ECE 661: Computer Vision

Homework 7, Fall 2018

Arindam Bhanja Chowdhury

abhanjac@purdue.edu

1 Overview

The goal of this homework is to implement a very simple image classification algorithm using Local Binary Pattern (LBP) features and Nearest Neighbor (NN) classifier. A train and a test dataset of images is provided containing images of 5 differet classes - beach, building, car, mountain, tree. The training dataset has 20 images of each class and the test dataset has 5 images of each class.

One sample image from each of the category is shown in the Fig 1 to 5.

The LBP features are to be extracted from all the images and a feature vector has to be constructed for each image. Then a NN classifier is used to assign the labels to the testing images by comparing feature vectors of testing images with the feature vectors of training images.

Figure 1: Sample image of beach.

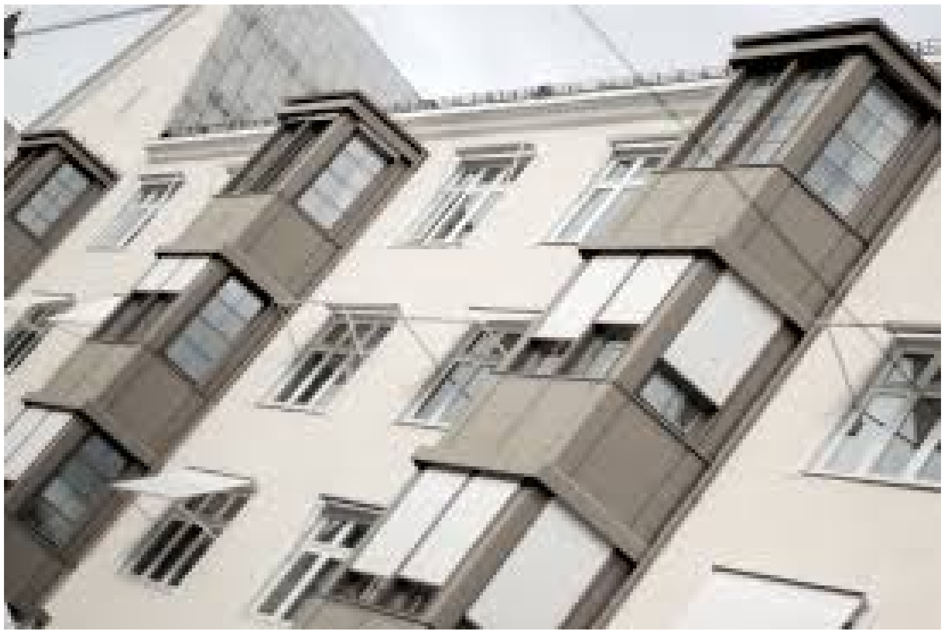

Figure 2: Sample image of building.

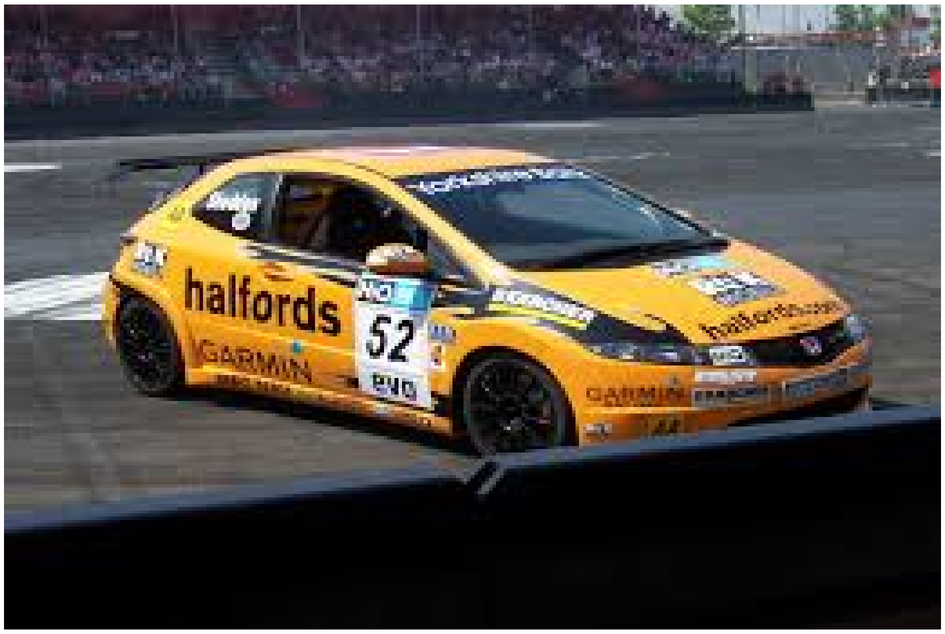

Figure 3: Sample image of car.

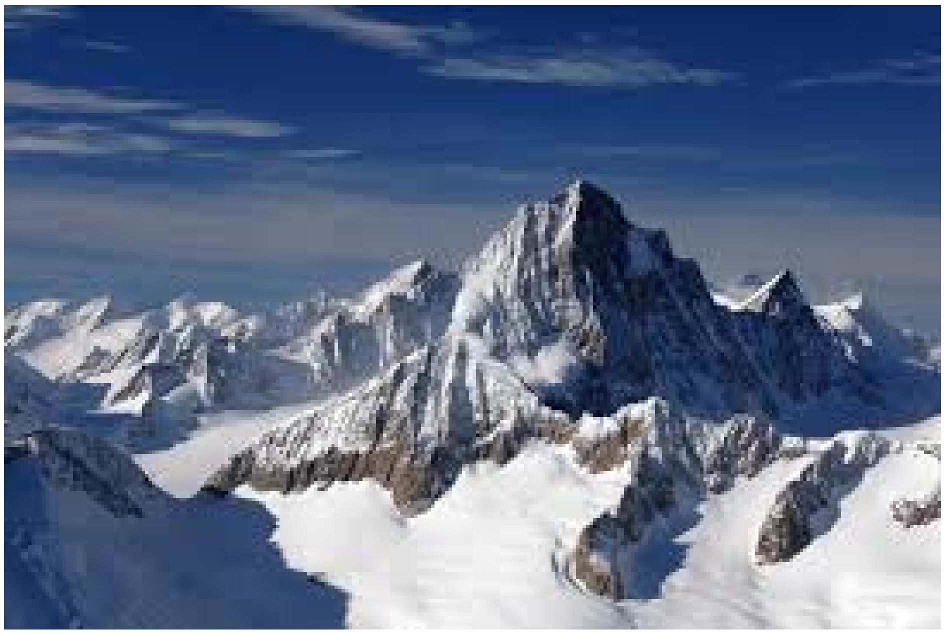

Figure 4: Sample image of mountain.

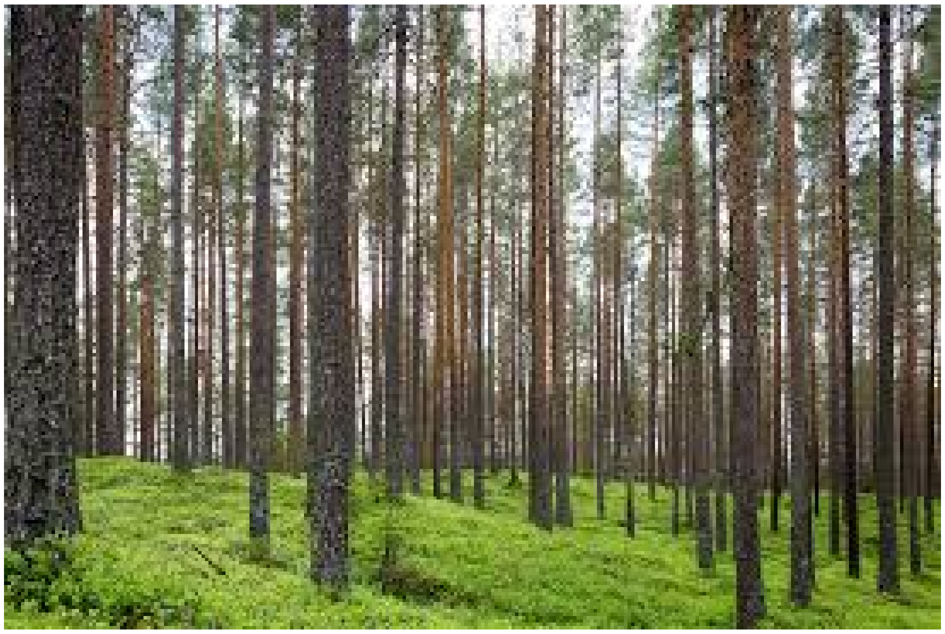

Figure 5: Sample image of tree.

2 LBP Feature Extraction

Consider the pixel a location r, c in the gray image. r is the row index and c is the column index. Then the position of P neighbors of this point in a cirular neighborhood of radius R is given by adding Δu and Δv to these r and c values. The Δu and Δv are given as follows:

$$\Delta u, \Delta v = Rcos(\frac{2\pi p}{P}), Rsin(\frac{2\pi p}{P}) \tag{1}$$

For the case of this homework P is considered as 8 and R is considered 1.

So, putting p = 0 to 7 in the above equation, we can find the coordinates of the neighbors as

$$p_r \ , \ p_c = r + u \ , \ c + \Delta v \tag{2}$$

Now these coordinates will not be exactly integer values for all the surrounding neighbors. So to find the gray level at this p_r, p_c location we have to use bilinear interpolation using the 4 neighboring corner pixels of this p_r, p_c location.

Let's say that the corner pixels are A, B, C, and D. A is the top left pixel to p_r, p_c, B is the top right, C is the bottom left and D is the bottom right pixel to p_r, p_c. Now A, B, C and D are real pixels so their coordinates can be obtained from p_r, p_c values as follows:

$$A_r \ , \ A_c = floor[p_r] \ , \ floor[p_c] \tag{3}$$
$$B_r \ , \ B_c = floor[p_r] \ , \ ceil[p_c] \tag{4}$$
$$C_r \ , \ C_c = ceil[p_r] \ , \ floor[p_c] \tag{5}$$
$$D_r \ , \ D_c = ceil[p_r] \ , \ ceil[p_c] \tag{6}$$

So the gray levels of the A, B, C, D pixels are given by:

$$A_{gray} = IMG[A_r, A_c] \tag{7}$$
$$B_{gray} = IMG[B_r, B_c] \tag{8}$$
$$C_{gray} = IMG[C_r, C_c] \tag{9}$$
$$D_{gray} = IMG[D_r, D_c] \tag{10}$$

Now, the gray level at the location p_r, p_c of the image is given by bilinear interpolation as follows:

$$IMG[p_r, p_c] = (1-\Delta K)(1-\Delta L)A_{gray} + \Delta L(1-\Delta K)B_{gray} + \Delta K(1-\Delta L)C_{gray} + \Delta K \Delta L D_{gray} \tag{11}$$

Where,

$$\Delta K \ , \ \Delta L = p_r - floor[p_r] \ , \ p_c - floor[p_c] \tag{12}$$

Now, after estimating the gtay levels at each of the P points on the circle, we threshold these gray levels with respect to the gray level value at the pixel at the center of the pixel. If the interpolated value at a point p is equal or greater than the value at the center, we set that point to 1. Otherwise, we set it to 0. This thresholding makes the pattern illumination invariant. This is explained in details in the Comment section.

Now, to make the binary pattern invariant to in-plane rotations of the image, we circularly rotate the binary pattern until a pattern with the lowest value is reached. So a a pattern like $[1, 0, 0, 1, 1, 1, 1, 1]$ after the circular rotations will stop with a minimum pattern as $[0, 0, 1, 1, 1, 1, 1, 1]$. This circular shift makes the pattern invariant to in-plane rotations. This is explained in details in the Comment section.

Now we have to encode these illumination and rotation invariant patterns with integral values.

Now the creators of LBP observed that the binary patterns which has a single run of 0's followed by a single run of 1's are the most important candidates for features.

So the encoding is made in the following way. If the binary pattern has exactly two runs, that is a run of 0's followed by a run of 1's, then corresponding code will be an integer between 0 and P-1. If the binary pattern is all 0's the code is 0. If the binary pattern is all 1's the code is P. If the binary pattern is 0's followed by 1's, the code is the number of 1's in the pattern. If the binary pattern has more than 2 runs, then the code is P+1.

So the code for the binary pattern $[0, 0, 1, 1, 1, 1, 1, 1]$ is 6, as there are 6 occurance of 1's.

Now all the patterns for all the pixels are created from an image and then a histogram with bins from 0 to P+1 is created to know how many pixels are there of each type of LBP pattern. The histogram is then normalized such that the total sum of the values of all the bins is 1.

The last P+1^{th} bin is then removed and the rest of the bins form a feature vector for the image. This is done because it is filled with the unwanted features which has more than 2 runs or 0's and 1's.

3 NN Classifier Training and Implementation

Now the feature vector of all the images are calculated. All of these are basically P+1 long vectors which represents the histogram of the LBP codes for the corresponding image. Each of the elements of this P+1 long vector gives the value of the corresponding bin in the histogram. Now the euclidean distance between the feature vectors of the test images are calculated from that of the training images. The training image with which the feature vector of the test image has the least distance, is the training image to which this test image is most similar to. The class of this training image is then given as output as the predicted class for the test image.

In this implementation we have used the class which is the predicted the maximum times by checking the K lowest distances of a test image with the training image set. Recommended K value is 5. But we got a better result by choosing K to be 7. The final LBP pattern images, and the histograms of the the images in Fig 1 to 5 are shown in the result section along with the confusion matrix.

4 Results

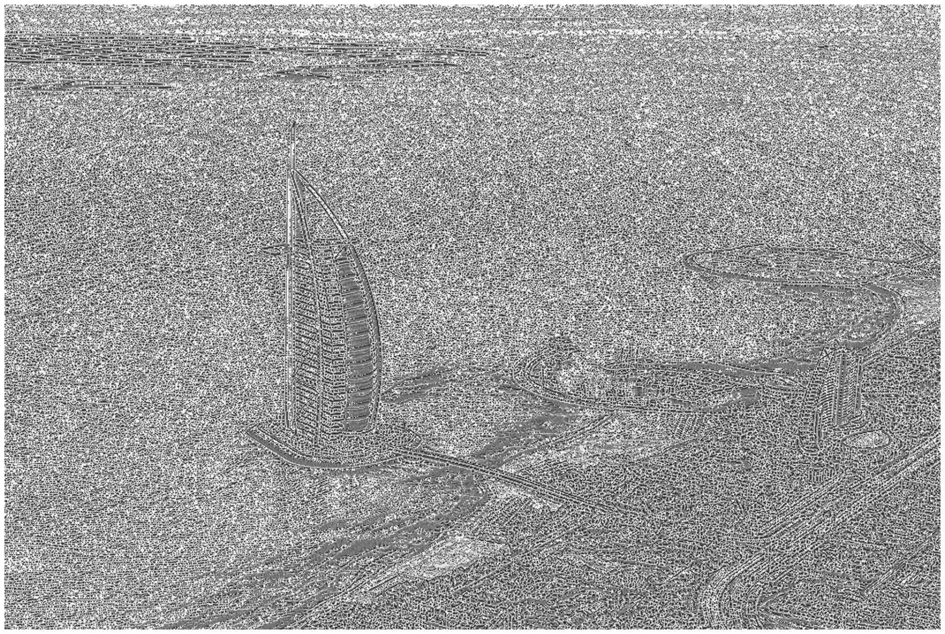

Figure 6: LBP pattern image of beach from Fig 1.

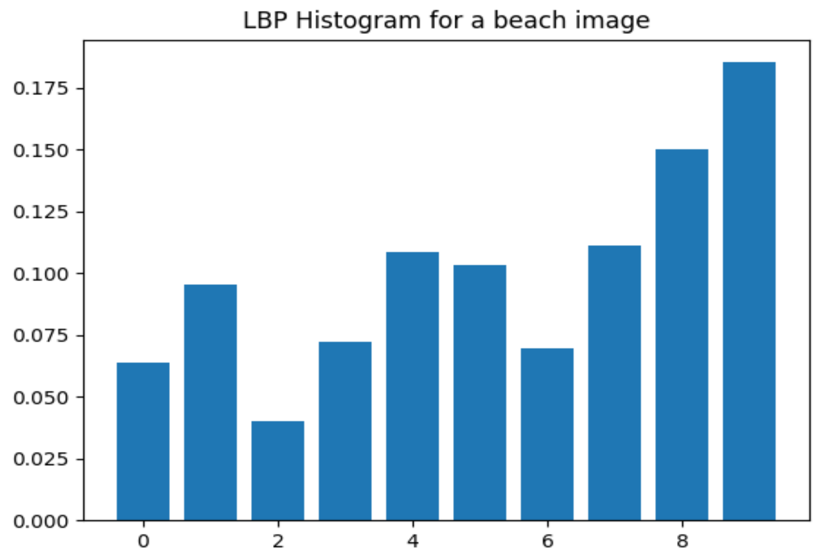

Figure 7: LBP Histogram feature vector for image of beach from Fig 1.

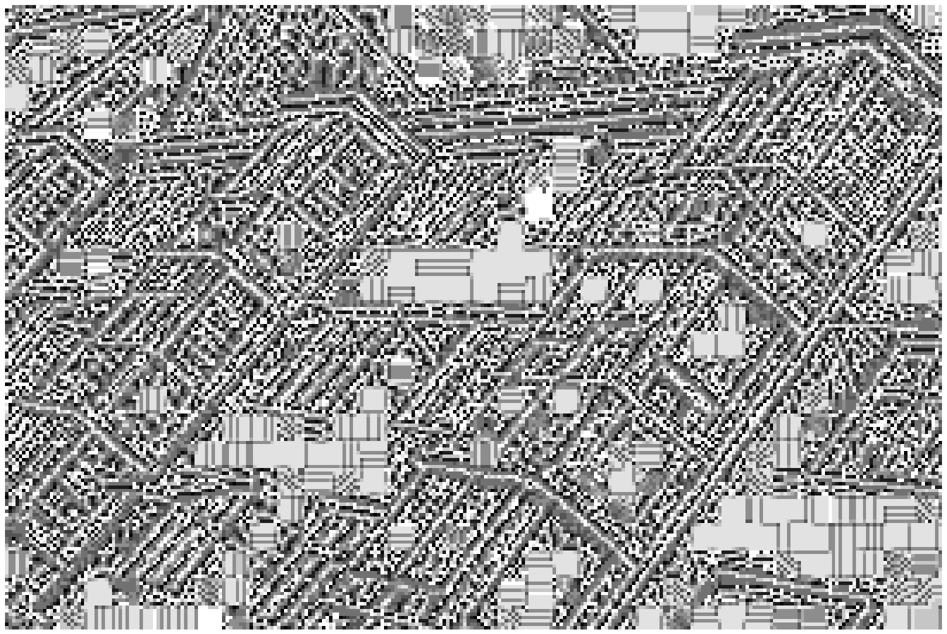

Figure 8: LBP pattern image of building from Fig 2.

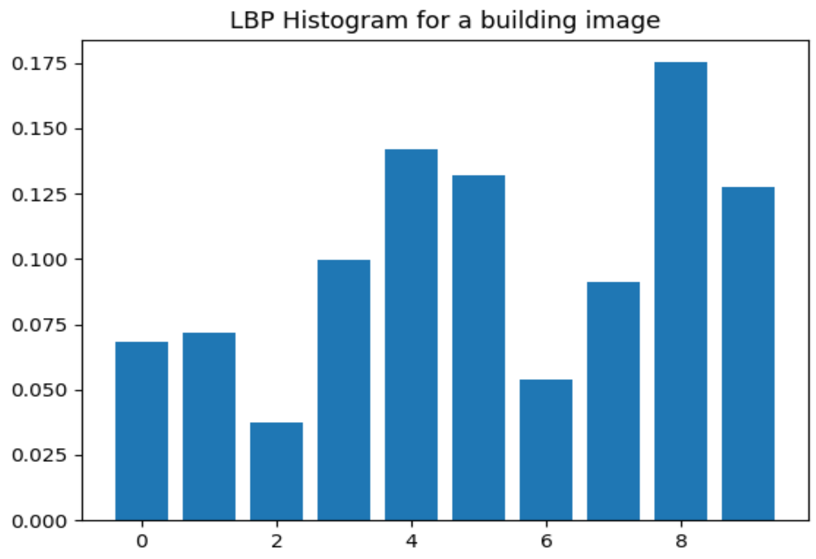

Figure 9: LBP Histogram feature vector for image of building from Fig 2.

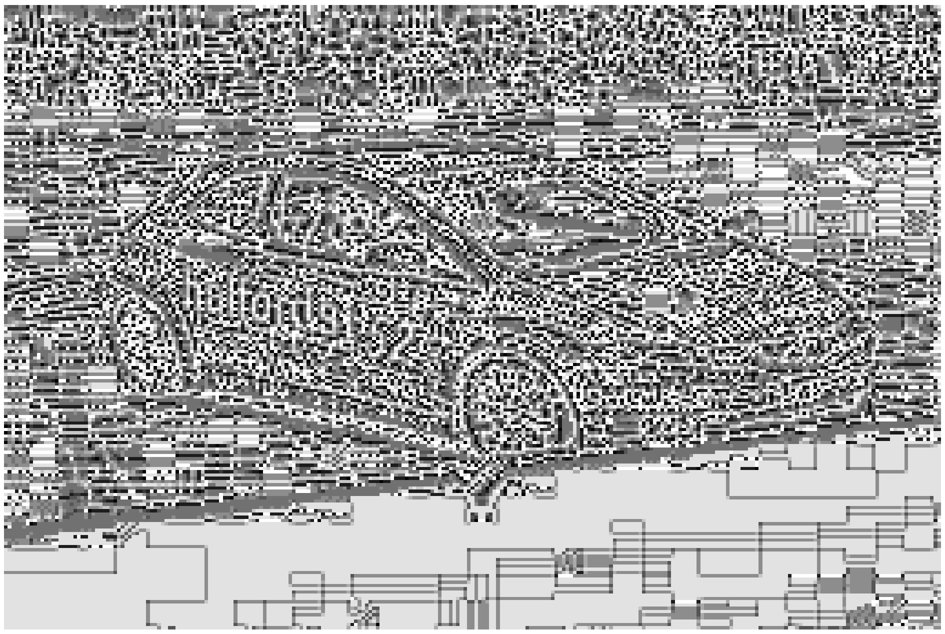

Figure 10: LBP pattern image of car from Fig 3.

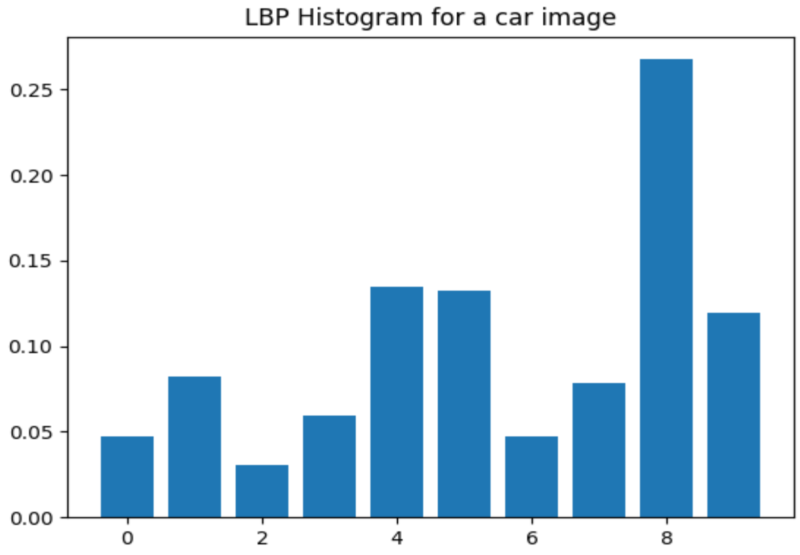

Figure 11: LBP Histogram feature vector for image of car from Fig 3.

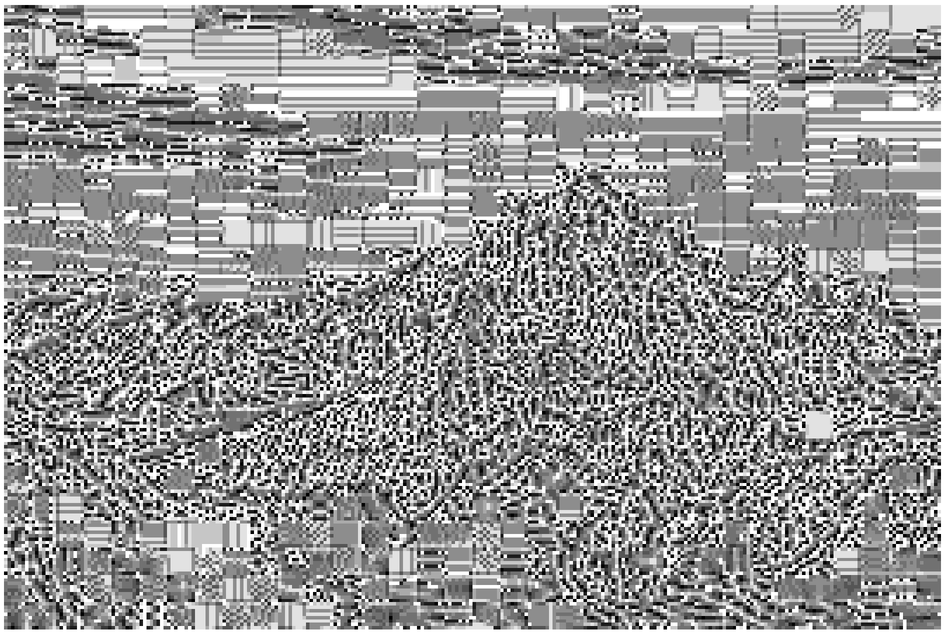

Figure 12: LBP pattern image of mountain from Fig 4.

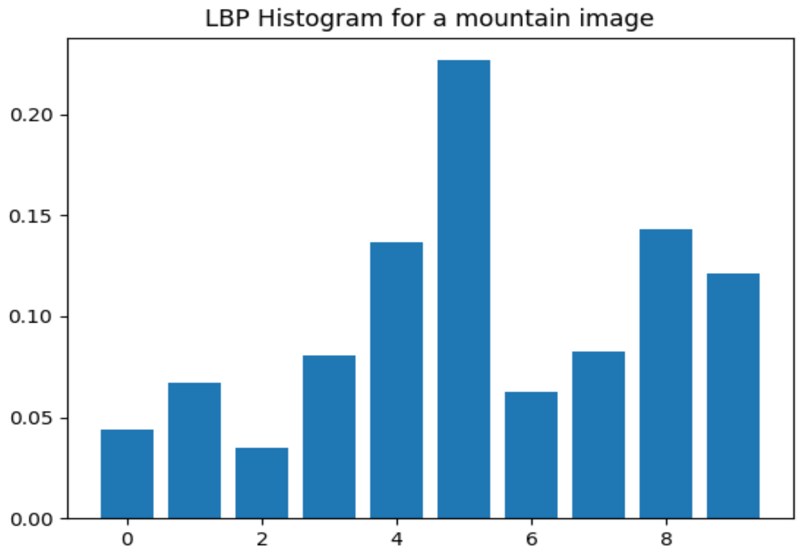

Figure 13: LBP Histogram feature vector for image of mountain from Fig 4.

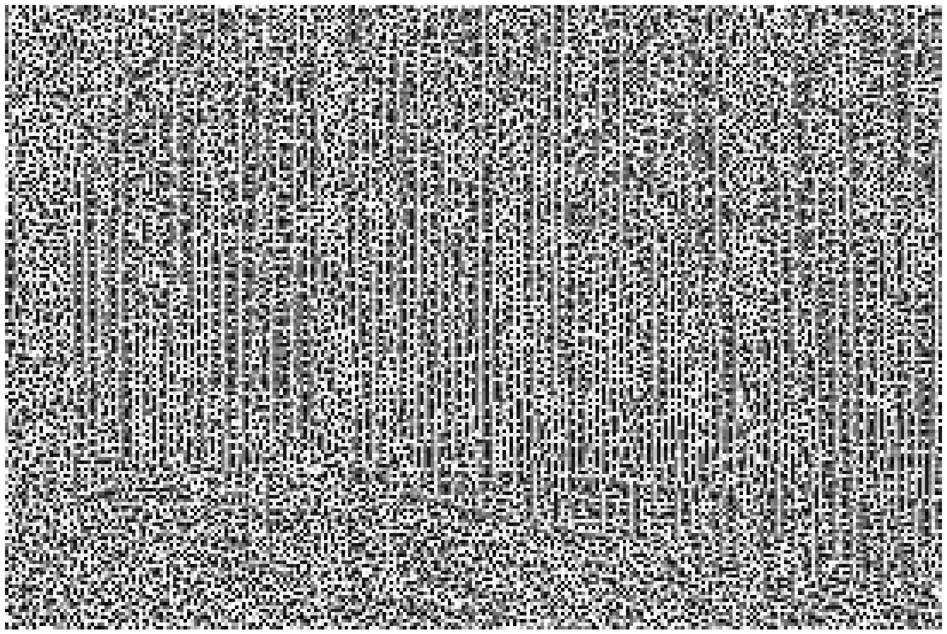

Figure 14: LBP pattern image of tree from Fig 5.

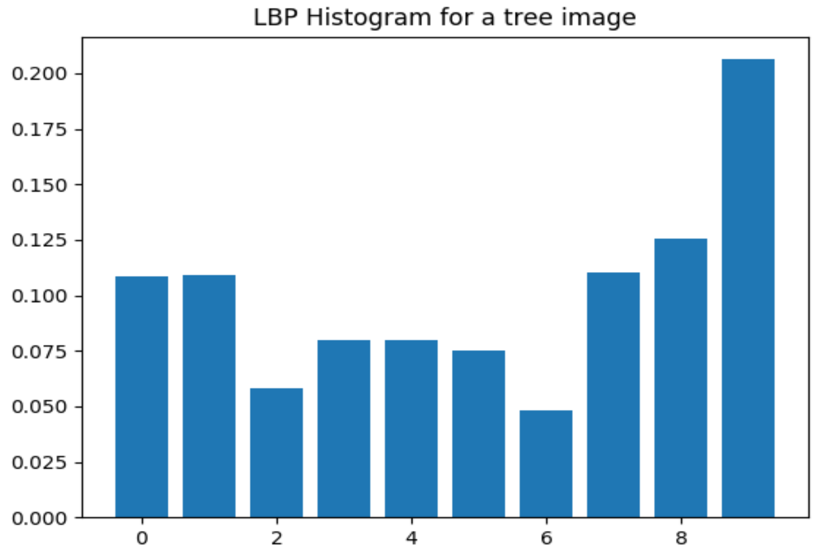

Figure 15: LBP Histogram feature vector for image of tree from Fig 5.

4.1 Confusion Matrix

	Beach	**Building**	**Car**	**Mountain**	**Tree**
Beach	4	0	0	1	0
Building	0	4	0	1	0
Car	2	1	2	0	0
Mountain	2	0	0	3	0
Tree	1	0	0	0	4

Table 1: Confusion Matrix

Overall Accuracy achieved is: **68%**

5 Observations

- The LBP and NN classifier are implemented in the exact same process as described in the previous section.
- Although the images are shown in color, they are read as grayscale images before deriving the pattern
- The confusion matrix is created that shows how the test images are misclassified and with which other class the images have been confused with.
- It shows that the class tree, beach and building has been predicted the most accurately. Car and mountain have been predicted least accurately.
- This LBP is not a very powerful classifier as per the observation. So it should be trained with more training examples to improve its performance.
- But it is pretty fast and has some great properties like illumination and rotation invariance.

6 Comments

- LBP is a good feature extractor for identifying the texture in the image. It is fast illumination and rotation invariant.
- While calculating the LBP pattern, the pixels in the surrounding neighorhoods are thresholded by the center pixel value to create the binary pattern. This is what makes the feature illumination invariant. Because if the contrast of the image changes then all the pixels in the neighborhood will have almost similar change in contrast as the center pixel. Hence, because of the thresholding, they will still end up having the same binary pattern.
- Finding the minimum value representation of the binary pattern is what makes the feature rotation invariant. Suppose, we find the binary pattern of a point in the image and then we rotate the image in plane, then find the binary pattern of the same point. These two binary patterns will be different because of the rotation. But they will have the same sequence of 1's and 0's. Hence if in both cases they are rotated such that they attain the minimum value,

then they will end up being the same binary pattern. So in-plane rotations may change the actual binary patterns, but the minimum binary pattern will stay the same.

- ecommended K value is 5. But we got a better result by choosing K to be 7.

```python
#!/usr/bin/env python

import numpy as np, cv2, os, time, math, copy, matplotlib.pyplot as plt
from scipy import signal, optimize


#===============================================================================
# ARINDAM BHANJA CHOWDHURY
# abhanjac@purdue.edu
# ECE 661 FALL 2018, HW 7
#===============================================================================

#===============================================================================
# FUNCTIONS CREATED IN HW7.
#===============================================================================

def normalize( img, scale=255 ):
    '''
    Normalize the image and scale it to the scale value.
    '''
    img = ( img - np.amin( img ) ) / ( np.amax( img ) - np.amin( img ) )
    img = img * 255
    img = np.asarray( img, dtype=np.uint8 )

    return img

#===============================================================================

def LBPatPixelLoc( img=None, x=None, y=None ):
    '''
    This function finds the Local Binary Pattern (LBP) of the input gray image at
    a pixel location given by x, y.
    '''
    if len( img.shape ) == 3:
        print( '\nERROR: Input image is not grayscale. Aborting.\n' )
        return

    imgH, imgW = img.shape

    # Return the value of the pixel themselves if they are on the boundaries.
    if x == 0 or x == imgW-1 or y == 0 or y == imgH-1:
        print( '\nERROR: Input pixels are on the boundary, cannot calculate LBPcode. ' \
                    'Aborting.\n' )
        return

#-------------------------------------------------------------------------------

    # Find the pattern.

    # Since a circular pattern is considered here, so the pixel values at the
    # 8-neighborhood of the given pixel should be weighted by the sin and cosine
    # values of angles made by their location coordinates with the center coordinate.
    P, R = 8, 1
    theta = 2 * math.pi / P

    p0 = img[y+1, x]
    p2 = img[y, x+1]
    p4 = img[y-1, x]
    p6 = img[y, x-1]

    #print( p0, p2, p4, p6 )

    # dK is along row and dL is along col.
    p = 1
    dU, dV = R * math.cos( theta * p ), R * math.sin( theta * p )
    py, px = y + dU, x + dV
    Ay, Ax = math.floor( py ), math.floor( px )
    By, Bx = math.floor( py ), math.ceil( px )
    Cy, Cx = math.ceil( py ), math.floor( px )
    Dy, Dx = math.ceil( py ), math.ceil( px )
    A, B, C, D = img[ Ay, Ax ], img[ By, Bx ], img[ Cy, Cx ], img[ Dy, Dx ]
    dK, dL = py - math.floor( py ), px - math.floor( px )
    p1 = (1-dK) * (1-dL) * A + (1-dK) * dL * B + dK * (1-dL) * C + dK * dL * D

    p = 3
    dU, dV = R * math.cos( theta * p ), R * math.sin( theta * p )
    py, px = y + dU, x + dV
    Ay, Ax = math.floor( py ), math.floor( px )
    By, Bx = math.floor( py ), math.ceil( px )
    Cy, Cx = math.ceil( py ), math.floor( px )
    Dy, Dx = math.ceil( py ), math.ceil( px )
    A, B, C, D = img[ Ay, Ax ], img[ By, Bx ], img[ Cy, Cx ], img[ Dy, Dx ]
    dK, dL = py - math.floor( py ), px - math.floor( px )
    p3 = (1-dK) * (1-dL) * A + (1-dK) * dL * B + dK * (1-dL) * C + dK * dL * D

    p = 5
    dU, dV = R * math.cos( theta * p ), R * math.sin( theta * p )
    py, px = y + dU, x + dV
    Ay, Ax = math.floor( py ), math.floor( px )
    By, Bx = math.floor( py ), math.ceil( px )
    Cy, Cx = math.ceil( py ), math.floor( px )
    Dy, Dx = math.ceil( py ), math.ceil( px )
    A, B, C, D = img[ Ay, Ax ], img[ By, Bx ], img[ Cy, Cx ], img[ Dy, Dx ]
    dK, dL = py - math.floor( py ), px - math.floor( px )
    p5 = (1-dK) * (1-dL) * A + (1-dK) * dL * B + dK * (1-dL) * C + dK * dL * D

    p = 7
    dU, dV = R * math.cos( theta * p ), R * math.sin( theta * p )
    py, px = y + dU, x + dV
    Ay, Ax = math.floor( py ), math.floor( px )
    By, Bx = math.floor( py ), math.ceil( px )
    Cy, Cx = math.ceil( py ), math.floor( px )
    Dy, Dx = math.ceil( py ), math.ceil( px )
    A, B, C, D = img[ Ay, Ax ], img[ By, Bx ], img[ Cy, Cx ], img[ Dy, Dx ]
    dK, dL = py - math.floor( py ), px - math.floor( px )
    p7 = (1-dK) * (1-dL) * A + (1-dK) * dL * B + dK * (1-dL) * C + dK * dL * D

    pList = [ p0, p1, p2, p3, p4, p5, p6, p7 ]
    #print( pList )
    pattern = [ '0' if p < img[y, x] else '1' for p in pList ]
    patternStr = ''.join( pattern )     # Converting to string.
    LBPcode = int( patternStr, 2 )      # Converting to decimal.
    #print( type( pattern ), pattern, LBPcode )
    #print( pattern )

#--------------------------------------------------------------------------------

    # Rotate the pattern to get the least decimal value.

    # Instead of rotating the pattern, the same pattern in repeated twice in sequence.
    # Then this pattern is scanned by a window of 8 to find the smallest value.
    # Scanning this repeated pattern with a window of 8 in sequence is equivalent
    # to circularly rotating the pattern.
    repeatedPattern = pattern * 2

    # Initializing.
    minPattern, minPatternStr, minLBPcode = pattern, patternStr, LBPcode

    for i in range( P ):        # P is 8 for our case.
        # Taking out a chunk of 8 consecutive elements.
        currentPattern = repeatedPattern[ i : i + P ]
        currentPatternStr = ''.join( currentPattern )   # Converting to string.
        currentLBPcode = int( currentPatternStr, 2 )    # Converting to decimal.

        if minLBPcode > currentLBPcode:     # Updating.
            minLBPcode = currentLBPcode
            minPattern = currentPattern
            minPatternStr = currentPatternStr

    #print( minLBPcode )
    #print( minPattern )

#-------------------------------------------------------------------------------

    # Calculating the number of sequences of 0's followed by 1's.

    # If the sequence is '00011011', then there are basically 2 sequences or 0's
    # followed by 1's. So this string is split by the pattern '01' the resulting
    # list length will give the number of sequences + 1.
    splittedPattern = minPatternStr.split('01')
    nSequences = len( splittedPattern ) - 1
    #print( splittedPattern )

    # The nSequences is equal to the 'number of runs' that was mentioned in lecture.

    # Encoding.
    encoding = P + 1        # This corresponds to the case with more than 1 sequences.

    if nSequences == 0:      # This is the case of all 0's (0) or all 1's (255).
        encoding = 0 if minLBPcode == 0 else P
    elif nSequences == 1:
        # In this case, sequence of 1's will appear as the second element of the list.
        # So counting the number of 1's in the second element.
        runOf1s = splittedPattern[1]

        # Converting the run of 1's (which is a string) into a list. So all the
        # 1's will now become independent elements of the list, and then the length
        # of the list + 1 will give the number of 1's.
        # The +1 is because, one 1 got removed during the split by '01'.
        encoding = len( list( runOf1s ) ) + 1

    #print( encoding )

    return encoding

#===============================================================================

if __name__ == '__main__':

    # TASK 1.1

    # Loading the images.

    trainFilepath = './imagesDatabaseHW7/training'
    testFilepath = './imagesDatabaseHW7/testing'
    exFilepath = './example'

    classNames = [ 'beach', 'building', 'car', 'mountain', 'tree' ]
    classIdx = { cl: idx for idx, cl in enumerate( classNames ) }

##-------------------------------------------------------------------------------

    ## This is only for testing if the LBPatPixelLoc function is working properly
    ## or not.
    ##img = np.reshape( [5,4,2,4,2,1,2,4,4], (3,3) )      # 1,1
    ##img = np.reshape( [2,4,0,0,0,2,0,2,4], (3,3) )      # 1,5
    ##img = np.reshape( [2,2,4,1,0,0,4,0,2], (3,3) )      # 1,6
    ##img = np.reshape( [2,4,2,1,2,1,4,0,4], (3,3) )      # 1,3
    #img = np.reshape( [4,2,1,2,4,4,4,1,5], (3,3) )      # 2,1
    ##img = np.reshape( [2,1,2,4,4,0,1,5,0], (3,3) )      # 2,2

    #print(img)
    #LBPatPixelLoc( img, 1, 1 )

##-------------------------------------------------------------------------------

    #img = cv2.imread( 'beach.jpg', 0 )

    #img = np.array( [ [5,4,2,4,2,2,4,0],
                      #[4,2,1,2,1,0,0,2],
                      #[2,4,4,0,4,0,2,4],
                      #[4,1,5,0,4,0,5,5],
                      #[0,4,4,5,0,0,3,2],
                      #[2,0,4,3,0,3,1,2],
                      #[5,1,0,0,5,4,2,3],
                      #[1,0,0,4,5,5,0,1] ] )

    #imgH, imgW = img.shape
    #print( img )
    #print( imgH, imgW )

    #LBPimg = np.zeros( ( imgH, imgW ), dtype=np.uint8 )

    #for y in range( 1, imgH-1 ):        # Ignoring the boundary pixels.
        #for x in range( 1, imgW-1 ):    # Ignoring the boundary pixels.
            #print(y, x)
            #encoding = LBPatPixelLoc( img, x, y )
            #LBPimg[ y, x ] = encoding

    ## Neglecting the bounding rows and columns of the image as nothing
    ## are encoded to those pixels. They are just 0s. But if they are not
    ## removed, then the counts of the number of actual encoded 0s in the
    ## histogram will be disrupted.
    ## So neglecting them before calculating the historgram.
    #LBPimg = LBPimg[ 1 : imgH-1, 1 : imgW-1 ]

    #P = 8
    #hist = cv2.calcHist( [LBPimg], channels=[0], mask=None, histSize=[P+2], \
                                            #ranges=[0, P+2] )
    ## Since different images can have different number of pixels, so the
    ## histogram has to be normalized before comparison. This is done by
    ## dividing the counts in all the bins by the total count of all bins.
    ##hist = hist / np.sum( hist )

    #hist = np.reshape( hist, (P+2) )    # Reshaping before plotting.

    #fig1 = plt.figure(1)
    #fig1.gca().cla()
    #plt.bar( np.arange( P+2 ), hist )  # Plot histogram.
    ##plt.show()

#==============================================================================

    # Training Set.

    # The histogram of all the training images are stored in a common list.
    # The corresponding class index of the images are also stored in another list.
    listOfTrainImgHist, listOfTrainImgClassIdx = [], []
    for clIdx, cl in enumerate( classNames ):
        trainingFolder = os.path.join( trainFilepath, cl )
        listOfImgs = os.listdir( trainingFolder )

        for idx, i in enumerate( listOfImgs ):

            imgFilePath = os.path.join( trainingFolder, i )
            img = cv2.imread( imgFilePath, 0 )      # Read image as grayscale.

            imgH, imgW = img.shape
            print( f'{idx+1}: {imgFilePath}, {imgW}x{imgH}' )

            LBPimg = np.zeros( ( imgH, imgW ), dtype=np.uint8 )

            for y in range( 1, imgH-1 ):        # Ignoring the boundary pixels.
                for x in range( 1, imgW-1 ):    # Ignoring the boundary pixels.
                    #print(y, x)
                    encoding = LBPatPixelLoc( img, x, y )
                    LBPimg[ y, x ] = encoding

            # Neglecting the bounding rows and columns of the image as nothing
            # are encoded to those pixels. They are just 0s. But if they are not
            # removed, then the counts of the number of actual encoded 0s in the
            # histogram will be disrupted.
            # So neglecting them before calculating the historgram.
            LBPimg = LBPimg[ 1 : imgH-1, 1 : imgW-1 ]

            P = 8
            hist = cv2.calcHist( [LBPimg], channels=[0], mask=None, histSize=[P+2], \
                                                         ranges=[0, P+2] )
            # Since different images can have different number of pixels, so the
            # histogram has to be normalized before comparison. This is done by
            # dividing the counts in all the bins by the total count of all bins.
            hist = hist / np.sum( hist )

            hist = np.reshape( hist, (P+2) )    # Reshaping before plotting.

            listOfTrainImgHist.append( hist )   # Storing the histogram in array.
            listOfTrainImgClassIdx.append( classIdx[ cl ] )

    # Converting the lists to arrays and saving them.
    arrOfTrainImgHist = np.array( listOfTrainImgHist )
    arrOfTrainImgClassIdx = np.array( listOfTrainImgClassIdx )

    np.savez( 'train_hist_arrays.npz', arrOfTrainImgHist, arrOfTrainImgClassIdx )
    print( 'File saved.' )

#==============================================================================

    # Testing Set.

    # The histogram of all the testing images are stored in a common list.
    # The corresponding class index of the images are also stored in another list.
    listOfTestImgHist, listOfTestImgClassIdx = [], []

    testingFolder = testFilepath
    listOfImgs = os.listdir( testingFolder )

    for idx, i in enumerate( listOfImgs ):

        imgFilePath = os.path.join( testingFolder, i )
        img = cv2.imread( imgFilePath, 0 )      # Read image as grayscale.

        imgH, imgW = img.shape
        print( f'{idx+1}: {imgFilePath}, {imgW}x{imgH}' )
        LBPimg = np.zeros( ( imgH, imgW ), dtype=np.uint8 )

        for y in range( 1, imgH-1 ):        # Ignoring the boundary pixels.
            for x in range( 1, imgW-1 ):    # Ignoring the boundary pixels.
                #print(y, x)
                encoding = LBPatPixelLoc( img, x, y )
                LBPimg[ y, x ] = encoding

        # Neglecting the bounding rows and columns of the image as nothing
        # are encoded to those pixels. They are just 0s. But if they are not
        # removed, then the counts of the number of actual encoded 0s in the
        # histogram will be disrupted.
        # So neglecting them before calculating the historgram.
        LBPimg = LBPimg[ 1 : imgH-1, 1 : imgW-1 ]

        P = 8
        hist = cv2.calcHist( [LBPimg], channels=[0], mask=None, histSize=[P+2], \
                                                ranges=[0, P+2] )
        # Since different images can have different number of pixels, so the
        # histogram has to be normalized before comparison. This is done by
        # dividing the counts in all the bins by the total count of all bins.
        hist = hist / np.sum( hist )

        hist = np.reshape( hist, (P+2) )    # Reshaping before plotting.

        listOfTestImgHist.append( hist )   # Storing the histogram in array.

        cl = i.split('_')[0]    # This is the class label of the test image.

        listOfTestImgClassIdx.append( classIdx[ cl ] )

    # Converting the lists to arrays and saving them.
    arrOfTestImgHist = np.array( listOfTestImgHist )
    arrOfTestImgClassIdx = np.array( listOfTestImgClassIdx )

    np.savez( 'test_hist_arrays.npz', arrOfTestImgHist, arrOfTestImgClassIdx )
    print( 'File saved.' )

#================================================================================

    # Example Set.

    # The histogram of all the Example images are stored in a common list.
    # The corresponding class index of the images are also stored in another list.
    listOfExImgHist, listOfExImgClassIdx = [], []

    exFolder = exFilepath
    listOfImgs = os.listdir( exFolder )

    for idx, i in enumerate( listOfImgs ):

        imgFilePath = os.path.join( exFolder, i )
        img = cv2.imread( imgFilePath, 0 )      # Read image as grayscale.

        imgH, imgW = img.shape
        print( f'{idx+1}: {imgFilePath}, {imgW}x{imgH}' )

        LBPimg = np.zeros( ( imgH, imgW ), dtype=np.uint8 )

        for y in range( 1, imgH-1 ):        # Ignoring the boundary pixels.
            for x in range( 1, imgW-1 ):    # Ignoring the boundary pixels.
                #print(y, x)
                encoding = LBPatPixelLoc( img, x, y )
                LBPimg[ y, x ] = encoding
        # Neglecting the bounding rows and columns of the image as nothing
        # are encoded to those pixels. They are just 0s. But if they are not
        # removed, then the counts of the number of actual encoded 0s in the
        # histogram will be disrupted.
        # So neglecting them before calculating the historgram.
        LBPimg = LBPimg[ 1 : imgH-1, 1 : imgW-1 ]

        P = 8
        hist = cv2.calcHist( [LBPimg], channels=[0], mask=None, histSize=[P+2], \
                                                  ranges=[0, P+2] )
        # Since different images can have different number of pixels, so the
        # histogram has to be normalized before comparison. This is done by
        # dividing the counts in all the bins by the total count of all bins.
        hist = hist / np.sum( hist )

        hist = np.reshape( hist, (P+2) )    # Reshaping before plotting.

        listOfExImgHist.append( hist )   # Storing the histogram in array.

        cl = i.split('.')[0]    # This is the class label of the example image.

        listOfExImgClassIdx.append( classIdx[ cl ] )

        fig1 = plt.figure(1)
        fig1.gca().cla()
        plt.bar( np.arange( P+2 ), hist )  # Plot histogram.
        plt.title( f'LBP Histogram for a {cl} image' )
        #plt.show()
        fig1.savefig( os.path.join( exFilepath, f'LBP_Histogram_{cl}_image.png' ) )

        LBPimg = normalize( LBPimg )
        #cv2.imshow( 'LBPimg', LBPimg )
        #cv2.waitKey(0)
        cv2.imwrite( os.path.join( exFilepath, f'LBP_image_{cl}.png' ), LBPimg )

    # Converting the lists to arrays and saving them.
    arrOfExImgHist = np.array( listOfExImgHist )
    arrOfExImgClassIdx = np.array( listOfExImgClassIdx )

    np.savez( 'example_hist_arrays.npz', arrOfExImgHist, arrOfExImgClassIdx )
    print( 'File saved.' )

#================================================================================

    # Nearest Neighbor Classification.

    npzFile = np.load( 'train_hist_arrays.npz' )

    arrOfTrainImgHist = npzFile[ 'arr_0' ]

    # Removing last bin as it consists of unwanted features.
    arrOfTrainImgHist = arrOfTrainImgHist[ :, :-1 ]

    arrOfTrainImgClassIdx = npzFile[ 'arr_1' ]

    print( arrOfTrainImgHist.shape )
    print( arrOfTrainImgClassIdx.shape )

    #npzFile = np.load( 'example_hist_arrays.npz' )
    npzFile = np.load( 'test_hist_arrays.npz' )

    arrOfTestImgHist = npzFile[ 'arr_0' ]

    # Removing last bin as it consists of unwanted features.
    arrOfTestImgHist = arrOfTestImgHist[ :, :-1 ]

    arrOfTestImgClassIdx = npzFile[ 'arr_1' ]

    print( arrOfTestImgHist.shape )
    print( arrOfTestImgClassIdx.shape )


    nTrainImgs = arrOfTrainImgHist.shape[0]
    nTestImgs = arrOfTestImgHist.shape[0]

    accuracy = 0

    for i in range( nTestImgs ):
        testHist = arrOfTestImgHist[i]
        testClassIdx = arrOfTestImgClassIdx[i]

        # Repeating the test hist and the corresponding test class idx as many
        # number of times as there are training images. This is done for ease of
        # subtraction.
        testHistArray = np.ones( ( nTrainImgs, 1 ) ) * testHist

        distArray = arrOfTrainImgHist - testHistArray
        distNormArray = np.linalg.norm( distArray, axis=1 )

        sortedIndex = np.argsort( distNormArray )   # Sorted indices.
        sortedDist = np.sort( distNormArray )       # Sorted values.

        # The shortest distance will correspond to the predicted class distance.
        # So the index (in the trainClassIdxArray) corresponding to this shortest
        # distance will give the predicted class.

        # Now we have to find the class corresponding to the least 5 distances.
        # The one which appears the most among these 5 choices, will be considered
        # the predicted class.
        predictionList = [0,0,0,0,0]

        # K for Nearest Neighbor classifier.
        K = 7

        for k in range( K ):
            predClassIdx = arrOfTrainImgClassIdx[ sortedIndex[ k ] ]
            # Increment the element of the predictionList corresponding to the
            # current predClassIdx.
            predictionList[ predClassIdx ] += 1

        #print( predictionList, '   ', end='' )      # Match pattern.

        predictionProbArray = np.array( predictionList ) * 100 / 7
        fig2 = plt.figure(2)
        fig2.gca().cla()
        plt.bar( classNames, predictionProbArray, color='r' )
        #plt.show()
        fig2.savefig( os.path.join( exFilepath, f'Prediction_prob_{classNames[i]}_image.png' ) )

        predClassIdx = predictionList.index( max( predictionList ) )
        #predClassIdx = arrOfTrainImgClassIdx[ sortedIndex[ 0 ] ]

        print( f'{i+1}, True: {testClassIdx}, Predicted: {predClassIdx}' )

        if predClassIdx == testClassIdx:        accuracy += 1

    accuracy = accuracy * 100 / nTestImgs

    print( f'Test Accuracy: {accuracy} % (K = {K}).' )
```