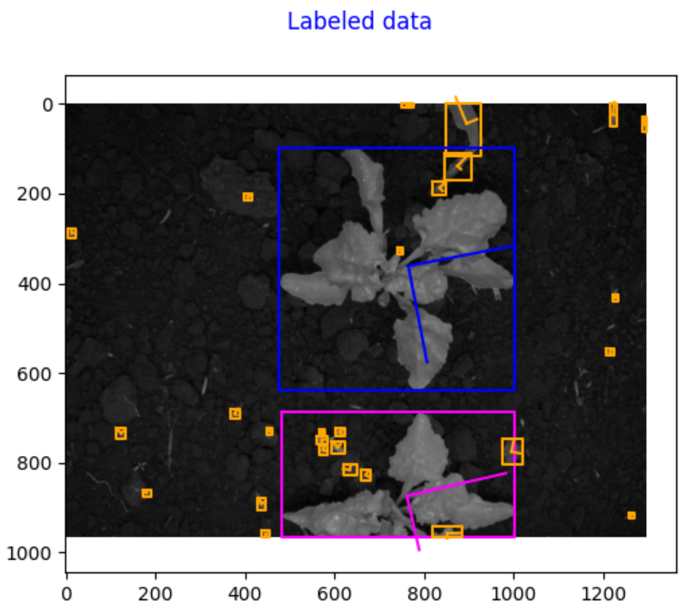

Figure 13: Verification of clusters against original image

11 Statistics on clustered data

You should be able to identify which cluster best matches the whole sugar beet plant (not isolated leaf, not half of the plant outside the captured area). Now do the following tasks:

11.1 Compute the mean and standard deviation

of minor and major axes and the blob area of each cluster. **Save these in the array variables: `mean_major_axis`, `sd_major_axis`, `mean_minor_axis`, `sd_minor_axis`, `mean_blob_area`, `sd_blob_area`,** so that `mean_blob_area[i]` has the mean blob area of objects in cluster `i`, and so on.

11.2 Do a hypothesis testing

to compare the means of the clusters corresponding to whole sugar beet plants and grass/garbage. **Save the p-value in the variable: `p_value`.**

11.3 Save the code

that loops through all the images, extracting all the blobs and clustering them, and doing the hypothesis testing, **into a file named `robo_multi_image.py`.**

12 Bibliography

Chebrolu, Nived and Lottes, Philipp and Schaefer, Alexander and Winterhalter, Wera and Burgard, Wolfram and Stachniss, Cyrill (2017). *Agricultural Robot Dataset for Plant Classification, Localization and Mapping on Sugar Beet Fields*, SAGE Publications Ltd STM.