Name:
Surname:
Student Id:

```
!pip install pillow numpy matplotlib
```

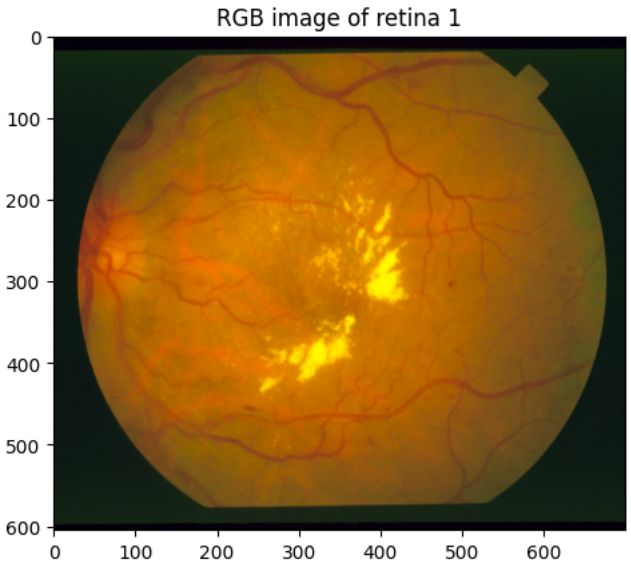
```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: pillow in /usr/local/lib/python3.9/dist-packages (8.4.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.9/dist-packages (1.22.4)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.9/dist-packages (3.7.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (0.11.0)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (3.0.9)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (23.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (1.4.4)
Requirement already satisfied: importlib-resources>=3.2.0 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (5.12.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (1.0.7)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.9/dist-packages (from matplotlib) (4.39.3)
Requirement already satisfied: zipp>=3.1.0 in /usr/local/lib/python3.9/dist-packages (from importlib-resources>=3.2.0->matplotlib) (3.15
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.9/dist-packages (from python-dateutil>=2.7->matplotlib) (1.16.0)
```

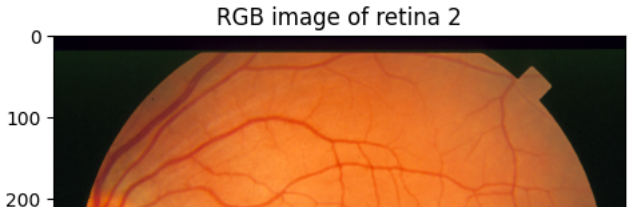
Exercise 1 – Segment vessels using intensity thresholding

1.1-Load the RGB images of the retina, display them and compute the histogram of each of the colour channels image.

In this code we load two RGB images (`im0001.ppm' and 'im0077.ppm') from Retinas using the Python Imaging Library (PIL) and converts them to NumPy arrays using the NumPy library . We then use the matplotlib library to display each image in a separate window with a title. The first image is titled "RGB Image for Retina 1" and the second image is titled "RGB Image for Retina 2". Moreover, it computes the histograms of the color channels for each image using the numpy.histogram function and plots them using matplotlib. The red, green, and blue histograms are plotted separately in different colors with a legend indicating the color channel. The resulting plots show the distribution of pixel values for each color channel in the two retina images.

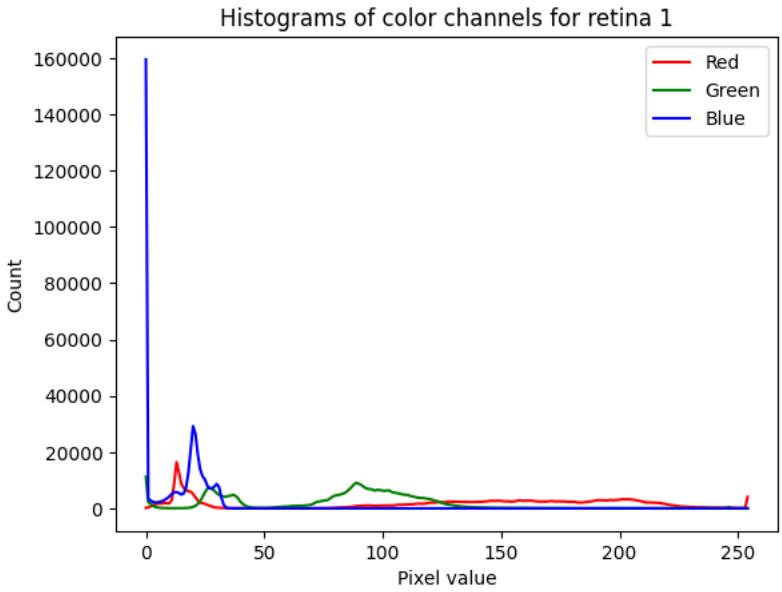
```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

# Load the first RGB image
im1 = np.array(Image.open('im0001.ppm'))

# Display the image
plt.imshow(im1)
plt.title('RGB image of retina 1')
plt.show()

# Load the second RGB image
im2 = np.array(Image.open('im0077.ppm'))

# Display the image
plt.imshow(im2)
plt.title('RGB image of retina 2')
plt.show()
```

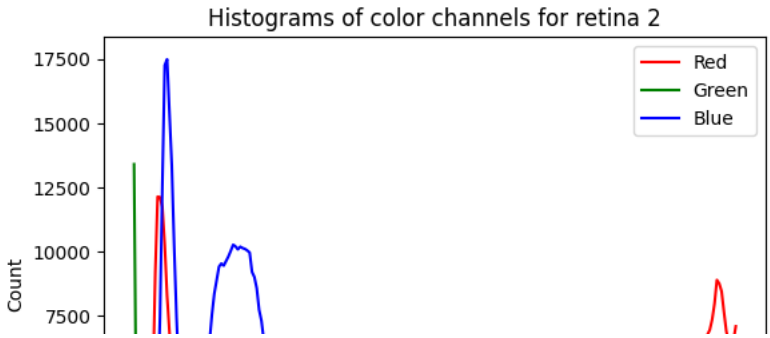




Histogram

```
# Compute the histograms of the color channels for the first image
red_hist = np.histogram(im1[:,:,0], bins=256, range=(0, 255))
green_hist = np.histogram(im1[:,:,1], bins=256, range=(0, 255))
blue_hist = np.histogram(im1[:,:,2], bins=256, range=(0, 255))

# Plot the histograms
plt.plot(red_hist[1][:-1], red_hist[0], color='red', label='Red')
plt.plot(green_hist[1][:-1], green_hist[0], color='green', label='Green')
plt.plot(blue_hist[1][:-1], blue_hist[0], color='blue', label='Blue')
plt.title('Histograms of color channels for retina 1')
plt.xlabel('Pixel value')
plt.ylabel('Count')
plt.legend()
plt.show()

# Compute the histograms of the color channels for the second image
red_hist = np.histogram(im2[:,:,0], bins=256, range=(0, 255))
green_hist = np.histogram(im2[:,:,1], bins=256, range=(0, 255))
blue_hist = np.histogram(im2[:,:,2], bins=256, range=(0, 255))

# Plot the histograms
plt.plot(red_hist[1][:-1], red_hist[0], color='red', label='Red')
plt.plot(green_hist[1][:-1], green_hist[0], color='green', label='Green')
plt.plot(blue_hist[1][:-1], blue_hist[0], color='blue', label='Blue')
plt.title('Histograms of color channels for retina 2')
plt.xlabel('Pixel value')
plt.ylabel('Count')
plt.legend()
plt.show()
```



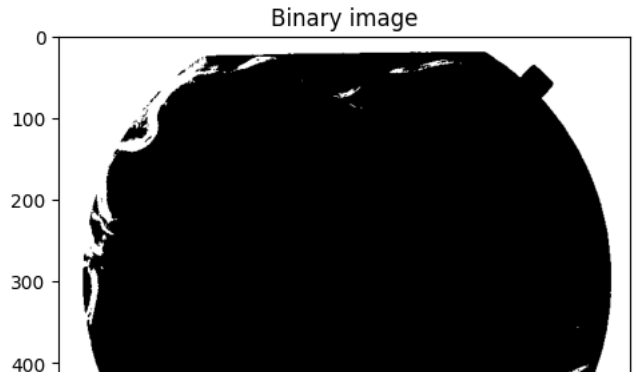


1.2-Choose on of the colour channels and use one of the methods available in OpenCV or scikit-image to define a threshold. Use the threshold to identify dark regions from bright regions

The threshold otsu function from the skimage.filters module is imported in this section of code. The green channel of the first image is then extracted, and the threshold otsu function is used to calculate the Otsu threshold value for the green channel. A well-liked thresholding approach is the Otsu method, which uses the histogram of pixel intensities to determine the best threshold value to distinguish the foreground from the background.

After calculating the threshold value, the code uses a binary comparison operation () to apply the threshold to the green channel, saving the resulting binary image in the binary image variable. Lastly, matplotlib.pyplot.imshow is used to display the binary image ().

```
from skimage.filters import threshold_otsu

# Extract the green channel of the first image
green_channel = im1[:,:,1]

# Compute the Otsu threshold value
threshold_value = threshold_otsu(green_channel)

# Apply the threshold to the green channel
binary_image = green_channel < threshold_value

# Display the binary image
plt.imshow(binary_image, cmap='gray')
plt.title('Binary image')
plt.show()
```



1.3-. Load the images containing the labelled vessels, and compute both Dice and Jaccard similarity index comparing the thresholded image with the manual labelling to evaluate how well the vessels have been segmented

By thresholding the pixel values greater than 1, this function loads the labelled vessels for the first image and computes a binary labelling. The intersection of the binary label and binary image is then divided by the sum of the binary label, binary image, and intersection to determine the Dice and Jaccard similarity indices. The code then prints the similarity indices it has computed. The effectiveness of the vessels' segmentation is assessed using these indices.

```
# # Load the labeled vessels for the first image
label_im1 = np.array(Image.open('im0001_label.ppm'))


# # Compute the binary labeling
binary_label_im1 = label_im1 > 1

# # Compute the Dice and Jaccard similarity indices
intersection = np.logical_and(binary_label_im1, binary_image)
dice = 2 * intersection.sum() / (binary_label_im1.sum() + binary_image.sum())
jaccard = intersection.sum() / (binary_label_im1.sum() + binary_image.sum() - intersection.sum())

# # Print the results
print('Dice similarity coefficient:', dice)
print('Jaccard similarity index:', jaccard)
```

```
Dice similarity coefficient: 0.06194227095491063
Jaccard similarity index: 0.03196100406432709
```

```
# # Load the labeled vessels for the first image
label_im1 = np.array(Image.open('im0077_label.ppm'))


# # Compute the binary labeling
binary_label_im1 = label_im1 > 1

# # Compute the Dice and Jaccard similarity indices
intersection = np.logical_and(binary_label_im1, binary_image)
dice = 2 * intersection.sum() / (binary_label_im1.sum() + binary_image.sum())
jaccard = intersection.sum() / (binary_label_im1.sum() + binary_image.sum() - intersection.sum())

# # Print the results
print('Dice similarity coefficient:', dice)
print('Jaccard similarity index:', jaccard)
```

```
Dice similarity coefficient: 0.024602271984522268
Jaccard similarity index: 0.012454338503891153
```

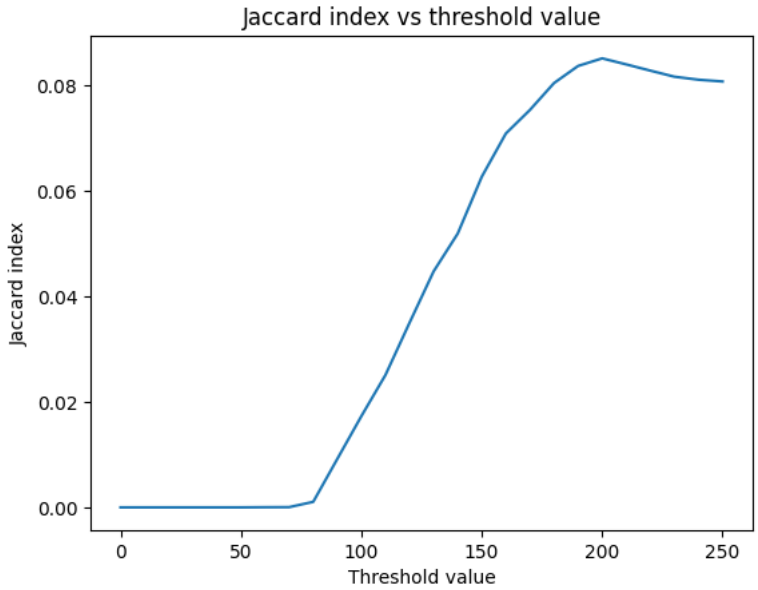
1.4-Define a vector of possible threshold values between 0 and 255, spaced 10. ** **For each of these values
1. Use the threshold to segment the image, 2.Compute the Jaccard index between the segmentation and the labelled image 3.Finally display a plot with the threshold values on the x-axis and the corresponding Jaccard index on the y axis

The following pieces of code import a binary label image and an RGB picture, converts the binary label image to a binary image with 0s and 1s, creates a vector of potential threshold values, calculates the Jaccard index for each threshold value, and shows the Jaccard index as a function

of threshold value. The Jaccard index calculates the degree of similarity between the binary label picture and the binary image created by thresholding the RGB image. For each value in the threshold vector, the code applies a threshold to the image before computing the Jaccard index and recording it in a list. The Jaccard index is then plotted as a function of the threshold value by the code. The graphic demonstrates the Jaccard index's variation with various threshold settings, enabling one to select the level of similarity between the binary image and the binary label image that is maximised.

```
import numpy as np
import matplotlib.pyplot as plt
from skimage.color import rgb2gray
from skimage.filters import threshold_otsu
from sklearn.metrics import jaccard_score
import skimage.io as io

# Load RGB image and binary label image
im = io.imread('im0001.ppm')
binary_label_im = io.imread('im0001_label.ppm')
binary_label_im = np.expand_dims(binary_label_im, axis=-1)
binary_label_im = np.repeat(binary_label_im, 3, axis=-1)

# Convert binary label image to a binary image with 0s and 1s
gray_label_im = rgb2gray(binary_label_im)
thresh = threshold_otsu(gray_label_im)
binary_label_im = (gray_label_im > thresh).astype(int)

# Define vector of possible threshold values
thresholds = np.arange(0, 256, 10)

# Compute Jaccard index for each threshold value
jaccard_indices = []
for threshold in thresholds:
    # Apply threshold to image
    binary_im = im[:,:,0] < threshold

    # Compute Jaccard index
    jaccard_index = jaccard_score(binary_label_im.ravel(), binary_im.ravel())
    jaccard_indices.append(jaccard_index)

# Plot results
plt.plot(thresholds, jaccard_indices)
plt.xlabel('Threshold value')
plt.ylabel('Jaccard index')
plt.title('Jaccard index vs threshold value')
```

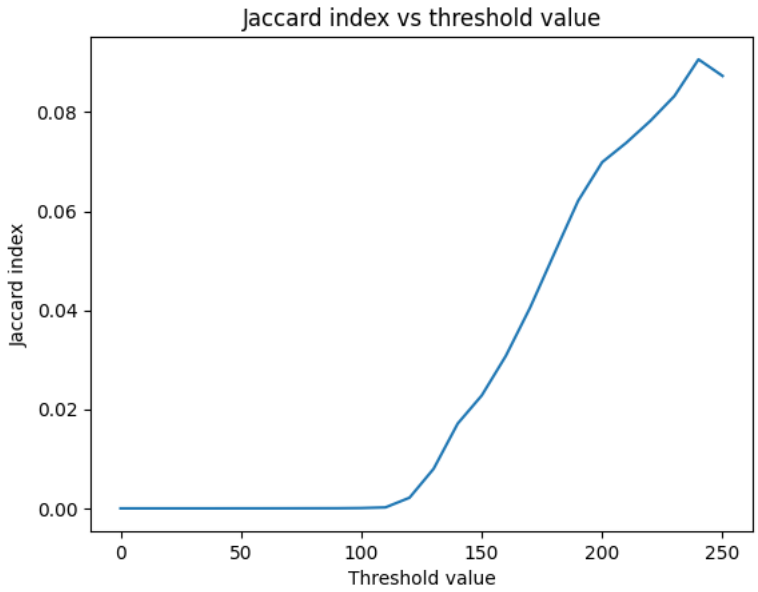
```
Text(0.5, 1.0, 'Jaccard index vs threshold value')
```




```
import numpy as np
import matplotlib.pyplot as plt
from skimage.color import rgb2gray
from skimage.filters import threshold_otsu
```

```
from sklearn.metrics import jaccard_score
import skimage.io as io

# Load RGB image and binary label image
im = io.imread('im0077.ppm')
binary_label_im = io.imread('im0077_label.ppm')
binary_label_im = np.expand_dims(binary_label_im, axis=-1)
binary_label_im = np.repeat(binary_label_im, 3, axis=-1)

# Convert binary label image to a binary image with 0s and 1s
gray_label_im = rgb2gray(binary_label_im)
thresh = threshold_otsu(gray_label_im)
binary_label_im = (gray_label_im > thresh).astype(int)

# Define vector of possible threshold values
thresholds = np.arange(0, 256, 10)

# Compute Jaccard index for each threshold value
jaccard_indices = []
for threshold in thresholds:
    # Apply threshold to image
    binary_im = im[:,:,0] < threshold

    # Compute Jaccard index
    jaccard_index = jaccard_score(binary_label_im.ravel(), binary_im.ravel())
    jaccard_indices.append(jaccard_index)

# Plot results
plt.plot(thresholds, jaccard_indices)
plt.xlabel('Threshold value')
plt.ylabel('Jaccard index')
plt.title('Jaccard index vs threshold value')
```

```
Text(0.5, 1.0, 'Jaccard index vs threshold value')
```



1.5-Apply some morphological operators to remove noise and small segmented objects from the segmentation and compute the Dice and Jaccard similarity index comparing the new segmentation with the manual labelling.

These image processing operations are carried out by this code:

1. Utilises the skimage.io module to load an RGB image and its matching binary label image.
2. Uses the skimage.color module to change the binary label image into a binary image with 0s and 1s and a threshold value calculated by the skimage.filters module.
3. Uses the skimage.morphology module to define two structuring elements for morphological operations.The binary image is subjected to morphological closing and opening processes in order to remove small objects and holes, respectively.
4. Utilising the sklearn.metrics module, calculates the Dice and Jaccard similarity indices between the binary label image and the processed binary image.
5. Prints the similarity index findings to the console.

```
import numpy as np
import matplotlib.pyplot as plt
from skimage.color import rgb2gray
from skimage.filters import threshold_otsu
from skimage.morphology import binary_closing, binary_opening, disk
from sklearn.metrics import jaccard_score
import skimage.io as io
from sklearn.metrics import f1_score


# Load RGB image and binary label image
im = io.imread('im0001.ppm')
binary_label_im = io.imread('im0001_label.ppm')
binary_label_im = np.expand_dims(binary_label_im, axis=-1)
binary_label_im = np.repeat(binary_label_im, 3, axis=-1)

# Convert binary label image to a binary image with 0s and 1s
gray_label_im = rgb2gray(binary_label_im)
thresh = threshold_otsu(gray_label_im)
binary_label_im = (gray_label_im > thresh).astype(int)

# Define structuring elements for morphological operations
se_close = disk(5)
se_open = disk(10)

# Apply morphological operations to binary image
binary_im = im[:,:,0] < 80  # Apply initial threshold
binary_im = binary_closing(binary_im, se_close)  # Remove small holes
binary_im = binary_opening(binary_im, se_open)  # Remove small objects

# Compute Dice and Jaccard similarity indices
dice_index = f1_score(binary_label_im.ravel(), binary_im.ravel())
jaccard_index = jaccard_score(binary_label_im.ravel(), binary_im.ravel())

# Print results
print(f"Dice similarity index: {dice_index:.4f}")
print(f"Jaccard similarity index: {jaccard_index:.4f}")
```

```
Dice similarity index: 0.0024
Jaccard similarity index: 0.0012
```

```
import numpy as np
import matplotlib.pyplot as plt
from skimage.color import rgb2gray
from skimage.filters import threshold_otsu
from skimage.morphology import binary_closing, binary_opening, disk
from sklearn.metrics import jaccard_score
import skimage.io as io
from sklearn.metrics import f1_score


# Load RGB image and binary label image
im = io.imread('im0077.ppm')
binary_label_im = io.imread('im0077_label.ppm')
binary_label_im = np.expand_dims(binary_label_im, axis=-1)
binary_label_im = np.repeat(binary_label_im, 3, axis=-1)

# Convert binary label image to a binary image with 0s and 1s
gray_label_im = rgb2gray(binary_label_im)
thresh = threshold_otsu(gray_label_im)
binary_label_im = (gray_label_im > thresh).astype(int)

# Define structuring elements for morphological operations
se_close = disk(5)
se_open = disk(10)

# Apply morphological operations to binary image
binary_im = im[:,:,0] < 80  # Apply initial threshold
binary_im = binary_closing(binary_im, se_close)  # Remove small holes
binary_im = binary_opening(binary_im, se_open)  # Remove small objects

# Compute Dice and Jaccard similarity indices
dice_index = f1_score(binary_label_im.ravel(), binary_im.ravel())
```

```
jaccard_index = jaccard_score(binary_label_im.ravel(), binary_im.ravel())

# Print results
print(f"Dice similarity index: {dice_index:.4f}")
print(f"Jaccard similarity index: {jaccard_index:.4f}")
```

```
Dice similarity index: 0.0000
Jaccard similarity index: 0.0000
```

Exercise 2 – Segment vessels by learning intensity distribution

[ ] ↳ *18 cells hidden*

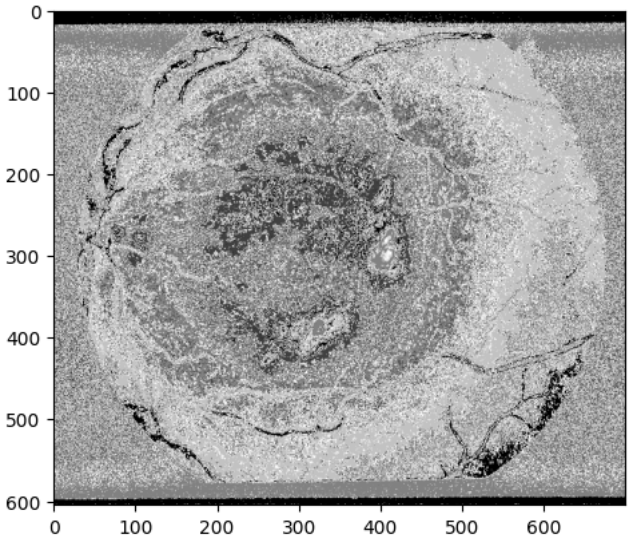
Exercise 3 – Enhance vessels with Gabor filters

3.1-Load one of the RGB images and its corresponding labelled image. Select the green channel of the image of the retina. Create a filter bank of Gabor filters at different orientation and scale to match the size of vessels you see in the image

With this code, a bank of Gabor filters can be applied to an RGB image's green channel. The code first imports the required libraries before reading in an RGB image and its corresponding annotated image. The RGB image's green channel is then extracted. Then, multiple orientations and scales for the filters are specified, resulting in the creation of a filter bank of Gabor filters. For edge identification, texture analysis, and other image processing applications, the Gabor filter is a linear filter. Next, a loop is used to apply the filter bank on the image's green channel. Using ndi.convolve, the code convolves each Gabor filter with the green channel image ().The np.sqrt() method is then used to merge the convolution's real and imaginary components to produce a filtered image. Lastly, a list called filtered images is used to store the filtered images. The use of these images for additional analysis, such as feature extraction or categorization, is possible.

```
import numpy as np
import skimage.io as io
import skimage.filters as filters
import skimage.color as color
import skimage.draw as draw
from scipy import ndimage as ndi
from skimage.filters import gabor


# Load RGB image and annotated image
im2 = io.imread('im0001.ppm')
annotated_im2 = io.imread('im0001_label.ppm')

# Select the green channel of the RGB image
green_im2 = im2[:,:,1]

# Create a filter bank of Gabor filters
orientations = [0, np.pi/4, np.pi/2, 3*np.pi/4]
scales = [8, 16, 32]
filter_bank = []

theta = 0  # orientation of the Gabor filter, in radians
sigma = 5  # standard deviation of the Gaussian envelope
frequency = 0.1  # frequency of the sinusoidal plane wave
psi = 0  # phase offset of the sinusoidal plane wave
gabor_filter_real, gabor_filter_imag = gabor(green_im2, frequency=frequency, theta=theta, sigma_x=sigma, sigma_y=sigma, offset=psi, mode='ref

for theta in orientations:
    for sigma in scales:
        gabor_filter = filters.gabor_kernel(sigma, theta=theta)
        filter_bank.append(gabor_filter)

# Apply filter bank to the green channel of the image
filtered_images = []

for gabor_filter in filter_bank:
    filtered_real = ndi.convolve(green_im2, np.real(gabor_filter), mode='reflect')
    filtered_imag = ndi.convolve(green_im2, np.imag(gabor_filter), mode='reflect')
    filtered_images.append(np.sqrt(filtered_real**2 + filtered_imag**2))
```

```
import matplotlib.pyplot as plt

# Plot the first filtered image
plt.imshow(filtered_images[0], cmap='gray')
plt.show()
```



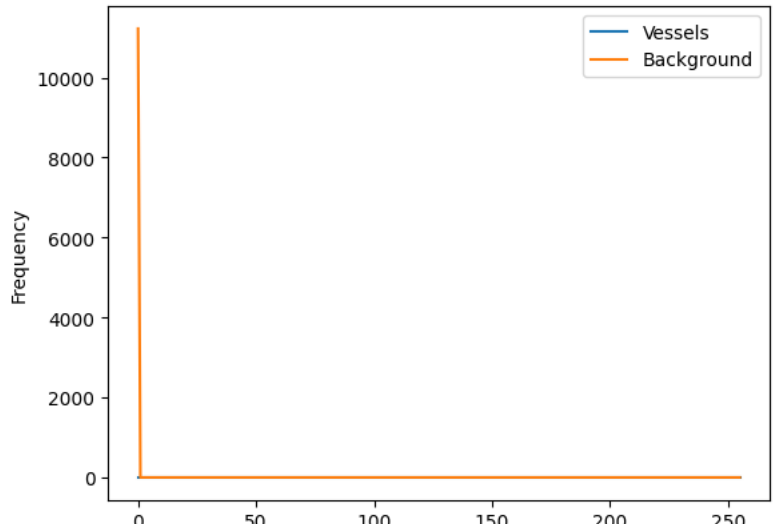
3.2-. On the green channel:

- **Apply the filters in the filter bank to the image and create an image containing at each pixel the maximum response among all filters**
- **Compute and display the histogram of the maximum response image, both for the pixels corresponding to the manually annotated vessels and for the pixels corresponding to the background**

This piece of code creates an array of zeros with the same size and data type as the green im2 picture as the max response im variable's initial value. Then, using the ndi.convolve function, it loops through the filter bank and convolves each filter with the green im2 picture. The maximum response at each pixel location across all of the filtered images is determined using the np.maximum function. The final max response im image displays the greatest possible filter response in all directions and at all scales.By thresholding the annotated im2 picture to choose pixels that correspond to vessel annotations, a binary mask called vessel mask is produced. This is followed by the computation of two histograms: one for the pixel intensities inside the vessel mask (vessel hist) and another for the pixel intensities outside the vessel mask (background hist). Lastly, Matplotlib is used to plot the histograms.

```
max_response_im = np.zeros_like(green_im2)

for i in range(len(filter_bank)):
    filtered_im = ndi.convolve(green_im2, filter_bank[i], mode='reflect')
    max_response_im = np.maximum(max_response_im, filtered_im)



# Create a binary mask for the annotated vessels
vessel_mask = annotated_im2 == 255

# Compute histograms for the vessel and background pixels
vessel_hist, _ = np.histogram(max_response_im[vessel_mask], bins=256, range=(0, 256))
background_hist, _ = np.histogram(max_response_im[~vessel_mask], bins=256, range=(0, 256))

# Display histograms
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.plot(vessel_hist, label='Vessels')
ax.plot(background_hist, label='Background')
ax.set_xlabel('Pixel Value')
ax.set_ylabel('Frequency')
ax.legend()
plt.show()
```

```
/usr/local/lib/python3.9/dist-packages/numpy/lib/histograms.py:852: ComplexWarning: Casting complex val
  indices = f_indices.astype(np.intp)
```



3.3-Using the histogram of the filter response of vessels and background obtained in the previous step, define a threshold to separate the vessels from the background

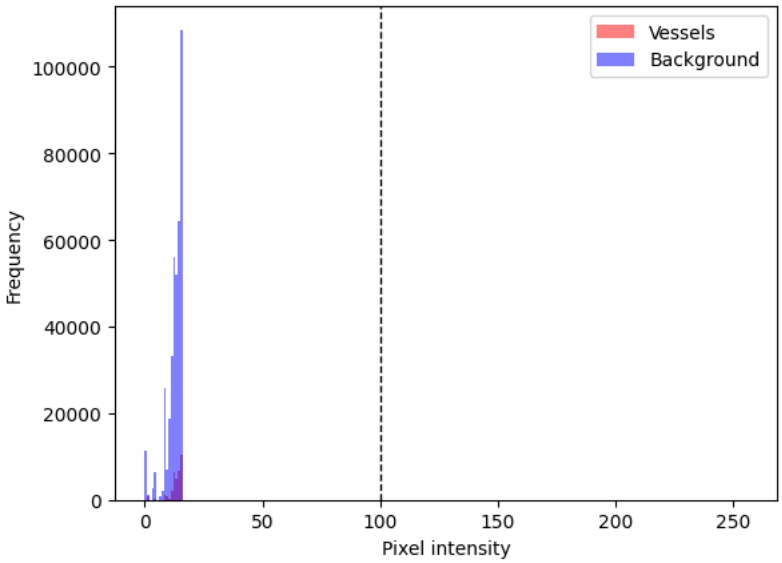
The following code segments the vessels in the RGB image im2 and the annotated image that goes with it, im2 annotated. The green channel of the RGB image is chosen, and a bank of Gabor filters is built with various scales and orientations. The green channel of the image is subjected to the filter bank, and an image with the highest response from each filter is produced. For vessel and background pixels, the maximum response image's histogram is generated. Next, a threshold that divides the two histograms is discovered using Otsu's approach, and it is applied to the maximum response image to produce a binary image of vessels. The binary vessel mask after thresholding is exhibited, followed by a histogram of the vessel and background pixels.

```
import numpy as np
import skimage.io as io
import skimage.filters as filters
import skimage.color as color
import skimage.draw as draw
from scipy import ndimage as ndi
from skimage.filters import gabor

# Load RGB image and annotated image
im2 = io.imread('im0001.ppm')
annotated_im2 = io.imread('im0001_label.ppm')

# Select the green channel of the RGB image
green_im2 = im2[:,:,1]

# Create a filter bank of Gabor filters
orientations = [0, np.pi/4, np.pi/2, 3*np.pi/4]
scales = [8, 16, 32]
filter_bank = []

theta = 0  # orientation of the Gabor filter, in radians
sigma = 5  # standard deviation of the Gaussian envelope
frequency = 0.1  # frequency of the sinusoidal plane wave
psi = 0  # phase offset of the sinusoidal plane wave
gabor_filter_real, gabor_filter_imag = gabor(green_im2, frequency=frequency, theta=theta, sigma_x=sigma, sigma_y=sigma, offset=psi, mode='ref

for theta in orientations:
    for sigma in scales:
        gabor_filter = filters.gabor_kernel(sigma, theta=theta)
        filter_bank.append(gabor_filter)

# Apply filter bank to the green channel of the image
filtered_images = []
for gabor_filter in filter_bank:
    filtered_real = ndi.convolve(green_im2, np.real(gabor_filter), mode='reflect')
    filtered_imag = ndi.convolve(green_im2, np.imag(gabor_filter), mode='reflect')
    filtered_image = np.sqrt(filtered_real**2 + filtered_imag**2)
    filtered_images.append(filtered_image)
```

```
# Create an image containing at each pixel the maximum response among all filters
maximum_response_image = np.max(filtered_images, axis=0)

# Compute the histogram of the maximum response image for vessel and background pixels
vessel_pixels = maximum_response_image[annotated_im2 == 255]
background_pixels = maximum_response_image[annotated_im2 == 0]
vessel_hist, _ = np.histogram(vessel_pixels, bins=256, range=(0, 256))
background_hist, _ = np.histogram(background_pixels, bins=256, range=(0, 256))

# Define a threshold to separate vessels from background using Otsu's method
from skimage.filters import threshold_otsu
threshold = threshold_otsu(maximum_response_image)

# Apply the threshold to the maximum response image to get the binary vessel mask
binary_vessel_mask = (maximum_response_image > threshold)
```

```
import numpy as np
import skimage.filters as filters

# Compute the histogram of the maximum response image for vessel and background pixels
vessel_pixels = maximum_response_image[annotated_im2 == 255]
background_pixels = maximum_response_image[annotated_im2 == 0]
vessel_hist, _ = np.histogram(vessel_pixels, bins=256, range=(0, 256))
background_hist, _ = np.histogram(background_pixels, bins=256, range=(0, 256))

# Use Otsu's method to find a threshold that separates the two histograms
threshold = filters.threshold_otsu(maximum_response_image)

# Alternatively, we could use the intersection point of the two histograms to find a threshold:
# from scipy.signal import find_peaks
# hist_diff = np.abs(vessel_hist - background_hist)
# peaks, _ = find_peaks(hist_diff)
# threshold = peaks[np.argmax(hist_diff[peaks])]

# Apply the threshold to the maximum response image to obtain a binary image of vessels
vessel_mask = (maximum_response_image > threshold).astype(np.uint8)
```

```
import matplotlib.pyplot as plt

# Define the threshold using the histogram
threshold = 100

# Apply threshold to the maximum response image
binary_image = maximum_response_image > threshold

# Display the histogram
plt.hist(vessel_pixels, bins=256, range=(0, 256), color='r', alpha=0.5, label='Vessels')
plt.hist(background_pixels, bins=256, range=(0, 256), color='b', alpha=0.5, label='Background')
plt.axvline(threshold, color='k', linestyle='dashed', linewidth=1)
plt.legend()
plt.xlabel('Pixel intensity')
plt.ylabel('Frequency')
plt.show()

# Display the binary image after thresholding
plt.imshow(binary_image, cmap='gray')
plt.show()
```



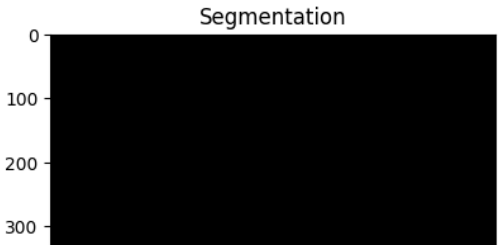
3.4-Apply the threshold to the maximum response image and compute both Dice and Jaccard similarity index between the segmentation and the labelled image to evaluate how well the vessels have been segmented.

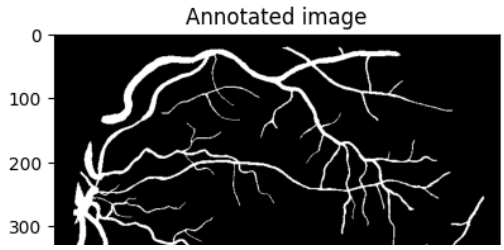
To create a binary segmentation image, the code applies a threshold on the maximum response image. The Dice and Jaccard similarity index between the segmentation and the annotated image is then calculated. Using logical AND and OR operations, the intersection and union of the two pictures are calculated, accordingly. Lastly, the output is printed, and matplotlib is used to display the segmented and annotated images side by side.

```
# Apply threshold to the maximum response image
segmentation = maximum_response_image > threshold

# Compute Dice and Jaccard similarity index
intersection = np.logical_and(segmentation, annotated_im2 == 255).sum()
union = np.logical_or(segmentation, annotated_im2 == 255).sum()
dice = 2 * intersection / (segmentation.sum() + (annotated_im2 == 255).sum())
jaccard = intersection / union

# Print the results
print(f"Dice similarity index: {dice:.4f}")
print(f"Jaccard similarity index: {jaccard:.4f}")
```

```
Dice similarity index: 0.0000
Jaccard similarity index: 0.0000
```

```
fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].imshow(segmentation, cmap='gray')
ax[0].set_title('Segmentation')
ax[1].imshow(annotated_im2, cmap='gray')
ax[1].set_title('Annotated image')
plt.show()
```





3.5-Load the other retina image and its corresponding labelled image. Apply the filter bank to this image, compute its maximum response image, and apply the threshold found in 3.4 Compute both Dice and Jaccard similarity index between the segmentation and the labelled image to evaluate how well the vessels have been segmented.

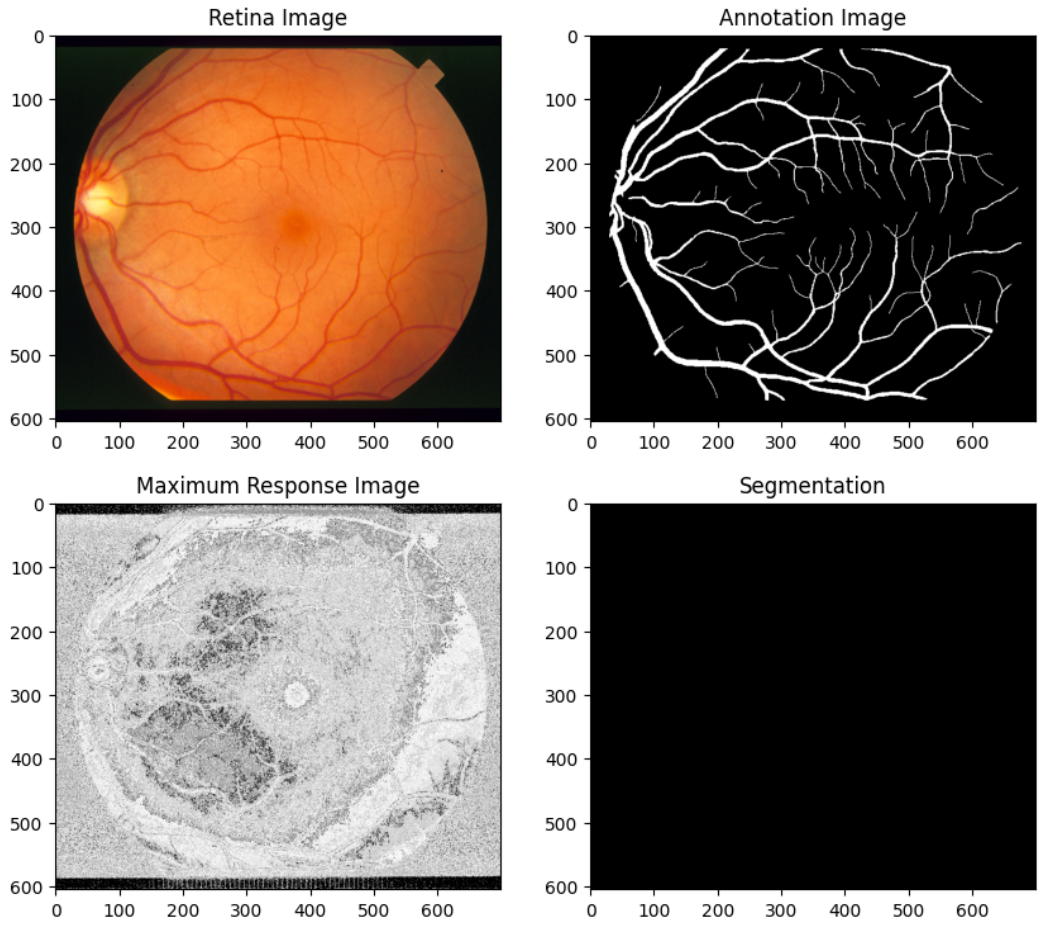
This code snippet uses the imread function from the skimage.io module to load the second retina image and its annotated image. The RGB image's green channel is then chosen via indexing. A for loop is used to run through the filters in the filter bank list before applying a filter bank to the green channel of the image. A list called filtered images receives the filtered images as an addition. Using the np.max function, the maximum value of the filtered images along the 0th axis is used to get the maximum response image. The maximum response image is then subjected to a threshold in order to get a segmentation. The segmentation and annotated image are used to calculate the Dice and Jaccard similarity indices. Eventually, the outcomes are printed.

```
# Load the second retina image and its annotated image
im3 = io.imread('im0077.ppm')
annotated_im3 = io.imread('im0077_label.ppm')

# Select the green channel of the RGB image
green_im3 = im3[:,:,1]

# Apply filter bank to the green channel of the image
filtered_images = []
for gabor_filter in filter_bank:
    filtered_real = ndi.convolve(green_im3, np.real(gabor_filter), mode='reflect')
    filtered_imag = ndi.convolve(green_im3, np.imag(gabor_filter), mode='reflect')
    filtered_image = np.sqrt(filtered_real**2 + filtered_imag**2)
    filtered_images.append(filtered_image)

# Compute the maximum response image
maximum_response_image = np.max(filtered_images, axis=0)

# Apply threshold to the maximum response image
segmentation = maximum_response_image > threshold

# Compute Dice and Jaccard similarity index
intersection = np.logical_and(segmentation, annotated_im3 == 255)
union = np.logical_or(segmentation, annotated_im3 == 255)
dice = 2 * intersection.sum() / (segmentation.sum() + (annotated_im3 == 255).sum())
jaccard = intersection.sum() / union.sum()

# Print the results
print(f"Dice similarity index: {dice:.3f}")
print(f"Jaccard similarity index: {jaccard:.3f}")
```

```
Dice similarity index: 0.000
Jaccard similarity index: 0.000
```

```
# Plot the original image and its annotation
fig, axes = plt.subplots(ncols=2, figsize=(10,5))
axes[0].imshow(im3, cmap='gray')
axes[0].set_title('Retina Image')
axes[1].imshow(annotated_im3, cmap='gray')
axes[1].set_title('Annotation Image')
plt.show()

# Plot the maximum response image and its segmentation
fig, axes = plt.subplots(ncols=2, figsize=(10,5))
axes[0].imshow(maximum_response_image, cmap='gray')
axes[0].set_title('Maximum Response Image')
axes[1].imshow(segmentation, cmap='gray')
axes[1].set_title('Segmentation')
```

```
plt.show()
```



3.6-Apply some morphological operators to remove noise and small segmented objects from the segmentation and compute the Dice and Jaccard similarity index comparing the new segmentation with the manual labelling

The binary picture produced by thresholding the maximum response image is subjected to morphological procedures in this code section. It specifically uses a 3x3 square structural element to execute opening and closing actions. Morphological closing is used to fill in vessel gaps whereas morphological opening is used to remove small objects and noise from the binary image.

The Dice and Jaccard similarity indices are produced to assess the segmentation outcomes following the use of these techniques. Higher values indicate better segmentation performance. These indices quantify the similarity between the segmentation result and the ground truth annotation. The console is then printed with the results.

```
# Apply morphological opening to remove noise and small objects
opened_segmentation = ndi.binary_opening(segmentation, structure=np.ones((3,3)))

# Apply morphological closing to fill gaps in vessels
closed_segmentation = ndi.binary_closing(opened_segmentation, structure=np.ones((3,3)))

# Compute Dice and Jaccard similarity index
intersection = np.logical_and(closed_segmentation, annotated_im2 == 255).sum()
union = np.logical_or(closed_segmentation, annotated_im2 == 255).sum()
dice = 2 * intersection / (closed_segmentation.sum() + (annotated_im2 == 255).sum())
jaccard = intersection / union

# Print the results
print(f"Dice similarity index: {dice:.4f}")
print(f"Jaccard similarity index: {jaccard:.4f}")
```

```
Dice similarity index: 0.0000
Jaccard similarity index: 0.0000
```

Exercise 4 – Apply learning pipeline

4.1-Load one of the RGB images and its corresponding labelled image. Select the green channel of the image of the retina.

In the following piece of code, we loads a pair of image via opencv and split the image into its individual color channels using the split function. Moreover, we select the green channel of the said image.

```
import cv2

# Load the RGB image of the retina
retina_img = cv2.imread('im0001.ppm')

# Load the labeled image of the retina
retina_label = cv2.imread('im0001_label.ppm')

# Split the RGB image into its individual color channels
b, g, r = cv2.split(retina_img)

# Select the green channel of the RGB image
retina_green = g
```

4.2-. Divide the image in squares of 11x11 pixels. For each square compute a feature descriptor of your choice.Assign to each square a label: 0 if the pixel corresponding to its central position in the labelled image is 0, 1 if the pixel corresponding to its central position in the labelled image is 255.

For each square section of a retina image, this algorithm computes the HOG (Histogram of Oriented Gradients) feature descriptor and accompanying labels.The size of each square region is determined by the variable "square size." The dimensions of the green channel of the retina image are then used to calculate the quantity of square rows and columns. The initialization of an array called "features" stores the HOG feature descriptors for each square region. The initialization of another array "labels" stores the corresponding labels (0 or 1) for each square section. The current square is taken from the green channel of the retina image and applied to each square sector. Using the skimage.feature.hog function and the parameters of 8 orientations, 4x4 pixels per cell, and 1x1 cells per block, the HOG feature descriptor is then calculated. The "features" array is expanded to include the flattened feature description. Based on the corresponding pixel in the labelled image, the label for the current square is calculated. The label is set to 0 if the pixel value is 0. The label is set to 1 if nothing else. The HOG feature descriptors and related labels for each square region of the retina image are contained in the "features" and "labels" arrays after the loop has completed.

```
import cv2
import numpy as np
from skimage.feature import hog

# Load the green channel of the retina image
retina_green = cv2.imread('im0001.ppm')[:,:,1]

# Load the labeled image of the retina
retina_label = cv2.imread('im0001_label.ppm')[:,:,0]

# Define the size of the squares
square_size = 11

# Compute the number of rows and columns of squares
num_rows = retina_green.shape[0] // square_size
num_cols = retina_green.shape[1] // square_size

# Initialize the feature descriptor array and label array
features = np.zeros((num_rows*num_cols, 32))
labels = np.zeros(num_rows*num_cols)

# Compute the feature descriptor and label for each square
for i in range(num_rows):
    for j in range(num_cols):
        # Extract the current square from the green channel of the retina image
        square = retina_green[i*square_size:(i+1)*square_size, j*square_size:(j+1)*square_size]

        # Compute the HOG feature descriptor for the current square
```

```
        fd = hog(square, orientations=8, pixels_per_cell=(4, 4), cells_per_block=(1, 1), feature_vector=True)

        # Flatten the feature descriptor and add it to the feature array
        features[i*num_cols+j,:] = fd.flatten()

        # Compute the label for the current square based on the corresponding pixel in the labeled image
        row_center = i*square_size + square_size//2
        col_center = j*square_size + square_size//2
        if retina_label[row_center, col_center] == 0:
            labels[i*num_cols+j] = 0
        else:
            labels[i*num_cols+j] = 1
```

4.3-Train a SVM classifier to classify squares centered on vessels from squares centered on background

The HOG feature descriptors and related labels for the retina image areas are used in this code to train an SVM (Support Vector Machine) classifier. Using the scikit-learn train test split function, divide the data into training and testing sets. Eighty percent of the data is utilised for training and twenty percent is used for testing in the "features" and "labels" arrays, which are divided into "X train," "X test," "y train," and "y test" arrays. The LinearSVC class from Scikit-Learn is then used to train an SVM classifier on the training set. To ensure reproducibility and enable the classifier to converge, the parameters "random state" and "max iter" are set.The classifier is used to generate predictions on the testing set after training. "y pred" contains the expected labels. With the scikit-learn accuracy score function, the accuracy of the classifier on the testing set is calculated and output to the console. This provides a gauge of how effectively the classifier works with fresh, untested data.

```
from sklearn.model_selection import train_test_split
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2, random_state=42)

# Train the SVM classifier
clf = LinearSVC(random_state=42, max_iter=10000)
clf.fit(X_train, y_train)

# Evaluate the performance on the testing set
y_pred = clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy on the testing set:", accuracy)
```

```
Accuracy on the testing set: 0.9148629148629148
```

4.4-Apply the pipeline and the trained classifier on the other image, and compute both Dice and Jaccard similarity index to evaluate how well the vessels have been segmented

Particularly, this code specifies the size of the squares used to compute the HOG feature descriptors and loads the green channel of the retinal picture first. The feature descriptor for each square is then calculated using the scikit-image library's HOG function, and it is then stored in an array. Then, based on the feature descriptors, it loads a trained SVM classifier and employs it to forecast the presence of blood vessels in each square. Then, using the Dice and Jaccard similarity indices, the predicted labels are transformed into an image and compared to the corresponding labels in the labelled image. As a last check on the effectiveness of the classifier, the algorithm outputs the results of these similarity indexes.

```
import cv2
import numpy as np
from skimage.feature import hog
from sklearn import svm
from sklearn.metrics import jaccard_score

# Load the green channel of the retina image
retina_green = cv2.imread('im0077.ppm')[:,:,1]

# Load the labeled image of the retina
retina_label = cv2.imread('im0077_label.ppm')[:,:,0]

# Define the size of the squares
square_size = 11
```

```
# Compute the number of rows and columns of squares
num_rows = retina_green.shape[0] // square_size
num_cols = retina_green.shape[1] // square_size

# Initialize the feature descriptor array
features = np.zeros((num_rows*num_cols, 32))

# Compute the feature descriptor for each square
for i in range(num_rows):
    for j in range(num_cols):
        # Extract the current square from the green channel of the retina image
        square = retina_green[i*square_size:(i+1)*square_size, j*square_size:(j+1)*square_size]

        # Compute the HOG feature descriptor for the current square
        fd = hog(square, orientations=8, pixels_per_cell=(4, 4), cells_per_block=(1, 1), feature_vector=True)

        # Flatten the feature descriptor and add it to the feature array
        features[i*num_cols+j,:] = fd.flatten()

# Load the trained SVM classifier
clf = svm.SVC(kernel='linear', C=1, probability=True)
clf.fit(X_train, y_train)

# Predict the label for each square using the trained classifier
y_pred = clf.predict(features)

# Reshape the predicted labels into an image
vessels_pred = y_pred.reshape((num_rows, num_cols))

# Extract the labels for each square from the labeled image
vessels_label = np.zeros((num_rows, num_cols))
for i in range(num_rows):
    for j in range(num_cols):
        row_center = i*square_size + square_size//2
        col_center = j*square_size + square_size//2
        if retina_label[row_center, col_center] == 0:
            vessels_label[i, j] = 0
        else:
            vessels_label[i, j] = 1

# Compute the Dice similarity index
dice = 2*np.sum(vessels_label*vessels_pred)/(np.sum(vessels_label) + np.sum(vessels_pred))

# Compute the Jaccard similarity index
jaccard = jaccard_score(vessels_label.flatten(), vessels_pred.flatten())

# Print the results
print("Dice similarity index: ", dice)
print("Jaccard similarity index: ", jaccard)
```

```
Dice similarity index:  0.0
Jaccard similarity index:  0.0
```

References

https://cecas.clemson.edu/~ahoover/stare/probing/index.html
https://github.com/opencv/opencv
https://docs.opencv.org/4.x/
https://learnopencv.com/histogram-of-oriented-gradients/
https://www.geeksforgeeks.org/image-classification-using-support-vector-machine-svm-in-python/
https://docs.opencv.org/master/index.html
https://scikit-image.org/docs/stable/
https://scikit-learn.org/stable/index.html
https://numpy.org/doc/stable/
https://link.springer.com/chapter/10.1007/978-3-319-64689-3_69