ECE 661 Computer Vision

Homework 7

Yi-Hong Liao | liao163@purdue.edu | PUID 0031850073

1. Logic

Construct 3 types of image texture descriptors and use the descriptors to train image classification frameworks.

2. Step

- Texture Descriptors
  i. Local Binary Pattern (LBP)
  Construct the binary pattern from each pixel by comparing the neighboring pixels with offsets
  $$(\Delta u, \Delta v) = \left(R\cos\left(\frac{2\pi p}{P}\right), R\sin\left(\frac{2\pi p}{P}\right)\right), \mathrm{p} = 0, 1, 2, \ldots, \mathrm{P}$$
  with the center pixel. Encode the min integer value form of the binary pattern with the runs of 0s and 1s and use it as the texture descriptor. I modified the implementation from Prof. Kak [1].
  ii. Gram Matrix
  The output of the VGG19 network layer relu5_1 is a N*W*H tensor. Where N is the number of channels and M = W*H is the number of spatial locations. Reshape the tensor to a vectorized feature map F of shape (N, M). Then, the Gram Matrix can be found by
  $$G = F \cdot F^T.$$
  We choose 1024 elements from the upper triangle of G as the Gram descriptor vector.
  iii. Adaptive Instance Normalization (AdaIN)
  From the feature vector F of the Gram matrix, calculate the mean and variance of each row and concatenate them to form the AdaIN texture descriptor.
- Image Classification Pipeline
  For the LBP descriptors, I use them to train a Polynomial SVM. For the Gram Matrix and AdaIN descriptors, I use them to train a Linear SVM.

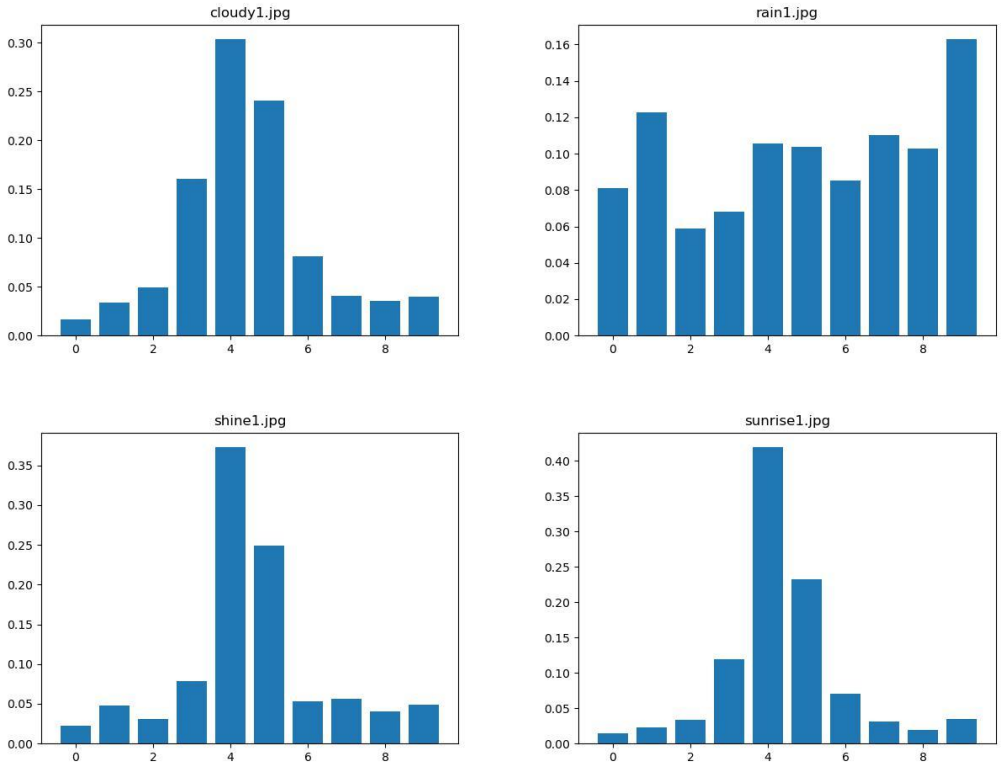
[1] Texture and Color Tutorial. URL https://engineering.purdue.edu/kak/Tutorials/TextureAndColor. pdf.

3. Result

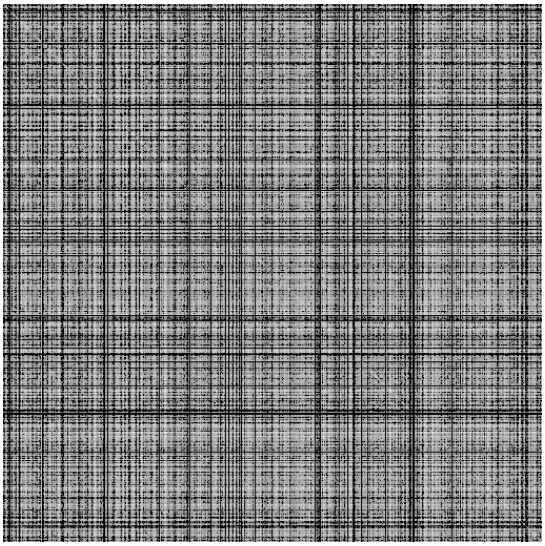
- **Theory question**

1. In GLCM method, the joint probability distribution $P(x_1, x_2)$ is estimated. Where x_1 is a random selected pixel and x_2 is a pixel with a distance d from x_1. The shape of the GLCM matrix will be $L \times L$ where L is the texture intensity levels. The texture is characterized by the joint probability distribution. LBP is also a statistical method to characterize image texture. The texture is characterized by comparing pixel's neighbors to the pixel itself. LBP is rotation invariant because the generated local binary pattern is rotated to a minimum integer value form. On the other hand, Gabor filter are structural texture characterizing method. Gabor filters are a set of filters that extract image intensity frequencies and directions. Several filters are applied to the image and the summation will be used to describe the texture.
2.
   (a) **Wrong.** RGB and HIS are not linearly related.
   (b) **Right.** L* is perceptual lightness and a* and b* for red, green, blue and yellow.
   (c) **Right.** Different spectral compositions of the illumination results in different measuring results.

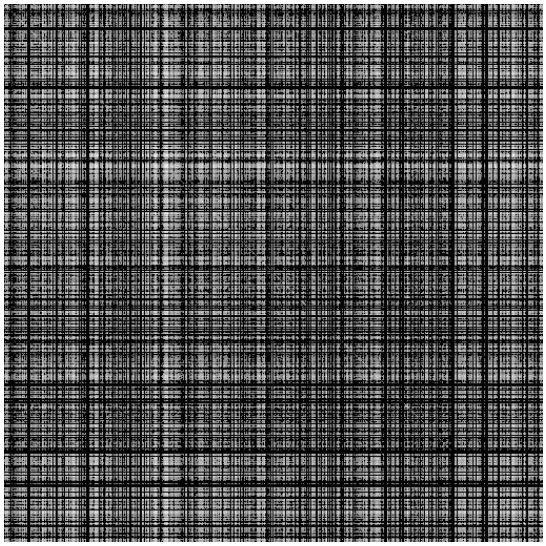
- **LBP Histogram**



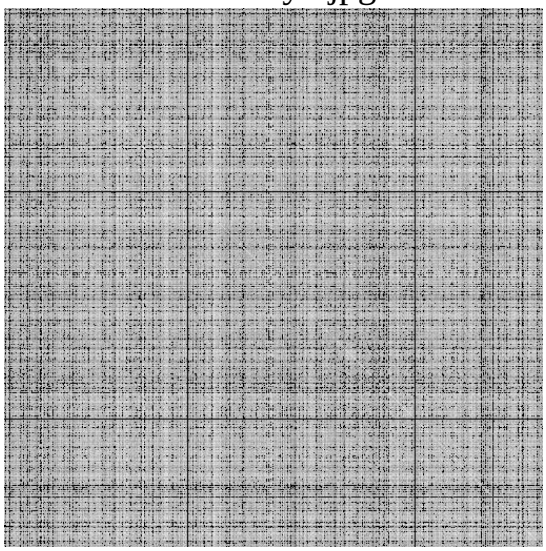
- **Gram Matrix (Take Log value for better visualization result)**



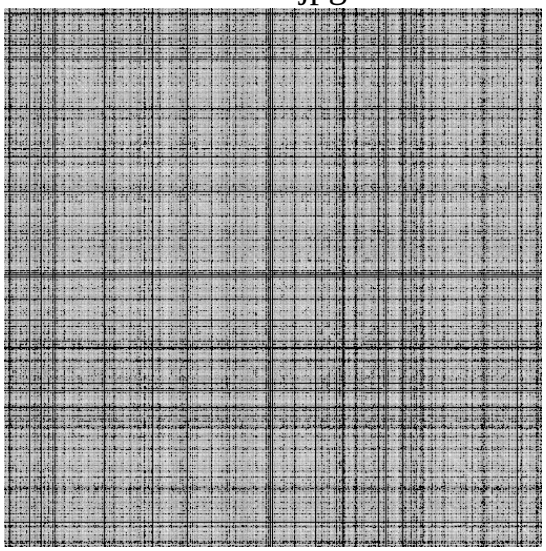
cloudy1.jpg



rain1.jpg



shine1.jpg

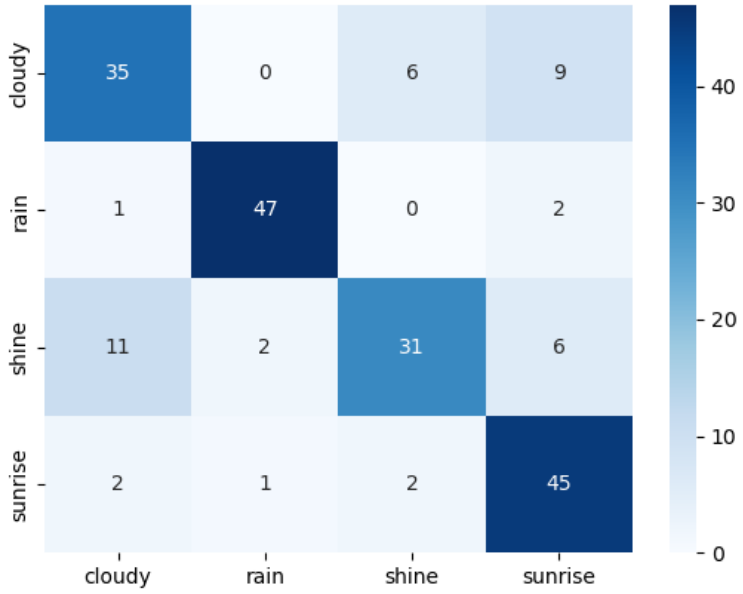


sunrise1.jpg

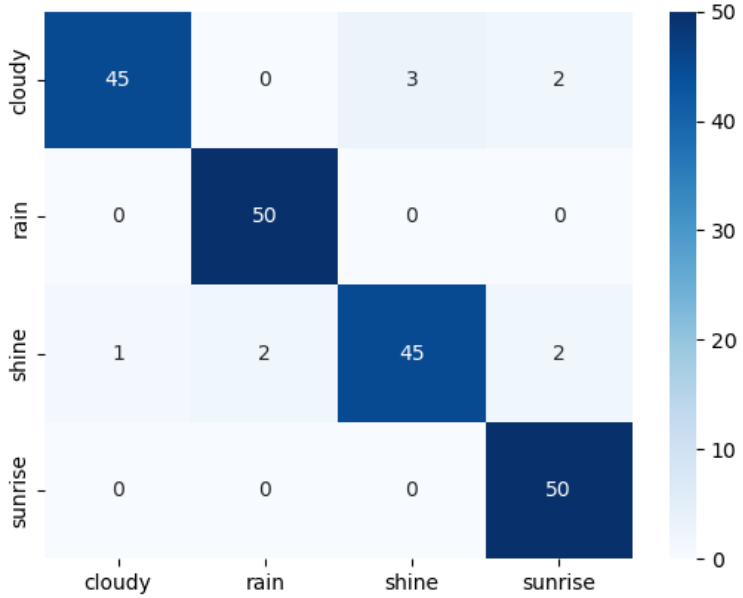
Not all the feature channels are strongly correlated. As the high intensity pixels in the Gram matrix visualization represent high correlated channels and low intensity pixels represent low correlated channels.

- **Image Classification Result**

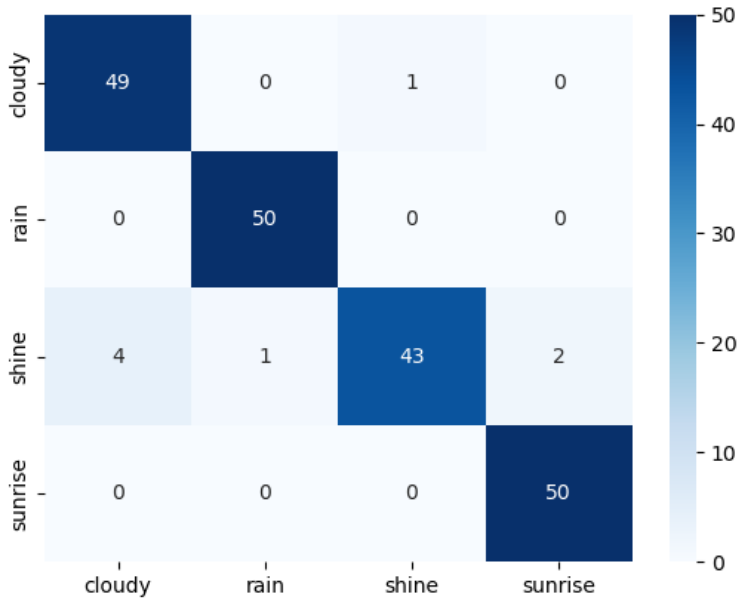
  - **LBP descriptor with SVM (polynomial kernel)**
    Accuracy: 79 %



  - **Gram matrix descriptor with SVM (linear kernel)**
    Accuracy: 95 %



- **AdaIN descriptor with SVM (linear kernel)**
  Accuracy: 96 %



The LBP descriptor performs worst with the classification accuracy of 79%. If LBP descriptors is used to train the same linear kernel SVM as the other two descriptors, the accuracy will only be 47.5 %. The AdaIN descriptor performs the best because it uses all the information in all channels. If we want to improve the accuracy of the Gram matrix descriptor, we can choose more elements in the Gram matrix to form the descriptor.

4. Source Code

```
#!/usr/bin/env python
# coding: utf-8

# In[29]:


import numpy as np
import matplotlib.pyplot as plt
import cv2
import math
import BitVector
import torch
import torch.nn as nn
from skimage import io, transform
from sklearn import svm
import os
from os import listdir
import re
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score
from sklearn.metrics import confusion_matrix
import seaborn as sn


# In[30]:


def lbp(image, R, P):

    ## modified from LBP.py
    ## Author: Avi Kak (kak@purdue.edu)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    image = cv2.resize(image, (64, 64), interpolation = cv2.INTER_AREA)
    h, w = image.shape[0], image.shape[1]

    # the parameter R is radius of the circular pattern
    # the number of points to sample on the circle

    rowmax, colmax = h-R, w-R
#     rowmax, colmax = 7, 7
    lbp_hist = [0 for t in range(P+2)]

    for i in range(R, rowmax):
        for j in range(R, colmax):
            pattern = []
            for p in range(P):
                # We use the index k to point straight down and l to point
to the
                # right in a circular neighborhood around the point (i,j).
And we
                # use (del_k, del_l) as the offset from (i,j) to the point
on the
                # R-radius circle as p varies.
```

```
                del_k, del_l = R * math.cos(2*math.pi*p/P), R *
math.sin(2*math.pi*p/P)
                if abs(del_k) < 0.001: del_k = 0.0
                if abs(del_l) < 0.001: del_l = 0.0
                k, l = i + del_k, j + del_l
                k_base,l_base = int(k),int(l)
                delta_k,delta_l = k-k_base,l-l_base

                if (delta_k < 0.001) and (delta_l < 0.001):
                    image_val_at_p = float(image[k_base][l_base])
                elif (delta_l < 0.001):
                    image_val_at_p = (1 - delta_k) * image[k_base][l_base]
+ delta_k * image[k_base+1][l_base]
                elif (delta_k < 0.001):
                    image_val_at_p = (1 - delta_l) * image[k_base][l_base]
+ delta_l * image[k_base][l_base+1]
                else:
                    image_val_at_p = (1-delta_k)*(1-
delta_l)*image[k_base][l_base] +                                        (1-
delta_k)*delta_l*image[k_base][l_base+1] +
delta_k*delta_l*image[k_base+1][l_base+1] +
delta_k*(1-delta_l)*image[k_base+1][l_base]
                if image_val_at_p >= image[i][j]:
                    pattern.append(1)
                else:
                    pattern.append(0)

            bv = BitVector.BitVector( bitlist = pattern )
            intvals_for_circular_shifts = [int(bv << 1) for _ in range(P)]
            minbv = BitVector.BitVector( intVal =
min(intvals_for_circular_shifts), size = P )

            bvruns = minbv.runs()

            if len(bvruns) > 2:
                lbp_hist[P+1] += 1
            elif len(bvruns) == 1 and bvruns[0][0] == '1':
                lbp_hist[P] += 1
            elif len(bvruns) == 1 and bvruns[0][0] == '0':
                lbp_hist[0] += 1
            else:
                lbp_hist[len(bvruns[1])] += 1

    lbp_hist = [i/sum(lbp_hist) for i in lbp_hist]

    return lbp_hist

def lbp_dataset(folder_dir):
    # get the path/directory
    X = []
    Y = []
    i = 0;
    for img in os.listdir(folder_dir):
        # check if the image ends with jpg
        if (img.endswith(".jpg") or img.endswith(".jpeg")):
            image = cv2.imread(folder_dir+'\\'+img)
            if(image is not None):
```

```
                res = re.findall('([a-zA-Z ]*)\d*.*', img)
                label = str(res[0])
                image = cv2.resize(image, (256, 256), interpolation =
cv2.INTER_AREA)
                lbp_hist = lbp(image, 1, 8)

#                 plt.figure()
#                 plt.bar(range(10), lbp_hist)
#                 plt.title(img)
#                 plt.savefig(img)

                X.append(lbp_hist)
                Y.append(label)
                i = i+1
                if i % 20 == 0:
                    print(i)
    return X, Y



# In[31]:


class VGG19(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            # encode 1-1
            nn.Conv2d(3, 3, kernel_size=(1, 1), stride=(1, 1)),
            nn.Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), padding_mode='reflect'),
            nn.ReLU(inplace=True),  # relu 1-1
            # encode 2-1
            nn.Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), padding_mode='reflect'),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False),

            nn.Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), padding_mode='reflect'),
            nn.ReLU(inplace=True),  # relu 2-1
            # encoder 3-1
            nn.Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), padding_mode='reflect'),
            nn.ReLU(inplace=True),

            nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False),
            nn.Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), padding_mode='reflect'),
            nn.ReLU(inplace=True),  # relu 3-1
            # encoder 4-1
            nn.Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), padding_mode='reflect'),
            nn.ReLU(inplace=True),
```

```
            nn.Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), padding_mode='reflect'),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), padding_mode='reflect'),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False),

            nn.Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), padding_mode='reflect'),
            nn.ReLU(inplace=True),  # relu 4-1
            # rest of vgg not used
            nn.Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), padding_mode='reflect'),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), padding_mode='reflect'),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), padding_mode='reflect'),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False),

            nn.Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), padding_mode='reflect'),
            nn.ReLU(inplace=True),  # relu 5-1
            # nn.Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), padding_mode='reflect'),
            # nn.ReLU(inplace=True),
            # nn.Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), padding_mode='reflect'),
            # nn.ReLU(inplace=True),
            # nn.Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), padding_mode='reflect'),
            # nn.ReLU(inplace=True)
        )

    def load_weights(self, path_to_weights):
        vgg_model = torch.load(path_to_weights)
        # Don't care about the extra weights
        self.model.load_state_dict(vgg_model, strict=False)
        for parameter in self.model.parameters():
            parameter.requires_grad = False

    def forward(self, x):
        # Input is numpy array of shape (H, W, 3)
        # Output is numpy array of shape (N_l, H_l, W_l)
        x = torch.from_numpy(x).permute(2, 0, 1).unsqueeze(0).float()
        out = self.model(x)
        out = out.squeeze(0).numpy()
        return out

def VGG19_dataset(path, folder_dir):
    # Load the model and the provided pretrained weights
    vgg = VGG19()
```

```
    vgg.load_weights(path+'vgg_normalized.pth')

    # get the path/directory
    X = []
    Y = []
    i = 0
    for img in os.listdir(folder_dir):
        # check if the image ends with jpg
        if (img.endswith(".jpg") or img.endswith(".jpeg")):
            image = cv2.imread(folder_dir+'\\'+img)
            if(image is not None):
                res = re.findall('([a-zA-Z ]*)\d*.*', img)
                label = str(res[0])
    #             print(label)

#                 # Read an image into numpy array
#                 x = io.imread(folder_dir+'\\'+img)
#                 # Resize the input image
#                 x = transform.resize(x, (256, 256))

                image = cv2.resize(image, (256, 256), interpolation =
cv2.INTER_AREA)

                # Obtain the output feature map
                ft = vgg(image)
                ft = ft.reshape(512, 256)
                Gram = ft.dot(np.transpose(ft))
                Gram_triu = Gram[np.triu_indices(512)]

#                 log_gram = np.log(Gram)
#                 cv2.imwrite(img, log_gram*255/np.amax(log_gram))

                np.random.seed(0)
                v_gram = np.random.choice(Gram_triu.flatten(), size=1024,
replace=False)
#                 print(v_gram)

                X.append(v_gram)
                Y.append(label)
                i = i+1
                if i % 20 == 0:
                    print(i)
    return X, Y

def VGG19_AdaIN_dataset(path, folder_dir):
    # Load the model and the provided pretrained weights
    vgg = VGG19()
    vgg.load_weights(path+'vgg_normalized.pth')

        # get the path/directory
    X = []
    Y = []
    i = 0
    for img in os.listdir(folder_dir):
        # check if the image ends with jpg
        if (img.endswith(".jpg") or img.endswith(".jpeg")):
            image = cv2.imread(folder_dir+'\\'+img)
```

```
            if(image is not None):
                res = re.findall('([a-zA-Z ]*)\d*.*', img)
                label = str(res[0])

                image = cv2.resize(image, (256, 256), interpolation =
cv2.INTER_AREA)

                # Obtain the output feature map
                ft = vgg(image)
                ft = ft.reshape(512, 256)
                ft_u = np.mean(ft, axis=1)
                ft_var = np.var(ft, axis=1)

                v_norm = np.concatenate(([ft_u], [ft_var]), axis=0)
                v_norm = np.reshape(v_norm, 1024, order='F')

                X.append(v_norm)
                Y.append(label)
                i = i+1
                if i % 20 == 0:
                    print(i)
    return X, Y


# In[32]:


if __name__ == '__main__':
    path = r'C:\Users\yhosc\Desktop\ECE661\HW7\HW7-Auxilliary\\'
    outputPath = r'C:\Users\yhosc\Desktop\ECE661\HW7\\'

#     X_train, Y_train = VGG19_dataset(path, path+'data\\tmp')

#     X_train, Y_train = lbp_dataset(path+'data\\training')
#     np.savez_compressed(path+'/lbp_training', X=X_train, Y=Y_train)

#     X_test, Y_test = lbp_dataset(path+'data\\testing')
#     np.savez_compressed(path+'/lbp_testing', X=X_test, Y=Y_test)

#     X_train, Y_train = VGG19_dataset(path, path+'data\\training')
#     np.savez_compressed(path+'/vgg19_training', X=X_train, Y=Y_train)

#     X_test, Y_test = VGG19_dataset(path, path+'data\\testing')
#     np.savez_compressed(path+'/vgg19_testing', X=X_test, Y=Y_test)

#     X_train, Y_train = VGG19_AdaIN_dataset(path, path+'data\\training')
#     np.savez_compressed(path+'/vgg19_AdaIN_training', X=X_train,
Y=Y_train)

#     X_test, Y_test = VGG19_AdaIN_dataset(path, path+'data\\testing')
#     np.savez_compressed(path+'/vgg19_AdaIN_testing', X=X_test, Y=Y_test)

#     print("finished")

    # lbp
    lbp_training = np.load(path+'/lbp_training.npz')
    lbp_testing = np.load(path+'/lbp_testing.npz')
```

```
    X_train, Y_train = lbp_training['X'], lbp_training['Y']
    X_test, Y_test = lbp_testing['X'], lbp_testing['Y']

#     model = svm.SVC(kernel='linear', C=1,
decision_function_shape='ovo').fit(X_train, Y_train)
    model = svm.SVC(kernel='poly', degree=3, C=1).fit(X_train, Y_train)
    model_pred = model.predict(X_test)

    accuracy = accuracy_score(Y_test, model_pred)
    print('Accuracy: ', "%.2f" % (accuracy*100))

    confusion_lbp = confusion_matrix(Y_test, model_pred, labels=["cloudy",
"rain", "shine", "sunrise"])

    plt.figure()
    htmp = sn.heatmap(confusion_lbp, annot=True, cmap="Blues",
xticklabels=["cloudy", "rain", "shine", "sunrise"], yticklabels=["cloudy",
"rain", "shine", "sunrise"])
    figure = htmp.get_figure()
    figure.savefig('lbp_heatmap.png')

    # VGram
    vgg19_training = np.load(path+'/vgg19_training.npz')
    vgg19_testing = np.load(path+'/vgg19_testing.npz')
    X_train, Y_train = vgg19_training['X'], vgg19_training['Y']
    X_test, Y_test = vgg19_testing['X'], vgg19_testing['Y']

    model = svm.SVC(kernel='linear', C=1,
decision_function_shape='ovo').fit(X_train, Y_train)
    model_pred = model.predict(X_test)

    accuracy = accuracy_score(Y_test, model_pred)
    print('Accuracy: ', "%.2f" % (accuracy*100))

    confusion_lbp = confusion_matrix(Y_test, model_pred, labels=["cloudy",
"rain", "shine", "sunrise"])

    plt.figure()
    htmp = sn.heatmap(confusion_lbp, annot=True, cmap="Blues",
xticklabels=["cloudy", "rain", "shine", "sunrise"], yticklabels=["cloudy",
"rain", "shine", "sunrise"])
    figure = htmp.get_figure()
    figure.savefig('VGram_heatmap.png')

    # AdaIN
    vgg19_AdaIN_training = np.load(path+'/vgg19_AdaIN_training.npz')
    vgg19_AdaIN_testing = np.load(path+'/vgg19_AdaIN_testing.npz')
    X_train, Y_train = vgg19_AdaIN_training['X'],
vgg19_AdaIN_training['Y']
    X_test, Y_test = vgg19_AdaIN_testing['X'], vgg19_AdaIN_testing['Y']

    model = svm.SVC(kernel='linear', C=1,
decision_function_shape='ovo').fit(X_train, Y_train)
    model_pred = model.predict(X_test)

    accuracy = accuracy_score(Y_test, model_pred)
    print('Accuracy: ', "%.2f" % (accuracy*100))
```

```
    confusion_lbp = confusion_matrix(Y_test, model_pred, labels=["cloudy",
"rain", "shine", "sunrise"])

    plt.figure()
    htmp = sn.heatmap(confusion_lbp, annot=True, cmap="Blues",
xticklabels=["cloudy", "rain", "shine", "sunrise"], yticklabels=["cloudy",
"rain", "shine", "sunrise"])
    figure = htmp.get_figure()
    figure.savefig('AdaIN_heatmap.png')
```