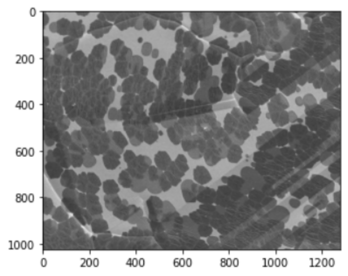
```
In [1]: # import all relevant libraries
        import numpy as np
        from PIL import Image
        import matplotlib.pyplot as plt
        from skimage import util
        from sklearn.cluster import KMeans, MiniBatchKMeans
```

Import the image

Now, we import the image as a numpy n-dimensional array and display the image and its resolution.

```
In [2]: img_in = np.array(Image.open('../data/test_kmeans2.tif').convert('L')) # import the grayscale image as an array

        print("Resolution of Image: ", img_in.shape) # display the resolution of the image
        img_plot = plt.imshow(img_in, cmap='gray') # display the image
```

```
Resolution of Image:  (1024, 1280)
```



Define the parameters

We define all the parameters that we can change.

```
In [3]: wsize = 30       # define the window size in pixels. This is the side length of the tiles.
        n_clusters = 5   # define the number of clusters. This is the "K" in "K-Means".
        stride = 3       # define the stride length. This is the number of pixels moved right or down between two tiles.
        seed = 197208    # define a random seed which will be used to determine the inital position of the centroids. This can be any random integer.
```

Pre-process the image

We make changes to the image to be able to run the method effectively.

```
In [4]: # define a function to add a some padding to the image so that we can have an integer number of tiles.

        def pad(img,wsize,stride=1):

            """Adds a few pixels to the image as padding so that we can have an integer number of tiles.

            Args:
                img (np.array of shape(img.shape)): Array containg the image.
                wsize (int): side length of tile.
                stride (int): number of pixels between two tiles (default=1).

            Returns:
                img (np.array): Padded image i.e. image with a few additional pixels.
            """


            height,width = img.shape
            if stride == 'block':
                adj = 0
                stride = wsize
            else:
                adj = 1

            px = wsize - height % stride - adj
            if px % 2 == 0:
                px = int(px/2)
                px = (px,px)
            else:
                px = int((px-1)/2)
                px = (px,px+1)

            py = wsize - width % stride - adj
            if py % 2 == 0:
                py = int(py/2)
                py = (py,py)
            else:
                py = int((py-1)/2)
                py = (py,py+1)

            return np.pad(img,pad_width=(px,py),mode='symmetric')
```

```
In [5]: # define a variable "X" which is a numpy array where each element is a tile.

        X = util.view_as_windows(
            pad(img_in,wsize=wsize,stride=stride), # add padding to the original image
            window_shape=(wsize,wsize),            # define the tile size
            step=stride)                           # define the stride length

        mask_dim = X.shape[:2]  # pick the first two values in the shape which correspond to the number of tiles.
                                # This can be called the reduced image where each pixel represents one tile.

        X=X.reshape(-1,wsize**2) # reshape "X" to get an array of the tiles as vectors. The resulting array has
                                 # the dimensions 'number of tiles x number of pixels per tile'.
```

Run the model

We define the standard and mini-batch k-means models and fit them to our data.

```
In [6]: kmeans = KMeans(
            n_clusters=n_clusters,
            random_state=seed) # create a KMeans object which contains the method

        %time kmeans = kmeans.fit(X) # fit the model to our pre-processed image data

        CPU times: user 4min 23s, sys: 1min 4s, total: 5min 28s
        Wall time: 30.2 s
```

```
In [7]: mb_kmeans = MiniBatchKMeans(
            n_clusters=n_clusters,
            random_state=seed) # create a MiniBatchKMeans object which contains the method

        %time mb_kmeans = mb_kmeans.fit(X) # fit the model to our pre-processed image data

        # Note the difference in time between KMeans and MiniBatchKMeans

        CPU times: user 11.3 s, sys: 2.69 s, total: 14 s
        Wall time: 3.32 s
```

Post-process the fitted data

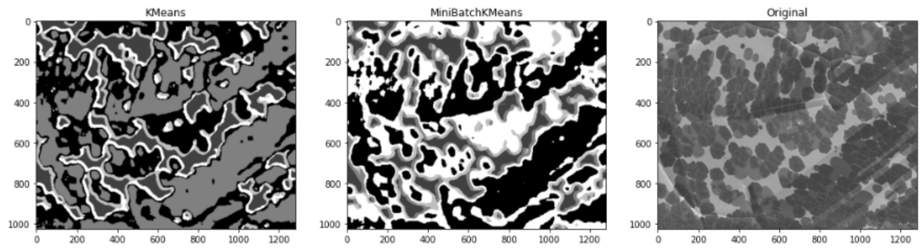
Each of the models return certain attributes that contain all the relevant data. See more: KMeans, MiniBatchKMeans

```
In [8]: mask = kmeans.labels_.reshape(*mask_dim) # reshape the "labels_" attribute to match the dimensions of the reduced image
        mask = Image.fromarray(mask) # convert the mask into an Image object

        mb_mask = mb_kmeans.labels_.reshape(*mask_dim) # reshape the "labels_" attribute to match the dimensions of the reduced image
        mb_mask = Image.fromarray(mb_mask) # convert the mask into an Image object
```

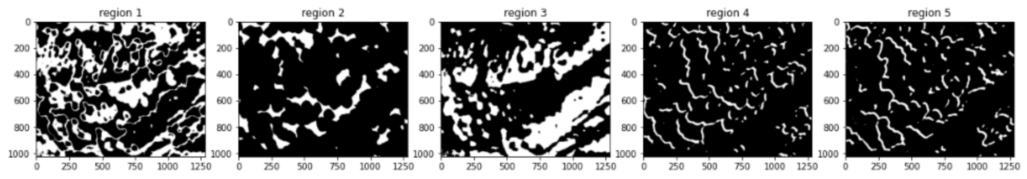
```
In [9]: clusters = np.array(mask.resize(img_in.shape[::-1]))+1 # scale the reduced image to the size of the input image
        mb_clusters = np.array(mb_mask.resize(img_in.shape[::-1]))+1 # scale the reduced image to the size of the input image

        # modify "mb_clusters" to match the colour scheme in "clusters". This is only for visual comparison between the two.
        mod_mb_clusters = np.where(mb_clusters == 5, 2,0) + np.where(mb_clusters == 4, 4,0) + np.where(mb_clusters == 1, 1,0) + np.where(mb_clusters == 3, 5,0) + np.where(mb_clusters == 2, 3,0)
```

Plot the segmented images

```
In [10]: fig, ax = plt.subplots(nrows=1,ncols=3, figsize=(18,6)) # create a figure with sub-plots

         # show the images segemented by KMeans and MiniBatchKMeans respectively
         ax[0].imshow(clusters, cmap='gray');
         ax[1].imshow(mod_mb_clusters, cmap='gray');
         ax[2].imshow(img_in, cmap='gray')

         # label the images
         ax[0].set_title('KMeans')
         ax[1].set_title('MiniBatchKMeans')
         ax[2].set_title('Original')

Out[10]: Text(0.5, 1.0, 'Original')
```



```
In [11]: # create a figure where each of the regions are plotted separately

         fig, ax = plt.subplots(nrows=1,ncols=n_clusters, figsize=(4*n_clusters,4))

         for ncl in range(1,n_clusters+1):
             ax[ncl-1].imshow(np.where(clusters == ncl, 1, 0), cmap='gray')
             ax[ncl-1].set_title('region %i' %ncl)
```



Varying the parameters

Now, we vary the stride length, number of clusters and the window size and see how it affects our result.

In [12]:
```
# define a function that does MiniBatchKMeans and directly outputs an array of the cluster values

def mbkmeans(n_clusters,seed,stride,wsize):
    mb_kmeans = MiniBatchKMeans(
        n_clusters=n_clusters,
        random_state=seed)

    mb_X = util.view_as_windows(
        pad(img_in,wsize=wsize,stride=stride),
        window_shape=(wsize,wsize),
        step=stride)

    mb_mask_dim = mb_X.shape[:2]
    mb_X=mb_X.reshape(-1,wsize**2)

    mb_kmeans = mb_kmeans.fit(mb_X)

    mb_mask = mb_kmeans.labels_.reshape(*mb_mask_dim)
    mb_mask = Image.fromarray(mb_mask)

    mb_clusters = np.array(mb_mask.resize(img_in.shape[::-1]))+1
    return mb_clusters
```

Vary the number of clusters:

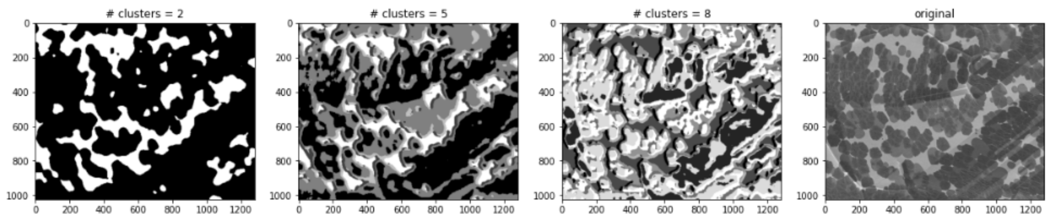
See the difference between having 2, 5 and 8 clusters. Note the differences in the time taken to execute the segmentation.

In [13]:
```
wsize = 30
nclusters = [2, 5, 8]
stride = 3
seed = 197208

fig, ax = plt.subplots(ncols=4, nrows=1, figsize=(20,5))
for iter, ncl in enumerate(nclusters):
    %time mb_cluster=mbkmeans(ncl,seed,stride,wsize)
    ax[iter].imshow(mb_cluster, cmap='gray');
    ax[iter].set_title('# clusters = %i' %ncl)
ax[3].imshow(img_in, cmap='gray')
ax[3].set_title('original')
```

```
CPU times: user 11.3 s, sys: 2.48 s, total: 13.8 s
Wall time: 1.46 s
CPU times: user 11.4 s, sys: 2.13 s, total: 13.5 s
Wall time: 1.32 s
CPU times: user 12.1 s, sys: 2.26 s, total: 14.4 s
Wall time: 1.39 s
```

Out[13]:
```
Text(0.5, 1.0, 'original')
```

```
Out[13]: Text(0.5, 1.0, 'original')
```



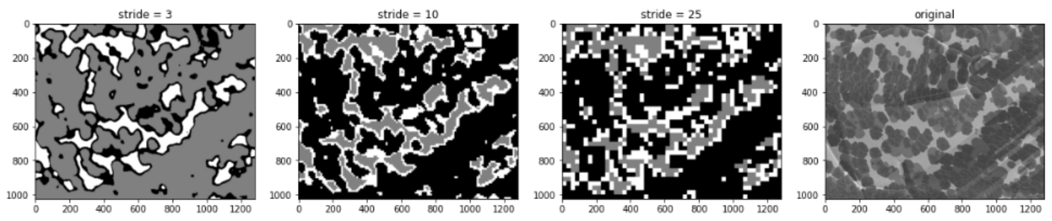
Vary the stride: ¶

See the difference between having a stride length of 3, 10 and 25 pixels. Note the differences in the time taken to execute the segmentation.

```
In [14]: wsize = 30
         nclusters = 3
         stride = [3,10,25]
         seed = 197208

         fig, ax = plt.subplots(ncols=4, nrows=1, figsize=(20,5))
         for iter, nst in enumerate(stride):
             %time mb_cluster=mbkmeans(nclusters,seed,nst,wsize)
             ax[iter].imshow(mb_cluster, cmap='gray');
             ax[iter].set_title('stride = %i' %nst)
         ax[3].imshow(img_in, cmap='gray')
         ax[3].set_title('original')
```

```
CPU times: user 8.94 s, sys: 1.8 s, total: 10.7 s
Wall time: 1.3 s
CPU times: user 1.64 s, sys: 853 ms, total: 2.49 s
Wall time: 156 ms
CPU times: user 802 ms, sys: 605 ms, total: 1.41 s
Wall time: 88.3 ms
```

```
Out[14]: Text(0.5, 1.0, 'original')
```



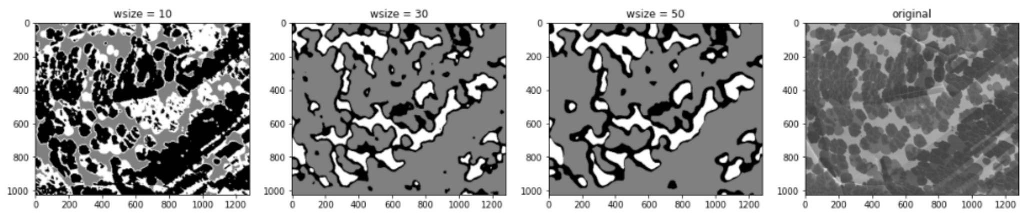
Vary the window size (wsize):

See the difference between having a window side length of 10, 30 and 50 pixels. Note the differences in the time taken to execute the segmentation.

```
In [15]: wsize = [10,30,50]
         nclusters = 3
         stride = 3
         seed = 197208

         fig, ax = plt.subplots(ncols=4, nrows=1, figsize=(20,5))
         for iter, nws in enumerate(wsize):
             %time mb_cluster=mbkmeans(nclusters,seed,stride,nws)
             ax[iter].imshow(mb_cluster, cmap='gray');
             ax[iter].set_title('wsize = %i' %nws)
         ax[3].imshow(img_in, cmap='gray')
         ax[3].set_title('original')
```

```
CPU times: user 739 ms, sys: 1.26 s, total: 2 s
Wall time: 414 ms
CPU times: user 9.54 s, sys: 2.12 s, total: 11.7 s
Wall time: 1.32 s
CPU times: user 2min 24s, sys: 43.5 s, total: 3min 8s
Wall time: 4min 16s
```

```
Out[15]: Text(0.5, 1.0, 'original')
```



```
In [ ]:
```