hw3_p1_soln

November 29, 2022

0.1 Problem 1: Image Classification Using Bag-of-words [20 pts]

We will now build a classifier to determine whether an input image contains a face. A training dataset has been provided consisting of images with and without faces. We will create a bag-of-words representation for images and use K-nearest neighbor classification.

Note: For this problem, it is not important to get high accuracies, but just be able to see how an example of an entire pipeline for image classification works.

0.2 Review of bag-of-words for image classification

We will solve image classification, with the relatively simple but effective bag-of-words approach. We treat an image as a set of regions and ignore the spatial relationships between different regions. The image classification problem can then be solved by counting the frequencies of different regions appearing in an image. Image classification using bag-of-words includes the following steps:

- **Region Selection**: Select some regions in the images so that we can extract features from those regions in the next step. The regions can be selected by feature detection methods like edge or corner detection, or you can just uniformly sample the image.
- **Feature Extraction**: We extract features from the selected regions. One commonly used feature is SIFT. We extract features from every image in the training set. These features are collected to compute the visual vocabulary for image representation.
- **Building visual vocabulary**: Once we have the features extracted from training images, we build a visual vocabulary by grouping them together to form a few clusters. We will use the k-means algorithm to group the features. The reason why we need this step is to reduce the redundancy in feature space and have a more concise feature representation of images.
- **Learning and recognition**: Given the visual vocabulary, each training image is represented by a histogram, where the features in the image populate bins that correspond to the visual word closest to them. Thereafter, we will use k-nearest neighbors to perform classification for a test image, also represented by a histogram using the same vocabulary.

0.3 Simple face classifier

We will make a classifier to tell whether there is a face in a given image. The dataset contains 200 images with faces and 200 images without faces. We will pick 100 images from each group for training and the other 100 images for testing. The skeleton code is given.

[21]:
```
import glob
import random
import numpy as np
from matplotlib import pyplot as plt
import cv2

# Parameters
posData = 'images/face/'
negData = 'images/nonface/'
posTrainRatio = 0.5
negTrainRatio = 0.5
imSize = 133, 200 # resize each image to this size
nIntPts = 200 # maximum number of interest points to be extracted from an image
wGrid = 5 # width of the small grids for uniform sampling
patchSize = 11 # an odd number indicate patch size for image patch feature
nCluster = 50 # number of cluster in k-means algorithm

def listImage(dataRoot):
    fileList = glob.glob(str(dataRoot + '*.jpg'))
    imNum = len(fileList)
    fileList = [fileList[i].replace('\\', '/') for i in range(imNum)]
    return fileList

def loadImage(imgName, imSize):
    # load image, resize and make RGB to grayscale
    img = cv2.imread(imgName)
    img = cv2.resize(img, (imSize[1], imSize[0]))
    img = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
    return img

def createSplit(imgList, ratio):
    random.shuffle(imgList)
    trainList = imgList[:round(len(imgList)*ratio)]
    testList = imgList[round(len(imgList)*ratio):]
    return trainList, testList

# Data preprocessing: Creaete Training / Testing Split
posList = listImage(posData)
negList = listImage(negData)
trainPosList, testPosList = createSplit(posList, posTrainRatio)
trainNegList, testNegList = createSplit(negList, negTrainRatio)
trainList = trainPosList + trainNegList
trainLabel = np.concatenate((np.ones(len(trainPosList)), np.
 ↪zeros(len(trainNegList))))
testList = testPosList + testNegList
testLabel = np.concatenate((np.ones(len(testPosList)), np.
 ↪zeros(len(testNegList))))
```

```
testPosLabel = np.ones(len(testPosList))
testNegLabel = np.zeros(len(testNegList))
```

0.4 1.1 Extract interest points from images [2 pts]

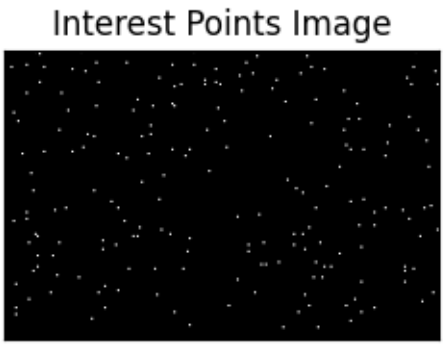
You will now try two methods for this:

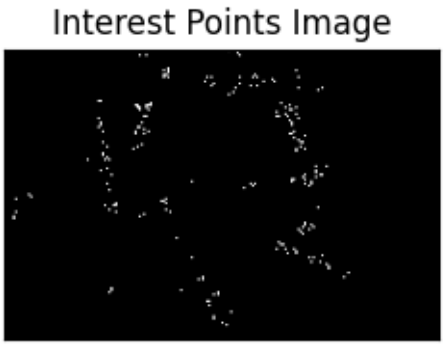
- **(a) Uniformly sample the images.** You can divide the image into regular grids and choose a point in each grid and then uniformly choose `nPts` number of interest points.
- **(b) Sample on corners.** First use the Harris Corner detector to detect corners in the image and then uniformly choose `nPts` number of interest points. (This has already been implemented for you as an example.)

[22]:
```
def uniformSampling(imSize, nPts, wGrid):
    ''' Uniformly sample the images.
    Args:
        imSize: size of images (height, width)
        nPts: maximum number of interest points to be extracted
        wGrid: width of the small grids
    Return:
        pts: a list of interest points (Yi, Xi)
    '''
    Y = np.linspace(0, imSize[0]-imSize[0]%wGrid, num=imSize[0]//wGrid,␣
↪endpoint = False, dtype=int)
    X = np.linspace(0, imSize[1]-imSize[1]%wGrid, num=imSize[1]//wGrid,␣
↪endpoint = False, dtype=int)
    YY, XX = np.meshgrid(Y, X, indexing='ij')
    perm = np.random.choice(wGrid*wGrid, (len(Y), len(X)))
    permYY = perm // wGrid
    permXX = perm % wGrid
    YY = YY + permYY
    XX = XX + permXX
    ptsY = np.reshape(YY, -1)
    ptsX = np.reshape(XX, -1)
    num = min(nPts, len(ptsY))
    choice = np.random.choice(len(ptsY), num, replace=False)
    ptsY = ptsY[choice]
    ptsX = ptsX[choice]
    pts = [(ptsY[i], ptsX[i]) for i in range(num)]
    return pts

def cornerSampling(img, nPts):
    '''
    Args:
        img: image which you want to extract interest points
        nPts: maximum number of interest points to be extracted
    Return:
```

```
        pts: a list of interest points (Yi, Xi)
    '''
    dst = cv2.cornerHarris(img,2,3,0.04)
    dst = cv2.dilate(dst,None)
    cor = np.zeros(img.shape)
    cor[dst>0.01*dst.max()]=[1]
    ptsY, ptsX = np.where(cor==1)
    num = min(nPts, len(ptsY))
    choice = np.random.choice(len(ptsY), num, replace=False)
    ptsY = ptsY[choice]
    ptsX = ptsX[choice]
    pts = [(ptsY[i], ptsX[i]) for i in range(num)]
    return pts

def plotInterestPoints(img, pts, idx=0):
    ptsMap = np.zeros(img.shape)
    ptsY = [Y for Y, X in pts]
    ptsX = [X for Y, X in pts]
    ptsMap[ptsY, ptsX] = 1
    plt.figure(idx)
    plt.subplot(121),plt.imshow(img,cmap = 'gray')
    plt.title('Original Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(122),plt.imshow(ptsMap,cmap = 'gray')
    plt.title('Interest Points Image'), plt.xticks([]), plt.yticks([])

# Here is the code for you to test your implementation
sampleImg = loadImage(trainList[0], imSize)
print('Interest Points extracted by uniform (above) and edge (below) method:')
ptsU = uniformSampling(imSize, nIntPts, wGrid)
plotInterestPoints(sampleImg, ptsU, idx=0)
ptsE = cornerSampling(sampleImg, nIntPts)
plotInterestPoints(sampleImg, ptsE, idx=1)
```

```
Interest Points extracted by uniform (above) and edge (below) method:
```





0.5 1.2. Extract features [2 pts]

You are required to try two kinds of features: * **(a) SIFT feature.** Here we use the SIFT implementation in *OpenCV* package (this has already been implemented for you as an example). * **(b) Image Patch feature.** Extract a small image patch around each feature point. You can decide the size of each patch and how many pixels it should cover on you own.

[23]:
```
def extractSIFTfeature(img, pts):
    '''
    Args:
        img: input image
        pts: detected interest points in the previous step
    Return:
        features: a list of SIFT descriptor features for each interest point
    '''
    #sift = cv2.xfeatures2d.SIFT_create()
    sift = cv2.SIFT_create()
    kp = [cv2.KeyPoint(float(ptsX), float(ptsY), 1) for ptsY, ptsX in pts]
    _, des = sift.compute(img, kp)
    features = [des[i] for i in range(des.shape[0])]
    return features

def extractImagePatchfeature(img, pts, patchSize):
    '''
    Args:
        img: input image
        pts: detected interest points in the previous step
        patchSize: an odd number indicate patch size
    Return:
        features: a list of image patch features (1-d) for each interest point
    '''
    d = (patchSize - 1)//2
```

```
    features = []
    for yi, xi in pts:
        try:
            patch = img[yi-d:yi+d+1,xi-d:xi+d+1]
            feat = np.resize(patch, (patch.size, ))
            if feat.shape[0] != patchSize*patchSize:
                continue
        except ValueError:
            pass
        features.append(feat)
    return features

# Here is the code for you to test your implementation
featSIFT = extractSIFTfeature(sampleImg, ptsE)
print('length of SIFT feature list', len(featSIFT))
print('dimension of feature', featSIFT[0].shape)
featPatch = extractImagePatchfeature(sampleImg, ptsE, patchSize)
print('length of Image Patch feature list', len(featPatch))
print('dimension of feature', featPatch[0].shape)
```

```
length of SIFT feature list 200
dimension of feature (128,)
length of Image Patch feature list 192
dimension of feature (121,)
```

0.6 1.3. Build visual vocabulary [4 pts]

Use k-means clustering to form a visual vocabulary. You can use the python k-means package. You can decide the number of clusters yourself. The default number of cluster centers in k-means is 50.

[24]:
```
from sklearn.cluster import KMeans

def getImgFeat(img, imSize, nIntPts, wGrid, patchSize, ptType, featType):
    ''' Output a list of detected features for an image
    Args:
        img: image which you want to extract interest points
        imSize: size of images (height, width)
        nIntPts: maximum number of interest points to be extracted
        wGrid: width of the small grids
        ptType: 'uniform' or 'corner' indicates the interest pts sampling method
        featType: 'sift' or 'patch' indicates the feature extraction method
    Return:
        extractFeatList: a list of image patch features (1-d) for each interest␣
 ↪point
    '''
    if ptType == 'uniform':
        intPts = uniformSampling(imSize, nIntPts, wGrid)
    elif ptType == 'corner':
```

```
        intPts = cornerSampling(img, nIntPts)
    else:
        assert False, 'ptType must be either uniform or corner'

    if featType == 'sift':
        extractFeatList = extractSIFTfeature(img, intPts)
    elif featType == 'patch':
        extractFeatList = extractImagePatchfeature(img, intPts, patchSize)
    else:
        assert False, 'featType must be either sift or patch'

    return extractFeatList

def collectFeat(trainList, imSize, nIntPts, wGrid, patchSize, ptType, featType):
    ''' Collect extracted features for each image among training data
    Args:
        trainList: list of images filepath
        imSize: size of images (height, width)
        nIntPts: maximum number of interest points to be extracted
        wGrid: width of the small grids
        ptType: 'uniform' or 'corner' indicates the interest pts sampling method
        featType: 'sift' or 'patch' indicates the feature extraction method
    Return:
        feats: (# of features, dim of feature) array
    '''
    feats = []
    for imgName in trainList:
        img = loadImage(imgName, imSize)
        feat = getImgFeat(img, imSize, nIntPts, wGrid, patchSize, ptType,␣
 ↪featType)
        feats = feats + feat
    feats = np.stack(feats, axis=0)
    return feats

def formVisualVocab(feats, nCluster):
    ''' Use k-means algorithm to find k cluster centers, output k-means model
    Args:
        feats: list of features collected from training data
        nCluster: number of cluster in k-means algorithm
    Return:
        model: k-means model for following steps
    '''
    feats = np.stack(feats, axis = 0)
    model = KMeans(n_clusters=nCluster, random_state=0).fit(feats)
    return model

# Here is the code for you to test your implementation
```

```
feats = collectFeat(trainList, imSize, nIntPts, wGrid, patchSize,␣
 ↪ptType='uniform', featType='sift')
print(feats.shape) # should be (total number of extracted features, dim of␣
 ↪feature)
model = formVisualVocab(feats, nCluster)
centers = model.cluster_centers_
print(centers.shape) # should be (nCluster, dim of feature)
```

```
(20000, 128)
(50, 128)
```

0.7 1.4. Compute histogram representation [4 pts]

Compute the histogram representation of each image, with bins defined over the visual words in the vocabulary. These histograms are the bag-of-words representations of images that will be used for image classification.

[25]:
```
def getHistogram(img, model, nCluster, imSize, nIntPts, wGrid, patchSize,␣
 ↪ptType, featType):
    '''
    Calculate histogram representation for single image
    Args:
        img: input image
        model: k-means model from previous step
        nCluster: number of cluster in k-means algorithm
        imSize: size of images (height, width)
        nIntPts: maximum number of interest points to be extracted
        wGrid: width of the small grids
        ptType: 'uniform' or 'corner' indicates the interest pts sampling method
        featType: 'sift' or 'patch' indicates the feature extraction method
    Output:
        hist: histogram representation (1-d) for input image
    '''
    feat = getImgFeat(img, imSize, nIntPts, wGrid, patchSize, ptType, featType)
    feat = np.stack(feat, axis=0)
    cl = model.predict(feat)
    hist = np.zeros(nCluster)
    for i in range(nCluster):
        hist[cl[i]] += 1
    return hist

def computeHistograms(trainList, model, nCluster, imSize, nIntPts, wGrid,␣
 ↪patchSize, ptType, featType):
    ''' Compute histogram representation for whole training dataset
    Input:
    - trainList: list of images filepath
    - model: k-means model from formVisualVocab
```

```
    - others: parameters
    Output:
    - hists: (# of images, nCluster) array - histogram representations among␣
↪training dataset
    '''
    hists = []
    for imgName in trainList:
        img = loadImage(imgName, imSize)
        hist = getHistogram(img, model, nCluster, imSize, nIntPts, wGrid,␣
↪patchSize, ptType, featType)
        hists.append(hist)
    hists = np.stack(hists, axis=0)
    return hists

# Here is the code for you to test your implementation
hist = getHistogram(sampleImg, model, nCluster, imSize, nIntPts, wGrid,␣
↪patchSize, ptType='uniform', featType='sift')
print(hist.shape) # should be (nCluster)
hists = computeHistograms(trainList, model, nCluster, imSize, nIntPts, wGrid,␣
↪patchSize, ptType='uniform', featType='sift')
print(hists.shape) # should be (# of training images, nCluster)
```

```
(50,)
(100, 50)
```

0.8 1.5. K nearest neighbor classifier [4 pts]

After building the visual vocabulary, we now do image classification using the nearest neighbors method. Given a new image, first represent it using the visual vocabulary and then find the closest representation in the training set. The test image is assigned the same category as its nearest neighbor in the training set. Next, to make the algorithm more robust, find the first K-nearest neighbors (for K = 3 and 5).

[26]:
```
from sklearn.neighbors import KNeighborsClassifier

def KNNclassifier(trainX, trainY, n_neighbors):
    ''' Return a KNN model by fitting training data (trainX, trainY)
    Args:
        trainX: a (# of images, nCluster) array of BoW features for training␣
↪data
        trainY: a (# of images) array of class label for training data
        n_neighbors: # of neighbors used in KNN classifier
    Return:
        model: KNN classifier model
    '''
    model = KNeighborsClassifier(n_neighbors)
    model.fit(trainX, trainY)
```

```
    return model

def getAccuracy(testX, testY, model):
    ''' Output the testing accuracy for KNN classifier model
    Args:
        testX: a (# of images, nCluster) array of BoW features for testing data
        testY: a (# of images) array of class label for testing data
        model: KNN classifier model
    Return:
        accu: accuracy of classification prediction on testing data
    '''
    accu = model.score(testX, testY)
    return accu

# Here is the code for you to test your implementation
X = [[0], [1], [2], [3]]
y = [0, 0, 1, 1]
model = KNNclassifier(X, y, n_neighbors=3)
print(model.predict([[1.1]])) # Should be [0]
Xp = [[0.5], [2.5]]
Yp = [0, 0]
acc = getAccuracy(Xp, Yp, model)
print(acc) # Should be 0.5
```

```
[0]
0.5
```

0.9 1.6. Calculate testing accuracy [4 pts]

Use `50 clusters` in k-means and keep other hyperparameters as default, try `K=3` and `K=5` for KNN classifier respectively, report the accuracy in the following two 2D tables. You should report the accuracy of each method on both positive and negative testing samples. Some of the methods may have poor accuracy. But that is fine, don't worry too much about accuracy. You will get full credit as long as you can correctly implement and reason about the various methods.

`K=3`

	Uniform	Uniform	Corners	Corners
	Positive	Negative	Positive	Negative
SIFT Feature	0.88	0.3	0.64	0.46
Image Patch	0.82	0.58	0.66	0.6

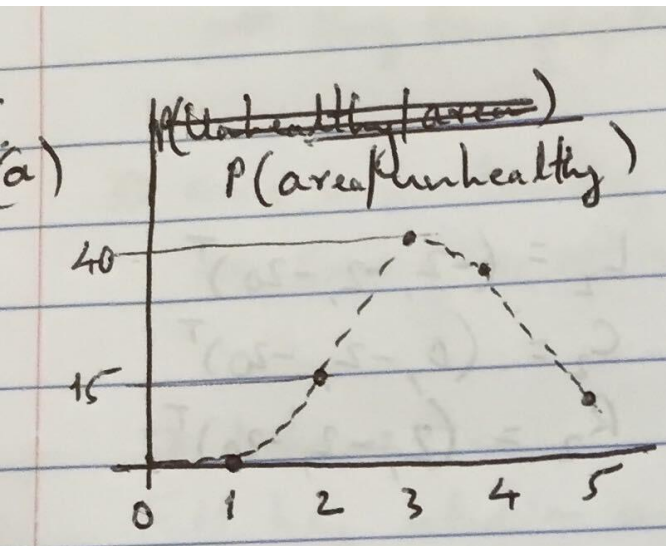
`K=5`

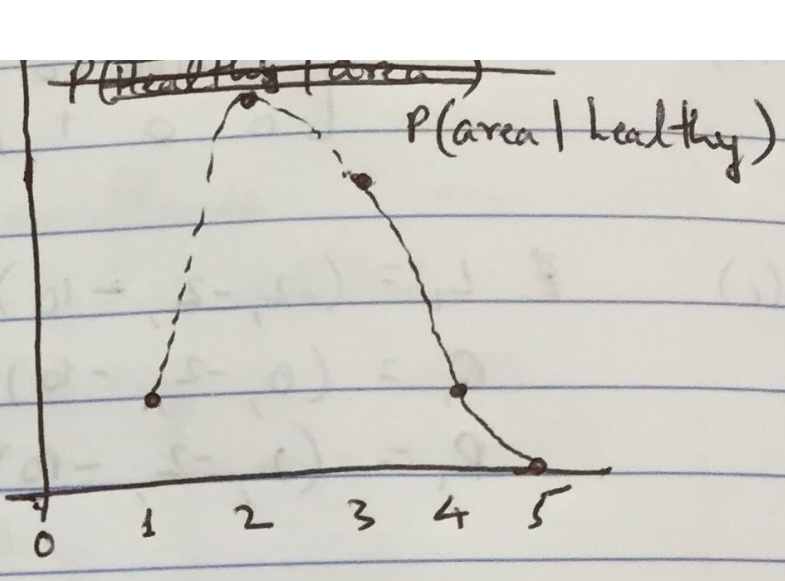
	Uniform	Uniform	Corners	Corners
	Positive	Negative	Positive	Negative
SIFT Feature	0.86	0.3	0.56	0.54

	Uniform	Uniform	Corners	Corners
Image Patch	0.74	0.5	0.7	0.72

[29]:
```
ptType = ['uniform', 'corner']
featType = ['sift', 'patch']
n_neighbors=5
for pt in ptType:
    for ft in featType:
        print('Doing with sampling type:', pt, ' and feature extraction type:␣
↪', ft)
        feats = collectFeat(trainList, imSize, nIntPts, patchSize, wGrid,␣
↪ptType=pt, featType=ft)
        model = formVisualVocab(feats, nCluster)
        trainX = computeHistograms(trainList, model, nCluster, imSize, nIntPts,␣
↪patchSize, wGrid, ptType=pt, featType=ft)
        clf = KNNclassifier(trainX, trainLabel, n_neighbors)
        testPosX = computeHistograms(testPosList, model, nCluster, imSize,␣
↪nIntPts, patchSize, wGrid, ptType=pt, featType=ft)
        testNegX = computeHistograms(testNegList, model, nCluster, imSize,␣
↪nIntPts, patchSize, wGrid, ptType=pt, featType=ft)
        posAccu = getAccuracy(testPosX, testPosLabel, clf)
        negAccu = getAccuracy(testNegX, testNegLabel, clf)
        print('Pos Accuracy :', posAccu, ' ; Neg Accuracy: ', negAccu)
```

```
Doing with sampling type: uniform  and feature extraction type:  sift
Pos Accuracy : 0.86  ; Neg Accuracy:  0.3
Doing with sampling type: uniform  and feature extraction type:  patch
Pos Accuracy : 0.74  ; Neg Accuracy:  0.5
Doing with sampling type: corner  and feature extraction type:  sift
Pos Accuracy : 0.56  ; Neg Accuracy:  0.54
Doing with sampling type: corner  and feature extraction type:  patch
Pos Accuracy : 0.7  ; Neg Accuracy:  0.72
```



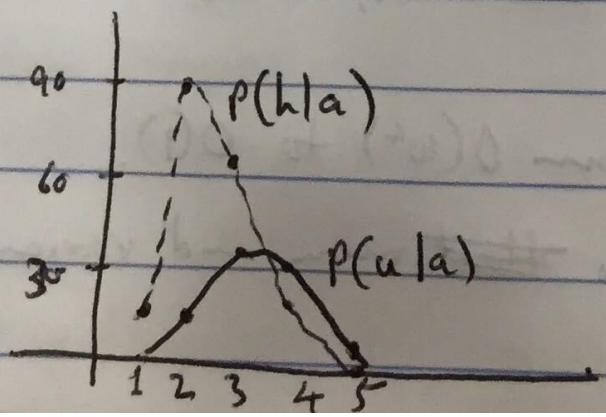


$P(u) = \frac{1}{3}$ $\quad P(h) = \frac{2}{3}$

$P(u|a) \sim P(a|u)\,P(u)$ $\quad P(h|a) = P(a|h)\,P(h)$

a	$P(a\|u)$	$P(u\|a)$
1	$\frac{0}{100}$	$\frac{0}{300}$
2	$\frac{15}{100}$	$\frac{15}{300}$
3	$\frac{40}{100}$	$\frac{40}{300}$
4	$\frac{35}{100}$	$\frac{35}{300}$
5	$\frac{10}{100}$	$\frac{10}{300}$

a	$P(a\|h)$	$P(h\|a)$
1	$\frac{10}{100}$	$\frac{20}{300}$
2	$\frac{45}{100}$	$\frac{90}{300}$
3	$\frac{35}{100}$	$\frac{70}{300}$
4	$\frac{10}{100}$	$\frac{20}{300}$
5	$\frac{0}{100}$	$\frac{0}{300}$



(d) Check whether $P(h|a) > P(u|a)$ for healthy.

5 → u 2 → h 3 → h

(e) Check whether $P(a|h) > P(a|u)$ for healthy.

5 → u 2 → h 3 → u