Theory question

I propose a hybrid approach that combines elements from Gabor Filters, Gram Matrix, and Channel Normalization (CN) with a multiscale pyramid representation. The algorithm begins by processing the image through a pyramid structure, where each level captures texture information at different resolutions. The final layer uses CN to normalize and enhance the texture features across channels, which ensures consistency across different lighting conditions. This detector can perform well on complex textures, such as natural scenes and patterns with intricate detail.

LBP implementation

To implement the LBP algorithm, RGB images were first converted to their respective HIS space, and the hue channel was extracted. The hue channel was used for LBP calculation and feature extraction. To calculate the LBP for each pixel in the Hue image, a circular neighborhood of P points with radius R was defined. The intensity of the center pixels was compared to its neighbors and assigned binary values of 0 and 1 if the neighbors is less or greater than the center pixel respectively. The binary pattern generated was rotated to find the minimum value and the histogram plotted for different images in different classes as shown in Figure 1-4. The image below shows excerpt of this implementation

```python
def rgb_to_hsi_pixel(r, g, b):
    # Normalize R, G, B to the range [0, 1]
    R = r / 255.0
    G = g / 255.0
    B = b / 255.0

    # Calculate M, m, and c
    M = max(R, G, B)
    m = min(R, G, B)
    c = M - m

    # Calculate Intensity (I)
    I = (R + G + B) / 3

    # Calculate Saturation (S)
    if I == 0:
        S = 0.0
    else:
        S = 1 - (m / I)

    # Calculate Hue (H)
    if c == 0:
        H = 0.0
    elif M == R:
        H = (60 * ((G - B) / c) + 360) % 360
    elif M == G:
        H = (60 * ((B - R) / c) + 120) % 360
    elif M == B:
        H = (60 * ((R - G) / c) + 240) % 360
    # # Normalize H to the range [0, 1] before returning
    H = H / 360.0
```

```python
def rgb_image_to_hsi(image):
    # Ensure the image is in RGB format (OpenCV loads images in BGR format by default)
    if image.shape[-1] == 3:  # Check if it's a color image with 3 channels
        image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    else:
        raise ValueError("The provided image does not have 3 channels (RGB).")

    # Prepare an empty array for HSI with 3 channels (H, S, I)
    hsi_array = np.zeros_like(image_rgb, dtype=float)

    # Iterate over each pixel to convert it to HSI
    for i in range(image_rgb.shape[0]):
        for j in range(image_rgb.shape[1]):
            r, g, b = image_rgb[i, j] / 255.0  # Normalize R, G, B to [0, 1]
            h, s, i_intensity = rgb_to_hsi_pixel(r, g, b)
            # Store the scaled H, S, and I values
            hsi_array[i, j] = [h * 255, s * 255, i_intensity * 255]

    # Extract the Hue channel and scale it to [0, 255]
    hue_channel = hsi_array[:, :, 0]

    # Convert the Hue channel to an 8-bit grayscale image (0-255 range)
    hue_image = Image.fromarray(hue_channel.astype('uint8'), 'L')

    return hue_image
```

```python
def calculate_lbp(images, R=1, P=8):
    lbp_hists=[]
    for image in images:

        # Extract hue value
        image=rgb_image_to_hsi(image)

        # Initialize LBP histogram
        lbp_hist = {t: 0 for t in range(P + 2)}

        image = image.resize((64, 64), Image.LANCZOS)
        image=np.array(image)
        width=image.shape[1]
        height=image.shape[0]
        # Constants
        # lbp = [[0 for _ in range(height)] for _ in range(width)]
        rowmax, colmax = height - R, width - R

        # Loop through image pixels
        for i in range(R, rowmax):
            for j in range(R, colmax):
                # print(f"npixel at ({i},{j}):")
                pattern = []

                # Generate pattern for the current pixel using P points
                for p in range(P):
                    # Calculate offsets for circular neighborhood
                    angle = 2 * math.pi * p / P
                    del_k = R * math.cos(angle)
                    del_l = R * math.sin(angle)

                    # Handle very small values close to zero for better stability
                    if abs(del_k) < 0.001: del_k = 0.0
                    if abs(del_l) < 0.001: del_l = 0.0

                    # Calculate neighboring pixel coordinates
                    k = i + del_k
                    l = j + del_l
                    k_base, l_base = int(k), int(l)

                    # Calculate interpolation values
                    delta_k = k - k_base
                    delta_l = l - l_base

                    # Fetch image values and compute the interpolated value at (k, l)
                    image_val_at_p = 0
                    if delta_k < 0.001 and delta_l < 0.001:
```

```python
                        image_val_at_p = float(image[k_base][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]
                    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]
                    else:
                        # Bilinear interpolation for fractional (k, l)
                        image_val_at_p = (
                            (1 - delta_k) * (1 - delta_l) * image[k_base][l_base] +
                            delta_k * (1 - delta_l) * image[k_base + 1][l_base] +
                            (1 - delta_k) * delta_l * image[k_base][l_base + 1] +
                            delta_k * delta_l * image[k_base + 1][l_base + 1]
                        )

                    # Append binary pattern based on comparison with center pixel value
                    pattern.append(1 if image_val_at_p >= image[i][j] else 0)

                # print(f"pattern: {pattern}")

                # Convert pattern to BitVector and compute the minimal bit rotation
                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)

                # print(f"minbv: {minbv}")

                # Calculate runs of consecutive bits in the minimal rotation
                bvruns = minbv.runs()
                encoding = None

                # Determine encoding based on the number and pattern of runs
                if len(bvruns) > 2:
                    lbp_hist[P + 1] += 1
                    encoding = P + 1
                elif len(bvruns) == 1 and bvruns[0][0] == '1':
                    lbp_hist[P] += 1
                    encoding = P
                elif len(bvruns) == 1 and bvruns[0][0] == '0':
                    lbp_hist[0] += 1
                    encoding = 0
                else:
                    lbp_hist[len(bvruns[1])] += 1
                    encoding = len(bvruns[1])
    lbp_hists.append(lbp_hist)            # print(f"encoding: {encoding}")

    return lbp_hists
```

```python
# Plot the histogram

def plot_hist(lbp_hists,img_names):
    for i,lbp_hist in enumerate(lbp_hists):
        plt.figure(figsize=(8, 5))
        plt.bar(list(lbp_hist.keys()), lbp_hist.values(), color='g')

        # Labeling the axes and title
        plt.xlabel('Index')
        plt.ylabel('Frequency')
        plt.title(f'Histogram of LBP Patterns for {img_names[i]}')
        plt.savefig(f'lpb_histogram_{img_names[i]}')
```

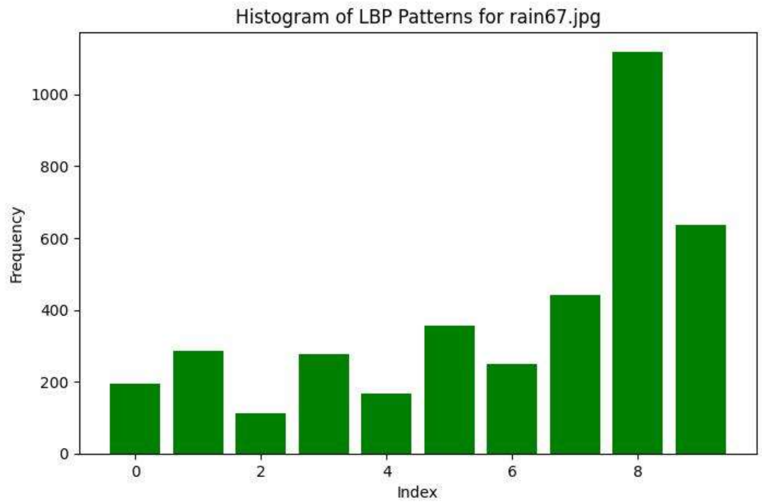

Figure 1. LBP histogram for rain class

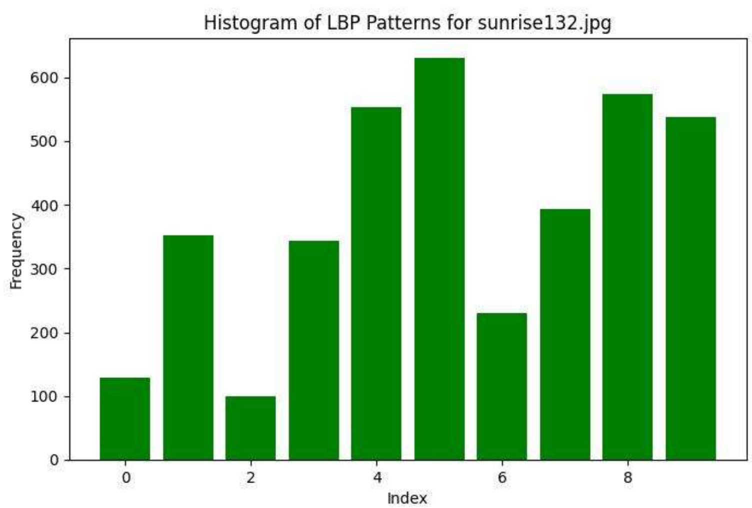

Figure 2. LBP histogram for sunrise class

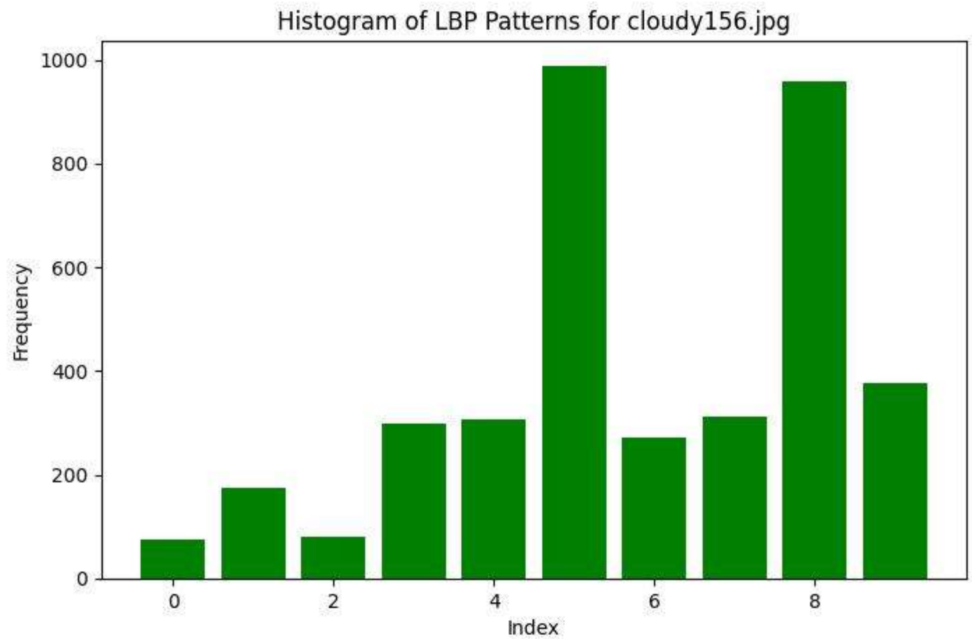

Figure 3. LBP histogram for cloudy class

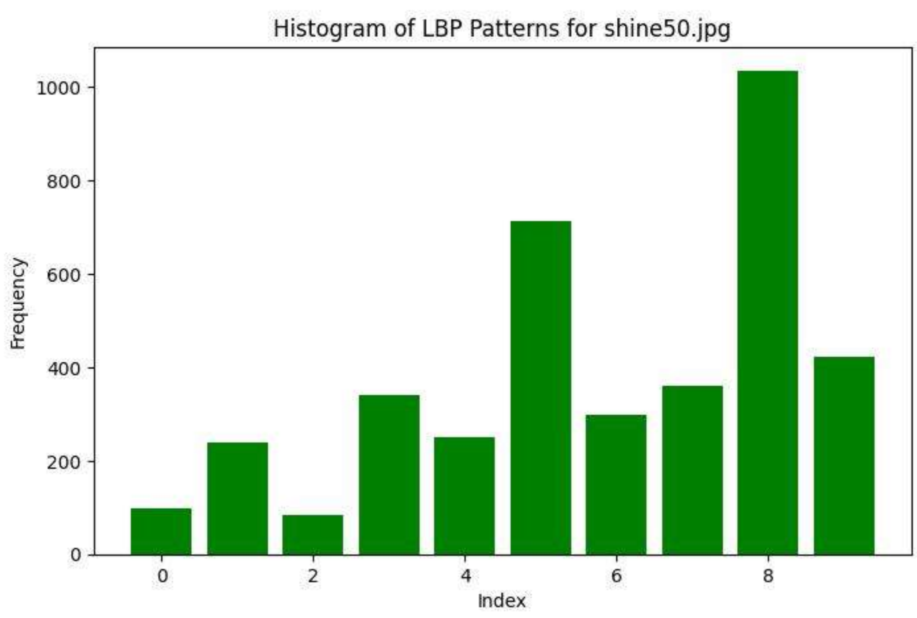

Figure 4. LBP histogram for shine class

Image preparation, gram matrix, feature extraction

The hue images were loaded based on their filenames and feature extraction was performed using VGG19, coarse and fine resnet50. The images were resized to 265 by 256and the extracted features were saved to an .npz file. The get_gm function was used to generate a Gaussian Mixtures from the extracted features. For each feature vector, it calculated the Gram matrix, flattens it, and samples 1024 elements randomly. These sampled elements form the Gaussian Mixture representation of the feature. The code snippet to perform that can be found below

```python
def get_images(dir, train=True):
    images = []
    labels = []

    # Define the directory path based on the mode (training or testing).
    data_dir = os.path.join(dir, "training" if train else "testing")

    # Iterate over sorted image files in the specified directory.
    for img_name in sorted(os.listdir(data_dir)):
        # Skip hidden files like .DS_Store (MacOS metadata files).
        if img_name.startswith('.'):
            continue

        # Identify the label based on the filename.
        label = next((classes.index(cls) for cls in classes if cls in img_name), -1)

        # Proceed only if a valid label was found.
        if label != -1:
            # Read the image using cv2.
            img_path = os.path.join(data_dir, img_name)
            img=cv2.imread(img_path)

            # Only add images with three channels (color images).
            if img is not None and img.shape[-1] == 3:
                labels.append(label)
                images.append(img)
    # print(labels)
    return labels, images


def get_features(model, images, labels, mode='train', modelname='vgg', config=None):
    features=[]
    for image in images:
        img=transform.resize(image,(256,256))
        if modelname=='resnet' and config=='coarse':
            feature,_=model(img)
        elif modelname=='resnet' and config=='fine':
            _, feature=model(img)
        else:
            feature=model(img)
        features.append(feature)
    np.savez(f'{modelname}_{mode}_{config}_feature.npz',labels=labels, features=features)
```

```python
def get_gm(features):
    gm_train = []
    np.random.seed(0)  # Set seed once
    for ft in features:
        ft = ft.reshape(512, -1)
        gm = ft @ ft.T
        gm_flat = gm.flatten()

        # Check if enough elements for sampling
        if len(gm_flat) < 1024:
            raise ValueError("Gram matrix is too small for sampling 1024 elements.")

        gm_sample = np.random.choice(gm_flat, 1024, replace=False)
        gm_train.append(gm_sample)

    return gm_train
```

The gram matrix was implemented and plotted for different features descriptors (vgg, coarse resnet50 and fine resnet50). The results can be seen in Figure 5 -7. The visualization code can be seen below.

```python
def plot_gram_matrix(model, gm_train, training_labels, classes, P, R):
    # Identify one sample index for each class
    sample_indices = []
    for class_name in classes:
        for i, label in enumerate(training_labels):
            if classes[label] == class_name:
                sample_indices.append(i)
                break  # Stop after finding the first match for each class
    # Plot Gram matrices for each identified sample
    for idx, class_name in zip(sample_indices, classes):
        plt.figure()
        gm_2D = gm_train[idx].reshape(32, 32)
        plt.imshow(gm_2D, cmap='viridis')  # Plot Gram matrix of the specific image
        plt.colorbar()
        plt.title(f"{class_name} - 2D Gram Matrix Heatmap ({model}_R={R}_P={P})")

        # Save each plot with a unique filename
        plt.savefig(f'{class_name}_2D_Gram_Matrix_Heatmap_{model}_R={R}_P={P}.png')
        plt.show()
```

VGG19

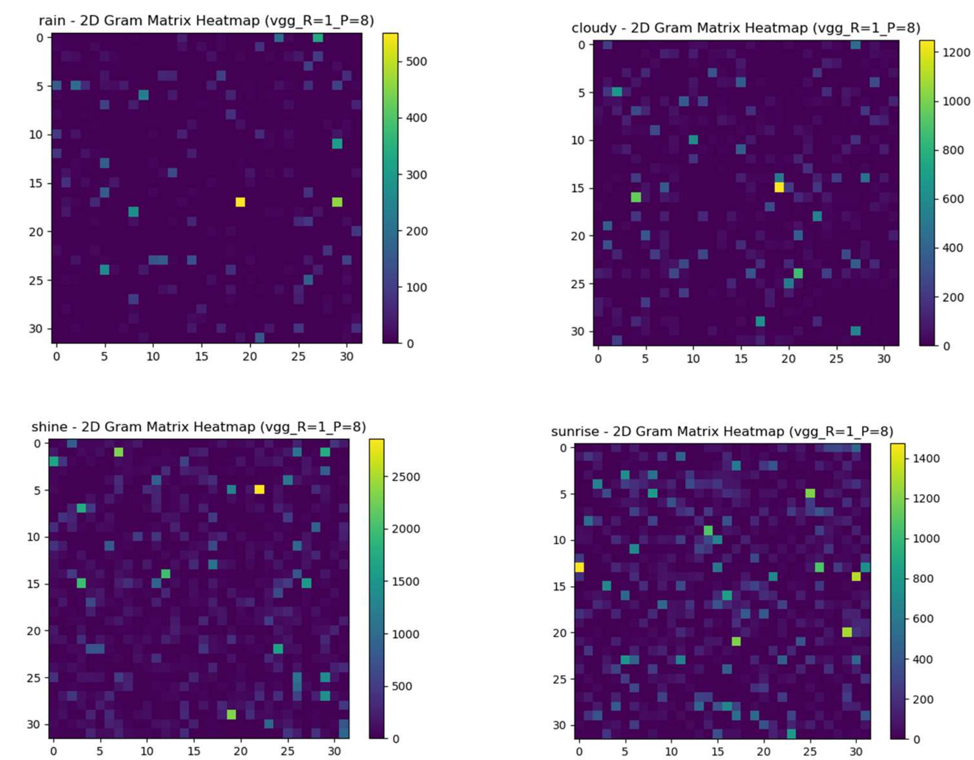

Figure 5. Gram matrix plot for VGG19

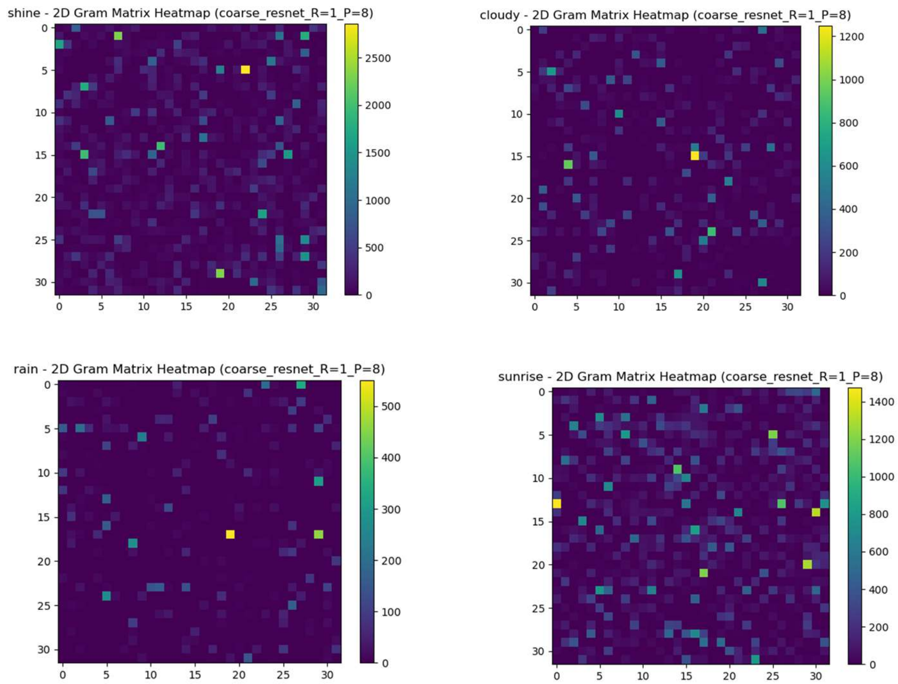

Figure 6. Gram matrix plot for coarse resnet50

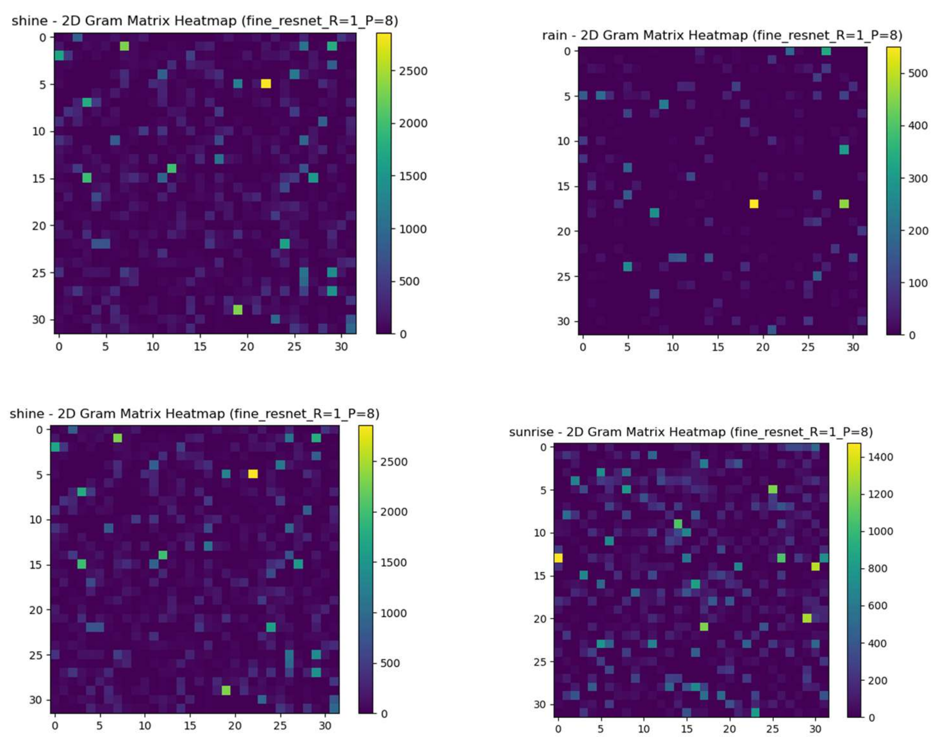

Figure 7. Gram matrix plot for fine resnet50

Confusion matrix

The confusion matrix for the results of the feature descriptor after training using SVM can be seen between Figures 8 - 10. The code snippet can be found below.

```
def plot_confusion_matrix(model, test_labels, pred_labels, R, P):
    # Calculate the confusion matrix
    cm = confusion_matrix(test_labels, pred_labels)

    # Plot the confusion matrix
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt="d")
    plt.xlabel("Predicted Label")
    plt.ylabel("True Label")
    plt.title(f"Confusion Matrix ({model}) R={R}, P={P}")
    plt.savefig(f'confusion_matrix ({model}) R={R}, P={P}.png')
```

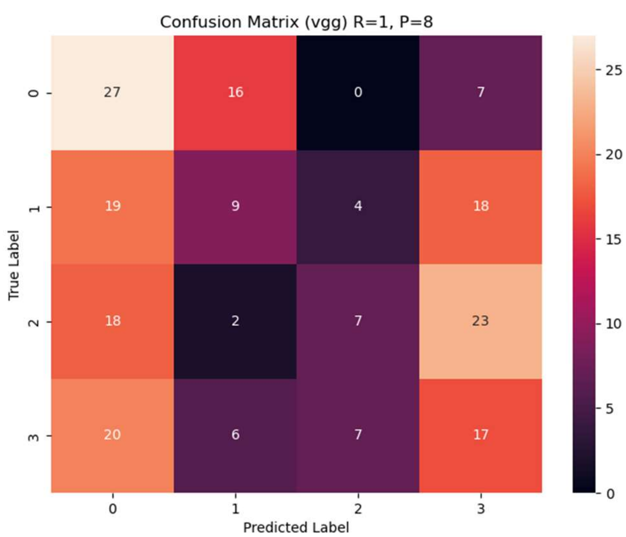

Figure 8. Confusion matrix for VGG19

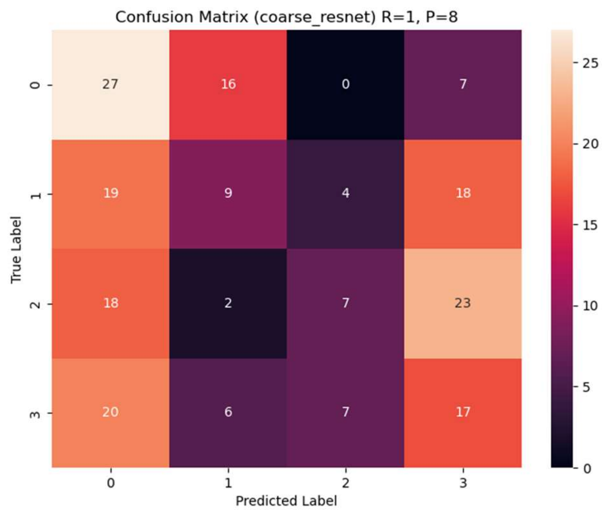

Figure 9. Confusion matrix for coarse resnet50

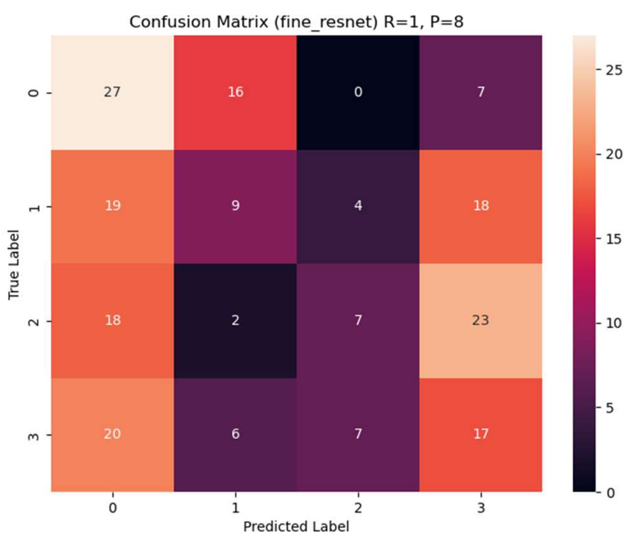

Figure 10. Confusion matrix for coarse resnet50

Correct and incorrect classification

The classification accuracy of the models are presented in Table 1 - 3. The correct and incorrect classification for each class for feature descriptor can be found in Figures 11 – 16. P=8 and R=1 WAS used because it had the best results for all descriptor. the code snippet to plot the graph can be seen below

```python
# Define a function to save images of correctly and incorrectly classified samples
def save_classification_images(model, test_images, test_labels, pred_labels, classes, P, R):
    # Initialize dictionaries to track saved images for each class
    saved_correct = {class_name: False for class_name in classes}
    saved_incorrect = {class_name: False for class_name in classes}

    for idx, (image, true_label, pred_label) in enumerate(zip(test_images, test_labels, pred_labels)):
        true_class = classes[true_label]
        pred_class = classes[pred_label]

        # Check if it's correctly or incorrectly classified and save accordingly
        if true_class == pred_class and not saved_correct[true_class]:  # Correctly classified
            plt.figure()
            plt.imshow(image)  # Assuming image is in RGB format
            plt.title(f"Correctly Classified: {true_class}")
            plt.xlabel(f"Predicted: {pred_class} | Ground Truth: {true_class}")
            plt.savefig(f"{model}_correct_{true_class}_P={P}_R={R}.png")
            plt.close()
            saved_correct[true_class] = True  # Mark as saved

        elif true_class != pred_class and not saved_incorrect[true_class]:  # Misclassified
            plt.figure()
            plt.imshow(image)  # Assuming image is in RGB format
            plt.title(f"Misclassified: {true_class} as {pred_class}")
            plt.xlabel(f"Predicted: {pred_class} | Ground Truth: {true_class}")
            plt.savefig(f"{model}_incorrect_{true_class}_P={P}_R={R}.png")
            plt.close()
            saved_incorrect[true_class] = True  # Mark as saved

        # Break the loop once we have both correct and incorrect for each class
        if all(saved_correct.values()) and all(saved_incorrect.values()):
            break
```

Table 1. Parameter selection VGG19

	Classification accuracy (%)		
	P		
R	8	18	24
1	30	28	28
2	16	16	16

Table 2. Parameter selection coarse resnet50

	Classification accuracy (%)		
	P		
R	8	18	24
1	30	30	18
2	17	18	22

Table 3. Parameter selection fine resnet50

	Classification accuracy (%)		
	P		
R	8	18	24
1	30	16	17

2	29	25	18

VGG19 - Correct

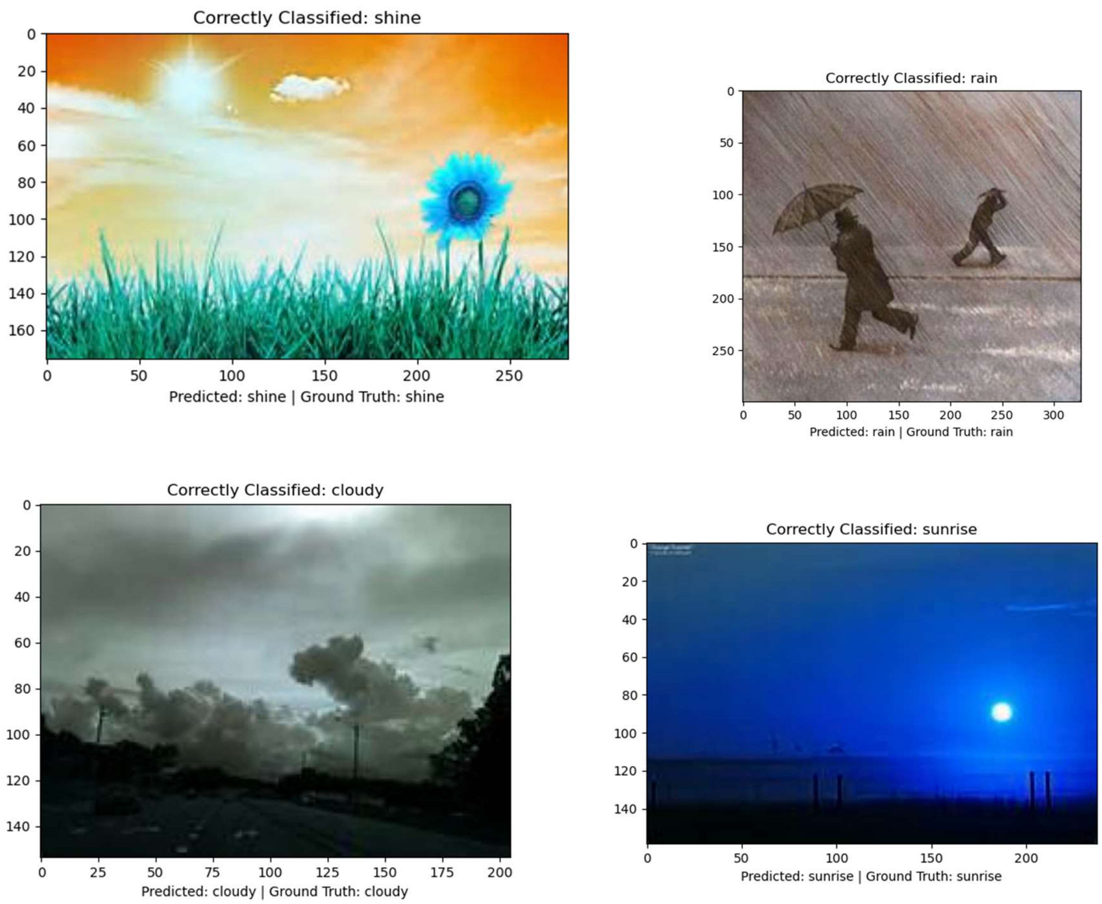

Figure 11. Correct predictions from VGG19

VGG19 - Incorrect

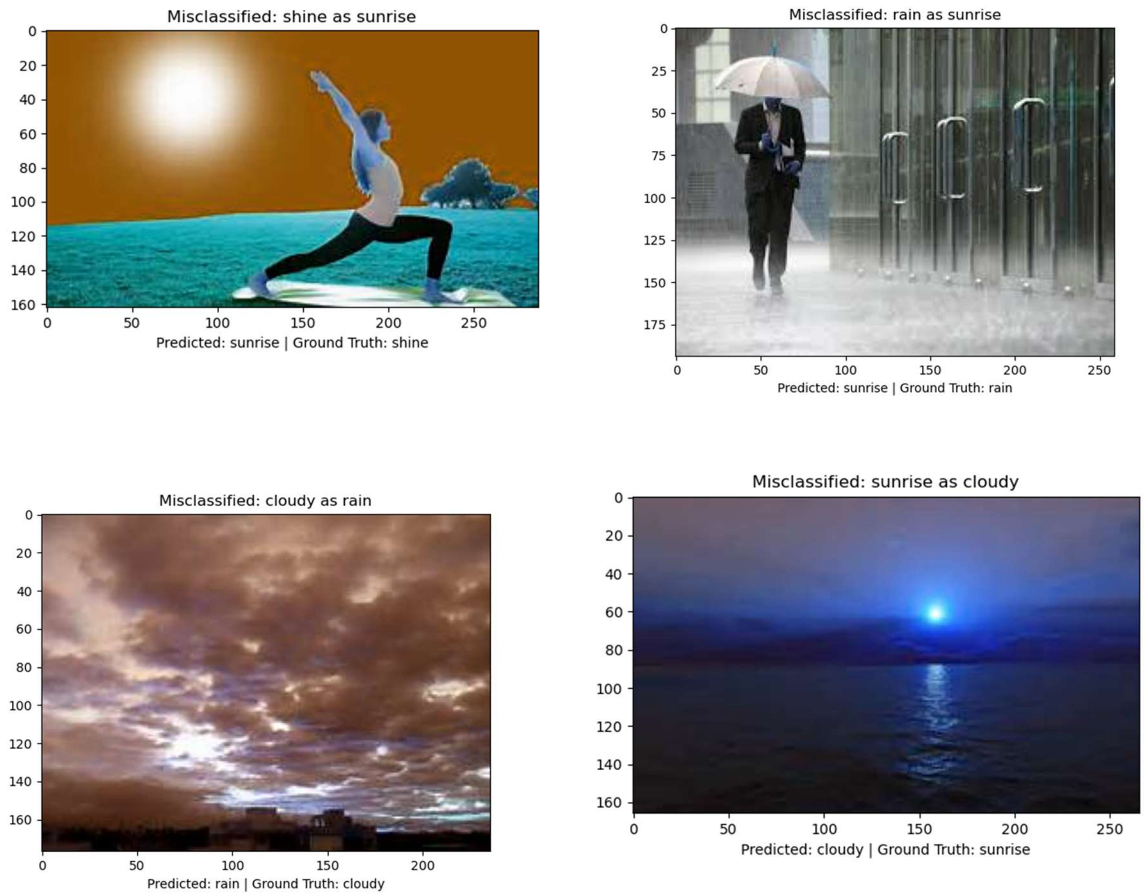

Figure 12. Incorrect predictions from VGG19

Coarse resnet50 - Correct

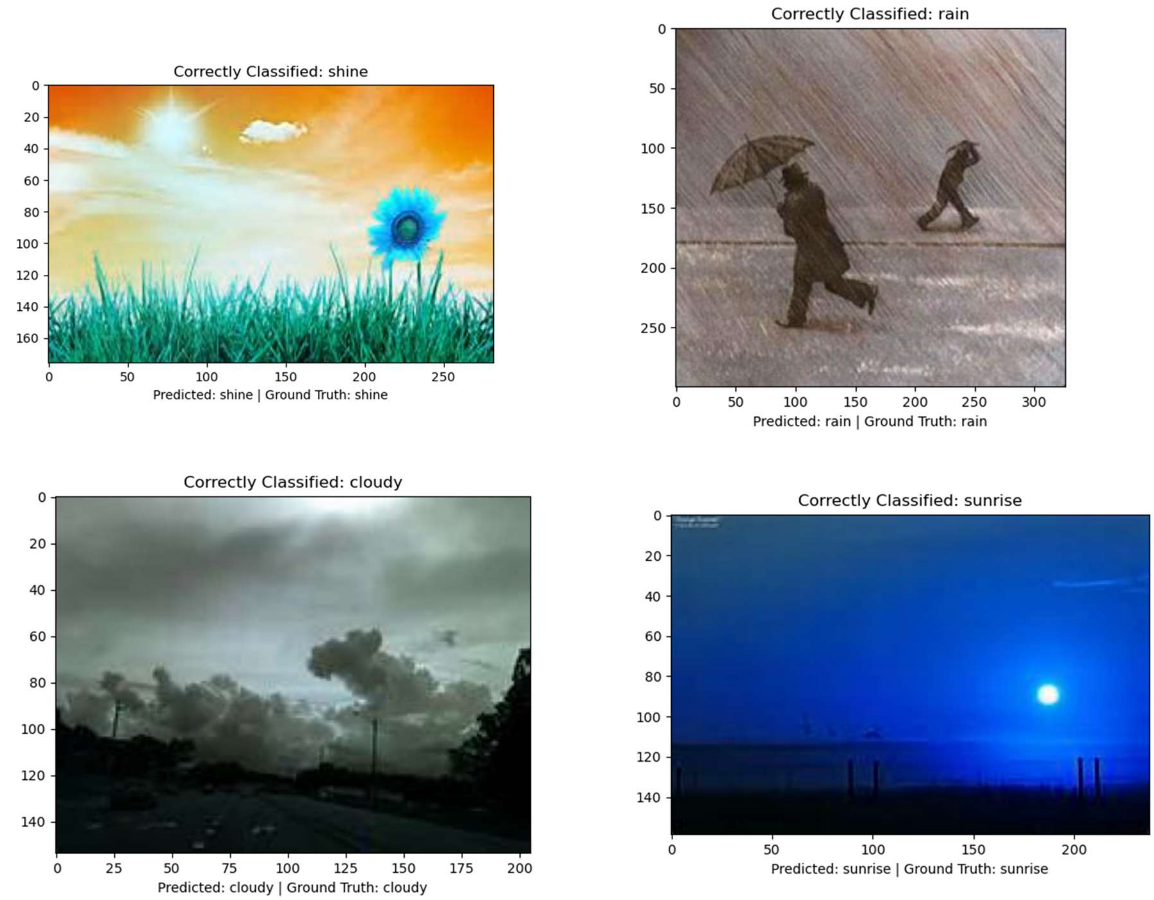

Figure 13. Correct predictions from coarse resnet50

Coarse resnet50 - Incorrect

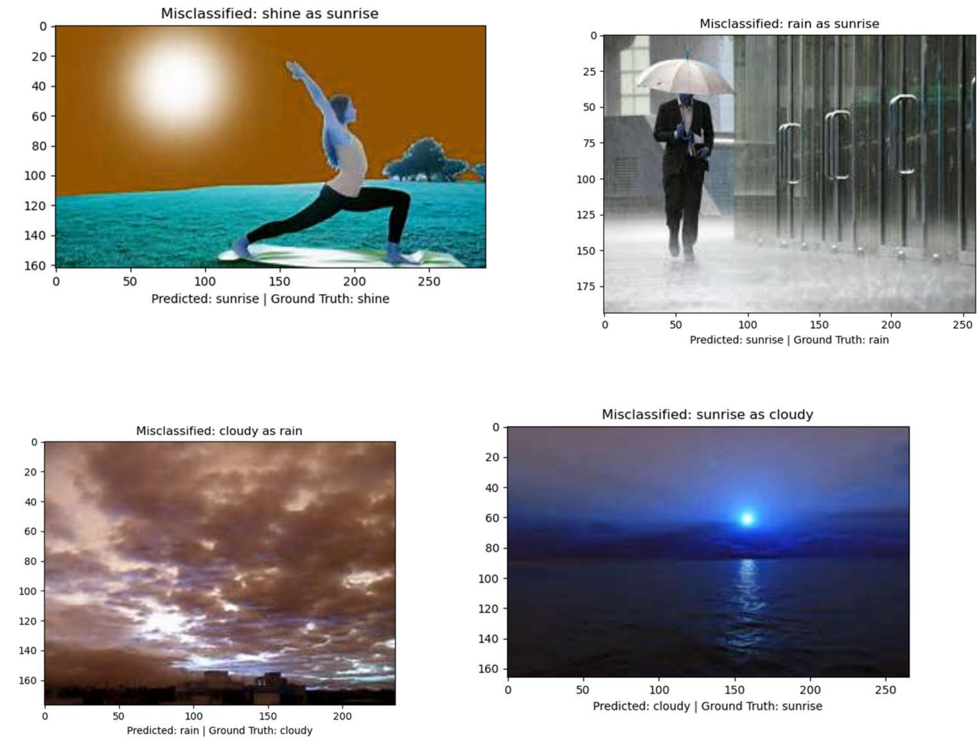

Figure 14. Incorrect predictions from coarse resnet50

Fine resnet50 - Correct

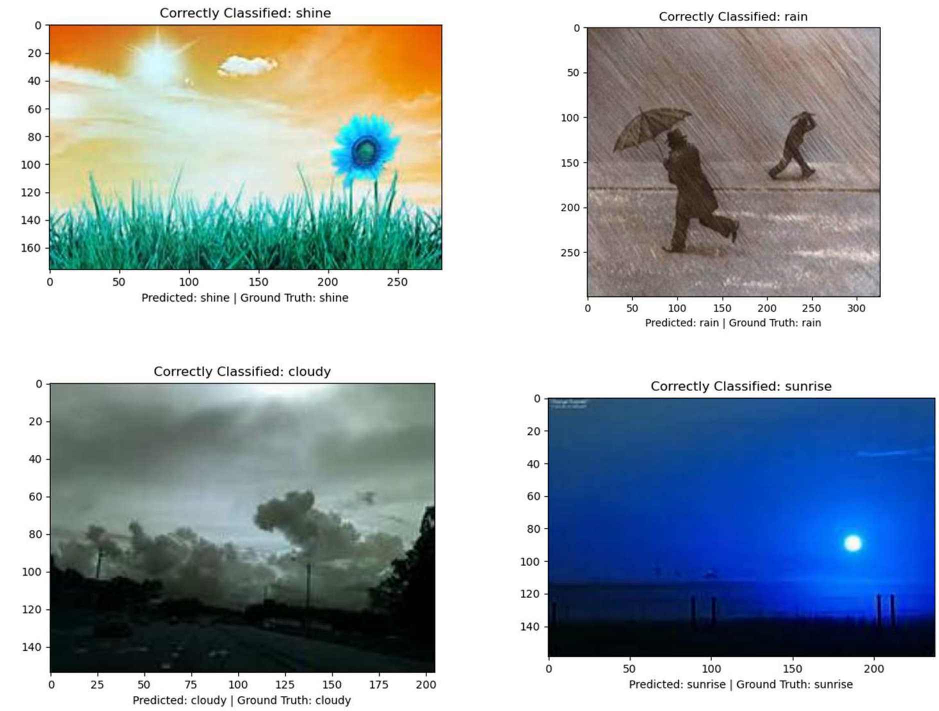

Figure 15. Correct predictions from fine resnet50

Fine resnet50 - Incorrect

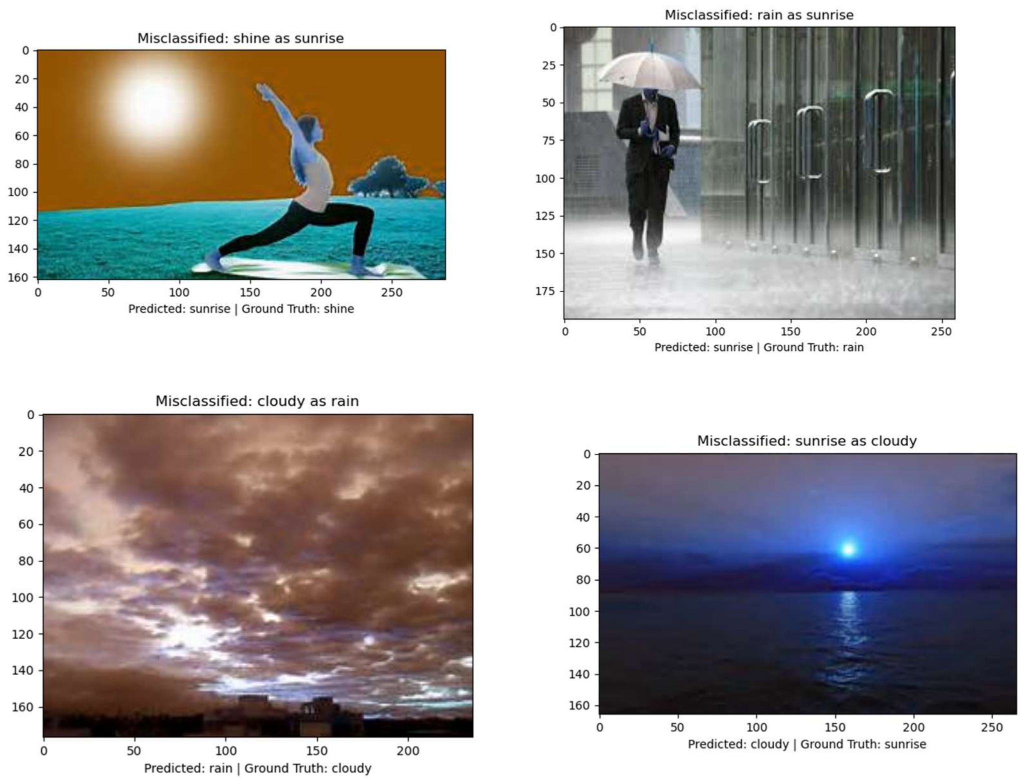

Figure 15. Incorrect predictions from fine resnet50

Notes:

- I believe there are some bugs in my code which I could not resolve before the deadline of the assignment. I tried to ensure the use of different values of P and R to improve result to no avail. All descriptors had the best classification accuracy of 30%.
- For the gram's matrices, it shows that there are little to no feature interactions or textures different spatial locations