Quantitative Big Imaging - Advanced segmentation

Anders Kaestner

Mar 19, 2025

CONTENTS

This is the lecture notes fot the 5th lecture of the Quantitative big imaging class given during the spring semester 2025 at ETH Zurich, Switzerland.

0.1 Advanced segmentation

0.1.1 Today's Outline

- Motivation
- Many Samples
- Difficult Samples
- Training / Learning
- Thresholding
- Automated Methods
- Hysteresis Method

- Feature Vectors
- K-Means Clustering
- Superpixels
- Working with Segmented Images
- Contouring

0.1.2 Load some needed modules

```
from skimage.io import imread
import matplotlib.pyplot as plt

import numpy as np
from skimage.morphology import dilation, opening, disk
from collections import OrderedDict
from skimage.data import page
import skimage.filters as flt
import pandas as pd
from skimage.filters import gaussian, median, threshold_triangle
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
from matplotlib.colors import ListedColormap
import matplotlib.colors as colors
%matplotlib inline

colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
```

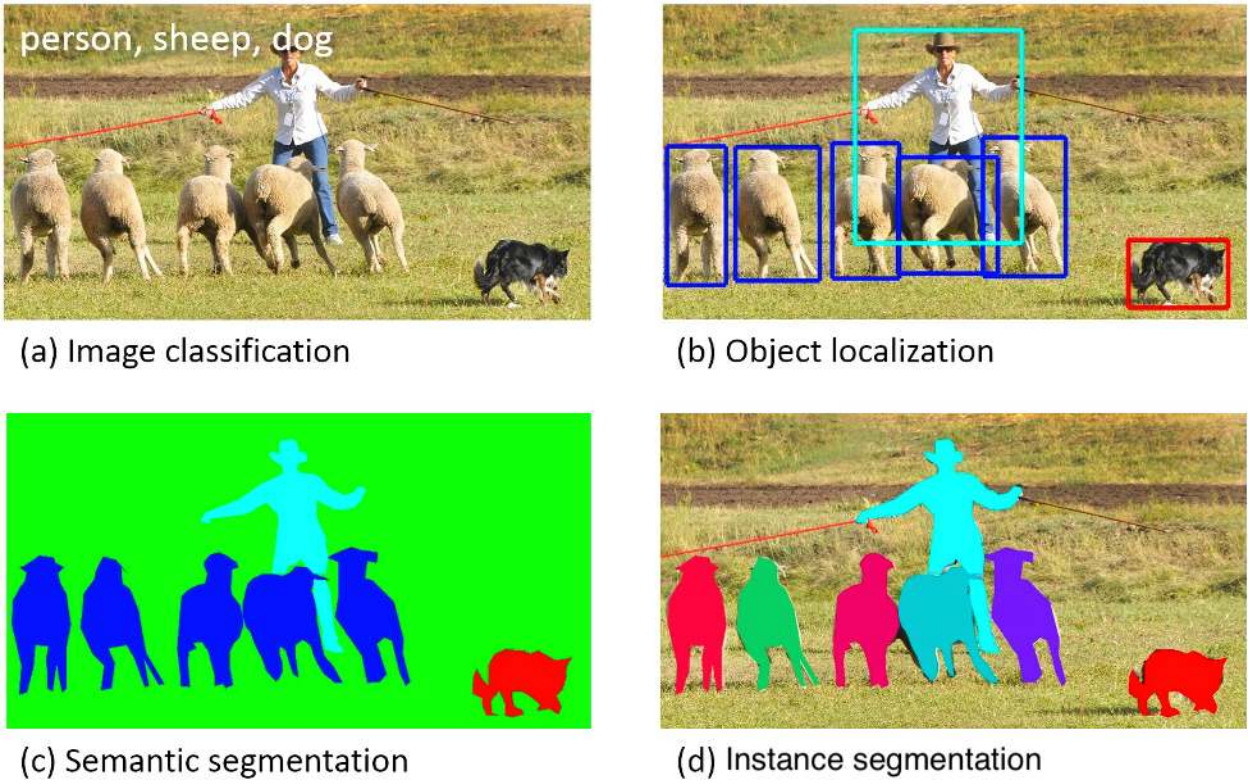
0.1.3 Previously on QBI

- Image acquisition and representations
- Enhancement and noise reduction
- Understanding models and interpreting histograms
- Ground Truth and ROC Curves

- Choosing a threshold
- Examining more complicated, multivariate data sets
- Improving segementation with morphological operations
- Filling holes
- Connecting objects
- Removing Noise
- Partial Volume Effect

0.1.4 Different types of segmentation

When we talk about image segmentation there are different meanings to the word. In general, segmenation is an operation that marks up the image based on pixels or pixel regions. This is a task that has been performed since beginning of image processing as it is a natural step in the workflow to analyze images - we must know which regions we want to analyze and this is a tedious and error prone task to perform manually. Looking at the figure below we two type of segmentation.



From Lin et al. 2015

- Object detection - identifies regions containing an object. The exact boundary does not matter so much here. We are only interested in a bounding box.
- Semantic segmentation - classifies all the pixels of an image into meaningful classes of objects. These classes are "semantically interpretable" and correspond to real-world categories. For instance, you could isolate all the pixels associated with a cat and color them green. This is also known as dense prediction because it predicts the meaning of each pixel.
- Instance segmentation - Identifies each instance of each object in an image. It differs from semantic segmentation in that it doesn't categorize every pixel. If there are three cars in an image, semantic segmentation classifies all the cars as one instance, while instance segmentation identifies each individual car.

0.2 Where segmentation fails

Segmentation using a single threshold is sensitive to different conditions…

In fact, segmentation is rarely an obvious task. What you want to find is often obscured by other image features and unwanted artefacts from the experiment technique. If you take a glance at the painting by Bev Doolittle, you quickly spot the red fox in the middle. Looking a little closer at the painting, you'll recognize two american indians on their spotted ponies. This example illustrates the problems you will encounter when you start to work with image segmentation.

Fig. 1: Cases making the segmentation task harder than just applying a single threshold.

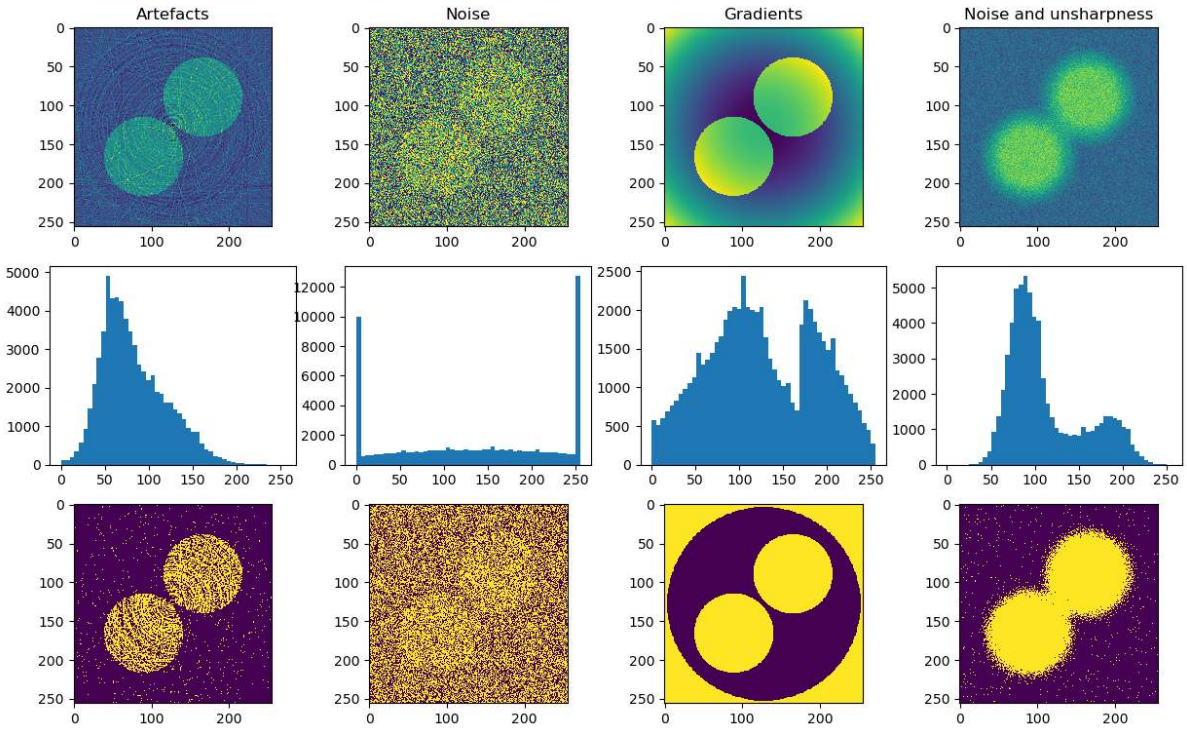
Woodland Encounter Bev Doolittle, 1981

0.2.1 Typical image features that make life harder

The basic segmentation shown in the previous lecture can only be used under good conditions when the classes are well separated. Images from many experiments are unfortunately not well-behaved in many cases. The figure below shows four different cases when an advanced technique is needed to segment the image. Neutron images often show a combination of all cases.

The impact on the segmentation performance of all these cases can be reduced by proper experiment planning. Still, sometimes these improvements are not fully implemented in the experiment to be able to fulfill other experiment criteria.

```
files  = ['class_art_in.png','class_wgn_in.png','class_cupped_in.png','class_wgn_
 ↪smooth_in.png']
titles = ['Artefacts','Noise','Gradients','Noise and unsharpness']

plt.figure(figsize=[15,9])
for i in range(4) :
    img = imread('data/'+files[i]).mean(axis=2)
    plt.subplot(3,4,i+1), plt.imshow(img,interpolation='none'), plt.title(titles[i])
    plt.subplot(3,4,i+5), plt.hist(img.ravel(),bins=50)
    plt.subplot(3,4,i+9), plt.imshow(120<img,interpolation='none')
```

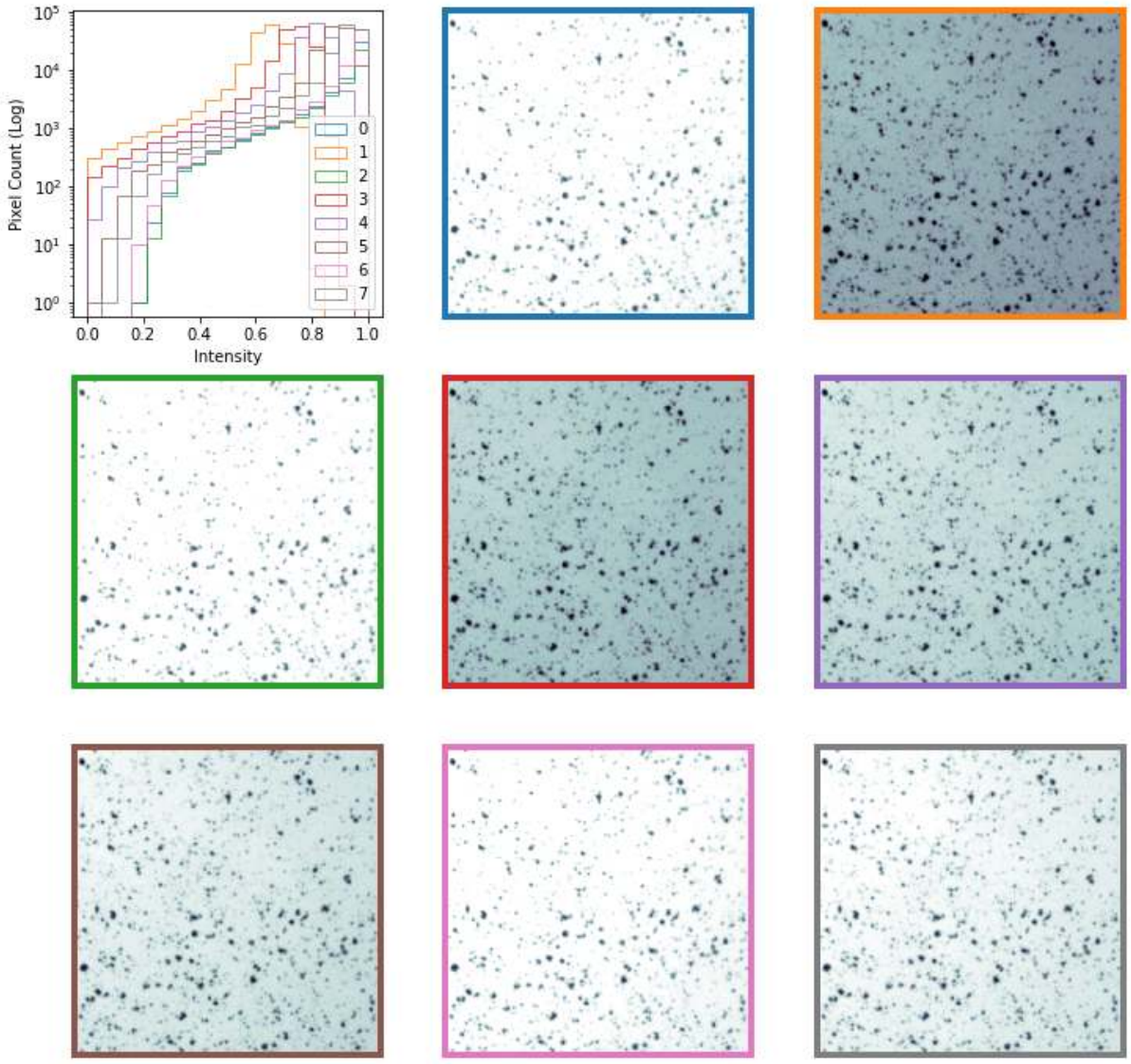


0.2.2 Inconsitent illumination on real data

Under realistic experiment conditions it may happen that the illumation fluctuates. This results in inconsistently illuminated images.

In this particular example of a cell colony, there is a random bias added to the images.

```
cell_img = imread("figures/Cell_Colony.jpg")/255.0
np.random.seed(2018)
m_cell_imgs = [cell_img+k for k in np.random.uniform(-0.25, 0.25, size = 8)]
fig, m_axs = plt.subplots(3, 3, figsize = (12, 12), dpi = 72)
ax1 = m_axs[0,0]
for i, (c_ax, c_img) in enumerate(zip(m_axs.flatten()[1:], m_cell_imgs)):
    ax1.hist(c_img.ravel(), np.linspace(0, 1, 20),
             label = '{}'.format(i), alpha = 0.75, histtype = 'step')
    c_ax.imshow(c_img, cmap = 'bone', vmin = 0, vmax = 1)
#     c_ax.axis('off')
    c_ax.set_xticks([]);
    c_ax.set_yticks([]);
    for axis in ['top','bottom','left','right']:
        c_ax.spines[axis].set_linewidth(4)
        c_ax.spines[axis].set_color(colors[i])

ax1.set_yscale('log', nonpositive = 'clip')
ax1.set_ylabel('Pixel Count (Log)')
ax1.set_xlabel('Intensity')
ax1.legend();
```



Apply a treshold

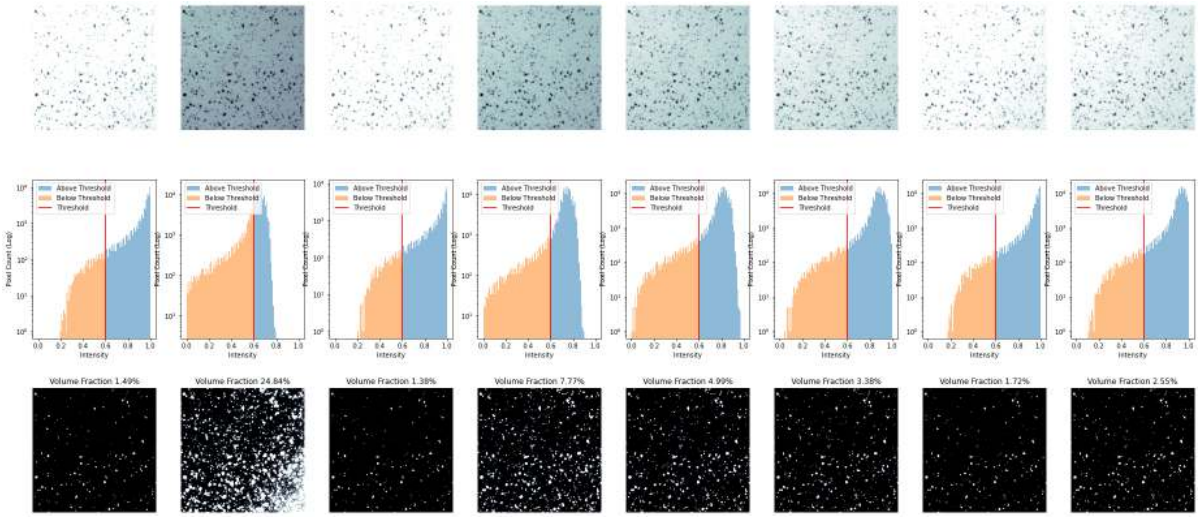
When we apply a constant threshold of 0.6 to the images with different illuminations we just looked at you will see that there are great differences in how the cells are represented.

```
fig, m_axs = plt.subplots(3,len(m_cell_imgs),
                          figsize = (4*len(m_cell_imgs), 5*3),
                          dpi = 72)
THRESH_VAL = 0.6

for i, ((ax0, ax1, ax2), c_img) in enumerate(zip(m_axs.transpose(), m_cell_imgs)):
    ax0.imshow(c_img, cmap = 'bone', vmin = 0, vmax = 1)
    ax0.axis('off')
    ax1.hist(c_img.ravel()[c_img.ravel()>THRESH_VAL],
             np.linspace(0, 1, 100),
             label = 'Above Threshold',
             alpha = 0.5)
```

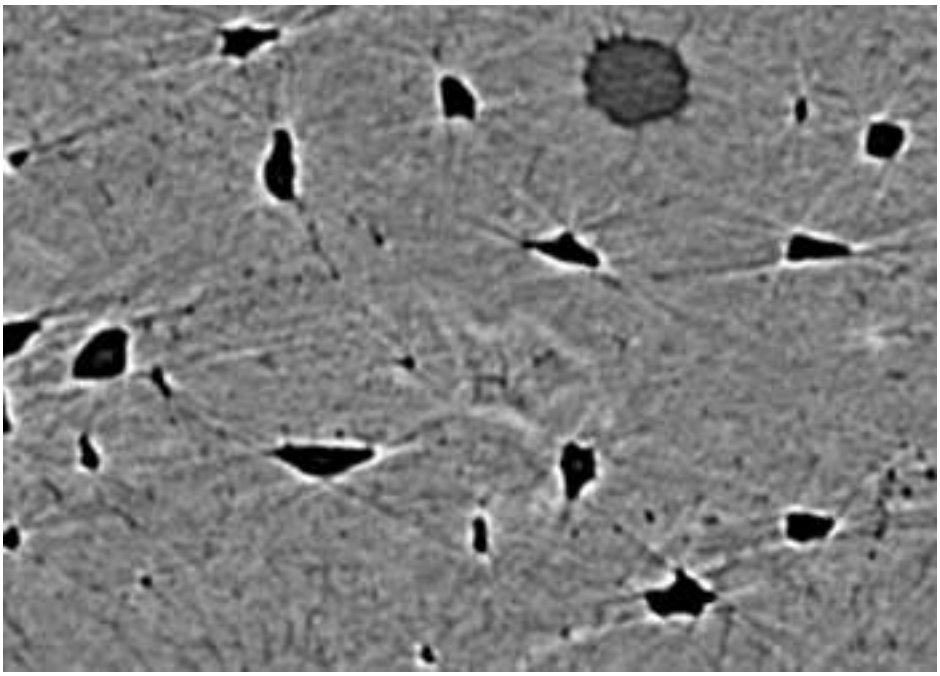
(continues on next page)

(continued from previous page)

```
    ax1.hist(c_img.ravel()[c_img.ravel()<THRESH_VAL],
             np.linspace(0, 1, 100),
             label = 'Below Threshold',
             alpha = 0.5)
    ax1.axvline(THRESH_VAL,color='r',label='Threshold');
    ax1.set_yscale('log', nonpositive = 'clip'); ax1.set_ylabel('Pixel Count (Log)');␣
↪ax1.set_xlabel('Intensity'); ax1.legend()
    ax2.imshow(c_img<THRESH_VAL, cmap = 'bone', vmin = 0, vmax = 1)
    ax2.set_title('Volume Fraction {0:2.2f}%'.format(100*np.mean(c_img.ravel()<THRESH_
↪VAL))), ax2.axis('off');
```



0.2.3 Where segmentation fails: Canaliculi



Here is a bone slice

1. Find the larger cellular structures (osteocyte lacunae)
2. Find the small channels which connect them together

The first task is easy using a threshold and size criteria (we know how big the cells should be)

The second is much more difficult because the small channels having radii on the same order of the pixel size are obscured by

- partial volume effects
- and noise.

0.2.4 Where segmentation fails: Brain Cortex

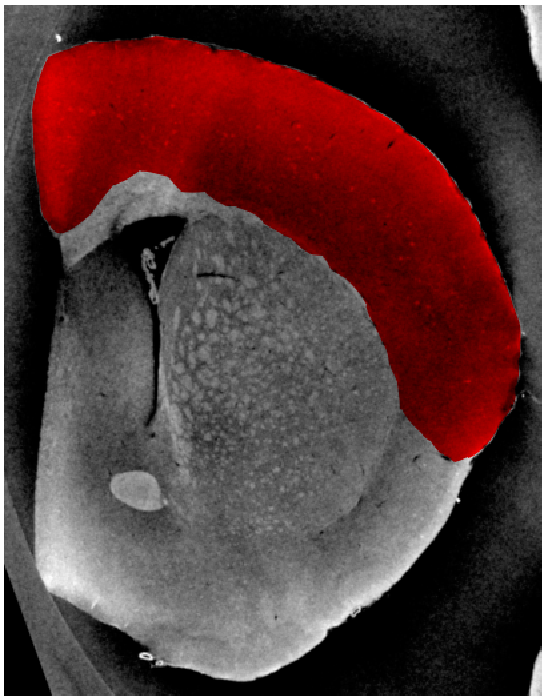


Fig. 2: Brain image with masked cortex.

- The cortex is barely visible to the human eye
- Tiny structures hint at where cortex is located

Our problem

- A simple threshold is insufficient to finding the cortical structures
- Other filtering techniques are unlikely to magicially fix this problem

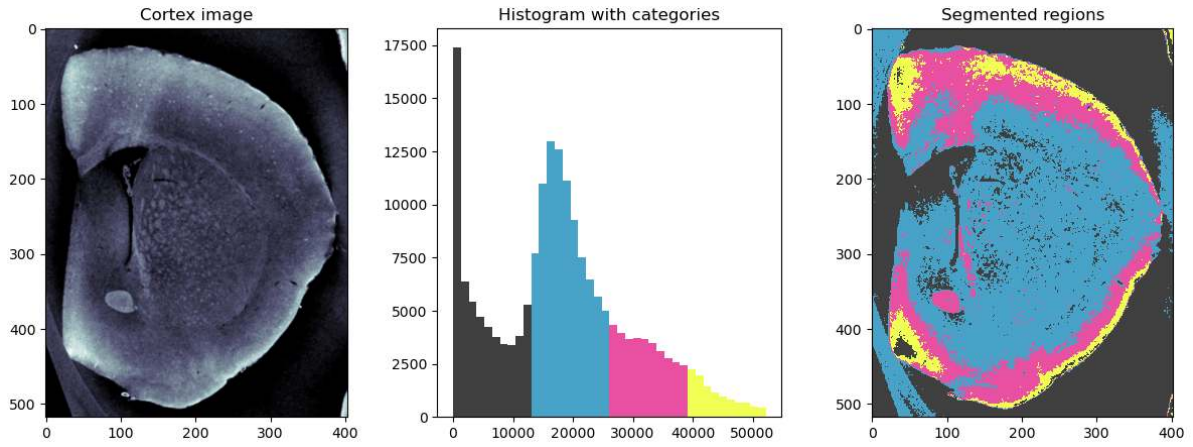
Segmentation - Apply thresholds

We saw in the previous lecture that we can segment images using one or more thresholds. This does not work on many images, like the brain cortex example here.

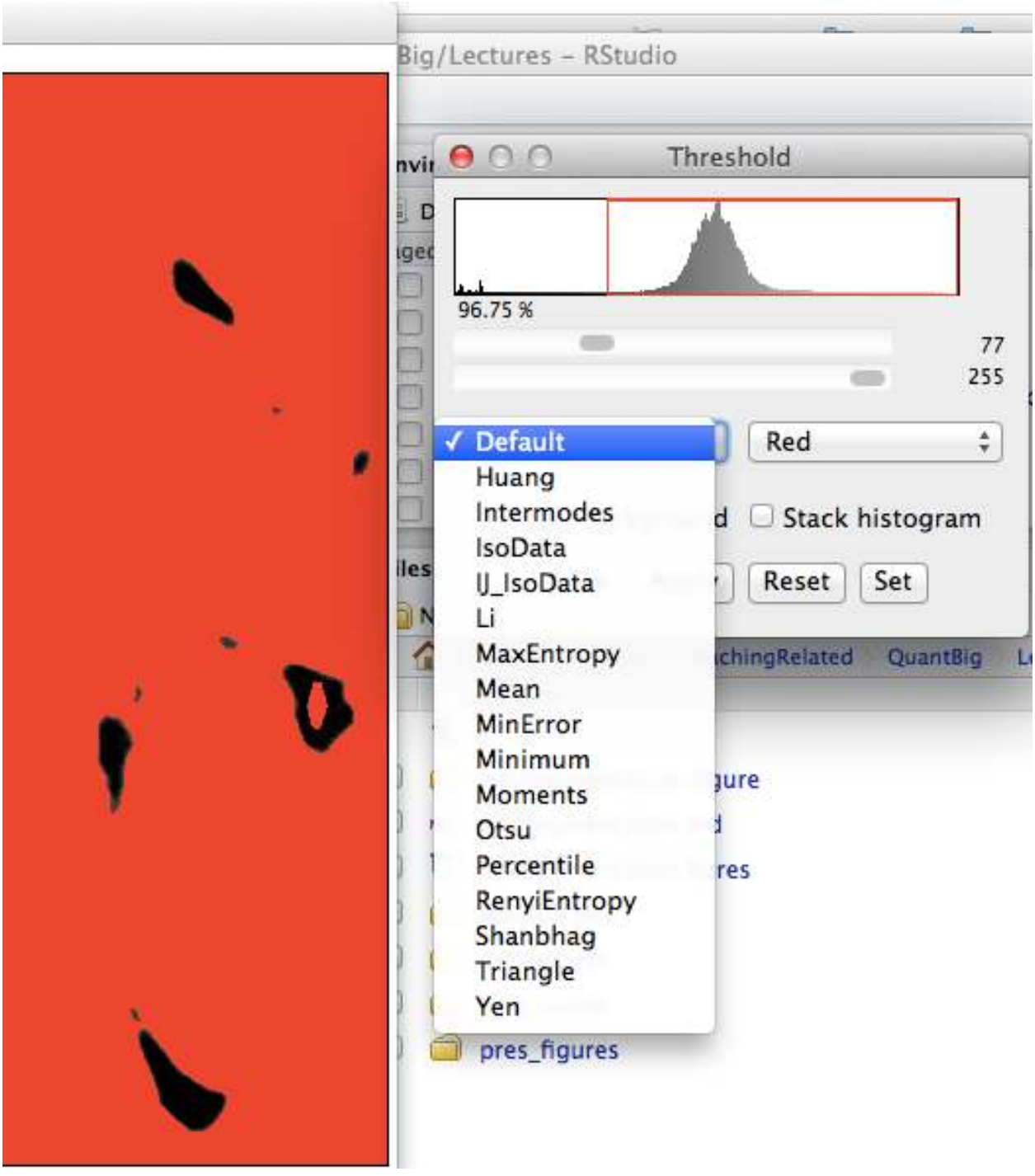
The principle of thresholding is a mapping of, mostly, a gray level or gray level interval to a specific class. Manually, this is done using the histogram of the image that guide the choice of threshold.

```
%matplotlib inline
from skimage.io import imread
import matplotlib.pyplot as plt
import numpy as np
```

```
cortex_img = imread("figures/cortex.png")
np.random.seed(2018)
fig, (ax1, ax2, ax3) = plt.subplots(1, 3,
                                    figsize = (15, 5))
ax1.imshow(cortex_img, cmap = 'bone')
ax1.set_title('Cortex image')

clst = np.array([[64,64,64],
                 [73,162,199],
                 [234,80,162],
                 [241,254,83]])/255.0
cmap = ListedColormap(clst)

thresh_vals = np.linspace(cortex_img.min(), cortex_img.max(), 4+2)[:-1]
out_img = np.zeros_like(cortex_img)
for i, (t_start, t_end) in enumerate(zip(thresh_vals, thresh_vals[1:])):
    thresh_reg = (cortex_img>t_start) & (cortex_img<t_end)
    ax2.hist(cortex_img.ravel()[thresh_reg.ravel()],color=clst[i])
    out_img[thresh_reg] = i
ax2.set_title('Histogram with categories')
ax3.imshow(out_img, cmap = cmap,interpolation='none');
ax3.set_title('Segmented regions');
```



0.2.5 Automated Threshold Selection



Given that applying a threshold is such a common and signficant step, there have been many tools developed to automatically (unsupervised) perform it. A particularly important step in setups where images are rarely consistent such as outdoor imaging which has varying lighting (sun, clouds). The methods are based on several basic principles.

0.3 Automated Methods

0.3.1 Histogram-based methods

Just like we visually inspect a histogram an algorithm can examine the histogram and find local minimums between two peaks, maximum / minimum entropy and other factors

- Otsu, Isodata, Intermodes, etc

0.3.2 Image based Methods

These look at the statistics of the thresheld image themselves (like entropy) to estimate the threshold

0.3.3 Results-based Methods

These search for a threshold which delivers the desired results in the final objects. For example if you know you have an image of cells and each cell is between 200-10000 pixels the algorithm runs thresholds until the objects are of the desired size

- More specific requirements need to be implemented manually

0.3.4 Histogram-based Methods

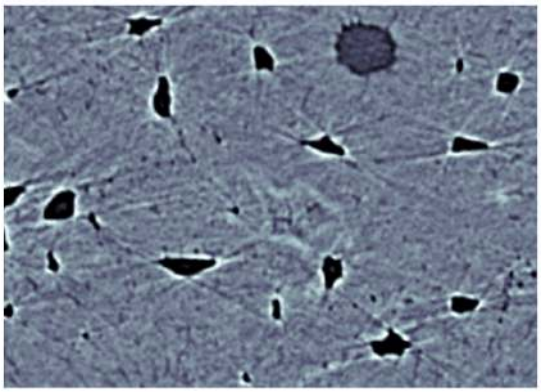
Taking a typical image of a bone slice, we can examine the variations in calcification density in the image

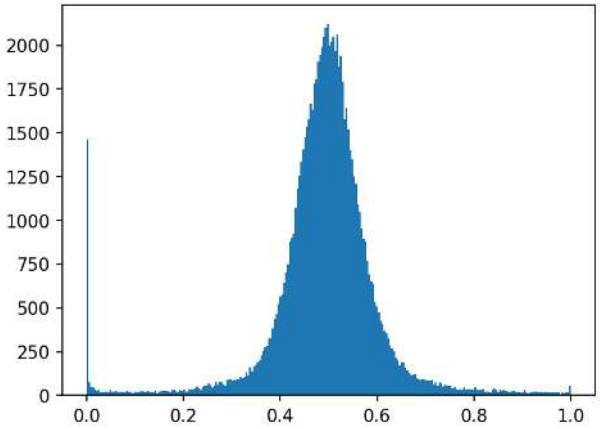
---

We can see in the histogram that there are two peaks, one at 0 (no absorption / air) and one at 0.5 (stronger absorption / bone)

```
%matplotlib inline
from skimage.io import imread
import matplotlib.pyplot as plt
import numpy as np
```

```
bone_img = imread("figures/bonegfiltslice.png")/255.0
np.random.seed(2018)

fig, (ax1, ax2) = plt.subplots(1, 2,figsize = (12, 4), dpi = 150)
ax1.imshow(bone_img, cmap = 'bone')
ax1.axis('off')
ax2.hist(bone_img.ravel(), bins=256);
```





0.3.5 Histogram based segmentation algorithms

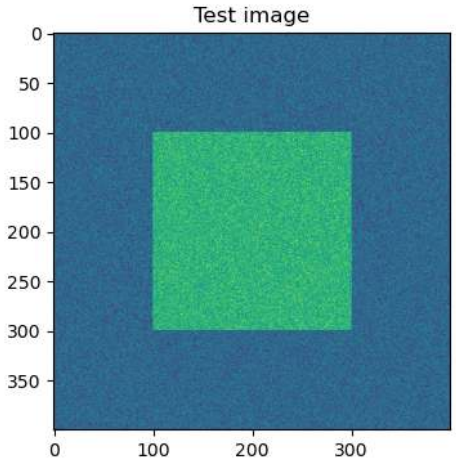
There are many thresholding algorithms based on the image histogram

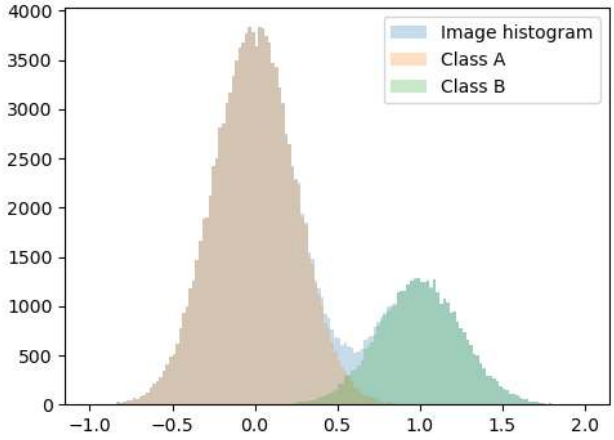
- Intermodes
- Otsu
- Isodata

We start with a toy example:

```
N=400
img=np.zeros((N,N))
img[N//4:(N//4+N//2),N//4:(N//4+N//2)]=1
nimg = img + np.random.normal(0,0.25,size=img.shape)
```

```
fig,ax=plt.subplots(1,2,figsize=(12,4))

ax[0].imshow(nimg)
ax[0].set_title('Test image')
bins=150
himg=ax[1].hist(nimg.ravel(),bins=bins,alpha=0.25, range=(-1,2), label='Image␣
↪histogram')
ha=ax[1].hist(nimg[img==0].ravel(),bins=bins,alpha=0.25, range=(-1,2),label='Class A')
hb=ax[1].hist(nimg[img==1].ravel(),bins=bins,alpha=0.25, range=(-1,2),label='Class B')
ax[1].legend();
```

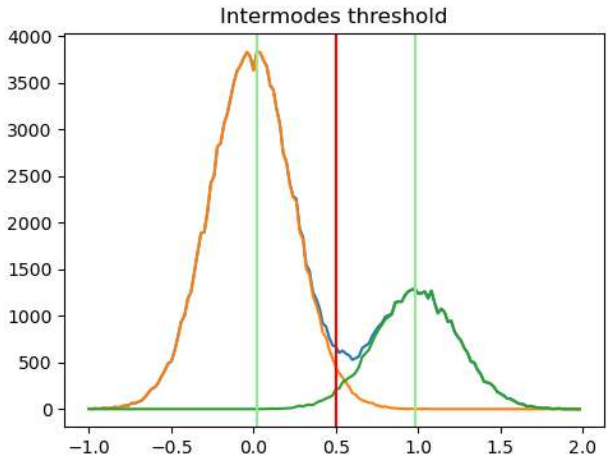


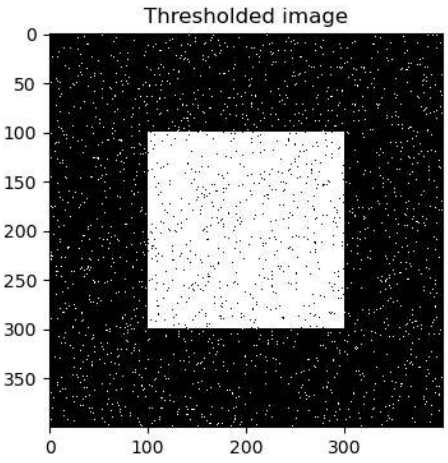


Intermodes

Take the point between the two modes (peaks) in the histogram

```
fig,ax = plt.subplots(1,2,figsize=[12,4])

ax[0].plot(himg[1][:-1],himg[0])
ax[0].plot(ha[1][:-1],ha[0])
ax[0].plot(hb[1][:-1],hb[0])

maxA=np.argmax(ha[0])
maxB=np.argmax(hb[0])
th=(maxB+maxA)//2

ax[0].axvline(ha[1][maxA],color='lightgreen')
ax[0].axvline(hb[1][maxB],color='lightgreen')
ax[0].axvline(ha[1][th],color='red')
ax[0].set_title('Intermodes threshold')

ax[1].imshow(ha[1][th]<nimg,cmap='gray',interpolation='none')
ax[1].set_title('Thresholded image');
```

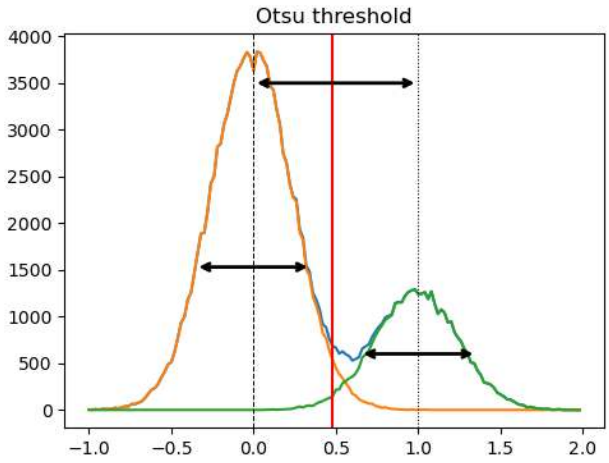


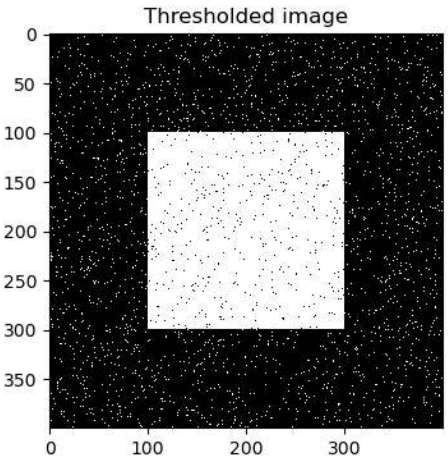


Otsu

Search and minimize intra-class (within) variance $\sigma_w^2(t) = \omega_{bg}(t)\sigma_{bg}^2(t) + \omega_{fg}(t)\sigma_{fg}^2(t)$

```
# This code is made for illustrating the behaviour of the Otsu method but does not␣
 ↪solve the Otsu problem
fig,ax = plt.subplots(1,2,figsize=[12,4])

from skimage.filters import threshold_otsu
ax[0].plot(himg[1][:-1],himg[0])
ax[0].plot(ha[1][:-1],ha[0])
ax[0].plot(hb[1][:-1],hb[0])

mA=nimg[img<0.5].mean()
sA=nimg[img<0.5].std()
mB=nimg[0.5<img].mean()
sB=nimg[0.5<img].std()

th = threshold_otsu(nimg)


def darrow(ax,a,b,y) :
    ax.annotate(
    '',
    xy=(a, y),
    xytext=(b, y),
    arrowprops=dict(arrowstyle='<->', lw=2, color='black')
    )

def sdarrow(ax,m,s,y) :
    darrow(ax,m-np.sqrt(2)*s,m+np.sqrt(2)*s,y)

ax[0].axvline(mA,color='black',linestyle='--',linewidth=0.75)
ax[0].axvline(mB,color='black',linestyle=':',linewidth=0.75)
ax[0].axvline(th,color='red')
ax[0].set_title('Otsu threshold')
sdarrow(ax[0],mA,sA,y=1530)
sdarrow(ax[0],mB,sB,y=600)
darrow(ax[0],mA,mB,3500)

ax[1].imshow(th<nimg,cmap='gray',interpolation='none')
ax[1].set_title('Thresholded image');
```

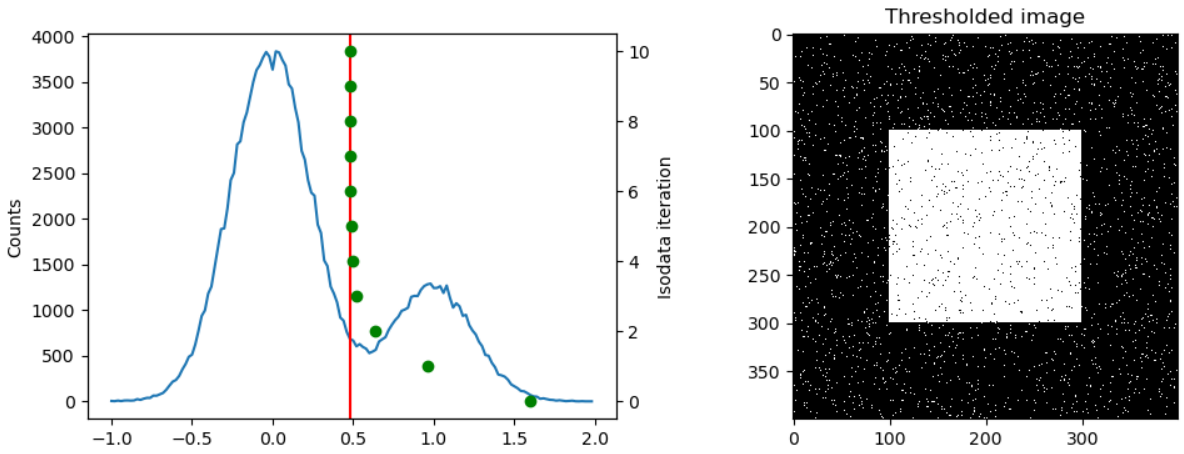




Isodata

- Initialize with: thresh $= \frac{\max(img)+\min(img)}{2}$
- *while* the thresh is changing
  - $bg = img <$ thresh, $obj = img >$ thresh
  - Update thresh $= (\text{avg}(bg) + \text{avg}(obj))/2$

```
def isodata(img) :
    th = [(img.max()-img.min())/2]

    th.append((img[img<th[-1]].mean()+img[th[-1]<img].mean())/2)

    while th[-1]!=th[-2] :
        th.append((img[img<th[-1]].mean()+img[th[-1]<img].mean())/2)

    return th[-1],th

th,history = isodata(nimg)

fig,ax=plt.subplots(1,2,figsize=(12,4))

ax[0].plot(himg[1][:-1],himg[0])
ax[0].axvline(history[-1],color='r')
ax[0].set_ylabel('Counts')
ax0r=ax[0].twinx()
ax0r.plot(history,np.arange(len(history)),'og')
ax0r.set_ylabel('Isodata iteration');

ax[1].imshow(th<nimg,cmap='gray',interpolation='none')
ax[1].set_title('Thresholded image');
```



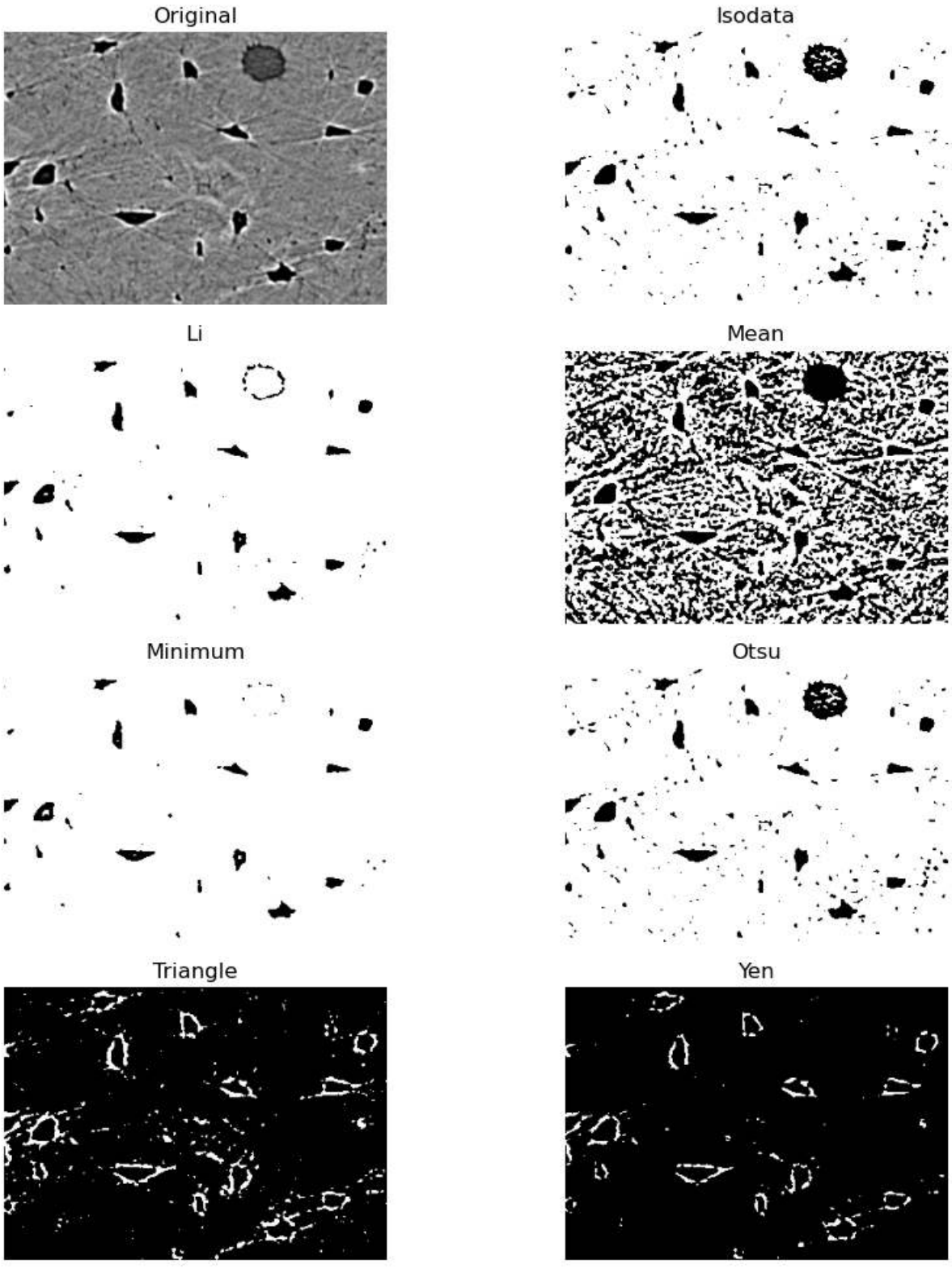
0.3.6 Try All Thresholds

- opencv2
- scikit-image

There are many methods and they can be complicated to implement yourself.

Both Fiji and scikit-image offers many of them as built in functions so you can automatically try all of them on your image.

```
bone_img = imread("figures/bonegfiltslice.png")/255.0

from skimage.filters import try_all_threshold

fig, ax = try_all_threshold(bone_img, figsize=(10, 10), verbose=False);
```


Original
Isodata
Li
Mean
Minimum
Otsu
Triangle
Yen

0.3.7 Pitfalls of different thresholding methods

While an incredibly useful tool, there are many potential pitfalls to these automated techniques.

Histogram-based

These methods are very sensitive to the distribution of pixels in your image and may work really well on images with equal amounts of each phase but work horribly on images which have very high amounts of one phase compared to the others

Image-based

These methods are sensitive to noise and a large noise content in the image can change statistics like entropy significantly.

Results-based

These methods are inherently biased by the expectations you have.

- If you want to find objects between 200 and 1000 pixels - you will!
- they just might not be anything meaningful.

0.3.8 Realistic Approaches for Dealing with these Shortcomings

Imaging science rarely represents the ideal world and will never be 100% perfect. At some point we need to write our master's thesis, defend, or publish a paper. These are approaches for more qualitative assessment we will later cover how to do this a bit more robustly with quantitative approaches

Model-based

One approach is to

- try and simulate everything (including noise) as well as possible
- and to apply these techniques to many realizations of the same image
- and qualitatively keep track of how many of the results accurately identify your phase or not.

Hint: >95% seems to convince most biologists

Sample-based

- Apply the methods to each sample and keep track of which threshold was used for each one.
- Go back and apply each threshold to each sample in the image
- and keep track of how many of them are correct enough to be used for further study.

Worst-case Scenario

Come up with the worst-case scenario (noise, misalignment, etc) and assess how inacceptable the results are. Then try to estimate the quartiles range (75% - 25% of images).

0.3.9 Hysteresis Thresholding

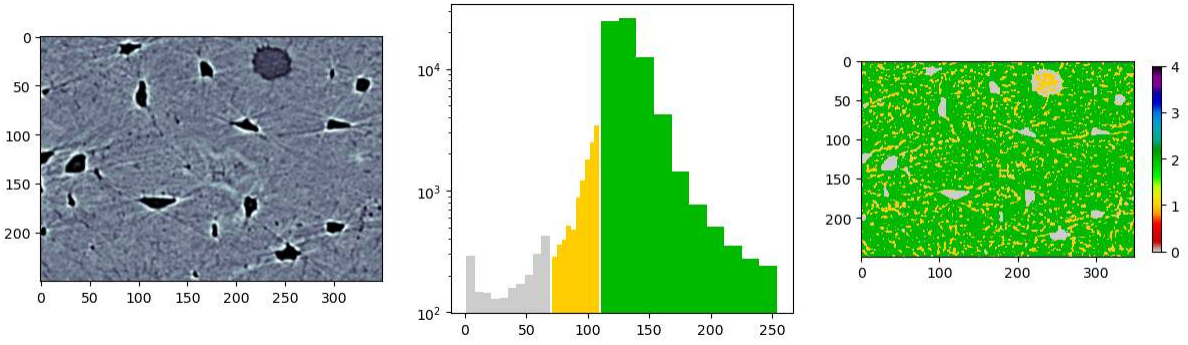
For some images a single threshold does not work

- large structures are very clearly defined
- smaller structures are difficult to differentiate (see partial volume effect)

ImageJ Source

```
%matplotlib inline
from skimage.io import imread
import matplotlib.pyplot as plt
import numpy as np
```

```
bone_img = imread("figures/bonegfiltslice.png")
np.random.seed(2018)
fig, (ax1, ax2, ax3) = plt.subplots(1, 3,
                                    figsize = (15, 4))
cmap = plt.cm.nipy_spectral_r
ax1.imshow(bone_img, cmap = 'bone')
thresh_vals = [0, 70, 110, 255]
out_img = np.zeros_like(bone_img)
for i, (t_start, t_end) in enumerate(zip(thresh_vals, thresh_vals[1:])):
    thresh_reg = (bone_img>t_start) & (bone_img<t_end)
    ax2.hist(bone_img.ravel()[thresh_reg.ravel()], color = cmap(i/(len(thresh_vals))))
    out_img[thresh_reg] = i
ax2.set_yscale("log", nonpositive='clip')
th_ax = ax3.imshow(out_img, cmap = cmap, vmin = 0, vmax = len(thresh_vals),
 ↪interpolation='none')
plt.colorbar(th_ax,shrink=0.6);
```



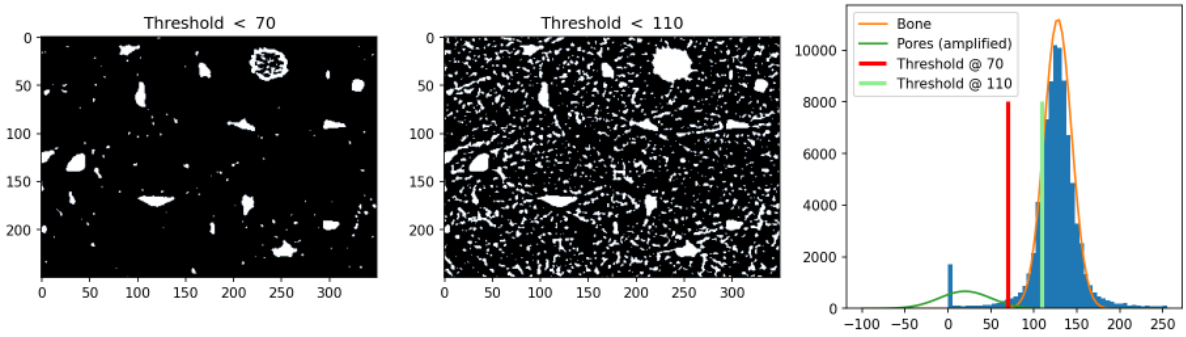
0.3.10 Goldilocks Situation

Here we end up with a goldilocks situation

- Mama bear and Papa Bear
- one is too low and the other is too high

The Goldilocks principle

```
fig, (ax1, ax2,ax3) = plt.subplots(1, 3, figsize = (15, 4), dpi=150)
ax1.imshow(bone_img<thresh_vals[1], cmap = 'bone')
ax1.set_title('Threshold $<$ %d' % (thresh_vals[1]))
ax2.imshow(bone_img<thresh_vals[2], cmap = 'bone')
ax2.set_title('Threshold $<$ %d' % (thresh_vals[2]));
import scipy.stats as stats
ax3.hist(bone_img.ravel(),bins=50);
x=np.linspace(-100,255,100)
ax3.plot(x,4.5e5*stats.norm.pdf(x,128,16),label="Bone")
ax3.plot(x,5e4*stats.norm.pdf(x,20,30),label="Pores (amplified)")
ax3.vlines([70],ymin=0,ymax=8000,color='r', label="Threshold @ 70",lw=3)
ax3.vlines([110],ymin=0,ymax=8000,color='lightgreen',label="Threshold @ 110",lw=3)
ax3.legend();
```



Baby Bear

We can thus follow a process for ending up with a happy medium of the two

Hysteresis Thresholding: Reducing Pixels

Now we apply the following steps.

1. Take the first threshold image with the highest (more strict) threshold
2. Remove the objects which are not cells (too small) using an opening operation.
3. Take a second threshold image with the higher value
4. Combine both threshold images
5. Keep the *between* pixels which are connected (by looking again at a neighborhood $\mathcal{N}$) to the *air* voxels and ignore the other ones. This goes back to our original supposition that the smaller structures are connected to the larger structures

```
%matplotlib inline
from skimage.morphology import dilation, opening, disk
from collections import OrderedDict
```

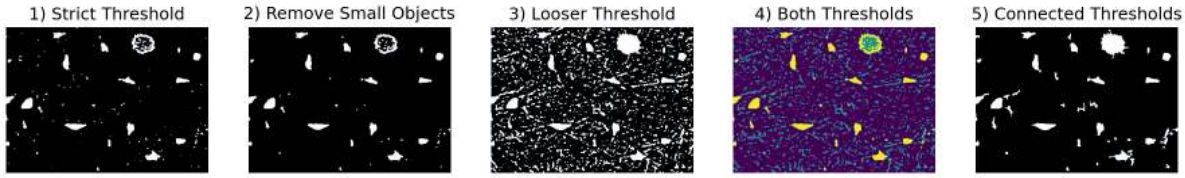
Thresholding with hysteresis

Step 1 Apply several thresholds to the image

```
thresh_vals = [0, 70, 110, 255]
step_list = OrderedDict()
step_list['Strict Threshold']     = bone_img<thresh_vals[1]
step_list['Remove Small Objects'] = opening(step_list['Strict Threshold'], disk(1))
step_list['Looser Threshold']     = bone_img<thresh_vals[2]
step_list['Both Thresholds']      = 1.0*step_list['Looser Threshold'] + 1.0*step_list[
 ↪'Remove Small Objects']
```

Step 2 Combine the different images $$I_{ConnectedThresholds} = \delta(I_{NoSmallObjects}) \cap I_{LooseThresh}$$

```
# the tricky part keeping the between images
step_list['Connected Thresholds'] = step_list['Remove Small Objects']

for i in range(10):
    step_list['Connected Thresholds'] = dilation(step_list['Connected Thresholds'] ,
                                                 disk(1.8)) & step_list['Looser␣
 ↪Threshold']

fig, ax_steps = plt.subplots(1, len(step_list), figsize = (15, 5), dpi = 150)

for i, (c_ax, (c_title, c_img)) in enumerate(zip(ax_steps.flatten(), step_list.
 ↪items()),1):
    c_ax.imshow(c_img, cmap = 'bone' if c_img.max()<=1 else 'viridis')
    c_ax.set_title('%d) %s' % (i, c_title)); c_ax.axis('off');
```



0.3.11 Multiple thresholds

It is not uncommon to have multiple classes in an image

Setting thresholds at 0.6 and 2.2 makes sense…

We see some misclassification that resembles a skin on the object!

Some further reading

Investigation the virtual skin effect

Let's look at three different edge profiles in the image

The virtual skin appears because the great transition from class 1 to 3 passes over several pixels and some of these pixels have values in the interval of class 2. These will be misclassified as class 2 and appear as the virtual skin in the segmented image.

How to avoid segmentation problems at the edges

Often the edge segmentation only invloves a few pixels that can be handled in different ways. E.g.

- Hysteresis
- Guarded edges and region growing

The guarded edge method uses an edge enhancing filter like a Laplacian filter to create an edge image. It is not nescessary to handle all edge but only those with high amplitudes in the Laplacian image. These pixels will be excluded from the thresholding inititally. Once the obvious pixels have been assigned we can handle the edge using region growing to fill up the missing edge information. we could also use a Laplacian of Gaussian filter to increase the robustness for low SNR.

0.4 More Complicated Images

As we briefly covered last time, many measurement techniques produce quite rich data.

- Digital cameras produce 3 channels of color for each pixel (rather than just one intensity)
- MRI produces dozens of pieces of information for every voxel which are used when examining different *contrasts* in the system.
- Raman-shift imaging produces an entire spectrum for each pixel
- Coherent diffraction techniques produce 2- (sometimes 3) diffraction patterns for each point. $\$I(x,y) = \hat{f}(x,y)\$$

0.4.1 Feature Vectors

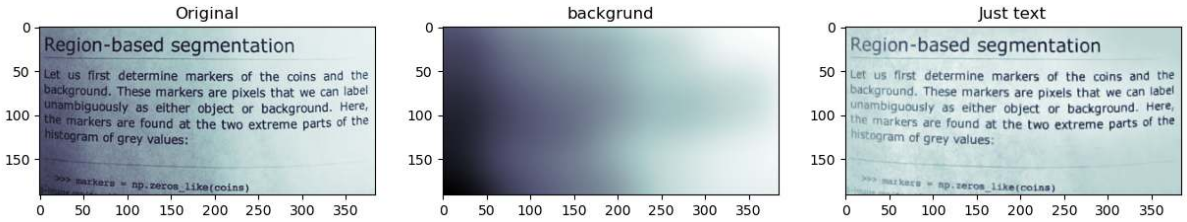
A pairing between spatial information (position) and some other kind of information (value). $\$\vec{x} \rightarrow \vec{f}\$$

We are used to seeing images in a grid format where the position indicates the row and column in the grid and the intensity (absorption, reflection, tip deflection, etc) is shown as a different color. We take an example here of text on a page.

This is an alternative to the top-hat method last week.

```
from skimage.data import page
import skimage.morphology as morph

page_image = page()
background = 255*gaussian(page_image, 20.0)
#background = morph.closing(page_image,morph.disk(51))
just_text = median(page_image, morph.disk(1))-background
```

```
fig,ax = plt.subplots(1,3,figsize=[15,5])
ax[0].imshow(page_image, cmap = 'bone'); ax[0].set_title('Original');
ax[1].imshow(background, cmap = 'bone'); ax[1].set_title('backgrund');
ax[2].imshow(just_text,  cmap = 'bone'); ax[2].set_title('Just text');
```



The gradient in the original is removed using a combination of filters to reconstruct the illumination

$$\text{JustText} = \underbrace{median_{disk}(img)}_{Noise} - \underbrace{G_{\sigma=20} * img}_{Illumination}$$

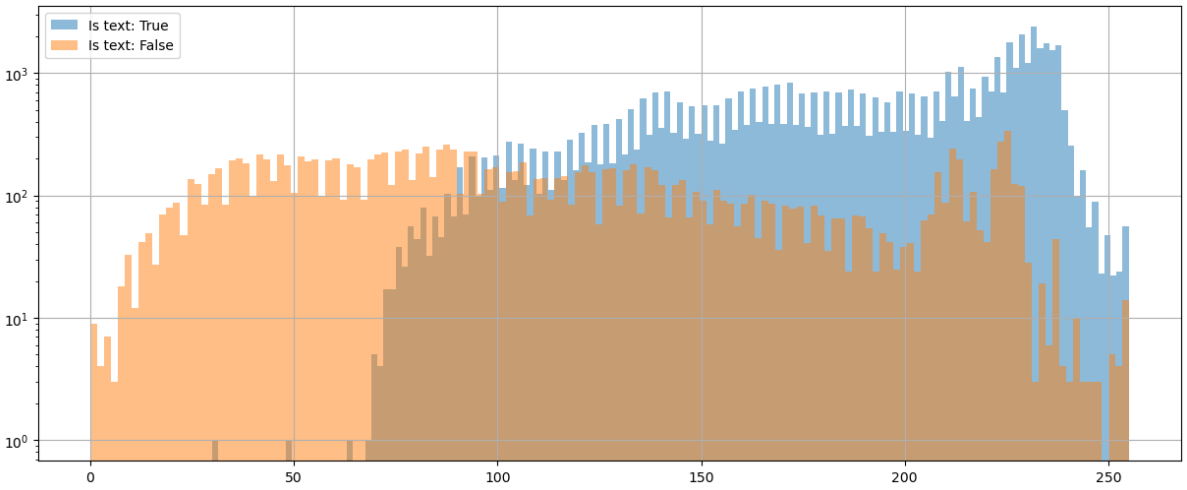
Let's create a feature table

```
xx, yy = np.meshgrid(np.arange(page_image.shape[1]),
           np.arange(page_image.shape[0]))
page_table = pd.DataFrame(dict(x = xx.ravel(),
                               y = yy.ravel(),
                               intensity = page_image.ravel(),
                              is_text = just_text.ravel()>0))
page_table.sample(10)
```

	x	y	intensity	is_text
38169	153	99	174	True
17741	77	46	161	True
38819	35	101	107	True
24130	322	62	234	True
15369	9	40	120	True
24015	207	62	120	False
6168	24	16	118	False
53301	309	138	227	True
8348	284	21	204	False
30094	142	78	181	True

Inspect the features - Intensity vs. IsText

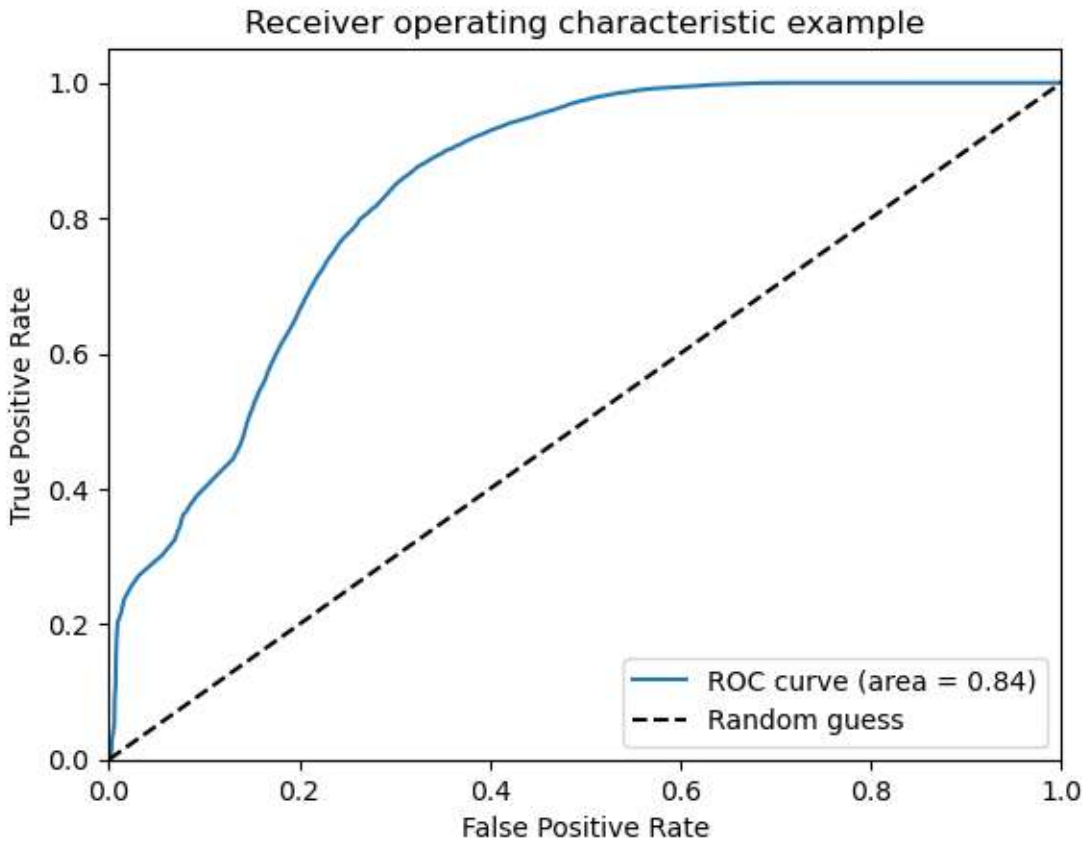
```
fig, ax1 = plt.subplots(1,1,figsize=(15,6))
for c_cat in [True,False] :
    page_table[page_table['is_text']==c_cat]['intensity'].hist(ax=ax1,
                                                               bins=150,
                                                               alpha=0.5,
                                                               label='Is text: {0}'.
 ↪format(c_cat))
ax1.set_yscale("log", nonpositive='clip')
ax1.legend();
```



What does the ROC curve look like?

```
from sklearn.metrics import roc_curve, roc_auc_score
fpr, tpr, _ = roc_curve    (page_table['is_text'], page_table['intensity'])
roc_auc     = roc_auc_score(page_table['is_text'], page_table['intensity'])
```

```
fig, ax = plt.subplots(1,1)
ax.plot(fpr, tpr, label='ROC curve (area = {0:0.2})'.format(roc_auc))
ax.plot([0, 1], [0, 1], 'k--',label='Random guess')
ax.set_xlim([0.0, 1.0])
ax.set_ylim([0.0, 1.05])
ax.set_xlabel('False Positive Rate');
ax.set_ylabel('True Positive Rate')
ax.set_title('Receiver operating characteristic example');
ax.legend(loc="lower right");
```



Adding Information

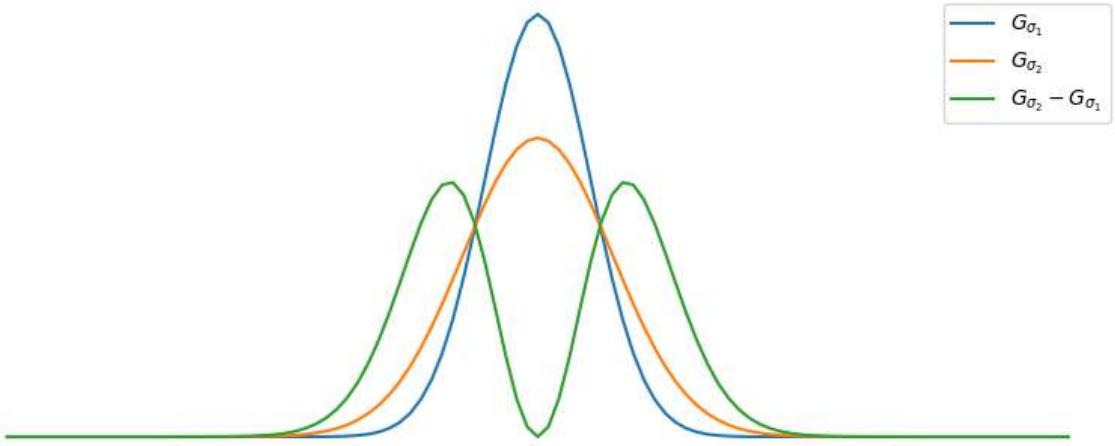
Here we can improve the results by adding information.

As we discussed in the third lecture (enhancement), edge-enhancing filters can be very useful for classifying images.

How about:

$$f_{DoG} = G_{\sigma_1} * f - G_{\sigma_2} * f = (G_{\sigma_1} - G_{\sigma_2}) * f, \quad \sigma_1 < \sigma_2$$

```
fig,ax=plt.subplots(1,figsize=(10,4))
x=np.linspace(-10,10,101)
g1=np.exp(-x**2/2)
g2=np.exp(-x**2/4)

ax.plot(x,g1/g1.sum(), label="$G_{\sigma_1}$")
ax.plot(x,g2/g2.sum(), label="$G_{\sigma_2}$")
ax.plot(x,(g2-g1)/(g2.sum()-g1.sum()),label="$G_{\sigma_2}-G_{\sigma_1}$")
ax.legend()
ax.axis("off");
```

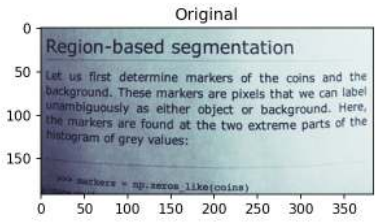


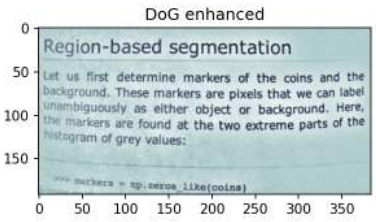
Testing with enhanced edges

```
def dog_filter(in_img, sig_1, sig_2):
    return gaussian(in_img, sig_1) - gaussian(in_img, sig_2)

page_edges = dog_filter(page_image, 0.5, 10)
page_table['edges'] = page_edges.ravel()
```

```
fig, ax = plt.subplots(1,3, figsize = (15, 4),dpi=150)
ax[1].imshow(page_image, cmap = 'bone'), ax[1].set_title('Original')
ax[2].imshow(page_edges, cmap = 'bone'), ax[2].set_title('DoG enhanced')

pd.plotting.table(data=page_table.sample(8).round(decimals=2), ax=ax[0], loc='center')
ax[0].axis('off');
```

	x	y	intensity	is_text	edges
31063	343	80	234	True	0.1
17677	13	46	115	True	0.01
11181	45	29	110	False	-0.04
55242	330	143	229	True	0.0
17891	227	46	218	True	0.05
1345	193	3	202	True	0.02
34692	132	90	157	True	0.05
23682	258	61	241	True	0.07

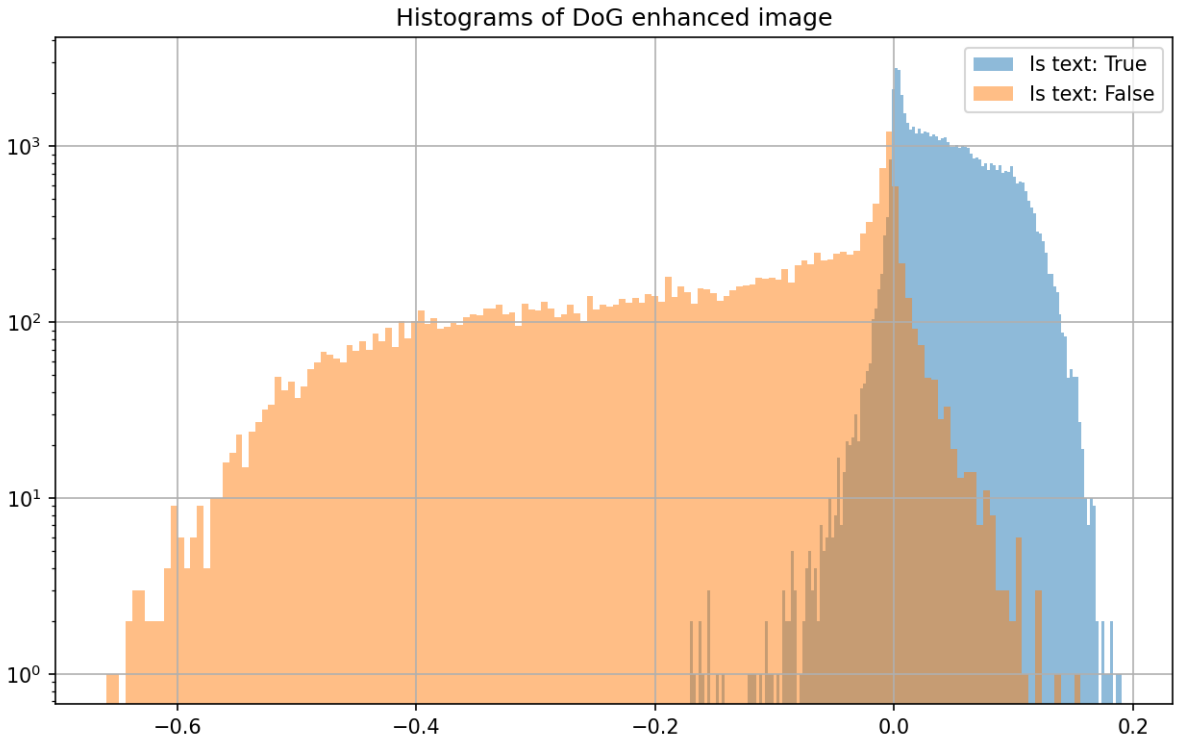




Histogram of enhanced image

Let's look at the gray level distribution after applying the enhancement filter.

```
fig, ax = plt.subplots(1, figsize=(10,6),dpi=150)

for c_cat in [True,False] :
    page_table[page_table['is_text']==c_cat]['edges'].hist(ax=ax,
                                                           bins=150,
                                                           alpha=0.5,
                                                           label='Is text: {0}'.
 ↪format(c_cat))
ax.set_yscale("log", nonpositive='clip')
ax.legend();
ax.set_title('Histograms of DoG enhanced image'); ax.set_yscale("log", nonpositive=
 ↪'clip'); ax.legend();
```



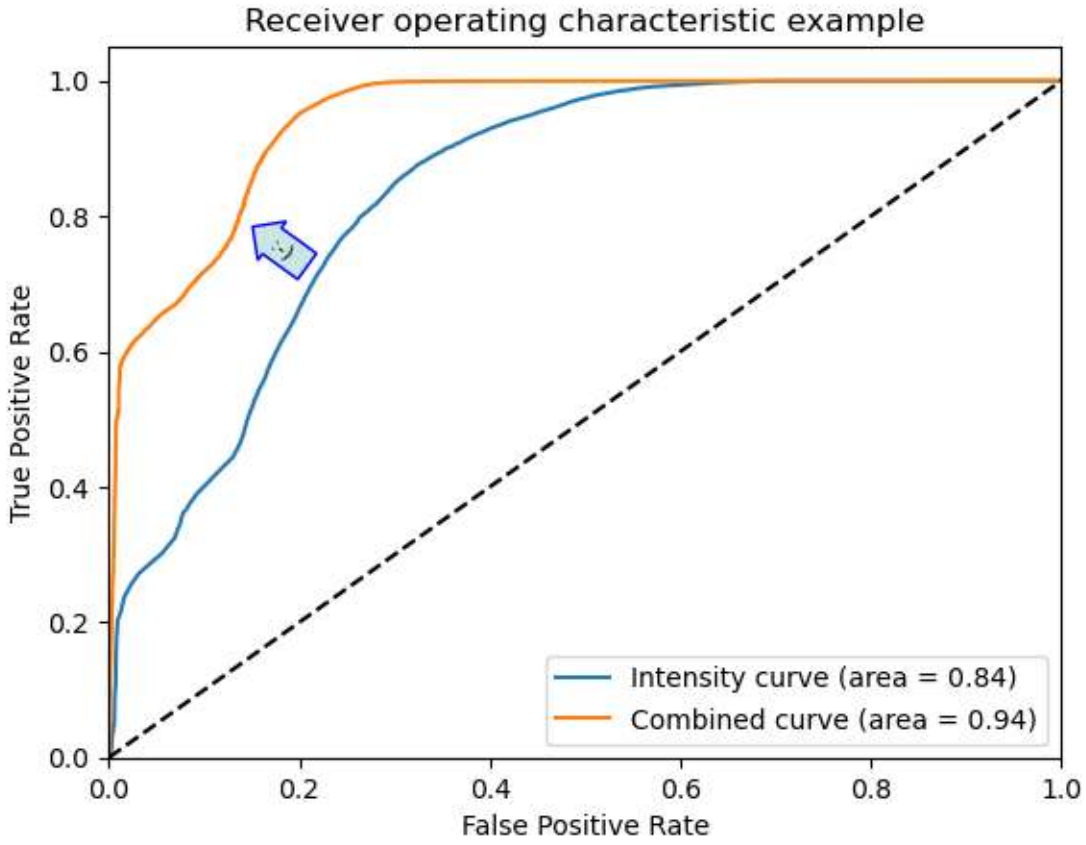
We see here that the text pixels have been compressed into a narrow interval with some tail toward the lower intensities.

Checking the ROC performance

Now, we can compute the ROC curve to see if the enhanced image performs better than the original.

```
from sklearn.metrics import roc_curve, roc_auc_score
fpr2, tpr2, _ = roc_curve(page_table['is_text'],
                          page_table['intensity']/1000.0+page_table['edges'])
roc_auc2      = roc_auc_score(page_table['is_text'],
                         page_table['intensity']/1000.0+page_table['edges'])
```

```
# Visualization
fig, ax = plt.subplots(1,1)
ax.plot(fpr, tpr, label='Intensity curve (area = %0.2f)' % roc_auc)
ax.plot(fpr2, tpr2, label='Combined curve (area = %0.2f)' % roc_auc2)
ax.plot([0, 1], [0, 1], 'k--')
ax.set_xlim([0.0, 1.0])
ax.set_ylim([0.0, 1.05])
ax.set_xlabel('False Positive Rate')
ax.set_ylabel('True Positive Rate')
ax.set_title('Receiver operating characteristic example')
ax.legend(loc="lower right");

bbox_props = dict(boxstyle="larrow", fc=(0.8, 0.9, 0.9), ec="b", lw=1)
t = ax.text(0.16, 0.75, " :-)  ", ha="left", va="center", rotation=-36,
            size=8,
            bbox=bbox_props)
```

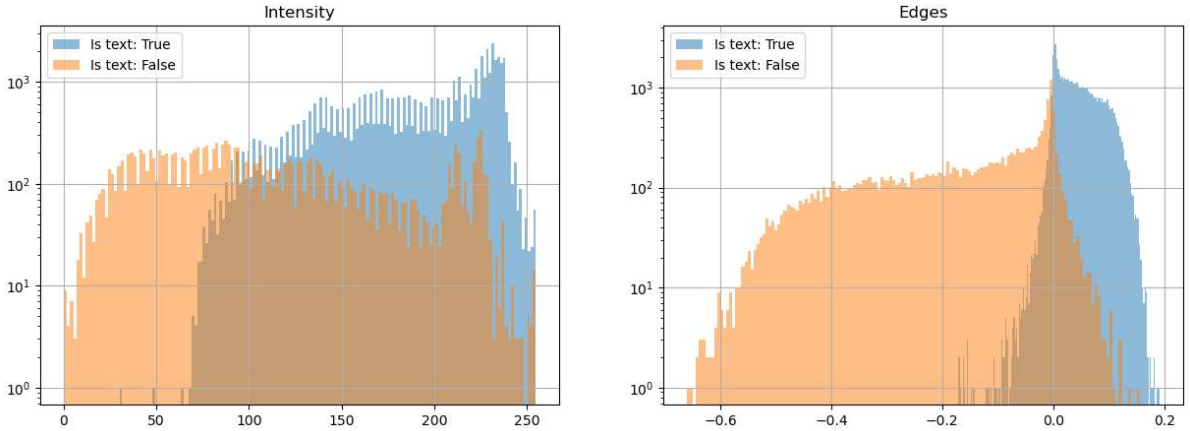


We can see that it is doing better already bey looking at the curves. Also the AUC confirms this which is great. We have made and improvement thanks to the preprocessing of the data.

Why does the second filter perform better?

We can see the reason for the performance improvement when we compare the histograms.

```
fig, (ax1, ax2) = plt.subplots(1,2, figsize = (15, 5))

for c_cat in [True,False] :
    page_table[page_table['is_text']==c_cat]['intensity'].hist(ax=ax1,
                                                               bins=150,
                                                               alpha=0.5,
                                                               label='Is text: {0}'.
 ↪format(c_cat))
ax1.set_yscale("log", nonpositive='clip')
ax1.legend();
ax1.set_title('Intensity')

for c_cat in [True,False] :
    page_table[page_table['is_text']==c_cat]['edges'].hist(ax=ax2,
                                                           bins=150,
                                                           alpha=0.5,
                                                           label='Is text: {0}'.
 ↪format(c_cat))
ax2.set_yscale("log", nonpositive='clip')
ax2.legend();
ax2.set_title('Edges');
```



The two classes are more or less overlapping each other in the original image. After filtering the gray levels are more compact for each class with some slight overlap in the tail distributions.

0.5 Clustering / Classification (Unsupervised)

Unsupervised segmentation method tries to make sense of the that without any prior knowledge. They may need an initialization parameter telling how many classes are expected in the data, but beyond that you don't have to provide much more information.

- Automatic clustering of multidimensional data into groups based on a distance metric
- Fast and scalable to petabytes of data (Google, Facebook, Twitter, etc. use it regularly to classify customers, advertisements, queries)
- **Input** = feature vectors, distance metric, number of groups

- **Output** = a classification for each feature vector to a group

0.5.1 Example data

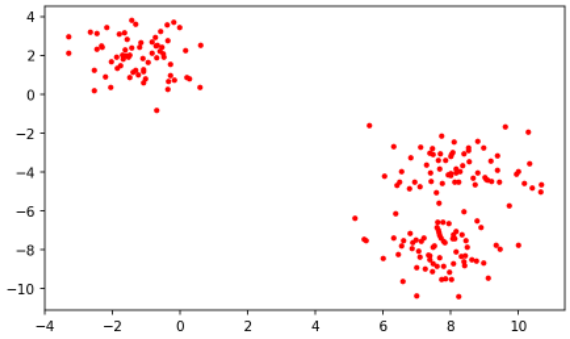
With clustering methods you aim to group data points together into a limited number of clusters. Here, we start to look at an example where each data point has two values.

The test data is generated using the `make_blobs` function.

```
test_pts = pd.DataFrame(make_blobs(n_samples=200, random_state=2018)[
                        0], columns=['x', 'y'])
```

```
fig, ax = plt.subplots(1,2, figsize = (15, 4),dpi=150)

ax[1].plot(test_pts.x, test_pts.y, 'r.')
pd.plotting.table(data=test_pts.sample(8).round(decimals=2), ax=ax[0], loc='center')
ax[0].axis('off');
```

	x	y
190	7.5	-7.8
96	8.43	-7.5
135	8.88	-6.84
73	5.16	-6.36
81	10.32	-3.54
22	8.75	-8.56
12	-2.22	0.91
119	-1.87	1.36



First clustering attempt

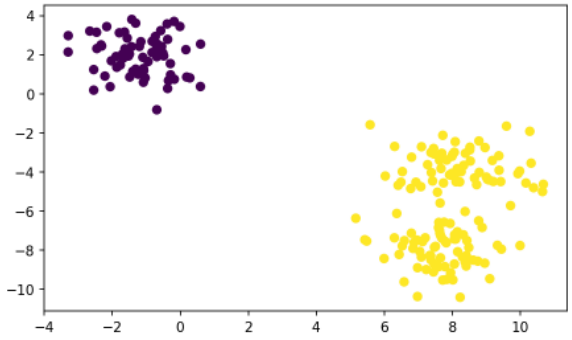
The generated data set has two obvoius clusters, but if look closer it is even possible to identify three clusters. We will now use this data to try the k-means algorithm.

```
km = KMeans(n_clusters=2, random_state=2018,n_init='auto')
n_grp = km.fit_predict(test_pts)
grp_pts = test_pts.copy()
grp_pts['group'] = n_grp
grp_pts.groupby(n_grp, group_keys=False).apply(lambda x: x.sample(5))
```

```
             x         y  group
174  -1.104400  1.871711      0
8    -0.737558  2.935987      0
101  -0.693470  1.900714      0
122  -0.578495  3.237306      0
27   -0.678995  2.546154      0
81   10.318225 -3.544903      1
57    8.529693 -3.438141      1
74    6.436361 -8.220122      1
193  10.677735 -4.628090      1
96    8.428999 -7.504711      1
```

```
# Plotting
fig, ax = plt.subplots(1,2, figsize = (15, 4),dpi=150)
ax[1].scatter(test_pts.x, test_pts.y, c=n_grp)
pd.plotting.table(data=grp_pts.groupby(n_grp, group_keys=False).apply(lambda x: x.
 ↪sample(5)), ax=ax[0], loc='center')

ax[0].axis('off');
```

	x	y	group
198	-0.012441330856636457	3.4512483449860842	0.0
27	-0.6789953617196197	2.5461542948097695	0.0
179	-0.17738613185767016	0.7492287541276457	0.0
171	-0.950993928259934	1.6560267667647177	0.0
6	-1.6821662750536035	1.8316796226438223	0.0
51	7.203155379695138	-7.3664998931149475	1.0
73	5.160910370056838	-6.364901813702201	1.0
195	7.720958193819376	-2.1207408297319548	1.0
61	7.834442888478978	-3.3832605092543555	1.0
175	9.37602062825143	-3.9077143464872677	1.0



0.5.2 K-Means Algorithm

We give as an initial parameter

- the number of groups we want to find
- and possibly a criteria for removing groups that are too similar

It is an iterative method that starts with a label image where each pixel has a random label assignment. Each iteration involves the following steps:

1. Compute current centroids based on the o current labels
2. Compute the value distance for each pixel to the each centroid. Select the class which is closest.
3. Reassign the class value in the label image.
4. Repeat until no pixels are updated.

The distance from pixel *i* to centroid *j* is usually computed as $||p_i - c_j||_2$.

Increasing the number of clusters

In this example we will use the blob data we previously generated and to see how k-means behave when we select different numbers of clusters.

Let's try k-means with N=2,3,4 clusters:

```
clusters=[2,3,4,5]
fig, axes = plt.subplots(1,len(clusters),figsize=(15,4))

for ax,N in zip(axes,clusters) :
    km    = KMeans(n_clusters=N, random_state=2018,n_init='auto');
    n_grp = km.fit_predict(test_pts)

    ax.scatter(test_pts.x, test_pts.y, c=n_grp)
    ax.set_title('{0} groups'.format(N))
```

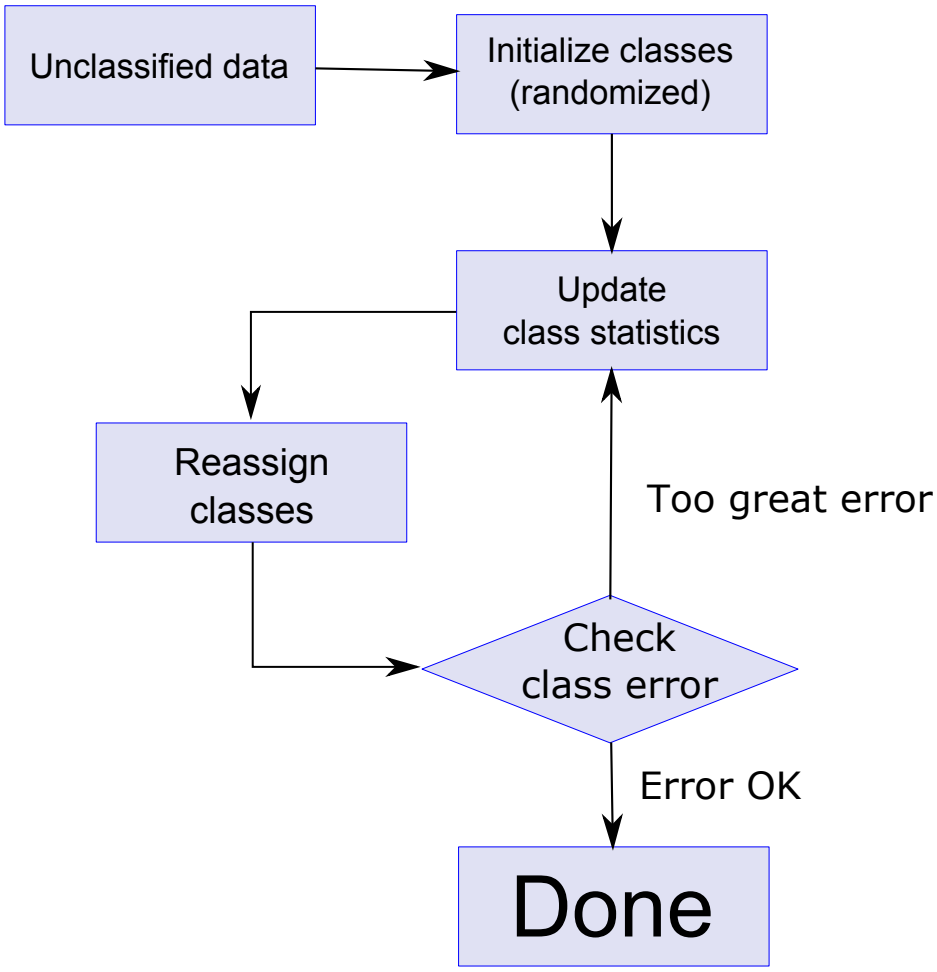
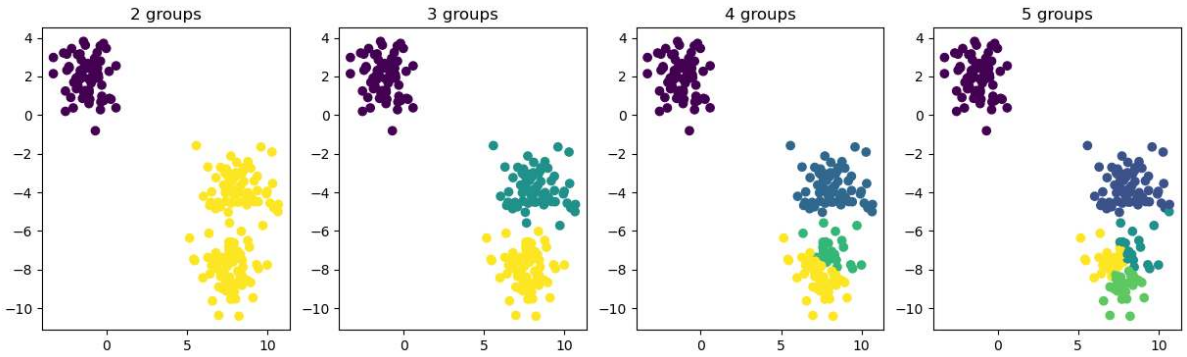


Fig. 1: Flow chart for the k-means clustering iterations.



Note: If you look for N groups you will always find N groups with K-Means, whether or not they make any sense

When we select two clusters there is a natural separation between the two clusters we easily spotted by just looking at the data. When the number of clusters is increased to three, we again see a cluster separation that makes sense. Now, when the number of clusters is increased yet another time we see that one of the clusters is split once more. This time it is how ever questionable if the number of clusters makes sense. From this example, we see that it is important to be aware of problems related to over segmentation.

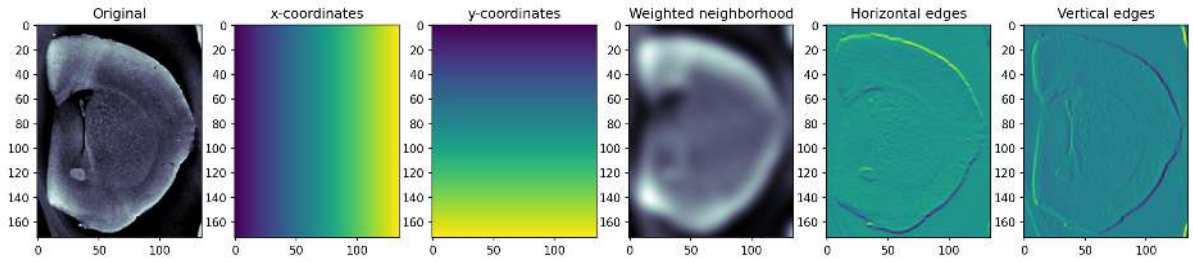
0.5.3 What vector space do we have?

- Sometimes represent physical locations (classify swiss people into cities)
- Can include intensity or color (K-means can be used as a thresholding technique when you give it image intensity as the vector and tell it to find two or more groups)
- Can also include orientation, shape, or in extreme cases full spectra (chemically sensitive imaging)

0.5.4 Add spatial information to k-means

It is important to note that k-means by definition is not position sensitive. Clustering is by definition not position sensitive, mainly measures distances between values and distributions. The position can however be included as additional components in the data vectors. You can also add neighborhood information using filtered images as additionals components of the feature vectors.

```
cortex_img = imread("figures/cortex.png")[::3, ::3]/1000.0
fig,ax = plt.subplots(1,6,figsize=(18,5),dpi=150);
xx, yy = np.meshgrid(np.arange(cortex_img.shape[1]),
                     np.arange(cortex_img.shape[0]))
ax[0].imshow(cortex_img,cmap='bone'), ax[0].set_title('Original')
ax[1].imshow(xx), ax[1].set_title('x-coordinates')
ax[2].imshow(yy), ax[2].set_title('y-coordinates')
ax[3].imshow(flt.gaussian(cortex_img, sigma=5),cmap='bone'), ax[3].set_title(
 ↪'Weighted neighborhood')
ax[4].imshow(flt.sobel_h(cortex_img)),ax[4].set_title('Horizontal edges')
ax[5].imshow(flt.sobel_v(cortex_img)),ax[5].set_title('Vertical edges');
```



K-Means Applied to Cortex Image

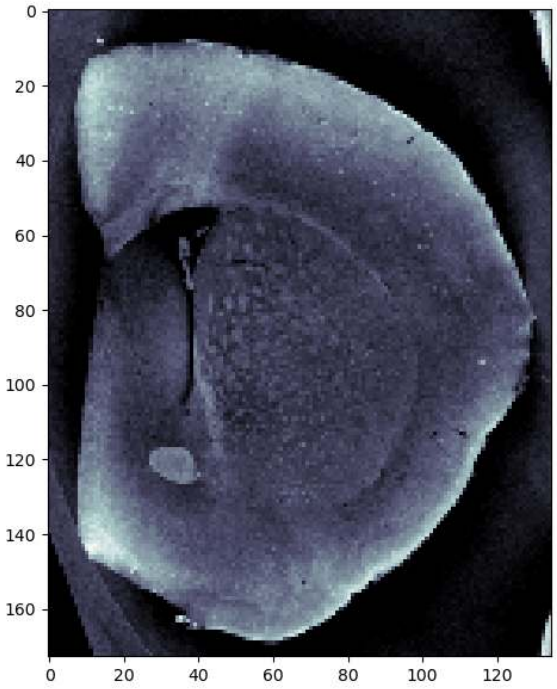
In this example we use position and intensity as feature vectors.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imread
%matplotlib inline
```

```
cortex_img = imread("figures/cortex.png")[::3, ::3]/1000.0
np.random.seed(2018)
xx, yy = np.meshgrid(np.arange(cortex_img.shape[1]),np.arange(cortex_img.shape[0]))

cortex_df = pd.DataFrame(dict(x=xx.ravel(),
                              y=yy.ravel(),
                              intensity=cortex_img.ravel()))
```

```
fig, (ax0,ax1) = plt.subplots(1, 2,figsize=(12, 8))
ax1.imshow(cortex_img, cmap='bone')

ccolors = plt.cm.BuPu(np.full(3, 0.1))

pd.plotting.table(data=cortex_df.sample(10), ax=ax0, loc='center',colColours=ccolors,
 ↪fontsize=20)
ax0.axis('off');
```

	x	y	intensity
18621	126.0	137.0	6.015
8830	55.0	65.0	18.961
23217	132.0	171.0	57.508
7605	45.0	56.0	5.81
13164	69.0	97.0	18.338
13064	104.0	96.0	16.929
8397	27.0	62.0	0.388
3556	46.0	26.0	35.116
7075	55.0	52.0	21.122
8819	44.0	65.0	15.514



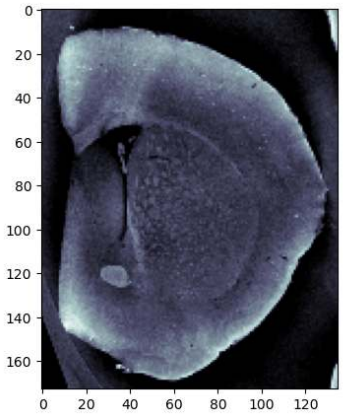
First segmentation attempt with k-means on brain image

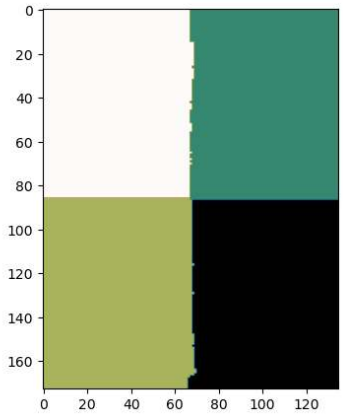
We use $N_{clusters} = 4$

```
from sklearn.cluster import KMeans
km = KMeans(n_clusters=4, random_state=2018,n_init='auto')
cortex_df['group'] = km.fit_predict(cortex_df[['x', 'y', 'intensity']].values)
cortex_seg         = cortex_df['group'].values.reshape(cortex_img.shape)
```

```
fig, (ax0,ax1, ax2) = plt.subplots(1, 3, figsize=(15, 5))
ax1.imshow(cortex_img, cmap='bone')
ax2.imshow(cortex_seg, cmap='gist_earth')

ccolors = plt.cm.BuPu(np.full(4, 0.1))

pd.plotting.table(data=cortex_df.groupby("group", group_keys=False).sample(n=3,␣
↪random_state=42), ax=ax0, loc='center',colColours=ccolors,fontsize=20)


ax0.axis('off');
```

	x	y	intensity	group
16733	128.0	123.0	8.593	0.0
22380	105.0	165.0	2.206	0.0
18621	126.0	137.0	6.015	0.0
7528	103.0	55.0	18.696	1.0
9439	124.0	69.0	26.291	1.0
4022	107.0	29.0	0.0	1.0
18157	67.0	134.0	18.878	2.0
19748	38.0	146.0	23.832	2.0
15562	37.0	115.0	16.825	2.0
5600	65.0	41.0	18.07	3.0
1795	40.0	13.0	36.024	3.0
8435	65.0	62.0	4.505	3.0





Why is the image segmented like this?

Rescaling components

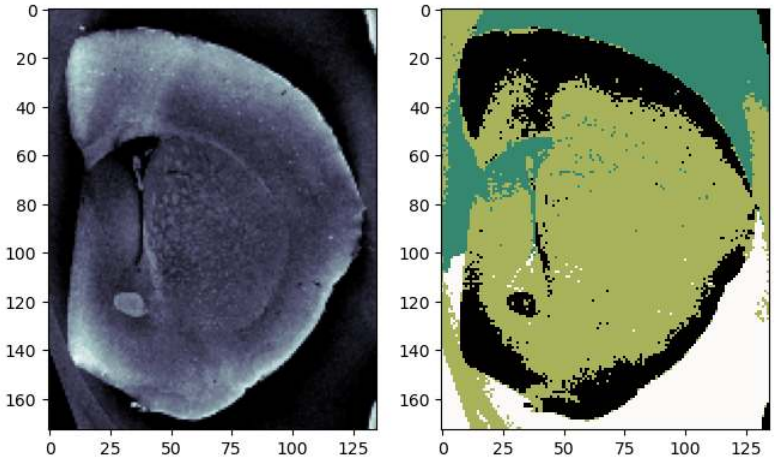
Since the distance is currently calculated by $||\vec{v}_i - \vec{v}_j||$ and the values for the position is much larger than the values for the *Intensity* , *Sobel* or *Gaussian* they need to be rescaled so they all fit on the same axis $\vec{v} = \{\frac{x}{10}, \frac{y}{10}, \text{Intensity}\}$

$N_{clusters}$=4

```
km = KMeans(n_clusters=4, random_state=2018,n_init='auto')
scale_cortex_df   = cortex_df.copy()
scale_cortex_df.x = scale_cortex_df.x/10
scale_cortex_df.y = scale_cortex_df.y/10
scale_cortex_df['group'] = km.fit_predict(scale_cortex_df[['x', 'y', 'intensity']].
 ↪values)
cortex_seg=scale_cortex_df['group'].values.reshape(cortex_img.shape)
```

```
fig, (ax0,ax1, ax2) = plt.subplots(1, 3,
                                   figsize=(12, 5))
ax1.imshow(cortex_img, cmap='bone')
ax2.imshow(cortex_seg,
           cmap='gist_earth',interpolation='none')

ccolors = plt.cm.BuPu(np.full(4, 0.1))
pd.plotting.table(data=scale_cortex_df.groupby(['group'], group_keys=False).
 ↪sample(n=4, random_state=42),
                  ax=ax0, loc='center',colColours=ccolors,fontsize=20)

ax0.axis('off');
```

	x	y	intensity	group
4345	2.5	3.2	27.912	0.0
6898	1.3	5.1	33.534	0.0
5218	8.8	3.8	30.619	0.0
6039	9.9	4.4	27.604	0.0
6082	0.7	4.5	1.244	1.0
4858	13.3	3.5	6.896	1.0
467	6.2	0.3	0.73	1.0
262	12.7	0.1	0.063	1.0
15687	2.7	11.6	12.343	2.0
10887	8.7	8.0	16.335	2.0
22703	2.3	16.8	15.821	2.0
16423	8.8	12.1	24.902	2.0
14979	12.9	11.0	2.364	3.0
21830	9.5	16.1	0.0	3.0
22000	13.0	16.2	0.0	3.0
22118	11.3	16.3	0.828	3.0



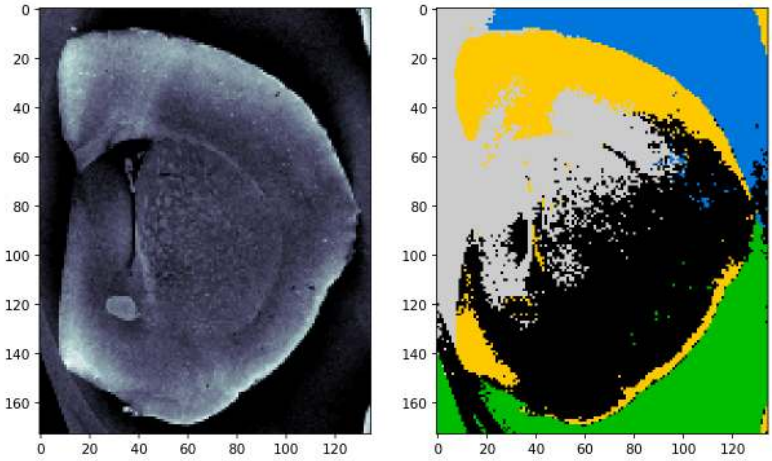
Let's try a different position scaling

$$\vec{v} = \left\{\frac{x}{5}, \frac{y}{5}, \text{Intensity}\right\}$$

$N_{clusters}$=5

```
km = KMeans(n_clusters=5, random_state=2019)
scale_cortex_df = cortex_df.copy()
scale_cortex_df.x = scale_cortex_df.x/5
scale_cortex_df.y = scale_cortex_df.y/5
scale_cortex_df['group'] = km.fit_predict(
    scale_cortex_df[['x', 'y', 'intensity']].values)

fig, (ax0,ax1, ax2) = plt.subplots(1, 3,
                                   figsize=(15, 8), dpi=150)
ax1.imshow(cortex_img, cmap='bone')
ax2.imshow(scale_cortex_df['group'].values.reshape(cortex_img.shape),
           cmap='nipy_spectral')
scale_cortex_df.groupby("group", group_keys=False).sample(n=3)

ccolors = plt.cm.BuPu(np.full(4, 0.1))

pd.plotting.table(
    data=scale_cortex_df.groupby("group", group_keys=False).sample(n=3),
    ax=ax0,
    loc="center",
    colColours=ccolors,
    fontsize=20
)


ax0.axis('off');
```

	x	y	intensity	group
15344	17.8	22.6	20.844	0.0
18124	6.8	26.8	18.779	0.0
9027	23.4	13.2	21.835	0.0
393	24.6	0.4	0.703	1.0
6464	23.8	9.4	0.113	1.0
1480	26.0	2.0	0.0	1.0
21826	18.2	32.2	0.0	2.0
22497	17.4	33.2	0.344	2.0
21594	25.8	31.8	0.0	2.0
4523	13.6	6.6	27.414	3.0
5487	17.4	8.0	27.415	3.0
19668	18.6	29.0	48.258	3.0
9215	7.0	13.6	0.08	4.0
14985	0.0	22.2	9.286	4.0
6701	17.2	9.8	17.978	4.0



0.5.5 When can clustering be used on images?

Clustering and in particular k-means are often used for imaging applications.

- Single images (Cortex example)

It does not make much sense to segment using only the image intensity with k-means. This would result in thresholding similar to the one provided by Otsu's method. If you, however, add spatial information like edges and positions it starts be interesting to use k-means for a single image.

- Bimodal data

In recent years, several neutron imaging instruments have have installed an X-ray source to provide complementary information to the neutron images. This is a great mix of information for k-means based segmenation. Another type of bimodal data is when a grating interferometry setup is used. This setup provides three images revealing different aspects of the samples. Again, here it a great combination as these data are

- Hyper-spectral data (materials science example)

Many materials have a characteristic response the neutron wavelength. This is used in many materials science experiments, in particular experiments performed at pulsed neutron sources. The data from such experiments result in a spectrum response for each pixel. This means each pixel can be a vector of >1000 elements.

0.6 Quad trees

Split the image in subregions until a criterion is fulfilled:

0.6.1 Principle

A quad tree is created by dividing image into four sections. Next you iterate the following steps until no more splits are made:

1. Compute metric (e.g. max-min or standard deviation) for each new sub-region
2. If the metric is greater than a threshold slit the region into four new regions
3. Otherwise leave it as is, the region is "constant" according to the criterion.

The figure below illustrates how an image is decomposed into a quad tree. Here you can see that the regions near edges are much smaller than in constant valued regions.

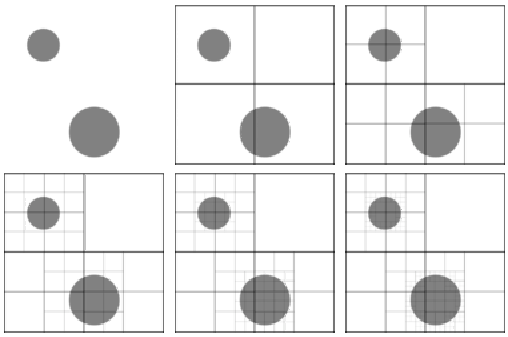


Fig. 1: A quad tree decomposition.

0.6.2 Quad tree example

Next we try to decompose an natural image as a quad tree.

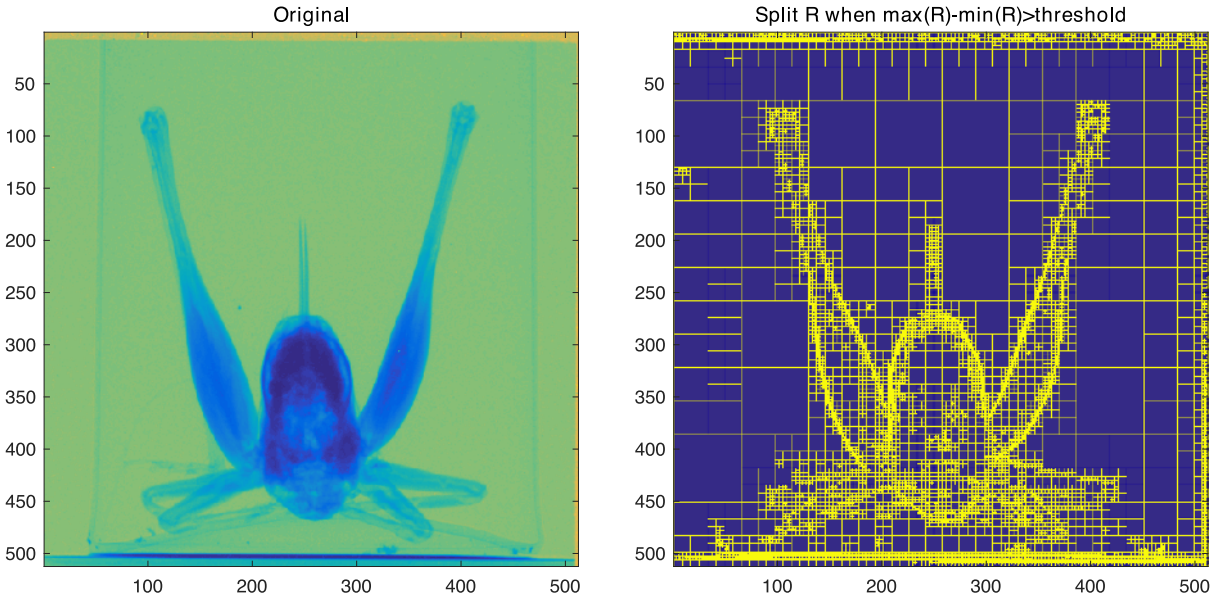


Fig. 2: A neutron radiograph of a grasshopper decomposed using a quad tree.

0.7 Superpixels

An approach for simplifying images by performing a clustering and forming super-pixels from groups of similar pixels. https://ivrl.epfl.ch/research/superpixels

DOI

0.7.1 Why use superpixels

Super pixels

- Drastically reduced data size,
- Serves as an initial segmentation showing spatially meaningful groups

---

A super-pixel example

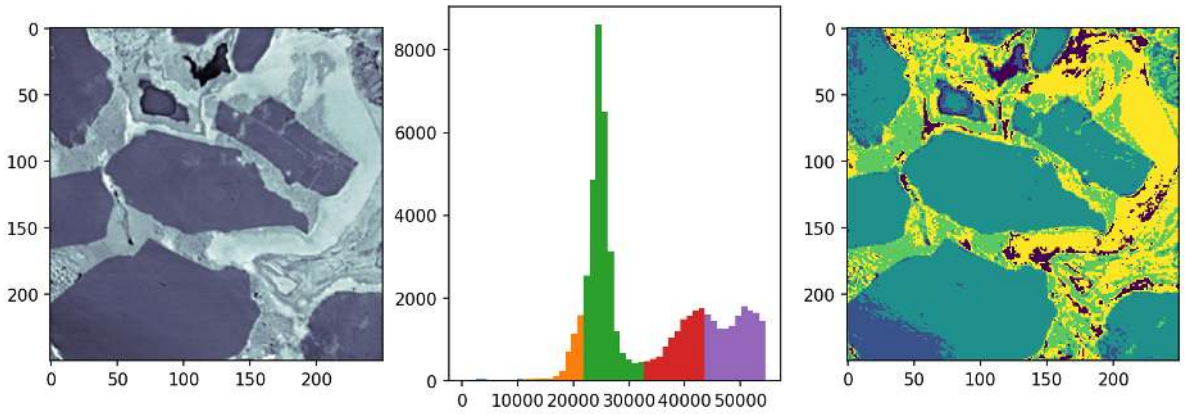
We start with an example of shale with multiple phases

- rock
- clay
- pore

Basic thresholds

We start the analysis with using plain threshold based on the histogram. You can clearly see in the histogram that the chosen thresholds will produce many misclassified pixels. This approach will give a hint on the regions but is not very precise.

```
shale_img = imread("figures/shale-slice.tiff")
np.random.seed(2018)
fig, (ax1, ax2, ax3) = plt.subplots(1, 3,
                                    figsize=(12, 4), dpi=150)
ax1.imshow(shale_img, cmap='bone')
thresh_vals = np.linspace(shale_img.min(), shale_img.max(), 5+2)[:-1]
out_img = np.zeros_like(shale_img)
for i, (t_start, t_end) in enumerate(zip(thresh_vals, thresh_vals[1:])):
    thresh_reg = (shale_img > t_start) & (shale_img < t_end)
    ax2.hist(shale_img.ravel()[thresh_reg.ravel()])
    out_img[thresh_reg] = i
# ax3.imshow(out_img, cmap='gist_earth',interpolation='none');
ax3.imshow(out_img, cmap='viridis',interpolation='none');
```



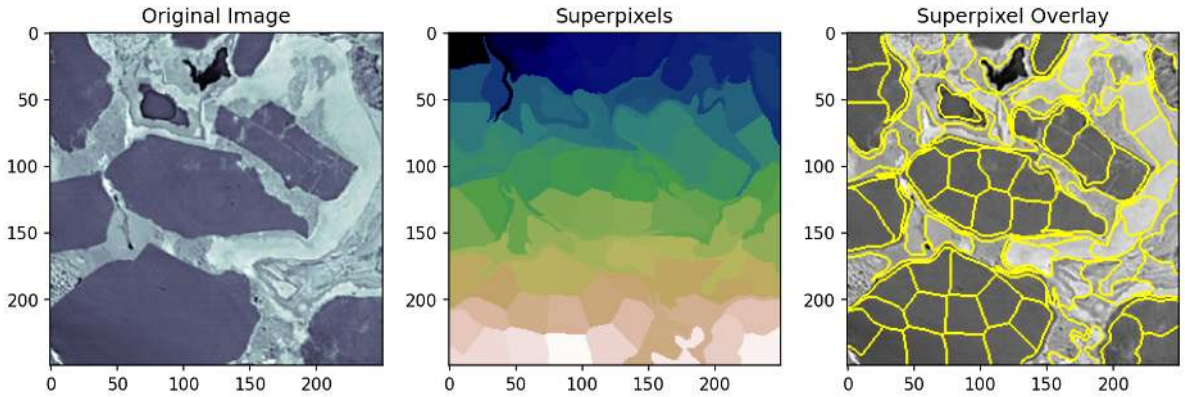
0.7.2 Super pixels

Super-pixels in as sense an evolution of the quad tree. They are not bound to the strict splitting scheme but can generate regions with limited variations of arbitrary shape.

Using the SLIC algorithm

```
from skimage.segmentation import slic, mark_boundaries

shale_segs = slic(shale_img,
                  n_segments=100,
                  compactness=5e-2,
                  sigma=3.0,
                  start_label=1,
                 channel_axis=None)
```

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 4), dpi=150)
ax1.imshow(shale_img, cmap='bone')
ax1.set_title('Original Image')
ax2.imshow(shale_segs, cmap='gist_earth')
ax2.set_title('Superpixels')
ax3.imshow(mark_boundaries(shale_img, shale_segs))
ax3.set_title('Superpixel Overlay');
```

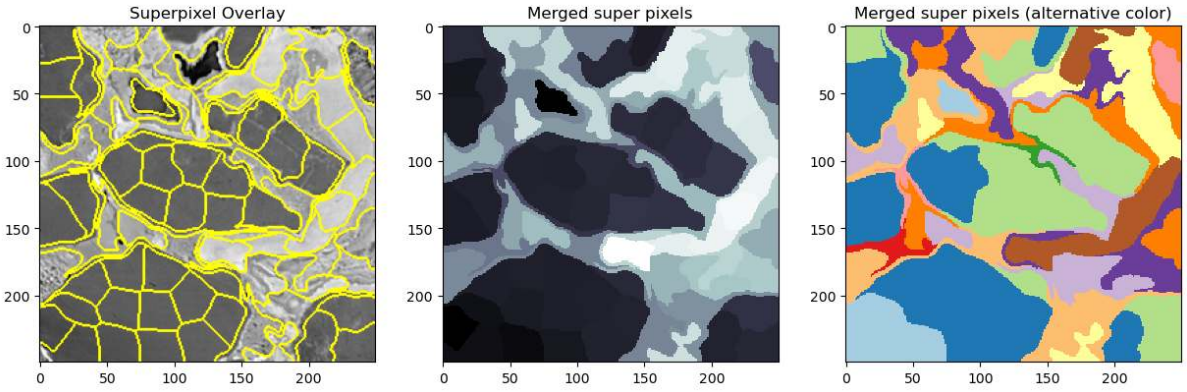


0.7.3 Merging super pixels

Usually, to many super pixels are identified. They are also not really representing the feature shapes in the image. The super-pixels currently also only have pixel indexes as values. In the next step we assign the average values of each super-pixel. This produces a patchy image that ideally is easier to interpret.

```
flat_shale_img = shale_img.copy()
for s_idx in np.unique(shale_segs.ravel()):
    flat_shale_img[shale_segs == s_idx] = np.mean(
        flat_shale_img[shale_segs == s_idx])
```

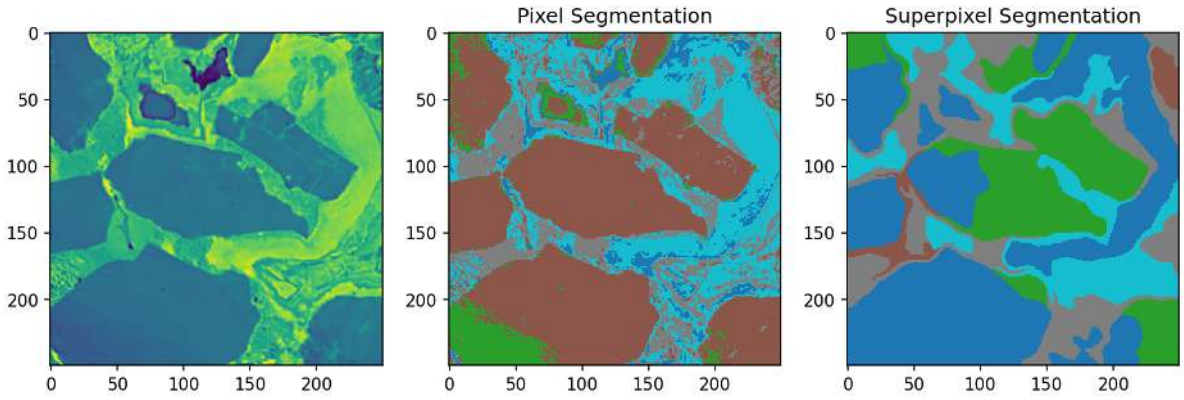
```
fig, (ax1,ax2,ax3) = plt.subplots(1, 3, figsize=(15, 5))
ax1.imshow(mark_boundaries(shale_img, shale_segs))
ax1.set_title('Superpixel Overlay')
ax2.imshow(flat_shale_img, cmap='bone',interpolation='none');
ax2.set_title('Merged super pixels')
ax3.imshow(flat_shale_img, cmap='Paired',interpolation='none');
ax3.set_title('Merged super pixels (alternative color)');
```



0.7.4 Segmentation using super pixels

```
thresh_vals = np.linspace(flat_shale_img.min(), flat_shale_img.max(), 5+2)[:-1]
sp_out_img = np.zeros_like(flat_shale_img)
for i, (t_start, t_end) in enumerate(zip(thresh_vals, thresh_vals[1:])):
    thresh_reg = (flat_shale_img > t_start) & (flat_shale_img < t_end)
    sp_out_img[thresh_reg] = i
```

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3,figsize=(12, 4), dpi=150)
ax1.imshow(shale_img, cmap='viridis')
ax2.imshow(out_img, cmap='tab10',interpolation='none')
ax2 .set_title('Pixel Segmentation')
ax3.imshow(sp_out_img, cmap='tab10',interpolation='none')
ax3.set_title('Superpixel Segmentation');
```



0.8 Probabilistic Models of Segmentation

A more general approach is to use a probabilistic model to segmentation. We start with our image $I(\vec{x})\ \forall \vec{x} \in \mathbb{R}^N$ and we classify it into two phases α and β

$$P(\{\vec{x}, I(\vec{x})\}|\alpha) \propto P(\alpha) + P(I(\vec{x})|\alpha) + P\left(\sum_{x' \in \mathcal{N}} I(\vec{x'})|\alpha\right)$$

- $P(\{\vec{x}, f(\vec{x})\}|\alpha)$ the probability a given pixel is in phase α given we know it's position and value (what we are trying to estimate)
- $P(\alpha)$ probability of any pixel in an image being part of the phase (expected volume fraction of that phase)
- $P(I(\vec{x})|\alpha)$ probability adjustment based on knowing the value of I at the given point (standard threshold)
- $P(f(\vec{x'})|\alpha)$ are the collective probability adjustments based on knowing the value of a pixels neighbors (very simple version of Markov Random Field approaches)

0.9 Summary

0.9.1 Histogram based thresholding

- Otsu and others
- Hysteresis thresholding

0.9.2 Clustering - K-means

- Add position information

0.9.3 Similar region segmentations

- Quad trees
- Super pixels

0.10 Supervised Segmentation Approaches

0.10.1 Overview

1. Methods
2. Pipelines
3. Classification
4. Regression
5. Segmentation

0.10.2 Reading Material

- Introduction to Machine Learning: ETH Course
- Decision Forests for Computer Vision and Medical Image Analysis
- U-Net: Convolutional Networks for Biomedical Image Segmentation
- U-Net Website

Load some modules for the notebook

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from skimage.io         import imread
from sklearn.datasets   import make_blobs
from sklearn.neighbors  import KNeighborsClassifier
from sklearn.ensemble   import RandomForestClassifier
from sklearn.tree       import export_graphviz
import graphviz
from sklearn.tree       import DecisionTreeClassifier
from sklearn.tree       import DecisionTreeRegressor
from sklearn.ensemble   import RandomForestRegressor
#from pipe_utils         import px_flatten_step, show_pipe, fit_img_pipe
from sklearn.pipeline   import Pipeline
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import FunctionTransformer
from sklearn.cluster    import KMeans
import plotsupport as ps
%matplotlib inline
```

0.11 Basic Methods Overview

Supervised segmentation is a two-step approach

0.11.1 Training

- The training phase is when the parameters of the model are *learned*
- Used training data with ground truth.

0.11.2 Prediction

- Provides responses on inputs using the trained model.
- Uses new unseen data.

There are a number of supervised methods we can use for

- classification,
- regression
- and both.

The training phase is when the parameters of the model are *learned* and involve putting inputs into the model and updating the parameters so they better match the outputs. This is a sort-of curve fitting (with linear regression it is exactly curve fitting).

The predicting phase is once the parameters have been set applying the model to new datasets. At this point the parameters are no longer adjusted or updated and the model is frozen. Generally it is not possible to tweak a model any more using new data but some approaches (most notably neural networks) are able to handle this.

There are a number of methods we can use for classification, regression and both. For the simplification of the material we will not make a massive distinction between classification and regression but there are many situations where this is not appropriate. Here we cover a few basic methods, since these are important to understand as a starting point for solving difficult problems. The list is not complete and importantly Support Vector Machines are completely missing which can be a very useful tool in supervised analysis. A core idea to supervised models is they have a training phase and a predicting phase.

0.12 Classification

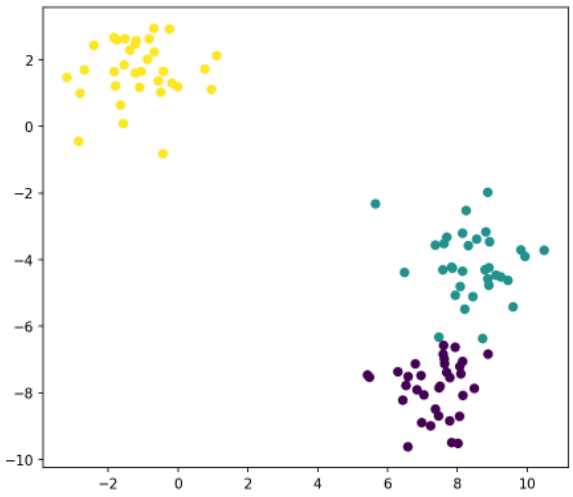
0.12.1 Lets create some data...

Here we create some bivariate data 'blobs' with Gaussian distribution. This time the blobs have classes assigned to them. The table

```
blob_data, blob_labels = make_blobs(n_samples=100,
                                    random_state=2018)
test_pts = pd.DataFrame(blob_data, columns=['x', 'y'])
test_pts['group_id'] = blob_labels
```

```
fig,ax = plt.subplots(1,2,figsize=(15,6),dpi=150)
ax[1].scatter(test_pts.x, test_pts.y,
              c=test_pts.group_id,
              cmap='viridis')
ccolors = plt.cm.BuPu(np.full(3, 0.1))
pd.plotting.table(data=test_pts.sample(10).round(decimals=2), ax=ax[0], loc='center',
 ↪colColours=ccolors)
ax[0].axis('off');
```

	x	y	group_id
76	5.65	-2.32	1.0
79	-1.52	2.63	2.0
92	7.62	-3.52	1.0
24	-0.83	2.63	2.0
99	-0.01	1.19	2.0
20	5.43	-7.46	0.0
41	9.94	-3.9	1.0
48	-2.85	-0.45	2.0
42	-1.79	1.22	2.0
57	7.37	-8.49	0.0



0.12.2 Nearest Neighbor (or K Nearest Neighbors)

The technique is as basic as it sounds, it basically finds the nearest point to what you have put in.

The *k nearest neighbors* algorithm makes the inference based on a point cloud of training point. When the model is presented with a new point it computes the distance to the closest points in the model. The *k* in the algorithm name indicates how many neighbors should be considered. E.g. *k=3* means that the major class of the three nearest neighbors is assigned the tested point.

Looking at the example below we would say that using three neighbors the

- Green hiker would claim he is in a spruce forest.
- Orange hiker would claim he is in a mixed forest.
- Blue hiker would claim he is in a birch forest.

Which cluster assigns the class?

Our k-nearest neighbor is trained for three classes.

```
plt.figure(figsize=[6,6],dpi=100)
plt.scatter(test_pts.x, test_pts.y,
              c=test_pts.group_id,
              cmap='viridis')
plt.plot(2,-2,'X',color='cornflowerblue',markersize=10,label='Point A')
plt.plot(7.7,-6.1,'o',color='darkorange',markersize=10,label='Point B')
plt.plot(8.5,-4,'P',color='deeppink',markersize=10,label='Point C')
plt.legend();
```

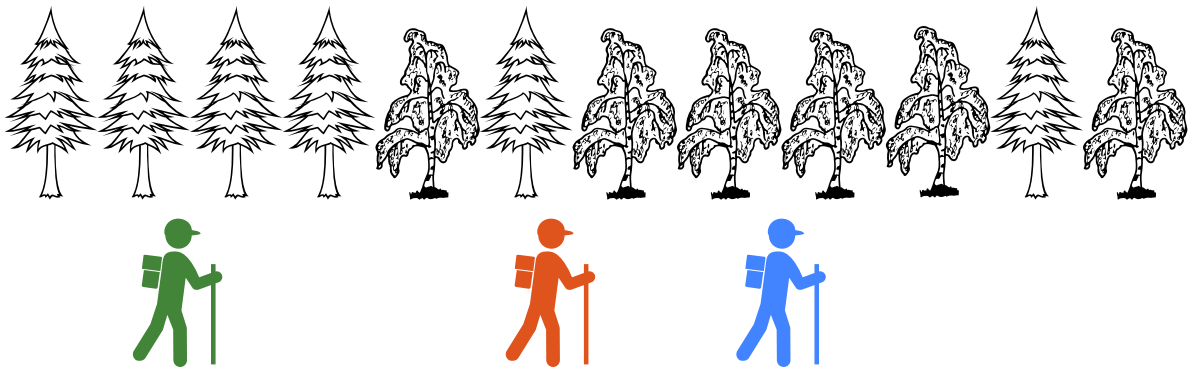
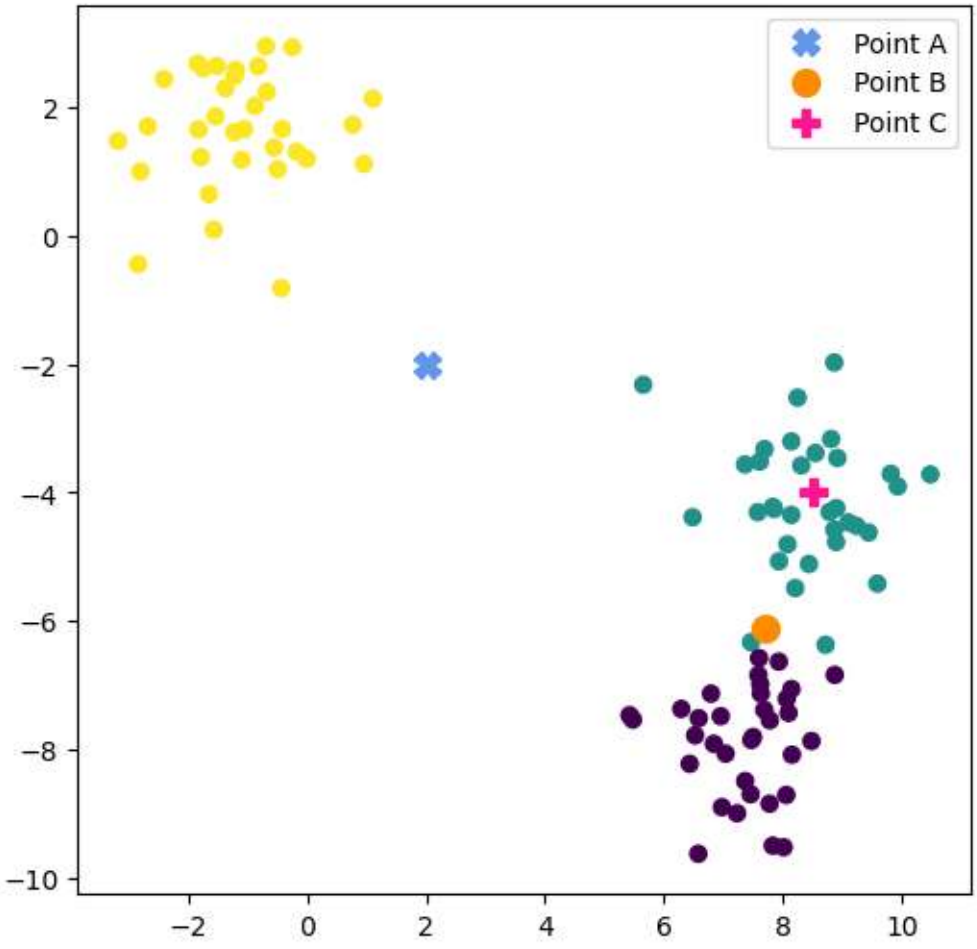


Fig. 1: Depending on the where the hiker is standing he makes the the conclusion that is either in a birch or spruce forest.



0.12.3 Text example

Training data

Value	1	2	3	4
Class	I	am	a	dog

Start the training

```
from sklearn.neighbors import KNeighborsClassifier
import numpy as np
k_class = KNeighborsClassifier(1)
k_class.fit(X=np.reshape([0, 1, 2, 3], (-1, 1)),
            y=['I', 'am', 'a', 'dog'])
```

```
KNeighborsClassifier(n_neighbors=1)
```

0.12.4 Nearest neighbor predictions

Basic test

Same values as training

```
print(k_class.predict(np.reshape([0, 1, 2, 3],
                                 (-1, 1))))
```

```
['I' 'am' 'a' 'dog']
```

Testing with different values

What happens if we don't enter exact matches?

```
print("Input  1.2 :",k_class.predict(np.reshape([1.2], (1, 1))))
print("Input  1.5 :",k_class.predict(np.reshape([1.5], (1, 1))))
print("Input  1.8 :",k_class.predict(np.reshape([1.8], (1, 1))))
print("Input 100 :",k_class.predict(np.reshape([100], (1, 1))))
```

```
Input  1.2 : ['am']
Input  1.5 : ['am']
Input  1.8 : ['a']
Input 100 : ['dog']
```

In this case we entered some data points between two categories, which for this classifier resulted in a rounding effect in the n=1 case.

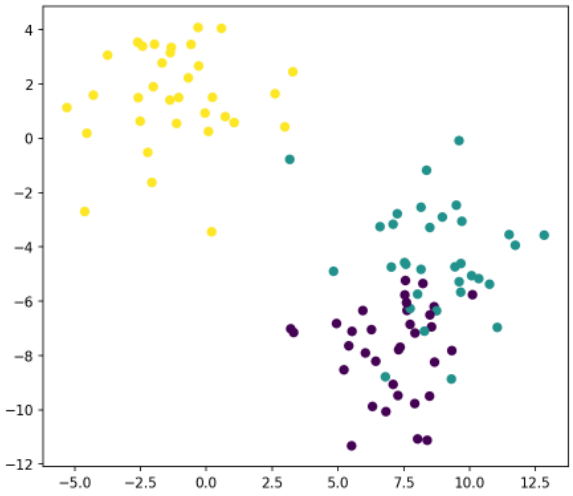
The last case our entered data point is far away from the trained data. Now, we get the last class in the list, which is the closest eventhough it is far away.

0.12.5 Let's come back to the blob data

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
```

```
blob_data, blob_labels = make_blobs(n_samples=100,
                                    cluster_std=2.0,
                                    random_state=2018)
test_pts = pd.DataFrame(blob_data, columns=['x', 'y'])
test_pts['group_id'] = blob_labels

fig, ax = plt.subplots(1,2,figsize=(15,6),dpi=150)
ax[1].scatter(test_pts.x, test_pts.y, c=test_pts.group_id, cmap='viridis')
ccolors = plt.cm.BuPu(np.full(3, 0.1))
pd.plotting.table(data=test_pts.sample(10).round(decimals=2), ax=ax[0], loc='center',
 ↪colColours=ccolors)
ax[0].axis('off');
```

	x	y	group_id
95	7.03	-4.74	1.0
74	8.65	-6.2	0.0
21	8.48	-9.5	0.0
33	7.74	-6.85	0.0
15	8.55	-6.95	0.0
36	7.28	-9.47	0.0
25	-1.13	0.55	2.0
27	-4.54	0.19	2.0
88	9.32	-7.82	0.0
6	-0.05	0.93	2.0



0.12.6 Training the model using one neighbor

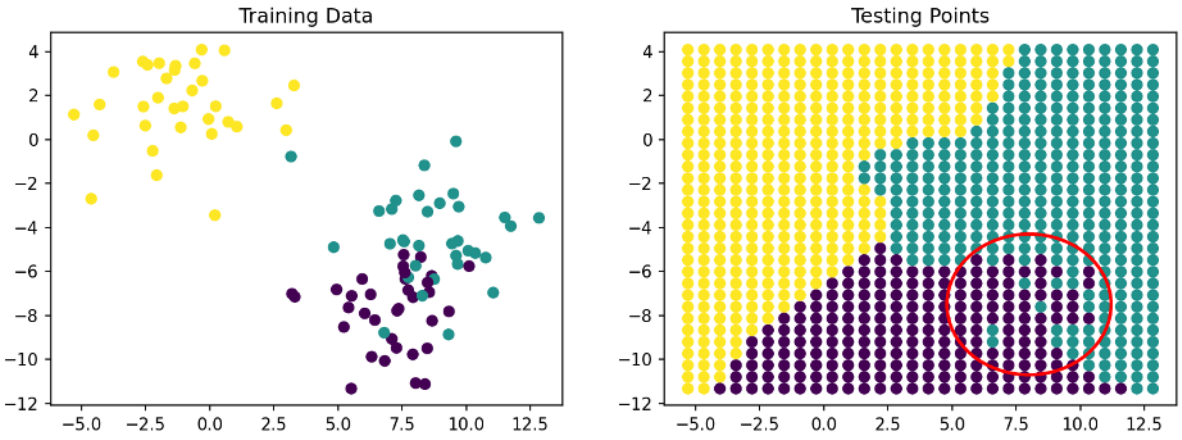
```
# Define classifier
k_class = KNeighborsClassifier(1)

# Train the classifier model with data
k_class.fit(test_pts[['x', 'y']], test_pts['group_id'])
```

```
KNeighborsClassifier(n_neighbors=1)
```

Resulting prediction map for a single neighbor

```
xx, yy = np.meshgrid(np.linspace(test_pts.x.min(), test_pts.x.max(), 30),np.
 ↪linspace(test_pts.y.min(), test_pts.y.max(), 30),indexing='ij');
grid_pts = pd.DataFrame(dict(x=xx.ravel(), y=yy.ravel()))
grid_pts['predicted_id'] = k_class.predict(grid_pts[['x', 'y']])
```

```
import matplotlib.patches as patches
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4),dpi=150)
ax1.scatter(test_pts.x, test_pts.y, c=test_pts.group_id, cmap='viridis');
ax1.set_title('Training Data');
ax2.scatter(grid_pts.x, grid_pts.y, c=grid_pts.predicted_id, cmap='viridis');
ax2.set_title('Testing Points');

circle = patches.Circle((8, -7.5), 3.2, color='red', fill=False,linewidth=2)

# Add the circle to the axis
ax2.add_patch(circle);
```



In this example we trained the classifier with some data points appearing in clouds. The next step is to see how the classifier performs on unknown data. This is done by creating a mesh of equidistant points, each point is then fed to the classifier and we can see where the decision boundaries are for the trained classifier. You can see that there are some irregularities along the boundaries between the classes. One reason is that the trainingdata has overlapping classes and the other is the trained model.

Here, we used a data frame for the presentation of the point tables.

0.12.7 Stabilizing Results - Increase number of neighbors

- We can see here that the result is thrown off by single points
- Prediction can be improved by using more than the nearest neighbor

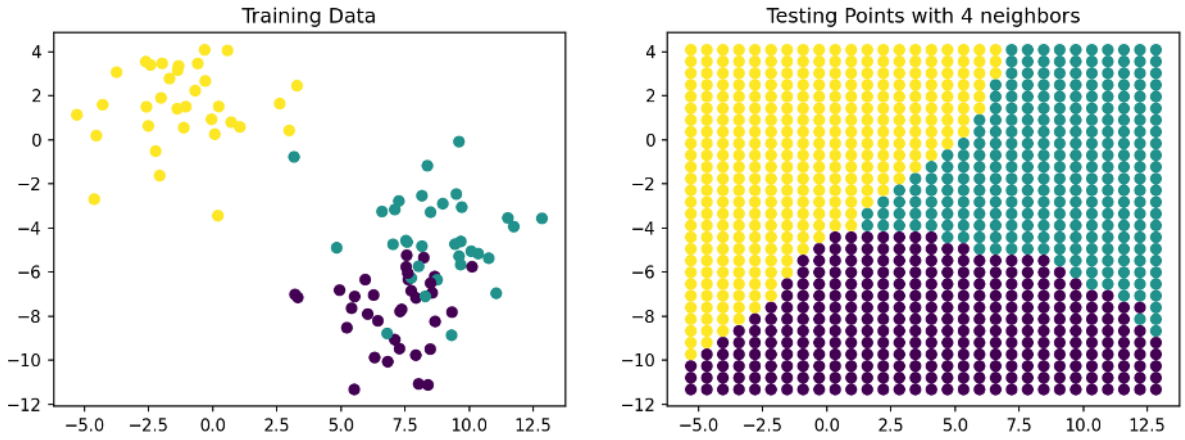
```
k_class = KNeighborsClassifier(n_neighbors=4)
k_class =  k_class.fit(test_pts[['x', 'y']], test_pts['group_id'])
xx, yy = np.meshgrid(np.linspace(test_pts.x.min(), test_pts.x.max(), 30),
                     np.linspace(test_pts.y.min(), test_pts.y.max(), 30),
                     indexing='ij'
                     )
```

(continues on next page)

(continued from previous page)

```
grid_pts = pd.DataFrame(dict(x=xx.ravel(), y=yy.ravel()))
grid_pts['predicted_id'] = k_class.predict(grid_pts[['x', 'y']])
```

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4),dpi=150)
ax1.scatter(test_pts.x, test_pts.y, c=test_pts.group_id, cmap='viridis');
ax1.set_title('Training Data');
ax2.scatter(grid_pts.x, grid_pts.y, c=grid_pts.predicted_id, cmap='viridis');
ax2.set_title('Testing Points with 4 neighbors');
```



0.13 Linear Regression

- Linear regression is a fancy-name for linear curve fitting
- Fitting a line through points (sometimes in more than one dimension).
- It is a very basic method,
  - is easy to understand,
  - interpret
  - and fast to compute

Fits the model

$X\theta = y$

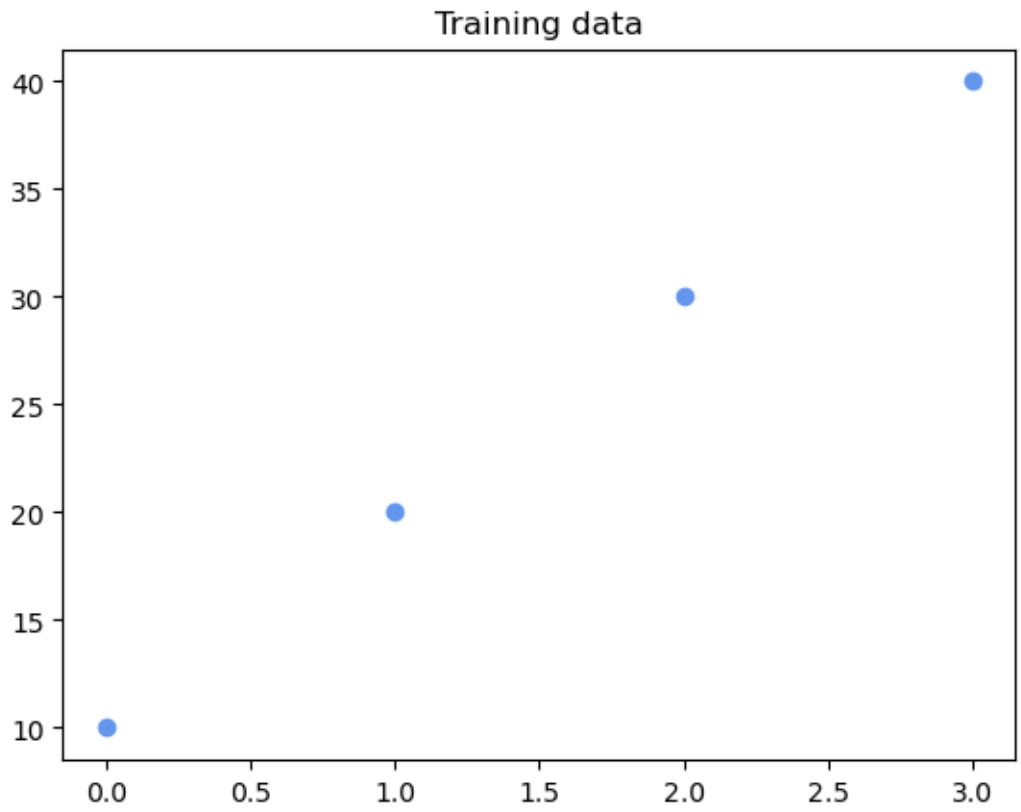
Where

- X is a matrix describing the model
- y the measured values
- θ the fitted parameters

0.13.1 We need data to fit...

```
from sklearn.linear_model import LinearRegression

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40] )

plt.plot(x,y,'o',color='cornflowerblue'); plt.title('Training data')

l_reg = LinearRegression()
l_reg.fit(X=np.reshape(x, (-1, 1)),y=y)

print("slope: {0:0.4}, intercept: {1:0.4}".format(l_reg.coef_[0], l_reg.intercept_))
```

```
slope: 10.0, intercept: 10.0
```



Let's try the model on some data points

```
print('An array of values:', l_reg.predict(np.reshape([0, 1, 2, 3], (-1, 1))))
print('x:',-100, '=>',       l_reg.predict(np.reshape([-100], (1, 1))))
print('x: ',500, '=>',       l_reg.predict(np.reshape([500], (1, 1))))
```

```
An array of values: [10. 20. 30. 40.]
x: -100 => [-990.]
x:  500 => [5010.]
```

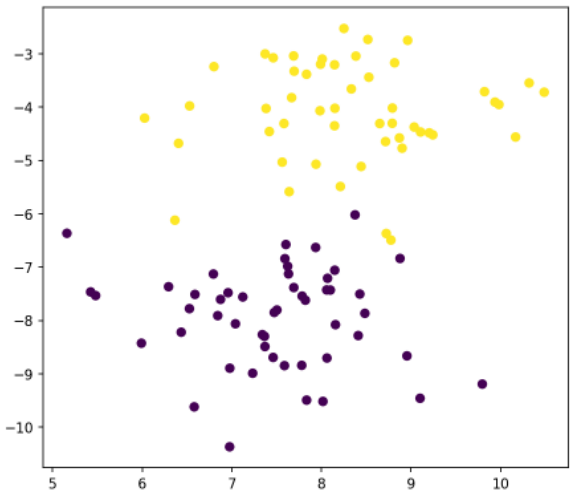
0.13.2 Regression on blob data

```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

import numpy as np
import pandas as pd
```

```
blob_data, blob_labels = make_blobs(centers=2, n_samples=100,
                                    random_state=2018)
test_pts = pd.DataFrame(blob_data, columns=['x', 'y'])
test_pts['group_id'] = blob_labels
```

```
fig, ax = plt.subplots(1,2,figsize=(15,6),dpi=150)
ax[1].scatter(test_pts.x, test_pts.y, c=test_pts.group_id, cmap='viridis')

ccolors = plt.cm.BuPu(np.full(3, 0.1))
pd.plotting.table(data=test_pts.sample(10).round(decimals=2), ax=ax[0], loc='center',
 ↪colColours=ccolors)
ax[0].axis('off');
```

	x	y	group_id
98	7.63	-7.13	0.0
24	8.78	-6.49	1.0
48	8.15	-4.02	1.0
61	8.16	-8.08	0.0
13	7.5	-7.8	0.0
67	7.34	-8.27	0.0
42	8.96	-2.75	1.0
88	6.98	-10.37	0.0
8	8.9	-4.77	1.0
63	7.47	-7.85	0.0



0.13.3 Train the regression model

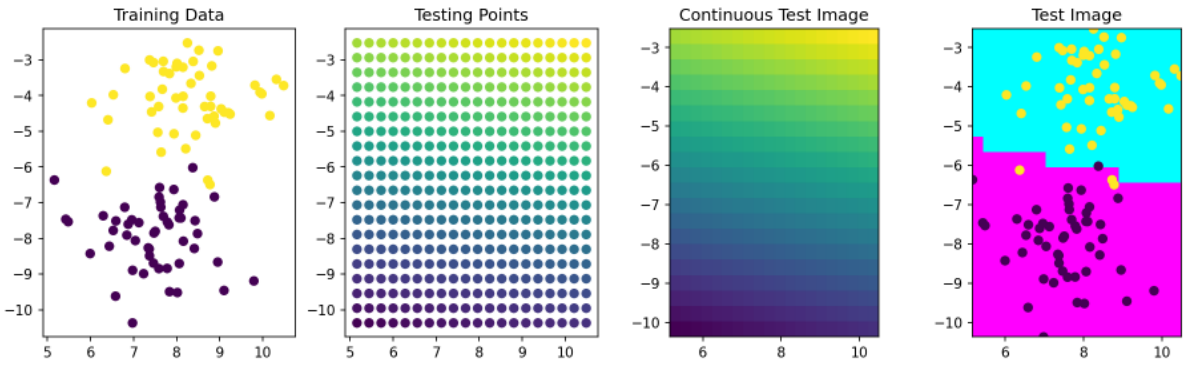
```
l_reg = LinearRegression()
l_reg.fit(test_pts[['x', 'y']], test_pts['group_id'])
print('Slope', l_reg.coef_)
print('Offset', l_reg.intercept_)
```

```
Slope [0.04813137 0.20391973]
Offset 1.346929708594462
```

Evaluate the regression model

```
xx, yy = np.meshgrid(np.linspace(test_pts.x.min(), test_pts.x.max(), 20),
                     np.linspace(test_pts.y.min(), test_pts.y.max(), 20),
                     indexing='ij'
                     )
grid_pts = pd.DataFrame(dict(x=xx.ravel(), y=yy.ravel()))
grid_pts['predicted_id'] = l_reg.predict(grid_pts[['x', 'y']])
```

```
fig, (ax1, ax2, ax3,ax4) = plt.subplots(1, 4, figsize=(15, 4),dpi=150)
ax1.scatter(test_pts.x, test_pts.y, c=test_pts.group_id, cmap='viridis');
ax1.set_title('Training Data')
ax2.scatter(grid_pts.x, grid_pts.y, c=grid_pts.predicted_id, cmap='viridis');
ax2.set_title('Testing Points')
ax3.imshow(grid_pts.predicted_id.values.reshape(
    xx.shape).T[::-1], cmap='viridis', extent=[test_pts.x.min(), test_pts.x.max(),␣
 ↪test_pts.y.min(), test_pts.y.max()])
ax3.set_title('Continuous Test Image');
ax4.imshow(grid_pts.predicted_id.values.reshape(
    xx.shape).T[::-1]<0.5, cmap='cool', extent=[test_pts.x.min(), test_pts.x.max(),␣
 ↪test_pts.y.min(), test_pts.y.max()])
ax4.scatter(test_pts.x, test_pts.y, c=test_pts.group_id, cmap='viridis');
ax4.set_title('Test Image');
```



0.14 Decision trees

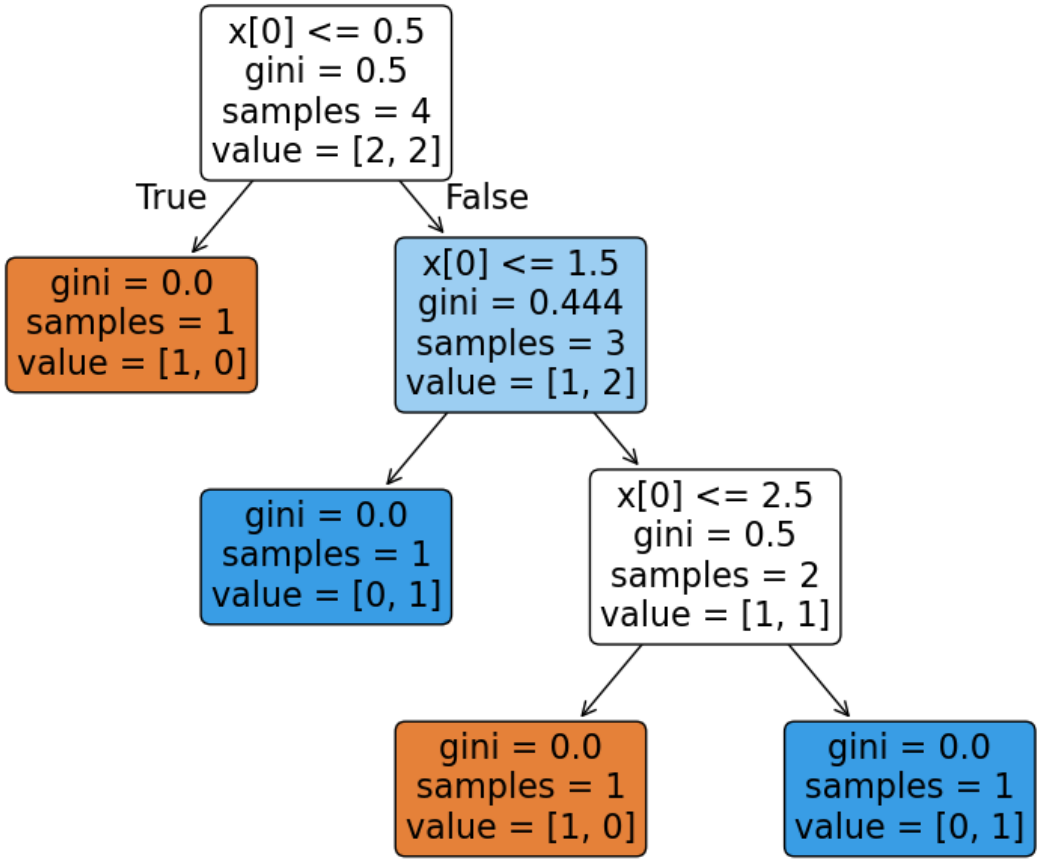
- SciKit Learn documentation on trees
- SciKit Learn trees explained
- Hastie et al., Elements Of Statistical Learning, 2009 Section 9.2 Trees.

```
from sklearn.tree import export_graphviz
import sklearn.tree as tree
import graphviz
from sklearn.tree import DecisionTreeClassifier
import numpy as np
from IPython.display import SVG

def show_tree(in_tree):
    return graphviz.Source(export_graphviz(in_tree, out_file=None))
```

0.14.1 Create a decision tree classifier

We want to identify odd numbers in the sequence [0, 1, 2, 3]

```
d_tree = DecisionTreeClassifier()
d_tree.fit(X=np.reshape([0, 1, 2, 3], (-1, 1)),
           y=[0, 1, 0, 1])
fig, axes = plt.subplots(nrows = 1,ncols = 1,figsize = (10,8))

tree.plot_tree(d_tree,filled=True,rounded=True,fontsize=16);
```



$$\text{Gini index} = 1 - \sum_{i=1}^{N} {P_i}^2$$

In the context of decision trees and machine learning, the Gini index is a measure of impurity used to evaluate the quality of a split in a decision tree. It is often used as a criterion for determining the optimal split when growing a decision tree.

The Gini index for a node in a decision tree is calculated as follows:

$$G = 1 - \sum_{i=1}^{c} p_i^2$$

Where:

- G is the Gini index for the node.
- c is the number of classes (or categories) in the dataset.
- p_i is the proportion of instances in the i-th class at the node.

The Gini index ranges from 0 to 1, with lower values indicating less impurity. A node with a Gini index of 0 means that all instances belong to the same class, making it a pure node. A node with a Gini index of 1 means that the distribution of classes is perfectly impure, with an equal proportion of instances from each class.

When building a decision tree, the algorithm evaluates various potential splits in the data and selects the split that minimizes the Gini index, as it aims to create child nodes that are as pure as possible. This process is repeated recursively for each node in the tree until certain stopping criteria are met, such as reaching a maximum depth or minimum number of samples per node.

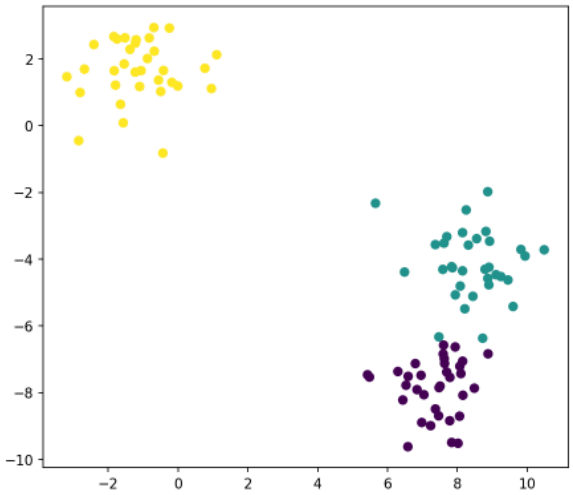
0.14.2 Decision trees on the blob data

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
%matplotlib inline
```

```
blob_data, blob_labels = make_blobs(n_samples=100, random_state=2018)
test_pts = pd.DataFrame(blob_data, columns=['x', 'y'])
test_pts['group_id'] = blob_labels
```

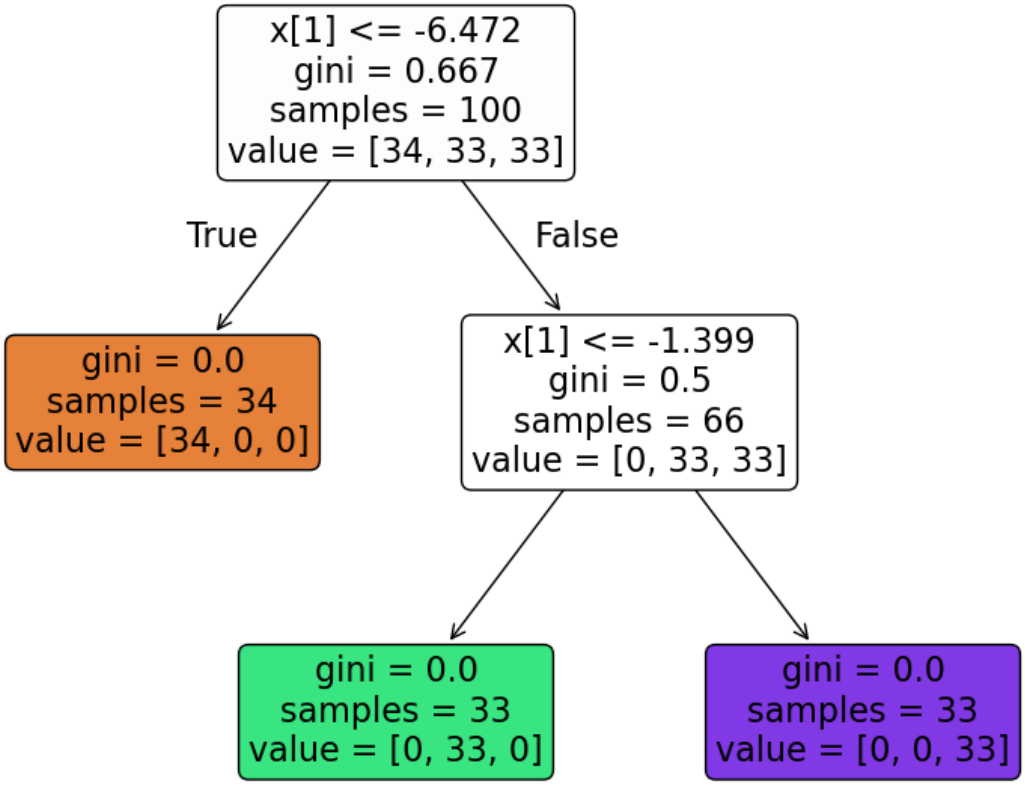
```
fig,ax = plt.subplots(1,2, figsize=(15,6),dpi=150)

ccolors = plt.cm.BuPu(np.full(3, 0.1))
pd.plotting.table(data=test_pts.sample(10).round(decimals=2), ax=ax[0], loc='center',
 ↪colColours=ccolors);
ax[0].axis('off');
ax[1].scatter(test_pts.x, test_pts.y, c=test_pts.group_id, cmap='viridis');
```

	x	y	group_id
75	9.59	-5.42	1.0
3	-0.69	2.94	2.0
53	8.82	-3.17	1.0
60	8.9	-4.24	1.0
38	7.78	-8.84	0.0
21	8.06	-8.71	0.0
66	8.02	-9.52	0.0
89	6.84	-7.91	0.0
0	8.55	-3.39	1.0
35	-3.18	1.47	2.0



Train the tree

```
d_tree = DecisionTreeClassifier()
d_tree.fit(test_pts[['x', 'y']],
           test_pts['group_id'])

fig, axes = plt.subplots(nrows = 1,ncols = 1,figsize = (10,8))
tree.plot_tree(d_tree,filled=True,rounded=True,fontsize=16);
```



Let's look at the decision map

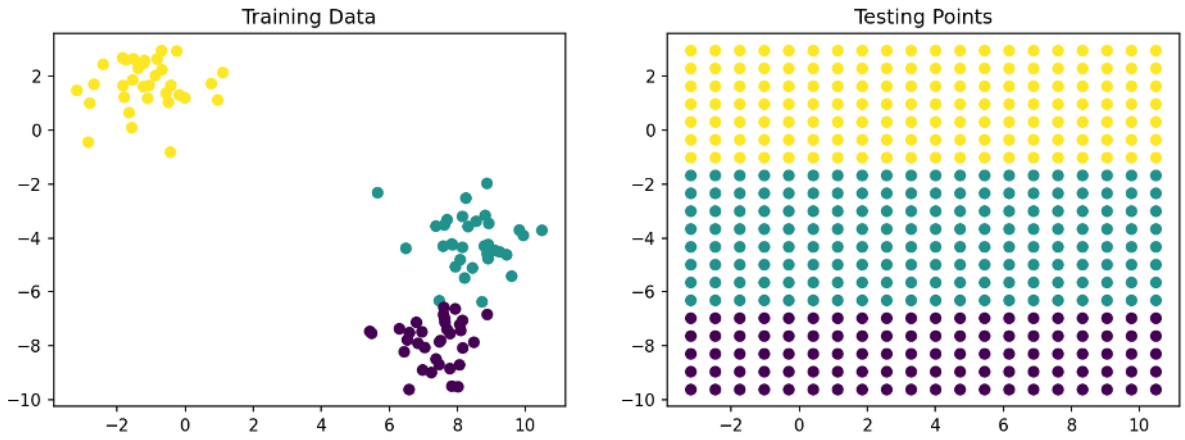
```
xx, yy = np.meshgrid(np.linspace(test_pts.x.min(), test_pts.x.max(), 20),
                     np.linspace(test_pts.y.min(), test_pts.y.max(), 20),
                     indexing='ij'
                     )
grid_pts = pd.DataFrame(dict(x=xx.ravel(), y=yy.ravel()))
grid_pts['predicted_id'] = d_tree.predict(grid_pts[['x', 'y']])
```

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4),dpi=150)
ax1.scatter(test_pts.x, test_pts.y, c=test_pts.group_id, cmap='viridis')
ax1.set_title('Training Data')
```

(continues on next page)

(continued from previous page)

```
ax2.scatter(grid_pts.x, grid_pts.y, c=grid_pts.predicted_id, cmap='viridis')
ax2.set_title('Testing Points');
```



0.14.3 Random Forests

Forests are basically the idea of taking a number of trees and bringing them together.

So rather than taking a single tree to do the classification, you divide the samples and the features to make different trees and then combine the results. One of the more successful approaches is called Random Forests or as a video

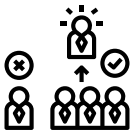


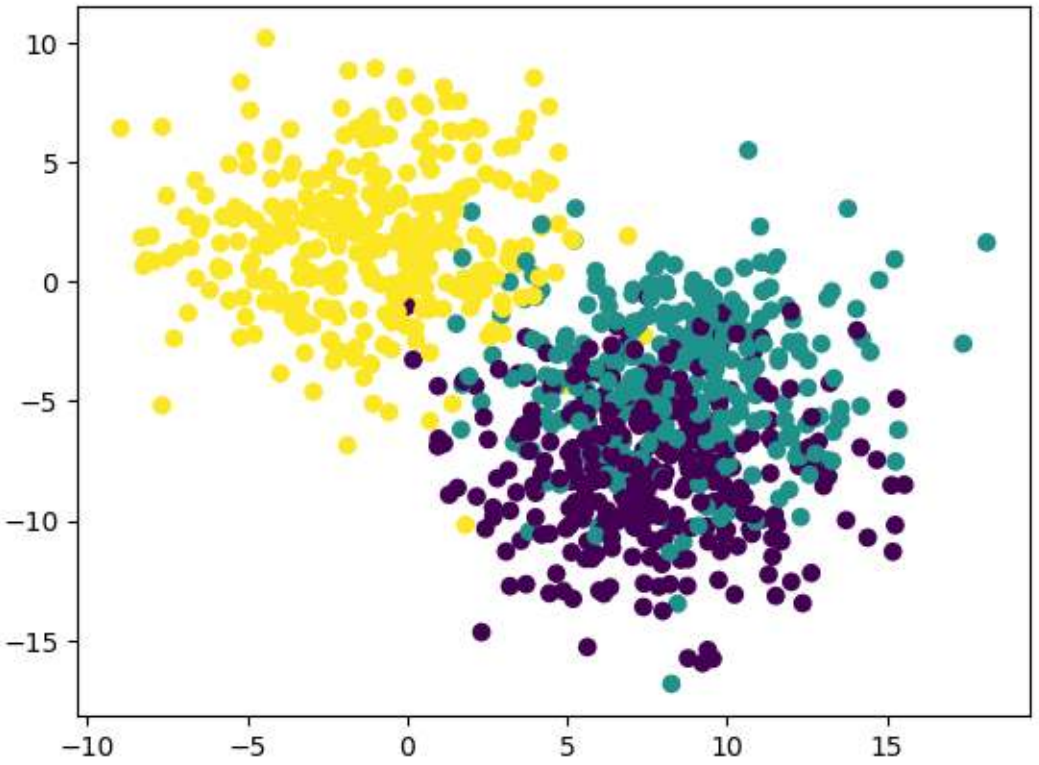
Fig. 1: Random forests train trees on different fractions of the data.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
%matplotlib inline
```

Let's make some new blob data

The data in the previous example was easily separated. In the next example we look at the case when the classes are severely overlapping.

```
blob_data, blob_labels = make_blobs(n_samples=1000,
                                    cluster_std=3,
                                    random_state=2018)
test_pts = pd.DataFrame(blob_data, columns=['x', 'y'])
test_pts['group_id'] = blob_labels
plt.scatter(test_pts.x, test_pts.y, c=test_pts.group_id, cmap='viridis');
```



Train a forest

Now we train a tiny forest with five trees.

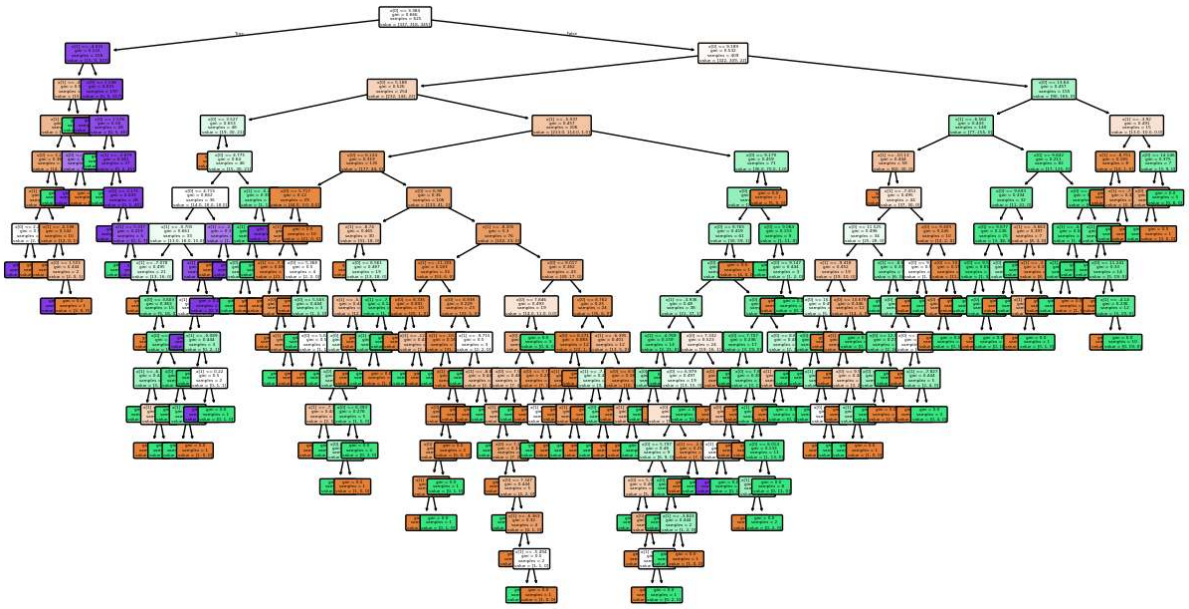
```
from sklearn.ensemble import RandomForestClassifier
rf_class = RandomForestClassifier(n_estimators=5, random_state=2018)
rf_class.fit(test_pts[['x', 'y']], test_pts['group_id'])

print('Build ', len(rf_class.estimators_), 'decision trees')
```

```
Build  5 decision trees
```

Inspect one tree

Looking at a single tree in the forest

```
fig, axes = plt.subplots(nrows = 1,ncols = 1,figsize = (15,8))
tree.plot_tree(rf_class.estimators_[1],filled=True,rounded=True,fontsize=3);
```



This thee is far more complicated than the tree in the previous example. What is more, this is also only one out of five trees that were trained on the overlapping blob data set.

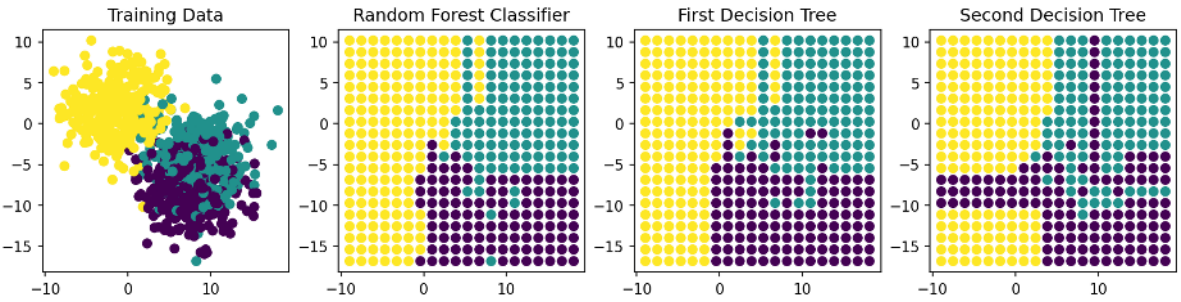
The combined outcome of the five trees in the forest will assingn the class in the prediction phase.

Looking at the performance of the forest

Here, we'll look at how the forest behaves and some of its trees.

```
xx, yy = np.meshgrid(np.linspace(test_pts.x.min(), test_pts.x.max(), 20),
                     np.linspace(test_pts.y.min(), test_pts.y.max(), 20),
                     indexing='ij'
                     )
grid_pts = pd.DataFrame(dict(x=xx.ravel(), y=yy.ravel()))
rnd_forest_classes = rf_class.predict(grid_pts[['x', 'y']]) # Prediction of the forest

rnd_tree1          = rf_class.estimators_[0].predict(grid_pts[['x', 'y']].values) #␣
 ↪Prediction of the first tree
rnd_tree2          = rf_class.estimators_[1].predict(grid_pts[['x', 'y']].values) #␣
 ↪Prediction of the second tree
```

```
# Visualization
fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(14, 3), dpi=150)
ax1.scatter(test_pts.x, test_pts.y, c=test_pts.group_id, cmap='viridis')
ax1.set_title('Training Data')
ax2.scatter(grid_pts.x, grid_pts.y, c=rnd_forest_classes, cmap='viridis')
ax2.set_title('Random Forest Classifier')

ax3.scatter(grid_pts.x, grid_pts.y, c=rnd_tree1, cmap='viridis')
ax3.set_title('First Decision Tree')

ax4.scatter(grid_pts.x, grid_pts.y, c=rnd_tree2, cmap='viridis')
ax4.set_title('Second Decision Tree');
```



We see here that the individual trees may not be so precise but the final result is still quite convincing.

0.15 Pipelines

We will use the idea of pipelines generically here to refer to the combination of steps that need to be performed to solve a problem.

Pipelines are a technical solution to reduce the amount of coding.

Consists of a series of

- transformations
- predictors
- Pipeline simplifies machine learning workflows
- Prevents data leakage by applying transformations correctly
- Reduces redundant code & makes hyperparameter tuning easier
- Useful for production models as everything is encapsulated in one object

0.15.1 Let's the return to the blobs

```
blob_data, blob_labels = make_blobs(n_samples=100,
                                    random_state=2018)
test_pts = pd.DataFrame(blob_data, columns=['x', 'y'])
test_pts['group_id'] = blob_labels
```

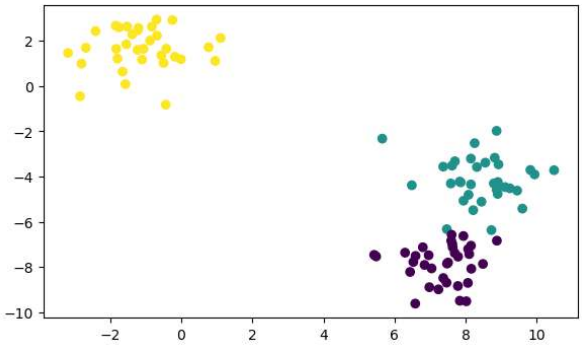
```
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 4))

cell_text = []
sampled_pts = test_pts.sample(5)  # Sample once outside the loop to maintain␣
↪consistency

for row in range(5):
    cell_text.append(sampled_pts.iloc[row].tolist())  # Convert Series to list

# Create the table
ax1.table(cellText=cell_text, colLabels=test_pts.columns, loc='center')
ax1.axis('off')
```

(continues on next page)

(continued from previous page)

```
# Scatter plot
ax2.scatter(test_pts.x, test_pts.y, c=test_pts.group_id, cmap='viridis')

plt.show()
```

x	y	group_id
9.594570288850747	-5.417980006657151	1.0
9.106493533312577	-4.465403344675514	1.0
-1.3775663332993913	2.2884693490895813	2.0
8.922199016509774	-3.463288012395787	1.0
-2.681981731748765	1.6952054230129885	2.0



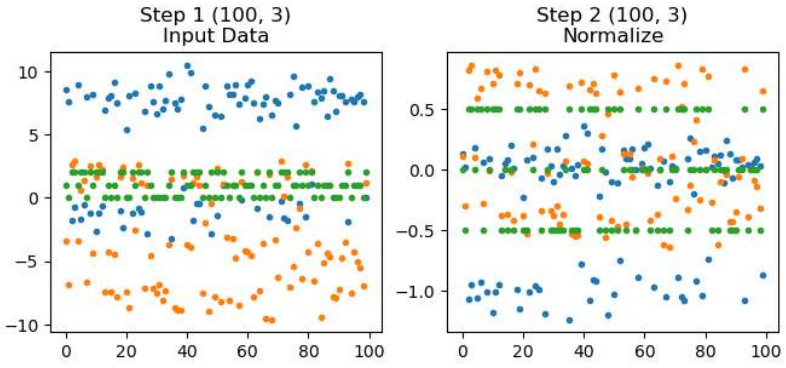
0.15.2 A basic pipeline

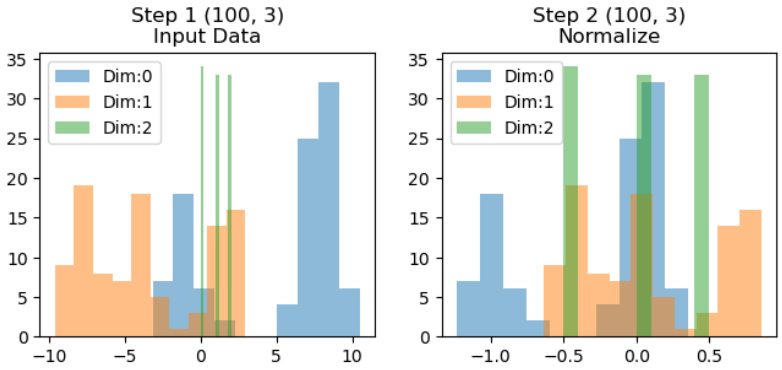
Our first pipeline has one step that normalizes the data using a RobustScaler

```
from pipe_utils import show_pipe # QBI utilities package
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import RobustScaler

simple_pipe = Pipeline([('Normalize', RobustScaler())])
simple_pipe.fit(test_pts.values)
```

```
Pipeline(steps=[('Normalize', RobustScaler())])
```

```
show_pipe(simple_pipe, test_pts.values,figsize=[12,3])
show_pipe(simple_pipe, test_pts.values, show_hist=True, show_legend=True,figsize=[12,
↪3])
```





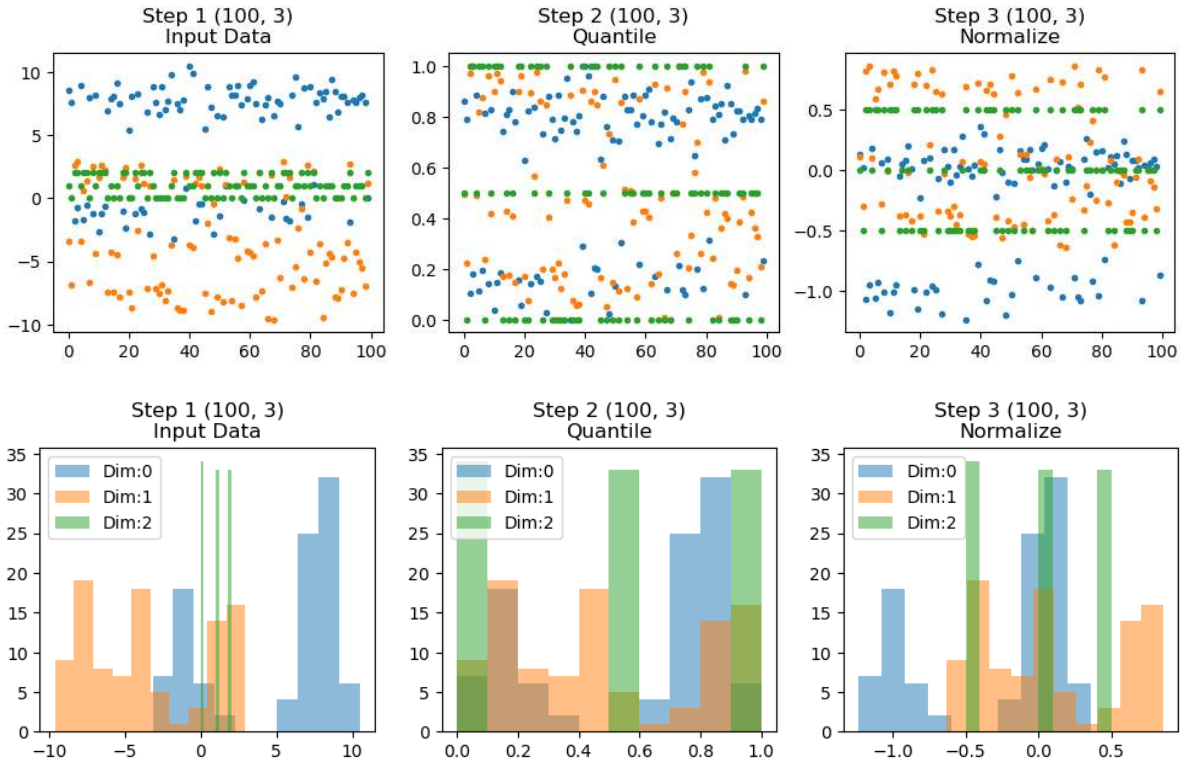
Adding tasks to the pipeline

Next we populate the pipeline with a QuantileTransformer to force the data into a uniform distribution.

```
from sklearn.preprocessing import QuantileTransformer
longer_pipe = Pipeline([('Quantile', QuantileTransformer(n_quantiles=2)),
                        ('Normalize', RobustScaler())])
longer_pipe.fit(test_pts.values)
```

```
Pipeline(steps=[('Quantile', QuantileTransformer(n_quantiles=2)),
                ('Normalize', RobustScaler())])
```

```
show_pipe(longer_pipe, test_pts.values,figsize=[12,3])
show_pipe(longer_pipe, test_pts.values, show_hist=True,figsize=[12,3])
```



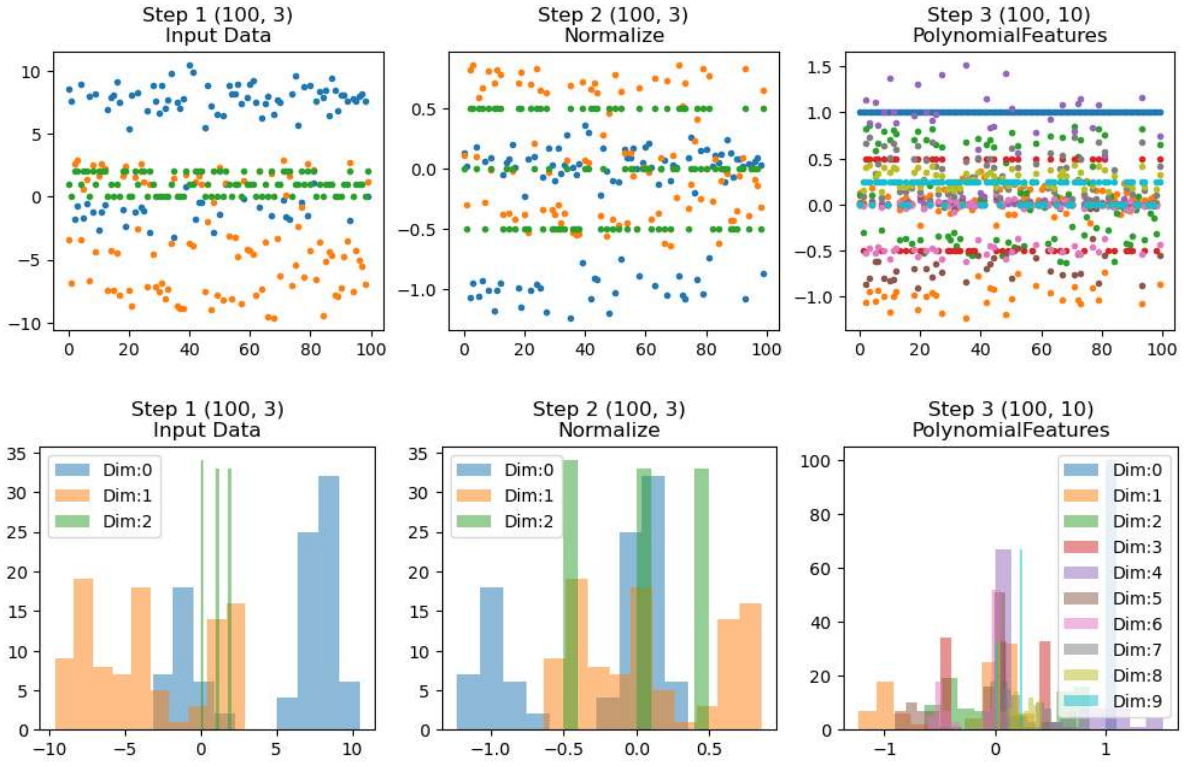
Adding polynomial features to the pipeline

We saw earlier that it can make sense to add polynomial features to the segmentation.

```
from sklearn.preprocessing import PolynomialFeatures
messy_pipe = Pipeline([
    ('Normalize', RobustScaler()),
    ('PolynomialFeatures', PolynomialFeatures(2))])
messy_pipe.fit(test_pts.values)
```

```
Pipeline(steps=[('Normalize', RobustScaler()),
                ('PolynomialFeatures', PolynomialFeatures())])
```

```
show_pipe(messy_pipe, test_pts.values,figsize=[12,3])
show_pipe(messy_pipe, test_pts.values, show_hist=True,figsize=[12,3])
```



0.15.3 Classification using a pipeline

A common problem of putting images into categories.

- The standard problem for this is classifying digits between 0 and 9 (MNIST).
- Fundamentally a classification problem is one where we are taking a large input (images, vectors, …) and trying to put it into a category.
  - Cats, Dogs
  - Cars, boats
  - etc.

A first example with pipeline processing

Let's load some images

- Images of numbers 0 to 9 (MNIST)
- 8×8 pixels
- 50 Samples

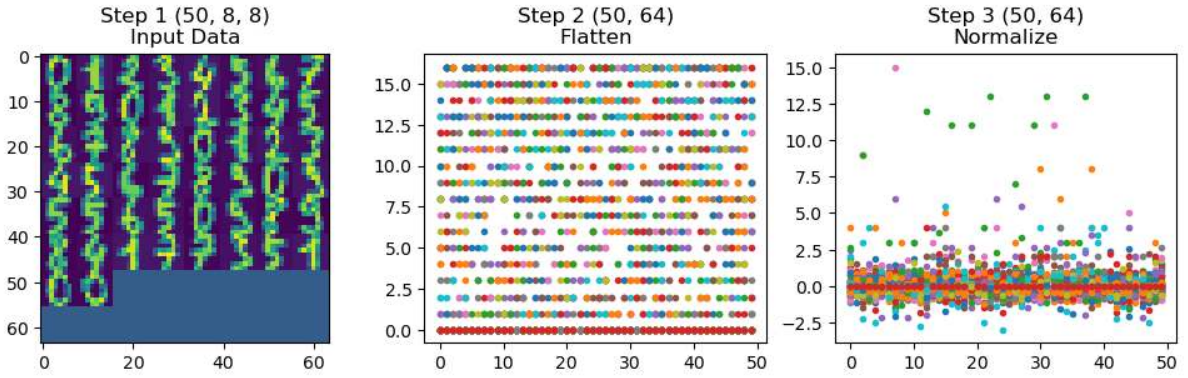
```
from sklearn.datasets import load_digits
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from pipe_utils import show_pipe
%matplotlib inline

# Load the number images
digit_ds = load_digits(return_X_y=False)

# Select the first 50 images and labels
img_data = digit_ds.images[:50]
digit_id = digit_ds.target[:50]
print('Image Data', img_data.shape)
```

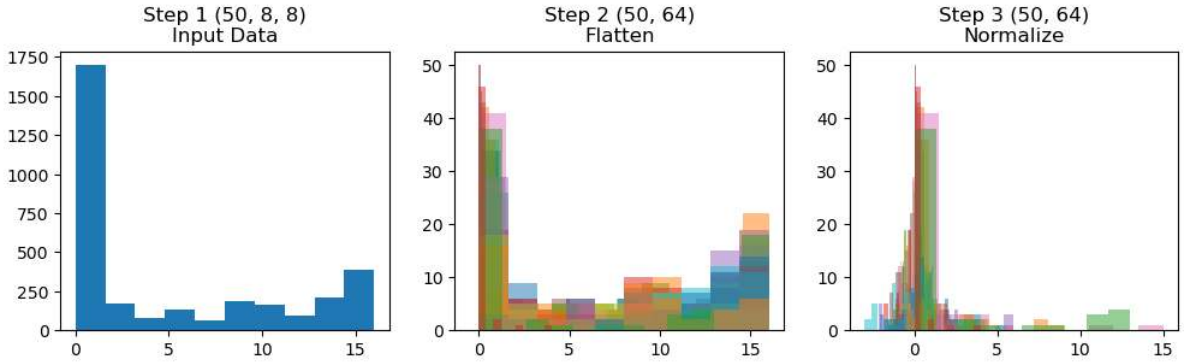
```
Image Data (50, 8, 8)
```

Run a preprocessing pipline

- Flatten images $8 \times 8 \rightarrow 1 \times 64$
- Normalize (robust scaling)

```
from pipe_utils import flatten_step
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import RobustScaler
digit_pipe = Pipeline([('Flatten', flatten_step),
                       ('Normalize', RobustScaler())])
digit_pipe.fit(img_data)

show_pipe(digit_pipe, img_data,figsize=[12,3])
show_pipe(digit_pipe, img_data, show_hist=True, show_legend=False,figsize=[12,3])
```

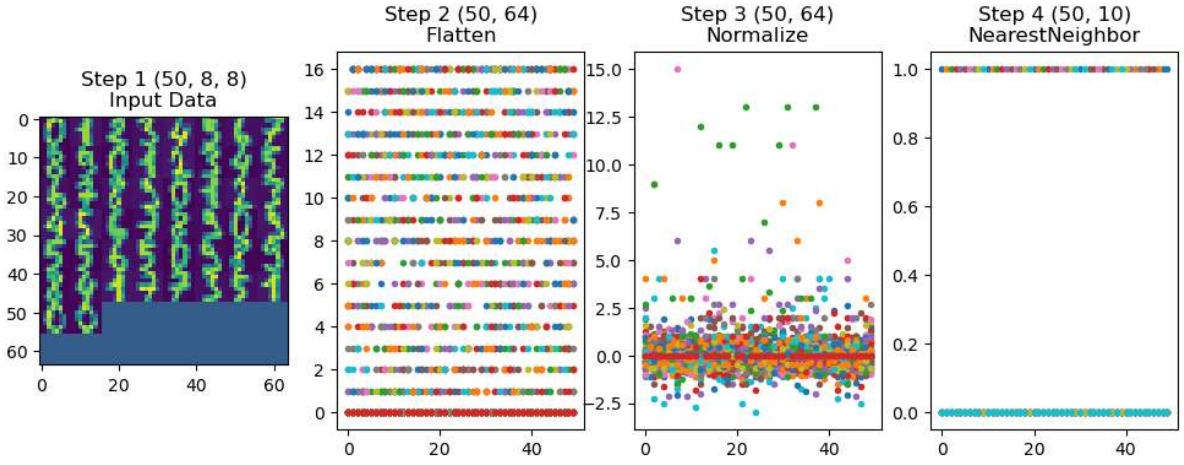




Add a classifier

- Add a K Nearest Neighbours classifier to the pipeline (K=1)
- Run the fit

```
from sklearn.neighbors import KNeighborsClassifier

digit_class_pipe = Pipeline([('Flatten', flatten_step),
                             ('Normalize', RobustScaler()),
                             ('NearestNeighbor', KNeighborsClassifier(1))])
digit_class_pipe.fit(img_data, digit_id)

show_pipe(digit_class_pipe, img_data,panels_in_row=4)
```



0.15.4 Test classifier performance

Let's test with traning data

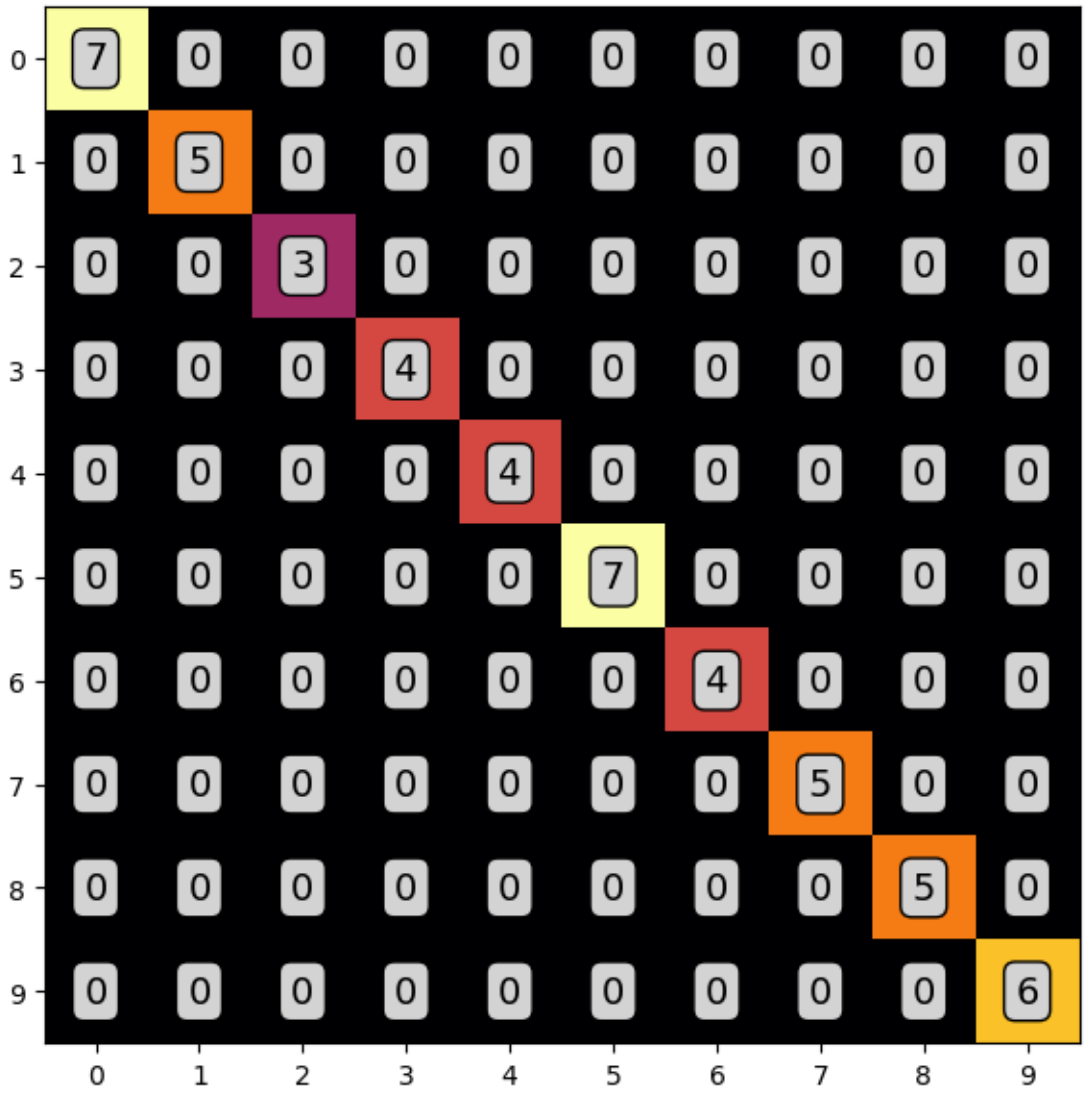
```
from sklearn.metrics import accuracy_score
pred_digit = digit_class_pipe.predict(img_data)

print('{0}% accuracy'.format(100*accuracy_score(digit_id, pred_digit)))
```

```
100.0% accuracy
```

How about the confusion matrix?

Let's look at the predictions in the confusion matrix.

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import plotsupport as ps
fig, ax1 = plt.subplots(1, 1, figsize=(8, 7), dpi=100)

ps.heatmap(confusion_matrix(digit_id, pred_digit),precision=0,ax=ax1)
```



Reporting the classifier output

We can also look at other performance metrics using a classification_report

```
from sklearn.metrics import classification_report
print(classification_report(digit_id, pred_digit))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         7
           1       1.00      1.00      1.00         5
           2       1.00      1.00      1.00         3
           3       1.00      1.00      1.00         4
           4       1.00      1.00      1.00         4
           5       1.00      1.00      1.00         7
           6       1.00      1.00      1.00         4
```

(continues on next page)

(continued from previous page)

```
           7       1.00      1.00      1.00         5
           8       1.00      1.00      1.00         5
           9       1.00      1.00      1.00         6

    accuracy                           1.00        50
   macro avg       1.00      1.00      1.00        50
weighted avg       1.00      1.00      1.00        50
```

0.15.5 Wow! We've built an amazing algorithm!

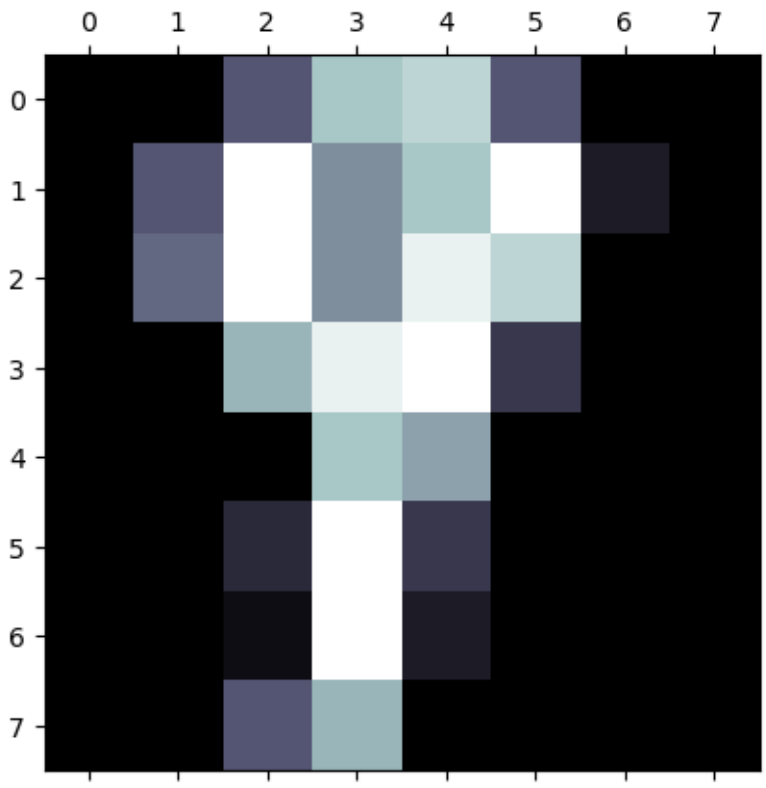
Let's patent it! Call Google!

Let's try again

This just too good. Let's make a new try using new, unseen data.

```
test_digit = np.array([[[0.,  0.,  6., 12., 13.,  6.,  0.,  0.],
                        [0.,  6., 16.,  9., 12., 16.,  2.,  0.],
                        [0.,  7., 16.,  9., 15., 13.,  0.,  0.],
                        [0.,  0., 11., 15., 16.,  4.,  0.,  0.],
                        [0.,  0.,  0., 12., 10.,  0.,  0.,  0.],
                        [0.,  0.,  3., 16.,  4.,  0.,  0.,  0.],
                        [0.,  0.,  1., 16.,  2.,  0.,  0.,  0.],
                        [0.,  0.,  6., 11.,  0.,  0.,  0.,  0.]]])
plt.matshow(test_digit[0], cmap='bone')
print('Prediction:', digit_class_pipe.predict(test_digit))
print('Real Value:', 9)
```

```
Prediction: [7]
Real Value: 9
```



0.15.6 Training, Validation, and Testing

Avoid the training "crime" in ML

https://www.kdnuggets.com/2017/08/dataiku-predictive-model-holdout-cross-validation.html

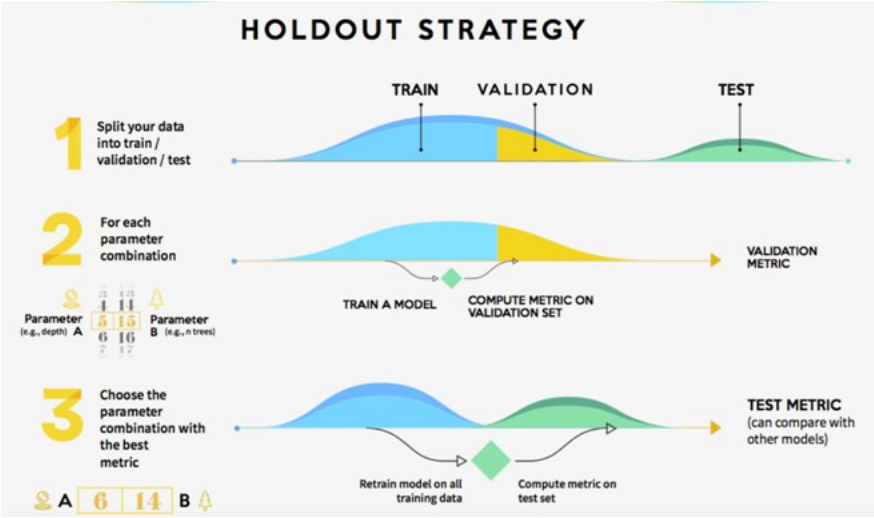


Fig. 1: Random forests train trees on different fractions of the data.

0.16 Regression using a pipeline

For regression, we can see

- it is very similarly to a classification
  - Predicts the category
- instead of trying to output discrete classes we can output on a continuous scale.
  - Predicts the **actual decimal number**.

```
from sklearn.datasets import load_digits
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from pipe_utils import show_pipe, flatten_step
%matplotlib inline
```

0.16.1 Load the digits data again

This time we load

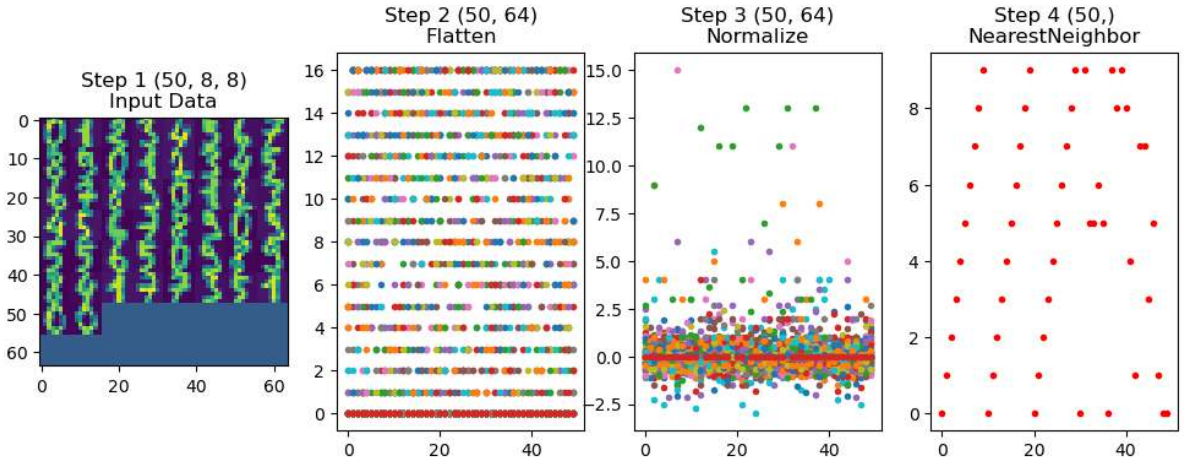
- 50 image/label pairs for the training
- 450 images/label pairs for the validation

```
digit_ds = load_digits(return_X_y=False)

img_data = digit_ds.images[:50]
digit_id = digit_ds.target[:50]

valid_data = digit_ds.images[50:500]
valid_id = digit_ds.target[50:500]
```

0.16.2 Run a KNeighbors regression

In this pipeline we will make a regression based on K-Neighbors.

```
from sklearn.neighbors import KNeighborsRegressor

digit_regress_pipe = Pipeline([('Flatten', flatten_step),
                               ('Normalize', RobustScaler()),
                               ('NearestNeighbor', KNeighborsRegressor(1))])
digit_regress_pipe.fit(img_data, digit_id)

show_pipe(digit_regress_pipe, img_data, panels_in_row=4)
```



0.17 Assessment of multi-category data

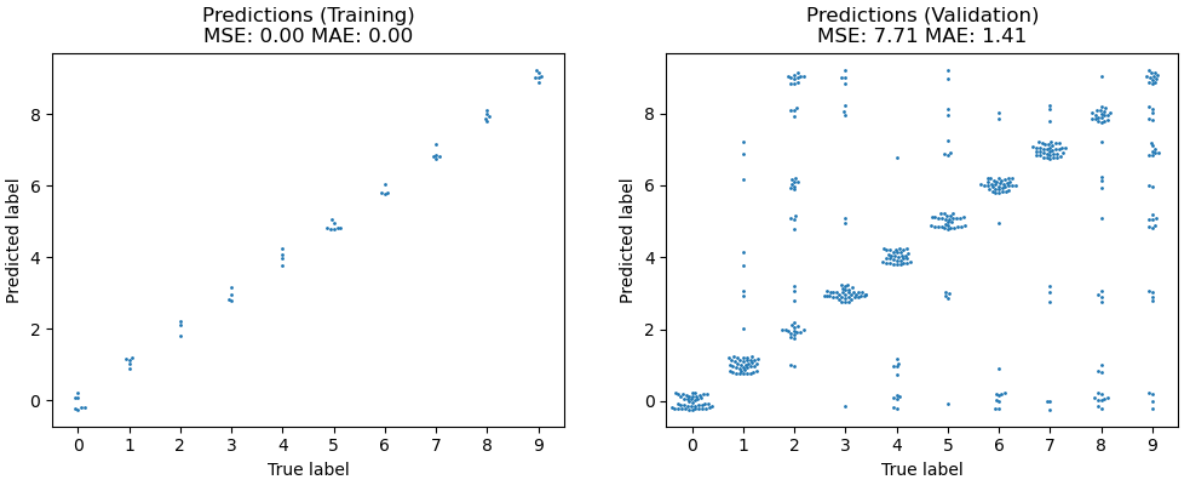
We can't use accuracy, ROC, precision, recall or any of these factors anymore since we don't have binary / true-or-false conditions we are trying to predict. We know have to go back to some of the initial metrics we covered in the first lectures.

$$MSE = \frac{1}{N}\sum\left(y_{predicted} - y_{actual}\right)^2$$

$$MAE = \frac{1}{N}\sum\left|y_{predicted} - y_{actual}\right|$$

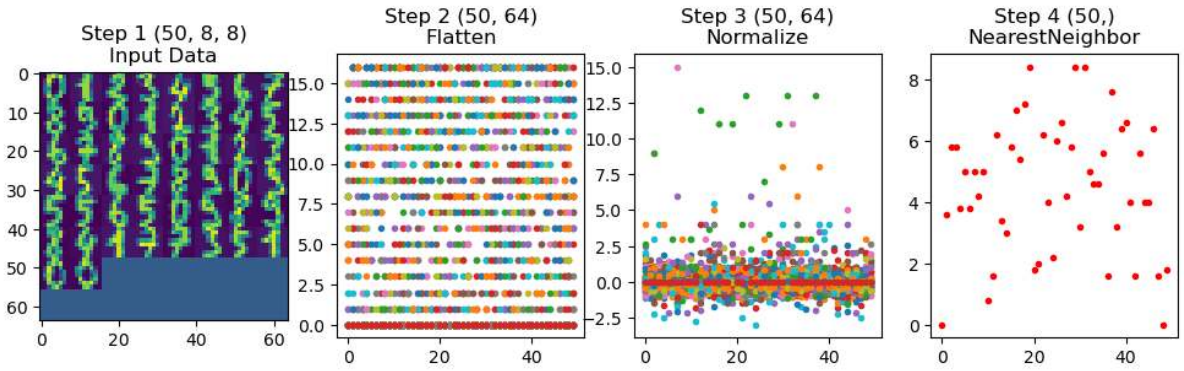
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4), dpi=100)
pred_train = digit_regress_pipe.predict(img_data)
jitter = lambda x: x+0.25*np.random.uniform(-1, 1, size=x.shape)
sns.swarmplot(x=digit_id, y=jitter(pred_train), ax=ax1,size=2)
ax1.set_title('Predictions (Training)\nMSE: %2.2f MAE: %2.2f' % (np.mean(np.
 ↪square(pred_train-digit_id)),
                                                                 np.mean(np.abs(pred_
 ↪train-digit_id))))
ax1.set_xlabel('True label'), ax1.set_ylabel('Predicted label')

pred_valid = digit_regress_pipe.predict(valid_data)
sns.swarmplot(x=valid_id, y=jitter(pred_valid), ax=ax2,size=2)
ax2.set_title('Predictions (Validation)\nMSE: %2.2f MAE: %2.2f' % (np.mean(np.
 ↪square(pred_valid-valid_id)),
                                                                   np.mean(np.
 ↪abs(pred_valid-valid_id))));
ax2.set_xlabel('True label'), ax2.set_ylabel('Predicted label');
```

```
/opt/anaconda3/envs/qbi2025/lib/python3.9/site-packages/seaborn/categorical.
 ↪py:3399: UserWarning: 6.1% of the points cannot be placed; you may want to␣
 ↪decrease the size of the markers or use stripplot.
  warnings.warn(msg, UserWarning)
```



0.17.1 Increasing neighbor count

```
digit_regress_pipe = Pipeline([('Flatten', flatten_step), ('Normalize',␣
 ↪RobustScaler()), ('NearestNeighbor', KNeighborsRegressor(5))])
digit_regress_pipe.fit(img_data, digit_id)

show_pipe(digit_regress_pipe, img_data, panels_in_row=4, figsize=[12,3])
```

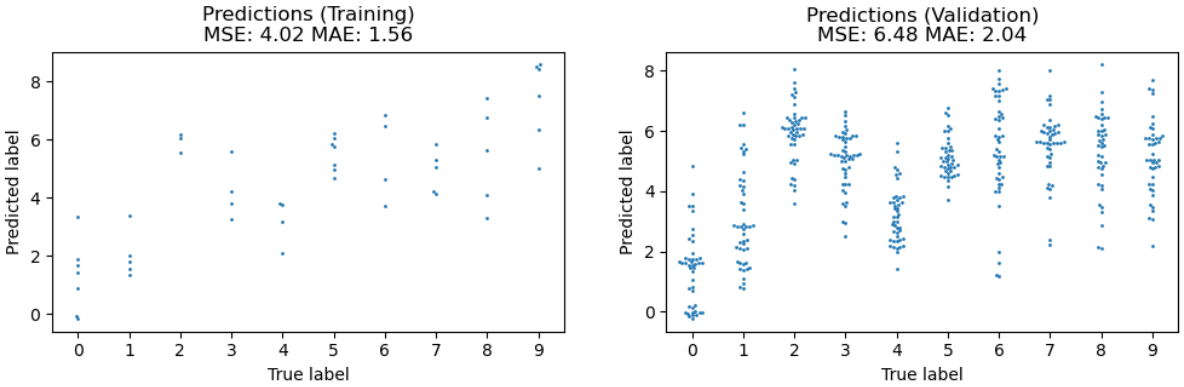



```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 3))
pred_train = digit_regress_pipe.predict(img_data)

sns.swarmplot(x=digit_id, y=jitter(pred_train), ax=ax1,size=2)
ax1.set_title('Predictions (Training)\nMSE: %2.2f MAE: %2.2f' % (np.mean(np.
 ↪square(pred_train-digit_id)),
                                                                 np.mean(np.abs(pred_
 ↪train-digit_id))))

ax1.set_xlabel('True label'), ax1.set_ylabel('Predicted label')
pred_valid = digit_regress_pipe.predict(valid_data)
sns.swarmplot(x=valid_id, y=jitter(pred_valid), ax=ax2,size=2)
ax2.set_title('Predictions (Validation)\nMSE: %2.2f MAE: %2.2f' % (np.mean(np.
 ↪square(pred_valid-valid_id)),
                                                                   np.mean(np.
```

(continues on next page)

(continued from previous page)

```
 ↪abs(pred_valid-valid_id))));
ax2.set_xlabel('True label'), ax2.set_ylabel('Predicted label');
```



0.18 Segmentation (Pixel Classification)

Previously Predict something based on the information in the image

- Single class (classification)
- Values (regression)

Segmentation Now we want to change problem:

- instead of assigning a single class for each image,
- we want a class or value for each pixel.

This requires that we restructure the problem.

0.18.1 Where segmentation fails: Mitochondria Segmentation in EM

- The mitocondria are visible and humans easily spot them
- Other structures have same gray levels
- SNR is not ideal

- A simple threshold is insufficient to finding the mitocondria structures
- Other filtering techniques are unlikely to magicially fix this problem

0.18.2 Let's try some methods to segment the mitochondria image

1. **Decision trees**
   - DecisionTreeRegressor
   - DecisionTreeRegressor with position
2. **Random forests**
   - Random forest + KMeans
   - Random forest + polynomials
   - Random forest + Filters
3. **Linear regression**
   - Neighborhood
4. **Nearest neighbor**
   - KNeighborsRegressor
5. **U-Net**

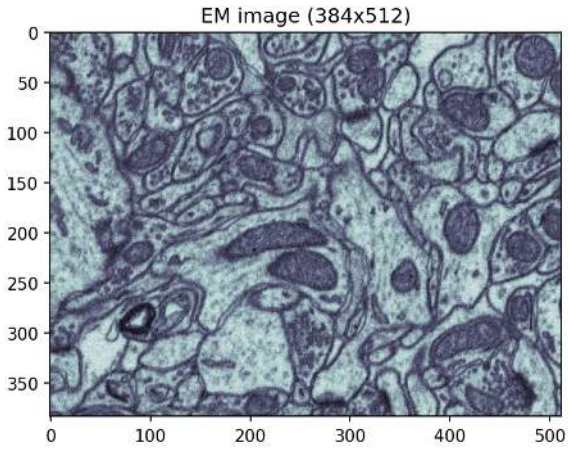
0.18.3 Preparing the mitchondria image and mask

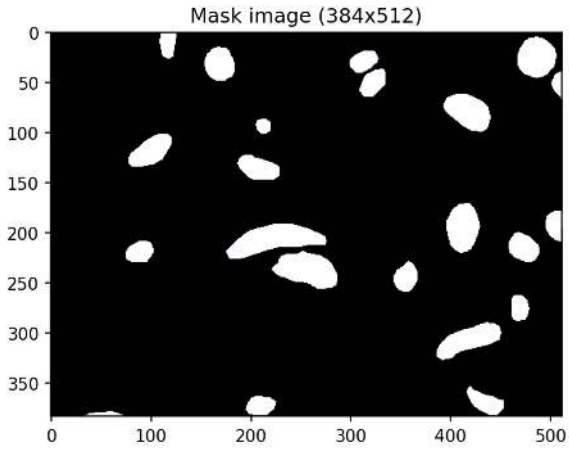
First we need image data for:

- Training
- Validation

```
cell_img = (imread("data/em_image.png")[::2, ::2])/255.0
cell_seg = imread("data/em_image_seg.png",
                  as_gray=True)[::2, ::2] > 0
np.random.seed(2018)
```

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 8), dpi=150)
ax1.imshow(cell_img, cmap='bone'); ax1.set_title("EM image ({0}x{1})".format(cell_img.
 ↪shape[0],cell_img.shape[1]))
ax2.imshow(cell_seg, cmap='bone'); ax2.set_title("Mask image ({0}x{1})".format(cell_
 ↪seg.shape[0],cell_seg.shape[1]));
```





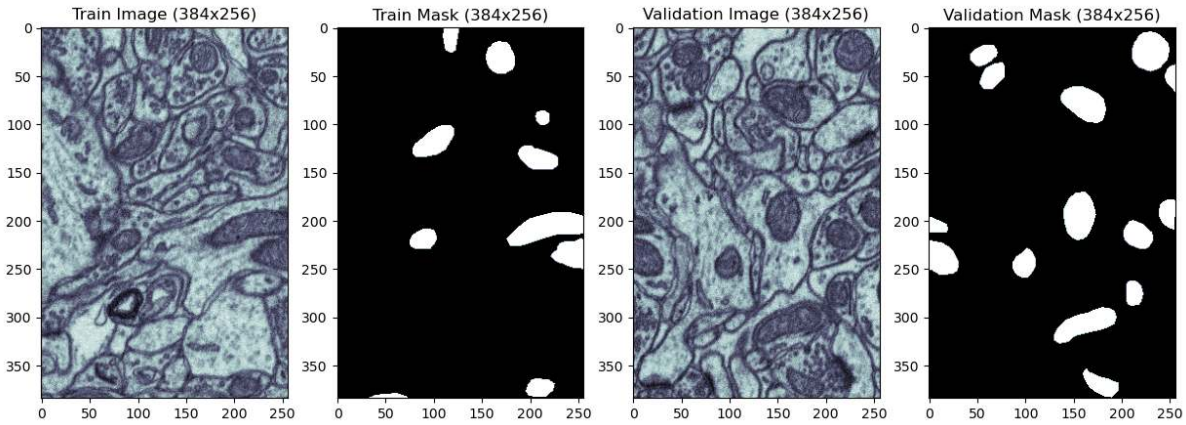
Training and validation data

We only have one image available…

Solution: Split it into two parts!

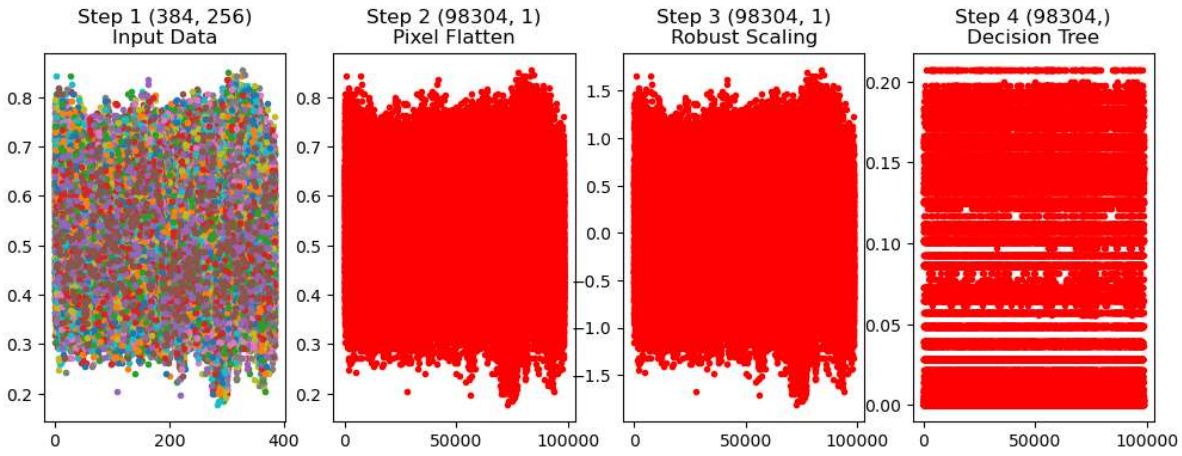
```
train_img, valid_img = cell_img[:, :256], cell_img[:, 256:]
train_mask, valid_mask = cell_seg[:, :256], cell_seg[:, 256:]
```

```
fig, ((ax1, ax2,ax3, ax4)) = plt.subplots(1, 4, figsize=(15, 5), dpi=100)
ax1.imshow(train_img, cmap='bone');  ax1.set_title('Train Image ({0}x{1})'.
 ↪format(train_img.shape[0],train_img.shape[1]))
ax2.imshow(train_mask, cmap='bone'); ax2.set_title('Train Mask ({0}x{1})'.
 ↪format(train_mask.shape[0],train_mask.shape[1]))

ax3.imshow(valid_img,  cmap='bone'); ax3.set_title('Validation Image ({0}x{1})'.
 ↪format(valid_img.shape[0],valid_img.shape[1]))
ax4.imshow(valid_mask, cmap='bone'); ax4.set_title('Validation Mask ({0}x{1})'.
 ↪format(valid_mask.shape[0],valid_mask.shape[1]));
```



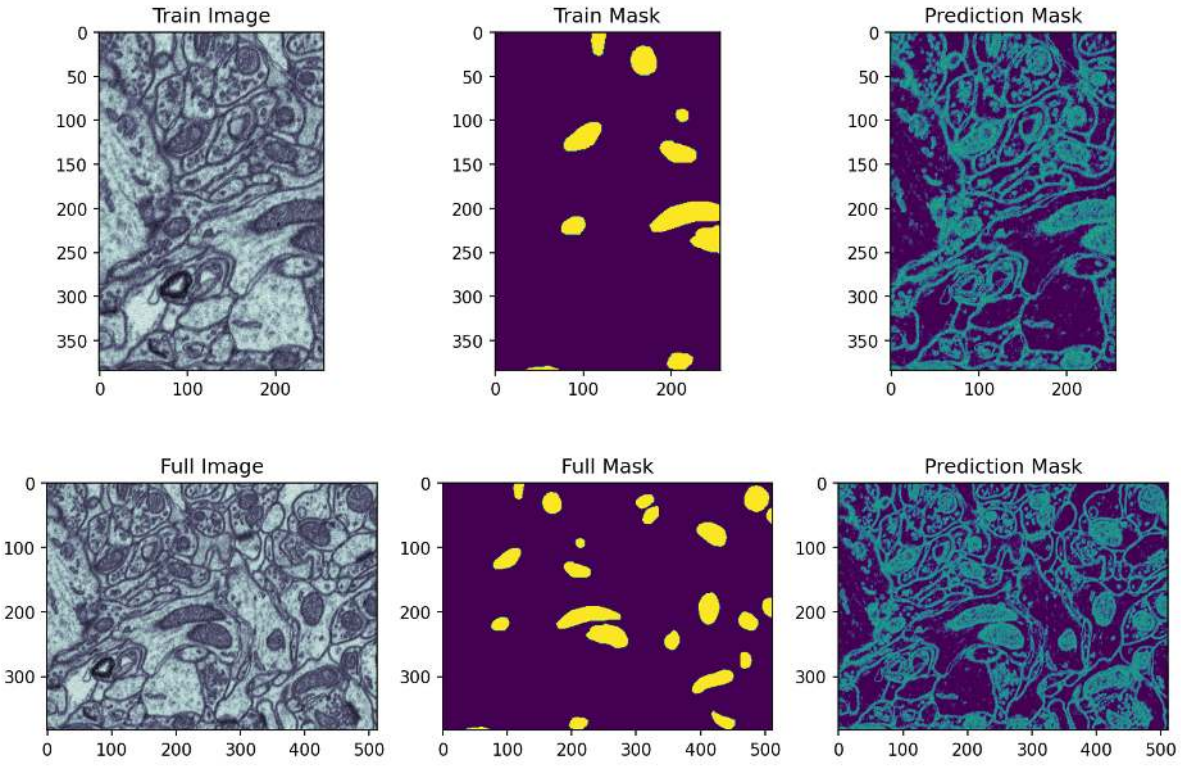
0.18.4 Try a Regression tree

```
from pipe_utils import px_flatten_func, fit_img_pipe
px_flatten_step = FunctionTransformer(px_flatten_func, validate=False)
rf_seg_model = Pipeline([('Pixel Flatten',  px_flatten_step),
                         ('Robust Scaling', RobustScaler()),
                         ('Decision Tree',  DecisionTreeRegressor())
                         ])

pred_func = fit_img_pipe(rf_seg_model, train_img, train_mask)
show_pipe(rf_seg_model, train_img,panels_in_row=4)
show_tree(rf_seg_model.steps[-1][1]);
```



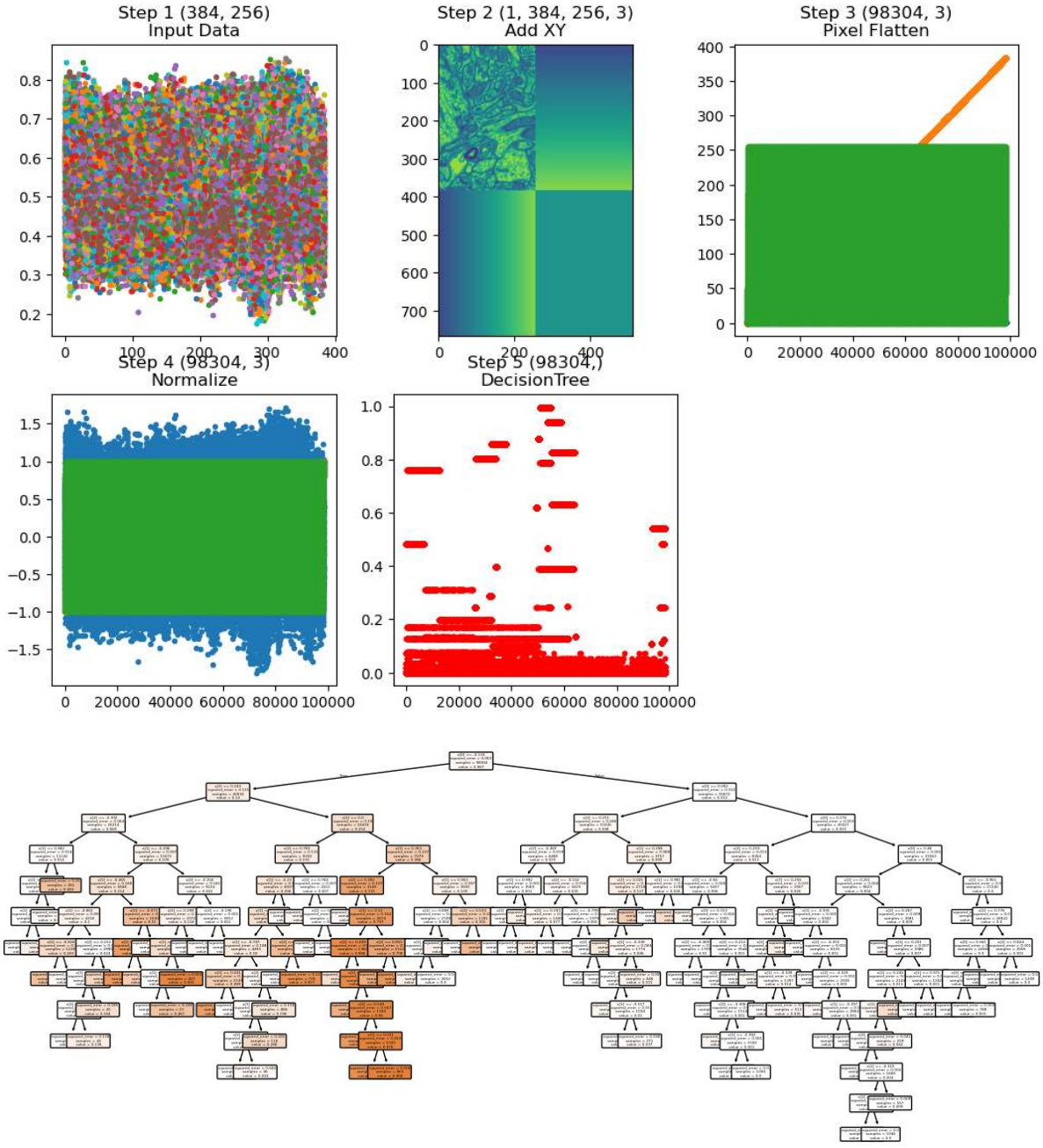
Segmentation results from the regression tree

```
fig, ((ax1, ax5, ax2), (ax3, ax6, ax4)) = plt.subplots(2, 3, figsize=(12, 8), dpi=150)
ax1.imshow(train_img, cmap='bone')
ax1.set_title('Train Image')

ax5.imshow(train_mask, cmap='viridis'); ax5.set_title('Train Mask')

ax2.imshow(pred_func(train_img)[:, :, 0], cmap='viridis', vmin=0, vmax=0.3); ax2.set_
↪title('Prediction Mask')

ax3.imshow(cell_img, cmap='bone'); ax3.set_title('Full Image')

ax6.imshow(cell_seg, cmap='viridis'); ax6.set_title('Full Mask')

ax4.imshow(pred_func(cell_img)[:, :, 0], cmap='viridis', vmin=0, vmax=0.3); ax4.set_
↪title('Prediction Mask');
```



0.18.5 Regression tree with *position information*

```
from pipe_utils import xy_step

rf_xyseg_model = Pipeline([('Add XY', xy_step),
                           ('Pixel Flatten', px_flatten_step),
                           ('Normalize', RobustScaler()),
                           ('DecisionTree', DecisionTreeRegressor(
                               min_samples_split=1000))
                           ])

pred_func = fit_img_pipe(rf_xyseg_model, train_img, train_mask)
show_pipe(rf_xyseg_model, train_img)
plt.figure(figsize=(15,6))
tree.plot_tree(rf_xyseg_model.steps[-1][1],fontsize=3,filled=True,rounded=True);
```



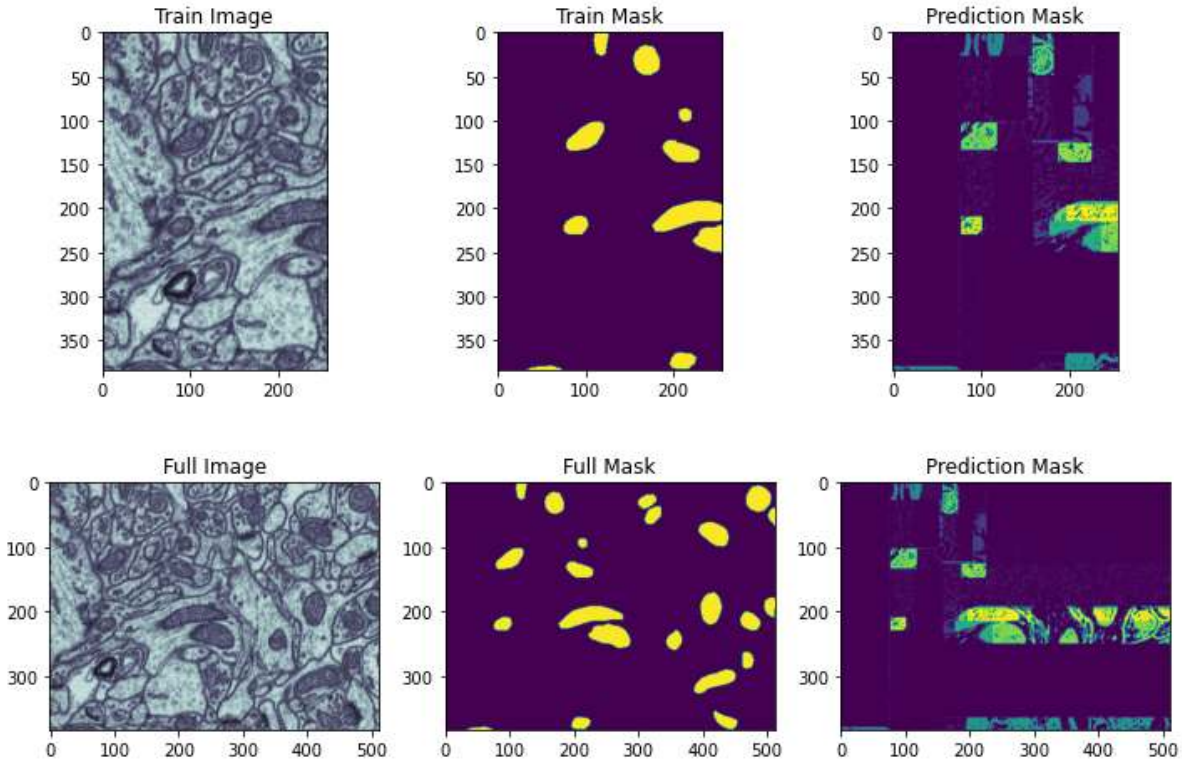
Did the segmentation performance improve?

```
fig, ((ax1, ax5, ax2), (ax3, ax6, ax4)) = plt.subplots(2, 3, figsize=(12, 8), dpi=72)
ax1.imshow(train_img, cmap='bone');ax1.set_title('Train Image')
ax5.imshow(train_mask, cmap='viridis'); ax5.set_title('Train Mask')
ax2.imshow(pred_func(train_img)[:, :, 0], cmap='viridis', vmin=0, vmax=1); ax2.set_
↪title('Prediction Mask')
ax3.imshow(cell_img, cmap='bone'); ax3.set_title('Full Image')
ax6.imshow(cell_seg, cmap='viridis'); ax6.set_title('Full Mask')
ax4.imshow(pred_func(cell_img)[:, :, 0], cmap='viridis', vmin=0, vmax=1);ax4.set_
```

(continues on next page)

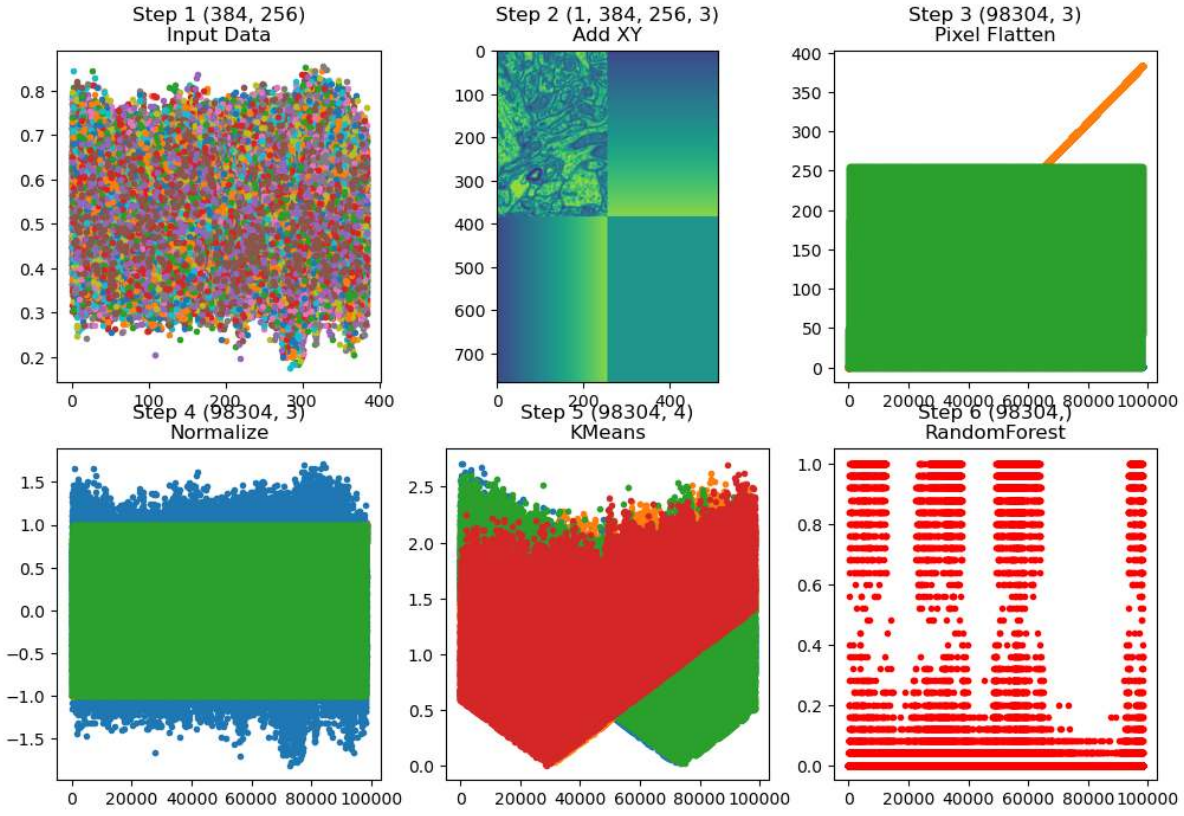
(continued from previous page)

```
↪title('Prediction Mask');
```



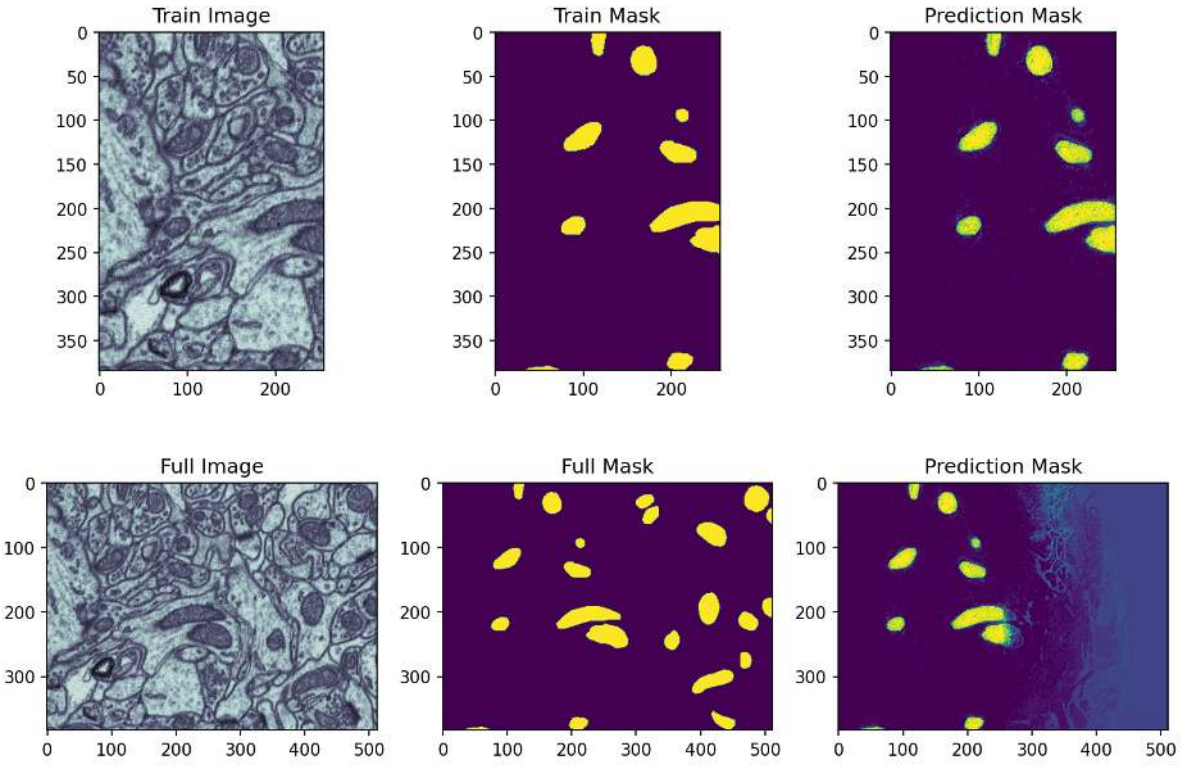
0.18.6 Combine K-Means and Random Forest Regression

```
from sklearn.cluster import KMeans
rf_xyseg_k_model = Pipeline([('Add XY', xy_step),
                             ('Pixel Flatten', px_flatten_step),
                             ('Normalize', RobustScaler()),
                             ('KMeans', KMeans(4,n_init='auto')),
                             ('RandomForest', RandomForestRegressor(n_estimators=25))
                             ])

pred_func = fit_img_pipe(rf_xyseg_k_model, train_img, train_mask)
show_pipe(rf_xyseg_k_model, train_img)
```



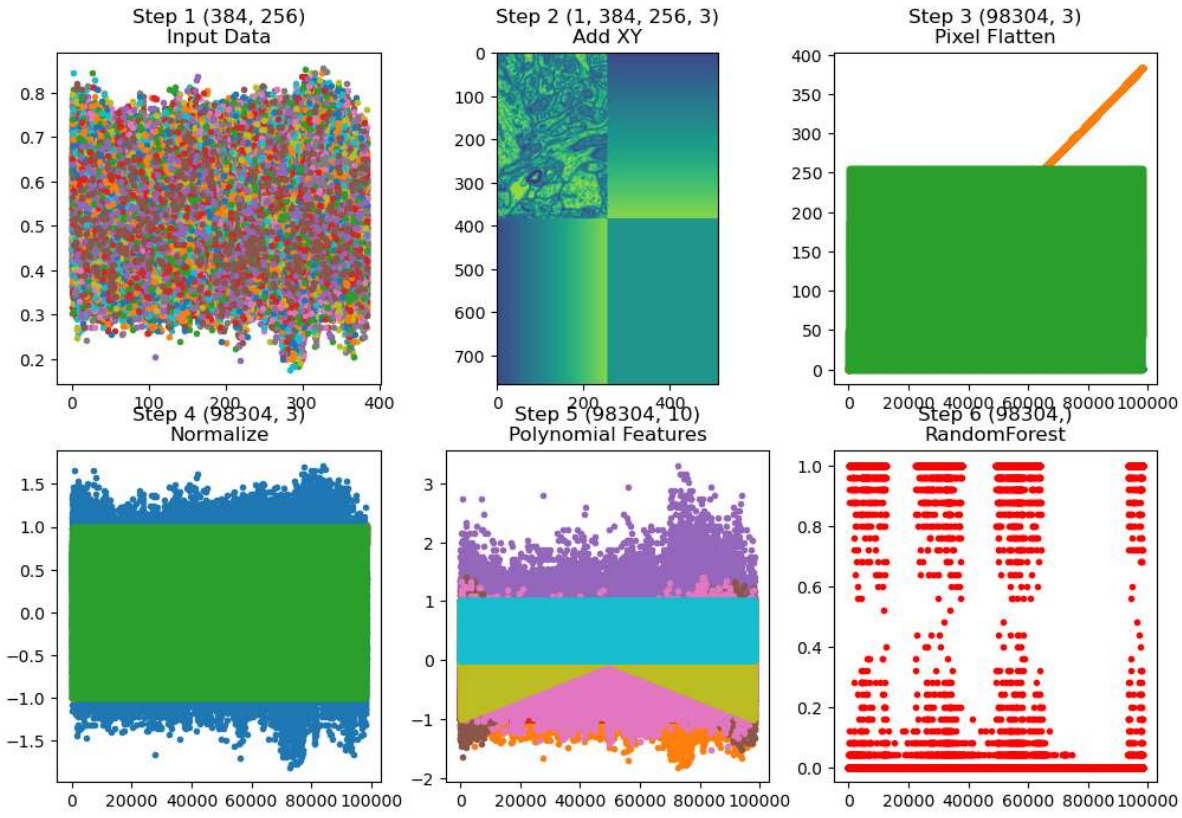
Did it improve now?

```
fig, ((ax1, ax5, ax2), (ax3, ax6, ax4)) = plt.subplots(2, 3, figsize=(12, 8), dpi=150)
ax1.imshow(train_img, cmap='bone'); ax1.set_title('Train Image')
ax5.imshow(train_mask, cmap='viridis'); ax5.set_title('Train Mask')
ax2.imshow(pred_func(train_img)[:, :, 0], cmap='viridis', vmin=0, vmax=1); ax2.set_
↪title('Prediction Mask')
ax3.imshow(cell_img, cmap='bone'); ax3.set_title('Full Image')
ax6.imshow(cell_seg, cmap='viridis'); ax6.set_title('Full Mask')
ax4.imshow(pred_func(cell_img)[:, :, 0], cmap='viridis', vmin=0, vmax=1); ax4.set_
↪title('Prediction Mask');
```



0.18.7 Trying polynomial features

```
from sklearn.preprocessing import PolynomialFeatures
from pipe_utils import add_xy_coord
xy_step = FunctionTransformer(add_xy_coord, validate=False)
rf_xyseg_py_model = Pipeline([('Add XY', xy_step),
                              ('Pixel Flatten', px_flatten_step),
                              ('Normalize', RobustScaler()),
                              ('Polynomial Features', PolynomialFeatures(2)),
                              ('RandomForest', RandomForestRegressor(n_estimators=25))
                              ])

pred_func = fit_img_pipe(rf_xyseg_py_model, train_img, train_mask)
show_pipe(rf_xyseg_py_model, train_img)
```

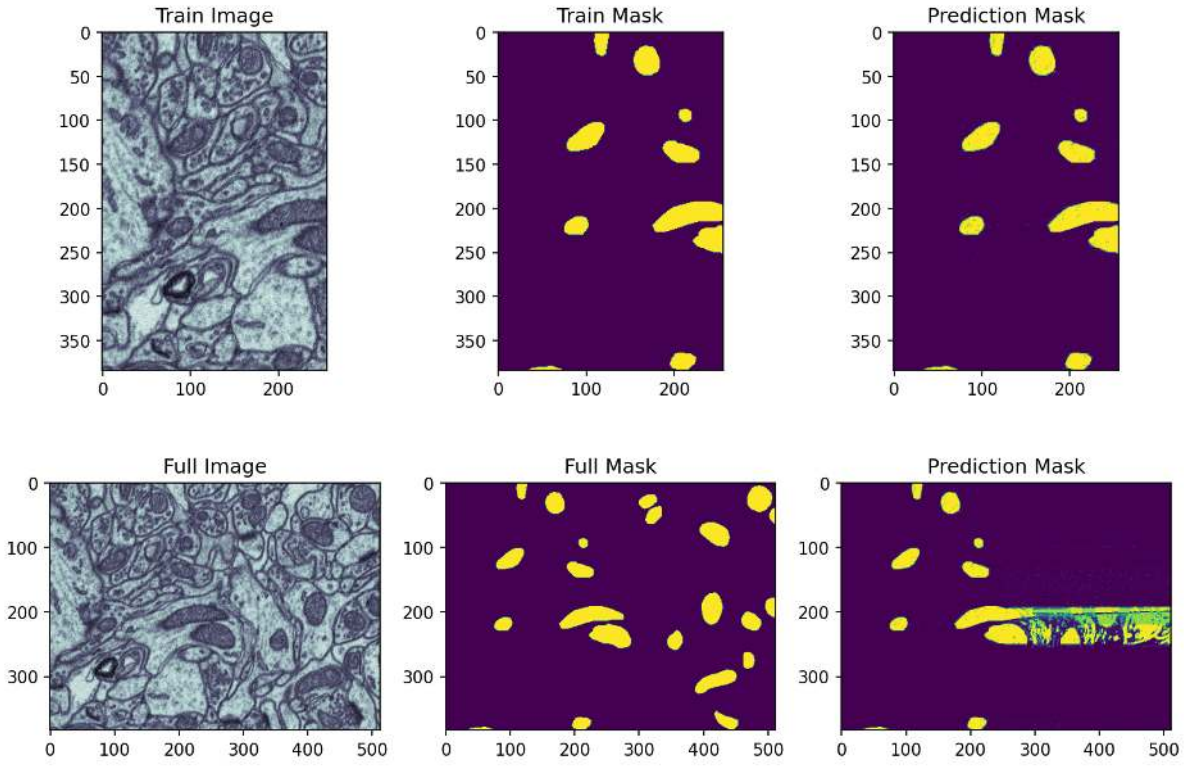


What happens with this complicated pipeline?

We…

- Added XY position
- Normalized
- Added polynomial features
- Used a random forest regressor with 25 trees

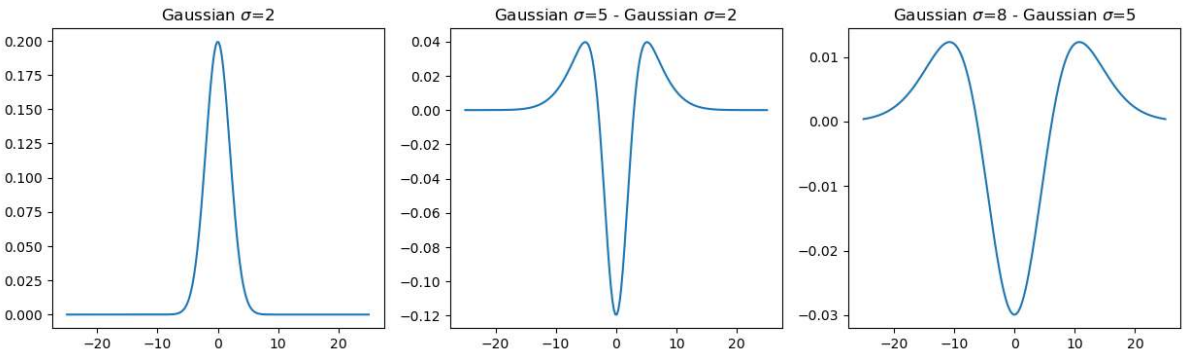
```
fig, ((ax1, ax5, ax2), (ax3, ax6, ax4)) = plt.subplots(2, 3, figsize=(12, 8), dpi=150)
ax1.imshow(train_img, cmap='bone')
ax1.set_title('Train Image')

ax5.imshow(train_mask, cmap='viridis'); ax5.set_title('Train Mask')

ax2.imshow(pred_func(train_img)[:, :, 0], cmap='viridis', vmin=0, vmax=1); ax2.set_
↪title('Prediction Mask')

ax3.imshow(cell_img, cmap='bone'); ax3.set_title('Full Image')

ax6.imshow(cell_seg, cmap='viridis'); ax6.set_title('Full Mask')

ax4.imshow(pred_func(cell_img)[:, :, 0], cmap='viridis', vmin=0, vmax=1); ax4.set_
↪title('Prediction Mask');
```



0.18.8 Adding smarter features

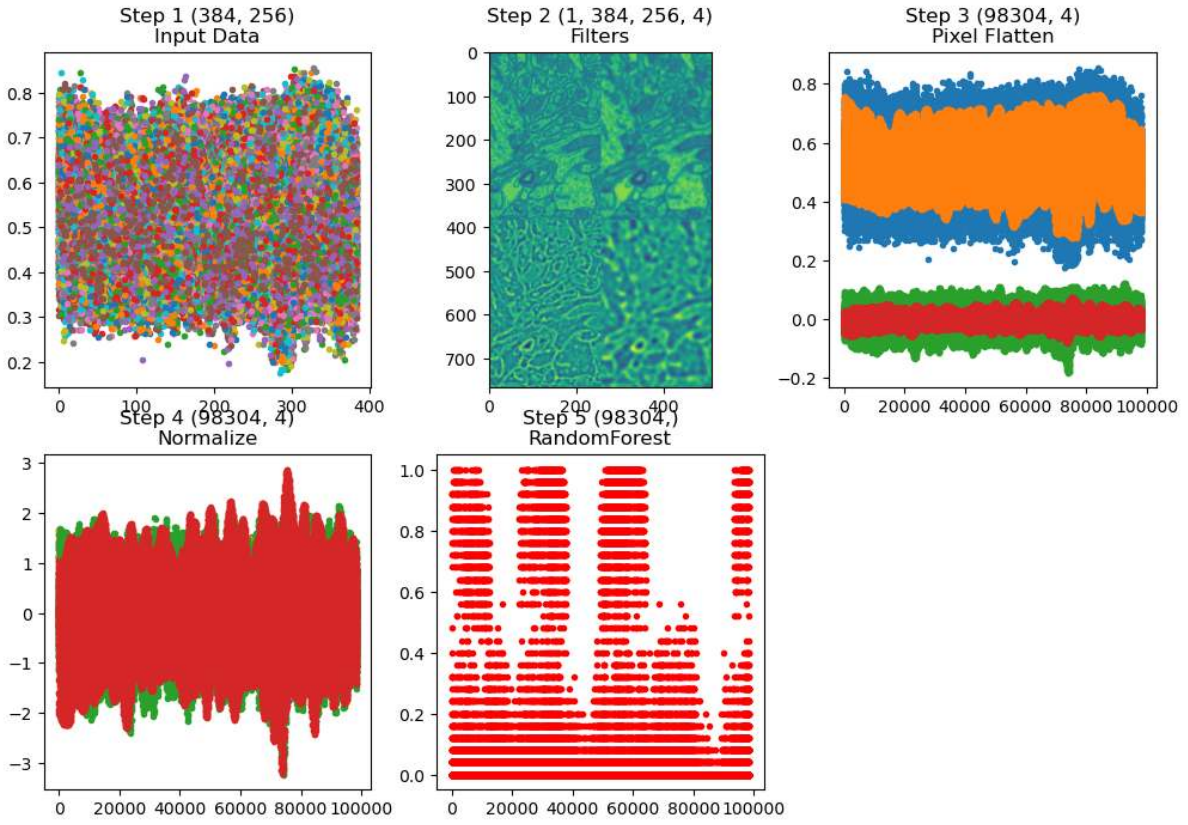
Here we add images with weighted neighborhood information using filters based on Gaussians

```
import scipy.stats as stats
x=np.linspace(-25,25,200)
fig,(ax1,ax2,ax3) = plt.subplots(1,3,figsize=[15,4])
ax1.plot(x,stats.norm.pdf(x,0,2)); ax1.set_title("Gaussian $\sigma$=2");
ax2.plot(x,stats.norm.pdf(x,0,5)-stats.norm.pdf(x,0,2)), ax2.set_title("Gaussian $\
↪sigma$=5 - Gaussian $\sigma$=2");
ax3.plot(x,stats.norm.pdf(x,0,8)-stats.norm.pdf(x,0,5)), ax3.set_title("Gaussian $\
↪sigma$=8 - Gaussian $\sigma$=5");
```



Pipeline with filters

```
from pipe_utils import filter_step
rf_filterseg_model = Pipeline([('Filters', filter_step),
                               ('Pixel Flatten', px_flatten_step),
                               ('Normalize', RobustScaler()),
                               ('RandomForest', RandomForestRegressor(n_
 ↪estimators=25))
                               ])

pred_func = fit_img_pipe(rf_filterseg_model, train_img, train_mask)
show_pipe(rf_filterseg_model, train_img)
```



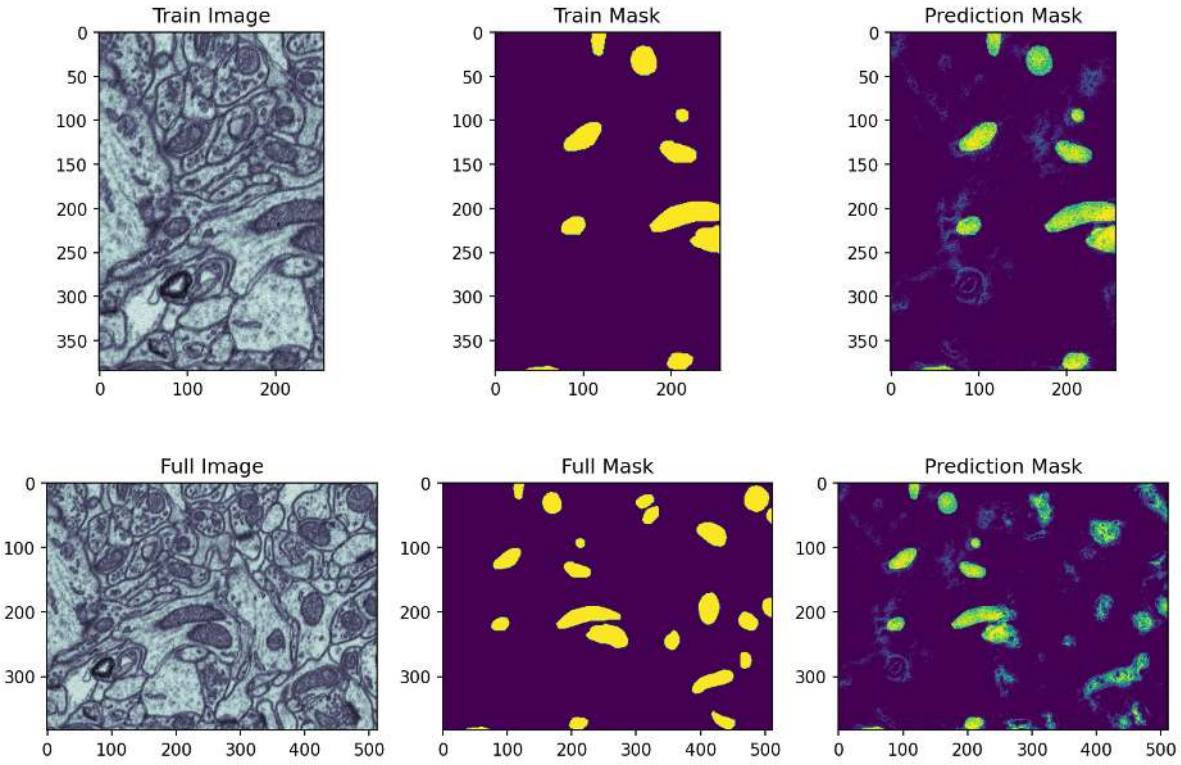
Results with features based on filters

```
fig, ((ax1, ax5, ax2), (ax3, ax6, ax4)) = plt.subplots( 2, 3, figsize=(12, 8),␣
 ↪dpi=150)
ax1.imshow(train_img, cmap='bone'); ax1.set_title('Train Image')

ax5.imshow(train_mask, cmap='viridis'); ax5.set_title('Train Mask')

ax2.imshow(pred_func(train_img)[:, :, 0], cmap='viridis', vmin=0, vmax=1); ax2.set_
 ↪title('Prediction Mask')

ax3.imshow(cell_img, cmap='bone'); ax3.set_title('Full Image')
```

(continues on next page)

(continued from previous page)

```
ax6.imshow(cell_seg, cmap='viridis'); ax6.set_title('Full Mask')

ax4.imshow(pred_func(cell_img)[:, :, 0], cmap='viridis', vmin=0, vmax=1); ax4.set_
↪title('Prediction Mask');
```



0.18.9 Using the Neighborhood

We can also include the whole neighborhood

- shifting the image in x and y by ± 1 pixel.
- Gives nine feature images

For the first example we will then use **linear regression** so we can see the exact coefficients that result.

Some code to create the neighborhood

```
from sklearn.preprocessing import FunctionTransformer

def add_neighborhood(in_x, x_steps=3, y_steps=3):
    if len(in_x.shape) == 2:   x = np.expand_dims(np.expand_dims(in_x, 0), -1)
    elif len(in_x.shape) == 3: x = np.expand_dims(in_x, -1)
    elif len(in_x.shape) == 4: x = in_x
    else:
        raise ValueError('Cannot work with images with dimensions {}'.format(in_x.
↪shape))
```

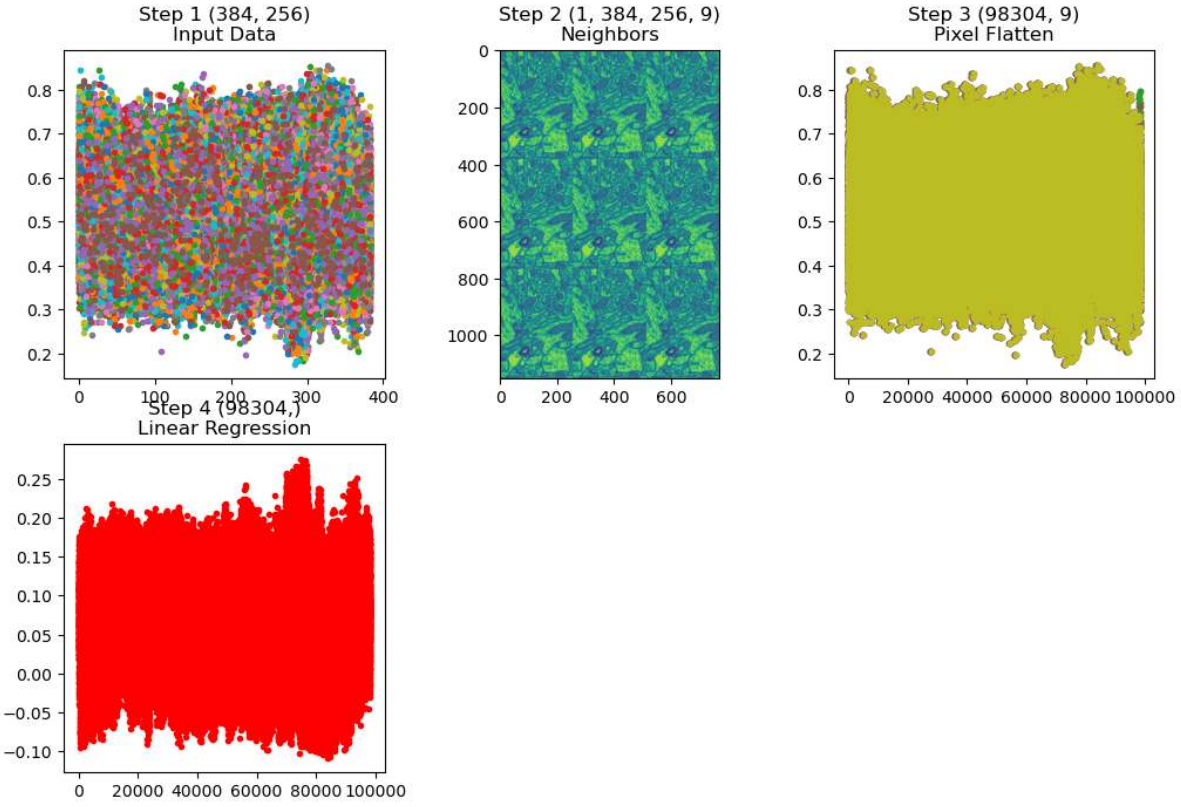
(continues on next page)

(continued from previous page)

```
    n_img, x_dim, y_dim, c_dim = x.shape
    out_imgs = []
    for i in range(-x_steps, x_steps+1):
        for j in range(-y_steps, y_steps+1):
            out_imgs += [np.roll(np.roll(x,axis=1, shift=i), axis=2, shift=j)]
    return np.concatenate(out_imgs, -1)

def neighbor_step(x_steps=3, y_steps=3):
    return FunctionTransformer(
        lambda x: add_neighborhood(x, x_steps, y_steps),
        validate=False)
```

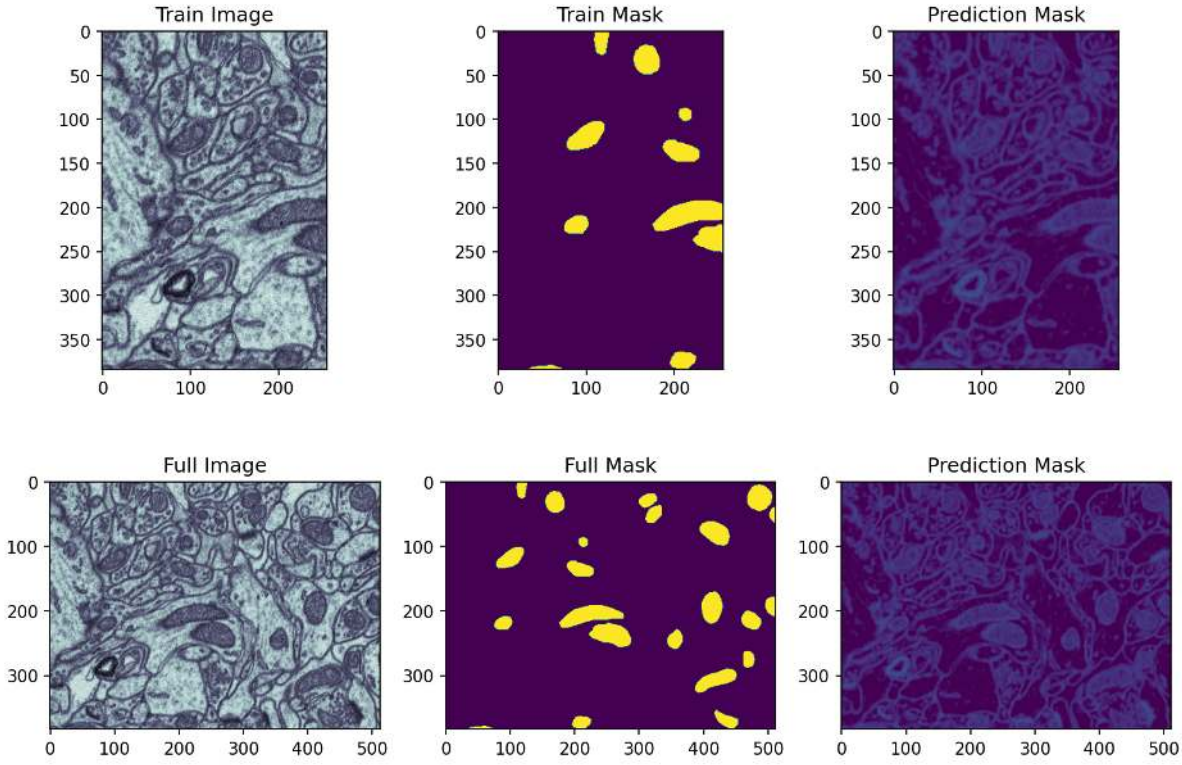
Pipeline with neighborhood features

```
from sklearn.linear_model import LinearRegression
linreg_neighborseg_model = Pipeline([('Neighbors', neighbor_step(1, 1)),
                                     ('Pixel Flatten', px_flatten_step),
                                     ('Linear Regression', LinearRegression())
                                     ])

pred_func = fit_img_pipe(linreg_neighborseg_model, train_img, train_mask)
show_pipe(linreg_neighborseg_model, train_img)
```



Result of neighborhood regression

```
fig, ((ax1, ax5, ax2), (ax3, ax6, ax4)) = plt.subplots(2, 3, figsize=(12, 8), dpi=150)
ax1.imshow(train_img, cmap='bone') ;ax1.set_title('Train Image')

ax5.imshow(train_mask, cmap='viridis'); ax5.set_title('Train Mask')

ax2.imshow(pred_func(train_img)[:, :, 0], cmap='viridis', vmin=0, vmax=1); ax2.set_
↪title('Prediction Mask')

ax3.imshow(cell_img, cmap='bone') ; ax3.set_title('Full Image')

ax6.imshow(cell_seg, cmap='viridis') ; ax6.set_title('Full Mask')

ax4.imshow(pred_func(cell_img)[:, :, 0], cmap='viridis', vmin=0, vmax=1); ax4.set_
↪title('Prediction Mask');
```



Why Linear Regression?

We choose linear regression so we could get easily understood coefficients.

The model fits $\vec{m}$ and b to the $\vec{x}_{i,j}$ points in the image $I(i,j)$ to match the $y_{i,j}$ output in the segmentation as closely as possible $$y_{i,j} = \vec{m} \cdot \vec{x_{i,j}} + b For a 3x3 case this looks like \vec{x}_{i,j} = [I(i-1,j-1), I(i-1,j), I(i-1,j+1) ... I(i+1,j-1), I(i+1,j), I(i+1,j+1)]$$

```
m = linreg_neighborseg_model.steps[-1][1].coef_
b = linreg_neighborseg_model.steps[-1][1].intercept_
print('M: [{:0.4}, {:0.4}, {:0.4}, {:0.4}, \033[1m{:0.4}\033[0m, {:0.4}, {:0.4}, {:0.
```

(continues on next page)

(continued from previous page)

```
 ↪4}, {:0.4}]'.format(m[0],m[1],m[2],m[3],m[4],m[5],m[6],m[7],m[8]))
print('b: {:0.4}'.format(b))
```

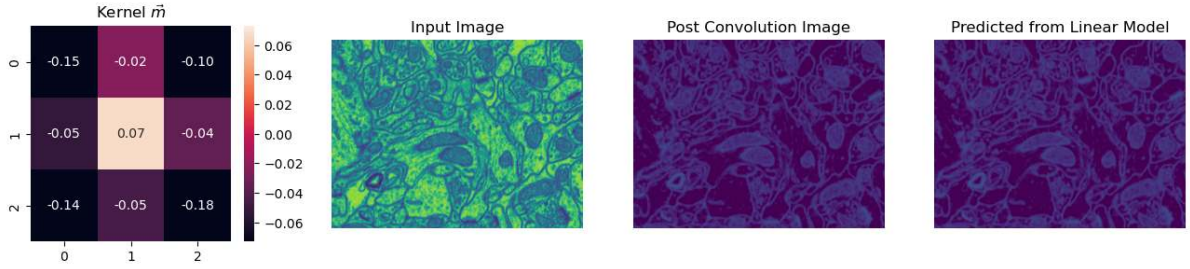
```
M: [-0.1501, -0.05296, -0.1412, -0.02355, 0.06657, -0.04512, -0.1015, -0.03607, -0.
 ↪1819]
b: 0.4213
```

Convolution

The steps we have here make up a convolution.

- What we have effectively done is use linear regression
- to learn which coefficients we should use in a convolutional kernel to get the best results

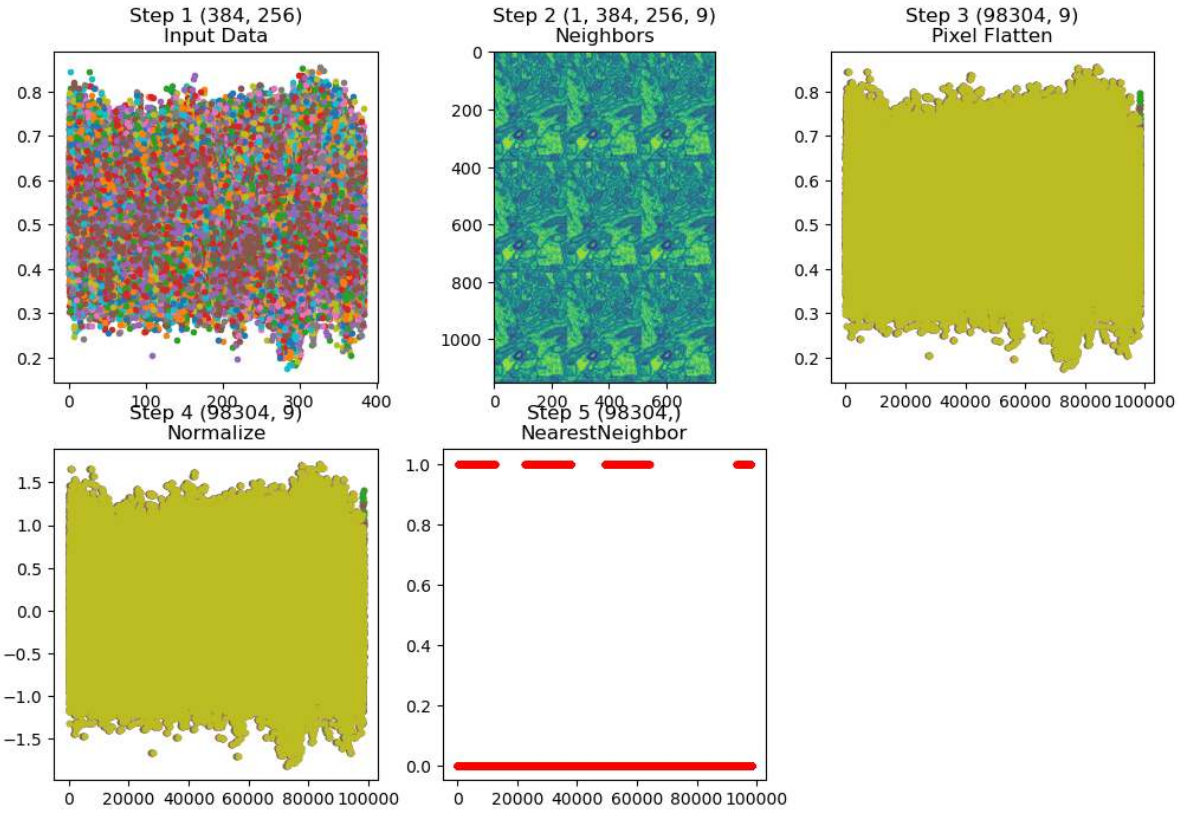
```
from scipy.ndimage import convolve
m_mat = m.reshape((3, 3)).T
fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(16, 3))
sns.heatmap(m_mat,
            annot=True,
            ax=ax1, fmt='2.2f',
            vmin=-m_mat.std(),
            vmax=m_mat.std())
ax1.set_title(r'Kernel $\vec{m}$')
ax2.imshow(cell_img)
ax2.set_title('Input Image')
ax2.axis('off')

ax3.imshow(convolve(cell_img, m_mat)+b,
           vmin=0,
           vmax=1,
           cmap='viridis')
ax3.set_title('Post Convolution Image')
ax3.axis('off')

ax4.imshow(pred_func(cell_img)[:, :, 0],
           cmap='viridis', vmin=0, vmax=1)
ax4.set_title('Predicted from Linear Model')
ax4.axis('off');
```



0.18.10 Nearest Neighbor

We can also use the neighborhood and nearest neighbor, this means for each pixel and its surrounds we find the pixel in the training set that looks most similar

```
nn_neighborseg_model = Pipeline([('Neighbors', neighbor_step(1, 1)),
                                 ('Pixel Flatten', px_flatten_step),
                                 ('Normalize', RobustScaler()),
                                 ('NearestNeighbor', KNeighborsRegressor(n_neighbors=1))
                                 ])

pred_func = fit_img_pipe(nn_neighborseg_model, train_img, train_mask)
show_pipe(nn_neighborseg_model, train_img)
```



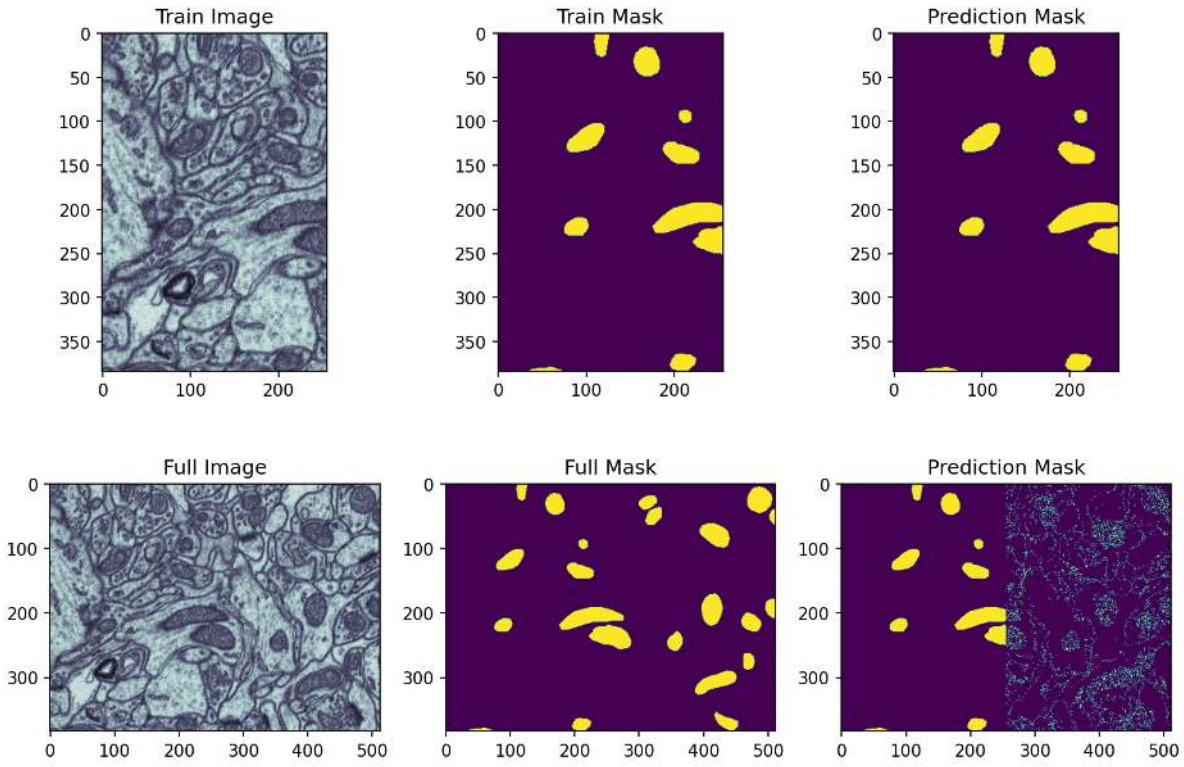
Results of the nearest neighbor

```
fig, ((ax1, ax5, ax2), (ax3, ax6, ax4)) = plt.subplots(
    2, 3, figsize=(12, 8), dpi=150)
ax1.imshow(train_img, cmap='bone');ax1.set_title('Train Image')

ax5.imshow(train_mask, cmap='viridis'); ax5.set_title('Train Mask')

ax2.imshow(pred_func(train_img)[:, :, 0], cmap='viridis', vmin=0, vmax=1); ax2.set_
↪title('Prediction Mask')

ax3.imshow(cell_img, cmap='bone'); ax3.set_title('Full Image')
```

(continues on next page)

(continued from previous page)

```
ax6.imshow(cell_seg, cmap='viridis'); ax6.set_title('Full Mask')

ax4.imshow(pred_func(cell_img)[:, :, 0], cmap='viridis', vmin=0, vmax=1)
ax4.set_title('Prediction Mask');
```



0.18.11 Summarizing the pipeline segmentations

- We have seen that a pipeline can be efficiently used for segmenation tasks.
  - The pipeline is easy to configure
  - It is easy to read
- Adding generated features can help improving the performance
- None of the method performed convincingly
  - Some failed on validation data
  - Other failed on all data, including training data!

  There are more pipeline examples in the lecture notes

0.19 Deep learning and convolutional networks

0.19.1 The basic neural network

A basic neural network is based on the concept of combining the weighted sums of non-linear activation functions in layers. The lines in the figure correspond to the weights and the circles are the activation functions.

Typincal activation functions are the sigmoid and ReLU functions.

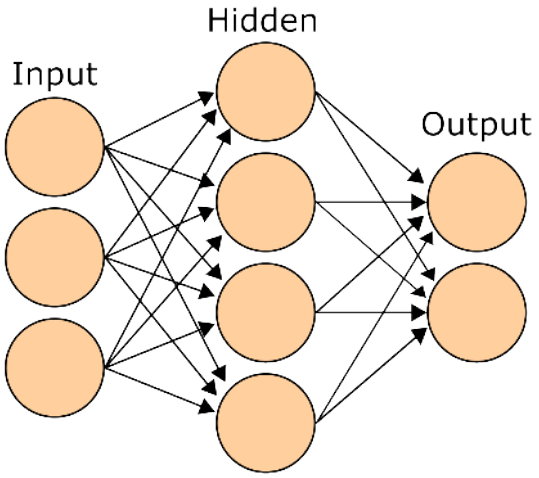


Fig. 1: Random forests train trees on different fractions of the data.

The network is used in two phases

1. Training: The weights are tuned by providing pairs of input and output data. The process is governed by an optimizer. This is also called back-propagation.
2. Prediction: The outcome probability for an input is computeed using forward propagation through the network.

These networks were introduced in the 90's but they had trouble with large complex information like images. They were also rather shallow at that time using only few layers.

The big breakthrough came with the deep models that also included convolution kernels in the nodes. The layers grew and also the number of layers. Which brought the convolutional neural networks and deep learning. This evolution was also leveraged by the availabiltiy of graphics cards as computational boost as these deep learning models are extremely computational intense.

0.19.2 Deep learning with a U-Net

The last approach we will briefly cover is the idea of U-Net a landmark paper from 2015 that dominates MICCAI submissions and contest winners today. A nice overview of the techniques is presented by Vladimir Iglovikov a winner of a recent Kaggle competition on masking images of cars slides

U-Net Diagram

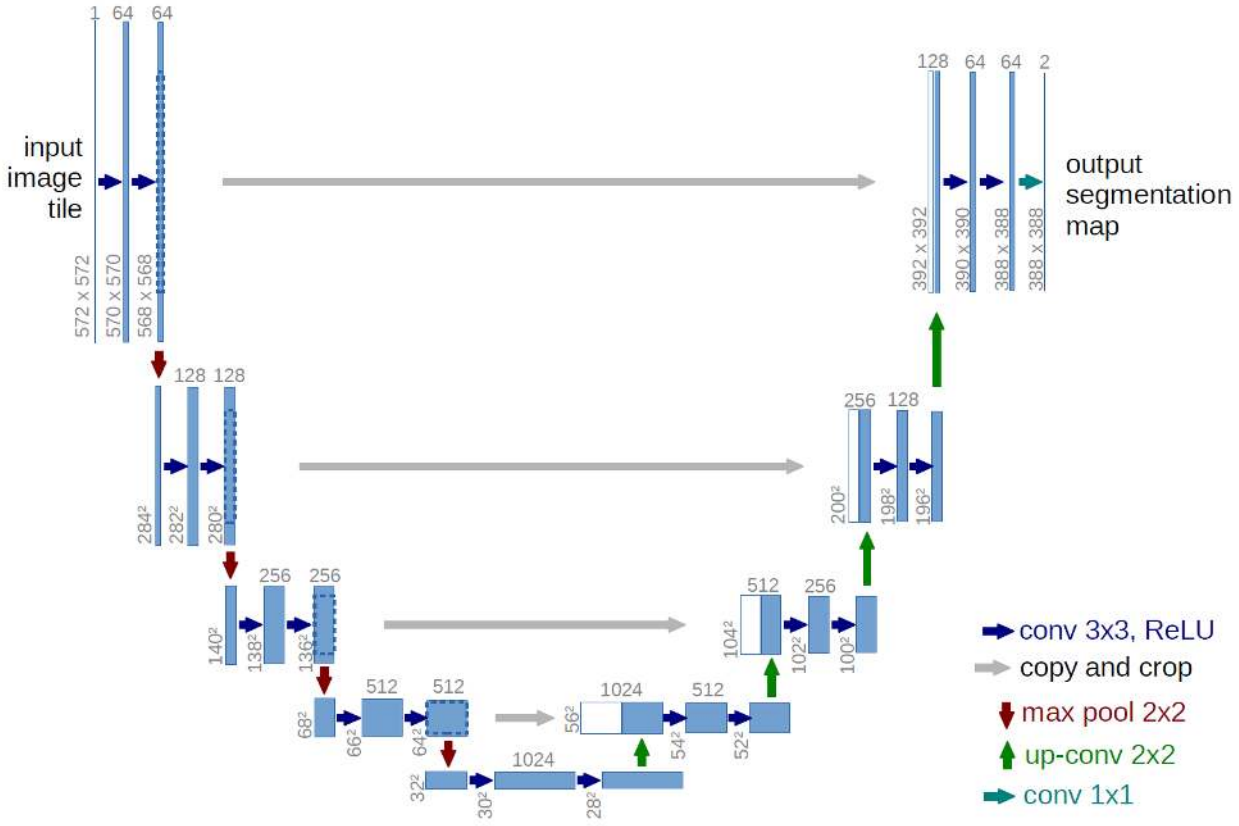


Fig. 2: Random forests train trees on different fractions of the data.

Let's build a small U-Net

```
# import numpy as np
# import skimage.io as io
# import matplotlib.pyplot as plt
# import tensorflow as tf
# from tensorflow.keras.models import Model
# from tensorflow.keras.models import Sequential
# from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D,␣
 ↪concatenate
# import tensorflow.keras.losses as losses
# from IPython.display import SVG
# # from keras.utils.vis_utils import model_to_dot
# from tensorflow.keras.utils import model_to_dot
# base_depth = 32

# t_unet = Sequential([
#     Conv2D(base_depth, kernel_size=(3, 3),   padding='same', activation='relu',␣
 ↪input_shape=(None, None, 1)),
#     Conv2D(base_depth, kernel_size=(3, 3),   padding='same', activation='relu'),
#     MaxPooling2D((2, 2)),
#     Conv2D(base_depth*2, kernel_size=(3, 3), padding='same', activation='relu'),
#     Conv2D(base_depth*2, kernel_size=(3, 3), padding='same', activation='relu'),
#     MaxPooling2D((2, 2)),
#     Conv2D(base_depth*4, kernel_size=(3, 3), padding='same', activation='relu'),
#     Conv2D(base_depth*4, kernel_size=(3, 3), padding='same', activation='relu'),
#     UpSampling2D((2, 2)),
#     Conv2D(base_depth*2, kernel_size=(3, 3), padding='same', activation='relu'),
#     Conv2D(base_depth*2, kernel_size=(3, 3), padding='same', activation='relu'),
#     UpSampling2D((2, 2)),
#     Conv2D(base_depth, kernel_size=(3, 3), padding='same', activation='relu'),
#     Conv2D(base_depth, kernel_size=(3, 3), padding='same', activation='relu'),
#     Conv2D(         1, kernel_size=(1, 1), padding='same', activation='sigmoid')
# ],name='SmallCNN')
```

```
import numpy as np
import skimage.io as io
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D,␣
 ↪concatenate
import tensorflow.keras.losses as losses

inputs = Input((None, None, 1))
base_depth = 64

conv1 = Conv2D(base_depth, (3, 3), activation='relu', padding='same')(inputs)
conv1 = Conv2D(base_depth, (3, 3), activation='relu', padding='same')(conv1)
pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

conv2 = Conv2D(base_depth*2, (3, 3), activation='relu', padding='same')(pool1)
conv2 = Conv2D(base_depth*2, (3, 3), activation='relu', padding='same')(conv2)
pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
```

(continues on next page)

(continued from previous page)

```
conv3 = Conv2D(base_depth*4, (3, 3), activation='relu', padding='same')(pool2)
conv3 = Conv2D(base_depth*4, (3, 3), activation='relu', padding='same')(conv3)

up4 = concatenate([UpSampling2D(size=(2, 2))(conv3), conv2], axis=3)
conv4 = Conv2D(base_depth*2, (3, 3), activation='relu', padding='same')(up4)
conv4 = Conv2D(base_depth*2, (3, 3), activation='relu', padding='same')(conv4)

up5 = concatenate([UpSampling2D(size=(2, 2))(conv4), conv1], axis=3)
conv5 = Conv2D(base_depth, (3, 3), activation='relu', padding='same')(up5)
conv5 = Conv2D(base_depth, (3, 3), activation='relu', padding='same')(conv5)

conv6 = Conv2D(1, (1, 1), activation='sigmoid')(conv5)

t_unet = Model(inputs=[inputs], outputs=[conv6])
```

```
2025-03-19 17:26:11.934503: I metal_plugin/src/device/metal_device.cc:1154] Metal␣
 ↪device set to: Apple M1 Max
2025-03-19 17:26:11.934536: I metal_plugin/src/device/metal_device.cc:296]␣
 ↪systemMemory: 32.00 GB
2025-03-19 17:26:11.934539: I metal_plugin/src/device/metal_device.cc:313]␣
 ↪maxCacheSize: 10.67 GB
WARNING: All log messages before absl::InitializeLog() is called are written to␣
 ↪STDERR
I0000 00:00:1742401571.934554 10572218 pluggable_device_factory.cc:305] Could not␣
 ↪identify NUMA node of platform GPU ID 0, defaulting to 0. Your kernel may not␣
 ↪have been built with NUMA support.
I0000 00:00:1742401571.934573 10572218 pluggable_device_factory.cc:271] Created␣
 ↪TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 0 MB␣
 ↪memory) -> physical PluggableDevice (device: 0, name: METAL, pci bus id:
 ↪<undefined>)
```

A network summary

```
t_unet.summary()
```

```
Model: "functional"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━┓
┃ Layer (type)          ┃ Output Shape          ┃       Param # ┃ Connected to          ┃
┡━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━┩
│ input_layer           │ (None, None,          │             0 │ -                     │
│ (InputLayer)          │ None, 1)              │               │                       │
├───────────────────────┼───────────────────────┼───────────────┼───────────────────────┤
│ conv2d (Conv2D)       │ (None, None,          │           640 │ input_layer[0][0]     │
│                       │ None, 64)             │               │                       │
├───────────────────────┼───────────────────────┼───────────────┼───────────────────────┤
│ conv2d_1 (Conv2D)     │ (None, None,          │        36,928 │ conv2d[0][0]          │
│                       │ None, 64)             │               │                       │
├───────────────────────┼───────────────────────┼───────────────┼───────────────────────┤
│ max_pooling2d         │ (None, None,          │             0 │ conv2d_1[0][0]        │
│ (MaxPooling2D)        │ None, 64)             │               │                       │
├───────────────────────┼───────────────────────┼───────────────┼───────────────────────┤
│ conv2d_2 (Conv2D)     │ (None, None,          │        73,856 │ max_pooling2d[0]…     │
│                       │ None, 128)            │               │                       │
```

(continues on next page)

(continued from previous page)

```
┏━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━┓
┃ conv2d_3 (Conv2D)   ┃ (None, None,      ┃    147,584 ┃ conv2d_2[0][0]    ┃
┃                     ┃ None, 128)        ┃            ┃                   ┃
┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
┃ max_pooling2d_1     ┃ (None, None,      ┃          0 ┃ conv2d_3[0][0]    ┃
┃ (MaxPooling2D)      ┃ None, 128)        ┃            ┃                   ┃
┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
┃ conv2d_4 (Conv2D)   ┃ (None, None,      ┃    295,168 ┃ max_pooling2d_1[… ┃
┃                     ┃ None, 256)        ┃            ┃                   ┃
┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
┃ conv2d_5 (Conv2D)   ┃ (None, None,      ┃    590,080 ┃ conv2d_4[0][0]    ┃
┃                     ┃ None, 256)        ┃            ┃                   ┃
┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
┃ up_sampling2d       ┃ (None, None,      ┃          0 ┃ conv2d_5[0][0]    ┃
┃ (UpSampling2D)      ┃ None, 256)        ┃            ┃                   ┃
┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
┃ concatenate         ┃ (None, None,      ┃          0 ┃ up_sampling2d[0]… ┃
┃ (Concatenate)       ┃ None, 384)        ┃            ┃ conv2d_3[0][0]    ┃
┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
┃ conv2d_6 (Conv2D)   ┃ (None, None,      ┃    442,496 ┃ concatenate[0][0] ┃
┃                     ┃ None, 128)        ┃            ┃                   ┃
┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
┃ conv2d_7 (Conv2D)   ┃ (None, None,      ┃    147,584 ┃ conv2d_6[0][0]    ┃
┃                     ┃ None, 128)        ┃            ┃                   ┃
┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
┃ up_sampling2d_1     ┃ (None, None,      ┃          0 ┃ conv2d_7[0][0]    ┃
┃ (UpSampling2D)      ┃ None, 128)        ┃            ┃                   ┃
┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
┃ concatenate_1       ┃ (None, None,      ┃          0 ┃ up_sampling2d_1[… ┃
┃ (Concatenate)       ┃ None, 192)        ┃            ┃ conv2d_1[0][0]    ┃
┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
┃ conv2d_8 (Conv2D)   ┃ (None, None,      ┃    110,656 ┃ concatenate_1[0]… ┃
┃                     ┃ None, 64)         ┃            ┃                   ┃
┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
┃ conv2d_9 (Conv2D)   ┃ (None, None,      ┃     36,928 ┃ conv2d_8[0][0]    ┃
┃                     ┃ None, 64)         ┃            ┃                   ┃
┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
┃ conv2d_10 (Conv2D)  ┃ (None, None,      ┃         65 ┃ conv2d_9[0][0]    ┃
┃                     ┃ None, 1)          ┃            ┃                   ┃
┗━━━━━━━━━━━━━━━━━━━━━┻━━━━━━━━━━━━━━━━━━━┻━━━━━━━━━━━━┻━━━━━━━━━━━━━━━━━━━┛
```

```
Total params: 1,881,985 (7.18 MB)
```

```
Trainable params: 1,881,985 (7.18 MB)
```

```
Non-trainable params: 0 (0.00 B)
```

A schematic rendering of the network

```
from IPython.display import SVG
from tensorflow.keras.utils import model_to_dot

dot_mod = model_to_dot(t_unet, show_shapes=False, show_layer_names=False, dpi=50)
dot_mod.set_rankdir('LR')
SVG(dot_mod.create_svg())
```

```
<IPython.core.display.SVG object>
```

0.19.3 New training data

We need to reduce the training image (to save time)

```
cell_img = ((io.imread("data/em_image.png")[::2, ::2])/255.0).astype(float)
cell_seg = (io.imread("data/em_image_seg.png", as_gray=True)[::2, ::2] > 0).
↪astype(float)

train_img, valid_img   = cell_img[:256, 50:250], cell_img[:256, -200:]
train_mask, valid_mask = cell_seg[:256, 50:250], cell_seg[:256, -200:]

# add channels and sample dimensions
def prep_img(x, n=1): return (
    prep_mask(x, n=n)-train_img.mean())/train_img.std() # This normalization is an␣
↪important step!

def prep_mask(x, n=1): return np.stack([np.expand_dims(x, -1)]*n, 0)

print('Training Data: image', train_img.shape, ', mask',train_mask.shape)
print('Validation Data: image', valid_img.shape, ', mask',valid_mask.shape)
```

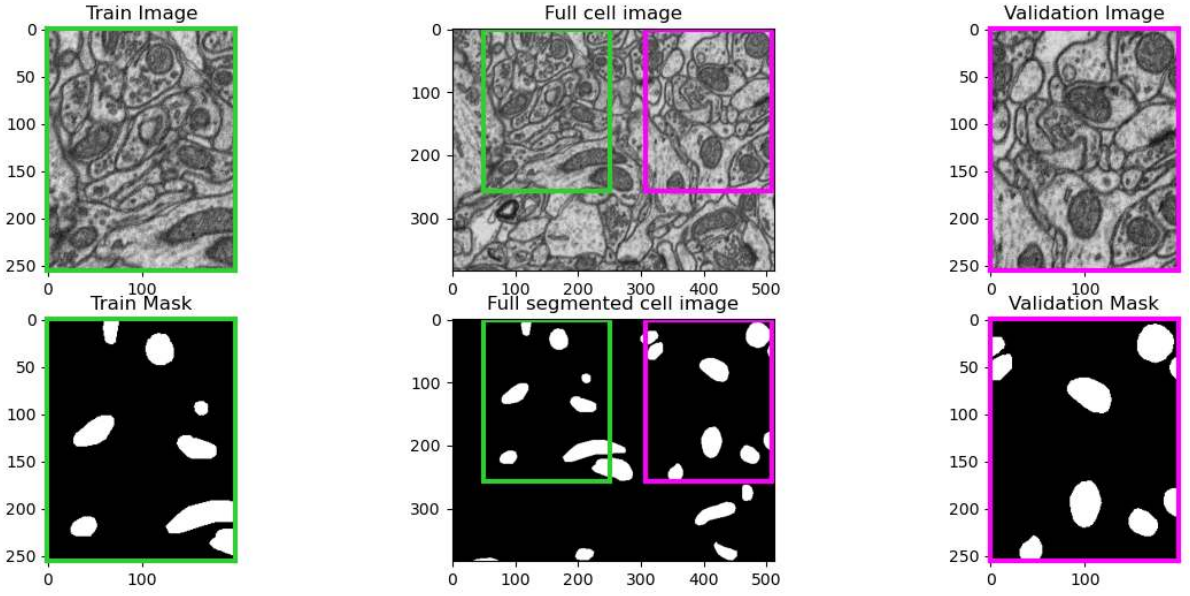
```
Training Data: image (256, 200) , mask (256, 200)
Validation Data: image (256, 200) , mask (256, 200)
```

```
from matplotlib.patches import Rectangle
from matplotlib.spines import Spine

r_train    = Rectangle((50,0),200,256,fc='none',ec='limegreen',lw=3)
# r_validate = Rectangle((256,0),cell_img.shape[1]-258,cell_img.shape[0]-1,fc='none',
↪ec='magenta',lw=3)
r_validate = Rectangle((308,0),200,256,fc='none',ec='magenta',lw=3)
fig, ax = plt.subplots(2, 3, figsize=(15, 6) )
ax=ax.ravel()
cmap='gray'

def coloraxes(ax,color) :
    for child in ax.get_children():
        if isinstance(child, Spine):
            child.set_color(color)
            child.set_linewidth(3)
```

(continues on next page)

(continued from previous page)

```
ax[0].imshow(train_img, cmap=cmap)
ax[0].set_title('Train Image')
coloraxes(ax[0],'limegreen')

ax[3].imshow(train_mask, cmap=cmap)
ax[3].set_title('Train Mask')
coloraxes(ax[3],'limegreen')

ax[1].imshow(cell_img,cmap=cmap)
ax[1].set_title('Full cell image')
ax[1].add_patch(r_train)
ax[1].add_patch(r_validate)

r_train    = Rectangle((50,0),200,256,fc='none',ec='limegreen',lw=3)
# r_validate = Rectangle((256,0),cell_img.shape[1]-258,cell_img.shape[0]-1,fc='none',
 ↪ec='magenta',lw=3)
r_validate = Rectangle((308,0),200,256,fc='none',ec='magenta',lw=3)
ax[4].imshow(cell_seg,cmap=cmap)
ax[4].set_title('Full segmented cell image')
ax[4].add_patch(r_train)
ax[4].add_patch(r_validate)

ax[2].imshow(valid_img, cmap=cmap)
ax[2].set_title('Validation Image')
coloraxes(ax[2],'magenta')
ax[5].imshow(valid_mask, cmap=cmap)
ax[5].set_title('Validation Mask');
coloraxes(ax[5],'magenta')
```



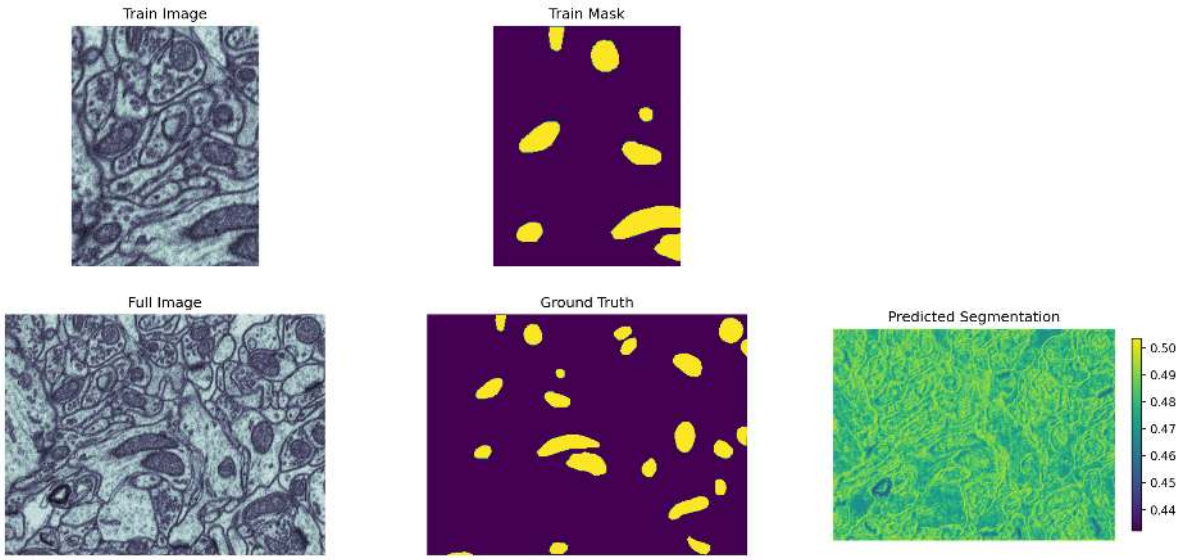
0.19.4 Results from Untrained Model

- We can make predictions with an untrained model (default parameters)
- but we clearly do not expect them to be very good

```
verbose=0 # 0 = silent, 2 = reports each epoch but no progress bar (set 0 for lecture␣
↪note generation)
unet_pred = t_unet.predict(prep_img(cell_img),verbose=verbose)[0, :, :, 0];
```

```
/opt/anaconda3/envs/qbi2025/lib/python3.9/site-packages/keras/src/models/
↪functional.py:237: UserWarning: The structure of `inputs` doesn't match the␣
↪expected structure.
Expected: ['keras_tensor']
Received: inputs=Tensor(shape=(1, 384, 512, 1))
  warnings.warn(msg)
2025-03-19 17:26:12.724374: I tensorflow/core/grappler/optimizers/custom_graph_
↪optimizer_registry.cc:117] Plugin optimizer for device_type GPU is enabled.
```

```
fig, m_axs = plt.subplots(2, 3,
                          figsize=(18, 8), dpi=150)
for c_ax in m_axs.flatten():
    c_ax.axis('off')
((ax1, ax2, _), (ax3, ax4, ax5)) = m_axs
ax1.imshow(train_img, cmap='bone')
ax1.set_title('Train Image')
ax2.imshow(train_mask, cmap='viridis')
ax2.set_title('Train Mask')

ax3.imshow(cell_img, cmap='bone')
ax3.set_title('Full Image')

ax4.imshow(cell_seg,
           cmap='viridis')
ax4.set_title('Ground Truth');

# a5=ax5.imshow(unet_pred,
#             cmap='viridis', vmin=0.48,vmax=0.52)
a5=ax5.imshow(unet_pred,
            cmap='viridis')
ax5.set_title('Predicted Segmentation')
fig.colorbar(a5,ax=ax5,shrink=0.8);
```



Note here that the predictions all are around 0.5, *i.e.*, close to a random guess.

0.19.5 A general note on the following demo

This is a very bad way to train a model;

- the loss function is poorly chosen,
- the optimizer can be improved the learning rate can be changed,
- the training and validation data **should not** come from the same sample (and **definitely** not the same measurement).

The goal is to be aware of these techniques and have a feeling for how they can work for complex problems

Training conditions

- Loss function - MAE
- Optimizer - Stochastic Gradient Decent and specifically the Adam optimizer.
- 20 Epochs (training iterations)
- Metrics
    1. Binary accuracy (percentage of pixels correct classified) $\$BA = \frac{1}{N}\sum_i (f_i == g_i)\$$
    2. Mean absolute error

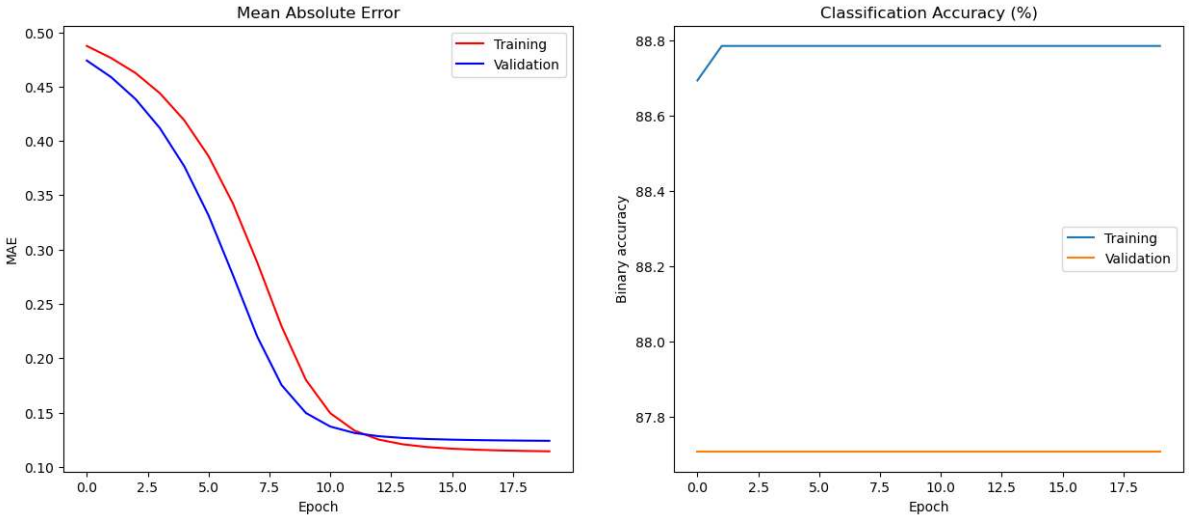
Another popular metric is the Dice score $\$DSC = \frac{2|X \cap Y|}{|X|+|Y|} = \frac{2\,TP}{2TP+FP+FN}\$$

Let's train the model

```
opt = tf.keras.optimizers.Adam(learning_rate=1e-4)

t_unet.compile(
            optimizer=opt,
            # we use a simple loss metric of mean-squared error to optimize
            loss="MSE",
            # we keep track of the number of pixels correctly classified and the mean␣
 ↪absolute error as well
            metrics=['binary_accuracy', 'mae'])

loss_history = t_unet.fit(prep_img(train_img),
                          prep_mask(train_mask),
                          validation_data=(prep_img(valid_img),
                                           prep_mask(valid_mask)),
                          epochs=20, verbose=verbose)
```

```
/opt/anaconda3/envs/qbi2025/lib/python3.9/site-packages/keras/src/models/
 ↪functional.py:237: UserWarning: The structure of `inputs` doesn't match the␣
 ↪expected structure.
Expected: ['keras_tensor']
Received: inputs=Tensor(shape=(None, 256, 200, 1))
  warnings.warn(msg)
```

```
fig, (ax1, ax2) = plt.subplots(1, 2,
                               figsize=(15,6))
ax1.plot(loss_history.epoch,
         loss_history.history['mae'], 'r-', label='Training')
ax1.plot(loss_history.epoch,
         loss_history.history['val_mae'], 'b-', label='Validation')
ax1.set_title('Mean Absolute Error')
ax1.set_xlabel('Epoch')
ax1.set_ylabel('MAE')
ax1.legend()

ax2.plot(loss_history.epoch,
         100*np.array(loss_history.history['binary_accuracy']), '-', label='Training')
ax2.plot(loss_history.epoch,
         100*np.array(loss_history.history['val_binary_accuracy']), '-', label=
 ↪'Validation')

ax2.set_xlabel('Epoch')
ax2.set_ylabel('Binary accuracy')
ax2.set_title('Classification Accuracy (%)')
ax2.legend();
```



Prediction results

```
unet_train = t_unet.predict(prep_img(train_img),verbose=verbose)[0, :, :, 0]
unet_pred  = t_unet.predict(prep_img(cell_img),verbose=verbose)[0, :, :, 0]
```

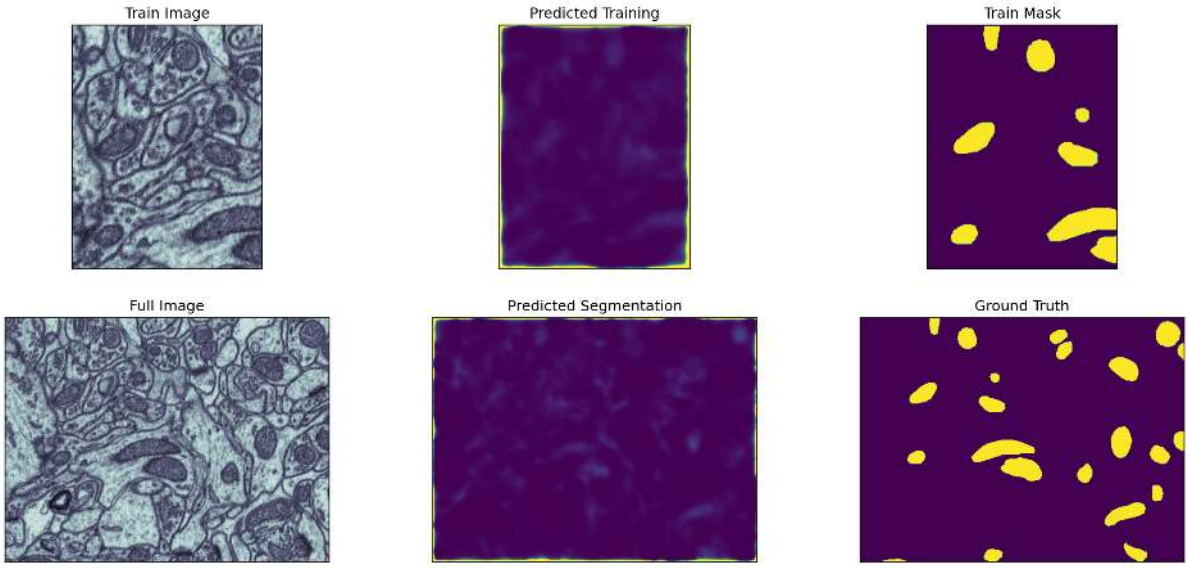
```
/opt/anaconda3/envs/qbi2025/lib/python3.9/site-packages/keras/src/models/
 ↪functional.py:237: UserWarning: The structure of `inputs` doesn't match the␣
 ↪expected structure.
Expected: ['keras_tensor']
Received: inputs=Tensor(shape=(1, 256, 200, 1))
  warnings.warn(msg)
/opt/anaconda3/envs/qbi2025/lib/python3.9/site-packages/keras/src/models/
 ↪functional.py:237: UserWarning: The structure of `inputs` doesn't match the␣
 ↪expected structure.
Expected: ['keras_tensor']
Received: inputs=Tensor(shape=(1, None, None, 1))
  warnings.warn(msg)
```

```
fig, m_axs = plt.subplots(2, 3, figsize=(18, 8), dpi=150)
for c_ax in m_axs.flatten():
    c_ax.set(xticks=[],yticks=[])
((ax1, ax15, ax2), (ax3, ax4, ax5)) = m_axs
ax1.imshow(train_img, cmap='bone')
ax1.set_title('Train Image')
vmin=0
vmax=0.01
ax15.imshow(unet_train, cmap='viridis', vmin=vmin, vmax=vmax)
ax15.set_title('Predicted Training')
ax2.imshow(train_mask, cmap='viridis')
ax2.set_title('Train Mask')

ax3.imshow(cell_img, cmap='bone')
ax3.set_title('Full Image')

ax4.imshow(unet_pred,
           cmap='viridis', vmin=vmin, vmax=vmax)
```

(continues on next page)

(continued from previous page)

```
ax4.set_title('Predicted Segmentation')

ax5.imshow(cell_seg,
           cmap='viridis')
ax5.set_title('Ground Truth');
```



0.19.6 Overfitting

Having a model with 470,000 free parameters means that it is quite easy to overfit the model by training for too long.

Overfitting is when:

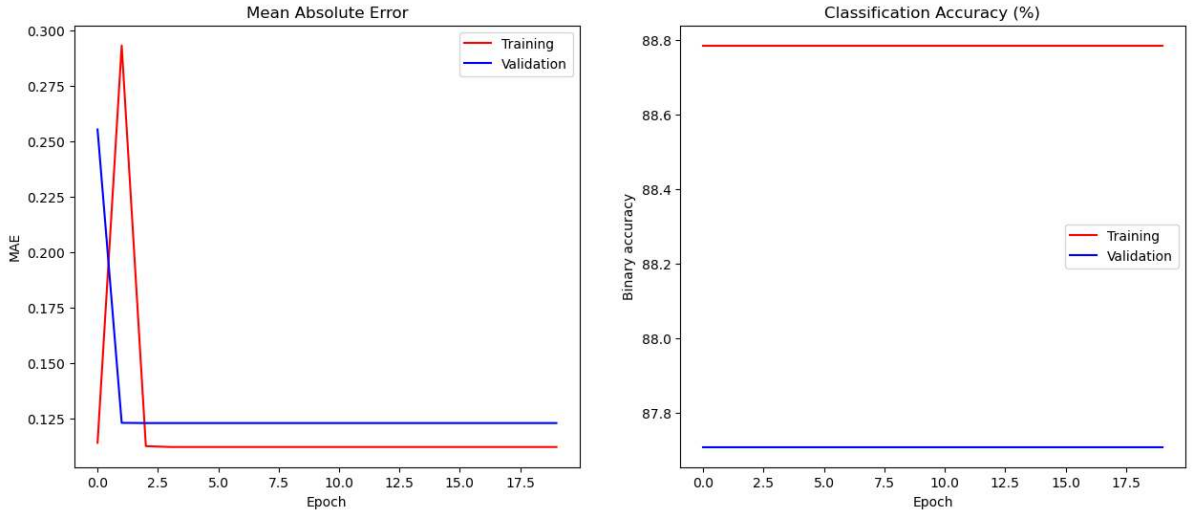
- The model has gotten very good at the training data
- but hasn't generalized to other kinds of problems

Consequence: The model starts to perform worse on regions that aren't exactly the same as the training.

```
t_unet.compile(
    # we use a simple loss metric of mean-squared error to optimize
    loss='mse',
    optimizer='Adam',
    # we keep track of the number of pixels correctly classified and the mean␣
 ↪absolute error as well
    metrics=['binary_accuracy', 'mae']
)

loss_history = t_unet.fit(prep_img(train_img),
                          prep_mask(train_mask),
                          validation_data=(prep_img(valid_img),
                                           prep_mask(valid_mask)),
                          epochs=20, verbose=verbose)
```

Loss history

```
fig, (ax1, ax2) = plt.subplots(1, 2,
                               figsize=(15,6))
ax1.plot(loss_history.epoch,
         loss_history.history['mae'], 'r-', label='Training')
ax1.plot(loss_history.epoch,
         loss_history.history['val_mae'], 'b-', label='Validation')
ax1.set_title('Mean Absolute Error')
ax1.set_xlabel('Epoch')
ax1.set_ylabel('MAE')
ax1.legend()

ax2.plot(loss_history.epoch,
         100*np.array(loss_history.history['binary_accuracy']), 'r-', label='Training
 ↪')
ax2.plot(loss_history.epoch,
         100*np.array(loss_history.history['val_binary_accuracy']), 'b-', label=
 ↪'Validation')
ax2.set_xlabel('Epoch')
ax2.set_ylabel('Binary accuracy')
ax2.set_title('Classification Accuracy (%)')
ax2.legend();
```



Prediction results

```
unet_train = t_unet.predict(prep_img(train_img),verbose=verbose)[0, :, :, 0]
unet_pred  = t_unet.predict(prep_img(cell_img),verbose=verbose)[0, :, :, 0]
```

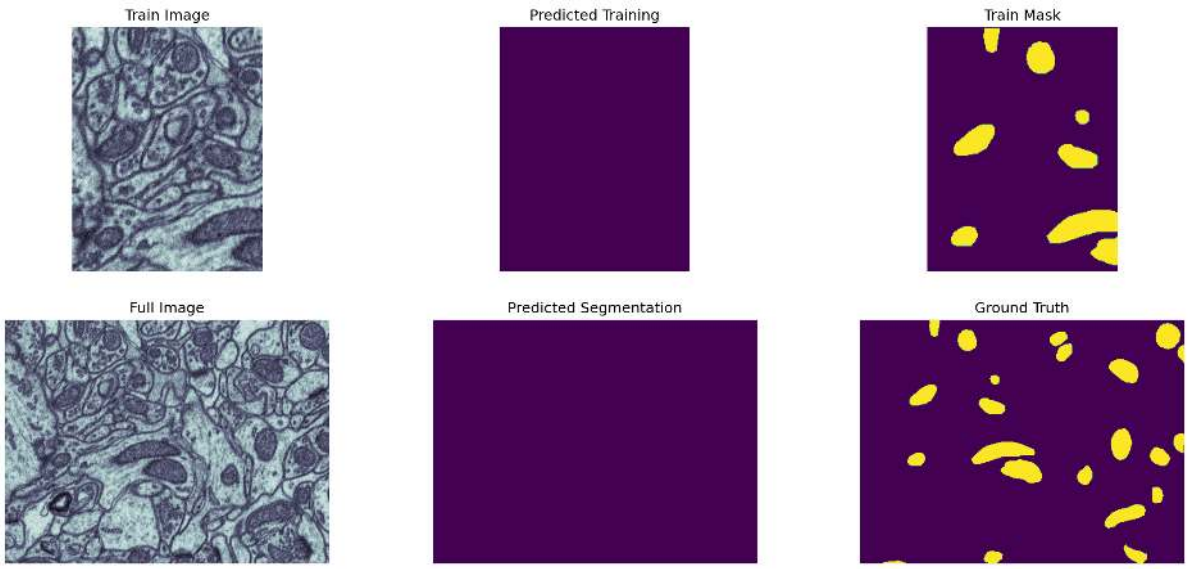
```
WARNING:tensorflow:5 out of the last 5 calls to <function TensorFlowTrainer.make_
 ↪predict_function.<locals>.one_step_on_data_distributed at 0x39da77940> triggered␣
 ↪tf.function retracing. Tracing is expensive and the excessive number of tracings␣
 ↪could be due to (1) creating @tf.function repeatedly in a loop, (2) passing␣
 ↪tensors with different shapes, (3) passing Python objects instead of tensors.␣
 ↪For (1), please define your @tf.function outside of the loop. For (2), @tf.
 ↪function has reduce_retracing=True option that can avoid unnecessary retracing.␣
```

(continues on next page)

(continued from previous page)

```
↪For (3), please refer to https://www.tensorflow.org/guide/function#controlling_
↪retracing and https://www.tensorflow.org/api_docs/python/tf/function for  more␣
↪details.
```

```
fig, m_axs = plt.subplots(2, 3,
                          figsize=(18, 8), dpi=150)
for c_ax in m_axs.flatten():
    c_ax.axis('off')
((ax1, ax15, ax2), (ax3, ax4, ax5)) = m_axs
ax1.imshow(train_img, cmap='bone')
ax1.set_title('Train Image')
ax15.imshow(unet_train, cmap='viridis', vmin=0, vmax=1)
ax15.set_title('Predicted Training')
ax2.imshow(train_mask, cmap='viridis')
ax2.set_title('Train Mask')

ax3.imshow(cell_img, cmap='bone')
ax3.set_title('Full Image')

ax4.imshow(unet_pred,
           cmap='viridis', vmin=0, vmax=1)
ax4.set_title('Predicted Segmentation')

ax5.imshow(cell_seg,
           cmap='viridis')
ax5.set_title('Ground Truth');
```



Some comments on the used U-net model

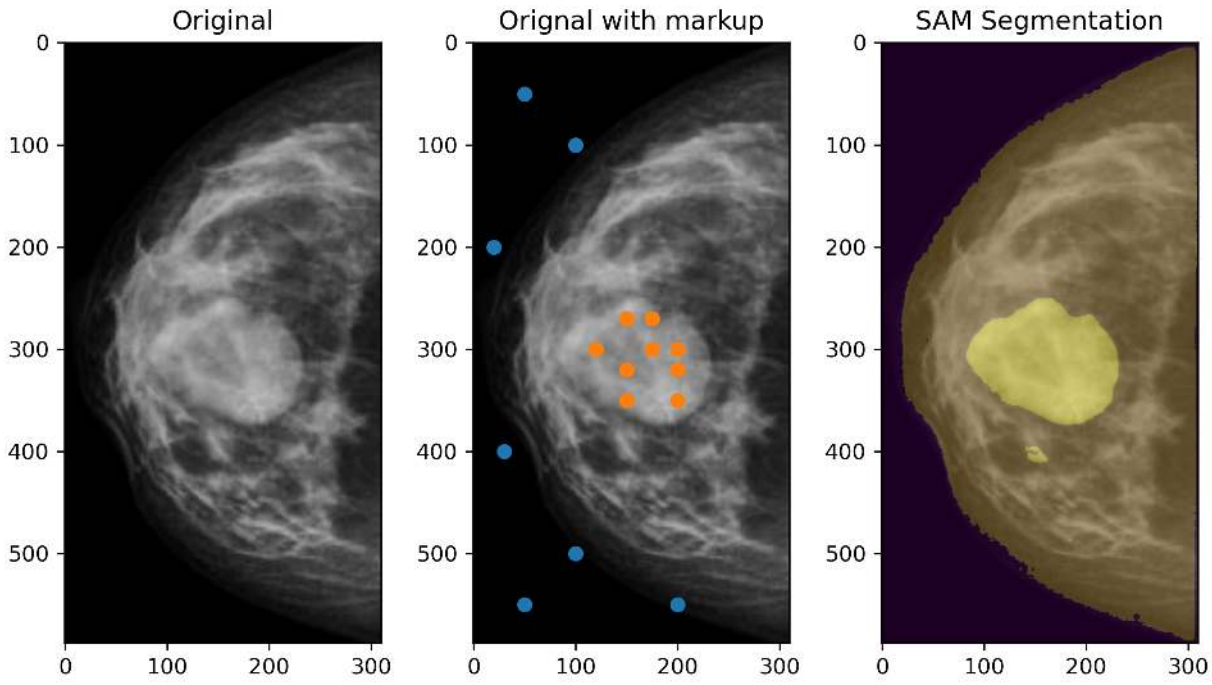
- This was a small model
- Trained on limited, far from ideal data

Mainly to show the building blocks and workflow.

0.19.7 The SAM model

Is a transformer model based on natural language processing models.

- Pretrained with a great variation of images
- Requires ques for the segmentation



Kirillov 2023 GitHub Example workbench

This example was done in short time to demonstrate the use of the SAM model. The markers were hand picked.

The model can be used as markup tool to generate ground truth images for other ML models and thus speed up the markup process.

0.19.8 Compare U-Nets and SAM

U-Net

- Is a classic, fully convolutional encoder–decoder architecture
- must be trained (or fine-tuned) on a labeled dataset for a specific segmentation task,
- often achieving high accuracy in that domain.

SAM

- Is a large, pre-trained, transformer-based foundation model
- Can segment virtually anything in a zero-shot manner,
- guided by user prompts.

0.20 Summary

- Concepts of supervised segmentation
- Supervised Classification
- Classification vs. Segmentation
  - Nearest neighbour
  - Trees
  - Random Forests
- Regression vs Classification
- Training, validation
- Deep learning