CSCI E-25

Extracting Features From Images

Steve Elston

Feature extraction from images is a fundamental aspect of computer vision. Image features must be extracted for many computer vision methods. Just a few of these are:

1. Object recognition.
2. Stitching images together.
3. Locating and identifying objects in images.
4. Models of motion from a series of images.
5. Stereo vision and depth estimation.

In these exercises we focus on classical methods for extracting features from images. In state-of-the-art practice features are created using deep neural networks. Will address learning features with deep neural networks extensively latter. Rest assured that some background in classical methods will help you understand and appreciate the deep learning methods.

To get started, execute the code in the cell below to import the packages you will need for the exercises.

In [ ]:
```
import skimage
from skimage import data
from skimage.filters.rank import equalize, entropy
import skimage.filters as skfilters
import skimage.feature as feature
import skimage.segmentation as segmentation
import skimage.morphology as morphology
import skimage.transform as transform
import skimage.util as util
from skimage.color import rgb2gray
from skimage import exposure
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [ ]:
```
skimage.__version__
```

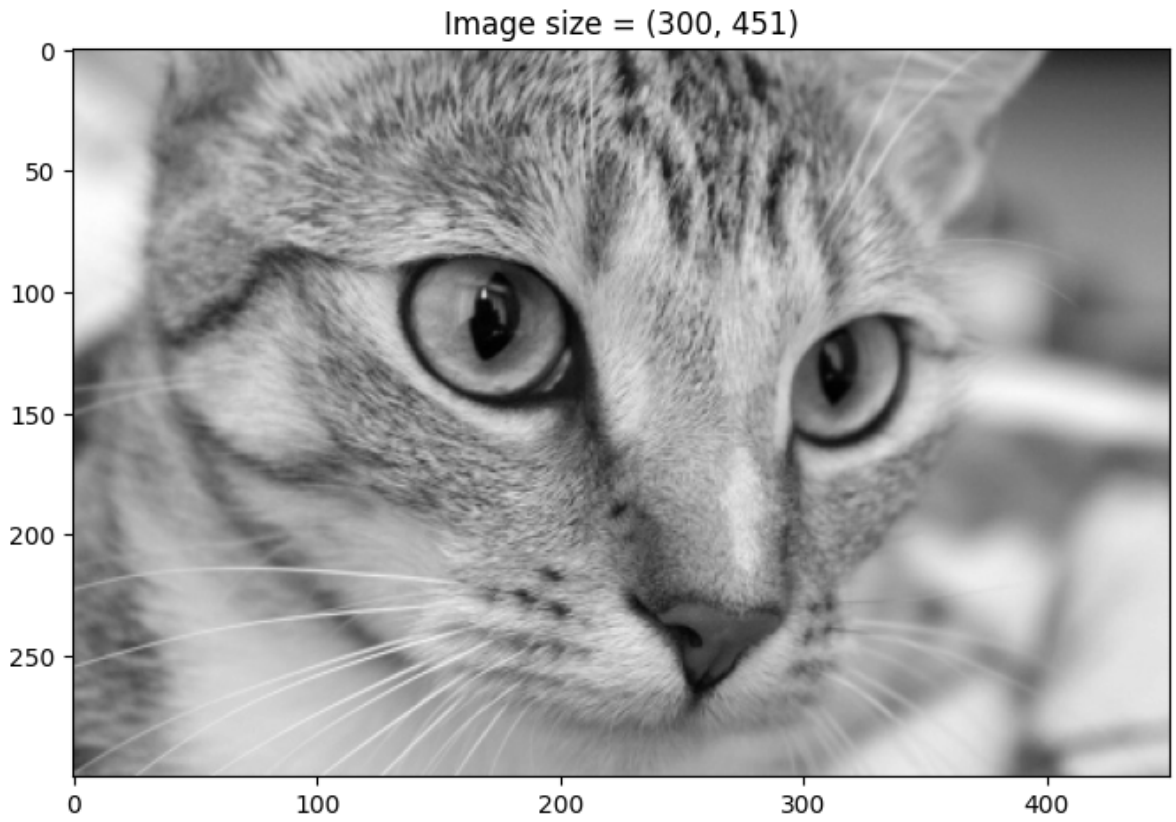
Out[ ]:
```
'0.19.3'
```

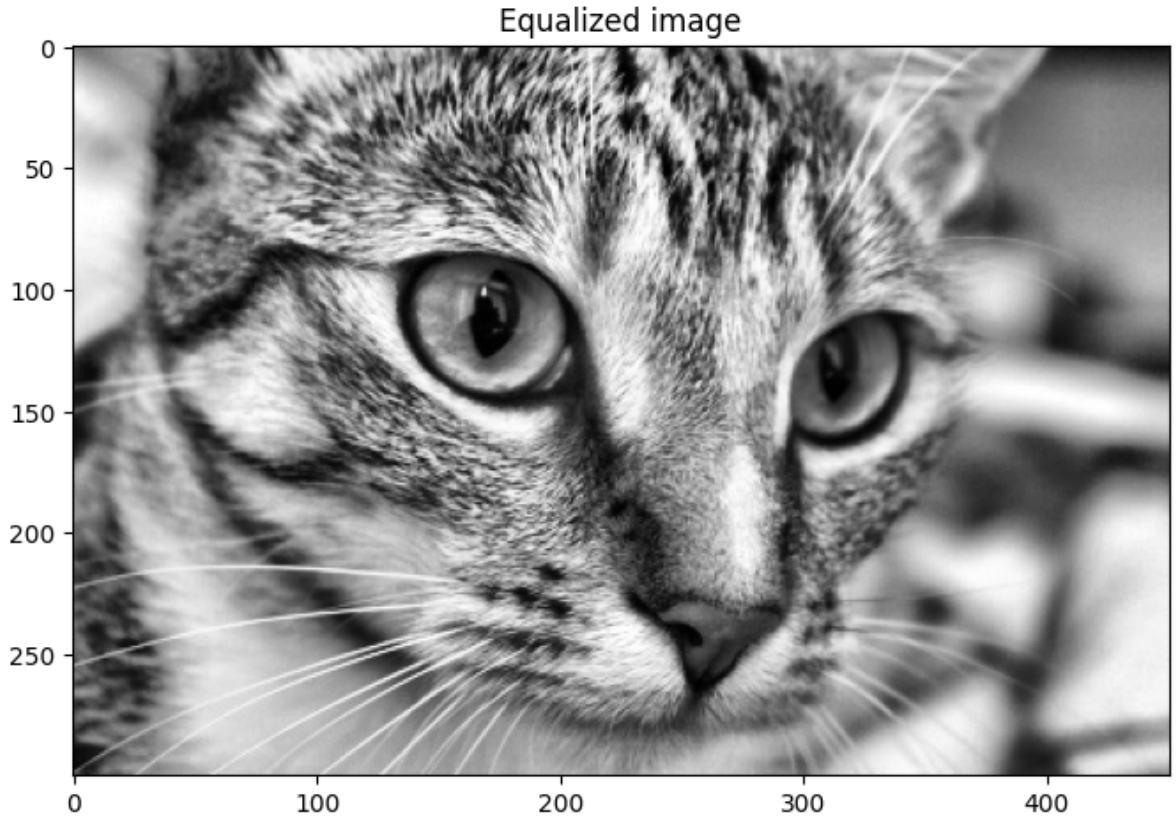
Preparing the Image

Regardless of the methods used an image must be prepared for feature extraction. Often many steps of correcting and filtering the image are required for proper feature extraction. An important step is the adjustment of the spectrum of the image. We need the spectrum to be as wide as possible. We call the transformation **pre-whitening**. The broad spectrum is considered white light has all spectral components equally represented. We have already discussed the process of whitening the spectrum of images, contrast improvement. An image with high contrast has a broad spectrum.

The cat image requires little preprocessing, beyond pre-whitening. Execute the code in the cell below to load and prepare the cat image.

In [ ]:
```
def plot_grayscale(img, title, h=8):
    plt.figure(figsize=(h, h))
    _=plt.imshow(img, cmap=plt.get_cmap('gray'))
    _=plt.title(title)

cat_image = rgb2gray(data.cat())
plot_grayscale(cat_image, 'Image size = ' + str(cat_image.shape))

cat_grayscale_equalized = exposure.equalize_adapthist(cat_image)
plot_grayscale(cat_grayscale_equalized, 'Equalized image')
```





> **Important note!** Unless otherwise stated, the `cat_grayscale_equalized` image should be used for subsequent exercises.

Edge Detection

Edges are a fundamental feature of am image. Edges are characterized by rapid changes in the intensity and are therefore associated with **high gradients**. The rapid change in intensity levels also means that the spectrum around an edge will have high-frequency components.

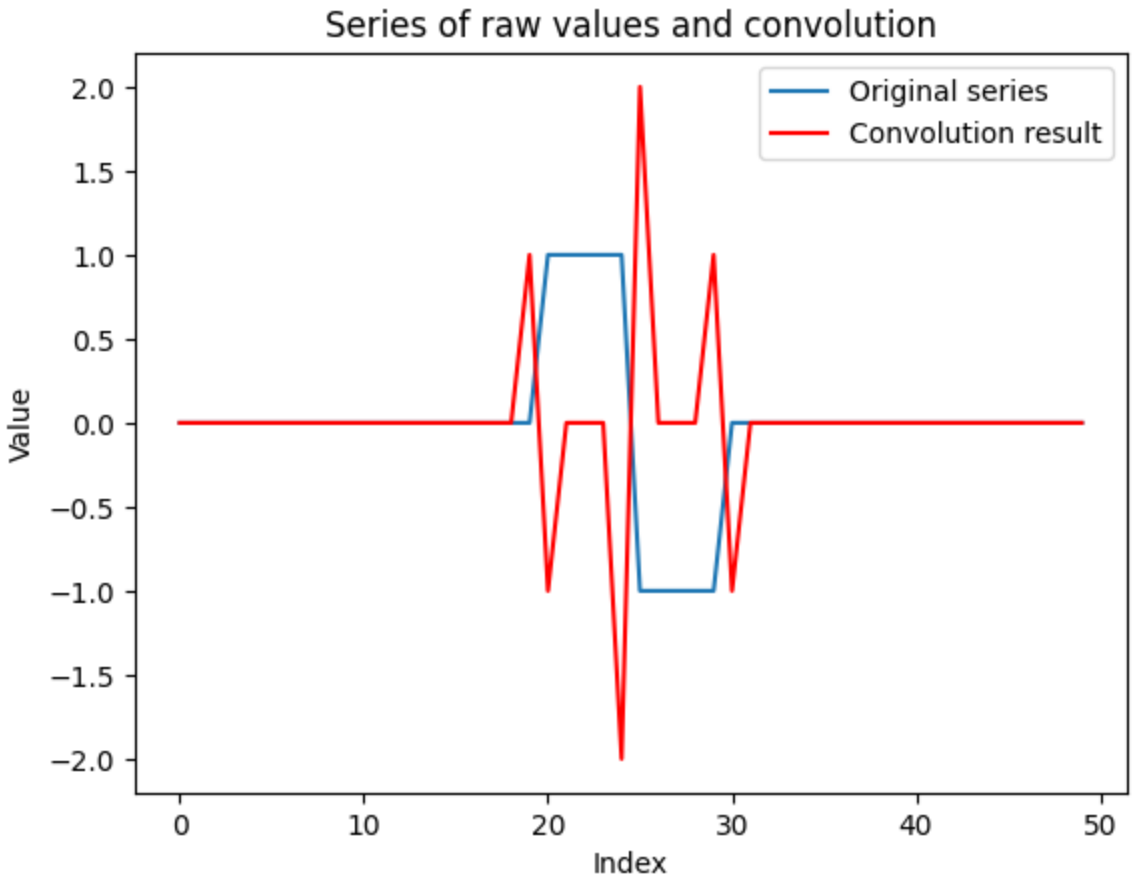
We can take advantage of the broad spectrum at edge features to create an edge detector algorithm. Recall that a Gaussian filter acts is low-pass, reducing the high frequency components. If we take the difference of the Gaussian filters with different bandwidths we can detect edges.

> **Exercise 3.1:** To gain a bit of understanding from a simple case you will create a simple one dimensional example of an edge detector. In this case, you will apply a convolutional gradient operator to a square wave function that is first positive and then negative. Aficionados of obscure functions will notice this is a Harr basis function (Harr, 1909). You will find and display the changes in gradient of this function by the following steps:

1. Create the 1-d convolution operator as a Numpy array: $[1.0, -2.0, 1.0]$.
2. Convolve the `series` with the operator using the numpy.convolve function. Make sure to set the `mode='same'` argument so the resulting series is the same length as the original.
3. Plot the result with the `plot_conv` function provided.

In [ ]:
```
def plot_conv(series, conv, span):
    x = list(range(series.shape[0]))
    plt.plot(x, series, label = 'Original series')
    plt.plot(x, conv, color = 'red', label = 'Convolution result')
    plt.legend()
    plt.xlabel('Index')
    plt.ylabel('Value')
    plt.title('Series of raw values and convolution')

series = np.concatenate((np.zeros((20,)), np.zeros((5,)) + 1.0, np.zeros((5,

print(series)

##  Your code goes here

#1. Create the 1-d convolution operator as a Numpy array: [1.0,-2.0,1.0].
convolution_operator = np.array([1.0, -2.0, 1.0])

# 2. Convolve the `series` with the operator using  the numpy.convolve funct
#    `mode='same'` argument so the resulting series is the same length as th
result = np.convolve(series, convolution_operator, mode='same')

# 3. Plot the result with the `plot_conv` function provided.
plot_conv(series, result, 1)
```

```
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  1.  1.  1.  1.  1. -1. -1. -1. -1. -1.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```



> Examine the plot and answer the following questions:
>
> 1. Why are the 3-point convolution operator values correct in terms of sign, and normalization?
> 2. Does the magnitude of the resulting series correctly represent the gradient of the original series, and why?
>
> **End of exercise.**

> **Answers:**

> 1. The 3-point convolution operator values [1.0, -2.0, 1.0] are correct in terms of sign and normalization because they detect changes in the gradient of the series. The operator values are set in a way that the convolution returns a positive value when the series has a positive gradient, a negative value when the series has a negative gradient, and zero when the series has a flat gradient.

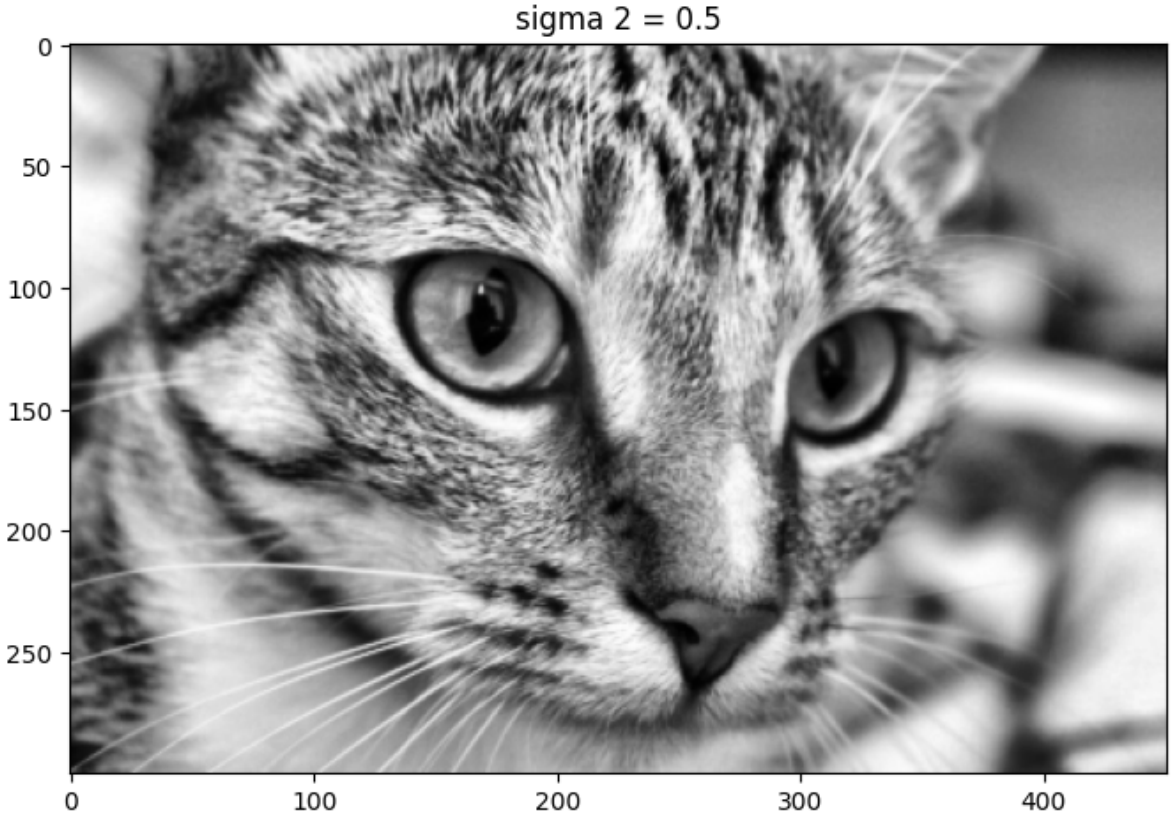
> 2. Yes, it does. Analyzing the plot of the resulting series we can see that it correctly reflects the convolution in each step. For instance, after applying the convolution operator [1.0, -2.0, 1.0] to the series, we get the following (I removed trailing zeroes to make it shorter):

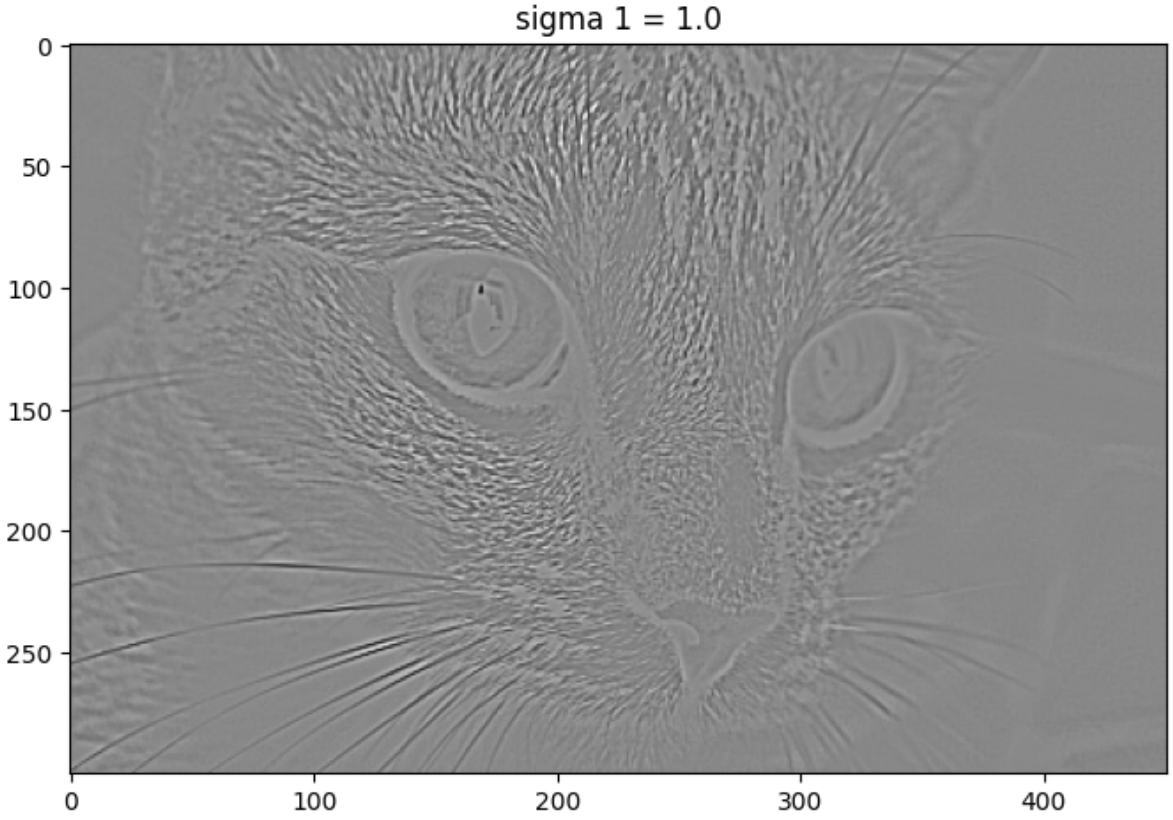
..., 1, -1, 0, 0, 0, -2, 2, 0, 0, 0, 1, -1, ...

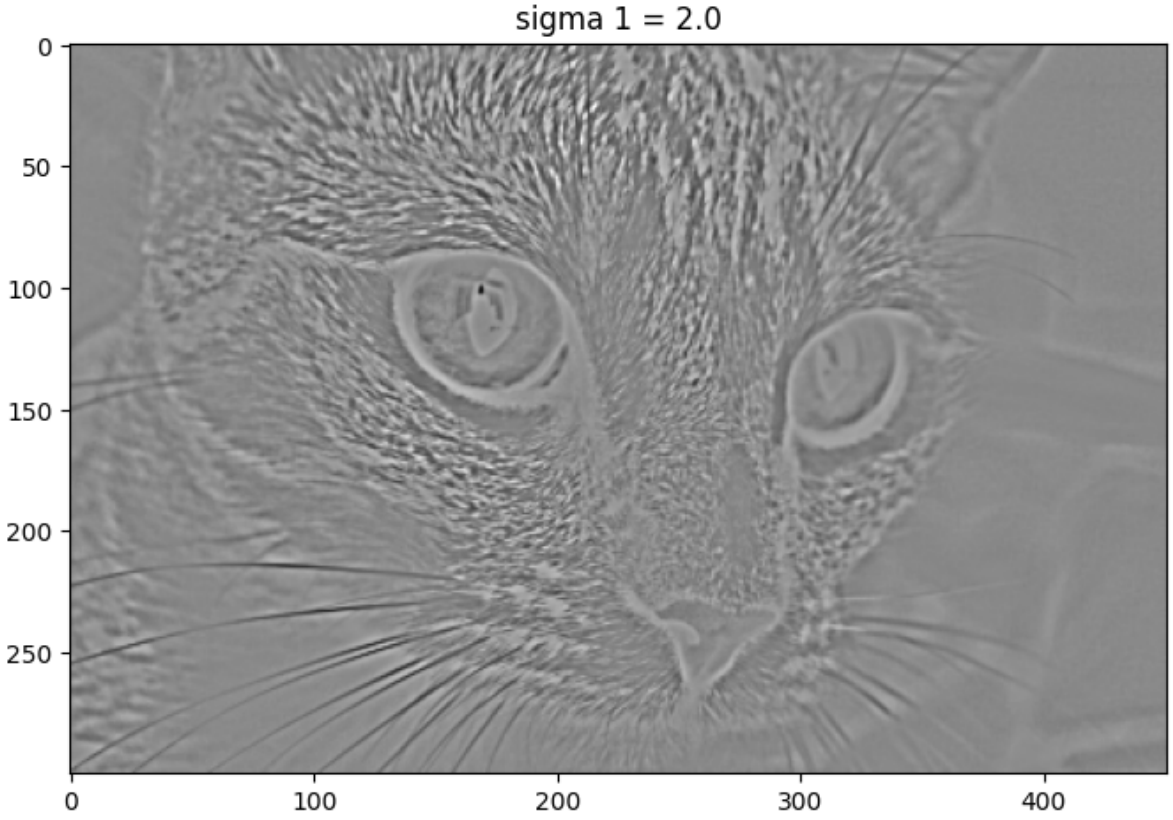
These values are the result of applying the convolution to each point in the series to determine the magniktude and length of the gradient.

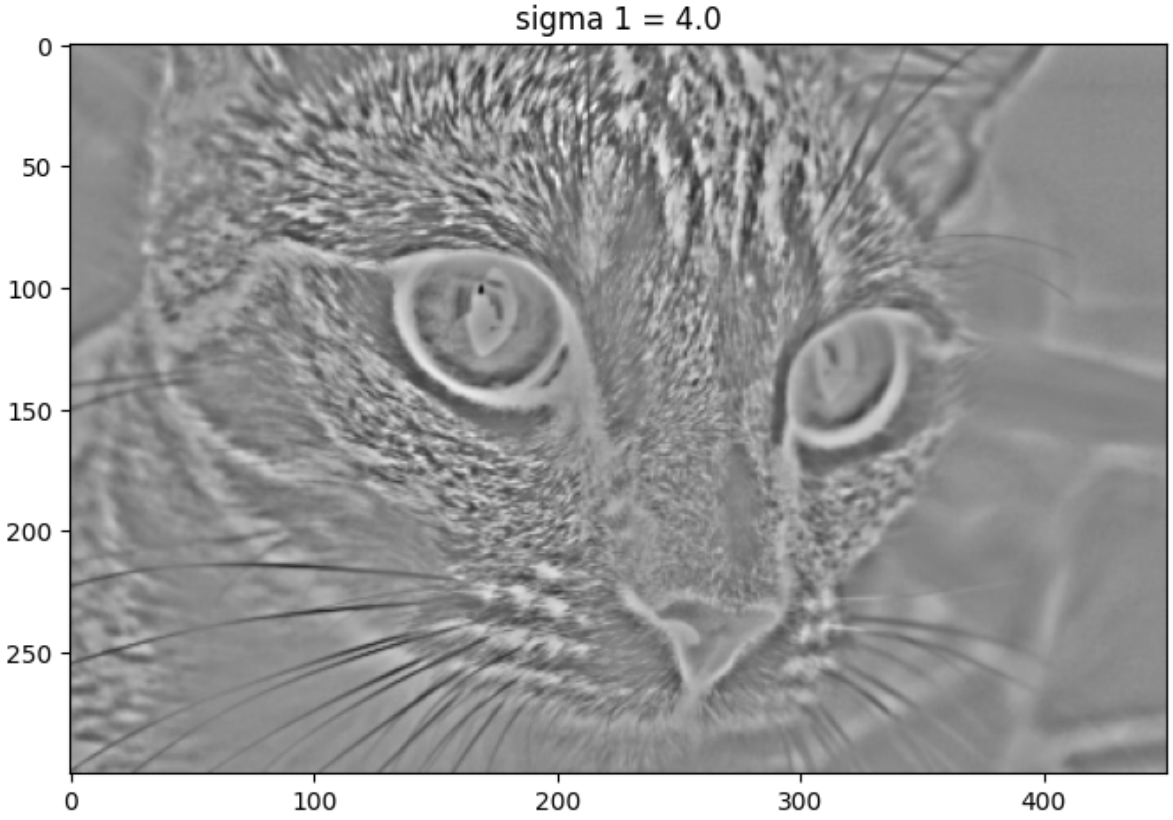
> **Exercise 3-2:** You will now create and test an edge detector using the difference of Gaussian filtered images. Starting with the equalized gray-scale cat image, perform the following steps:
>
> 1. Define a first vector of values of $\sigma_1 = [1.0, 2.0, 4.0, 8.0, 16.0, 32.0, 64.0]$ and a second value of $\sigma_2 = 0.5$.
> 2. For each value of the σ_1 compute an image which is the difference between the Gaussian filtered images computed with σ_1 and σ_2.
> 3. Display each resulting image, with the value of σ_1 in the title.

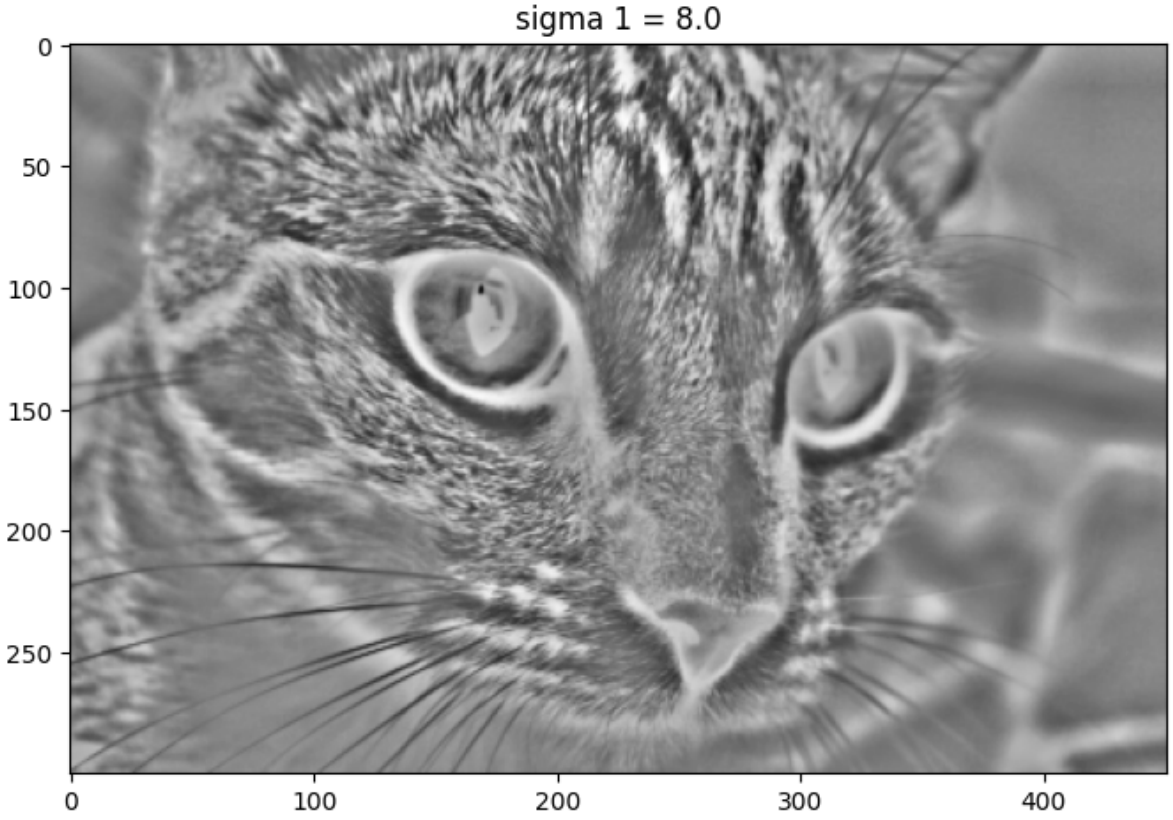
In [ ]:
```
sigma1 = [1.0, 2.0, 4.0, 8.0, 16.0, 32.0, 64.0]
sigma2 = 0.5

##  Your code goes here

# 2. For each value of the sigma_1 compute an image which is the difference
#    filtered images computed with sigma_1 and sigma_2
cat_sigma2 = skfilters.gaussian(cat_grayscale_equalized, sigma = 0.5)
plot_grayscale(cat_sigma2, f"sigma 2 = {sigma2}")

cat_sigma1 = []

for sigma in sigma1:
    cat_sigma1.append(skfilters.gaussian(cat_grayscale_equalized, sigma = si

# 3. Display each resulting image, with the value of $\sigma_1$ in the title
for i in range(len(cat_sigma1)):
    plot_grayscale(cat_sigma1[i], 'sigma 1 = ' + str(sigma1[i]))
```

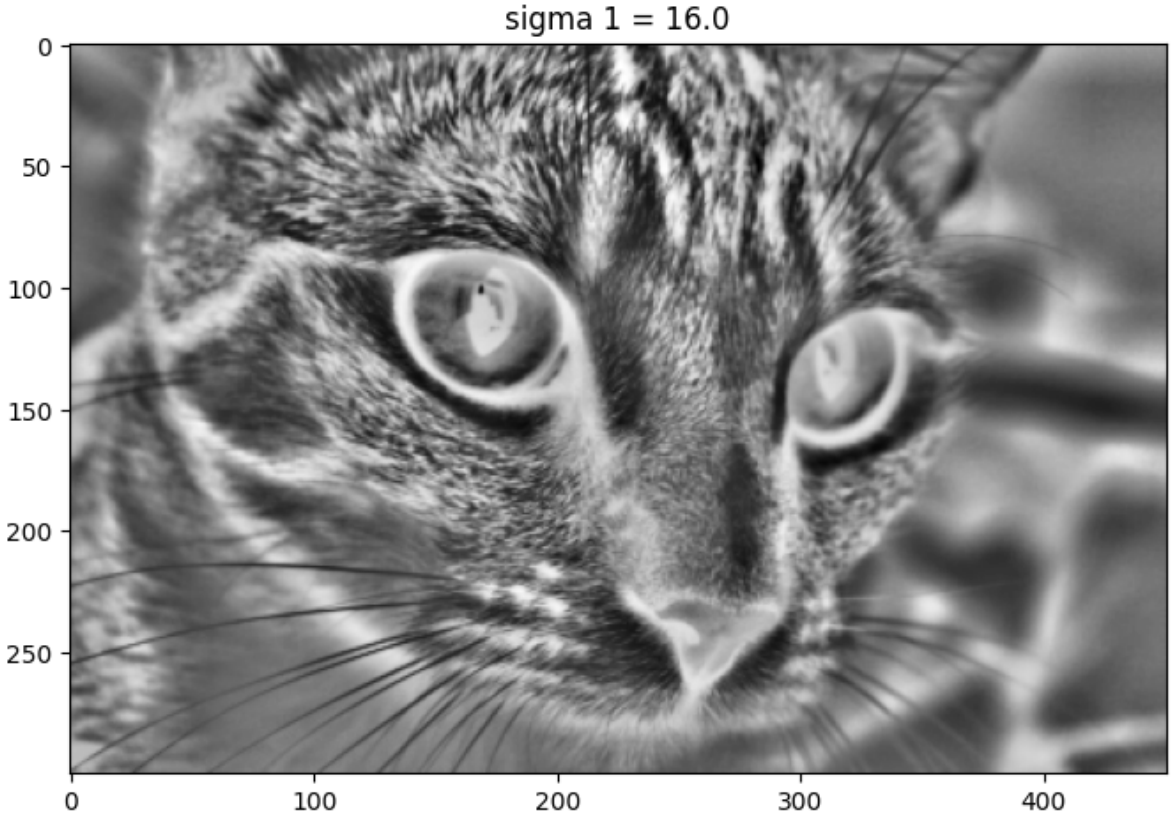

sigma 2 = 0.5
0
50
100
150
200
250
0
100
200
300
400

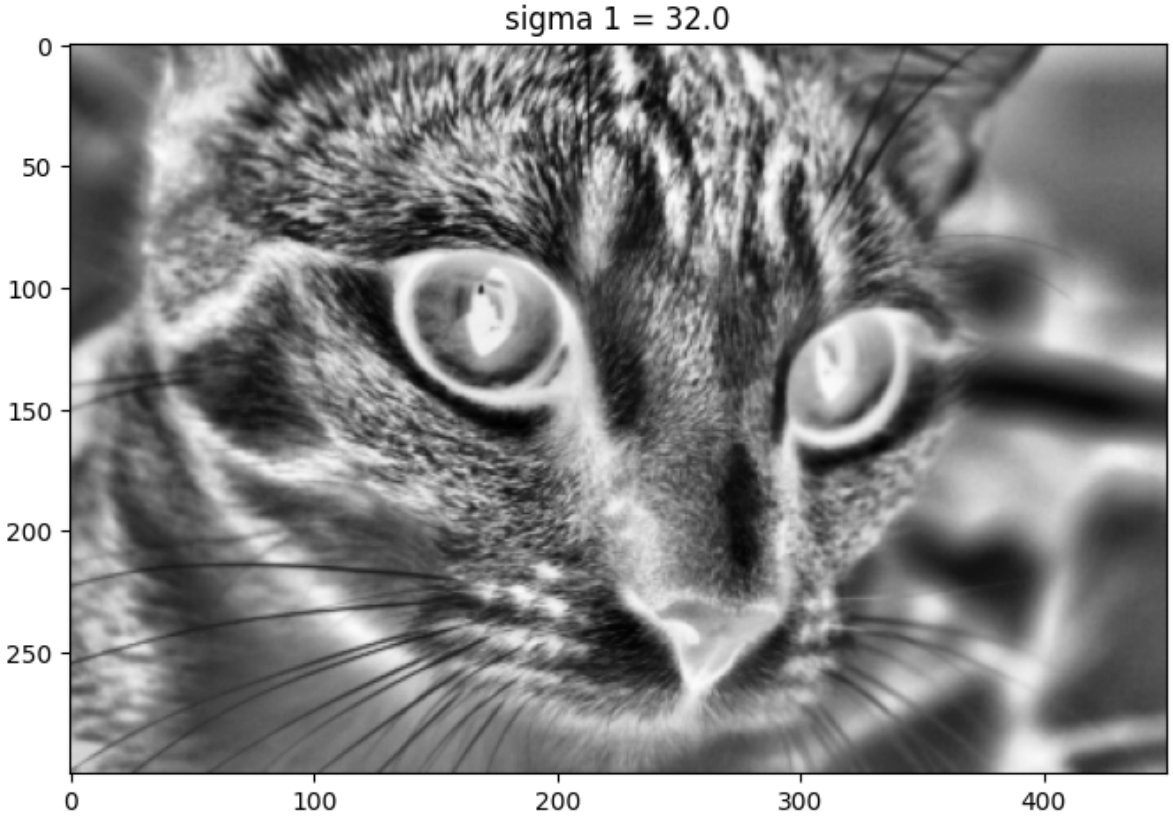

sigma 1 = 1.0
0
50
100
150
200
250
0
100
200
300
400

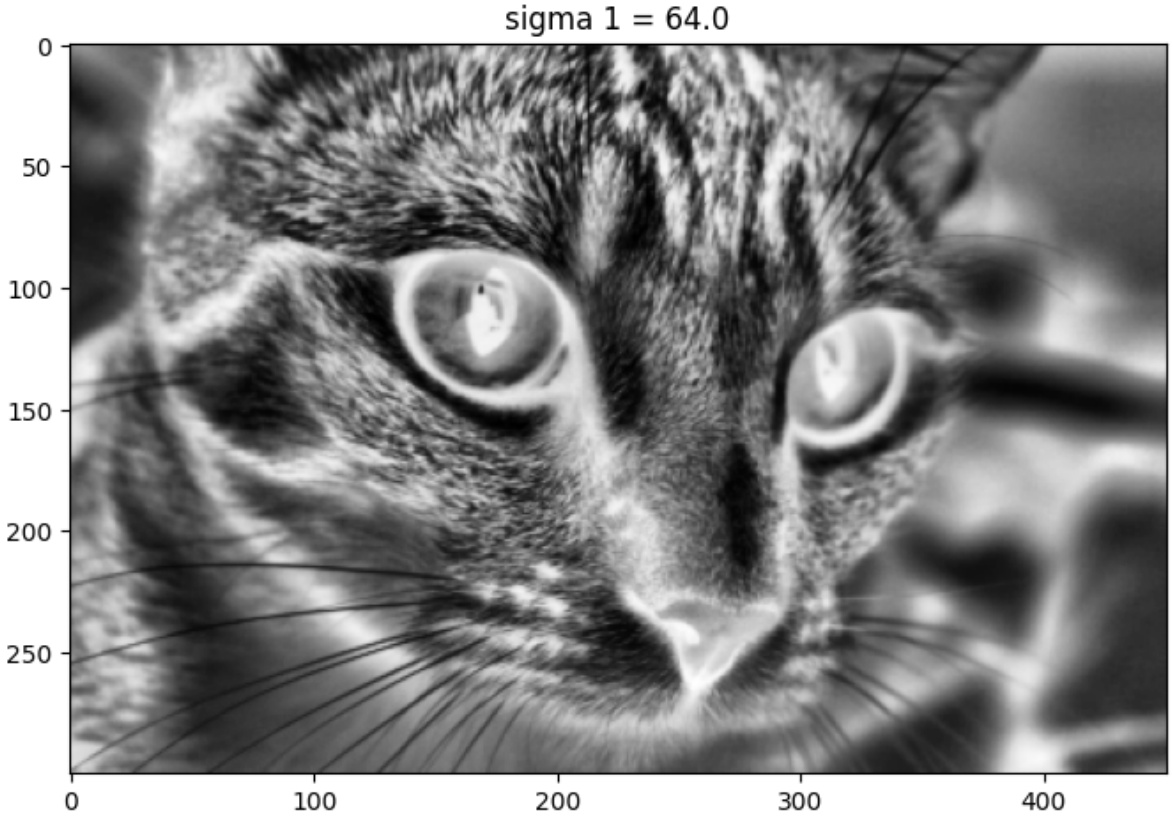

sigma 1 = 2.0
0
50
100
150
200
250
0
100
200
300
400


sigma 1 = 4.0
0
50
100
150
200
250
0
100
200
300
400


sigma 1 = 8.0
0
50
100
150
200
250
0
100
200
300
400

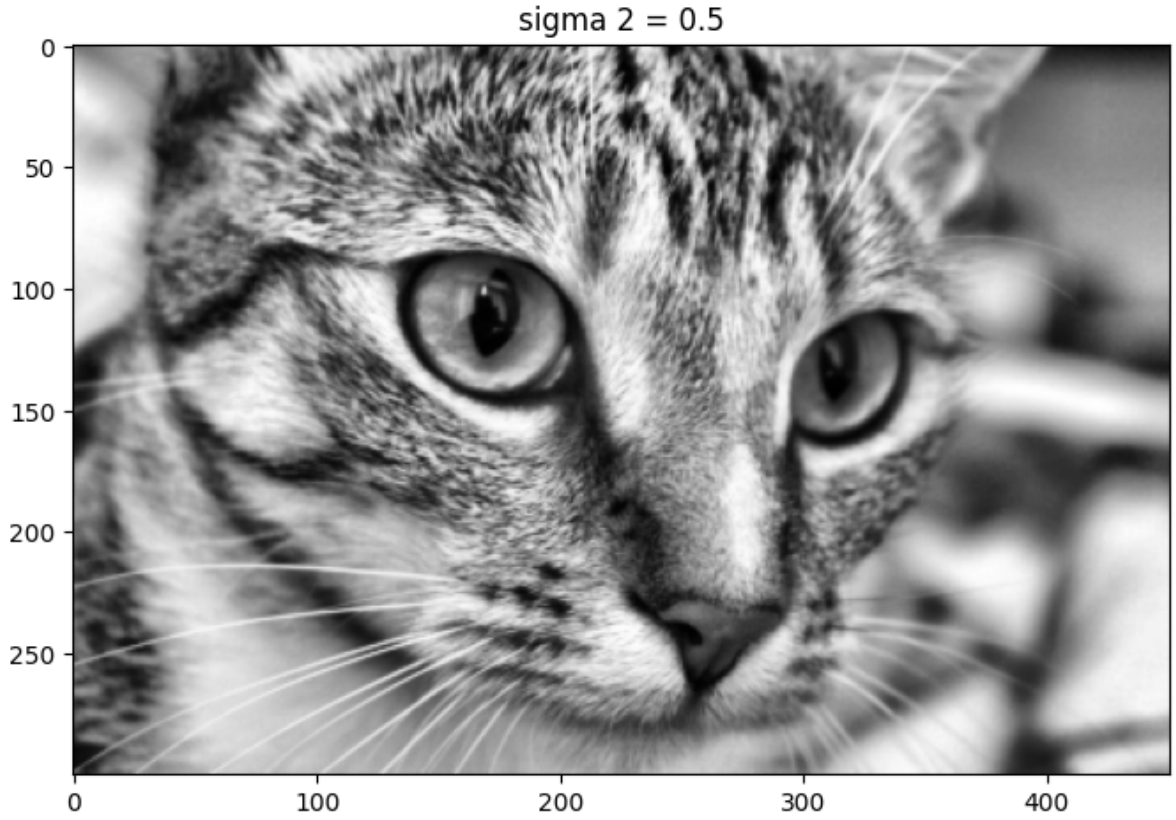

sigma 1 = 16.0
0
50
100
150
200
250
0
100
200
300
400

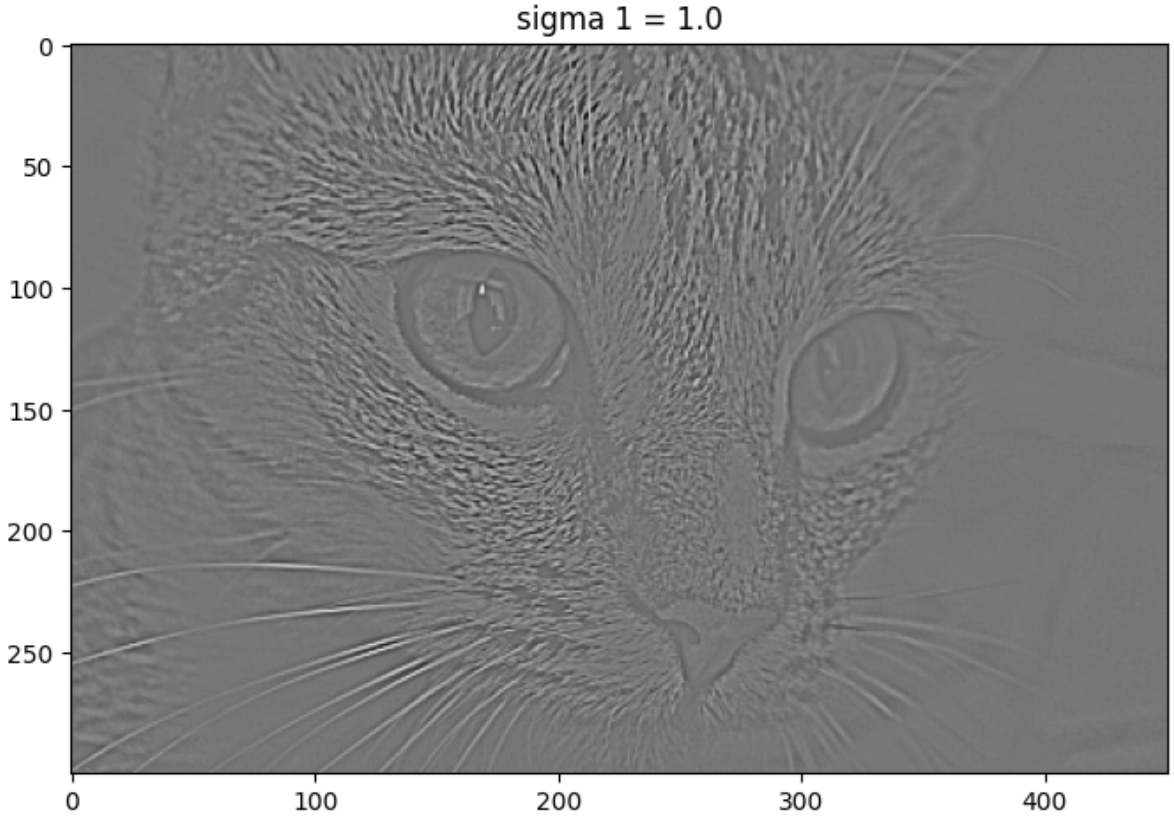


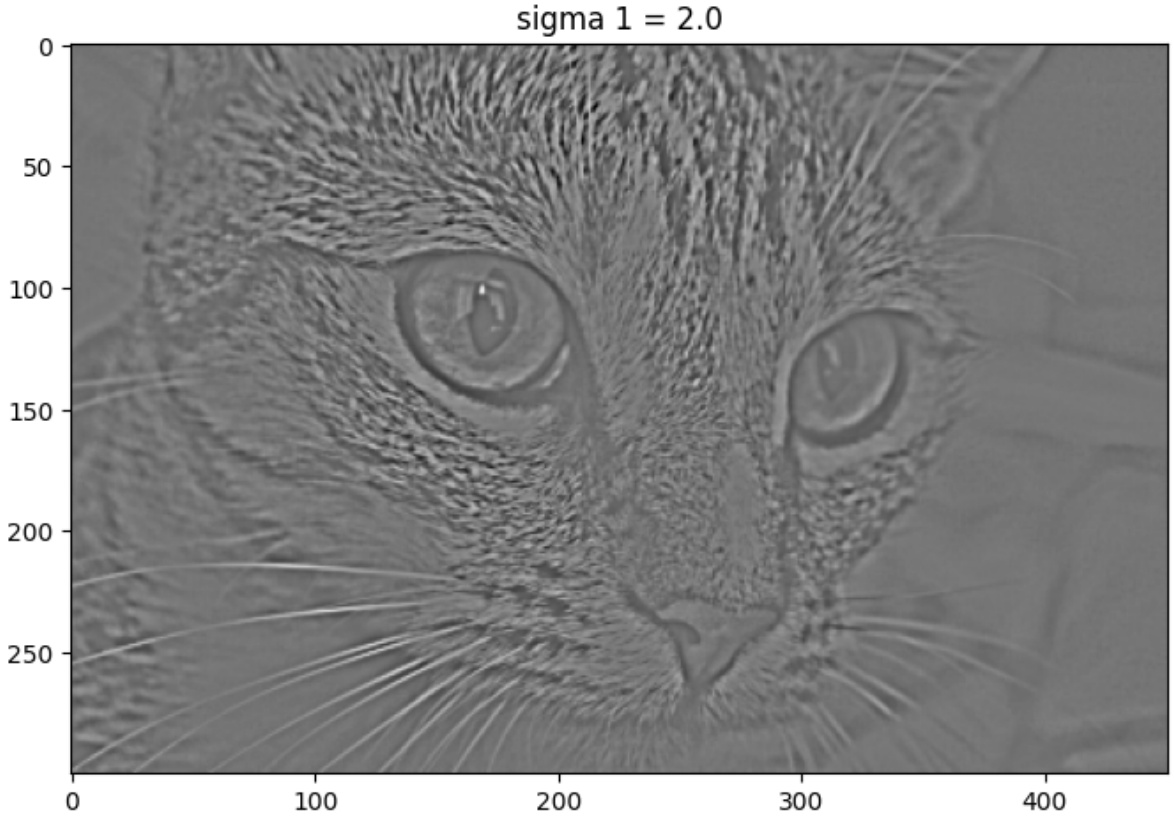


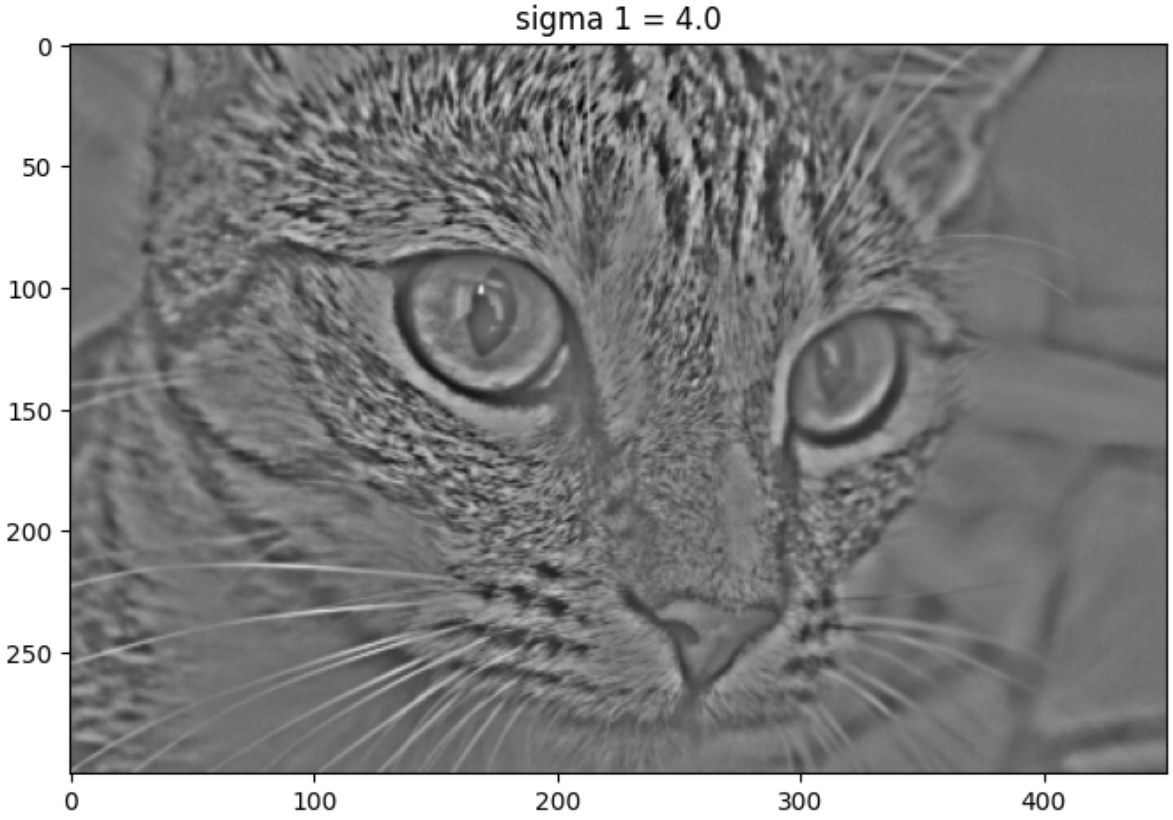
In [ ]:
```
# For experimientation, I also calculated the reversed difference to follow
# said that usually this calculation is done by taking the image with lower

# 2. For each value of the sigma_1 compute an image which is the difference
#    filtered images computed with sigma_1 and sigma_2
```

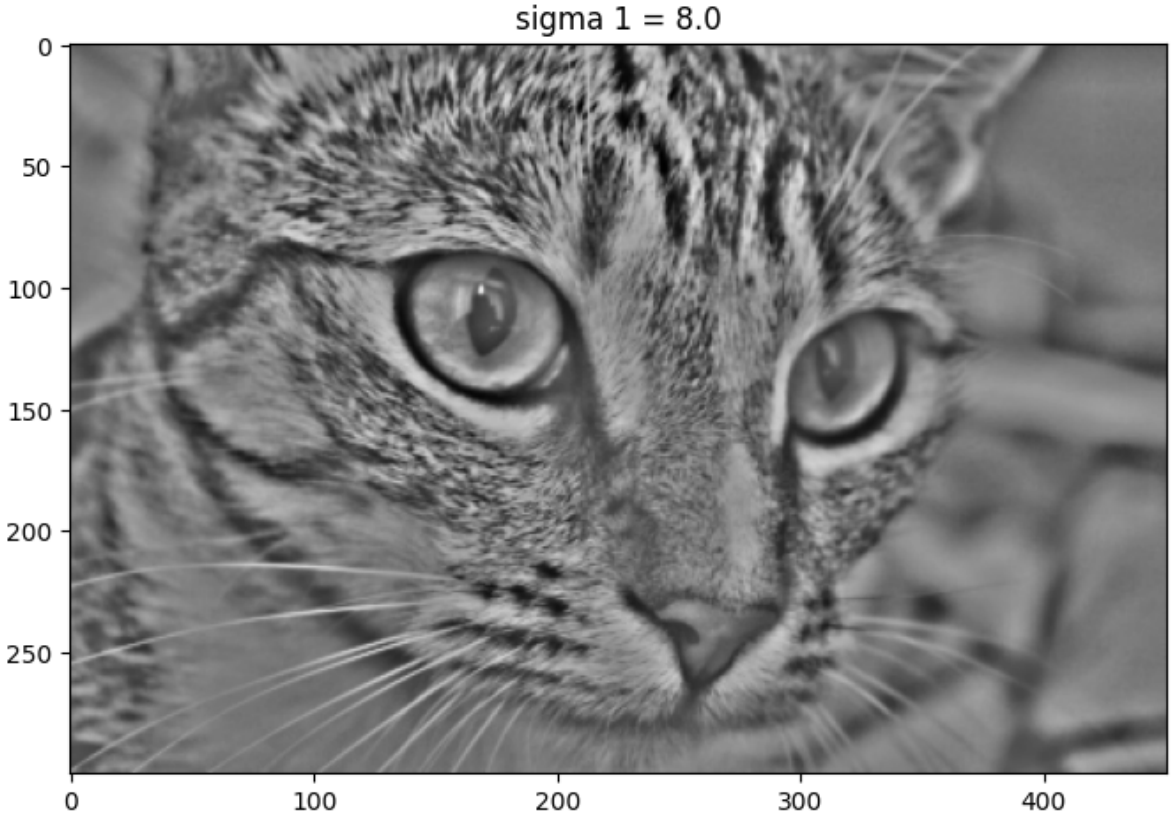
```
cat_sigma2 = skfilters.gaussian(cat_grayscale_equalized, sigma = 0.5)
plot_grayscale(cat_sigma2, f"sigma 2 = {sigma2}")

cat_sigma1 = []

for sigma in sigma1:
    cat_sigma1.append(cat_sigma2 - skfilters.gaussian(cat_grayscale_equalize

# 3. Display each resulting image, with the value of $\sigma_1$ in the title
for i in range(len(cat_sigma1)):
    plot_grayscale(cat_sigma1[i], 'sigma 1 = ' + str(sigma1[i]))
```

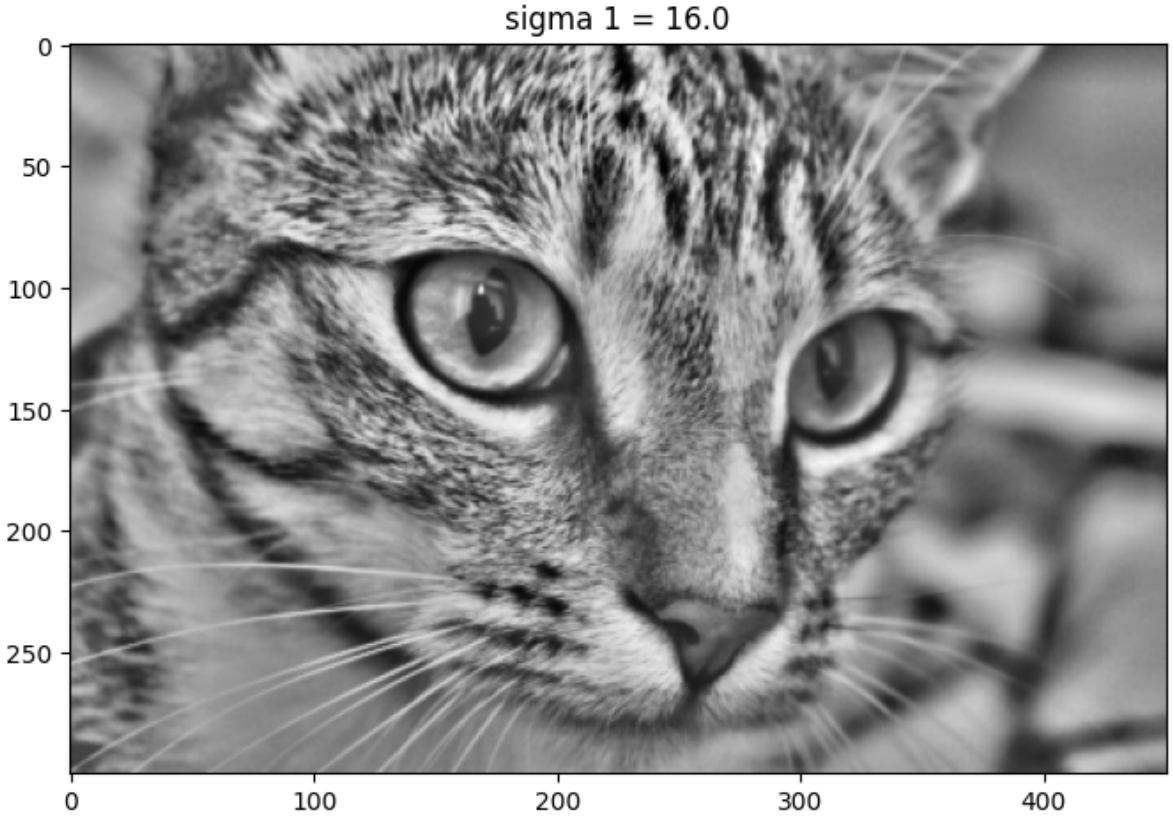


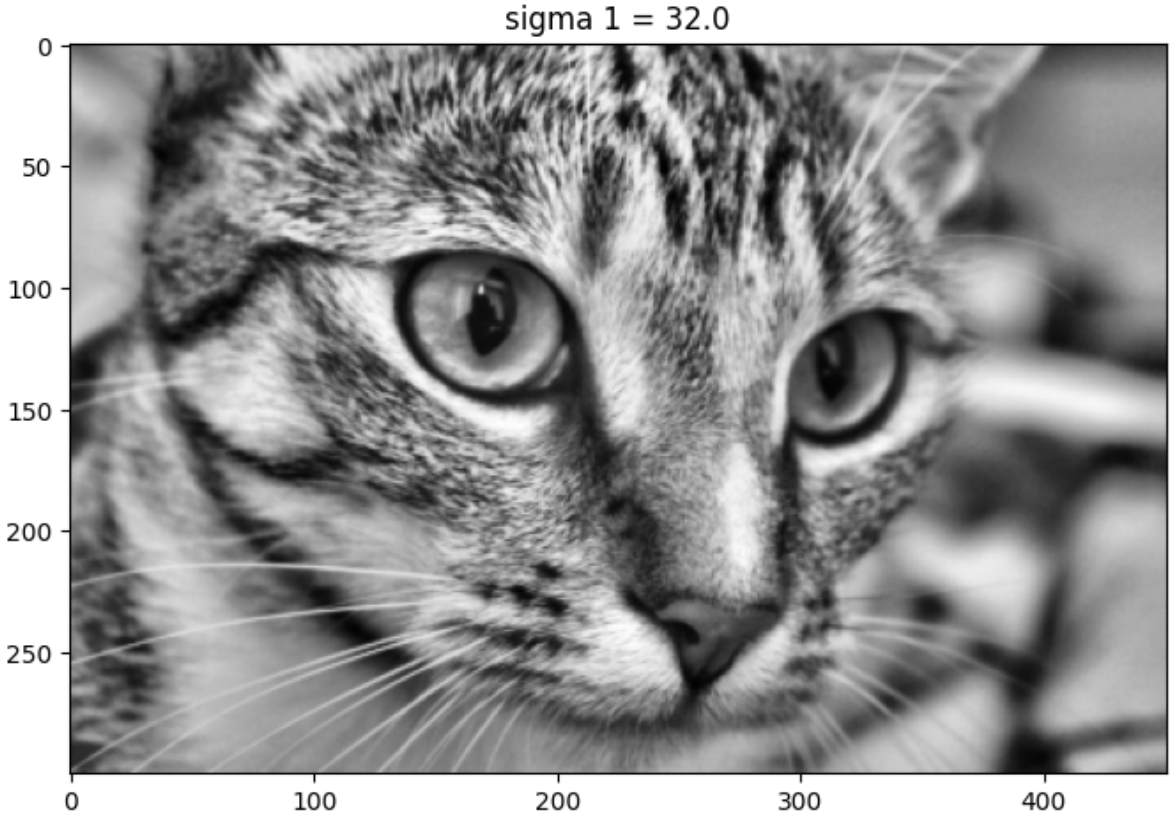

sigma 1 = 1.0
0
50
100
150
200
250
0
100
200
300
400

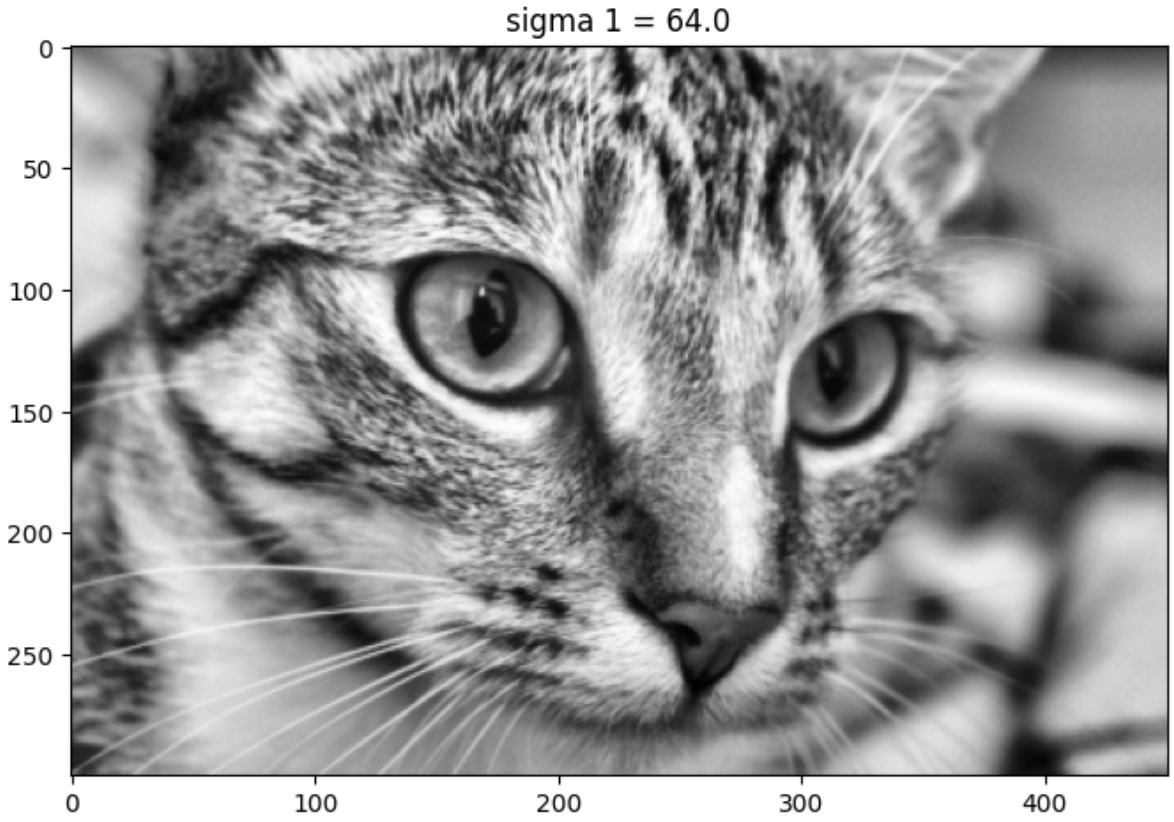

sigma 1 = 2.0
0
50
100
150
200
250
0
100
200
300
400


sigma 1 = 4.0
0
50
100
150
200
250
0
100
200
300
400


sigma 1 = 8.0
0
50
100
150
200
250
0
100
200
300
400


sigma 1 = 16.0
0
50
100
150
200
250
0
100
200
300
400


sigma 1 = 32.0
0
50
100
150
200
250
0
100
200
300
400



> Answer the following questions:
>
> 1. How does the gradient measurement of the images change as the value of σ_1 changes and what does this tell you about the scale of features in the image?
> 2. Given the bandwidths of these Gaussian filters compared to the filter with $\sigma = 0.5$, do the changes of gradient measured make sense and why?
> 3. Does it appear that the difference of Gaussian filters has detected high gradients with different orientations on the image, and why?
>
> **End of exercise.**

> **Answers:**

How does the gradient measurement of the images change as the value of sigma_1 changes and what does this tell you about the scale of features in the image?

1. The gradient measurement of the image changes as the value of sigma changes. As the value of sigma increases, the Gaussian filter becomes more effective and the gradient measurement of the image decreases. This means that the edges of the image become less sharp and more blurry as the sigma value increases. On the other hand, as the sigma value decreases, the Gaussian filter becomes less effective and the gradient measurement of the image increases. This means that the edges of the image become sharper and more defined as the sigma value decreases.

2. Yes, they do make sense. For instance, looking at the image with sigma1 = 1.0 and sigma2 = 0.5 you can see that the edges of the image are well defined but the image has less information that the one with sigma sigma1 = 64 and sigma2 = 0.5. This is because the closer sigma1 and sigma2 only the higher attenuated frequencies remain.

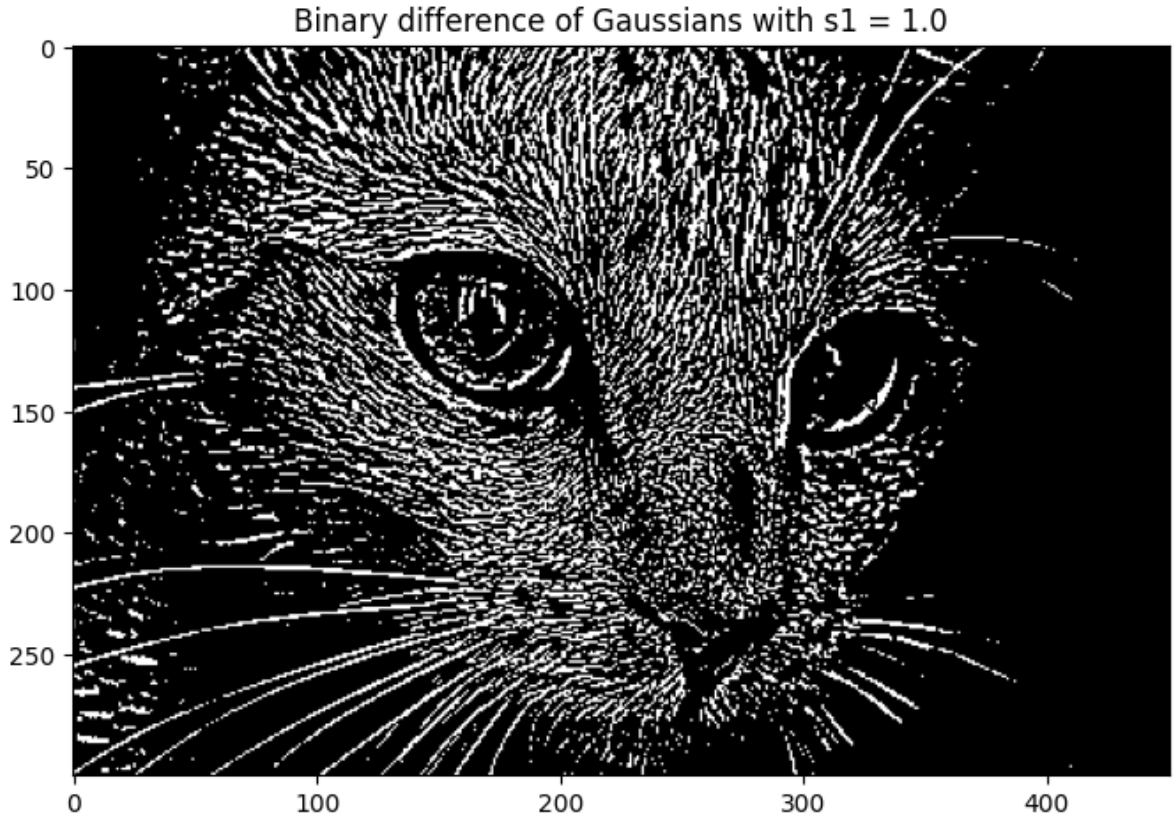
3. Yes, this is because the difference of Gaussian filters is a high-pass filter that detects edges as high gradients with different orientations.

Exercise 3-3: One approach to finding edges is to look for areas of maximum gradient in an image. The difference of Gaussian filters has a strong response to large gradients in image intensity that define edges. In many cases a boolean indicator of an edge is required. To create such an edge indicator from the difference of Gaussian filtered images (equalized gray-scale) with $\sigma_1 = 1.0$ and $\sigma_2 = 0.5$ perform the following steps:
1 Use the skimage.filters.threshold_otsu function to compute a threshold value for the gradient.
2. Apply the threshold and create an image of boolean type.
3. Display the boolean image.

In [ ]:
```
s1 = 1.0

##  Your code goes here
s2 = 0.5

cat_gaussian_difference = skfilters.gaussian(cat_grayscale_equalized, sigma

# 1 Use the skimage.filters.threshold_otsu function to compute a threshold v
threshold = skfilters.threshold_otsu(cat_gaussian_difference)

# 2. Apply the threshold and create an image of boolean type.
print(threshold)
cat_binary_gaussian = cat_gaussian_difference >= threshold
```

```
cat_binary_gaussian = cat_binary_gaussian.astype(bool)

# 3. Display the boolean image.
plot_grayscale(cat_binary_gaussian, 'Binary difference of Gaussians with s1
```

```
0.018776546335513955
```



> Examine the results and answer the following questions:
>
> 1. Has the binarization process captured the edges at all possible orientations, and why?
> 2. Based on your observation, do you think the edge features are sufficient to identify the animal in the image as a cat within reasonable certainty?
>
> **End of exercise.**

> **Answers:**

> 1. Yes, it has. The binarization process is able to do it because it cancels the low frequencies and keeps the high gradients.

> 2. Yes, I do. All main characteristics such as eyes, nose, mouth, whiskers and hair are easily recognizable. It is true however, that the ears and the spots are mainly gone.

Now that you have a bit of experience with edge detection, we will explore a few of the many commonly used edge detection algorithms. As you proceed notice how different algorithms produce different results.

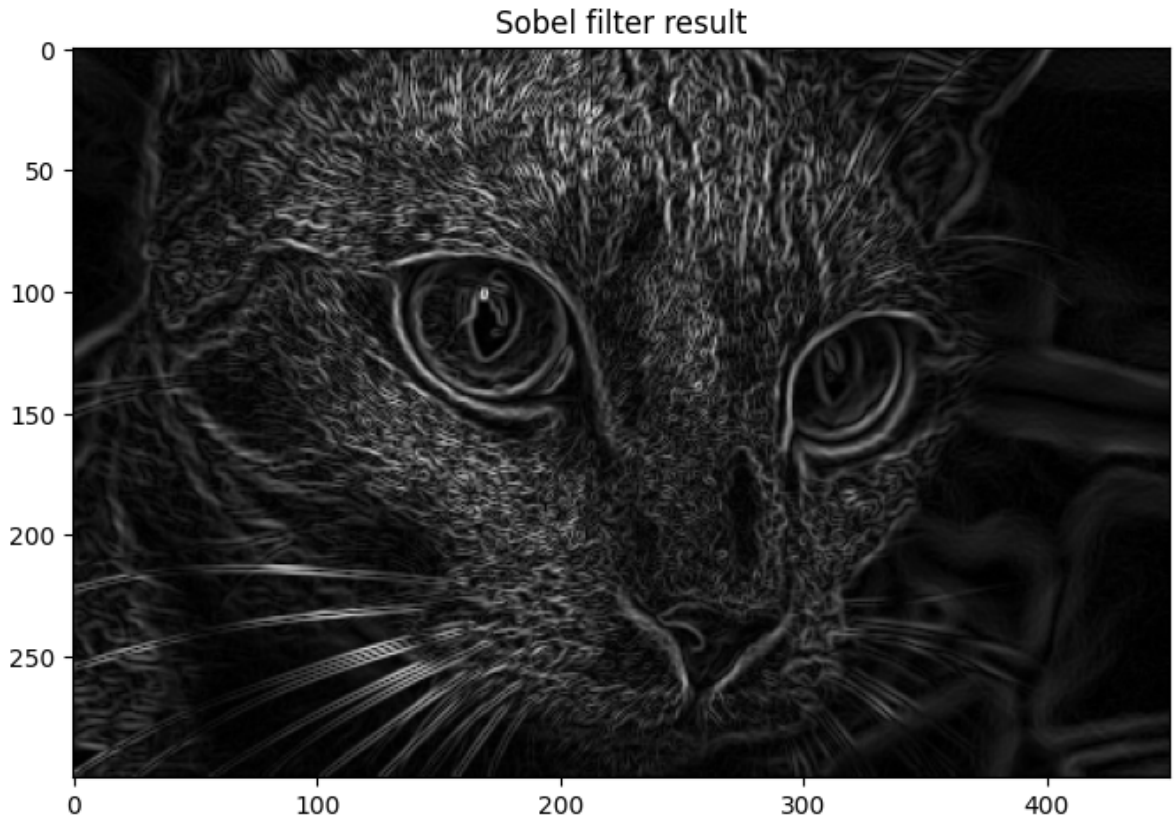
Sobel edge detector

The Sobel edge detector uses the gradients of the pixel values to find edges. Edges are characterized by high gradient magnitude.

Like many feature detectors, a thresholding process must be applied to the gradients found with the Sobel filter. The result is a binary image showing the detected edges. The higher the threshold value, the fewer edge features that will be displayed. This is an example of **nonmaximal suppression**, a widely used method to filter features in computer vision.

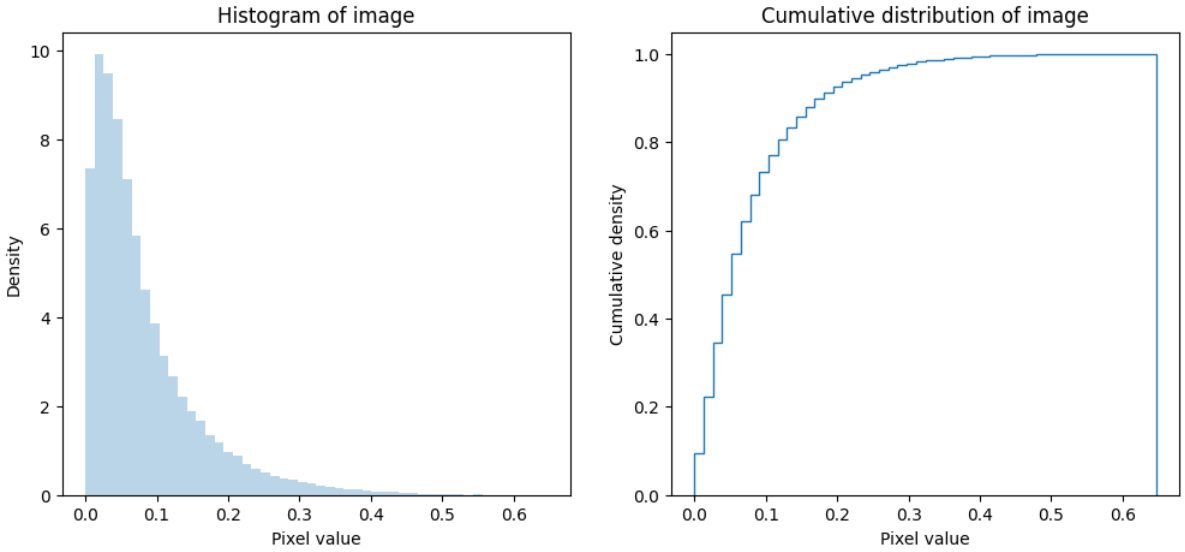
> **Exercise 3-4:** The Sobel filter is another form of edge detector. You will now apply the skimage.filters.sobel function to the equalized gray-scale cat image. In this case you will simply compute the norm and not the directions. Then, display the result.

In [ ]:
```
##  Your code goes here

cat_sobel = skfilters.sobel(cat_grayscale_equalized)

plot_grayscale(cat_sobel, 'Sobel filter result')
```



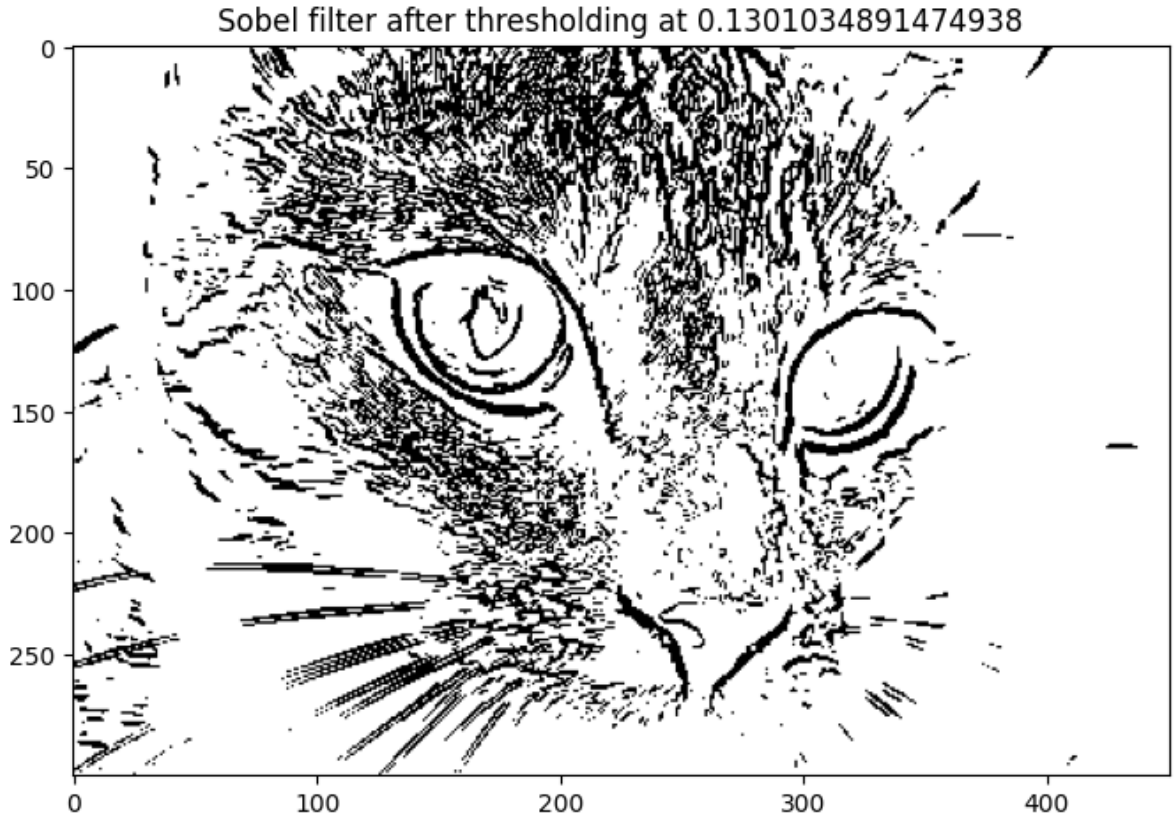
> 3. To examine the density of the Sobel filter output, execute the code in the cell below.

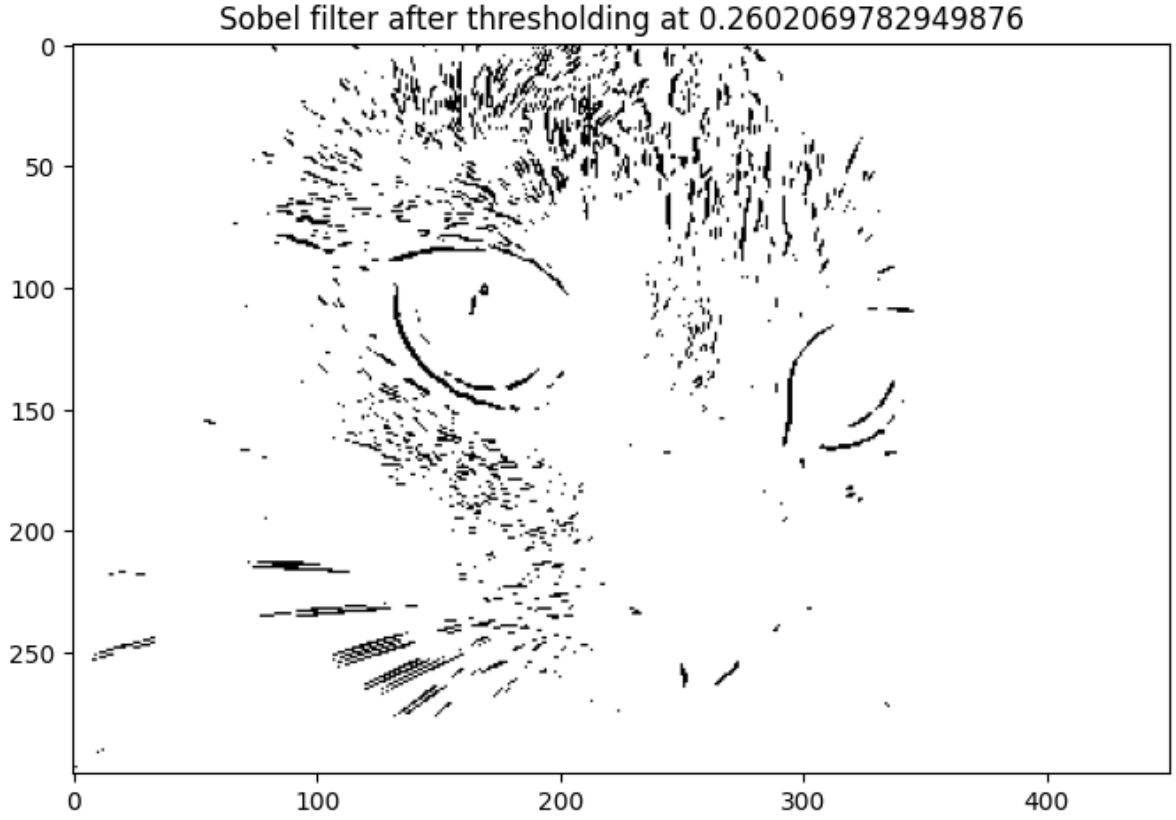
In [ ]:
```
def plot_gray_scale_distribution(img):
    '''Function plots histograms a gray scale image along
    with the cumulative distribution'''
    fig, ax = plt.subplots(1,2, figsize=(12, 5))
    ax[0].hist(img.flatten(), bins=50, density=True, alpha=0.3)
    ax[0].set_title('Histogram of image')
    ax[0].set_xlabel('Pixel value')
    ax[0].set_ylabel('Density')
    ax[1].hist(img.flatten(), bins=50, density=True, cumulative=True, histty
    ax[1].set_title('Cumulative distribution of image')
    ax[1].set_xlabel('Pixel value')
    ax[1].set_ylabel('Cumulative density')
    plt.show()

plot_gray_scale_distribution(cat_sobel)
```

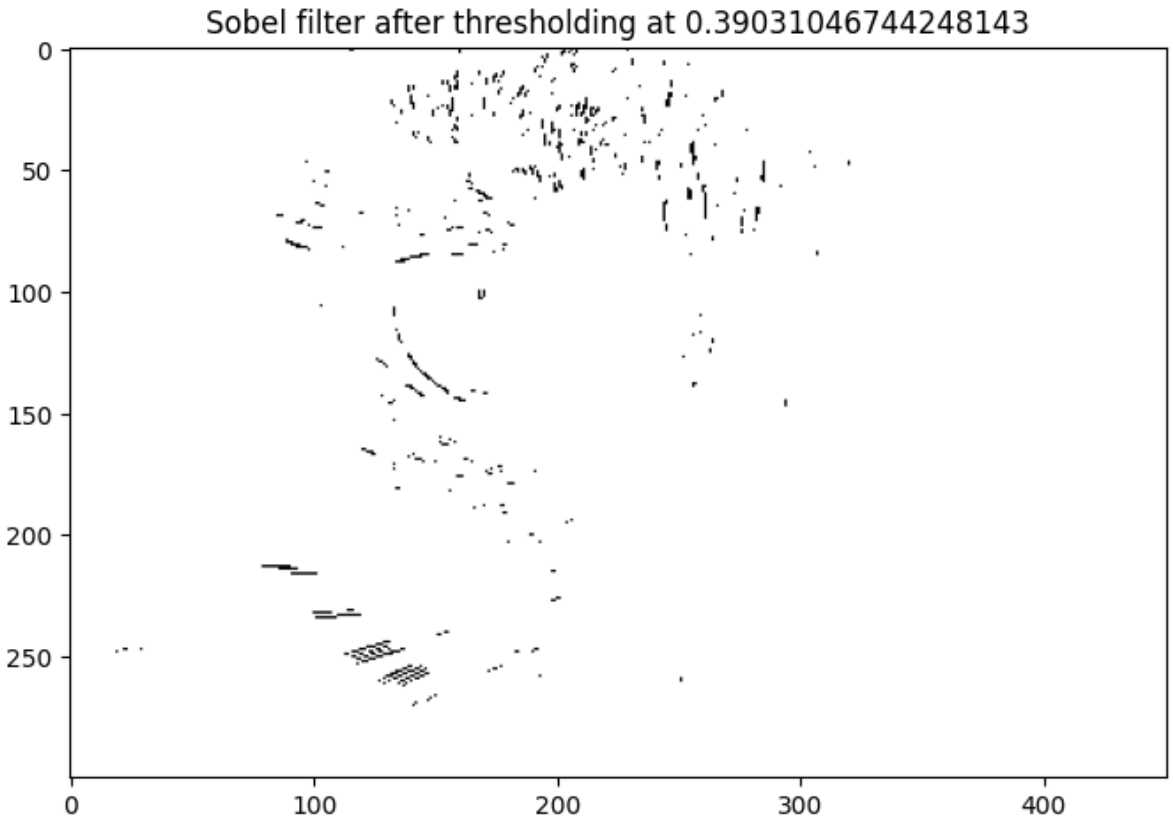


4. Next compute the binary image use the skimage.filters.threshold_otsu function to find a threshold. In a loop itterate over multipliers $[1.0, 2.0, 3.0]$. For each multiplier, apply the product of the multiplier times the threshold found and display the result. Makes sure you display the threshold value for each image. **Hint:** Notice that you can biniarize the image by taking pixels greater than or equal to or less than or equal to the threshold.

In [ ]:
```
##  Your code goes here

threshold = skfilters.threshold_otsu(cat_sobel)

multipliers = [1.0, 2.0, 3.0]

for multiplier in multipliers:
    cat_binary_sobel = cat_sobel <= threshold * multiplier
    plot_grayscale(cat_binary_sobel, "Sobel filter after thresholding at " +
```


Sobel filter after thresholding at 0.1301034891474938
0
50
100
150
200
250
0
100
200
300
400


Sobel filter after thresholding at 0.2602069782949876
0
50
100
150
200
250
0
100
200
300
400



> Compare the edges found with the Sobel filter with those found by the difference of Gaussian filters. Answer the following questions:
>
> 1. Which aspects of the edges found by the two methods are substantially similar?
> 2. How do the edge features detected by the two methods differ, in particular in terms of texture, continuity, compactness, etc?
> 3. How do the edges found with the Sobel filter change with the threshold value, and why does this result make sense?

> **Answers:**

> 1. The edges we get with both methods are similar in the sense that both detect the main characteristics of the cat such as eyes, nose, mouth, whiskers and hair while preserving their shape and orientation.

> 2. The edges found by the Gaussian filter are much smoother, with a texture that is easy to read because it preserves the main features of the original image. the image is continous in all its points and it looks compact. On the other hand, the Sobel filter produces an image that filters out the details to preserve only the maincharacteristics. This results in an image with discontinuous edges making it look less compact.

> 3. As the threshold value increases, only the strongest features remain being captured by the Sobel filter.

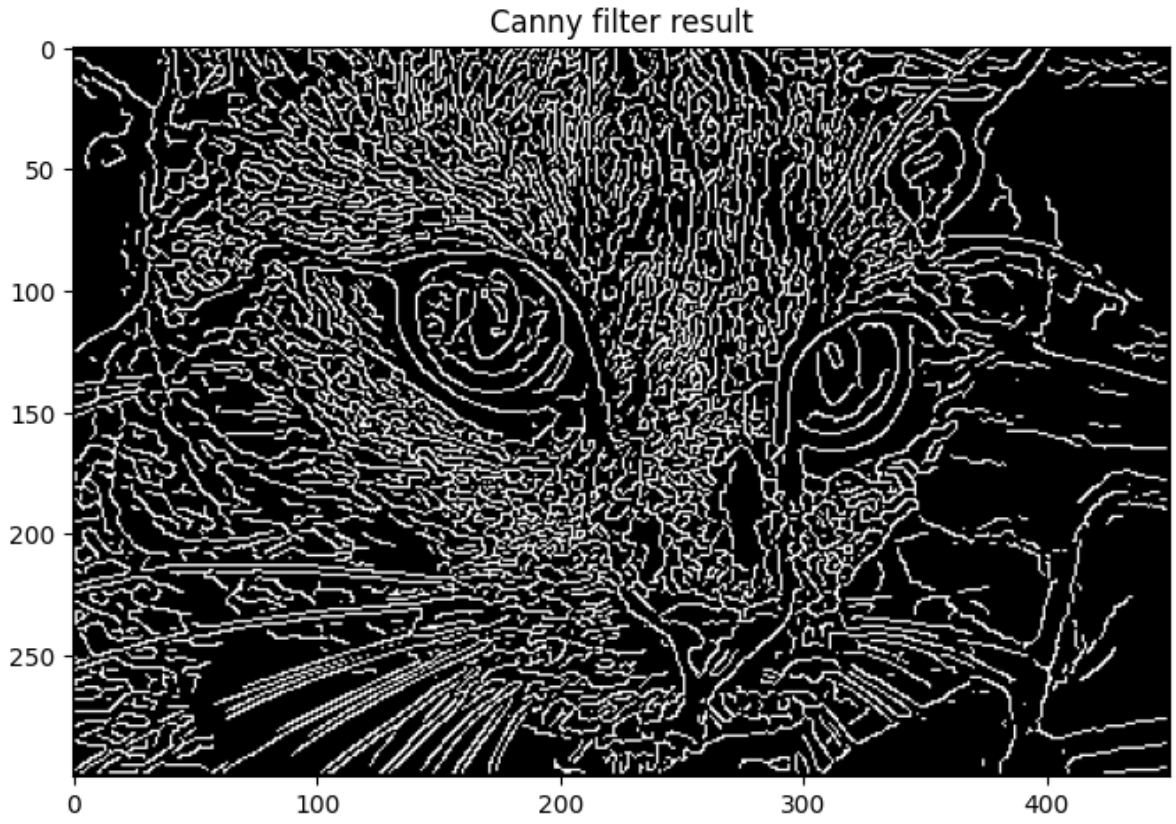
Yes, the result makes sense because the filter removes more and more features as the threshold value increases:

cat_sobel <= threshold * multiplier

Canny edge detector

As a point of comparison, run the code in the cell below to see the results of the Canny filter edge detector.

In [ ]:
```
cat_canny = feature.canny(cat_grayscale_equalized)
plot_grayscale(cat_canny, 'Canny filter result')
```



Compare the edges found with the Canny operator to those found with the Sobel algorithm. Notice that the edges found with Canny are thinner. Many parts of the cat's face, like the facial hair patterns form closed or nearly closed shapes, outlining the edge of each pattern element.

Canny edge detector is a more sophisticated process that uses a Gaussian filter to smooth the image, computes the gradient magnitude and direction, applies non-maximal

suppression, and uses hysteresis thresholding to detect edges. Compared with the Sobel filter, the Canny filter is more robust to noise and produces thinner edges preserving more information than the Sobel filter.

Corner Detection

Along with edges, corners are another fundamental feature of images. Detection of corners is fundamentally more difficult than edges:

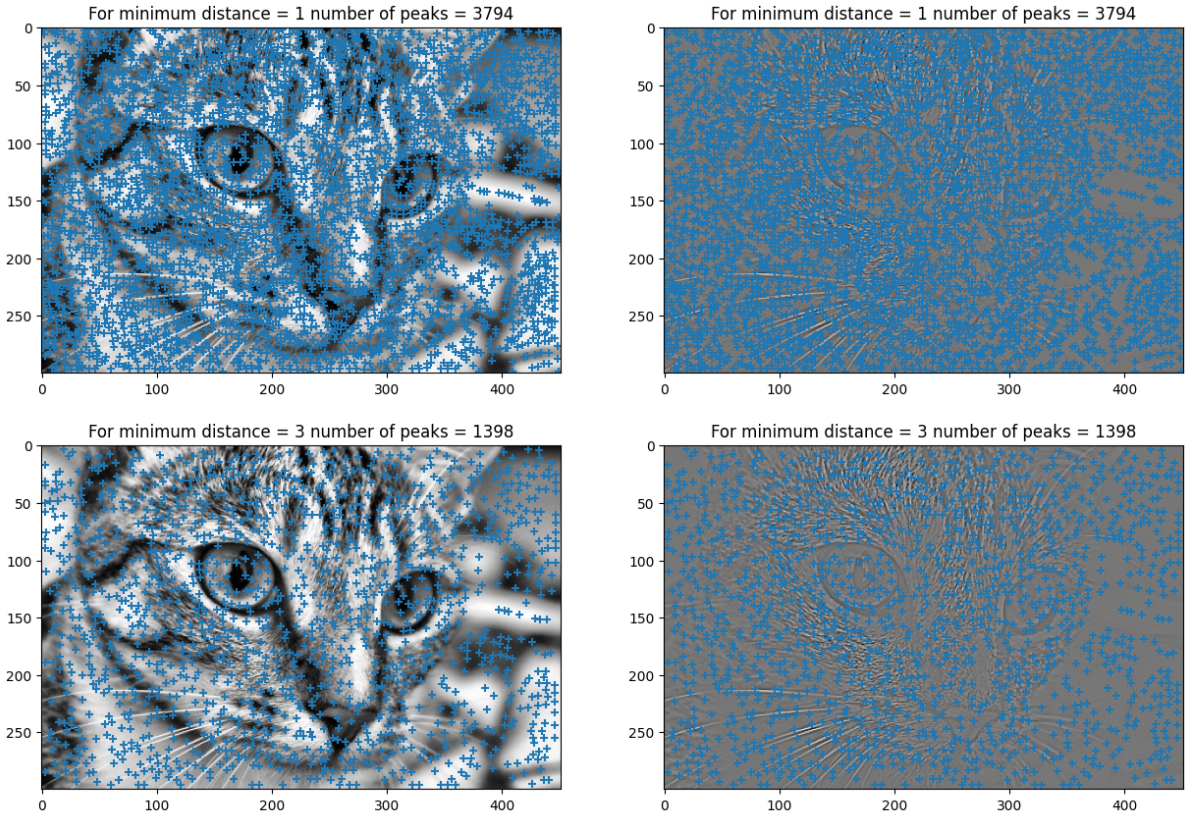
1. Corners are 2-dimensional features, and require more than just a first order gradient for detection.
2. Corners orientation based on the directions of edges forming them.
3. The angel forming the corner is a fundamental characteristic and changing the angel changes the characteristic of the corner.

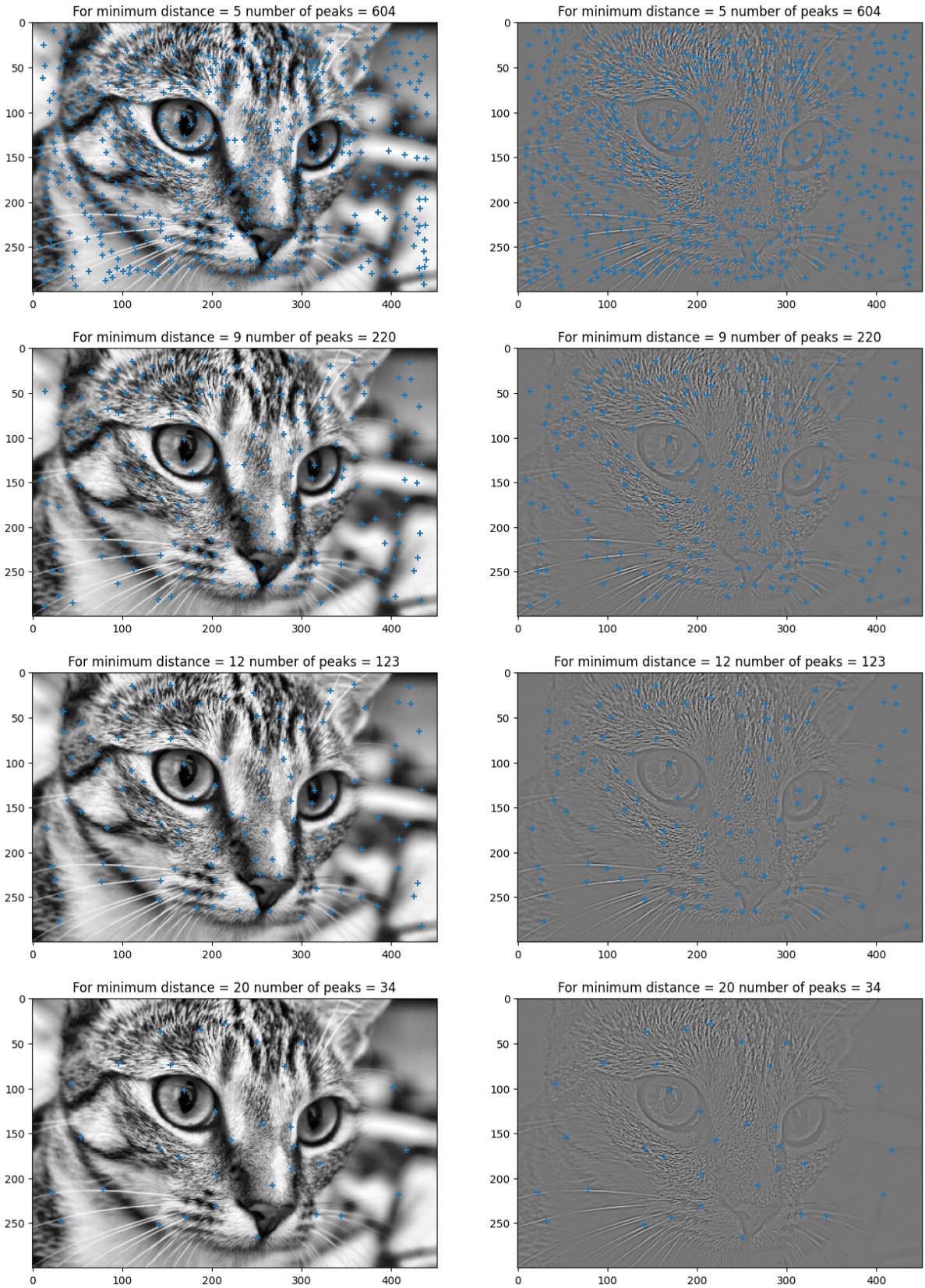
As a result of these fundamental characteristics, corner detectors must be based on metrics of multi-dimensional changes in intensity. As an example, the **Sobel** edge detector is a 2×2 array of all possible second partial derivatives. This formulation allows us to determine both the presence and the orientation (direction) of corners.

> **Exercise 3-5:** The Harris corner detector is one of the many widely use algorithms to detect corner features in images. The detector finds a set of x-y coordinates for each corner that meets a threshold of minimum distance. The threshold is used to perform nonmaximal suppression. Apply the skimage.feature.corner_harris function to the equalized gray scale cat image by the following steps:
>
> 1. Apply the Harris corner detector to the equalized cat gray-scale image.
> 2. Iterate over values of the `min_distance` argument, $[1, 3, 5, 9, 12, 20]$ and then perform tee steps below.
> 3. Use the skimage.features.corner_peaks to filter the corner features using the `min_distance` value.
> 4. Print the minimum distance and the number of corner features the filtering.
> 5. Side by side, plot the gray-scale and the gradient image computed with the difference of Gaussian filtered images with $\sigma_1 = 2.0$ and $\sigma_2 = 0.5$. Superimpose on both images the locations of the corners detected using a '+' marker.

In [ ]:
```
##  Your code goes here

# 1. Apply the Harris corner detector to the equalized cat gray-scale image.
```

```
cat_harris = feature.corner_harris(cat_grayscale_equalized)

# 2. Iterate over values of the `min_distance` argument, [1, 3, 5, 9, 12, 20
min_distances = [1, 3, 5, 9, 12, 20]

for distance in min_distances:
    # 3. Use the skimage.features.corner_peaks to filter the corner features
    cat_harris_peaks = feature.corner_peaks(cat_harris, min_distance = dista

    # 4. Print the minimum distance and the number of corner features the fi
    # 5. Side by side, plot the gray-scale and the gradient image computed w
    #    images with sigma_1 = 2.0 and sigma_2 = 0.5. Superimpose on both im
    #    using a '+' marker.
    fig, ax = plt.subplots(1, 2, figsize = (15, 15))
    _ = ax[0].imshow(cat_grayscale_equalized, cmap = plt.get_cmap('gray'))
    _ = ax[0].scatter(cat_harris_peaks[:, 1], cat_harris_peaks[:, 0], marker
    _ = ax[0].set_title('For minimum distance = ' + str(distance) + ' number
    _ = ax[1].imshow(cat_gaussian_difference, cmap = plt.get_cmap('gray'))
    _ = ax[1].scatter(cat_harris_peaks[:, 1], cat_harris_peaks[:, 0], marker
    _ = ax[1].set_title('For minimum distance = ' + str(distance) + ' number

    plt.show()
```





> Examine the results and images and answer these questions:
>
> 1. How can you describe the change in the number of corner features as the minimum distance is increased, and why?

> 2. Do the features that remain as the minimum distance increases appear more robust (or obviously corners to the eye) given the gradient computed with the difference of Gaussians?
> 3. Consider a trade-off between using a large set of features which may better represent a specific image vs. a sparser but more robust set of features. How might you decide how to optimally filter the features?

> **Answers:**

> 1. The minimum distance parameter in the corner_peaks function is used to filter out the features that are too close to each other. As the minimum distance increases, the number of features decreases because the filter removes the features that are too close to each other.

> 2. Not necessarily. Given the distance restriction, many of the most obvious corners are removed.

> 3. Probably by assigning a score to the features and then applying the filter in a way that starts filtering the lower scores and keeping the highest ones. The score could be based on the gradient of the features and any other characteristic that help to determine their robustness.

Eigenvalues and Corner Detection

Now that you have worked with a corner detection algorithm, you will create your own using eigenvalues. This algorithm is a simplified version of the well-known SIFT algorithm. The algorithm uses the ratio of the eigenvalues to find corners with changes in gradient. The steps of this algorithm are:

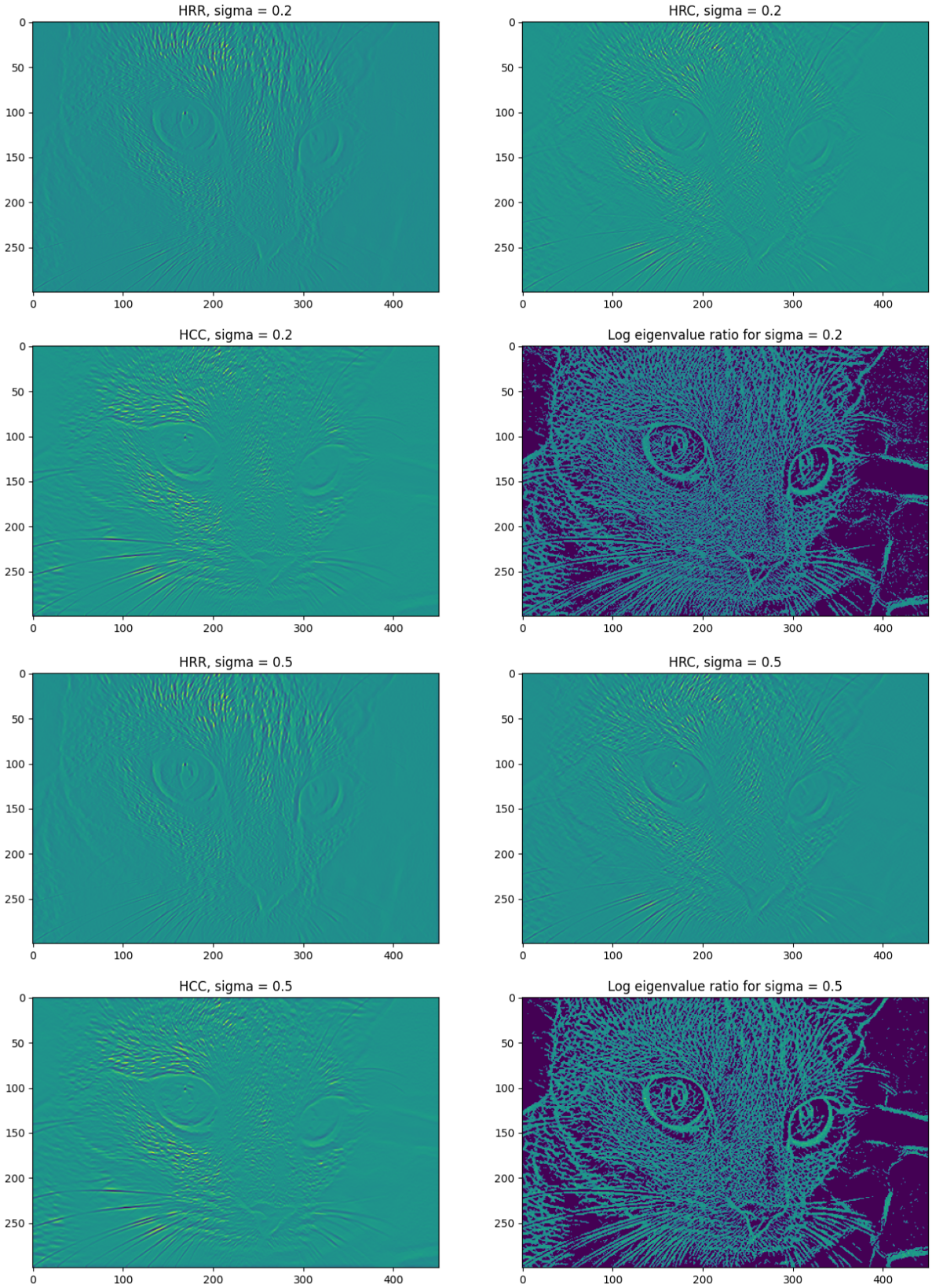
1. The Hessian for each pixel in the image is computed using the operator span specified, σ.
2. The eigenvalues of the Hessians are computed.
3. The ratio of the largest to the second eigenvalue is computed.
4. The eigenvalue ratio is thresholded to reduce the number of features.

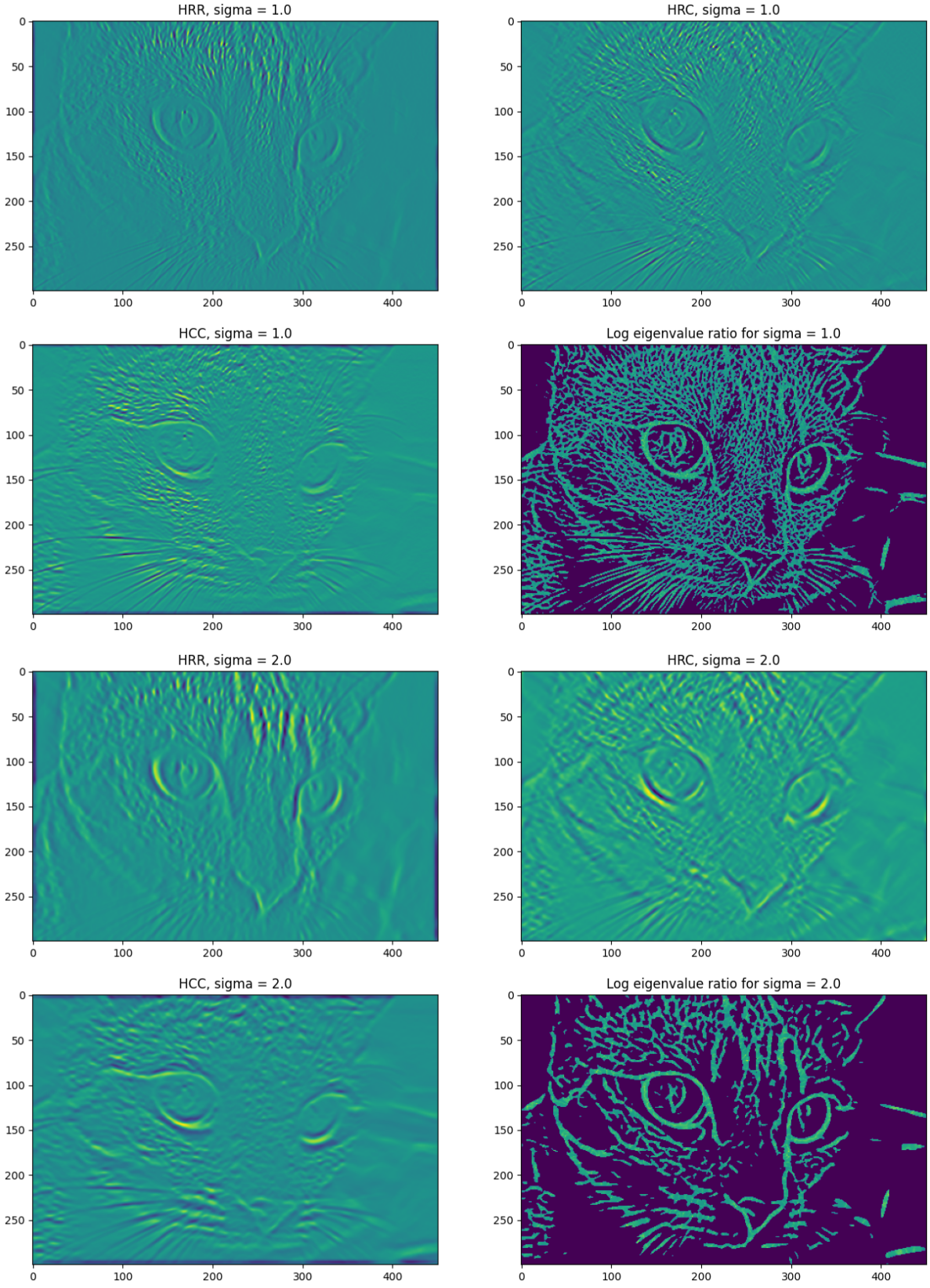
> **Exercise 3-6:** You will now implement the corner detection and examine the results by the following steps:
>
> 1. Compute the Hessian of the equalized gray-scale cat image using skimage.feature.hessian_matrix, iterated over values of $\sigma = [0.2, 0.5, 1.0, 2.0, 4.0]$.

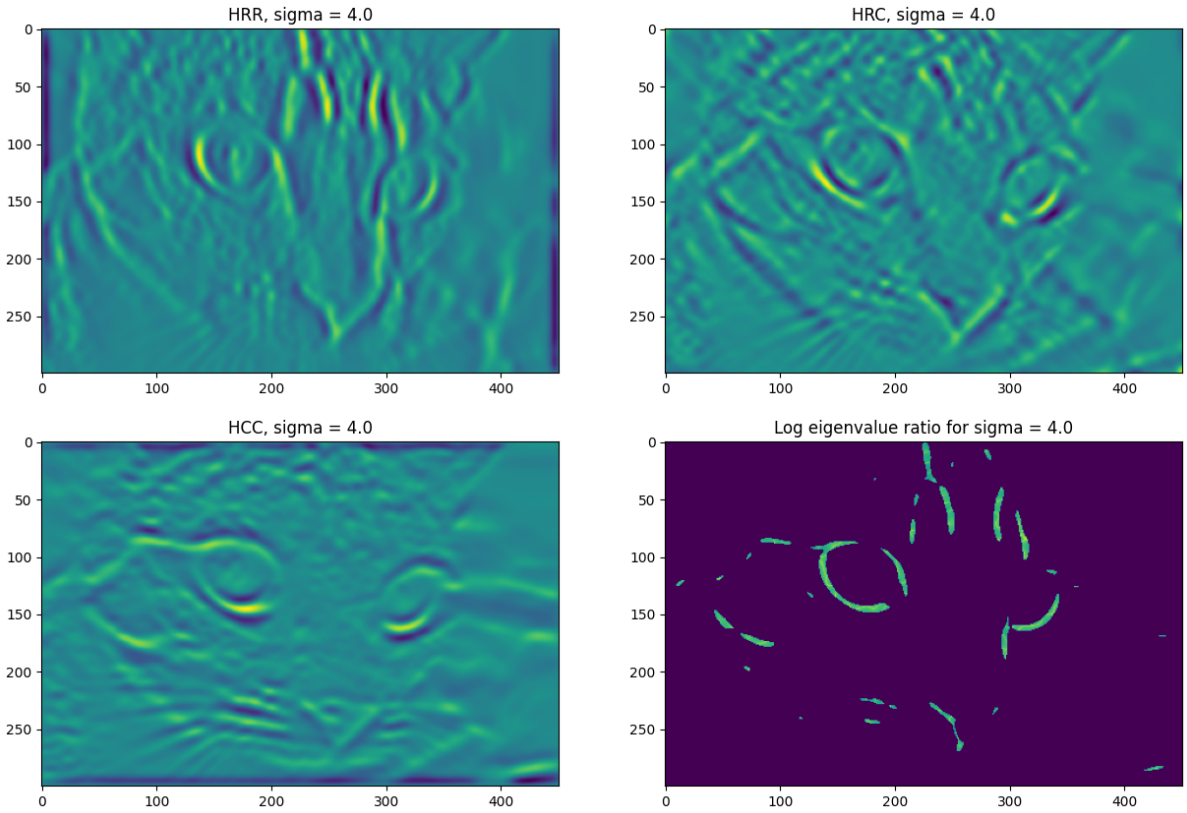
2. For each Hessian compute the eigenvalues using skimage.feature.hessian_matrix_eigvals.
3. Compute the ratio of the magnitudes or absolute values of the first and second eigenvalues. These values can be found in the first (largest) and second elements of the list returned by the `hessina_matrix_eigvals` function.
4. Apply a threshold of 0.01 to the largest largest eigenvalues. If the larges eigenvalue is less than the threshold set the eigenvalue ratio to 10^{-6}.
5. Display images for the each of the 3 components of the Hessian, $[HRR, HRC, HCC]$, along with the image of the log of the thresholded eigenvalue ratio. Make sure to label all images with the operator span, each of the Hessian components, or log eigenvalue ratio.

In [ ]:
```
sigma_list = [0.2,0.5,1.0,2.0,4.0]
threshold = 0.01

##  Your code goes here

# 1. Compute the Hessian of the equalized gray-scale cat image using skimage
#    iterated over values of sigma = [0.2,0.5,1.0,2.0,4.0]
hessian = {}
hessian = {s: feature.hessian_matrix(cat_grayscale_equalized, sigma = s) for

# 2. For each Hessian compute the eigenvalues using skimage.feature.hessian_
eigenvalues = {k: feature.hessian_matrix_eigvals(hessian[k]) for k in hessia

# 3. Compute the ratio of the magnitudes or absolute values of the first and
#    These values can be found in the first (largest) and second elements of
#    the `hessina_matrix_eigvals` function.

cnt = 0
for k in eigenvalues:
    eig_ratio = np.where(eigenvalues[k][0] < threshold, 10e-6,
                np.abs(eigenvalues[k][0]) / np.abs(eigenvalues[k][1]))

    sigma = sigma_list[cnt]
    cnt += 1

    fig, ax = plt.subplots(2, 2, figsize = (15, 10))
    ax = ax.flatten()
    ax[0].imshow(hessian[k][0])
    ax[0].set_title('HRR, sigma = ' + str(sigma))
    ax[1].imshow(hessian[k][1])
    ax[1].set_title('HRC, sigma = ' + str(sigma))
    ax[2].imshow(hessian[k][2])
    ax[2].set_title('HCC, sigma = ' + str(sigma))
    _ = ax[3].imshow(np.log(eig_ratio)) #, cmap=plt.get_cmap('gray')
    _ = ax[3].set_title('Log eigenvalue ratio for sigma = ' + str(sigma))
```

```
    _ = plt.show()
#hessian
```




HRR, sigma = 1.0
HRC, sigma = 1.0
HCC, sigma = 1.0
Log eigenvalue ratio for sigma = 1.0
HRR, sigma = 2.0
HRC, sigma = 2.0
HCC, sigma = 2.0
Log eigenvalue ratio for sigma = 2.0



> Answer the following questions:
>
> 1. How can you describe the how the components of the Hessian change as the span of the operator, σ, changes and why?
> 2. What differences can you see between the components of the Hessian and why? *Hint*, pay careful attention to the orientations of the detected features.
> 3. What differences do you see in the feature image as the span of the Hessian operator, σ, changes and why?
>
> **End of exercise**.

> **Answers:**
>
> 1. The Hessian components change as the span of the operator sigma changes because the Hessian of a pixel is a matrix that contains all second-order derivatives computed for that pixel. As the span of the operator increases, the second order derivatives are more smoothed out.

> 2. The Hessian components are the second-order derivatives of the image in the x, y, and xy directions. Note that HRR (second partial derivative with respect to x) is more sensitive to vertical features and HCC (second partial derivative with respect to y) is more sensitive to

> horizontal features. From here we get that as the span of the operator increases, the second order derivatives are more smoothed out, which produces the removal of less notorious features.

> 3. I already discussed that the feature image is more smoothed out as the span of the operator increases. From the eigenvalue perspective we can see that the increment of sigma produces a reduction of the gradients. This is what makes just the more notorious features to be detected but at the same time be well defined.

Interest Point Descriptors

In many computer vision applications, including image stitching, stereo vision, and flow (motion tracking), require the unique identification of **interest points** or **key points**.

The HOG Algorithm

The orientation of corners can be used as part of a representation or feature map of an image. As with corner detection, a great many algorithms for determining corner orientation have been developed. Here, we will only work with the **HOG** or **histograms of oriented gradients** algorithm. The algorithm finds the directional bin with maximum value for a histogram of the gradients over patches of the image.

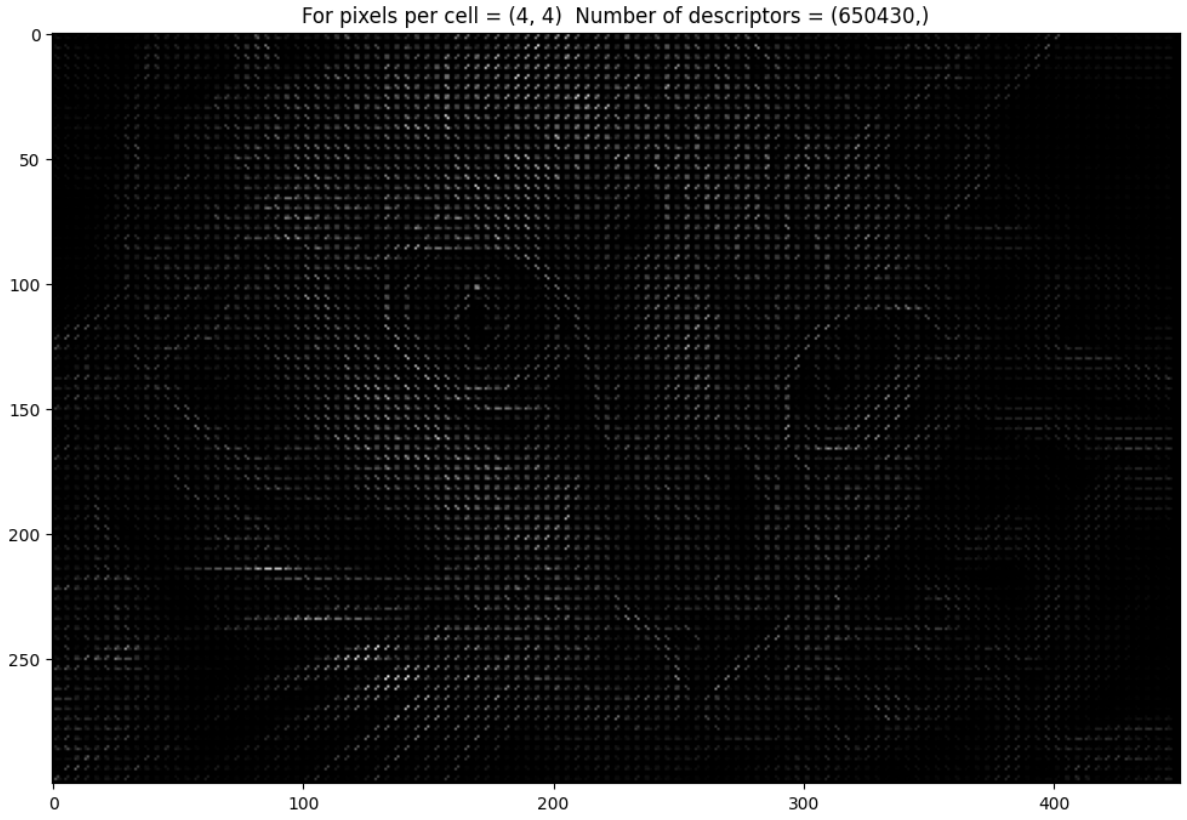
The skimage.feature.hog function returns a list. The first element of the list is a vector of the magnitude of the gradients. The second element of the list is a 2×2 array of orientation vectors. When a visualization is performed, the long axis of the markers shows the orientation and the width of the marker indicates the magnitude of the gradient.

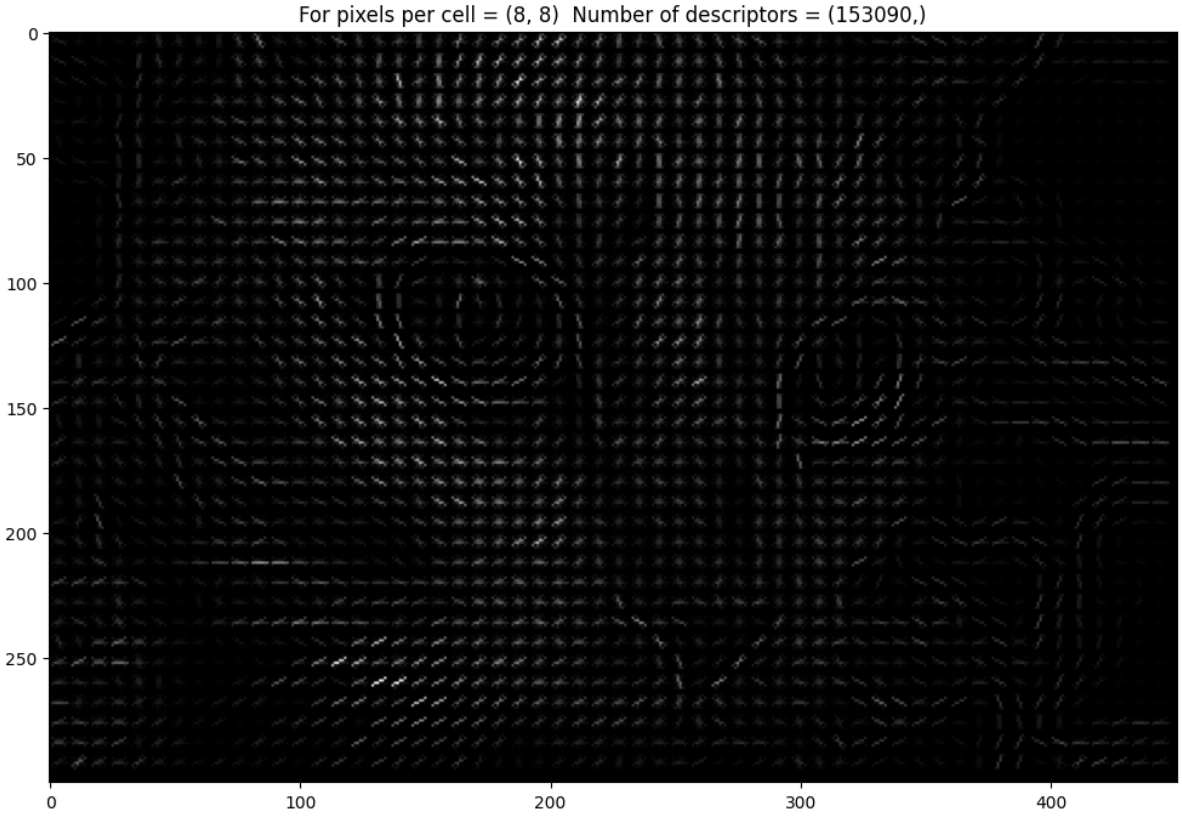
> **Exercise 3-7:** You will now apply the HOG algorithm to the equalized gray-scale cat image. You will compare the results of using a different number of pixels per cell to estimate the histogram of 9 orientations by the following steps:
>
> 1. Iterate over tuples for the `pixels_per_cell` argument with dimensions, $[(4, 4), (8, 8), (16, 16), (32, 32)]$ for the convolutional operator, and for each tuple do the rest of these steps.
> 2. Compute the gradient orientations using the `pixels_per_cell=pixels` and with `visualize=True` arguments.
> 3. Print the pixels per cell tuple, with the dimensions of the operator and the number of HOGs returned by the function in the title of each

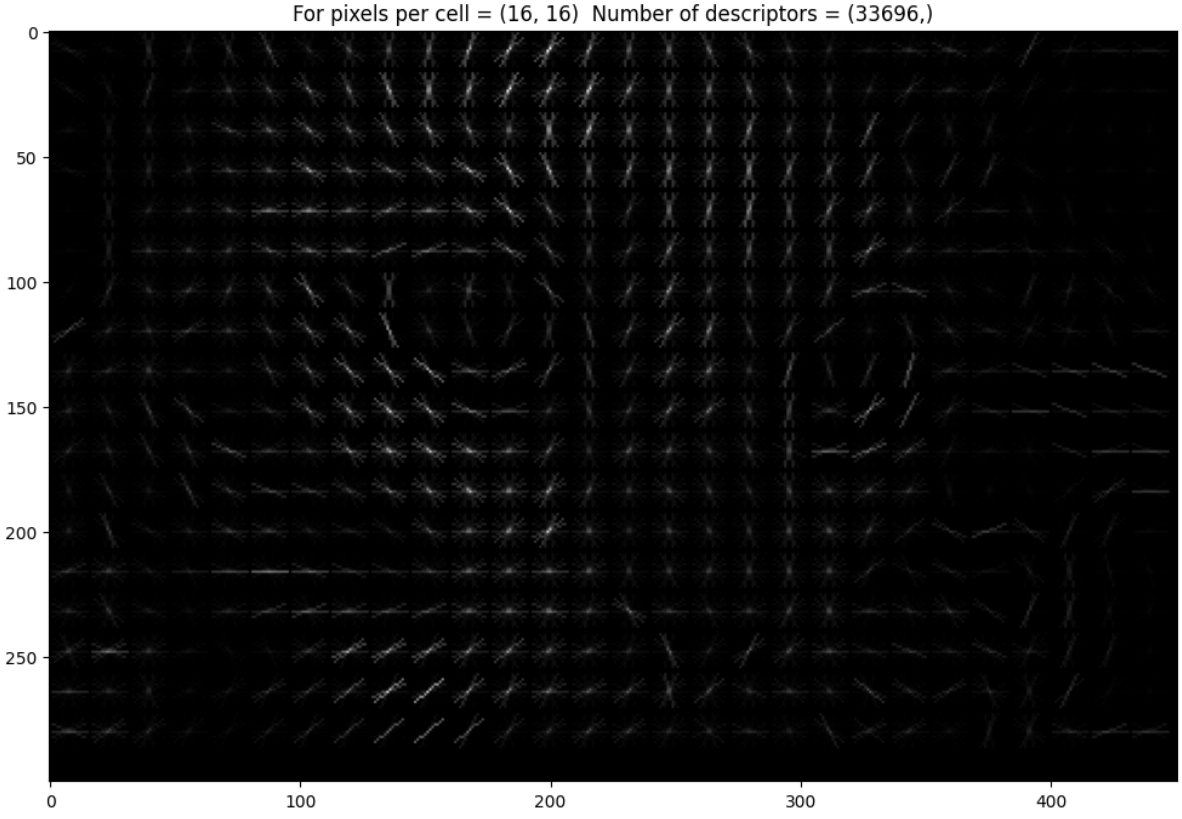
> image. The number of HOGs is in the first element of the list returned by the `hog` function.

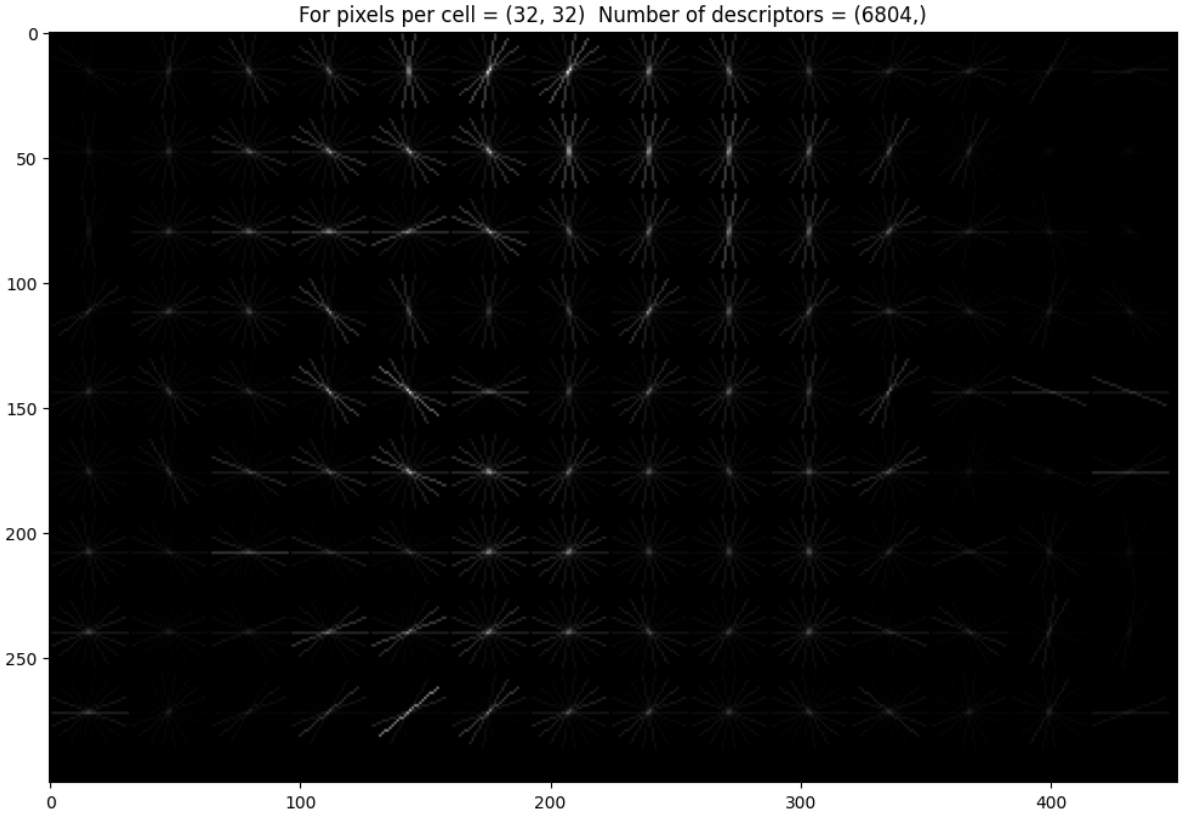
4. Display the gray-scale image of the corner orientations.

In [ ]:
```
##  Your code goes here

# 1. Iterate over tuples for the `pixels_per_cell` argument with dimensions,
#    operator, and for each tuple do the rest of these steps.
pixels_per_cell = [(4, 4), (8, 8), (16, 16), (32, 32)]

for pixels in pixels_per_cell:
    # 2. Compute the gradient orientations using the `pixels_per_cell=pixels
    cat_corner_orientations = feature.hog(cat_grayscale_equalized, pixels_pe

    # 3. Print the pixels per cell tuple, with the dimensions of the operato
    #    title of each image. The number of HOGs is in the first element of
    pixel = str(pixels[0]) + ' x ' + str(pixels[1])

    # 4. Display the gray-scale image of the corner orientations.
    fig, ax = plt.subplots(figsize = (12, 15))
    _ = ax.imshow(cat_corner_orientations[1], cmap = plt.get_cmap('gray'))
    _ = ax.set_title('For pixels per cell = ' + str(pixels) + '  Number of c
```




For pixels per cell = (8, 8) Number of descriptors = (153090,)
0
50
100
150
200
250
0
100
200
300
400


For pixels per cell = (16, 16) Number of descriptors = (33696,)
0
50
100
150
200
250
0
100
200
300
400



> Examine the results and answer the following questions:
>
> 1. How has the density of the corner direction features changes as the pixels per cell dimensionality increases and why?
> 2. How does the magnitude of the gradient change as the pixels per cell and scale increases and does this make sense given the algorithm and why?
> 3. How does the smoothness and consistency of the gradient directions as the pixels per cell dimensionality or scale increases and does this make sense given the algorithm and why?
> 4. As the size of the operator increases, you can see that the representation of the cat image changes, becoming more abstracted. If you goal is to robustly represent the object what do you think the trade-off will be and why?
>
> **End of exercise.**

> **Answers:**

> 1. According to the Professor's explanation in Ed and as we can see the images above, density of the corner directions features and pixels per cell dimensionality have an inverse relationship. This is because there are less pixels to calculate the HOGs.

2. The magnitude of the gradient decreases as the pixels per cell and scale increases. This is because as the pixels per cell and scale increases, the gradient points to more directions, which makes the magnitude of the gradient decrease. It does make sense.

3. The smoothness and consistency of the gradient directions decrease as the pixels per cell dimensionality or scale increase. It does mae sense.

4. It depends what the goal is. If the goal is to get detailed information about the object, we should use a small operator. If the goal is to identify the most prominent features only, large operators will be the right way to go.

The BRIEF Algorithm

As with edge and corner detectors, a great many **interest point** or **key point** detection algorithms have been developed. Here we will only work with one, the BRIEF algorithm. The BRIEF algorithm creates a hash of the gradient directions over patches of the image. These hashes can be matched between images to find common interest points.

Exercise 3-8: You will now apply the sskimage.feature.BRIEF function to the equalized gray-scale cat image by the following steps:

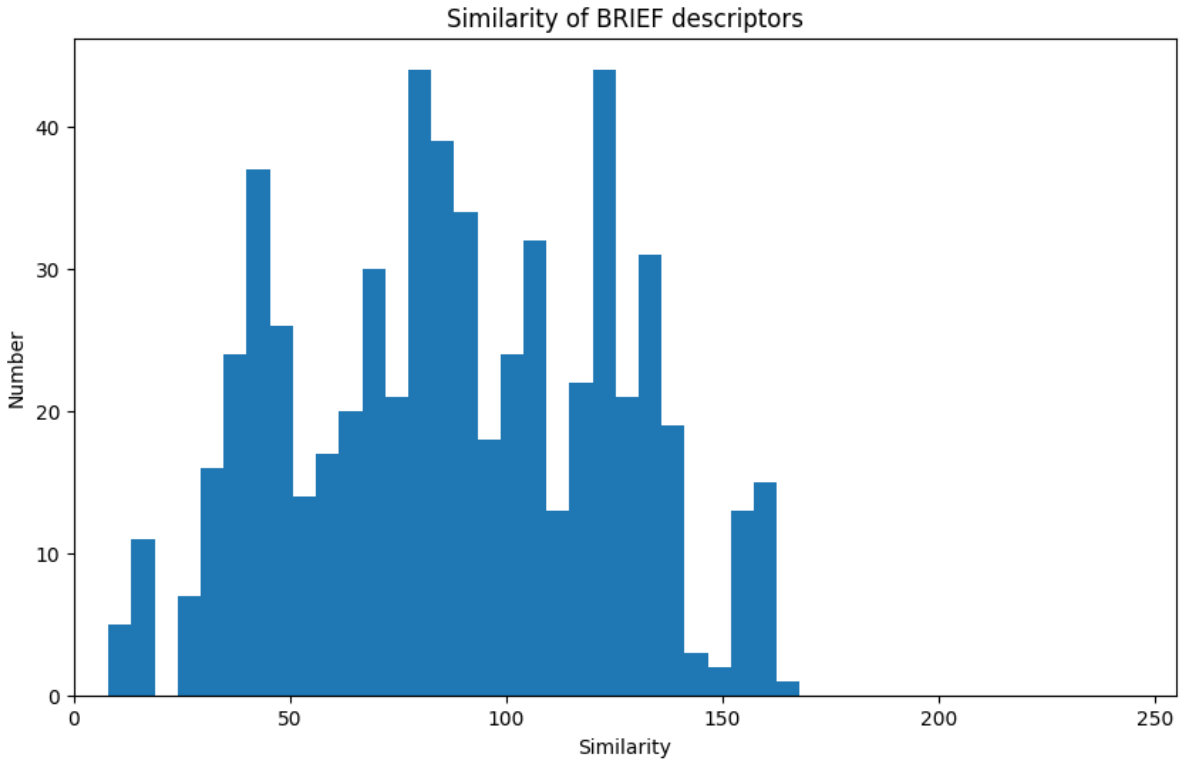
1. Apply the Harris corner detection algorithm to the equalized gray-scale cat image.
2. Find the Harris corner peaks with `min_distance=5`, and print the number of peaks found.
3. Initialize the BRIEF feature extractor with `patch_size=5`.
4. Extract the BRIEF descriptor hashes from the cat gray-scale image and the corner peaks found. The function returns an object with many attributes. The descriptor hash is the `.descriptor` attribute.
5. Print the first 4 hashes, being careful to label them.
6. Compute the **Hamming similarity** between the first hash and all subsequent hashes.
7. Display a histogram of the Hamming distances using 30 bins and the range limited to the possible similarity values. Make sure you label the axes and provide a meaningful title for your chart.

In [ ]:
```
##  Your code goes here

# 1. Apply the Harris corner detection algorithm to the equalized gray-scale
cat_harris = feature.corner_harris(cat_grayscale_equalized)
```

```
# 2. Find the Harris corner peaks with `min_distance=5`, and print the numbe
cat_harris_peaks = feature.corner_peaks(cat_harris, min_distance = 5)
print(f'Harris peaks found: {cat_harris_peaks.shape[0]}')

# 3. Initialize the BRIEF feature extractor with `patch_size = 5`.
brief_extractor = feature.BRIEF(patch_size = 5)

# 4. Extract the BRIEF descriptor hashes from the cat gray-scale image and t
#    function returns an object with many attributes. The descriptor hash is
brief_extractor.extract(cat_grayscale_equalized, cat_harris_peaks)
cat_brief_hashes = brief_extractor.descriptors

# 5. Print the first 4 hashes, being careful to label them.
for i in range(4):
    print(f'\nHash {i}:\n{cat_brief_hashes[i]}')

# 6. Compute the **Hamming similarity** between the first hash and all subse
similarity = np.sum(cat_brief_hashes[0] != cat_brief_hashes[1:], axis = 1)

# 7. Display a histogram of the Hamming distances using 30 bins and the rang
#    similarity values. Make sure you label the axes and provide a meaningfu
fig, ax = plt.subplots(figsize = (10, 6))
plt.hist(similarity, bins = 30)
plt.xlim((0, 255))
_ = plt.xlabel('Similarity')
_ = plt.ylabel('Number')
_ = plt.title('Similarity of BRIEF descriptors')
```

```
Harris peaks found: 604

Hash 0:
[False  True False False False False False  True False False  True  True
 False False False False False False  True False False False  True False
 False False False  True False False False False  True False False False
 False False False  True False False False False False False False False
 False False  True  True  True False False  True False False  True False
 False False False False False False  True  True False False False False
 False False  True False False False  True  True  True False False False
 False False  True False False False False  True False  True False  True
 False  True  True  True False False False False False False  True False
 False False False False  True False False  True  True False False False
 False False False False False False False False False False False  True
  True False False False False  True False  True False False False False
 False False  True False False False False  True False  True False False
 False False False False False False  True False False  True False False
 False False False  True False  True False False  True  True False  True
 False  True False False False  True False False False False False  True
  True  True  True False  True False False  True False False False False
 False  True False  True  True  True False  True  True  True False False
 False False False False  True  True  True False False False False False
 False False False False  True False  True False False  True False  True
  True  True  True  True False False  True False False  True  True False
 False  True False False]

Hash 1:
[False False  True False False False  True False  True False False False
  True False False False False  True False  True False  True False False
 False False  True False False  True  True  True False False False  True
  True  True  True False  True False  True False False False False  True
 False False False False False  True  True False  True  True False False
  True False False  True  True False False  True False False  True  True
  True  True False  True False  True False  True False False  True  True
 False  True False False  True  True  True  True False  True  True  True
 False False  True False False  True False False False False False False
  True  True False  True False False False False  True  True False  True
  True  True False  True  True False  True  True False  True False False
 False False False False False False  True False False  True False  True
 False False False False False False False False  True False  True False
  True False False False  True  True False  True  True False False False
  True False False False False False  True False False False False  True  True
 False False  True False False False False  True False False  True False
 False False False False  True  True  True  True False False False False
 False False False False False False False False False False False False
  True False False False False False  True  True  True False False  True
  True False  True False  True False  True False  True False False False
 False False  True False  True  True False False False False False  True
 False False  True  True]

Hash 2:
[False False  True False False False  True  True  True False  True False
 False False False False False  True False  True False  True  True False
 False False False False False  True False  True False False False  True
  True  True  True False False False  True False False False False  True
 False False False False False  True  True  True  True  True False False
```

```
 False False False  True  True False False False False False  True  True
  True False False False False  True  True False  True False  True  True
 False  True False False  True  True False False  True False  True False
 False  True False  True  True  True False  True False False  True False
 False False False  True False False False  True False  True False  True
  True  True False  True  True False False  True False  True  True  True
 False False False False False False  True False False  True False  True
 False False False False  True False False False  True False  True False
 False False False  True  True  True False False False  True False  True
  True False False False False  True False False False  True  True  True
 False False  True False  True  True False  True False False False  True
 False  True False False False False False False False  True False False
 False False False False False  True False False  True False  True False
  True False False False False False False  True  True False False  True
  True False  True False False False False False False False  True  True
 False False False  True  True False False False False False  True  True
 False False False  True]

Hash 3:
[False  True  True False False False False  True False False  True False
 False False  True False False False  True False False  True  True False
 False False False  True False  True False False  True False False False
 False False False  True False False False False False False False False
 False False  True  True  True False  True  True False False  True False
  True False False False False False  True False False False False  True
 False False  True False False False  True False  True False False  True
 False False  True False  True False False False  True False False False
 False  True False  True  True False False  True False False  True False
 False False False False  True False False  True False  True False False
 False False False  True False  True False False False False  True  True
  True False False False False  True False  True False False False False
 False False  True False  True False False  True  True  True  True False
 False False False  True False  True  True False  True  True False  True
  True False False  True False False False False  True  True False  True
 False False False False  True  True False False False False False  True
  True  True  True False False False False False False  True False False
 False False False  True  True  True False False  True False  True False
 False False False False  True False False False False False False  True
 False False False False False False False False  True  True  True  True
  True False False  True False False  True False False  True  True False
 False  True False False]
```



> Examine histogram of the hash similarity values. Keeping in mind the maximum and minimum possible values, what statement can you make about how strong the similarity is between these interest points. Try to explain your reasoning.
> **End of exercise.**

> **Answer:**

> Taking into account that there were 604 Harris picks found and that possible values range between 0 and 255, we can see that the Hamming similarity is reasonably strong. There are no values with big number of ocurrencies (i.e. 10% or more) but there are a few over 5%. Also, having all cases within a reduced range between ~5 and ~170, where probably just one vaue has only one occurency, we can say that many of the cases are similar.

Texture

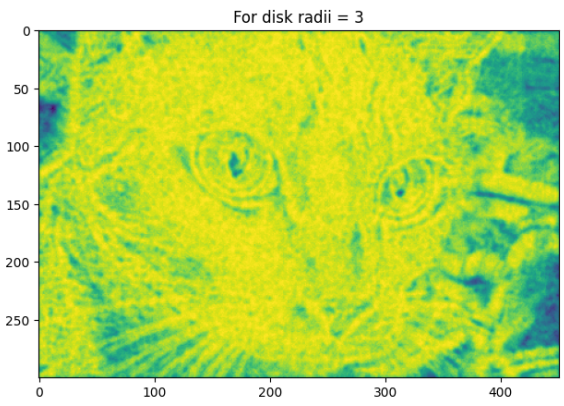
Texture is a ubiquitous feature of nearly all images. However, stating exactly what texture is, and how to measure it, is not simple or strait forward. In general, we can say that texture is a function of local variation of the image. Typical measures used to quantify texture are:

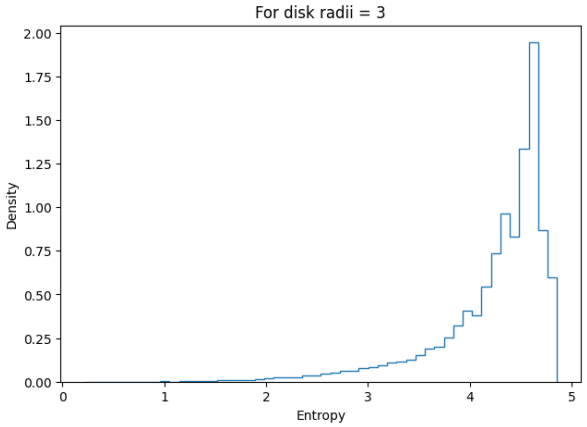
- Covariance over local patches of the image where regions with higher covariance have rougher textures.
- Entropy computed over local patches of the image where regions with higher entropy have rougher textures.

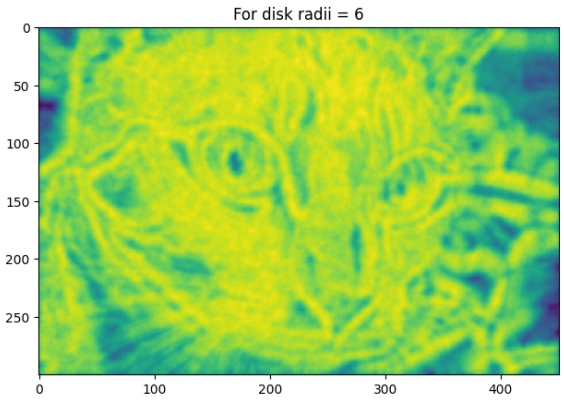
Here we will focus on local entropy to measure texture using the skimage.filters.rank.entropy function. ids

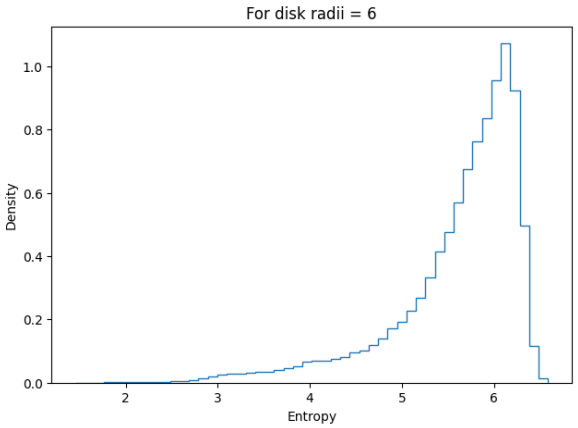
> **Exercise 3-9:** You will now use local entropy as a measure of texture on the equalized gray-scale cat image by doing the following.
>
> 1. Iterate over the disk patch radii, $[3, 6, 12, 24, 48, 96]$, and for each value do each of the following steps.
> 2. Compute the local entropy of the equalized gray-scale cat image using the skimage.morphology.disk for the patch argument, and using the diameter value. Make sure you convert the image to unsigned 8-bit integer with skimage.util.img_as_ubyte.
> 3. Side by side, display the image and a histogram of the entropy values with 50 bins. Include a title with the disk diameter and axes labels for the histogram.

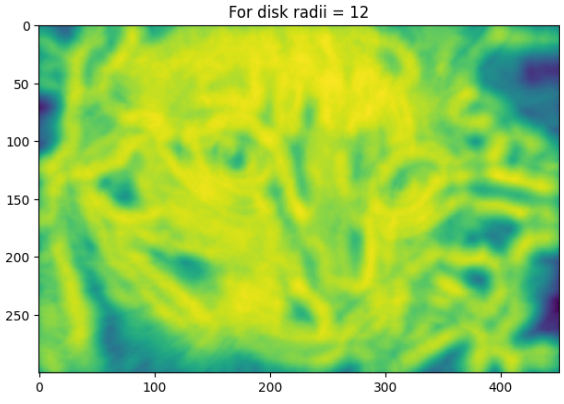
In [ ]:
```
##  Your code goes here

# 1. Iterate over the disk patch radii, [3, 6, 12, 24, 48, 96], and for each
radii = [3, 6, 12, 24, 48, 96]

for radii in radii:

# 2. Compute the local entropy of the equalized gray-scale cat image using t
#    patch argument, and using the diameter value. Make sure you convert the
#    with skimage.util.img_as_ubyte.
    cat_entropy = entropy(skimage.util.img_as_ubyte(cat_grayscale_equalized)


# 3. Side by side, display the image and a histogram of the entropy values w
#    the disk diameter and axes labels for the histogram.
    fig, ax = plt.subplots(1, 2, figsize = (16, 5))
    _ = ax[0].imshow(cat_entropy)
    _ = ax[0].set_title('For disk radii = ' + str(radii))
    _ = ax[1].hist(cat_entropy.flatten(), density = True, bins = 50, histtyp
    _ = ax[1].set_title('For disk radii = ' + str(radii))
    _ = ax[1].set_ylabel('Density')
    _ = ax[1].set_xlabel('Entropy')
```

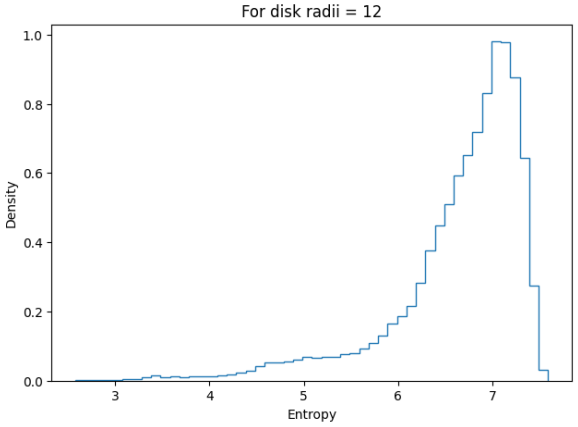

For disk radii = 3

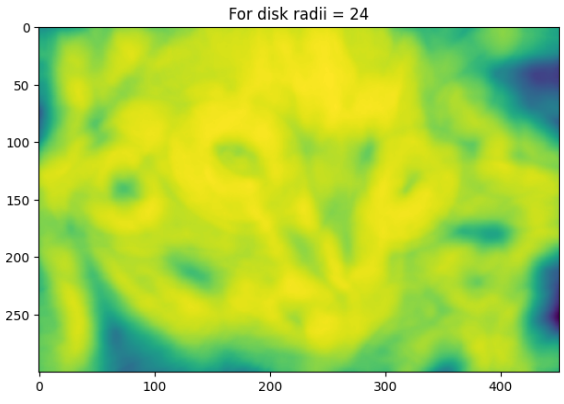

For disk radii = 3
Density
Entropy

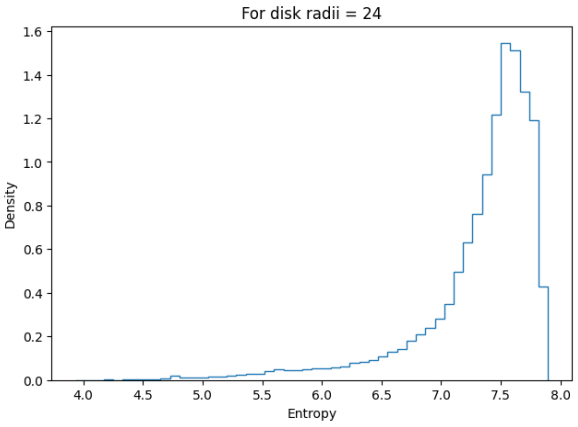

For disk radii = 6

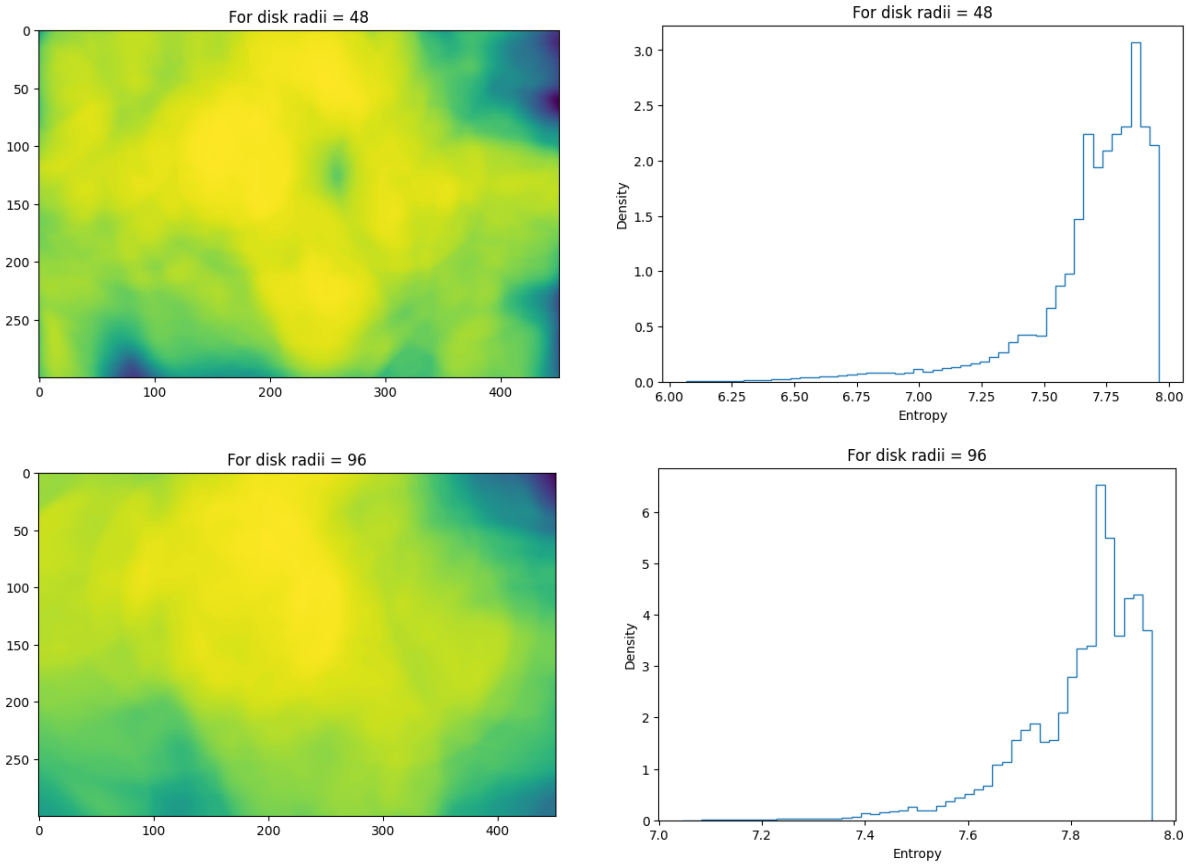

For disk radii = 6
Density
Entropy


For disk radii = 12


For disk radii = 12
Density
Entropy


For disk radii = 24


For disk radii = 24
Density
Entropy



> Examine these results and answer the following questions:
>
> 1. How can you describe the change in the image with operator diameter, and what does this tell you about the scale of the features?
> 2. At what scale does the histogram exhibit distinctly multi=modal behavior, and what does this mean in terms of possibly segmenting the image into regions with common properties?
>
> **End of exercise.**

> **Answers:**

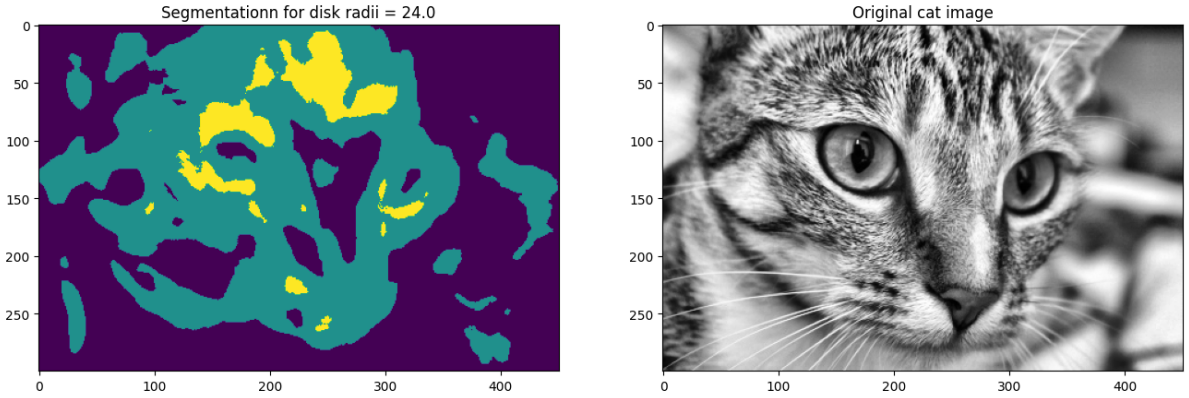
> 1. entropy is a base-2 logarithmic function that represents the minimum number of bits to encode the local level distribution. As the diameter increases, the entropy increases as well making the image more complex and the features 'mixed'. To get better definition of the features it is needed to use small diameters.

> 2. It seems to be at the scale of 48 where the histogram exhibits multi-modal behavior. This means that by segmenting the image into regions it is easier to determine where the most notorious points of interest are.

Local entropy shows texture properties of an image. There are several uses for these properties in CV algorithm. Segmentation of an image into regions by texture is one such algorithm.

The code in the cell below segments the image into 3 regions by value of the entropy. Execute the code and examine the result.

In [ ]:
```
radii = 24.0
threshold1 = 7.8
threshold2 = 7.5
cat_entropy_96 = entropy(util.img_as_ubyte(cat_grayscale_equalized), morphol
cat_entropy_96 = np.where(cat_entropy_96 > threshold1, 1.0, np.where(cat_ent

fig, ax = plt.subplots(1, 2, figsize=(16,5))
ax = ax.flatten()
_=ax[0].imshow(cat_entropy_96)
_=ax[0].set_title('Segmentationn for disk radii = ' + str(radii))
_=ax[1].imshow(cat_grayscale_equalized, cmap=plt.get_cmap('gray'))
_=ax[1].set_title('Original cat image')
```



Compare the regions found by the segmentation to the original gray-scale image. Notice how the segmentation of the image by texture reasonably matches the large-scale features of the cat image. This segmentation could be used to mask the areas of image of interest, for example.