Q1_v2

April 26, 2024

Lets start by developing functions to compare unique fingerprints. This is heavily based on the fingerprint recognition example from class.

[ ]:
```
from os import path
if not path.exists('utils.py'): # Assumes utils is in same directory as ipynb
    !wget https://biolab.csr.unibo.it/samples/fr/files.zip
    !unzip files.zip
```

[ ]:
```
import utils
import math
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
import os
from utils import *
from ipywidgets import interact
from PIL import Image
```

Before making the GUI, we need to define functions to evaluate the fingerprint

[ ]:
```
def fingerprint_segment(fingerprint):
    # Calculate the local gradient (using Sobel filters)
    gx, gy = cv.Sobel(fingerprint, cv.CV_32F, 1, 0), cv.Sobel(fingerprint, cv.
↪CV_32F, 0, 1)
    #show((gx, 'Gx'), (gy, 'Gy'))
    # Calculate the magnitude of the gradient for each pixel
    gx2, gy2 = gx**2, gy**2
    gm = np.sqrt(gx2 + gy2)
    #show((gx2, 'Gx**2'), (gy2, 'Gy**2'), (gm, 'Gradient magnitude'))
    # Integral over a square window
    sum_gm = cv.boxFilter(gm, -1, (25, 25), normalize = False)
    #show(sum_gm, 'Integral of the gradient magnitude')
    # Use a simple threshold for segmenting the fingerprint pattern
    thr = sum_gm.max() * 0.2
    mask = cv.threshold(sum_gm, thr, 255, cv.THRESH_BINARY)[1].astype(np.uint8)
    #show(fingerprint, mask, cv.merge((mask, fingerprint, fingerprint)))

    # Return the necessary parts for next step in fingerprint detection
```

```
    return gx2, gy2, gx, gy, mask
```

```
[ ]: def local_ridge_estimation(fingerprint):
    gx2, gy2, gx, gy, mask = fingerprint_segment(fingerprint)
    W = (23, 23)
    gxx = cv.boxFilter(gx2, -1, W, normalize = False)
    gyy = cv.boxFilter(gy2, -1, W, normalize = False)
    gxy = cv.boxFilter(gx * gy, -1, W, normalize = False)
    gxx_gyy = gxx - gyy
    gxy2 = 2 * gxy

    orientations = (cv.phase(gxx_gyy, -gxy2) + np.pi) / 2 # '-' to adjust for y␣
 ↪axis direction
    sum_gxx_gyy = gxx + gyy
    strengths = np.divide(cv.sqrt((gxx_gyy**2 + gxy2**2)), sum_gxx_gyy, out=np.
 ↪zeros_like(gxx), where=sum_gxx_gyy!=0)
    #show(draw_orientations(fingerprint, orientations, strengths, mask, 1, 16),␣
 ↪'Orientation image')
    return orientations, mask
```

```
[ ]: def estimate_frequency(fingerprint):
    orientations, mask = local_ridge_estimation(fingerprint)
    region = fingerprint[10:90,80:130]
    #show(region)
    # before computing the x-signature, the region is smoothed to reduce noise
    smoothed = cv.blur(region, (5,5), -1)
    xs = np.sum(smoothed, 1) # the x-signature of the region
    #print(xs)
    x = np.arange(region.shape[0])
    #f, axarr = plt.subplots(1,2, sharey = True)
    #axarr[0].imshow(region,cmap='gray')
    #axarr[1].plot(xs, x)
    #axarr[1].set_ylim(region.shape[0]-1,0)
    #plt.show()
    # Find the indices of the x-signature local maxima
    local_maxima = np.nonzero(np.r_[False, xs[1:] > xs[:-1]] & np.r_[xs[:-1] >=␣
 ↪xs[1:], False])[0]
    x = np.arange(region.shape[0])
    #plt.plot(x, xs)
    #plt.xticks(local_maxima)
    #plt.grid(True, axis='x')
    #plt.show()
    # Calculate all the distances between consecutive peaks
    distances = local_maxima[1:] - local_maxima[:-1]
    #print(distances)
    # Estimate the ridge line period as the average of the above distances
    ridge_period = np.average(distances)
```

```
    #print(ridge_period)
    return orientations, ridge_period, mask
```

[ ]:
```
def fingerprint_enhancement(fingerprint):
    orientations, ridge_period, mask = estimate_frequency(fingerprint)
    # Create the filter bank
    or_count = 8
    gabor_bank = [gabor_kernel(ridge_period, o) for o in np.arange(0, np.pi, np.
↪pi/or_count)]
    #show(*gabor_bank)
    # Filter the whole image with each filter
    # Note that the negative image is actually used, to have white ridges on a␣
↪black background as a result
    nf = 255-fingerprint
    all_filtered = np.array([cv.filter2D(nf, cv.CV_32F, f) for f in gabor_bank])
    #show(nf, *all_filtered)
    y_coords, x_coords = np.indices(fingerprint.shape)
    # For each pixel, find the index of the closest orientation in the gabor␣
↪bank
    orientation_idx = np.round(((orientations % np.pi) / np.pi) * or_count).
↪astype(np.int32) % or_count
    # Take the corresponding convolution result for each pixel, to assemble the␣
↪final result
    filtered = all_filtered[orientation_idx, y_coords, x_coords]
    # Convert to gray scale and apply the mask
    enhanced = mask & np.clip(filtered, 0, 255).astype(np.uint8)
    #show(fingerprint, enhanced)
    return enhanced, mask
```

[ ]:
```
def detect_minutia_positions(fingerprint):
    enhanced, mask = fingerprint_enhancement(fingerprint)
    # Binarization
    _, ridge_lines = cv.threshold(enhanced, 32, 255, cv.THRESH_BINARY)
    #show(fingerprint, ridge_lines, cv.merge((ridge_lines, fingerprint,␣
↪fingerprint)))
    # Thinning
    skeleton = cv.ximgproc.thinning(ridge_lines, thinningType = cv.ximgproc.
↪THINNING_GUOHALL)
    #show(skeleton, cv.merge((fingerprint, fingerprint, skeleton)))
    def compute_crossing_number(values):
        return np.count_nonzero(values < np.roll(values, -1))
    # Create a filter that converts any 8-neighborhood into the corresponding␣
↪byte value [0,255]
    cn_filter = np.array([[  1,  2,  4],
                          [128,  0,  8],
                          [ 64, 32, 16]
```

```
                        ])
    # Create a lookup table that maps each byte value to the corresponding␣
↪crossing number
    all_8_neighborhoods = [np.array([int(d) for d in f'{x:08b}'])[::-1] for x␣
↪in range(256)]
    cn_lut = np.array([compute_crossing_number(x) for x in␣
↪all_8_neighborhoods]).astype(np.uint8)
    # Skeleton: from 0/255 to 0/1 values
    skeleton01 = np.where(skeleton!=0, 1, 0).astype(np.uint8)
    # Apply the filter to encode the 8-neighborhood of each pixel into a byte␣
↪[0,255]
    neighborhood_values = cv.filter2D(skeleton01, -1, cn_filter, borderType =␣
↪cv.BORDER_CONSTANT)
    # Apply the lookup table to obtain the crossing number of each pixel from␣
↪the byte value of its neighborhood
    cn = cv.LUT(neighborhood_values, cn_lut)
    # Keep only crossing numbers on the skeleton
    cn[skeleton==0] = 0
    # crossing number == 1 --> Termination, crossing number == 3 --> Bifurcation
    minutiae = [(x,y,cn[y,x]==1) for y, x in zip(*np.where(np.isin(cn, [1,3])))]
    #show(draw_minutiae(fingerprint, minutiae), skeleton,␣
↪draw_minutiae(skeleton, minutiae))
    # A 1-pixel background border is added to the mask before computing the␣
↪distance transform
    mask_distance = cv.distanceTransform(cv.copyMakeBorder(mask, 1, 1, 1, 1, cv.
↪BORDER_CONSTANT), cv.DIST_C, 3)[1:-1,1:-1]
    #show(mask, mask_distance)
    filtered_minutiae = list(filter(lambda m: mask_distance[m[1], m[0]]>10,␣
↪minutiae))
    #show(draw_minutiae(fingerprint, filtered_minutiae), skeleton,␣
↪draw_minutiae(skeleton, filtered_minutiae))
    return all_8_neighborhoods, neighborhood_values, filtered_minutiae, cn
```

[ ]:
```
def minutia_directions(fingerprint):
    all_8_neighborhoods, neighborhood_values, filtered_minutiae, cn =␣
↪detect_minutia_positions(fingerprint)

    def compute_next_ridge_following_directions(previous_direction, values):
        next_positions = np.argwhere(values!=0).ravel().tolist()
        if len(next_positions) > 0 and previous_direction != 8:
            # There is a previous direction: return all the next directions,␣
↪sorted according to the distance from it,
            #                                 except the direction, if any, that␣
↪corresponds to the previous position
            next_positions.sort(key = lambda d: 4 - abs(abs(d -␣
↪previous_direction) - 4))
```

```python
        if next_positions[-1] == (previous_direction + 4) % 8: # the direction of the previous position is the opposite one
            next_positions = next_positions[:-1] # removes it
    return next_positions

r2 = 2**0.5 # sqrt(2)

# The eight possible (x, y) offsets with each corresponding Euclidean distance
xy_steps = [(-1,-1,r2),( 0,-1,1),( 1,-1,r2),( 1, 0,1),( 1, 1,r2),( 0, 1,1),(-1, 1,r2),(-1, 0,1)]

# LUT: for each 8-neighborhood and each previous direction [0,8],
#      where 8 means "none", provides the list of possible directions
nd_lut = [[compute_next_ridge_following_directions(pd, x) for pd in range(9)] for x in all_8_neighborhoods]

def follow_ridge_and_compute_angle(x, y, d = 8):
    px, py = x, y
    length = 0.0
    while length < 20: # max length followed
        next_directions = nd_lut[neighborhood_values[py,px]][d]
        if len(next_directions) == 0:
            break
        # Need to check ALL possible next directions
        if (any(cn[py + xy_steps[nd][1], px + xy_steps[nd][0]] != 2 for nd in next_directions)):
            break # another minutia found: we stop here
        # Only the first direction has to be followed
        d = next_directions[0]
        ox, oy, l = xy_steps[d]
        px += ox ; py += oy ; length += l
    # check if the minimum length for a valid direction has been reached
    return math.atan2(-py+y, px-x) if length >= 10 else None

valid_minutiae = []
for x, y, term in filtered_minutiae:
    d = None
    if term: # termination: simply follow and compute the direction
        d = follow_ridge_and_compute_angle(x, y)
    else: # bifurcation: follow each of the three branches
        dirs = nd_lut[neighborhood_values[y,x]][8] # 8 means: no previous direction
        if len(dirs)==3: # only if there are exactly three branches
            angles = [follow_ridge_and_compute_angle(x+xy_steps[d][0], y+xy_steps[d][1], d) for d in dirs]
```

```
                if all(a is not None for a in angles):
                    a1, a2 = min(((angles[i], angles[(i+1)%3]) for i in␣
↪range(3)), key=lambda t: angle_abs_difference(t[0], t[1]))
                    d = angle_mean(a1, a2)
        if d is not None:
            valid_minutiae.append( (x, y, term, d) )

    #show(draw_minutiae(fingerprint, valid_minutiae))

    return valid_minutiae
```

[ ]:
```
def create_local_structures(fingerprint):
    valid_minutiae = minutia_directions(fingerprint)
    # Compute the cell coordinates of a generic local structure
    mcc_radius = 70
    mcc_size = 16

    g = 2 * mcc_radius / mcc_size
    x = np.arange(mcc_size)*g - (mcc_size/2)*g + g/2
    y = x[..., np.newaxis]
    iy, ix = np.nonzero(x**2 + y**2 <= mcc_radius**2)
    ref_cell_coords = np.column_stack((x[ix], x[iy]))
    mcc_sigma_s = 7.0
    mcc_tau_psi = 400.0
    mcc_mu_psi = 1e-2

    def Gs(t_sqr):
        """Gaussian function with zero mean and mcc_sigma_s standard␣
↪deviation, see eq. (7) in MCC paper"""
        return np.exp(-0.5 * t_sqr / (mcc_sigma_s**2)) / (math.tau**0.5 *␣
↪mcc_sigma_s)

    def Psi(v):
        """Sigmoid function that limits the contribution of dense minutiae␣
↪clusters, see eq. (4)-(5) in MCC paper"""
        return 1. / (1. + np.exp(-mcc_tau_psi * (v - mcc_mu_psi)))
    # n: number of minutiae
    # c: number of cells in a local structure

    xyd = np.array([(x,y,d) for x,y,_,d in valid_minutiae]) # matrix with all␣
↪minutiae coordinates and directions (n x 3)

    # rot: n x 2 x 2 (rotation matrix for each minutia)
    d_cos, d_sin = np.cos(xyd[:,2]).reshape((-1,1,1)), np.sin(xyd[:,2]).
↪reshape((-1,1,1))
    rot = np.block([[d_cos, d_sin], [-d_sin, d_cos]])
```

```
    # rot@ref_cell_coords.T : n x 2 x c
    # xy : n x 2
    xy = xyd[:,:2]
    # cell_coords: n x c x 2 (cell coordinates for each local structure)
    cell_coords = np.transpose(rot@ref_cell_coords.T + xy[:,:,np.
 ↪newaxis],[0,2,1])

    # cell_coords[:,:,np.newaxis,:]      :  n x c  x 1 x 2
    # xy                                 : (1 x 1) x n x 2
    # cell_coords[:,:,np.newaxis,:] - xy :  n x c  x n x 2
    # dists: n x c x n (for each cell of each local structure, the distance␣
 ↪from all minutiae)
    dists = np.sum((cell_coords[:,:,np.newaxis,:] - xy)**2, -1)

    # cs : n x c x n (the spatial contribution of each minutia to each cell of␣
 ↪each local structure)
    cs = Gs(dists)
    diag_indices = np.arange(cs.shape[0])
    cs[diag_indices,:,diag_indices] = 0 # remove the contribution of each␣
 ↪minutia to its own cells

    # local_structures : n x c (cell values for each local structure)
    local_structures = Psi(np.sum(cs, -1))

    '''@interact(i=(0,len(valid_minutiae)-1))
    def test(i=0):
        show(draw_minutiae_and_cylinder(fingerprint, ref_cell_coords,␣
 ↪valid_minutiae, local_structures, i))'''
    #print(valid_minutiae)
    return fingerprint, valid_minutiae, local_structures, ref_cell_coords
```

Here are some extra functions separate from the fingerprint recognition functions to save and load fingerprint data

[ ]:
```
def save_fingerprint(fingerprint, valid_minutiae, local_structures,␣
 ↪ref_cell_coords, name):
    directory = 'fingerprint_database'
    if not os.path.exists(directory):
        os.makedirs(directory)
    # Save fingerprint to png in database
    cv.imwrite(os.path.join(directory, f'{name}.png'), fingerprint)
    # Save other data to npz
    np.savez(os.path.join(directory, f'{name}.npz'),␣
 ↪valid_minutiae=valid_minutiae, local_structures=local_structures)
    #print('fingerprint saved')
```

[ ]:
```python
def load_fingerprint(name):
    directory = 'fingerprint_database'
    # Load fingerprint
    fingerprint = cv.imread(os.path.join(directory, f'{name}.png'))
    # Load data
    data = np.load(os.path.join(directory, f'{name}.npz'))
    valid_minutiae = data['valid_minutiae']
    #print(f'minutiae load {valid_minutiae}')
    local_structures = data['local_structures']
    return fingerprint, valid_minutiae, local_structures
```

[ ]:
```python
def compare_fingerprints(f1, m1, ls1, ref_cell_coords, name2):
    # Returns true is similarity is above a threshold, false otherwise

    f2, m2, ls2 = load_fingerprint(name2)

    # Compute all pairwise normalized Euclidean distances between local␣
↪structures in v1 and v2
    # ls1                       : n1 x  c
    # ls1[:,np.newaxis,:]       : n1 x  1 x c
    # ls2                       : (1 x) n2 x c
    # ls1[:,np.newaxis,:] - ls2 : n1 x  n2 x c
    # dists                     : n1 x  n2
    dists = np.linalg.norm(ls1[:,np.newaxis,:] - ls2, axis = -1)
    dists /= np.linalg.norm(ls1, axis = 1)[:,np.newaxis] + np.linalg.norm(ls2,␣
↪axis = 1) # Normalize as in eq. (17) of MCC paper

    # Select the num_p pairs with the smallest distances (LSS technique)
    num_p = 5 # For simplicity: a fixed number of pairs
    pairs = np.unravel_index(np.argpartition(dists, num_p, None)[:num_p], dists.
↪shape)
    score = 1 - np.mean(dists[pairs[0], pairs[1]]) # See eq. (23) in MCC paper
    #print(f'Comparison score: {score:.2f}')
    '''
    @interact(i = (0,len(pairs[0])-1), show_local_structures = False)
    def show_pairs(i=0, show_local_structures = False):
        show(draw_match_pairs(f1, m1, ls1, f2, m2, ls2, ref_cell_coords, pairs,␣
↪i, show_local_structures))
'''
    if score == 1:
        return 2, score, f2, m2
    elif score > 0.7:
        return 1, score, f2, m2
    else:
        return 0, score, f2, m2
```

To determine if the fingerprint is within the database already, we need iterate our comparison function over all the fingerprints in the database

```
[ ]: def in_database(f1, m1, ls1, ref_cell_coords):
        directory = 'fingerprint_database'
        files = os.listdir(directory)
        png_names = [os.path.splitext(file)[0] for file in files if file.endswith('.
     ↪png')]
        #print(png_names)
        best_score = 0
        best_name = ''
        best_print = None
        best_minutiae = None
        for name in png_names:
            same_fingerprint, score, f, m = compare_fingerprints(f1, m1, ls1,␣
     ↪ref_cell_coords, name)
            if same_fingerprint == 1:
                return 1, name, score, f, m
            if same_fingerprint == 2:
                return 2, name, score, f, m
            if score > best_score:
                best_score = score
                best_name = name
                best_print = f
                best_minutiae = m

        return 0, best_name, best_score, best_print, best_minutiae
```

```
[ ]: def make_minutiae_drawable(minutiae_from_file):
        return [[int(value) if index < 3 else value for index, value in␣
     ↪enumerate(minutiae)] for minutiae in minutiae_from_file]
```

Now we can create a GUI. Here I use ipywidgets for simplicity with ipynb.

```
[ ]: import os
     import io
     import ipywidgets as widgets
     from IPython.display import display
     import cv2 as cv

     # Make GUI components
     import_text = widgets.Text(placeholder='Enter Image Name (will search folder␣
     ↪specified in code for .tif)')
     fingerprint_image = widgets.Image()
     import_button = widgets.Button(description='Import Image')

     minutiae_image = widgets.Image()
     minutiae_label = widgets.Label(value='')
     minutiae_label.layout.align_self = 'center'
```

```
message_label = widgets.Label(value='')
name_text = widgets.Text(placeholder='Fingerprint Name')
upload_button = widgets.Button(description='Upload to Database')

fingerprint_folder = 'fingerprint_samples'



def on_import_button_clicked(b):
    image_name = import_text.value
    location = os.path.join(os.getcwd(), fingerprint_folder, image_name + '.
 ↪tif')
    location_png = os.path.join(os.getcwd(), fingerprint_folder, image_name + '.
 ↪png')
    #print(location_png)
    try:
        fingerprint = cv.imread(location_png, cv.IMREAD_GRAYSCALE)
        _, imbyte = cv.imencode('.png', fingerprint)
        #show(image, f'Fingerprint with size (w,h): {image.shape[::-1]}')
        fingerprint_image.value = imbyte.tobytes()
        # Get Local Structures and compare with Database
        global f, m, ls, rcc
        f, m, ls, rcc = create_local_structures(fingerprint)
        _, imbyte2 = cv.imencode('.png', draw_minutiae(fingerprint,m))
        fingerprint_image.value = imbyte2.tobytes()
        in_data, name, score, closest_f, closest_m = in_database(f, m, ls, rcc)
        _, imbyte3 = cv.imencode('.png', draw_minutiae(cv.cvtColor(closest_f,␣
 ↪cv.COLOR_BGR2GRAY), make_minutiae_drawable(closest_m)))
        minutiae_image.value = imbyte3.tobytes()
        minutiae_label.value = f'^{name} image^'
        if in_data == 0:
            message_label.value = f'Closest to {name} with similarity␣
 ↪{score*100:.0f}%\nThis fingerprint was not found in the database, you can␣
 ↪assign it a name and upload below'
            full_box.children = [left_box, middle_box, right_box2]
        elif in_data == 2:
            message_label.value = f'Fingerprint is already in database as␣
 ↪{name} with {score*100:.0f}% similarity'
            full_box.children = [left_box, middle_box, right_box1]
        elif in_data == 1:
            message_label.value = f'Fingerprint is already in database as␣
 ↪{name} with {score*100:.0f}% similarity. Do you want to save this as␣
 ↪{name}_1?'
            full_box.children = [left_box, middle_box, right_box3]
    except FileNotFoundError:
```

```
        raise FileNotFoundError(f'File location {location_png} was not found')

def on_upload_button_clicked(b):
    files = os.listdir('fingerprint_database')
    png_names = [os.path.splitext(file)[0] for file in files if file.endswith('.
 ↪png')]
    name = name_text.value
    if name == "" or name is None:
        message_label.value = 'Invalid name, try again'
    elif name in png_names:
        message_label.value = 'Name in database already, try again'
    else:
        save_fingerprint(f,m,ls,rcc,name)
        message_label.value = f'Fingerprint data was saved as {name}.png and␣
 ↪{name}.npz'
        full_box.children = [left_box, middle_box, right_box1]


upload_button.on_click(on_upload_button_clicked)
import_button.on_click(on_import_button_clicked)


left_box = widgets.VBox([fingerprint_image, import_text, import_button])
middle_box = widgets.VBox([minutiae_image, minutiae_label])
right_box1 = widgets.VBox([message_label])
right_box2 = widgets.VBox([message_label, name_text, upload_button])
right_box3 = widgets.VBox([message_label, upload_button])
full_box = widgets.HBox([left_box, middle_box, right_box1])
display(full_box)
```

```
HBox(children=(VBox(children=(Image(value=b''), Text(value='',␣
 ↪placeholder='Enter Image Name (will search fold…
```