ECE 661 Homework 7

Michael Goldberg

October 30 2024

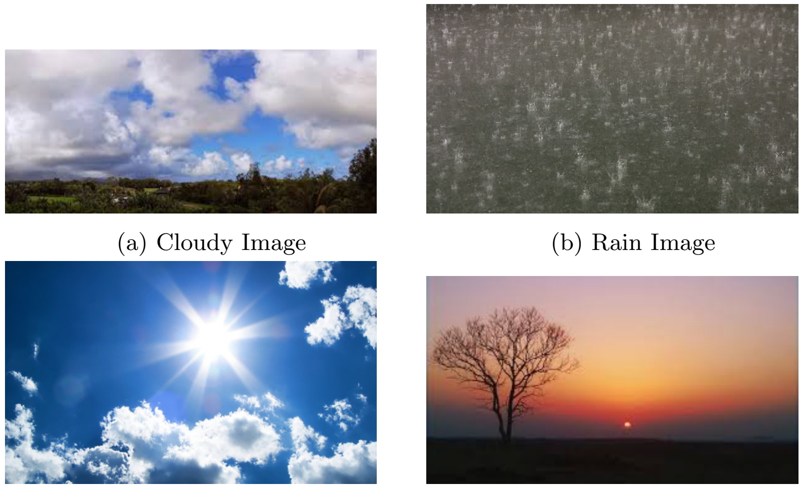
1 Theoretical Question

1.1 Custom Texture Detector

I think a cool texture detector could extend on the idea behind LBP, but instead of using a single channel, use multiple channels. Texture is encoded and interpreted very richly with the human experience, and doesn't solely lie within hue or brightness alone. Therefore, I propose a 3-channel concatenated LBP algorithm. The way this works is one first converts their image into the HSV colorspace, as the three channels are more independent of image features from one another than something like RGB. For example, if we were to image a yellow ball, the red channel and green channel would encode relatively the same information - however the hue channel and saturation channel would encode vastly different information. There could probably be some cases pointed out about where HSV has this same phenomenon happen, but with my experience with image processing, the HSV map tends to have more independent channels, so I will explore using HSV for now. The next step is to apply LBP to each of these channels individually. We then finally concatenate all three LBP histograms together to form a vector of size $(num_{neighbors} + 2) * 3$. The larger feature vector allows us to encode a richer representation of the image's texture, and should fare better than traditional LBP in a classifier. I will show a comparison of LBP versus complex LBP when subjected to an SVM classifier in the later programming section when I go over my SVM.

2 Programming Tasks

The following task was to take a dataset of images taken in four classifications of weather: cloudy, rain, shine, and sunrise and to train an SVM classifier with varying texture descriptors. The four descriptors used were Gram Matrices from VGG19 encodings, Gram Matrices from ResNet50-Coarse encodings, Gram Matrices from ResNet50-Fine encodings, and LBP histograms. The following are examples of an image from each class:



(a) Cloudy Image

(b) Rain Image

(c) Shine Image

(d) Sunrise Image

2.1 LBP Implementation

The first step was to create the LBP feature extraction algorithm from scratch. LBP is performed on a single-channel image, and we were instructed to use the hue channel of the images converted from RGB to HSV. The LBP algorithm assigns a texture value to a pixel by counting how many neighboring pixel values are larger than it and forming a histogram of these counts, with some exceptions which I will go into later. LBP takes a few parameters, mainly the radius of the search r as well as how many neighbor points to consider N. The algorithm starts by calculating the angle that each of these neighbor points form, and calculating their subpixel locations using the following equation.

$$x_n = j + r\cos\theta_n y_n = i - r\sin\theta_n \tag{1}$$

Here, i and j represent the integer coordinates of the central pixel, $n\epsilon N$ represents the index of the nth neighbor value, and θ_n represents the angle of the nth neighbor point. The subpixel value of this location can then be calculated using bilinear interpolation.

Once this subpixel value is calculated, we can create a binary word of size N bits, where each bit corresponds to a certain neighbor. If that particular neighbor's value is larger than the central pixel's value, its corresponding bit is set to 1.

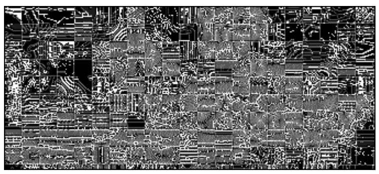
This word is then circularly rotated such that the longest string of 0s are at the left. We then can assign the value of the central pixel using the following rules:

- If the word is uniform, meaning there is a single transition from 0s to 1s when reading left to right, then the central pixel is assigned an integer equal to the number of 1s in the word

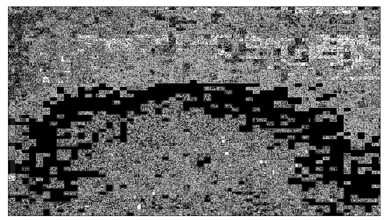
- If the word is not uniform, meaning there is a transition from 1s to 0s, then the central pixel is assigned an integer equal to $N+1$

This process is repeated for all pixels across the image, and a histogram is created counting up the values assigned to each pixel and placing it in each bin. Since the binary word is N elements long, there are $N+2$ bins (considering the cases of non-uniform words and words equal to 0).

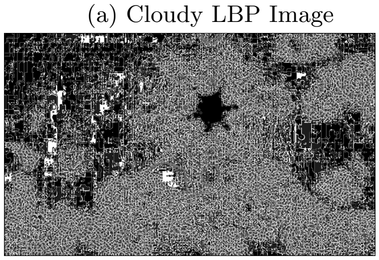
Here are what the LBP images and histograms look like for each of the original images shown above with $N=8$ and $r=1$:



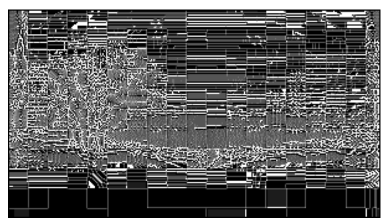
(a) Cloudy LBP Image



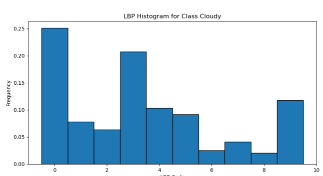
(b) Rain LBP Image



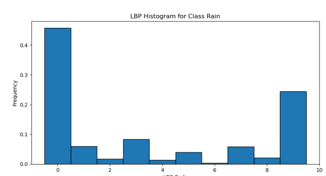
(c) Shine LBP Image



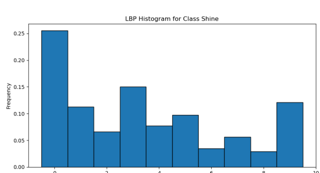
(d) Sunrise LBP Image



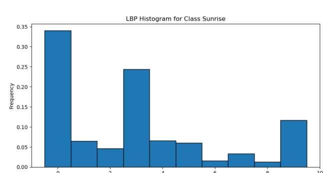
(a) Cloudy Histogram



(b) Rain Histogram



(c) Shine Histogram



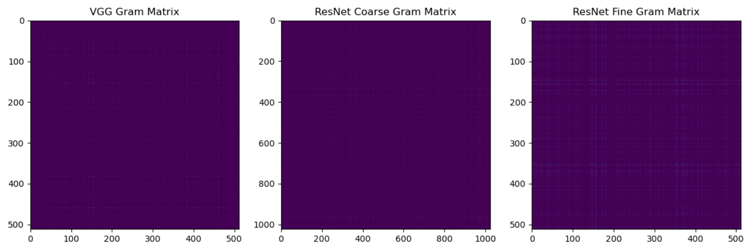
(d) Sunrise Histogram

2.2 Gram Matrices

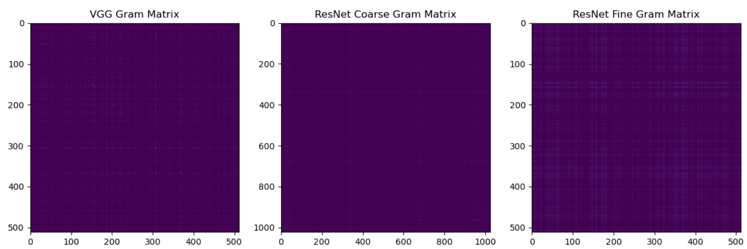
Gram matrices are formed by simply taking a feature vector F_l and multiplying it by its transpose:

$$G = F_l * F_l^T \tag{2}$$

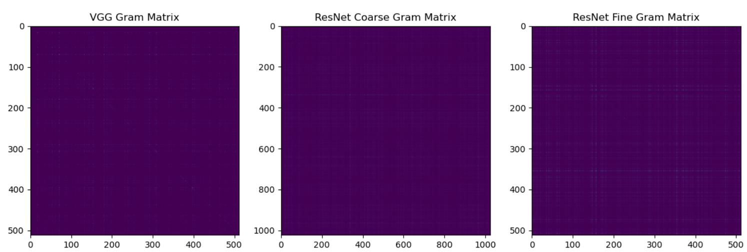
Both VGG19 and ResNet encode images into feature vectors, so I just had to perform the above calculation on the results of each of the encodings to find their gram matrices. Below are the gram matrics formed from VGG19, ResNet50-Coarse, and ResNet50-Fine on the above original images. Additionally, from here on out, I will refer to these encodings as vgg, resnet-coarse, and resnet-fine. Please zoom in to see the slight differences between them.



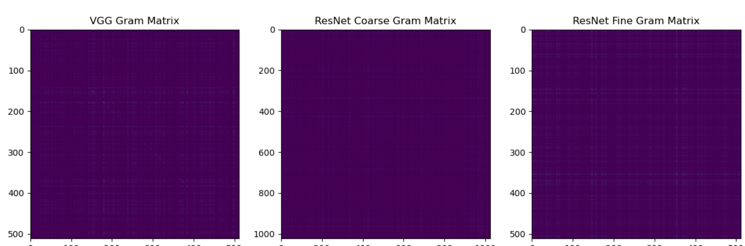
(a) Cloudy Gram Matrices



(b) Rain Gram Matrices



(c) Shine Gram Matrices



(d) Sunrise Gram Matrices

However, because G is a square symmetric matrix this is both redundant and incompatible with an SVM. We therefore need to transform G into a 1D vector, and remove the duplicate entries due to symmetry. Therefore, all that was done was to flatten the upper-triangular representation of G, and that was the feature vector used for training the SVM with vgg, resnet-coarse, and resnet-fine.

2.3 Results

An SVM was trained using scikit-learn with the following best-performing hyperparameters for vgg, resnet-coarse, and resnet-fine:

- batch size = 1024
- kernel = ”linear”
- probability = True
- C = 1

However, these hyperparameters performed best for LBP:

- batch size = 1024
- kernel = ”rbf”
- probability = True
- C = 10000
- N (number of neighbors for LBP) = 16
- r (radius for LBP) = 2

Here are the resultant confusion matrices and accuracies for each texture descriptor:

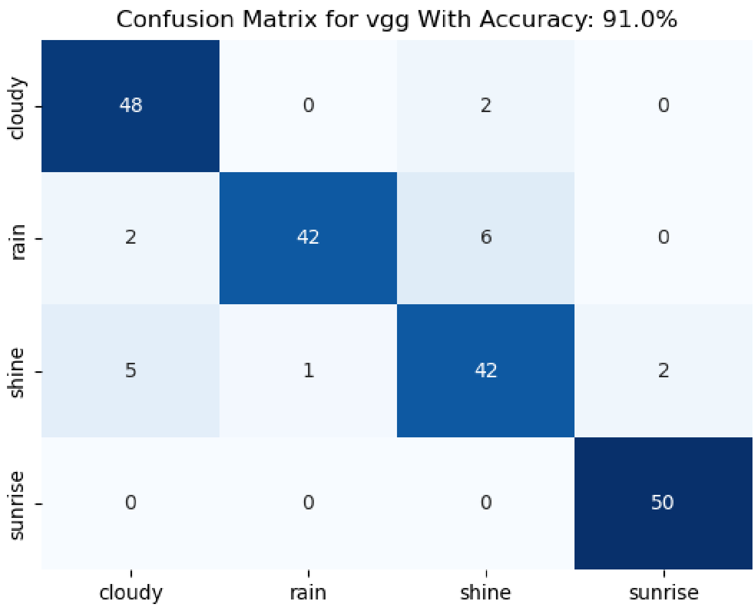


Figure 5: VGG Confusion Matrix, Accuracy = 91%

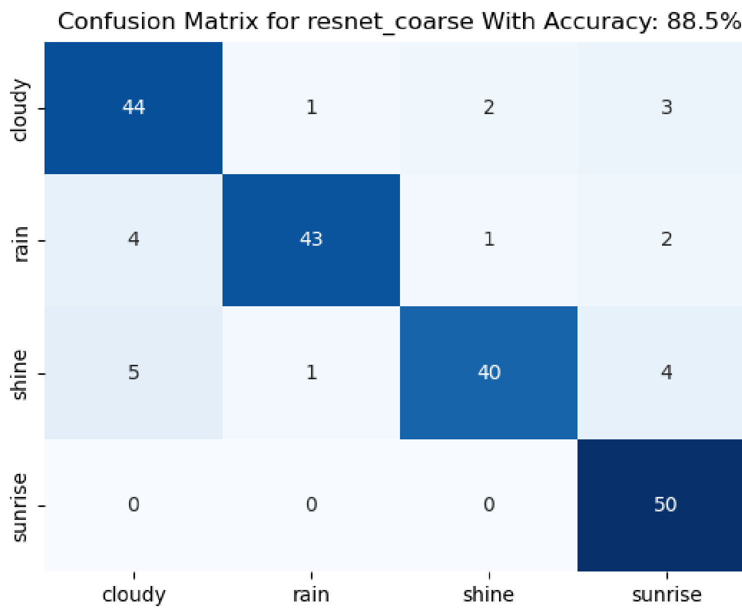


Figure 6: Resnet-Coarse Confusion Matrix, Accuracy = 88.5%

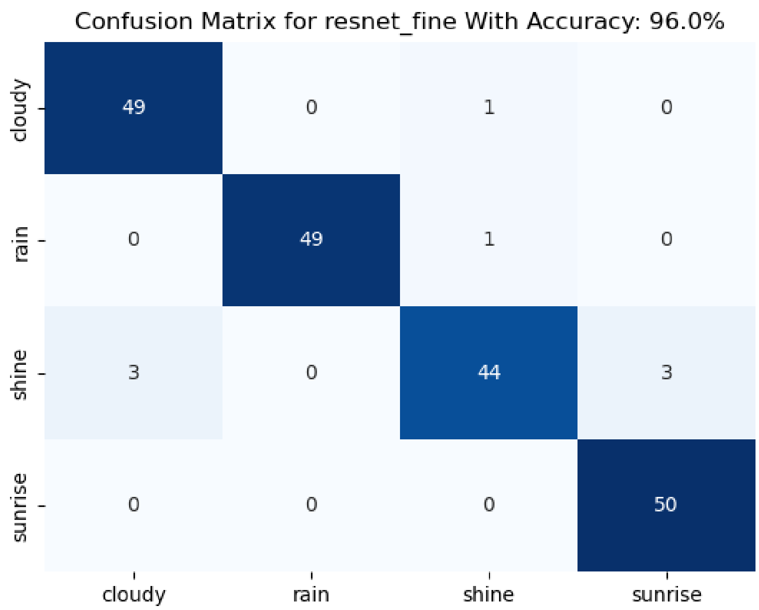


Figure 7: Resnet-Fine Confusion Matrix, Accuracy = 96%

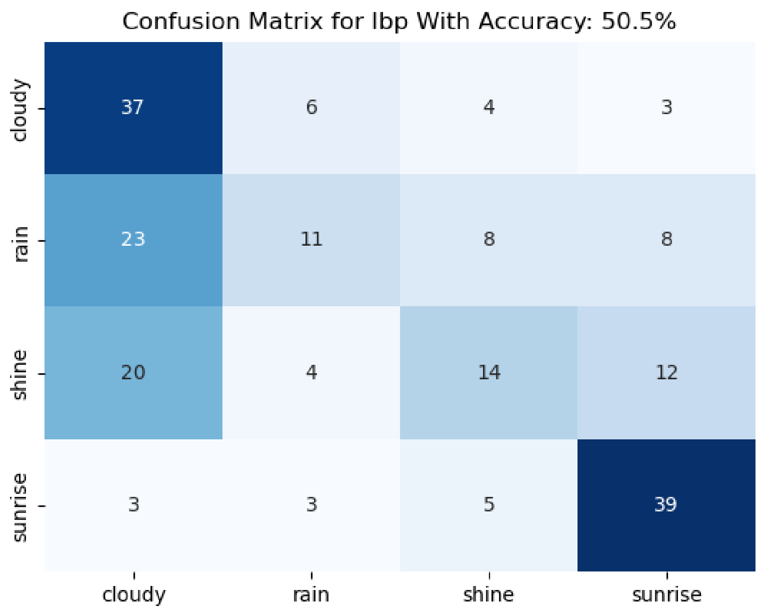


Figure 8: LBP Confusion Matrix, Accuracy = 50.5%

Let's take at some images that were correctly classified:

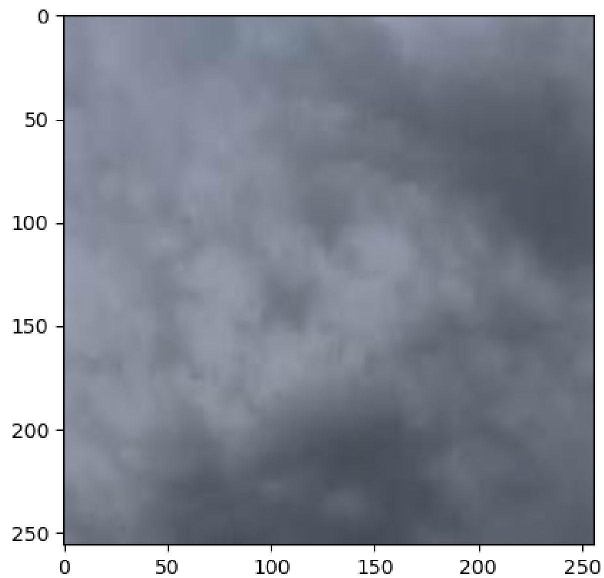


Figure 9: Correctly Classified Image with VGG Descriptors: Predicted=Actual=Cloudy

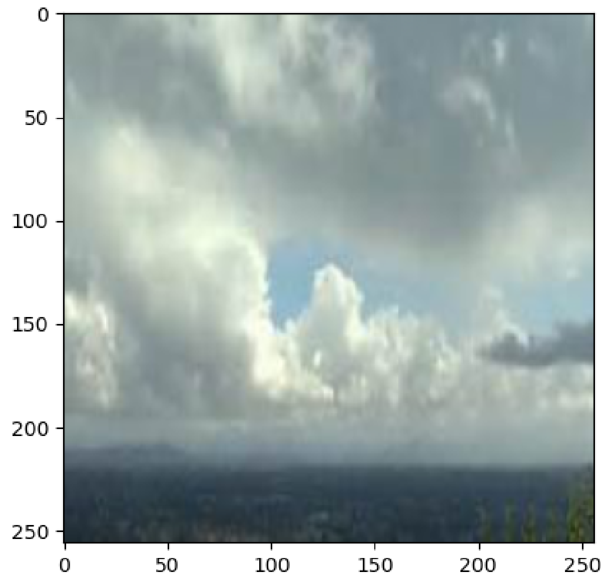


Figure 10: Correctly Classified Image with Resnet-Coarse Descriptors: Predicted=Actual=Cloudy

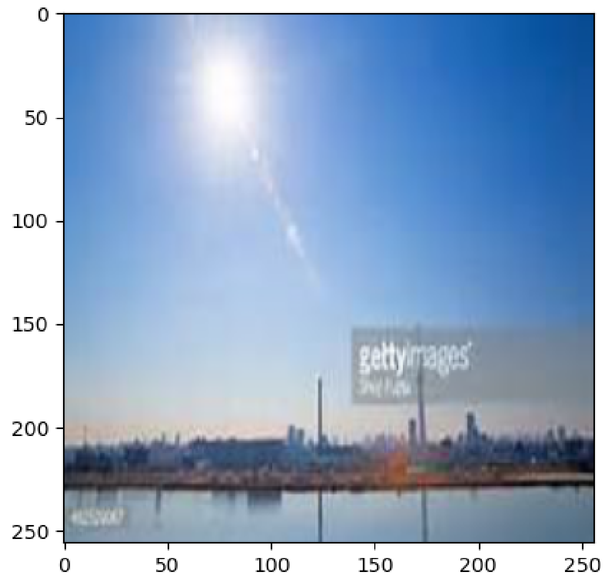


Figure 11: Correctly Classified Image with Resnet-Fine Descriptors: Predicted=Actual=Shine

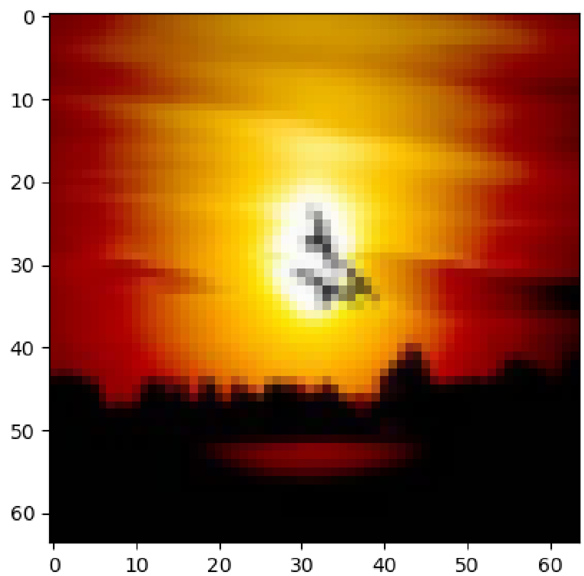


Figure 12: Correctly Classified Image with LBP Descriptors: Predicted=Actual=Sunrise

For vgg and both resnet feature descriptors, I was able to get classification accuracy at close to or above 90%, with the best performer being resnet-fine at 96% accuracy. It is clear, however, why LBP works significantly worse than the other texture descriptors, and that is because a histogram of size N is too small to depict the richer representation of texture that the other strategies can convey. This is why I settled on my custom texture descriptor being the 3-channel LBP algorithm, as this triples the size of the feature vector, while also taking advantage of the other channels to extract unique texture features from the image. Below is a comparison of the complex LBP trained on the same SVM with the same hyperparameters as the standard LBP implementation:

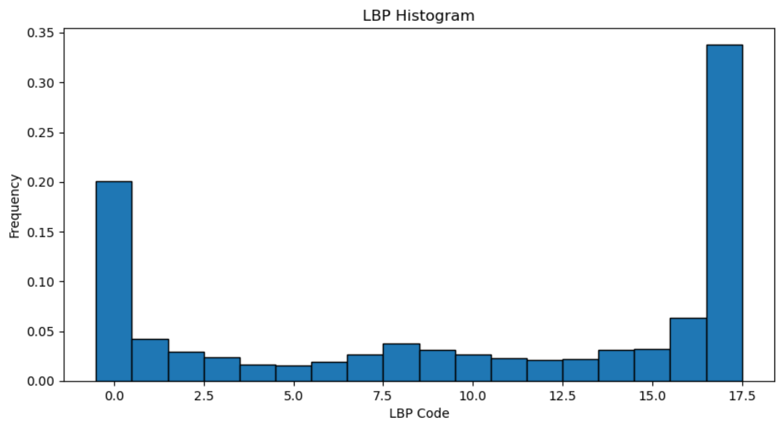


Figure 13: LBP Histogram

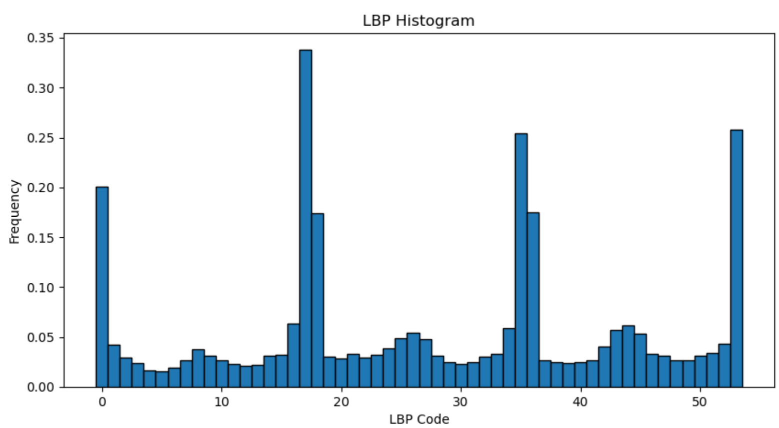


Figure 14: Complex LBP Histogram

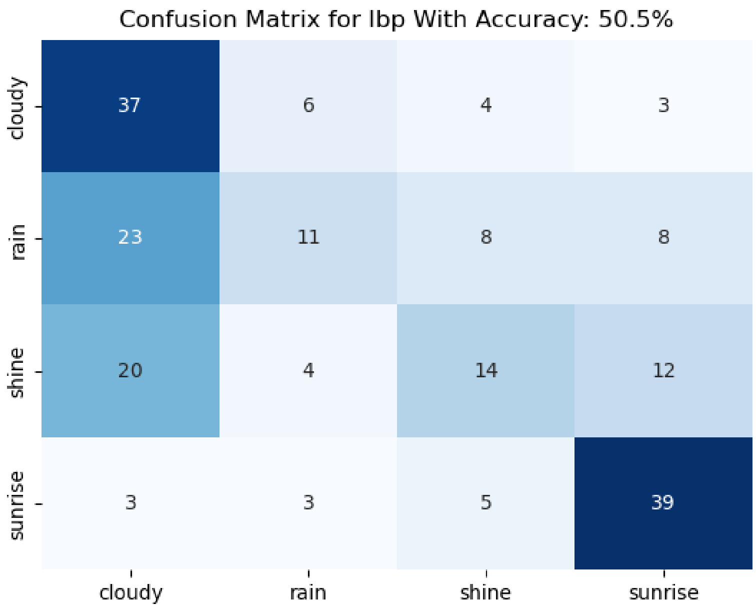


Figure 15: LBP Confusion Matrix, Accuracy = 50.5%

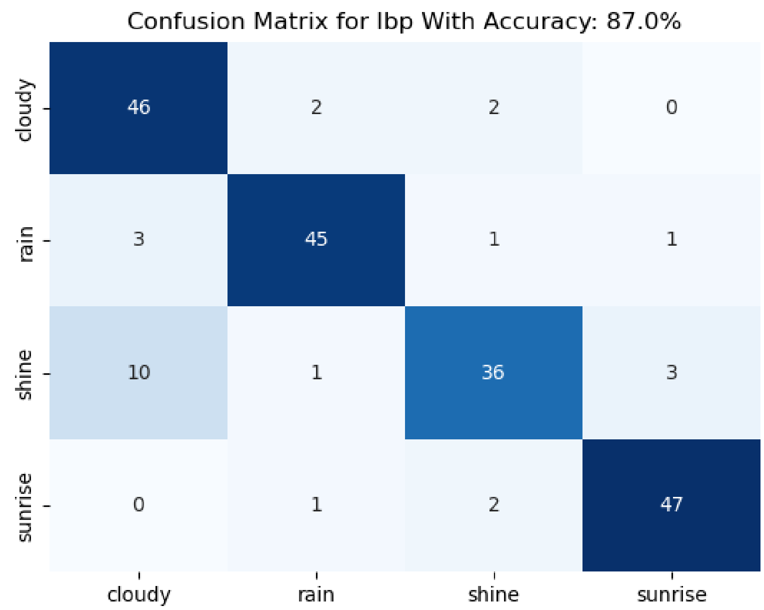


Figure 16: Complex LBP Confusion Matrix, Accuracy = 87%

This definitely increased the performance of the LBP methodology significantly, and it's pretty cool to see a custom implementation come to fruition like this.

However none of these descriptors were without error. Below are some example images that were misclassified by the networks:

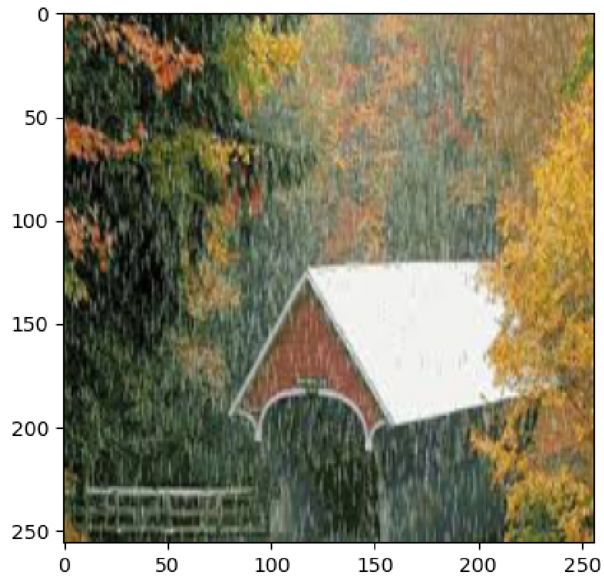


Figure 17: Incorrectly Classified Image with VGG Descriptors: Predicted=Shine, Actual=Rain

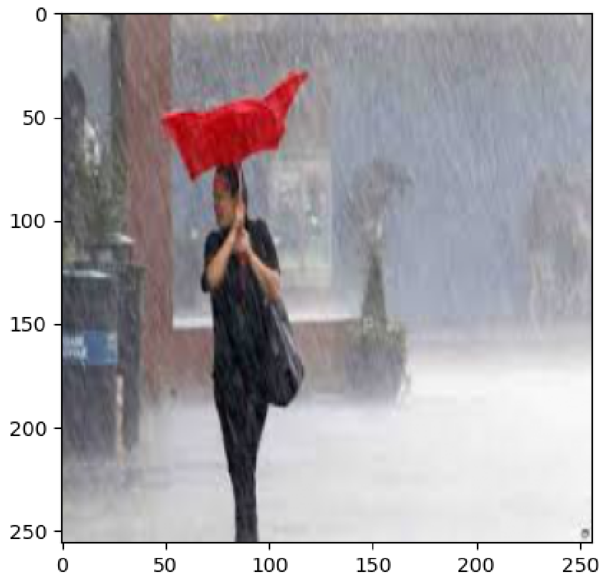


Figure 18: Incorrectly Classified Image with Resnet-Coarse Descriptors: Predicted-Shine, Actual=Rain

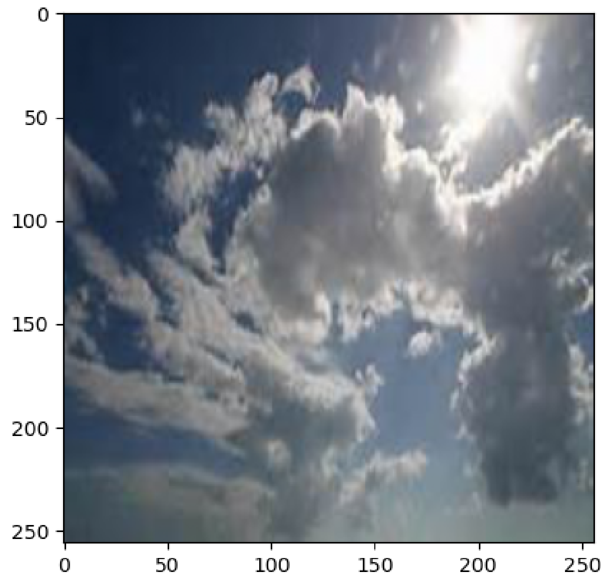


Figure 19: Incorrectly Classified Image with Resnet-Fine Descriptors: Predicted=Shine, Actual=Cloudy

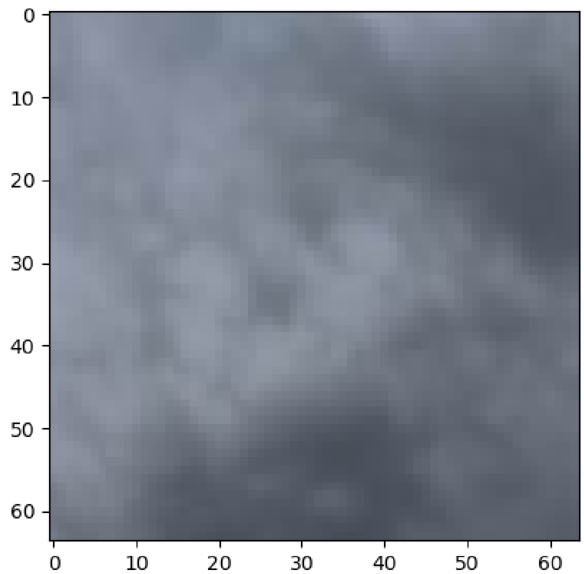


Figure 20: Incorrectly Classified Image with LBP Descriptors: Predicted=Rain, Actual=Cloudy

3 Code

```
import os
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import importlib
import seaborn as sns

from skimage import io, transform
from skimage.measure import block_reduce
from torchvision.models import ResNet50_Weights
from sklearn import svm
from sklearn.metrics import classification_report, accuracy_score,
    confusion_matrix, ConfusionMatrixDisplay
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from tqdm import tqdm

num_neighbors = 16
lbp_radius = 2

downsample_size = 32
```

```python

device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")

label_map = {"cloudy": 0, "rain": 1, "shine": 2, "sunrise": 3}
reverse_label_map = ["cloudy", "rain", "shine", "sunrise"]

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), #1/2

            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), #1/4
            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), #1/8

            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),
```

```python
            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), #1/16

            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, weights_only=True)
        # 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(0, 3, 1, 2).float().to(device)
        out = self.model(x)
        out = out.cpu().numpy()
        return out

    def extract_features(self, images, batch_size=32):
        vgg_features = []

        for i in range(0, len(images), batch_size):
            batch_images = images[i:i + batch_size]

            features = self.forward(batch_images)
            vgg_features.append(features)

            torch.mps.empty_cache()

        return np.vstack(vgg_features)

def class_for_name(module_name, class_name):
    # load the module, will raise ImportError if module cannot be loaded
    m = importlib.import_module(module_name)
    return getattr(m, class_name)
```

```python

class CustomResNet(nn.Module):
    def __init__(self,
                 encoder='resnet50',
                 weights=ResNet50_Weights.DEFAULT):

        super(CustomResNet, self).__init__()
        assert encoder in ['resnet18', 'resnet34', 'resnet50', 'resnet101', 'resnet152'], "Incorrect encoder type"
        # if encoder in ['resnet18', 'resnet34']:
        #     filters = [64, 128, 256, 512]
        # else:
        #     filters = [256, 512, 1024, 2048]
        resnet = class_for_name("torchvision.models", encoder)(weights=weights)

        for parameter in resnet.parameters():
            parameter.requires_grad = False

        self.firstconv = resnet.conv1  # H/2
        self.firstbn = resnet.bn1
        self.firstrelu = resnet.relu
        self.firstmaxpool = resnet.maxpool  # H/4

        # encoder
        self.layer1 = resnet.layer1  # H/4
        self.layer2 = resnet.layer2  # H/8
        self.layer3 = resnet.layer3  # H/16

    def forward(self, x):
        """
        Coarse and Fine Feature extraction using ResNet
        Coarse Feature Map has smaller spatial sizes.
        Arg:
            x: (np.array) [H,W,C]
        Return:
            xc: (np.array) [C_coarse, H/16, W/16]
            xf: (np.array) [C_fine, H/8, W/8]
        """
        x = torch.from_numpy(x).permute(0, 3, 1, 2).float().to(device)

        x = self.firstrelu(self.firstbn(self.firstconv(x))) #1/2
        x = self.firstmaxpool(x) #1/4

        x = self.layer1(x) #1/4
        xf = self.layer2(x) #1/8
        xc = self.layer3(xf) #1/16

        # convert xc, xf to numpy
        xc = xc.cpu().detach().numpy()
        xf = xf.cpu().detach().numpy()
        return xc, xf

    def extract_features(self, images, batch_size=32):
        coarse_features = []
        fine_features = []
```

```python

        for i in range(0, len(images), batch_size):
            batch_images = images[i:i + batch_size]

            features_coarse, features_fine = self.forward(batch_images)

            coarse_features.append(features_coarse)
            fine_features.append(features_fine)

            torch.mps.empty_cache()

        return np.vstack(coarse_features), np.vstack(fine_features)

def bilinear_interpolate(image, x, y):
    # get the four surrounding pixel values
    x0, y0 = int(x), int(y)
    x1, y1 = min(x0 + 1, image.shape[1] - 1), min(y0 + 1, image.shape[0] - 1)

    # calculate weights for each pixel based on distance from integer pixel locations
    wa = (x1 - x) * (y1 - y)
    wb = (x1 - x) * (y - y0)
    wc = (x - x0) * (y1 - y)
    wd = (x - x0) * (y - y0)

    # weighted sum to find interpolated value
    interpolated_value = wa * image[y0, x0] + wb * image[y1, x0] + wc * image[y0, x1] + wd * image[y1, x1]
    return interpolated_value

def rgb2hsv(image):
    # cv2.imshow("original", image)

    # normalize the image and convert from bgr to rgb
    image = np.asarray(image, dtype=float) / 255.0

    r, g, b = image[:, :, 0], image[:, :, 1], image[:, :, 2]

    chroma_max = np.maximum(np.maximum(r, g), b) # maximum chroma is the maximum value across all three RGB channels
    chroma_min = np.minimum(np.minimum(r, g), b) # minimum chroma is the minimum value across all three RGB channels
    delta = chroma_max - chroma_min # delta is the difference in max vs. min chroma

    hue = np.zeros_like(chroma_max)
    mask = delta != 0

    # Using well-established conversion between rgb and hue, using masking to prevent division by 0 and max_chroma checking to determine which equation to use
    hue[mask & (chroma_max == r)] = ((g[mask & (chroma_max == r)] - b[mask & (chroma_max == r)]) / delta[mask & (chroma_max == r)]) % 6
```

```
    hue[mask & (chroma_max == g)] = ((b[mask & (chroma_max == g)] - r[mask & (chroma_max == g)]) / delta[mask & (chroma_max == g)]) + 2
    hue[mask & (chroma_max == b)] = ((r[mask & (chroma_max == b)] - g[mask & (chroma_max == b)]) / delta[mask & (chroma_max == b)]) + 4

    hue *= 60 # convert to degrees on the color wheel
    hue[hue < 0] += 360 # prevent negative values

    # Using well-established saturation calculation where S = 0 if chroma_max = 0 and S = delta / chroma_max otherwise
    saturation = np.zeros_like(chroma_max)
    saturation[chroma_max != 0] = delta[chroma_max != 0] / chroma_max[chroma_max != 0]

    value = chroma_max

    hsv_image = np.stack([hue, saturation, value], axis=-1)

    return hsv_image

def check_if_code_uniform(code):

    # if code is 0 or negative, nonuniform for purposes of lbp
    if code <= 0:
        return False

    # collect the unlabelled binary representation of the code
    binary = bin(code)[2:]

    # set previous bit to first bit
    prev_bit = binary[0]

    # if there is ever a situation where the previous bit is 1 and the next bit is 0, nonuniform
    for bit in binary[1:]:
        if(prev_bit == "1" and bit == "0"):
            return False
        prev_bit = bit

    return True

def lbp_histogram(lbp_image, num_bins=num_neighbors + 2):
    histogram, _ = np.histogram(lbp_image.ravel(), bins=num_bins, range=(0, num_bins))

    # normalization
    histogram = histogram.astype("float")
    histogram /= (histogram.sum() + 1e-6)

    return histogram

def lbp_descriptor(image, radius=lbp_radius, num_neighbors=num_neighbors):

    # collect hue channel of image and normalize to 0-255
```

```
    image = (rgb2hsv(image)[:, :, 0] / 360.0) * 255.0


    height, width = image.shape

    lbp_image = np.zeros((height, width), dtype=np.uint8)

    # creation of array of angles that each neighbor point makes with the central point
    angles = [2 * np.pi * i / num_neighbors for i in range(num_neighbors)]

    for i in range(radius, height - radius):
        for j in range(radius, width - radius):
            center = image[i, j]
            lbp_code = 0

            for idx, angle in enumerate(angles):
                # collecting subpixel value of neighbor point
                x = j + radius * np.cos(angle)
                y = i - radius * np.sin(angle)

                # bilinear interpolation to determine neighbor's subpixel value
                neighbor = bilinear_interpolate(image, x, y)

                # setting bits to 1 if their corresponding neighbor is greater than the center pixel
                lbp_code |= (neighbor > center) << idx

            min_val = lbp_code

            # circularly shifting to produce the largest number of zeros on the left of the lbp code
            for _ in range(num_neighbors):
                lbp_code = (lbp_code >> 1) | ((lbp_code & 1) << (num_neighbors - 1))
                min_val = min(min_val, lbp_code)

            # if code is 0, set lbp label to 0. If code uniform, set to number of bits that are 1. If nonuniform, set label to num_neighbors + 1. This is in accordance with the handout
            if(min_val == 0):
                lbp_image[i, j] = 0
            elif(check_if_code_uniform(min_val)):
                lbp_image[i, j] = bin(min_val).count('1')
            else:
                lbp_image[i, j] = num_neighbors + 1

    return lbp_histogram(lbp_image)

def complex_lbp_descriptor(image, radius=lbp_radius, num_neighbors=num_neighbors):
    height, width = image.shape

    lbp_image = np.zeros((height, width), dtype=np.uint8)

```

```
    # creation of array of angles that each neighbor point makes with the central point
    angles = [2 * np.pi * i / num_neighbors for i in range(num_neighbors)]

    for i in range(radius, height - radius):
        for j in range(radius, width - radius):
            center = image[i, j]
            lbp_code = 0

            for idx, angle in enumerate(angles):
                # collecting subpixel value of neighbor point
                x = j + radius * np.cos(angle)
                y = i - radius * np.sin(angle)

                # bilinear interpolation to determine neighbor's subpixel value
                neighbor = bilinear_interpolate(image, x, y)

                # setting bits to 1 if their corresponding neighbor is greater than the center pixel
                lbp_code |= (neighbor > center) << idx

            min_val = lbp_code

            # circularly shifting to produce the largest number of zeros on the left of the lbp code
            for _ in range(num_neighbors):
                lbp_code = (lbp_code >> 1) | ((lbp_code & 1) << (num_neighbors - 1))
                min_val = min(min_val, lbp_code)

            # if code is 0, set lbp label to 0. If code uniform, set to number of bits that are 1. If nonuniform, set label to num_neighbors + 1. This is in accordance with the handout
            if(min_val == 0):
                lbp_image[i, j] = 0
            elif(check_if_code_uniform(min_val)):
                lbp_image[i, j] = bin(min_val).count('1')
            else:
                lbp_image[i, j] = num_neighbors + 1

    return lbp_histogram(lbp_image)

def form_gram_matrix_vector(feature_tensor):
    # Collect sizes of input tensor
    batch_size = feature_tensor.shape[0]

    C = int(feature_tensor.shape[1] / downsample_size)

    # Setting to size int(C * (C + 1) / 2) due to only collecting upper triangular portion of gram matrix
    gram_tensor = np.zeros((batch_size, int(downsample_size * (downsample_size + 1) / 2)))

    # Go through each element in tensor and calculate gram matrix and turn to flattened upper triangular vector
```

```python
    for i in range(batch_size):
        feature_vector = feature_tensor[i]

        F_l = feature_vector.reshape(feature_vector.shape[0], feature_vector.shape[1] * feature_vector.shape[2])
        G = F_l @ F_l.T

        G = block_reduce(G, block_size=(C, C), func=np.mean)

        upper_triangular = G[np.triu_indices_from(G)]

        gram_tensor[i] = upper_triangular.flatten()

    return gram_tensor

def plot_gram_matrices(vgg_gram, resnet_coarse_gram, resnet_fine_gram):
    # This function only works if you input the 2D (nontensor) versions of the gram matrices and is for visualization purposes only
    fig, axis = plt.subplots(1, 3, figsize=(15, 5))

    axis[0].imshow(vgg_gram, cmap='viridis')
    axis[0].set_title("VGG Gram Matrix")

    axis[1].imshow(resnet_coarse_gram, cmap='viridis')
    axis[1].set_title("ResNet Coarse Gram Matrix")

    axis[2].imshow(resnet_fine_gram, cmap='viridis')
    axis[2].set_title("ResNet Fine Gram Matrix")

    plt.show()

def load_images(directory):
    images, labels, lbp_images = [], [], []

    # Loop through all jpg files in given directory, extract the label and append image and label to output lists
    for filename in os.listdir(directory):
        if filename.endswith(".jpg"):
            img_path = os.path.join(directory, filename)
            image = io.imread(img_path)

            if len(image.shape) == 2:  # Grayscale image
                image = np.stack((image,)*3, axis=-1)  # Convert to RGB by repeating the channel
            elif image.shape[2] == 4:  # RGBA image
                image = image[:, :, :3]  # Convert to RGB by discarding the alpha channel

            image = transform.resize(image, (256, 256), anti_aliasing=True, mode='reflect')
            lbp_image = transform.resize(image, (64, 64), anti_aliasing=True, mode='reflect')

            images.append(image)
            lbp_images.append(lbp_image)
```

```python

            if filename.startswith("cloudy"):
                labels.append(label_map["cloudy"])
            elif filename.startswith("rain"):
                labels.append(label_map["rain"])
            elif filename.startswith("shine"):
                labels.append(label_map["shine"])
            else:
                labels.append(label_map["sunrise"])

    # Returning numpy arrays of output lists
    return np.array(images), np.array(labels), np.array(lbp_images)

def create_lbp_tensor(images):
    lbp_histograms = []

    # Run lbp on each image and form a tensor for all images in set
    for i in range(int(images.shape[0])):
        lbp_histograms.append(lbp_descriptor(images[i]))
        print("processed LBP for {} images out of {}".format(i + 1,
     images.shape[0]))

    return np.vstack(lbp_histograms)

def create_complex_lbp_tensor(images):
    complex_lbp_histograms = []

    for i in range(int(images.shape[0])):
        image = rgb2hsv(images[i])
        image_hue = (image[:, :, 0] / 360.0) * 255.0
        image_sat = (image[:, :, 1]) * 255.0
        image_val = (image[:, :, 2]) * 255.0

        hue_hist = complex_lbp_descriptor(image_hue)
        sat_hist = complex_lbp_descriptor(image_sat)
        val_hist = complex_lbp_descriptor(image_val)

        histogram = np.hstack([hue_hist, sat_hist, val_hist])
        complex_lbp_histograms.append(histogram)
        print("processed Complex LBP for {} images out of {}".
    format(i + 1, images.shape[0]))


    return np.vstack(complex_lbp_histograms)


def train_svm(train_features, train_labels, test_features,
    test_labels, batch_size, namestring):

    # Create the SVM classifier
    if(namestring == "lbp"):
        # Best performing classifier for LBP
        classifier = svm.SVC(kernel='rbf', probability=True, C
    =10000)
    else:
        # Best performing classifier for non-LBP
        classifier = svm.SVC(kernel='linear', probability=True)
```

```python

    # Calculate the number of batches
    num_batches = int(np.ceil(len(train_features) / batch_size))

    # Train the SVM in batches with a progress bar
    for i in tqdm(range(num_batches), desc='Training SVM'):
        start_idx = i * batch_size
        end_idx = min((i + 1) * batch_size, len(train_features))
        X_batch = train_features[start_idx:end_idx]
        y_batch = train_labels[start_idx:end_idx]

        # Fit the model on the current batch
        classifier.fit(X_batch, y_batch)

    # Make predictions on the test set
    y_pred = classifier.predict(test_features)

    accuracy = accuracy_score(test_labels, y_pred)
    print("Accuracy:", accuracy)

    # Generate the confusion matrix
    conf_matrix = confusion_matrix(test_labels, y_pred)
    print("Confusion Matrix:")
    print(conf_matrix)

    # Visualization of confusion matrix
    sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", cbar=False, xticklabels=reverse_label_map, yticklabels=reverse_label_map)
    plt.title("Confusion Matrix for {} With Accuracy: {}%".format(namestring, accuracy * 100))
    plt.show()

    return classifier

def plot_histogram(histogram):
    plt.figure(figsize=(10, 5))
    plt.bar(np.arange(len(histogram)), histogram, width=1, edgecolor='black')
    plt.xlabel('LBP Code')
    plt.ylabel('Frequency')
    plt.title("LBP Histogram")
    plt.show()

def find_correct_and_incorrect_predictions(classifier, test_feature_vector, test_images, test_labels):
    correct_found = 0 # Setting to counter because all classifiers were identifying same correct image. By collecting a large amount of correct images (in this case 15) I can get some variety
    incorrect_found = False

    for i in range(test_feature_vector.shape[0]):
        y_pred = classifier.predict(test_feature_vector[i].reshape(1, -1))[0]

        if((y_pred != test_labels[i]) and not incorrect_found):
```

```
            incorrect_found = True
            print("Incorrect Image Found")
            print("Predicted class: ", reverse_label_map[y_pred])
            print("Actual class: ", reverse_label_map[test_labels[i]])

            plt.imshow(test_images[i])
            plt.show()
        if((y_pred == test_labels[i]) and correct_found < 15):
            correct_found += 1
            print("Correct Image Found")
            print("Predicted class: ", reverse_label_map[y_pred])
            print("Actual class: ", reverse_label_map[test_labels[i]])

            plt.imshow(test_images[i])
            plt.show()
        if(correct_found >= 15 and incorrect_found):
            break

if __name__ == '__main__':
    train_dir = "data/training"
    test_dir = "data/testing"
    encoder_name = "resnet50"

    preprocessing = False

    # Loading images
    print("loading training images")
    train_images, train_labels, train_lbp_images = load_images(train_dir)
    print("done loading training images, now loading testing")
    test_images, test_labels, test_lbp_images = load_images(test_dir)
    print("done loading testing images")

    if(preprocessing):
        # ---------- Pre processing -----------------------#
        # Saving labels so preprocessing only needs to be run once
        torch.save(train_labels, "train_labels.pt")
        torch.save(test_labels, "test_labels.pt")

        # Encoding images with VGG19
        vgg = VGG19()
        vgg.load_weights("vgg_normalized.pth")
        vgg.to(device)
        vgg_feature_train = vgg.extract_features(train_images)
        vgg_feature_test = vgg.extract_features(test_images)

        # Encoding images with ResNet Coarse and Fine
        resnet = CustomResNet(encoder=encoder_name)
        resnet.to(device)
        resnet_coarse_feature_train, resnet_fine_feature_train= resnet(train_images)
        resnet_coarse_feature_test, resnet_fine_feature_test= resnet(test_images)

```

```python
    # Calculating gram matrices for vgg and resnet feature vectors
    vgg_gram_train = form_gram_matrix_vector(vgg_feature_train)
    resnet_coarse_gram_train = form_gram_matrix_vector(resnet_coarse_feature_train)
    resnet_fine_gram_train = form_gram_matrix_vector(resnet_fine_feature_train)
    vgg_gram_test = form_gram_matrix_vector(vgg_feature_test)
    resnet_coarse_gram_test = form_gram_matrix_vector(resnet_coarse_feature_test)
    resnet_fine_gram_test = form_gram_matrix_vector(resnet_fine_feature_test)


    # Encoding images with LBP
    lbp_feature_train = create_lbp_tensor(train_lbp_images)
    lbp_feature_test = create_lbp_tensor(test_lbp_images)

    # Custom texture extractor Complex LBP
    complex_lbp_feature_train = create_complex_lbp_tensor(train_lbp_images)
    complex_lbp_feature_test = create_complex_lbp_tensor(test_lbp_images)

    print("VGG Gram Train Size: ", vgg_gram_train.shape)
    print("VGG Gram Test Size: ", vgg_gram_test.shape)

    print("ResNet Coarse Gram Train Size: ", resnet_coarse_gram_train.shape)
    print("ResNet Coarse Gram Test Size: ", resnet_coarse_gram_test.shape)

    print("ResNet Fine Gram Train Size: ", resnet_fine_gram_train.shape)
    print("ResNet Fine Gram Test Size: ", resnet_fine_gram_test.shape)

    print("LBP Feature Train Size: ", lbp_feature_train.shape)
    print("LBP Feature Test Size: ", lbp_feature_test.shape)

    print("Complex LBP Feature Train Size: ", complex_lbp_feature_train.shape)
    print("Complex LBP Feature Test Size: ", complex_lbp_feature_test.shape)

    # Saving all tensors so preprocessing only needs to be run once
    torch.save(vgg_gram_train, 'vgg_train.pt', pickle_protocol=4)
    torch.save(resnet_coarse_gram_train, 'resnet_coarse_train.pt', pickle_protocol=4)
    torch.save(resnet_fine_gram_train, 'resnet_fine_train.pt', pickle_protocol=4)
    torch.save(lbp_feature_train, 'lbp_train.pt', pickle_protocol=4)
    torch.save(complex_lbp_feature_train, 'complex_lbp_train.pt', pickle_protocol=4)
```

```
        torch.save(vgg_gram_test, 'vgg_test.pt', pickle_protocol=4)
        torch.save(resnet_coarse_gram_test, 'resnet_coarse_test.pt', pickle_protocol=4)
        torch.save(resnet_fine_gram_test, 'resnet_fine_test.pt', pickle_protocol=4)
        torch.save(lbp_feature_test, 'lbp_test.pt', pickle_protocol=4)
        torch.save(complex_lbp_feature_test, 'complex_lbp_test.pt', pickle_protocol=4)

        print("TENSORS SAVED")

    else:
        # Loading tensors from paths
        train_labels = torch.load('train_labels.pt')
        test_labels = torch.load('test_labels.pt')

        vgg_gram_train = torch.load('vgg_train.pt')
        resnet_coarse_gram_train = torch.load('resnet_coarse_train.pt')
        resnet_fine_gram_train = torch.load('resnet_fine_train.pt')
        lbp_feature_train = torch.load('lbp_train.pt')
        complex_lbp_feature_train = torch.load('complex_lbp_train.pt')

        vgg_gram_test = torch.load('vgg_test.pt')
        resnet_coarse_gram_test = torch.load('resnet_coarse_test.pt')
        resnet_fine_gram_test = torch.load('resnet_fine_test.pt')
        lbp_feature_test = torch.load('lbp_test.pt')
        complex_lbp_feature_test = torch.load('complex_lbp_test.pt')

        # Printing size of everything to make sure tensors loaded correctly
        print("Train Labels Size: ", train_labels.shape)
        print("Test Labels Size: ", test_labels.shape)

        print("VGG Gram Train Size: ", vgg_gram_train.shape)
        print("VGG Gram Test Size: ", vgg_gram_test.shape)

        print("ResNet Coarse Gram Train Size: ", resnet_coarse_gram_train.shape)
        print("ResNet Coarse Gram Test Size: ", resnet_coarse_gram_test.shape)

        print("ResNet Fine Gram Train Size: ", resnet_fine_gram_train.shape)
        print("ResNet Fine Gram Test Size: ", resnet_fine_gram_test.shape)

        print("LBP Feature Train Size: ", lbp_feature_train.shape)
        print("LBP Feature Test Size: ", lbp_feature_test.shape)

        print("Complex LBP Feature Train Size: ", complex_lbp_feature_train.shape)
```

```
    print("Complex LBP Feature Test Size: ", complex_lbp_feature_test.shape)


    # Normalizing input feature tensors
    scaler = StandardScaler()
    vgg_gram_train = scaler.fit_transform(vgg_gram_train)
    vgg_gram_test = scaler.transform(vgg_gram_test)
    resnet_coarse_gram_train = scaler.fit_transform(resnet_coarse_gram_train)
    resnet_coarse_gram_test = scaler.transform(resnet_coarse_gram_test)
    resnet_fine_gram_train = scaler.fit_transform(resnet_fine_gram_train)
    resnet_fine_gram_test = scaler.transform(resnet_fine_gram_test)

    # Setting batch size for training and feature descriptor type
    batch_size = 1024
    feature_type = "vgg" # "vgg", "resnet_coarse", "resnet_fine", "lbp"


    if(feature_type == "vgg"):
        classifier = train_svm(vgg_gram_train, train_labels, vgg_gram_test, test_labels, batch_size, feature_type)
        find_correct_and_incorrect_predictions(classifier, vgg_gram_test, test_images, test_labels)
    elif(feature_type == "resnet_coarse"):
        classifier = train_svm(resnet_coarse_gram_train, train_labels, resnet_coarse_gram_test, test_labels, batch_size, feature_type)
        find_correct_and_incorrect_predictions(classifier, resnet_coarse_gram_test, test_images, test_labels)
    elif(feature_type == "resnet_fine"):
        classifier = train_svm(resnet_fine_gram_train, train_labels, resnet_fine_gram_test, test_labels, batch_size, feature_type)
        find_correct_and_incorrect_predictions(classifier, resnet_fine_gram_test, test_images, test_labels)
    elif(feature_type == "lbp"):
        classifier_lbp = train_svm(lbp_feature_train, train_labels, lbp_feature_test, test_labels, batch_size, feature_type)
        classifier_complex_lbp = train_svm(complex_lbp_feature_train, train_labels, complex_lbp_feature_test, test_labels, batch_size, feature_type)

        find_correct_and_incorrect_predictions(classifier_lbp, lbp_feature_test, test_lbp_images, test_labels)

    else:
        print("Invalid feature type, please enter 'vgg', 'resnet_coarse', 'resnet_fine', or 'lbp'")

```

```
```

Listing 1: Python Code