Q1.1.1

Gaussian

A Gaussian filter will have a blurring effect, with more prominent blurring manifesting from a larger kernel size.

Laplacian of the Gaussian

The Laplacian of the Gaussian is used for general ridge detection, as opposed to the two listed below, which specify edges in one direction (vertical or horizontal).

Derivative of the Gaussian in the X-Direction

The derivative of the Gaussian in the X-Direction is used to identify the vertical edges of an image.

Derivative of the Gaussian in the Y-Direction

The derivative of the Gaussian in the Y-Direction is used to identify the horizontal edges of an image.

Why multiple scales?

Applying the filter at multiple scales allows one to extract features that may be more or less prominent depending on its pixel size in an image. For example, taking the Gaussian blur at lower scales of a tree will generally preserve the features of the branches, but at higher scales the tree turns into more of an outline. There is no way beforehand to be certain which features, the branches or the trees, will hold the most relevant information for out application.

Q1.1.2

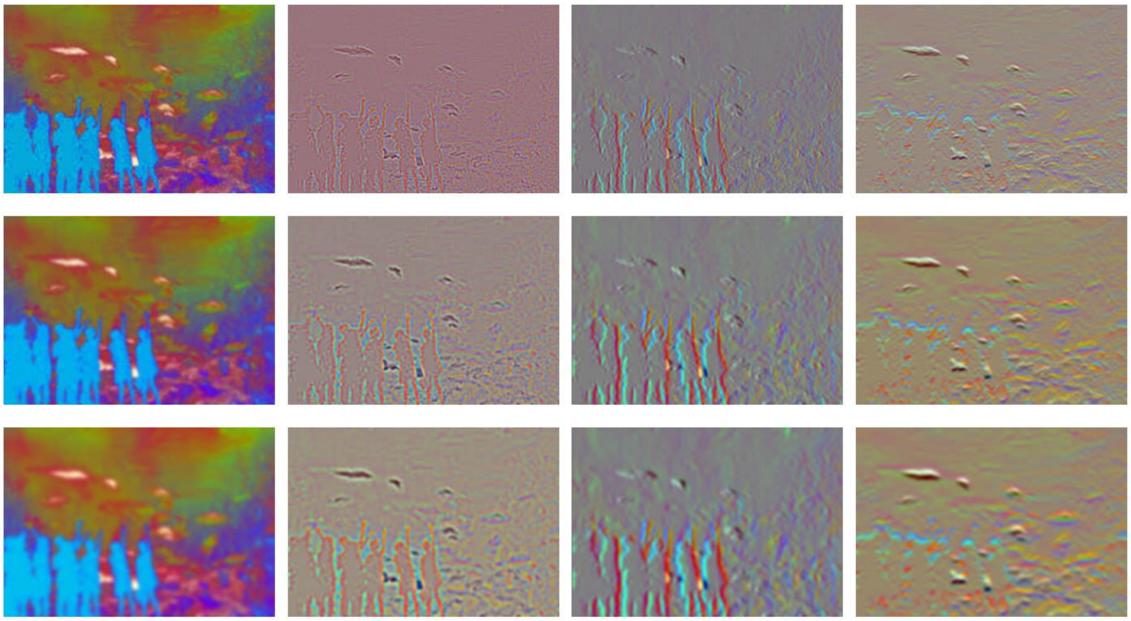
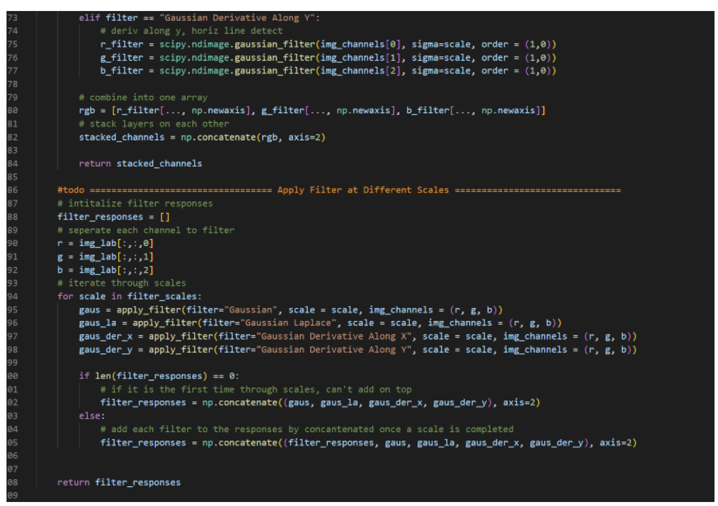


Fig 1. Filter responses for “aquarium/sun aztvjgubyrgvirup.jpg” taken at scales of [0, 1, 2], with the Gaussian Filter, Laplacian of the Gaussian, the Derivative of the Gaussian in the X-Direction, and the Derivative of the Gaussian in the Y-Direction, respectively.

```python
def extract_filter_responses(opts, img):
    """
    Extracts the filter responses for the given image.

    [input]
    * opts    : options
    * img     : numpy.ndarray of shape (H,W) or (H,W,3)
    [output]
    * filter_responses: numpy.ndarray of shape (H,W,3F)
    """

    filter_scales = opts.filter_scales # array [1,2]
    # ----- TODO -----
    #todo ================================= Ensure Num Channels ==========================================
    # check the channels and adjust accordingly
    img_channel = img.shape[-1]
    if img_channel == 3: #! normal
        pass
    elif img.ndim == 2: #! greyscale
        # stack the img on itself along the depth, or the last axis
        img = np.stack((img, img, img), axis=2)
    elif img_channel == 4: #! four channels
        # using indexing, take every value in the first two dims, but only the first
        # three channels
        img = img[:,:,2]
    else:
        raise ValueError(f"Your image is of dimension {img_channel}, anticipated channels are 1, 3, or 4.")

    #todo ================================= Change to Lab Color Space =================================
    img_lab = skimage.color.rgb2lab(img) # convert to lab color system
    #todo ================================= Streamline Filter Application =============================

    def apply_filter(filter: str, scale: int, img_channels: tuple) -> np.ndarray:
        """ Apply one of four filters to each seperate channel and then into one image.

        Args:
            filter (str): Modes: Gaussian, Gaussian Laplace, Gaussian Derivate Along X, and Gaussian Derivative Along Y.
            Input is case sensitive.
            scale (int): Refers to the scale that the filter is subject to, corresponds to the size of the filter being applied.
            img_channels (tuple): Since each channel is filtered seperately and then combined, this tuple must be of len 2.

        Returns:
            np.ndarray: Returns an np array of each channel that has been filtered seperately and then combined.
        """
        if filter == "Gaussian":
            # normalize vals, blurring effect
            r_filter = scipy.ndimage.gaussian_filter(img_channels[0], sigma=scale, order = 0)
            g_filter = scipy.ndimage.gaussian_filter(img_channels[1], sigma=scale, order = 0)
            b_filter = scipy.ndimage.gaussian_filter(img_channels[2], sigma=scale, order = 0)
        elif filter == "Gaussian Laplace":
            # notice sharp changes,
            r_filter = scipy.ndimage.gaussian_laplace(img_channels[0], sigma=scale)
            g_filter = scipy.ndimage.gaussian_laplace(img_channels[1], sigma=scale)
            b_filter = scipy.ndimage.gaussian_laplace(img_channels[2], sigma=scale)
        elif filter == "Gaussian Derivative Along X":
            # deriv along x, vert line detect
            r_filter = scipy.ndimage.gaussian_filter(img_channels[0], sigma=scale, order = (0,1))
            g_filter = scipy.ndimage.gaussian_filter(img_channels[1], sigma=scale, order = (0,1))
            b_filter = scipy.ndimage.gaussian_filter(img_channels[2], sigma=scale, order = (0,1))
```



```python
        elif filter == "Gaussian Derivative Along Y":
            # deriv along y, horiz line detect
            r_filter = scipy.ndimage.gaussian_filter(img_channels[0], sigma=scale, order = (1,0))
            g_filter = scipy.ndimage.gaussian_filter(img_channels[1], sigma=scale, order = (1,0))
            b_filter = scipy.ndimage.gaussian_filter(img_channels[2], sigma=scale, order = (1,0))

        # combine into one array
        rgb = [r_filter[..., np.newaxis], g_filter[..., np.newaxis], b_filter[..., np.newaxis]]
        # stack layers on each other
        stacked_channels = np.concatenate(rgb, axis=2)

        return stacked_channels

    #todo ================================= Apply Filter at Different Scales ==============================
    # intitalize filter responses
    filter_responses = []
    # seperate each channel to filter
    r = img_lab[:,:,0]
    g = img_lab[:,:,1]
    b = img_lab[:,:,2]
    # iterate through scales
    for scale in filter_scales:
        gaus = apply_filter(filter="Gaussian", scale = scale, img_channels = (r, g, b))
        gaus_la = apply_filter(filter="Gaussian Laplace", scale = scale, img_channels = (r, g, b))
        gaus_der_x = apply_filter(filter="Gaussian Derivative Along X", scale = scale, img_channels = (r, g, b))
        gaus_der_y = apply_filter(filter="Gaussian Derivative Along Y", scale = scale, img_channels = (r, g, b))

        if len(filter_responses) == 0:
            # if it is the first time through scales, can't add on top
            filter_responses = np.concatenate((gaus, gaus_la, gaus_der_x, gaus_der_y), axis=2)
        else:
            # add each filter to the responses by concantenated once a scale is completed
            filter_responses = np.concatenate((filter_responses, gaus, gaus_la, gaus_der_x, gaus_der_y), axis=2)


    return filter_responses
```

Fig 2. Extract_filter_responses() code.

Q1.2

```python
def compute_dictionary_one_image(args):
    """ Takes in an image path, opens the image, and save to a temporary file.

    Args:
        arg (tuple): In the form (opts, img_path)
    """
    # ----- TODO -----
    # extract params from args
    img_path = args[1]
    opts = args[0]

    # open the image and change to ndarray
    img = Image.open(join(opts.data_dir, img_path))
    img = np.array(img).astype(np.float32) / 255

    # obtain the filter response
    filter_response = extract_filter_responses(opts=opts, img=img) # filter_response.shape = (H, W, 3F)

    #fb_size = filter_response.shape[2] # needed to reshape the filter response with proper depth
    # reshape filter_response to get a list of all pixels, with a depth the size of the filter bank (3 * filters * num scales)
    # this represents a list of all the pixels
    #fr_list = np.concatenate(filter_response, axis=-1)
    H = filter_response.shape[0] # height of fr
    W = filter_response.shape[1] # width of fr
    fbs = filter_response.shape[2] # how many filters we took
    fr_list = np.zeros(shape=(H*W, fbs)) # turns the image into a list

    k = 0 # allows you to iterate through all pixels
    for i in range(0, H):
        for j in range(0, W):
            fr_list[k] = filter_response[i][j][:]
            k += 1

    nPixels = fr_list.shape[0]
    # use alpha to get a subset of random pixels -> np.array of len alpha
    alpha_pixels = np.random.choice(nPixels, opts.alpha) # this will get a list of alpha pixels from the number of available pixels
    alpha_fr = fr_list[alpha_pixels] # get the response at the locations of alpha pixels

    temp_file_path = join(opts.feat_dir + "/" + img_path[0:-4] + ".npy")
    try:
        np.save(temp_file_path, alpha_fr)
    except FileNotFoundError:
        # if the file wasn't found then the img category folder isnt made, make one
        img_dir = img_path.split("/")[0]
        os.mkdir(join(opts.feat_dir + "/" + img_dir)) # go to feat dir and make new directory
        np.save(temp_file_path, alpha_fr)

    print(f"Finished saving {temp_file_path}...")
    return temp_file_path
```

Fig 3. Compute_dictionary_one_image() code.

```python
def compute_dictionary(opts, n_worker=1):
    """
    Creates the dictionary of visual words by clustering using k-means.

    [input]
    * opts          : options
    * n_worker      : number of workers to process in parallel

    [saved]
    * dictionary : numpy.ndarray of shape (K,3F)
    """
    data_dir = opts.data_dir
    feat_dir = opts.feat_dir
    out_dir = opts.out_dir
    K = opts.K

    train_files = open(join(data_dir, "train_files.txt")).read().splitlines()
    # ----- TODO -----
    # list of params to pass to compute_dictionary_one_img: unique training file path, same feat_dict path
    pool_params = [(opts, img_path) for img_path in train_files]
    with multiprocessing.Pool(processes=n_worker) as pool:
        results = pool.map(compute_dictionary_one_image, pool_params) # map all the image files to a process
    pool.close() # close the pool
    print("Finished extracting filter all responses.")

    # load back up all the npy files into a list of filter responses
    print("Loading all temporary files...")
    filter_responses = []
    for temp_file in results:
        print(f"Loading {temp_file}...")
        fr_img = np.load(temp_file)
        # list of fr size (image, alpha, filters)
        filter_responses.append(fr_img)

    # change to np.array to resize
    #filter_responses = np.concatenate(filter_responses, axis=0)
    T = len(filter_responses) # num images
    #alpha = filter_responses.shape[1] # also opts.alpha
    fb_size = 3 * len(opts.filter_scales) * 4 # filter bank size
    alphaT = opts.alpha * T
    # resize to (alpha*t) x 3F
    alpha_fr = np.zeros(shape=(alphaT, fb_size))
    k = 0
    for i in range(0, T):
        for j in range(0, opts.alpha):
            alpha_fr[k] = filter_responses[i][j]
            k += 1

    print("Loaded all filter responses...")
    # calculate the kmeans and save the dictionary
    print("Started calculating K-means...")
    kmeans = KMeans(n_clusters=K).fit(alpha_fr)
    dictionary = kmeans.cluster_centers_
    print("Finished calculating K-means, saving cluster centers...")
    # example code snippet to save the dictionary
    np.save(join(out_dir, 'dictionary.npy'), dictionary)
```

Fig 4. Compute_dictionary() code.

Q1.3

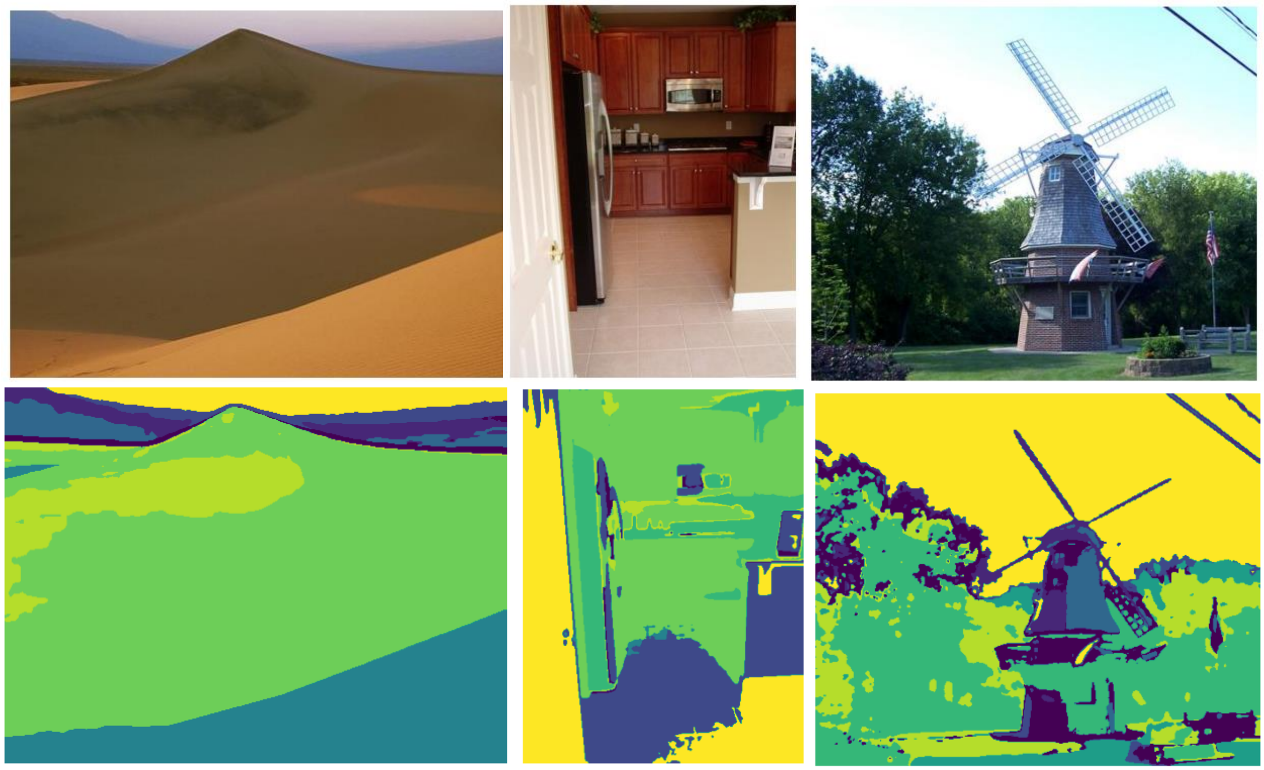


Fig 5. Original RGD images and visualized wordmaps for "desert/sun_aaqyzvrweabdxjzo.jpg", "kitchen/sun_aaqhazmhbhefhakh.jpg", and "windmill/sun_bxpvmlxprftjwynu.jpg" (respectively). The dictionary used for mapping was created with K=10, α=25, and filter responses at scales [0,1, 2].

Comments

These wordmaps are more than responsible, one can observe countless major boundary changes. While not comprehensive, major features like independent bodies, changes in light intensity, and foreground/background all seem to be present in the images.

```
def get_visual_words(opts, img, dictionary):
    """
    Compute visual words mapping for the given img using the dictionary of visual words.

    [input]
    * opts    : options
    * img     : numpy.ndarray of shape (H,W) or (H,W,3)
    * dictionary : numpy.ndarray of shape (K,3F)

    [output]
    * wordmap: numpy.ndarray of shape (H,W)
    """

    # ----- TODO -----
    feature_responses = extract_filter_responses(opts, img)

    fr_H = feature_responses.shape[0]
    fr_W = feature_responses.shape[1]
    feature_bank = feature_responses.shape[2]

    # now we have a list of pixel responses across 36 channels
    fr_list = np.zeros(shape=(fr_H*fr_W, feature_bank))
    k = 0 # allows you to iterate through all pixels
    for i in range(0, fr_H):
        for j in range(0, fr_W):
            fr_list[k] = feature_responses[i][j][:]
            k += 1

    # for each pixel, finds the distance to a cluster center across 36 channels
    dists =  scipy.spatial.distance.cdist(fr_list, dictionary)
    # for each pixel in dist, we have 10 distances, find the min dist
    # this min dist shows which cluster the image belongs to
    word_map = np.argmin(dists, axis=1)

    word_map = word_map.reshape(fr_H, fr_W, -1)

    return word_map
```

Fig 6. Get_visual_words() code.

Q2.1

```
def get_feature_from_wordmap(opts, wordmap, img_path):
    """
    Compute histogram of visual words.

    [input]
    * opts      : options
    * wordmap   : numpy.ndarray of shape (H,W)

    [output]
    * hist: numpy.ndarray of shape (K)
    """

    K = opts.K
    # ----- TODO -----

    hist = np.zeros(shape=(K)) # init hist size

    # resize wordmap to a 1D array
    wm_H = wordmap.shape[0]
    wm_W = wordmap.shape[1]
    wordmap_list = np.resize(wordmap, new_shape=(wm_H*wm_W))

    # go through clusters (1-10) and count how many times it appears i
    for cluster in range(0, K):
        occur = np.count_nonzero(wordmap_list == cluster, axis = 0)
        hist[cluster-1] = occur #

    # # perform l1 normalization on the histogram entries
    L1_norm = np.linalg.norm(hist, ord=1)

    # # normalize hist
    # # this now represents the freq of which features occur
    hist = hist/L1_norm

    return hist
```

Fig 7. Get_feature_from_wordmap() code.

Q2.2

```python
def get_feature_from_wordmap_SPM(opts, wordmap, img_path):
    """
    Compute histogram of visual words using spatial pyramid matching.
    [input]
    * opts      : options
    * wordmap   : numpy.ndarray of shape (H,W)
    [output]
    * hist_all: numpy.ndarray of shape K*(4^(L+1) - 1) / 3
    """
    K = opts.K
    L = opts.L
    # ----- TODO -----
    #todo =============================== find cells for each layer ====================================
    layers = [] # create a layers array to hold each layers cells
    for i in range(0,  L + 1):
        # find and floor the number of cells in each row - will tell you how many pixels to jump below
        cell_W = int(wordmap.shape[1] // (2 ** i))
        cell_H = int(wordmap.shape[0] // (2 ** i))
        # calculate the cells for the finest layer
        cells = []

        for x in range(0,wordmap.shape[0],cell_H):
            for y in range(0,wordmap.shape[1],cell_W):
                    if len(cells) < (2**i * 2**i):
                        cell = wordmap[x:x+cell_H,y:y+cell_W, :]
                        if cell.shape[0] == cell_H and cell.shape[1] == cell_W:
                            cells.append(np.array(cell))

        layers.append(cells)
    #todo ============================= get feature maps =============================================
    list_hists = []
    for layer in layers:
        layer_hist = []
        for cell in layer:
            cell_hist = get_feature_from_wordmap(opts, cell, img_path)
            layer_hist.append(cell_hist)

        layer_hist = np.concatenate(layer_hist, axis = 0)
        list_hists.append(layer_hist)
    #todo =============================== obtain layer weights ==========================================
    weights = []
    for layer in range(0, L + 1):
        if layer == 0 or layer == 1:
            weight = 2 ** -L
        else:
            weight = 2**(layer-L-1)
        weights.append(weight)
    #todo ============================== get total histogram ========================================
    # apply the weights
    for i in range(0, len(list_hists) - 1):
        list_hists[i] = weights[i] * list_hists[i]

    hist_all = np.concatenate(list_hists, axis = 0)
    L1_norm = np.linalg.norm(hist_all, ord=1)
    hist_all = hist_all/L1_norm
    return hist_all
```

Fig 8. Get_feature_from_wordmap_SPM() code.

Q2.3

```python
def similarity_to_set(word_hist, histograms):
    """
    Compute similarity between a histogram of visual words with all training image histograms.

    [input]
    * word_hist: numpy.ndarray of shape (K)
    * histograms: numpy.ndarray of shape (N,K)

    [output]
    * sim: numpy.ndarray of shape (N)
    """

    # ----- TODO -----

    # find the minimum betweeen an individual word_hist and the N list of histograms
    # this will refer to the overlap between the word_hist and each histogram accross the K dimension
    mins = np.minimum(word_hist, histograms)

    # sum those minimum values to find the similarity
    # higher values indicate a higher change of similarity
    sim = np.sum(mins, axis = 1)

    return sim
```

Fig 9. Similarity_to_set() code.

Q2.4

```python
def build_recognition_system(opts, n_worker=1):
    """
    Creates a trained recognition system by generating training features from all training images.

    [input]
    * opts        : options
    * n_worker  : number of workers to process in parallel

    [saved]
    * features: numpy.ndarray of shape (N,M)
    * labels: numpy.ndarray of shape (N)
    * dictionary: numpy.ndarray of shape (K,3F)
    * SPM_layer_num: number of spatial pyramid layers
    """
    #

    data_dir = opts.data_dir
    out_dir = opts.out_dir
    SPM_layer_num = opts.L

    train_files = open(join(data_dir, "train_files.txt")).read().splitlines()
    train_labels = np.loadtxt(join(data_dir, "train_labels.txt"), np.int32)
    dictionary = np.load(join(out_dir, "dictionary.npy"))

    # ----- TODO -----

    # all we need to do is get the features from SPM
    # need to obtain the image features
    pool_params = [(opts, img_path, dictionary) for img_path in train_files]
    with multiprocessing.Pool(processes=n_worker) as pool:
        results = pool.starmap(get_image_feature, pool_params) # can take multiple arguments
    pool.close()

    # results is a list of N hists, convert to a matrix
    features = np.stack(results, axis = 0)
    print("Done Getting Features. Shape: ", features.shape)
    # example code snippet to save the learned system
    np.savez_compressed(join(out_dir, 'trained_system.npz'),
        features=features,
        labels=train_labels,
        dictionary=dictionary,
        SPM_layer_num=SPM_layer_num,
    )
```

Fig 10. Build_recognition_system() code.

```python
def get_image_feature(opts, img_path, dictionary):
    """
    Extracts the spatial pyramid matching feature.

    [input]
    * opts      : options
    * img_path  : path of image file to read
    * dictionary: numpy.ndarray of shape (K, 3F)


    [output]
    * feature: numpy.ndarray of shape (K)
    """

    # ----- TODO -----

    #load the img
    img = Image.open(join(opts.data_dir, img_path))
    img = np.array(img).astype(np.float32) / 255
    # # get the wordmap from the image
    wordmap = visual_words.get_visual_words(opts, img, dictionary)
    # use the visual words to get the SPM features
    feature = get_feature_from_wordmap_SPM(opts, wordmap, img_path)

    print(f"Obtaining features for images...")

    return feature
```

Fig 11. get_image_feature() code.

Q2.5

Confusion Matrix

Table 1. Confusion Matrix for the test set of images, with the rows corresponding to predicted image class based on the model and the columns corresponding to actual image class. The model's parameters were K=10, L=1, α=25, and filter scales of [1, 2, 4, 8, 8$\sqrt{2}$].

Classes:	Aquarium	Desert	Highway	Kitchen	Laundromat	Park	Waterfall	Windmill
Aquarium	31	0	1	4	3	2	3	5
Desert	1	32	6	3	4	1	0	8
Highway	2	6	27	2	1	4	2	4
Kitchen	4	4	0	29	12	3	3	1
Laundromat	1	4	0	8	22	3	9	2
Park	2	0	2	2	4	32	8	6
Waterfall	6	2	2	1	2	2	22	3
Windmill	3	2	12	1	2	3	3	20

Accuracy

The model's accuracy, with the parameters specified for Table 1, was roughly 53.9%.

```python
def evaluate_recognition_system(opts, n_worker=1):
    """
    Evaluates the recognition system for all test images and returns the confusion matrix.

    [input]
    * opts        : options
    * n_worker  : number of workers to process in parallel

    [output]
    * conf: numpy.ndarray of shape (8,8)
    * accuracy: accuracy of the evaluated system
    """

    data_dir = opts.data_dir
    out_dir = opts.out_dir

    trained_system = np.load(join(out_dir, "trained_system.npz"))
    dictionary = trained_system["dictionary"]

    # using the stored options in the trained system instead of opts.py
    test_opts = copy(opts)
    test_opts.K = dictionary.shape[0]
    test_opts.L = trained_system["SPM_layer_num"]

    test_files = open(join(data_dir, "test_files.txt")).read().splitlines()
    test_labels = np.loadtxt(join(data_dir, "test_labels.txt"), np.int32)

    # ----- TODO -----
    # get a histogram for each test_file
    pool_params = [(opts, img_path, dictionary) for img_path in test_files]
    with multiprocessing.Pool(processes=n_worker) as pool:
        test_hists = pool.starmap(get_image_feature, pool_params) # can take multiple arguments
    pool.close()
    # all the histograms of trained system
    trained_hists = trained_system["features"]
    # all the labels of the trained images
    train_labels = np.loadtxt(join(data_dir, "train_labels.txt"), np.int32)

    conf = np.zeros(shape=(8,8))
    correct = 0
    accuracy = 0
    for i in range(0, len(test_hists) - 1):
        sim = similarity_to_set(test_hists[i], trained_hists) # find the similarity vector, len = trained_hists
        max_sim = max(sim) # find the max similarity score
        max_loc = np.where(sim == max_sim) # location of max similarity
        # max_sim is the hist the test val is most like > get its position in the trained_hists list > map that to a label
        predicted_label = train_labels[max_loc[0][0]]

        if predicted_label == test_labels[i]:
            correct += 1

        accuracy = round((correct/(i+1))*100, 2)
        print(f"Expected Result: {test_labels[i]} | Predicted Result: {predicted_label} | Percent Complete: {round(((i+1)/len(test_hists))*100, 2)}% | Accuracy: {accuracy}%")
        # add to conf matrix: predicted along axis 0, actual along axis 1
        conf[predicted_label][test_labels[i]] += 1

    accuracy = np.trace(conf)/np.sum(conf)
    return conf, accuracy
```

Fig 12. get_image_feature() code.

Q2.6

Find the Failures

Based on the confusion matrix (Table 1), a high level of Highway images were being misclassified as Windmill images and a high level Laundromat images were being misclassified as Kitchen images.

In terms of rationalizing these mismatches, the Laundromat/Kitchen pair is straightforward. They are two of the three categories that represent indoor scenes. Unlike aquariums, which have a lot of blue, the kitchen and laundromat images show many similarities. For example, both classes have overhead lights which deposit light in a similar way on surfaces. Also, many images from both classes have an offset vanishing point and which provide similar perspectives of surfaces and edges.

The Highway and Windmill images have some more nuanced similarities. A more prominent one being the horizon line being roughly centered in the image for both classes, the ability to clearly see the sky and the ground in both images. However, other classes also show this to some degree (Desert, Waterfall). It seems there is another distinguishing factor, which may be the fact that there is a lot of smaller features in the lower half of both Windmill and Highway images. There are vehicles on the lower half of highway images, and in some windmill images we see farm animals, people, and even some cars showing up in the lower half.

Q3.1

Table of Ablation

Table 2. Table of ablation for various model parameters. Models with a star (*) are initial parameters.

K (Features)	α (Pixels)	Filter Scales	L (Layer)	Accuracy (%)
10*	25*	$[1, 2, 4, 8, 8\sqrt{2}]$*	1*	53.8*
100	25	$[1, 2, 4, 8, 8\sqrt{2}]$	1	61.1
50	25	$[1, 2, 4, 8, 8\sqrt{2}]$	2	62.7
10	25	$[1, 2, 4, 8, 8\sqrt{2}, 16]$	1	52.6
100	25	$[1, 2, 4, 8, 8\sqrt{2}]$	2	67.2

Changes

The first change was to implement a higher number of cluster centers, which correspond to features. This was done to attempt to give the model more information to glean from the training set of images, so be able to distinguish. From an implementation sense, it makes our word histograms longer ($K(4^{(L+1)}$-1 = 1500), which allowed our model to obtain more information when determining an image's similarity to one from the training set.

Following this same line of reasoning, I increased the layer number to 2 and the reduced the features K parameter to 50, to give roughly double the original amount of histogram entries to compare. This led to an increase in model performance, of no surprise.

Finally, to receive a final model accuracy of 67.2%, I set K=100 and L=2. This generates a histogram length of just over four times that of the default parameters. As to why I chose to go in the direction of features and layers for model performance, I saw no increase in accuracy when changing the filter scales from the default parameters. I could reason that increased filter scales could prove beneficial at some parameters, but the need to adjust all four parameters made that path less desirable.

Within my initial testing, I operated on the reasoning for longer histograms we obtained more accurate prediction. However, I could also foresee receiving diminishing returns on the number of features being obtained to where the obtaining a larger feature histogram outweighs the nominal increase in model accuracy.