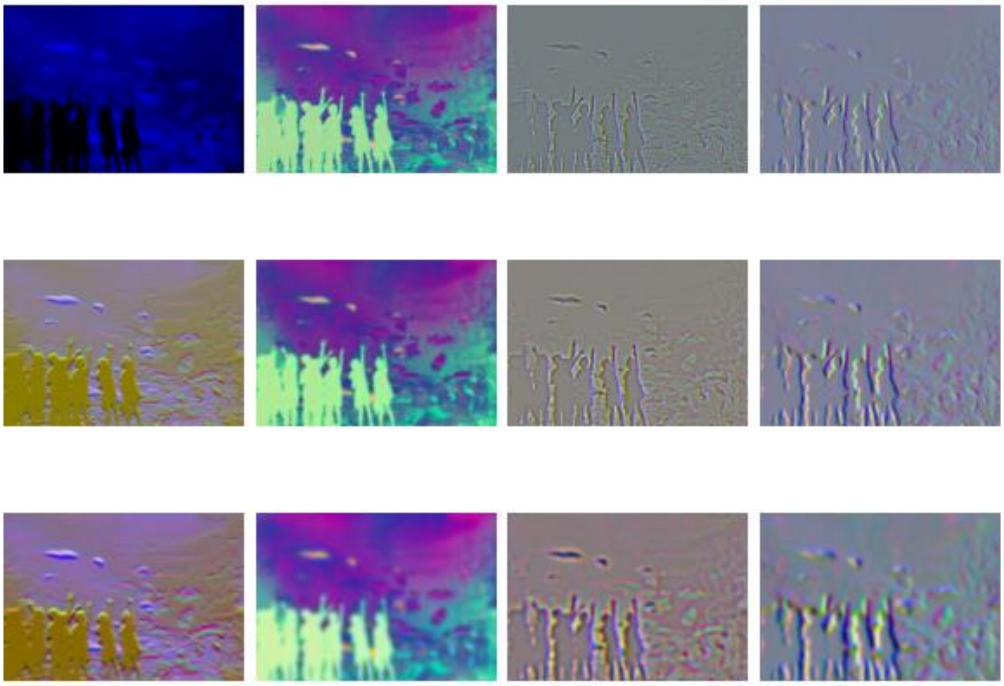
Q1.1.1

Gaussian filter can pick up the low frequency details and remove noise in image. Laplacian of Gaussian filter is using a Gaussian filter to remove noise and smooth the image, then using Laplacian filters to find areas of rapid change (edges) in images. Derivative of Gaussian in the x direction filter picks up texture features along the x axis. Derivative of Gaussian in the y direction filter picks up texture features along the y axis.

We need multiple scales of filter to detect different sizes of edges and compare the effects of smoothing and edges detection.

Q1.1.2

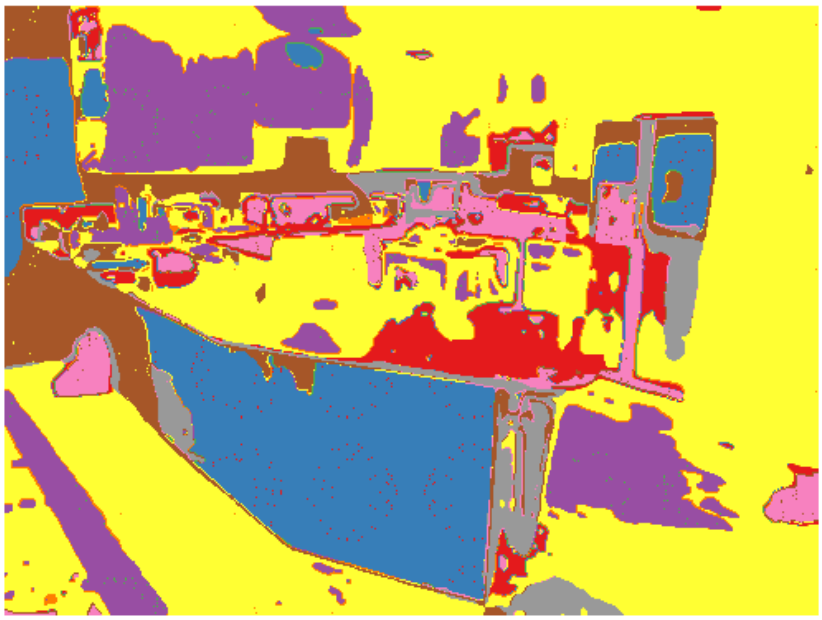


Q1.2

```
def compute_dictionary_one_image(img_path, opts):
    '''
    Extracts a random subset of filter responses of an image and save
it to disk
    This is a worker function called by compute_dictionary

    Your are free to make your own interface based on how you
implement compute_dictionary
    '''

    # ----- TODO -----
    img_path = join(opts.data_dir, img_path)
    img = Image.open(img_path)
    img = np.array(img).astype(np.float32)/255
    responses = extract_filter_responses(opts, img)
    # reshape (M,N,3F) to (MxN,3F)
    responses_reshape = responses.reshape(responses.shape[0] *
responses.shape[1], responses.shape[2])
    np.random.shuffle(responses_reshape)
    filter_output = responses_reshape[0:opts.alpha, :]      # select
alpha pixels randomly
    save_path = 'tmp_' + str(img_path.split('/')[-1][:-4])
    np.save(save_path, filter_output)
    pass

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

    #  For testing purpose, you can create a train_files_small.txt to
```

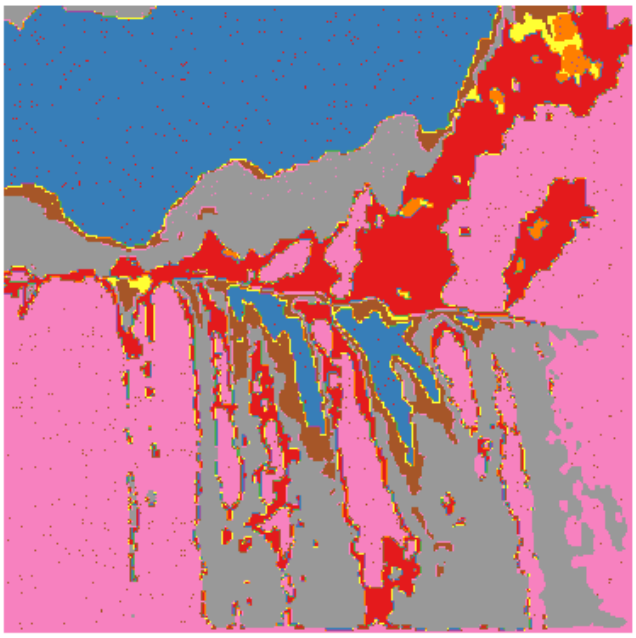
```
only load a few images.
    train_files = open(join(data_dir,
'train_filess.txt')).read().splitlines()
    # ----- TODO -----
    p = multiprocessing.Pool(processes=n_worker)
    partial_compute = partial(compute_dictionary_one_image,
opts=opts)
    p.map(partial_compute, train_files)

    for i in range(len(train_files)):
        # e.g. 'aquarium/sun_asgtepdmsxsrqqvy.jpg' --
'tmp_sun_asgtepdmsxsrqqvy.npy'
        cur_path = 'tmp_' + str(train_files[i].split('/')[-1][:-4]) +
'.npy'
        cur = np.load(cur_path)
        os.remove(cur_path)
        if i == 0:
            filter_responses = np.array(cur)
            continue
        filter_responses = np.concatenate((filter_responses, cur),
axis=0)

    kmeans =
sklearn.cluster.KMeans(n_clusters=K).fit(filter_responses)
    dictionary = kmeans.cluster_centers_
    np.save(join(out_dir, 'dictionary.npy'), dictionary)
    pass
```

Q1.3



This picture shows the features like edges of closets, cookers, floor, table and fridge. But the bag on floor is recognized as a similar feature to microwave.

The boundaries in this picture is very clear, the main feature is desert, which is painted in brown. And the textures of sky and distant mountains are also depicted in different colors. The shadow side of desert is in different color in terms of sunlight generating edges.



The features about sky, mountain and trees are pretty evident, since these objects have very different textures. However, the edges between waterfall and trees are not that clear because of some shadows in original picture.

Q2.1

```
def get_feature_from_wordmap(opts, wordmap):
    '''
    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 -----
    wordmap = wordmap.flatten()
    hist = np.bincount(wordmap, minlength=K)
    total = sum(hist)
    hist = hist/total
    return hist
```

Q2.2

```
def get_feature_from_wordmap_SPM(opts, wordmap):
    '''
    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)/3)
    '''
    K = opts.K
    L = opts.L
    # ----- TODO -----
    hist_all = np.empty((0,), dtype=float)
    H, W = wordmap.shape
    weights = np.array([0.25]*(L+1))
    for i in range(2, L+1):
        weights[i] = weights[i-1]*2
    # calculate the last layer(layer=L) and store hist results in
memo
    step_row, step_col = H // (2 ** L), W // (2 ** L)
    last_hist = np.zeros((2**L, 2 ** L, K))
    for i in range(2**L):
        for j in range(2**L):
            cur_cell = wordmap[i*step_row:(i+1)*step_row,
j*step_col:(j+1)*step_col]    # extract cur cell
            last_hist[i, j, :] = get_feature_from_wordmap(opts,
cur_cell)   # store the hist of cur cell
    hist_all = np.append(last_hist.flatten() * weights[L], hist_all)

    # calculate layers(<L) using the results in the previous layer
    pre_hist = np.copy(last_hist)
    for layer in range(L - 1, -1, -1):
        cur_hist = np.zeros((2 ** layer, 2 ** layer, K))
        for i in range(2 ** layer):
            for j in range(2 ** layer):
                cur_hist[i, j, :] = np.sum(pre_hist[i * 2:(i + 1) * 2,
j * 2:(j + 1) * 2, :], axis=(0, 1))
        hist_all = np.append(cur_hist.flatten() * weights[layer],
hist_all)
        pre_hist = cur_hist

    hist_all = hist_all/sum(hist_all)
    return hist_all
```

Q2.3

```
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 -----
    mins = np.minimum(word_hist, histograms)
    return np.sum(mins, axis=1)
```

Q2.4

```
def get_image_feature(args):
    '''
    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*(4^L-1)/3)
    '''

    # ----- TODO -----
    opts, img_path, dictionary = args
    img_path = join(opts.data_dir, img_path)
    img = Image.open(img_path)
    img = np.array(img).astype(np.float32) / 255
    wordmap = visual_words.get_visual_words(opts, img, dictionary)
    feature = get_feature_from_wordmap_SPM(opts, wordmap)
    save_path = 'tmp_feature' + str(img_path.split('/')[-1][:-4])
    np.save(save_path, feature)

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 -----
    p = multiprocessing.Pool(processes=n_worker)
    list_args = [(opts, img_path, dictionary) for img_path in
train_files]
    p.map(get_image_feature, list_args)
    for i in range(len(train_files)):
        # e.g. 'aquarium/sun_asgtepdmsxsrqqvy.jpg' --
'tmp_feature_sun_asgtepdmsxsrqqvy.npy'
        cur_path = 'tmp_feature' + str(train_files[i].split('/')[-
1][:-4]) + '.npy'
        cur = np.load(cur_path)
        os.remove(cur_path)
        if i == 0:
            features = np.array(cur)
            continue
        features = np.concatenate((features, cur), axis=0)

    ## 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,
    )
```

Q2.5

Confusion matrix:

```
[[33  1  2  0  3  2  4  5]
 [ 1 33  9  1  2  0  1  3]
 [ 2  9 30  0  0  2  1  6]
 [ 1  4  2 34  8  0  1  0]
 [ 1  4  8  9 20  5  0  3]
 [ 6  0  4  2  1 32  4  1]
 [ 3  1  2  0  5  9 28  2]
 [ 0  5  6  1  2  4  0 32]]
```

Accuracy: 0.605

60.5%

Q2.6

We can notice that the class with the least prediction precision is laundromat class. It only has 20 samples being predicted correctly, which is the lowest among all classes. And the most likely other wrong classes that bags-of-words approach predicts for this class are highway and kitchen, having the number of 8 and 9, respectively. The reason why laundromat class is more difficult to predict may be the edges and features in laundromat scenes are not as evident as other scenes. The main feature in laundromat is washing machine, which is less easy to detect than other features in aquarium and desert.

For example, in this picture above which is in the train samples of laundromat class, two persons, lights and plants also appear besides washing machines. So the main edge features can be weakened. We can also find many other similar photos in this train sets, so one way to improve prediction accuracy is to choose the photos with clear features.

Similarly, the other classes with low accuracy, such as waterfall, also has these kind of problems. On the contrary, classes like aquarium, desert and windmill are predicted more accurately because of their obvious features and better edges in train photos.

Q3.1

| filter_scales | K | alpha | L | Accuracy |
|---|---|---|---|---|
| [1, 2] | 10 | 25 | 3 | 60.5% |
| [1, 2, 4] | 10 | 25 | 3 | 54.0% |
| [1, 2] | 15 | 25 | 3 | 62.0% |
| [1, 2] | 15 | 50 | 3 | 59.0% |
| [1, 2] | 15 | 50 | 4 | 63.0% |
| [1, 2] | 15 | 50 | 5 | 56.5% |
| [1, 2] | 15 | 75 | 4 | **64.7%** |
| [1, 2, 4] | 15 | 75 | 4 | 60.5% |
| [1, 2] | 20 | 50 | 4 | 62.5% |
| [1, 2] | 20 | 75 | 4 | 59.75% |

In the table above, I first add filter scales from [1, 2] to [1, 2, 4], and the accuracy decreases because of low K, alpha and L I have set, they can't extract enough features from images. So I tune back filter scales and increase K from 10 to 15, which turns out accuracy increases because more clusters can be formed. Then I increase L by one but it seems L=5 is a bit too much of its weights as 2 considering only selecting 50 pixels from filter responses. So I tune alpha to 75 and accuracy increases to 64.7%, just around our goal 65%.

After reaching our goal, I also tried increasing filter scales and K but it turns out the final accuracy result never surpass our goal 65%.

Q3.2

(1) what you did:

I tried to change the weights of spatial pyramids matching and print their corresponding accuracy. I tuned the first two layer weights from both 1/4 to 1/8 + 1/4, or both 1/8. And I also tried to change the multiply between layers from 2 to 4.

(2) what you expected would happen:

I expected to see better accuracy with the same other parameters compared with original weights. Because I think if I set the start weights small, I can use more layers to compute.

(3) what actually happened:

The results turned out that all accuracy results under my new weights setting were worse than original settings. For example, the accuracy produced by setting 1/8 and 1/4 to the first two layers was only 58.8%, which is lower than 60.5%. It seemed like the approach that setting the first two weights as 1/4 then double them for each new layer, is really a precious experience gaining from many empirical experiments.

In terms of I just changed the weights, custom.py is still main.py. All the difference is the initial process of weights array in the function get_feature_from_wordmap_SPM:

```
weights = np.array([0.25]*(L+1))
for i in range(2, L+1):
    weights[i] = weights[i-1]*2
```