Purdue University

ECE 661 Computer Vision

Homework 7

Submission: Arjun Kramadhati Gopi

Email: akramadh@purdue.edu

Theory Questions

The reading material for Lecture 15 presents three different approaches to characterizing the texture in an image: 1) using the Grayscale Co- Occurrence Matrix (GLCM); 2) with Local Binary Pattern (LBP) his- tograms; and 3) using a Gabor Filter Family. Explain succinctly the core ideas in each of these three methods for measuring texture in im- ages. (You are not expected to write more than a dozen sentences on each).

Grayscale Co-Occurrence Matrix - GLCM

- Roster scan image pixel by pixel - left to right and top to bottom.
- Set parameter d for the distance value.
- For each pixel in the scan, consider pixel at distance d.
- Increment the matrix GLCM at [gl1,gl2] and [gl2,gl1] by 1. Where gl1 is the greylevel at the given pixel and gl2 is the greylevel at the point at distance d.
- Characterise the matrix with factors like entropy, homogeneity and contrast. This will be the feature vector for the image used to train and classify.

Local Binary Patterns - LBP

- For each pixel in image, consider a circular boundary of radius R on which there are p number of points.
- Calculate greylevels at each of the p points on the circle.
- If greylevel at the point is lower than level at center then assign 0, else 1. Obtain rotation invariant pattern.
- Construct histogram of the frequency of the bin encoding. This will be the texture feature of the image.

Gabor Filter Family

- It banks on the property that most textures have repetitively occuring micro-patterns.
- Can be thought of as a highly localised Fourier transform in which the localization is achieved by applying a Gaussian decay function to the pixels.
- Gaussian weighting gives the localization of the pattern.
- The direction of the periodicities in the underlying Fourier kernel is used to characterize a texture in that direction.

*With regard to representing color in images, answer Right or Wrong for the following questions: (a) RGB and HSI are just linear variants of each other. (b) The color space L*a*b* is a nonlinear model of color perception. (c) Measuring the true color of the surface of an object is made difficult by the spectral composition of the illumination.*

(a) RGB and HSI are just linear variants of each other - **RIGHT**
(b) The color space L*a*b* is a nonlinear model of color perception - **RIGHT**
(c) Measuring the true color of the surface of an object is made difficult by the spectral composition of the illumination - **RIGHT**

Programming Tasks

The task is to build an image classification algorithm using the Local Binary Pattern (LBP) features and K-Nearest Neighbors (kNN) classifier. We will briefly discuss the following main concepts:

1. LBP algorithm
2. kNN algorithm

Local Binary Patterns (LBP)

The LBP algorithm is used to generate texture features from images. The texture features are invariant to transnational and rotational changes. To achieve this, we will have to accomplish the following tasks:

1. Extract binary patterns of inter-pixel variations in the image.
2. Generate rotation invariant representations from the Local Binary Patterns.
3. Encoding the minIntVal forms of the Local Binary Patters.

Extract binary patterns of the inter-pixel variations of the Image

The main idea behind this is to run 0s and 1s around a given pixel. What this means is that for each pixel, we define a circular boundary of radius **R** on which reside **P** number of points.

```
-----------------------------------------------
    |       |       |       |       |       |
    |       |       |       |       |       |
    |       |       |       |       |       |
-----------------------------------------------
    |       |       | p=4   |       |       |
    |       |  p=5  |   x   | p=3   |       |
    |       |    x  |       | x     |       |
-----------------------------------------------
    |       |       |       |       |       |
    |       |   x   |   X   |   x   |       |
    |       | p=6   |       |  p=2  |       |
-----------------------------------------------
    |       |     x |       | x     |       |
    |       |   p=7 |   x   | p=1   |       |
    |       |       | p=0   |       |       |
-----------------------------------------------
    |       |       |       |       |       |
    |       |       |       |       |       |
    |       |       |       |       |       |
-----------------------------------------------
```

Figure 1: **Image source: Dr. Avi Kak's tutorial on Modelling texture and color in images**

In the figure above, we see 8 neighboring points on the circle of unit radius. We estimate the coordinates of these points using the following relation:

$$(\Delta u, \Delta v)_p = (R\cos\frac{2\pi p}{P}, R\sin\frac{2\pi p}{P}) \tag{1}$$

Where p ranges from 0 to P-1. Here, $P = 8$ and $R = 1$. We compute the coordinates of the points by adding the $\Delta u, \Delta v$ values to the x and y coordinates. Once we have the coordinates of these points on the circle, our job is to estimate the pixel values at those coordinates. But, it is evident that the estimated point coordinates will not exactly be an actual pixel coordinate in the image. Therefore, we will need to interpolate the pixel value at these coordinates using the 4 surrounding pixel values from actual pixels.

```
                    \Del l
                  --------->
                           .
                 A         .      B
                   --------------
           |      |        .      |
  \Del k   |      |        .      |
           |      |        .      |
           V      |        .      |
           .........|....... x      |
                  |               |
                   --------------
                 C                D
```

Figure 2: **Image source: Dr. Avi Kak's tutorial on Modelling texture and color in images**

We consider 4 pixels A, B, C and D as shown in the figure above. These pixels form a box which encapsulates the estimated point coordinate at p. Using the four known pixel values, we then interpolate the pixel value at p using the following relation:

$$greylevelatp = (1-\Delta k)(1-\Delta l)A + (1-\Delta k)\Delta l B + \Delta k(1-\Delta l)C + \Delta k\Delta l D \tag{2}$$

Using the above equation, we compute the grey levels at each of the 8 points on the circle. We then threshold these 8 values against the grey level at the center of the circle which is the pixel under consideration from the beginning. If the grey level at the point is greater than or equal to the level at the center then we assign a value of 1 to that point. If it is less than the grey level at the center we assign a value of 0 to the point. Thereby we obtain a binary vector for each of the pixel in the image. We use these binary vectors to generate rotation invariant representations in the next step.

Generate rotation invariant representations of the Local Binary Patterns

The idea behind generating a rotation invariant representation is to find the minIntVal by rotating the image around the pixel under consideration. When we rotate the image around the pixel at the center of the given circle, we will see that the points on the circle will also move in the same circular manner. These points simply shift from one place to another on the same circle. For each such shift (or rotation) we calculate the binary vector for the same center point/pixel. We then convert these binary representations to their decimal equivalent values. The least decimal value in this list would be that of the

binary pattern which is rotation invariant!
The idea behind this logic is derived from the idea that most number of 0's occupy the most significant positions when they are representing the rotation invariant form.
In this homework we make use of Dr. Avinash Kak's BitVector module on top of some own implementations to get the rotation invariant representations of the Local Binary Patterns.

Encoding the minIntVal forms of the Local Binary Patters

In the previous step, we saw how we obtained the rotation invariant representation of the LBP at each pixel in the entire image. These representation are pixel wise representations. We need an encoding system to make this an image-wise representation from the pixel-wise representation that it is at the moment. According to the creators of the LBP, we can make use of the following rules to encode the pixels in the image based on their minIntValues. The rules are as follows:

1. If the binary sequence consists only of 0s we encode the pixel as 0
2. If the binary sequence consists only of 1s we encode the pixel as P. Where P is the total number of points on the circular vicinity.
3. If the number of runs is 2 then the encoding of the pixel is the number of 1s in the sequence.
4. If the number of runs is greater than 2 then the encoding of the pixel is P+1.

Using the encoding of every pixel in the entire image, we construct a normalised histogram which has P+2 bins starting from 0 and ending at the P+1th bin.
This histogram will act as the feature vector for the image.

k-Nearest Neighbors (kNN)

Once we obtain the feature vectors of every image in the training data set, the question is how to use them to predict the image classes for the images in the testing data set. For this, we make use of the kNN algorithm.
The idea is very simple. We store the feature vector of each image matched against the class of the image. For every new image we need predictions for:

1. We calculate the feature vector of the image.
2. We then estimate a list of k matches whose feature vectors resembles the feature vector of the image.
3. Out of the k matches, we chose the image class which repeats the most number of times in the list.
4. This label is then assigned to the given image and the prediction is deemed complete.

To compute the k nearest matches, we use a variety of distance calculations to measure the entries with the least distance. Eucledian distance, dot product distance and cosine distance are few of the popular ones. In this homework I have implemented the Eucledian based matching algorithm.

Results

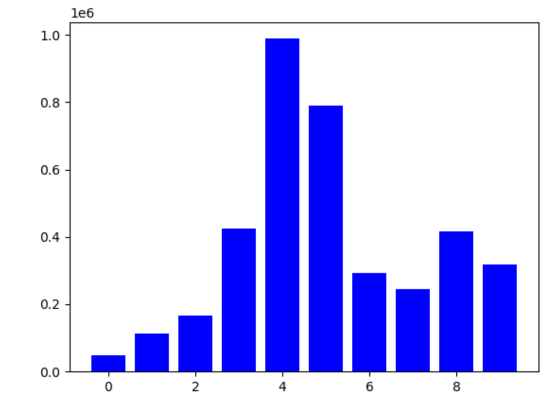

Figure 3: **Class Beach - Sample LBP Histogram**

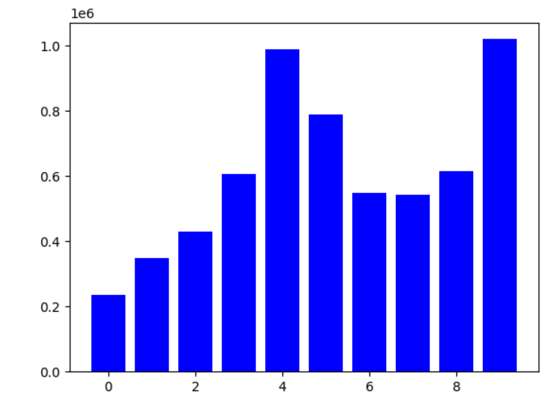

Figure 4: **Class Building - Sample LBP Histogram**

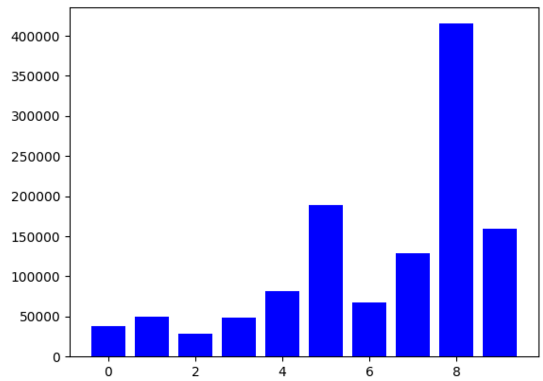

Figure 5: **Class Mountain - Sample LBP Histogram**

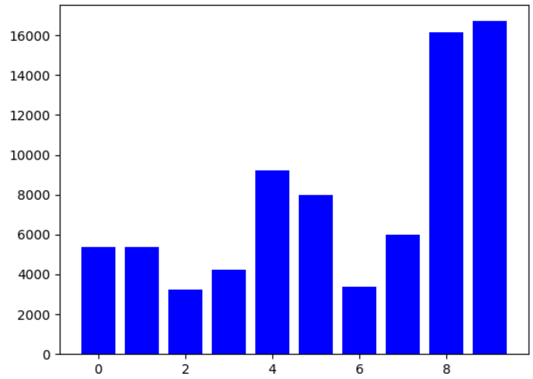

Figure 6: **Class Car - Sample LBP Histogram**

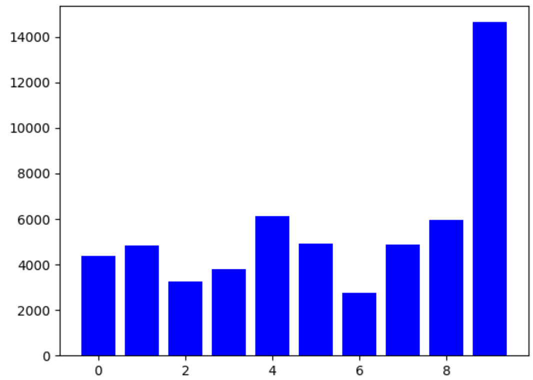

Figure 7: **Class Tree - Sample LBP Histogram**

Confusion Matrix

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

Accuracy calculation:

$$Accuracy = \frac{Truepredictions}{Totalpredictions} * 100 \tag{3}$$

We have True predictions = 17 from the confusion matrix Total Predictions have to be 25 since there are 5 images each in the five classes. Therefore, the accuracy is:

$$Accuracy = \frac{17}{25} * 100 = 68\% \tag{4}$$

Overall, we see that this is a decent image classifier. We used the parameters:

1. R value 1
2. P value 8
3. k value 5 for kNN Classifier

Source Code

Part of the kNN classifier was inspired from the implementation found in: LINK

1

```
"""
Computer Vision - Purdue University - Homework 7

Author : Arjun Kramadhati Gopi, MS-Computer & Information
    Technology, Purdue University.
Date: Oct 19, 2020


[TO RUN CODE]: python3 classifier.py
Output:
    [labels]: Predictions for the input images
"""

import cv2 as cv
from matplotlib import pyplot as plt
import numpy as np
import math
import BitVector
import pickle
import os
from collections import Counter


class Imageclassifier:

    def __init__(self, training_directory, testing_directory,
        parameterR, parameterP, kvalue, Train=False):
        """
        Initialise the image classifier object used to either
            train or test the classification of images
        :param training_directory: Directory of the training set
        :param testing_directory: Directory of the testing set
        :param parameterR: Radius for the circular boundary
        :param parameterP: Number of points on the circular
            boundary
        :param kvalue:k nearest neighbors needed for match
        :param Train: Training yes or no
        """
        self.classdict = dict()
        self.imagedict = dict()
        self.histogramdict = dict()
        self.database = []
        self.cmatrix = np.zeros((5, 5), dtype='int')
        if not Train:
```

```
            training_directory = testing_directory
        self.classcount = len(os.listdir(training_directory))
        for element in os.listdir(training_directory):
            self.classdict[element] = len(os.listdir(
               training_directory+'/'+element))
            templist = []
            for image in sorted(os.listdir(training_directory
               +'/'+element)):
                print(os.listdir(training_directory+'/'+element))
                print(image)
                origimage = cv.imread(training_directory+'/'+
                   element+'/'+image)
                imageread = np.zeros((origimage.shape[0],
                   origimage.shape[1], origimage.shape[2]), dtype
                   ='uint8')
                image_gray = np.zeros((origimage.shape[0],
                   origimage.shape[1]), dtype='uint8')
                imageread = cv.imread(training_directory+'/'+
                   element+'/'+image)
                image_gray = cv.cvtColor(imageread, cv.
                   COLOR_BGR2GRAY)
                templist.append(image_gray)
            self.imagedict[element]=templist
        self.parameterR = parameterR
        self.parameterP = parameterP
        self.kneighbors = kvalue
        print(self.imagedict['beach'][0].shape)

    def get_pixel_value(self, queuetuple, delu, delv, centerX,
       centerY):
        """
        This function implements the bilinear interpolation
           method used to get the pixel value
        or the grey value at the point p
        :param queuetuple: Location of the image
        :param delu: change in value in x direction
        :param delv: change in value in y direction
        :param centerX: the point at the center of the circle
           under consideration
        :param centerY: the point at the center of the circle
           under consideration
        :return: greylevel at the point p
        """
        image = self.imagedict[queuetuple[0]][queuetuple[1]]
        if (delu < 0.01) and (delv < 0.01):
            interpolated_greylevel = float(image[centerX][centerY
               ])
        elif (delv < 0.01):
            interpolated_greylevel = (1 - delu) * image[centerX][
               centerY] + delu * image[centerX + 1][centerY]
        elif (delu < 0.01):
            interpolated_greylevel = (1 - delv) * image[centerX][
               centerY] + delv * image[centerX][centerY + 1]
```

```
        else:
            interpolated_greylevel = (1 - delu) * (1 - delv) * image[centerX][centerY] + (1 - delu) * delv * image[centerX][centerY + 1] + delu * delv * \
                                     image[centerX + 1][centerY + 1] + delu * (1 - delv) * image[centerX + 1][centerY]
        return interpolated_greylevel

    def build_histogram(self, histogram, runs):
        if len(runs) > 2:
            histogram[self.parameterP + 1] += 1
        elif len(runs) == 1 and runs[0][0] == '1':
            histogram[self.parameterP] += 1
        elif len(runs) == 1 and runs[0][0] == '0':
            histogram[0] += 1
        else:
            histogram[len(runs[1])] += 1
        return histogram

    def generate_texture_feature(self, queuetuple, Train = False):
        """
        This function implements the building of the Local Binary Pattern histogram for the give image
        :param queuetuple: Location of the image in the dictionary
        :return: None. Stores the histogram in a dictionary
        """
        histogram = {bins: 0 for bins in range(self.parameterP + 2)}
        greyimage = self.imagedict[queuetuple[0]][queuetuple[1]]
        for row in range(self.parameterR, greyimage.shape[0]-self.parameterR-1):
            # print(str(row) + " out of " + str(greyimage.shape[0] - self.parameterR - 1))
            for column in range(self.parameterR, greyimage.shape[1]- self.parameterR-1):
                binarypatternforpoint = []
                for pointnumber in range(self.parameterP):
                    delu = self.parameterR * math.cos(2 * math.pi * pointnumber / self.parameterP)
                    delv = self.parameterR * math.sin(2 * math.pi * pointnumber / self.parameterP)
                    if abs(delu) < 0.001: delu = 0.0
                    if abs(delv) < 0.001: delv = 0.0
                    greylevel = self.get_pixel_value(queuetuple, delu, delv, int(row+delu), int(column+delv))
                    if greylevel >= greyimage[row][column]:
                        binarypatternforpoint.append(1)
                    else:
                        binarypatternforpoint.append(0)
```

```python
                bitvector = BitVector.BitVector(bitlist=binarypatternforpoint)
                intvals_for_circular_shifts = [int(bitvector << 1) for _ in range(self.parameterP)]
                minimum_bit_vector = BitVector.BitVector(intVal=min(intvals_for_circular_shifts), size=self.parameterP)
                runs = minimum_bit_vector.runs()
                histogram = self.build_histogram(histogram, runs)
        if Train:
            self.histogramdict[queuetuple] = histogram
            plt.bar(list(histogram.keys()), histogram.values(), color='b')
            path = 'histograms/' + str(queuetuple[0]) + '/'
            plt.savefig(path + 'Class_{}'.format(queuetuple[0]) + '_ImageNum_{}'.format(int(queuetuple[1])) + '.png')
        if not Train:
            return histogram

    def save_histograms_of_all(self, filename):
        """
        Saves the Local Binary Pattern histograms of every image
        :param filename: File name for the data base
        :return: None
        """
        file = open(filename,'wb')
        pickle.dump(self.histogramdict, file)
        file.close()

    def load_data(self, filename):
        """
        Loads the database from the saved .obj file. The database contains
        the LBP histograms of very image in training set
        :param filename: File name of the database being retrieved.
        :return: Load and store the database in a dictionary
        """
        blist =[]
        bblist =[]
        clist =[]
        mlist =[]
        tlist =[]
        beachlist =np.zeros((20,10))
        buildinglist =np.zeros((20,10))
        carlist =np.zeros((20,10))
        mountainlist =np.zeros((20,10))
        treelist=np.zeros((20,10))
        print(self.classdict['tree'])
        file = open(filename, 'rb')
        database = pickle.load(file)
        file.close()
```

```
        for element,index in database:
            if element[0] =='beach':
                blist.append(database.get(element))
            if element[0] =='building':
                bblist.append(database.get(element))
            if element[0] =='mountain':
                mlist.append(database.get(element))
            if element[0] =='car':
                clist.append(database.get(element))
            if element[0] =='tree':
                tlist.append(database.get(element))

        for index in range(len(blist)):
            beachlist[index, :] = np.array(list(blist[index].values()))
        for index in range(len(bblist)):
            buildinglist[index, :] = np.array(list(bblist[index].values()))
        for index in range(len(clist)):
            carlist[index, :] = np.array(list(clist[index].values()))
        for index in range(len(mlist)):
            mountainlist[index, :] = np.array(list(mlist[index].values()))
        for index in range(len(tlist)):
            treelist[index, :] = np.array(list(tlist[index].values()))

        histogram_all = np.zeros((100, 11))
        for i in range(5):
            index1 = 20 * i
            index2 = index1 + 20
            histogram_all[index1:index2, 0] = i
        histogram_all[:, 1:] = np.concatenate((beachlist, buildinglist, carlist, mountainlist, treelist), axis=0)
        self.database = histogram_all
        print('loaded data successfully')

    def knn_classify(self, list_histograms_class, numberoftestingimages = 5, numberoftrainingimages =20):
        """
        This function implements the knnparameter-Nearest Neighbor algorithm. We use the Eucledian distance to
        calculate the nearest matches.
        :param list_histograms_class: List of all the LBP histograms of a particular class
        :param numberoftestingimages: Number of images in the testing set
        :param numberoftrainingimages: Number of images in the training set
        :param nClass: Number of classes to predict
        :return: returns the index of the label of the classes
```

```
        """
        knnparameter = self.kneighbors
        training_histogram_all = self.database
        result_hist = np.zeros((numberoftestingimages, 10))
        condition1 = numberoftrainingimages * 1
        condition2 = numberoftrainingimages * 2
        condition3 = numberoftrainingimages * 3
        condition4 = numberoftrainingimages * 4
        condition5 = numberoftrainingimages * 5
        for index_class in range(len(list_histograms_class)):
            label_list = np.zeros((numberoftestingimages, knnparameter), dtype='int')
            labelindex = np.zeros(numberoftestingimages, dtype='int')
            result_hist[index_class, :] = np.array(list(list_histograms_class[index_class].values()))
        eucledian_distance = np.zeros((numberoftestingimages, training_histogram_all.shape[0]))
        for imageindex in range(numberoftestingimages):
            for imagetrain in range(training_histogram_all.shape[0]):
                eucledian_distance[imageindex, imagetrain] = np.linalg.norm(result_hist[imageindex, :] - training_histogram_all[imagetrain, 1:])
            sorted_distance = np.argsort(eucledian_distance[imageindex, :])
            for k_idx in range(knnparameter):
                if (sorted_distance[k_idx] <= (condition1)):
                    label_list[imageindex, k_idx] = 0
                elif (sorted_distance[k_idx] <= (condition2)):
                    label_list[imageindex, k_idx] = 1
                elif (sorted_distance[k_idx] <= (condition3)):
                    label_list[imageindex, k_idx] = 2
                elif (sorted_distance[k_idx] <= (condition4)):
                    label_list[imageindex, k_idx] = 3
                elif (sorted_distance[k_idx] <= (condition5)):
                    label_list[imageindex, k_idx] = 4
            labelindex[imageindex], freq = Counter(list(label_list[imageindex, :])).most_common(1)[0]
        return labelindex

    def predict_and_analyse(self,blist,bblist,mlist,clist,tlist):
        """
        This function takes the histograms of the testing set and uses them to
        predict the class labels for each of the images in the testing set. The results are
        collated in a confusion matrix
        :param blist: List of histograms for beach class
        :param bblist: List of histograms for building class
        :param mlist: List of histograms for mountain class
        :param clist: List of histograms for the car class
        :param tlist: List of histograms for the tree class
```

```
        :return: Prints the final confusion matrix.
        """

        label_index = self.knn_classify(blist)
        label_unique, label_unique_count = np.unique(label_index, return_counts=True)
        self.cmatrix[0, label_unique] = label_unique_count
        label_index = self.knn_classify(bblist)
        label_unique, label_unique_count = np.unique(label_index, return_counts=True)
        self.cmatrix[1, label_unique] = label_unique_count
        label_index = self.knn_classify(mlist)
        label_unique, label_unique_count = np.unique(label_index, return_counts=True)
        self.cmatrix[2, label_unique] = label_unique_count
        label_index = self.knn_classify(clist)
        label_unique, label_unique_count = np.unique(label_index, return_counts=True)
        self.cmatrix[3, label_unique] = label_unique_count
        label_index = self.knn_classify(tlist)
        label_unique, label_unique_count = np.unique(label_index, return_counts=True)
        self.cmatrix[4, label_unique] = label_unique_count
        print('Prediction complete')
        print('Printing confusion matrix...')
        print(self.cmatrix)


if __name__ == "__main__":
    """
    Code begins here
    """
    tester = Imageclassifier("imagesDatabaseHW7/training", "imagesDatabaseHW7/testing", 1, 8, 5)
    Train = False
    if Train:
        for element in os.listdir("imagesDatabaseHW7/training"):
            for index in range(len(os.listdir("imagesDatabaseHW7/training" + '/' + element))):
                print('training image class: '+element+' __ ' + str(index))
                tester.generate_texture_feature((element, index))
        print('Training complete. Saving histogram dictionary...')
        tester.save_histograms_of_all('histograms.obj')
        print('Saving complete')
    tester.load_data('histograms.obj')
    btestlist =[]
    bbtestlist =[]
    mtestlist = []
    ctestlist = []
    ttestlist = []
    testdict = dict()
```

```python
    for element in os.listdir("imagesDatabaseHW7/testing"):
        for index in range(len(os.listdir("imagesDatabaseHW7/testing" + '/' + element))):
            print('testing image class: ' + element + ' __ ' + str(index))
            hist = tester.generate_texture_feature((element,index))
            if element == 'beach':
                btestlist.append(hist)
            if element == 'building':
                bbtestlist.append(hist)
            if element == 'mountain':
                mtestlist.append(hist)
            if element == 'car':
                ctestlist.append(hist)
            if element == 'tree':
                ttestlist.append(hist)
    tester.predict_and_analyse(btestlist,bbtestlist,mtestlist,ctestlist,ttestlist)
```