# MWDB Project Report: Phase 1

Extracting and learning about feature descriptors

Prepared by: Akshay Malhotra

## Team Members:

Akshay Malhotra
Shivansh Mittal
Siddhant Srivastava
Aniket Devle
Kevin Shah
Augustus Crosby

## Abstract

As the world moves towards automation in order to focus its attention on other tasks, processing multimedia objects quickly and efficiently has grown in importance. For instance, security surveillance and driverless car testing are two such applications where latency should be low and the capacity to tolerate false negatives is low. Therefore, expressing multimedia objects without redundancy with the help of the right data model is extremely important.

In this phase of the project, we experiment and learn about different ways (feature descriptors) to express a grayscale image and its significance. We also compare these feature descriptors (individually and combined) with those of other images using appropriate distance measures in order to get similar images. In this phase, we experimented with the following feature descriptors:

1. Color Moments
2. Extended Local Binary Patterns (ELBP)
3. Histogram of Oriented Gradients (HOG)

### Keywords

Color Moments (Mean, Standard Deviation, Skewness)
Extended Local Binary Patterns (ELBP)
Histogram of Oriented Gradients (HOG)
p-norm distance

# Introduction

## Terminology

Color Moments
This is a feature descriptor that can be computed over an image and helps understand the color distribution of the image. There are many parts to color moments, however, for this phase of the project we will focus on the lower-order components of color moments: mean, standard deviation, and skewness.

Extended Local Binary Patterns
The extended local binary pattern is an extension of the existing LBP algorithm that is used for texture classification with the addition of a rotational invariant variance measure. This feature descriptor works well for spatial features.
The input to an LBP algorithm is an image and the output is an LBP matrix of the same dimensions of the image. These values are calculated for each pixel in the original image by comparing its value to those of its surrounding neighbors (8 neighbors if radius=1). For each surrounding pixel, if the value is greater or equal than the center pixel in question (for which value is being calculated), 1 is allocated otherwise 0 is. This forms a binary code for each pixel in the image. These binary values are then binned into *n* number of bins where the *n-1* bin signifies non-uniform texture code.

Histogram of Oriented Gradients
This is a feature descriptor that is used to understand the change in gradients over the image and is popularly used for edge detection. In this algorithm, a sliding window of size *nxn* is taken into consideration. For each element in the sliding window, gradient orientation and values are calculated and then binned. The gradient value of a pixel is calculated by taking into account the absolute value of the difference of horizontal and vertical pixels and finding the distance between them. Gradient orientation is calculated by taking the $\tan^{-1}$ of the absolute value of the difference of horizontal and vertical pixels.

cwd
Current working directory. Any reference of cwd refers to the current directory from which one operates.

## Goal Description

The goal of this phase is to implement multiple feature descriptors over grayscale images in order to understand their significance and performance. These feature descriptors are then compared to each other using different distance measures, thereby, helping achieve a better understanding of different distance measures. Broadly speaking, this phase of the project has 5 tasks:

Task 0: This task deals with downloading the Olivetti faces dataset.
This dataset contains 400 grayscale images for 10 different subjects. Therefore, there are 10 unique grayscale images for each subject. Each greyscale image of the subject was taken at different times (between 1992-1994), with varying lighting conditions, facial expressions, and facial details. Moreover, all the 400 images are of equal dimensions, ie, 64x64.

Task 1: Given a grayscale image and data model (local binary patterns, and histogram of oriented gradients), a feature descriptor has to be derived corresponding to that image. This feature descriptor then has to be saved.

Task 2: Given a folder of grayscale images, feature descriptors for all the images in the folder have to be derived. These feature descriptor then has to be saved.

Task 3: Given a folder of grayscale images, image_id, data model, and *k*, *k* most similar images have to be derived from the folder of images using the given data model. These *k* images then have to be saved.

Task 4: Given a folder of grayscale images, image_id, and *k*, *k* most similar images have to be derived from the folder of images using all data models. These images then have to be saved.

## Assumptions

The following assumptions are made in order to accomplish this phase of the project:

1. The test dataset will contain grayscale images.
2. The test dataset will contain images of the dimensions 64x64.
3. Development can be done in any language and using any environment.
4. Default parameters provided in the assignment can be used. If any other values are used, the thought process behind using different values has to be described.
5. Libraries built around feature descriptors and distance measures can be used as long as their inner-working is understood.

# Description of the proposed solution/implementation

## Task 0

Given:

- No input.

Output:

- The Olivetti faces dataset using sklearn and stored in a directory.

1. Fetch dataset using sklearn.datasets, store in the *data* variable.
   Obtained using the fetch_olivetti_faces function. This dataset has 400 greyscale images.

```
# Fetching data from sklearn

data = fetch_olivetti_faces(data_home=None, shuffle=False, random_state=0, download_if_missing=True)
print ("Type of data: " + str(type(data)))
```

```
Type of data: <class 'sklearn.utils.Bunch'>
```

2. Images and targets are stored in different variables for easier access.

```
images = data.images
target = data.target
print (images[0])
```

```
[[0.30991736 0.3677686  0.41735536 ... 0.37190083 0.3305785  0.30578512]
 [0.3429752  0.40495867 0.43801653 ... 0.37190083 0.338843   0.3140496 ]
 [0.3429752  0.41735536 0.45041323 ... 0.38016528 0.338843   0.29752067]
 ...
 [0.21487603 0.20661157 0.2231405  ... 0.15289256 0.16528925 0.17355372]
 [0.20247933 0.2107438  0.2107438  ... 0.14876033 0.16115703 0.16528925]
 [0.20247933 0.20661157 0.20247933 ... 0.15289256 0.16115703 0.1570248 ]]
```

3. Store all the images in ".tif" format.
   Images are stored in tiff format as pixel intensities are float-based, storing them in any other format would lead to lossy images unless converted in the range of [0,256].
   Iterate through *images* and create a file name based on the subject and imageId of the

subject. Format of naming files: <subjectId_imageId>.tif

```
# Saving images in .tif format

file_names = []
prev_subject = None
subject_image = 0

for index in range(0,images.shape[0]):
    subject = target[index]
    if (prev_subject != None and subject != prev_subject):
        subject_image = 0
    file_name = str(subject) + "_" + str(subject_image) + ".tif"
    file_names.append(file_name)
    file_path = WRITE_DIR_IMG / file_name

    im = Image.fromarray(images[index])
    im.save(file_path)

    subject_image += 1
    prev_subject = subject
```

```
# Saving target in .npy file

file_path = WRITE_DIR_TAR / "target"
filename_target = np.array([file_names,target])
np.save(str(file_path), filename_target, )
```

## Task 1

Given:

- *IMAGE_ID*: ImageId for which feature descriptors have to be calculated
- *MODEL*: Data model to be used to calculating feature descriptors

Output:

- Feature descriptor of *IMAGE_ID* is calculated based on *MODEL* and stored in a file.

**About Utils.py**

All functions used for calculating and storing feature descriptors are stored in Utils.py. These functions will be used in future tasks to calculate feature descriptors.

Some important functions are mentioned below:

- create_windows(): This function used to create windows of an image.

```python
def create_windows(image=None):
    # Exceptions
    if image is None:
        raise Exception("create_windows: image input param missing")

    # Creating windows over the image
    windows = []
    step = 8
    for x in range(0, image.shape[0], step):
        for y in range(0, image.shape[1], step):
            window = image[x:x + step, y:y + step]
            windows.append(window)
    return np.array(windows)
```

- calculate_color_moments: This function is used to generate color moments in the order [mean, standard deviation and skewness] given an array of windows. For each window, color moments are calculated and appended to each other. Libraries Used: numpy, scipy.

```python
def calculate_color_moments(windows=None):
    # Exceptions
    if windows is None:
        raise Exception('calculate_color_moments: windows input param missing')

    # Calculating color moments over windows
    means = []
    stds = []
    skews = []
    for index in range(0, len(windows)):
        # Extracting window
        window = windows[index]
        flatten_window = window.flatten()

        # Calculating mean, std deviation and skewness for window
        mean = np.mean(window)
        std = np.std(window)
        skew = scipy.stats.skew(flatten_window)

        # Appending results of each window to a list
        means.append(mean)
        stds.append(std)
        skews.append(skew)

    color_moments = []
    color_moments.extend(means)
    color_moments.extend(stds)
    color_moments.extend(skews)
    return color_moments
```

- calculate_elbp(): This function is used to calculate ELBP of an image. It also performs bining of the results of ELBP is calculated. Libraries Used: skimage, numpy.

```
def calculate_elbp(image=None, points=None, radius=None, method=None):
    if image is None or points is None or radius is None or method is None:
        raise Exception("calculate_elbp: image/points/radius/method input param missing")

    elbp = local_binary_pattern(image, P=points, R=radius, method=method)

    # Bining the results of ELBP
    bins = 2 ** points
    bining_results = np.histogram(elbp, bins=bins, range=(0.0, float(bins)))

    return bins, bining_results
```

- calculate_hog(): This function is used to calculate the HOG of an image. Libraries used: skimage

```
def calculate_hog(image=None, orientations=None, pixels_per_cell=None, cells_per_block=None):
    if image is None or orientations is None or pixels_per_cell is None or cells_per_block is None:
        raise Exception("calculate_hog: image/orientationspixels_per_cell/pixels_per_cell/cells_per_block input param "
                        "missing")

    fd, hog_image = hog(image,
                        orientations=orientations,
                        pixels_per_cell=pixels_per_cell,
                        cells_per_block=cells_per_block,
                        visualize=True)

    return fd, hog_image
```

If *MODEL*=”cm8x8”, then:

1. Create windows of size 8x8 and store them into *windows*. This can be done using the create_windows() method in Utils.py
2. Calculate color moments of *IMAGE_ID* and store them to *color_moments*.
   This can be done using the calculate_color_moments() method in Utils.py
   Color moments of results will have dimensions (1,192).

```
# Creation on 64 windows (8x8) from original image
windows = Utils.create_windows(np_image)

# Calculating mean, std. deviation and skewness for each 8x8 window
color_moments = np.array(Utils.calculate_color_moments(windows))
color_moments_reshaped = np.reshape(color_moments, (3,64))
```

3. Plot and store result onto disk.

```
for index in range(0,len(windows)):
    # Extracting window
    window = windows[index]

    # Plotting results in a figure
    ax[index].imshow(window, cmap='gray')
    ax[index].set_xticks([])
    ax[index].set_yticks([])
    ax[index].set_title("Mean: {:.2f}\nStd: {:.2f}\nSkew: {:.2f}".format(colo
    plt.subplots_adjust(left=None, bottom=None, right=None, top=None, wspace=

# Write path for color moments
color_moments_write_path = WRITE_DIR / "color_moments.csv"
color_moments_img_write_path = WRITE_DIR / "color_moments.png"

# Writing results into a file
Utils.write_color_moments(data=color_moments, path=color_moments_write_path)
fig.savefig(color_moments_img_write_path)
```

If *MODEL*=”elbp”, then:

1. Calculate ELBP of an image using the calculate_elbp() function in Utils.py, store it into *bins* and *bining_results*

   Parameter: Radius = 1. This parameter helps us change the neighborhood of pixels that can be taken into consideration. With the value of 1, immediate pixels are picked up for calculating LBP.
   Parameter: Points = 8. This parameter helps us change the number of pixels that should be taken into consideration (taken along the circumference). This helps us take into account the number of points that will help determine the texture. With radius=1, 8 pixels are taken into consideration for calculating the LBP value of a pixel, ie, 3 pixels above and below and 1 pixel on the side.
   Parameter: method: “ror”. This parameter helps us tune the manner in which LBP is calculated. By selecting the value of “ror”, LBP is calculated by taking rotation invariance into account.

   We bin/quantize the results to obtain a feature descriptor of equal sizes. The number of bins can be calculated by taking into account the number of points that were used to calculate LBP. As the number of points increases, the number of bits the represent the LBP value also increases. Therefore, the number of bins is equal to $2^{points}$ and we segregate the LBP values according to these bins.

```
# Calculation of ELBP for the image of interest
points = 8
radius = 1
method = 'ror'
elbp = local_binary_pattern(image, P=points, R=radius, method=method)
bins, bining_results = Utils.calculate_elbp(image=np_image, points=points, radius=radius, method=method)
```

2. Plot and store ELBP results (data in *bins*).

```
# Plotting bining results of ELBP

plt.figure(figsize=(15,7))
plt.bar(bining_results[1][1:], bining_results[0] ,width=2)
plt.xticks(np.arange(0, 270, 10))
plt.xlabel("ELBP Values")
plt.ylabel("Frequency")
plt.title("Plotting ELBP Feature Descriptor")

elbp_img_write_path = WRITE_DIR / "elbp.png"
plt.gcf().savefig(elbp_img_write_path)

# Write path for ELBP
elbp_write_path = WRITE_DIR / "elbp.csv"
# Writing results into a file# Writing results into a file
Utils.write_elbp(bining_results=bining_results, path=elbp_write_path)
```

If *MODEL*=”hog”, then:

1. Calculate HOG results using calculate_hog() in Utils, store results in *fd* and *fd_image*.

   Parameter: orientations = 9. This parameter helps in determining the number of bins that are to be considered. As the range of $\tan^{-1}$ is between –90 and 90 degrees, I chose a value of 9 to get bins of the width of 20 degrees. With the increase of more bins, we get a feature vector of higher dimensions, thereby, expressing more details of the image (low-level information).
   Parameter: pixels_per_cell = (4, 4). This parameter helps determine the number of pixels that should belong to a block. With the value of (4, 4), we were able to generate detailed HOG pictures and feature descriptors compared to (8, 8). The number of values being binned per block is lower, thereby, expressing more details about a smaller area. This also increased the sparsity in the vector.
   Parameter: cells_per_block = (2, 2). This accounts for the number of cells to be considered as a block.
   Parameter: visualize = True. This helps us plot a HOG picture.

   Result feature vector was of dimensions: (1,8100). [15x15 (blocks) x 36 (bins per block)]

```
if MODEL == "hog":
    # Calculating hog feature descriptor
    fd, hog_image = Utils.calculate_hog(np_image, orientations=9, pixels_per_cell=(4, 4), cells_per_block=(2,2))
```

2. Stores result from *fd* and *fd_image*.

```
# Save HOG image
hog_img_write_path = WRITE_DIR / "hog.png"
plt.gcf().savefig(hog_img_write_path)

# Write path for HOG
hog_write_path = WRITE_DIR / "hog.csv"
# Writing results into a file
Utils.write_hog(path=hog_write_path, feature_values=fd)
```

## Task 2

Given:

- *IMGS_DIR*: Folder of images to be used for comparison

Output:

- Compute all feature descriptors for images stores in *IMGS_DIR*. Store results in a folder.

Important: All functions used to derive feature descriptors are mentioned in Utils.py and are used throughout the task.

1. Extract all image filenames in *IMGS_DIR*. Sort filenames and store them in *img_file_names*.
2. For each *file_name* in *img_file_names*:
    a. Prepare all paths related to reading and writing for *file_name*.

```
# Preparing all paths related to the file
file_base_name = os.path.splitext(file_name)[0]
file_path = IMGS_DIR / file_name
write_dir = WRITE_DIR / file_base_name
write_dir.mkdir(parents=True, exist_ok=True)
write_color_moments_path = WRITE_DIR / file_base_name / "color_moments.csv"
write_elbp_path = WRITE_DIR / file_base_name / "elbp.csv"
write_hog_path = WRITE_DIR / file_base_name / "hog.csv"
```

    b. Extract and store image into *np_img*.

```
# Extracting the image in an nparray for analysis
img = Image.open(str(file_path))
np_img = np.array(img)
```

    c. Calculate and store color moments using calculate_color_moments()
    d. Calculate and store ELBP feature descriptor using calculate_elbp()
    e. Calculate and store HOG feature descriptor moments using calculate_hog()

```
# Calculating and storing color moments of an image
windows = Utils.create_windows(np_img)
color_moments = np.array(Utils.calculate_color_moments(windows))
Utils.write_color_moments(data=color_moments, path=write_color_moments_path)

# Calculating and storing ELBP of an image
points = 8
radius = 1
method = 'ror'
bins, bining_results = Utils.calculate_elbp(image=np_img, points=points, radius=radius, method=method)
Utils.write_elbp(bining_results=bining_results, path=write_elbp_path)

# Calculating and storing HOG of an image
features, hog_image = Utils.calculate_hog(np_img, orientations=9, pixels_per_cell=(4, 4), cells_per_block=(2, 2))
Utils.write_hog(path=write_hog_path, feature_values=features)
```

## Task 3

Given:

- *K*: The number of similar images to be derived
- *IMAGE_ID*: ImageId for which similar images have to be extracted
- *IMGS_DIR*: Folder of images to be used for comparison
- *MODEL*: Data model to be used to calculating feature descriptors

Output:

- Compute and extract the *K* most similar images from *IMGS_DIR* when compared to *IMAGE_ID* based on *MODEL*.

Important: All functions used to derive feature descriptors are mentioned in Utils.py and are used throughout the task.

We first extract all the images in the folder and maintain a map *filename_img* where the key is the filename and the value is the nparray image.

```
# Extracting all the images in the folder

# Extracting all the file names
img_file_names = [f for f in os.listdir(IMGS_DIR) if os.path.isfile(os.path.join(IMGS_DIR, f)) and ".png" in str(os.path.join(IMGS_DIR, f))]
img_file_names.sort()

filename_img = {}

for file_name in img_file_names:

    # Preparing all paths related to the file
    file_base_name = os.path.splitext(file_name)[0]
    file_path = IMGS_DIR / file_name

    # Extracting the image in an nparray for analysis
    img = Image.open(str(file_path))
    np_img = np.array(img)

    filename_img[file_base_name] = np_img
```

Then, based on the value of *MODEL*, control flows into the if-else block appropriate.

If the input *MODEL* is ”cm8x8”:

1. Create a map *filename_cm*. (key=filename, value=color moment)
   In order to create this, we iterate on the previously created *filename_img*, calculate the color moments for each of the images (using the functions written in Utils.py) and then store them in the map *filename_cm*.

```
# Compute Color Moments for all the images and store them in a dictionary
# Dictionary: image_id -> image
filename_cm = {}
for filename in filename_img:
    image = filename_img[filename]
    windows = Utils.create_windows(image)
    image_cm = Utils.calculate_color_moments(windows)
    filename_cm[filename] = image_cm
```

2. Pop color moment of *IMAGE_ID* from *filename_cm*

```
# Delete the image of interest once color moments are calculated
image_cm = filename_cm[IMAGE_ID]
del filename_cm[IMAGE_ID]
```

3. Compute the distance between color moments of images in *filename_cm* and *image_cm*. Store results in list *distances* where each element is a tuple of (filename, distance)
   We calculate the distance using **L2-norm (euclidean distance)** as mean, standard deviation and skewness are statistical measures that do not involve angles. Over here, the distribution of the color in images does not matter as much the images are all of the equal size and are focused on the face of the subject.

```
# Compute distance between all images in the folder and image of interest
# Store results in a dictionary: image_id -> distance
distances = []
for filename in filename_cm:
    test_cm = filename_cm[filename]
    dist = np.linalg.norm(np.array(test_cm) - np.array(image_cm))
    distances.append((filename, dist))
```

4. Sort *distances* in ascending order and slice and keep first *K* elements.
   Display and save the first *K* images.

```
# Sort all the distances (ascending), slice and keep only k images
distances.sort(key=lambda x: x[1])
distances = distances[:K]

# Plot most similar images
fig, ax = plt.subplots(nrows=1, ncols=K, figsize=(17, 17))
ax = ax.flatten()
index = 0
for image_id, distance in distances:
    ax[index].imshow(filename_img[image_id], cmap='gray')
    ax[index].set_xticks([])
    ax[index].set_yticks([])
    ax[index].set_title("Order: " + str(index+1) +"\nImage ID: " + image_id + "\n Distance: " + str(distance))
    plt.subplots_adjust(left=None, bottom=None, right=None, top=None, wspace=None, hspace=0.45)
    index += 1

# Save the plotted figure
plt.savefig(CM_WRITE_PATH)
```

If the input *MODEL* is ”elbp”:

1. Create a map *filename_elbp*. (key=filename, value=elbp feature descriptor)
   In order to create this, we iterate on the previously created *filename_img*, calculate

ELBP for each of the images (using the functions written in Utils.py) and then store them in the map *filename_elbp*.

```
filename_elbp = {}
for filename in filename_img:
    image = filename_img[filename]
    bins, bining_results = Utils.calculate_elbp(image=image, points=8, radius=1, method='ror')
    filename_elbp[filename] = bining_results[0]
```

2. Pop ELBP of *IMAGE_ID* from *filename_elbp*

```
# Delete the image of interest once ELBP is calculated
image_elbp = filename_elbp[IMAGE_ID]
del filename_elbp[IMAGE_ID]
```

3. Compute the distance between ELBP of images in *filename_elbp* and *image_elbp*. Store results in list *distances* where each element is a tuple of (filename, distance)
   We calculate the distance using **L2-norm (euclidean distance)** for the same reasons mentioned in cm8x8.

```
# Compute distance between all images in the folder and image of interest
# Store results in a dictionary: image_id -> distance
distances = []
for filename in filename_elbp:
    test_elbp = filename_elbp[filename]
    dist = np.linalg.norm(np.array(test_elbp) - np.array(image_elbp))
    distances.append((filename, dist))
```

4. Sort *distances* in ascending order and slice and keep first *K* elements.
   Display and save the first *K* images.

```
# Sort all the distances (ascending), slice and keep only k images
distances.sort(key=lambda x: x[1])
distances = distances[:K]

# Plot most similar images
fig, ax = plt.subplots(nrows=1, ncols=K, figsize=(17, 17))
ax = ax.flatten()
index = 0
for image_id, distance in distances:
    ax[index].imshow(filename_img[image_id], cmap='gray')
    ax[index].set_xticks([])
    ax[index].set_yticks([])
    ax[index].set_title("Order: " + str(index+1) + "\nMethod: " + str(MODEL) + "\nImage ID: " + image_id + "\n Distance: " + str(distance))
    plt.subplots_adjust(left=None, bottom=None, right=None, top=None, wspace=None, hspace=0.45)
    index += 1

# Save the plotted figure
plt.savefig(ELBP_WRITE_PATH)
```

If the input *MODEL* is ”hog”:

1. Create a map *filename_hog*. (key=filename, value=HOG feature descriptor)
   In order to create this, we iterate on the previously created *filename_img*, calculate the

HOG for each of the images (using the functions written in Utils.py) and then store them in the map *filename_hog*.

```
filename_hog = {}
for filename in filename_img:
    image = filename_img[filename]
    features, _ = Utils.calculate_hog(image, orientations=9, pixels_per_cell=(4, 4), cells_per_block=(2, 2))
    filename_hog[filename] = features
```

2. Pop HOG feature descriptor of *IMAGE_ID* from *filename_hog*

```
# Delete the image of interest once ELBP is calculated
image_hog = filename_hog[IMAGE_ID]
del filename_hog[IMAGE_ID]
```

3. Compute the distance between HOG feature descriptors of images in *filename_hog* and *image_hog*. Store results in list *distances* where each element is a tuple of (filename, distance)
   We calculate the distance using **L2-norm (euclidean distance)** for the same reasons mentioned in cm8x8.

```
# Compute distance between all images in the folder and image of interest
# Store results in a dictionary: image_id -> distance
distances = []
for filename in filename_hog:
    test_hog = filename_hog[filename]
    dist = np.linalg.norm(np.array(test_hog) - np.array(image_hog))
    distances.append((filename, dist))
```

4. Sort *distances* in ascending order and slice and keep first *K* elements.
   Display and save the first *K* images.

```
# Sort all the distances (ascending), slice and keep only k images
distances.sort(key=lambda x: x[1])
distances = distances[:K]

# Plot most similar images
fig, ax = plt.subplots(nrows=1, ncols=K, figsize=(17, 17))
ax = ax.flatten()
index = 0
for image_id, distance in distances:
    ax[index].imshow(filename_img[image_id], cmap='gray')
    ax[index].set_xticks([])
    ax[index].set_yticks([])
    ax[index].set_title("Order: " + str(index+1) + "\nMethod: " + str(MODEL) + "\nImage ID: " + image_id + "\n Distance: " + str(distance))
    plt.subplots_adjust(left=None, bottom=None, right=None, top=None, wspace=None, hspace=0.45)
    index += 1

# Save the plotted figure
plt.savefig(HOG_WRITE_PATH)
```

## Task 4

Given:

- *K*: The number of similar images to be derived
- *IMAGE_ID*: ImageId for which similar images have to be extracted
- *IMGS_DIR* : Folder of images to be used for comparison

Output:

- Compute and extract the *K* most similar images from *IMGS_DIR* when compared to *IMAGE_ID* based on a combination of all models.

Important: All functions used to derive feature descriptors are mentioned in Utils.py and are used throughout the task.

We first extract all the images in the folder and maintain a map *filename_img* where the key is the filename and the value is the nparray image.

```
# Extracting all the images in the folder

# Extracting all the file names
img_file_names = [f for f in os.listdir(IMGS_DIR) if os.path.isfile(os.path.join(IMGS_DIR, f)) and ".png" in str(os.path.join(IMGS_DIR, f))]
img_file_names.sort()

filename_img = {}

for file_name in img_file_names:

    # Preparing all paths related to the file
    file_base_name = os.path.splitext(file_name)[0]
    file_path = IMGS_DIR / file_name

    # Extracting the image in an nparray for analysis
    img = Image.open(str(file_path))
    np_img = np.array(img)

    filename_img[file_base_name] = np_img
```

1. The following feature descriptors are calculated as previously calculated in Task 3 and their distances are computed when compared to *IMAGE_ID*:
    - Color Moments (cm8x8)
    - ELBP Feature Descriptor
    - HOG Feature Descriptor

2. Store overall distance of *IMAGE_ID* using all feature descriptors into *overall_distance*. Once the distances are obtained (*cm_distances, elbp_distances, hog_distances)*, they are all sorted on the basis of filenames. This means that the 1st element of *cm_distances, elbp_distances* and *hog_distances* are all distances of the same file with *IMAGE_ID*.We loop over all these distances and add them to determine the “total distance” of an image in the folder from *IMAGE_ID*.

```
overall_distance = []
for item in zip(cm_distances, elbp_distances, hog_distances):
    if (item[0][0] == item[1][0] == item[2][0]):
        filename = item[0][0]
        distance = item[0][1] + item[1][1] + item[2][1]
        overall_distance.append((filename, distance))
    else:
        raise Exception("Cannot compute overall distance")
```

3. Sort *overall_distance* in ascending order and slice and keep first *K* elements. Display and save the first *K* images.


```
overall_distance = []
for item in zip(cm_distances, elbp_distances, hog_distances):
    if (item[0][0] == item[1][0] == item[2][0]):
        filename = item[0][0]
        distance = item[0][1] + item[1][1] + item[2][1]
        overall_distance.append((filename, distance))
    else:
        raise Exception("Cannot compute overall distance")
```

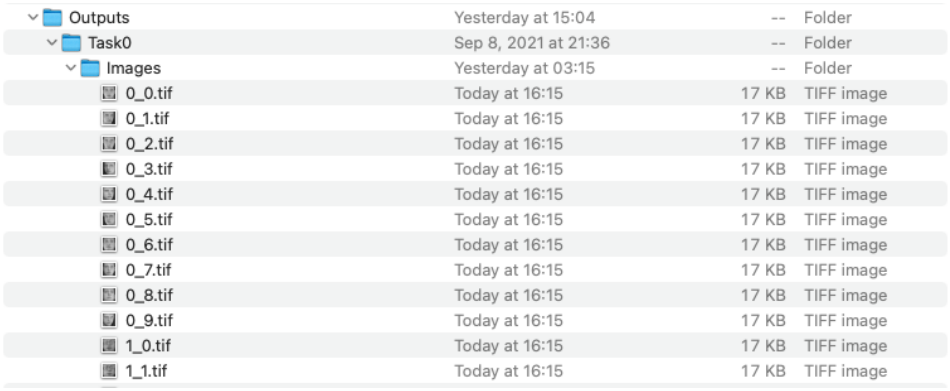
# Interface specifications

cwd: Current working directory (location of programs)

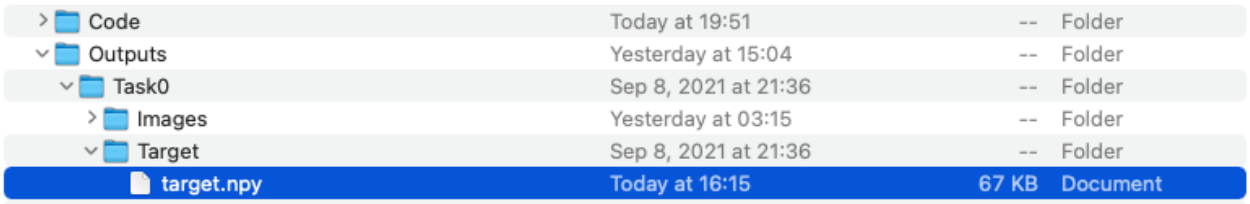
## Task 0

**Input**:
No input required
**Output**:

- Images stored in .tif in:
  cwd.parent/Outputs/Task0/Images
  Format: <subject_imageId>




- Targets are stored in
  cwd.parent /Outputs/Task0/Target/target.npy




## Task 1

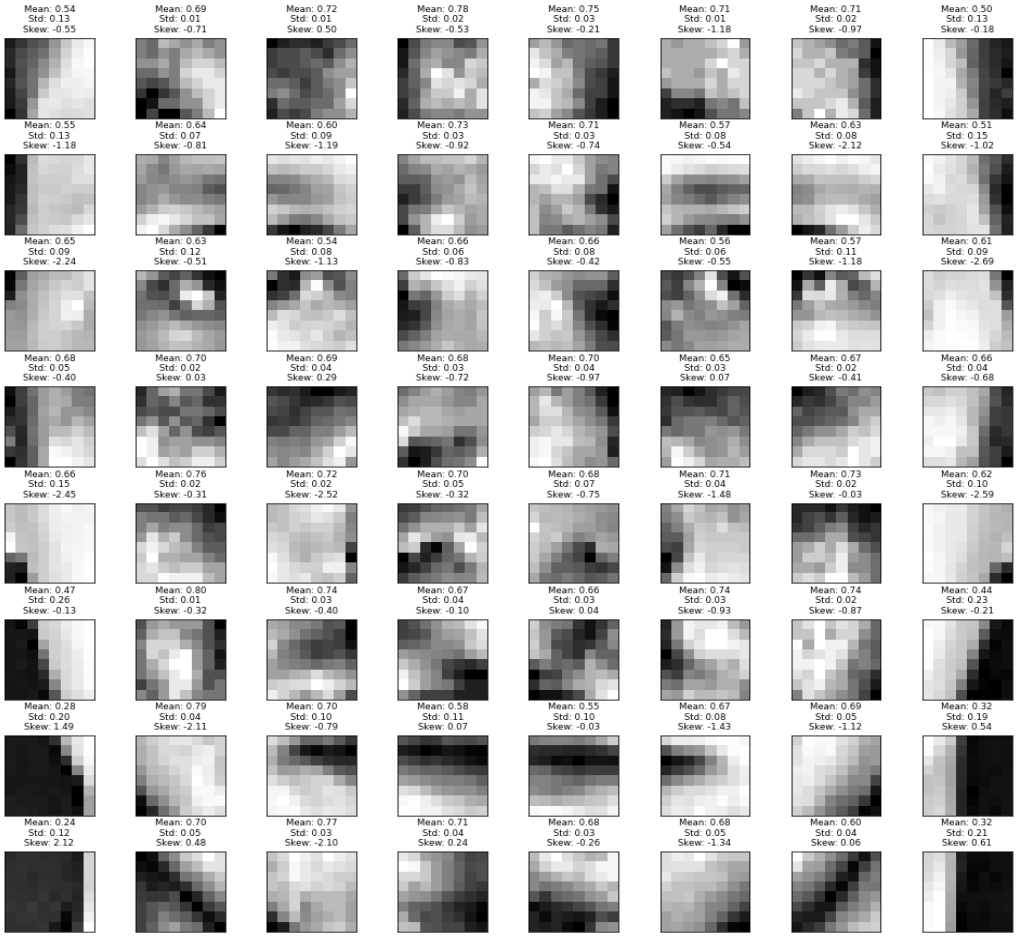
**Input**:

- IMAGE_ID (in the format: <subjectId, imageId>)
- MODEL ("cm8x8", "elbp", "hog")

**Output**:

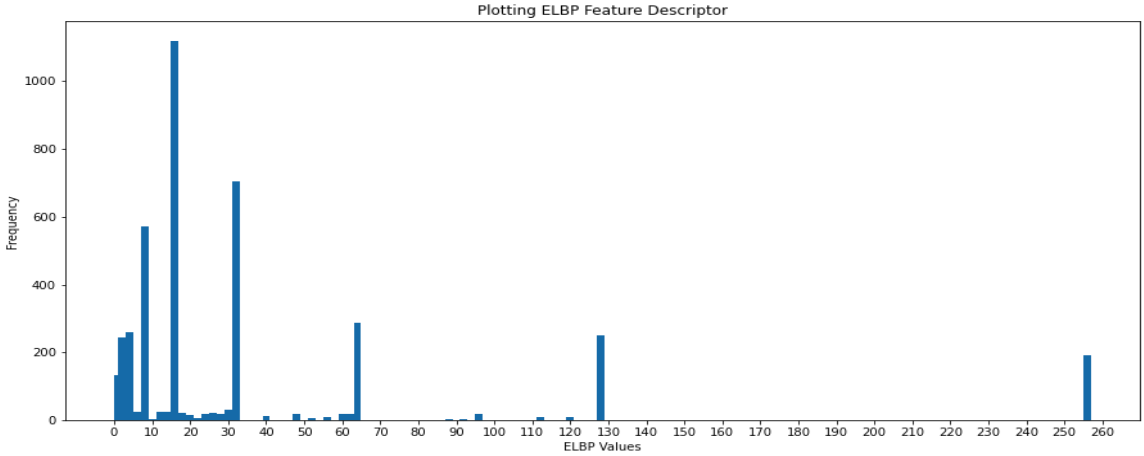
- If MODEL=”cm8x8”:
  cwd.parent/Outputs/Task1/color_moments.csv

cwd.parent/Outputs/Task1/color_moments.png

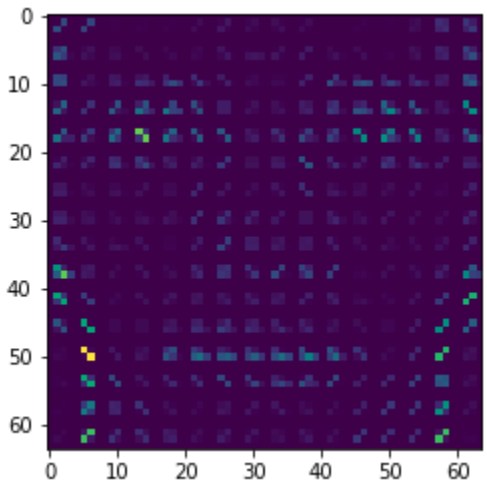



The above file demonstrates the color moments for each of the 8x8 window that was sliced.

- If MODEL=”elbp”:
  cwd.parent/Outputs/Task1/elbp.csv
  cwd.parent/Outputs/Task1/elbp.png



- If MODEL=”hog”:
  cwd.parent/Outputs/Task1/hog.csv
  cwd.parent/Outputs/Task1/hog.png



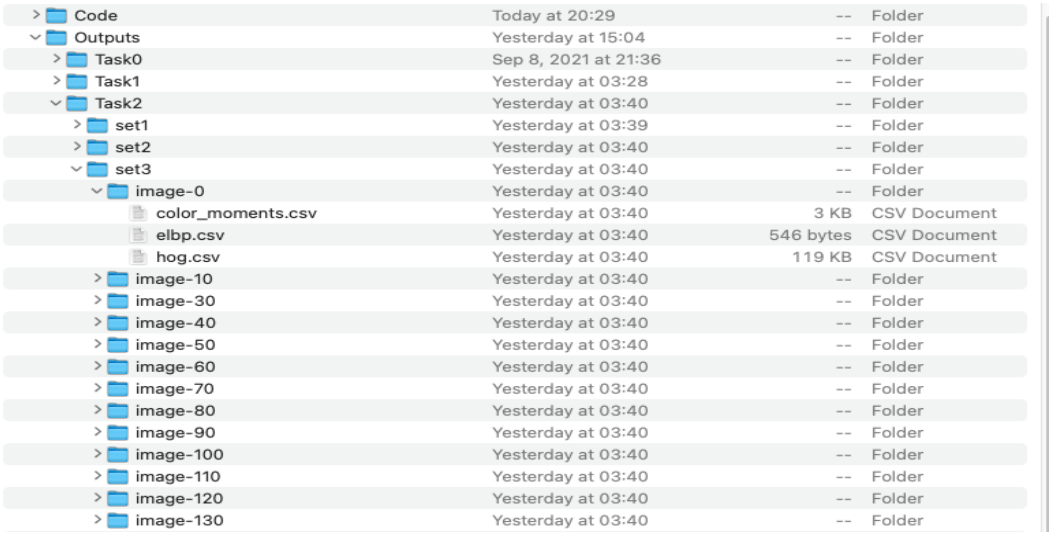

## Task 2

**Input**:

- IMGS_DIR: Path to the images directory
  Default value: cwd.parentOutputs/test_imgage_sets/set1
- WRITE_DIR: Write directory
  Default value: cwd.parentOutputs/Task2/set1

**Output**:

- All output will be stored in WRITE_DIR
- Inside WRITE_DIR, there will be a folder corresponding to images in IMGS_DIR
- Inside each folder, there will be 3 files: color_moments.csv, elbp.csv and hog.csv



## Task 3

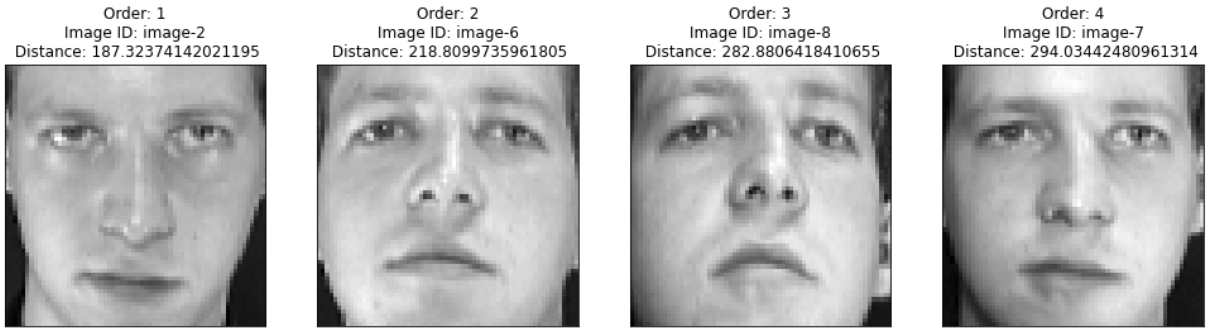
**Input**:

- IMGS_DIR: Images read path.
  Default: cwd.parent/Outputs/test_imgage_sets/set3
- WRITE_DIR: Write path.
  Default: cwd.parent/Outputs/Task3/set3
- K: Total number of similar images
  Default 4
- MODEL: Modal used for computation of descriptors ['cm8x8', 'elbp', 'hog']
  Default: “cm8x8”
- IMAGE_ID: ImageID
  Default: image-0

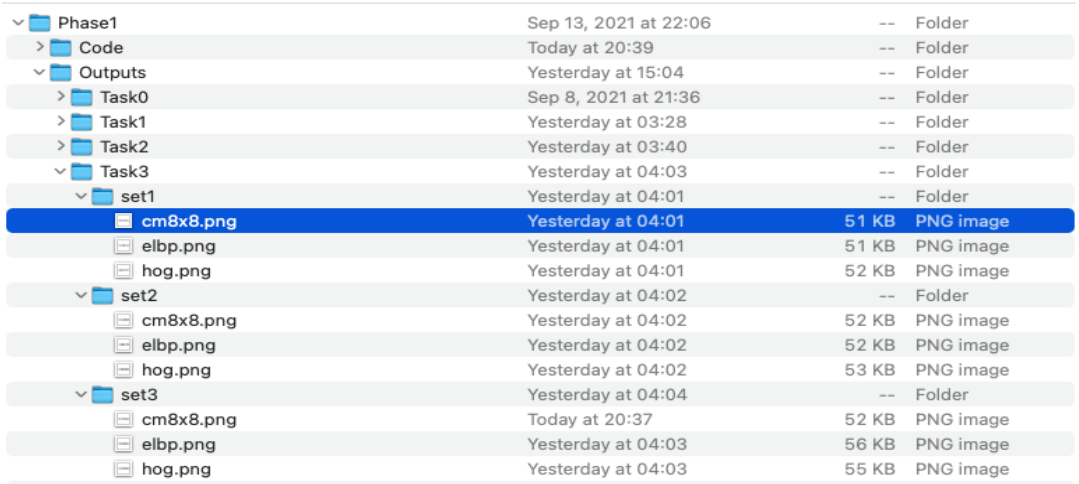
  Note: IMAGE_ID should be an image that belongs inside IMGS_DIR

**Output**:

- If MODEL=”cm8x8”, path: WRITE_DIR/cm8x8.png




- If MODEL=”elbp”, path: WRITE_DIR/elbp.png
- If MODEL=”hog”, path: WRITE_DIR/hog.png



## Task 4

**Input**:

- IMGS_DIR: Images read path.
  Default: cwd.parent/Outputs/test_imgage_sets/set3
- WRITE_DIR: Write path.
  Default: cwd.parent/Outputs/Task3/set3
- K: Total number of similar images
  Default 4
- IMAGE_ID: ImageID
  Default: image-0

**Output**:

- Path: WRITE_DIR/result.png

# System requirements/installation and executuion instructions

System Requirements

1. The system must be installed with Python 3.8.
2. Jupyter Notebook must be installed on the system.
   ```
   conda install -c conda-forge jupyterlab
   pip install jupyterlab
   ```
3. The following libraries must be installed:
   - scipy
   - numpy
   - matplotlib
   - PIL
   - pathlib
   - skimage
   - sklearn
4. These libraries can be installed using Anaconda (conda) or Pip.
5. These libraries can be installed on a system level or a virtual environment.

Execution Instructions:

1. All tasks have completed in individual files: Task0.ipynb, Task1.ipynb, Task2.ipynb, Task3.ipynb, Task4.ipynb.
2. Enter the input parameters requested on the top of the file as requested.
3. Run through all the cells to obtain the desired output.

# Related work


1. Mdakane, L and Van den Bergh, F. 2012. Extended local binary pattern features for improving settlement type classification of quickbird images. In: PRASA 2012: Twenty-Third Annual Symposium of the Pattern Recognition Association of South Africa, Pretoria, South Africa, 29-30 November 2012
   Description: This publication compares the results of previously discovered techniques of LBP and their performance on quick bird images. It also discusses an extension on the LBP technique where rotational invariant variance measure is added, thereby, producing better results as it takes contrast into account.

2. N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, 2005, pp. 886-893 vol. 1, doi: 10.1109/CVPR.2005.177.
   Description: This publication compares previous techniques that were used in edge detection and how HOG outperforms previous techniques. They discuss how the results vary based on fine-scale gradients, fine orientation binning, relatively coarse spatial binning.

# Conclusions

General Conclusions:

- Color Moments is a feature descriptor that is based on color distribution. It can be used in the case of image retrieval.
- ELBP is an extended version of LBP where rotational invariant variance measure is taken into account. It is used to understand more about the texture of an image.
- HOG is a feature descriptor that helps detect changes in gradients and it's direction. This can work great to detect certain portions of images that are different in contrast like edge detection.

Conclusions with respect to the dataset:
The feature vectors individually work well for what they are meant to convey. Color moments help us understand the color distribution of an image and tell us more about the basic characteristics of an image. ELBP helped us understand more about the texture of the image and tells us more about the local distribution of the image. HOG on the other hand gives us gradient information and tells us if there is a change in contrast towards any direction. This is particularly useful when detecting edges over a satellite image or a human face, like in our example. However, none of these feature models were able to help achieve the desired results. Images related to the subject in question were expected to be extracted as similar images for most of the models but this was not the case. This is likely because of the distance measure that was used to compare images used and the usage of an inappropriate data model to express the image.

# Bibliography

Mdakane, L and Van den Bergh, F. 2012. Extended local binary pattern features for improving settlement type classification of quickbird images. In: PRASA 2012: Twenty-Third Annual Symposium of the Pattern Recognition Association of South Africa, Pretoria, South Africa, 29-30 November 2012

N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, 2005, pp. 886-893 vol. 1, doi: 10.1109/CVPR.2005.177.

Contributors to Wikimedia projects. "Local Binary Patterns - Wikipedia." *Wikipedia, the Free Encyclopedia*, Wikimedia Foundation, Inc., 17 Nov. 2009, https://en.wikipedia.org/wiki/Local_binary_patterns.

Contributors to Wikimedia projects. "Color moments - Wikipedia." *Wikipedia, the Free Encyclopedia*, Wikimedia Foundation, Inc., 14 Jun. 2014, https://en.wikipedia.org/wiki/Color_moments.

# Appendix

Phase 1 of the MWDB project was an individual project for all members of the team. As a team, we supported each other very well and helped each other to make sure nobody was lacking behind. We actively discussed with each other regarding our ideas and thoughts on Discord and also met after call to understand more about the assignment.