16-720

Shrinivas Ramasubramanian
shrinivr@andrew.cmu.edu
Assignment 1

September 2024

Q1.1.1 (5 points): What properties do each of the filter functions pick up? (See Fig 3) Try to group the filters into broad categories (e.g., all the Gaussians). Why do we need multiple scales of filter responses? Answer in your write-up.

Sol:

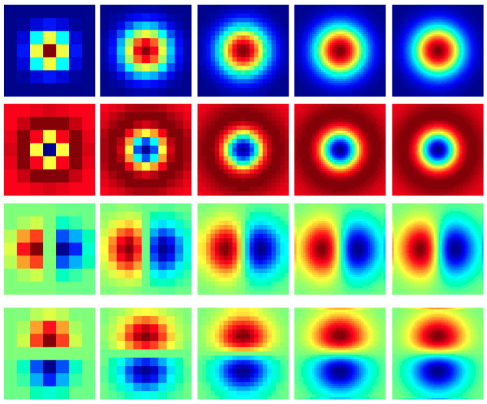


Figure 1: Multi-scale filter bank

- **Row 1**: This is the Gaussian filter that smooths the image and reduces noise from high frequencies. It preserves smooth, non-changing regions and suppresses or filters out regions with a high degree of variation. The Gaussian filter is expressed as:

$$G(x,y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

  where σ controls the scale of smoothing. A larger σ results in more aggressive smoothing, capturing broader structures while suppressing finer details.

- **Row 2**: This is the Laplace of Gaussian filter, which detects edges in the image. This omni-directional, detecting edges in all directions. The Laplacian filter is expressed as follows and The LoG filter is sensitive to areas of rapid intensity change, highlighting edges and transitions between different regions in the image:

$$\text{LoG}(x,y) = -\frac{1}{\pi\sigma^4}\left(1 - \frac{x^2+y^2}{2\sigma^2}\right) \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

- **Row 3**: This filter is the derivative of the Gaussian in the x-direction, often referred to as a Gaussian derivative filter. It highlights horizontal edges by computing the rate of intensity change in the horizontal direction. The derivative of the Gaussian in the x-direction is given by:

$$\frac{\partial G(x,y)}{\partial x} = -\frac{x}{2\pi\sigma^4} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

  This filter enhances horizontal edges while maintaining smoothness due to the Gaussian component.

- **Row 4**: This is the derivative of the Gaussian in the y-direction, detecting vertical edges by computing intensity changes in the vertical direction. This filter emphasizes vertical edges, similar to the x-derivative but oriented to capture vertical transitions. The derivative of the Gaussian in the y-direction is expressed as:

$$\frac{\partial G(x,y)}{\partial y} = -\frac{y}{2\pi\sigma^4} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

Q1.1.2 (10 points): For the code, loop through the filters and the scales to extract responses. Since color images have 3 channels, you are going to have a total of $3F$ filter responses per pixel if the filter bank is of size F. Note that in the given dataset, there are some gray-scale images. For those gray-scale images, you can simply duplicate them into three channels. Then output the result as a $3F$ channel image. Complete the function `visual_words.extract_filter_responses(opts, img)` and return the responses as `filter_responses`. We have provided you with template code, with detailed instructions commented inside.

Remember to check the input argument image to make sure it is a floating point type with range $[0, 1]$, and convert it if necessary. Be sure to check the number of input image channels and convert it to 3-channel if it is not. Before applying the filters, use the function `skimage.color.rgb2lab()` to convert your image into the Lab color space, which is designed to more effectively quantify color differences with respect to human perception. (See here for more information.)

If the input image is an $M \times N \times 3$ matrix, then `filter_responses` should be a matrix of size $M \times N \times 3F$. Make sure your convolution function call handles image padding along the edges sensibly.

Apply all 4 filters with at least 3 scales on `aquarium/sun_aztvjgubyrgvirup.jpg`, and visualize the responses as an image collage as shown in Fig 4.

The included helper function `util.display_filter_responses` (which expects a list of filter responses with those of the Lab channels grouped together with shape $M \times N \times 3$) can help you to create the collage. Submit the collage of images in your write-up.

Sol:

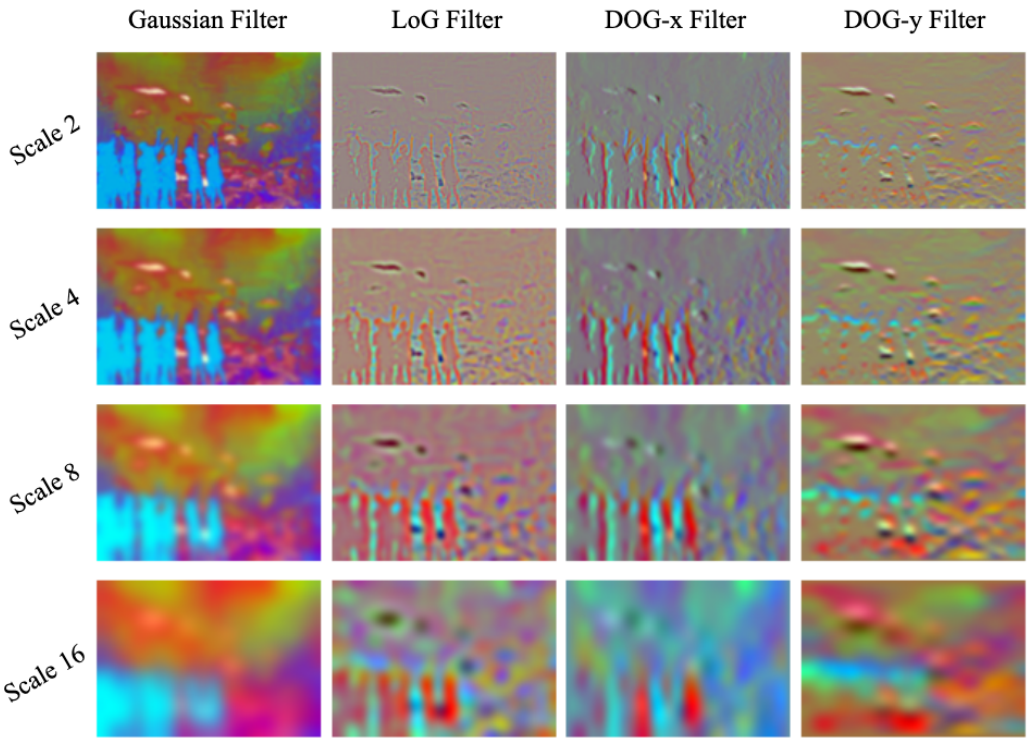


Figure 2: Results of applying multi-scale filter banks on an image

Creating Visual Words

You will now create a dictionary of visual words from the filter responses using k-means. After applying k-means, similar filter responses will be represented by the same visual word. You will use a dictionary with a fixed size. Instead of using all of the filter responses (which might exceed the memory capacity of your computer), you will use responses at α random pixels. The location of these random pixels can differ between each training image. If there are T training images, then you should collect a matrix of filter responses over all the images that is $\alpha T \times 3F$, where F is the filter bank size.

Then, to generate a visual words dictionary with K words (opts.K), you will cluster the responses with k-means using the function `sklearn.cluster.KMeans` as follows:

```
kmeans = sklearn.cluster.KMeans(n_clusters=K).fit(filter_responses)
dictionary = kmeans.cluster_centers_
```

If you like, you can pass the `njobs` argument into the `KMeans()` object to utilize parallel computation.

Q1.2 (10 points):

Write the functions

- `visual_words.compute_dictionary(opts, n_worker)`
- `visual_words.compute_dictionary_one_image(args)` (optional, multi-processing)

Given a dataset, these functions generate a dictionary. The overall goal of `compute_dictionary()` is to load the training data, iterate through the paths to the image files to read the images, and extract αT filter responses over the training files, and call k-means. This can be slow to run; however, the images can be processed independently and in parallel. Inside `compute_dictionary_one_image()`, you should read an image, extract the responses, and save to a temporary file. Here `args` is a collection of arguments passed into the function.

Inside `compute_dictionary()`, you should load all the training data and create subprocesses to call `compute_dictionary_one_image()`. After all the subprocesses finish, load the temporary files back, collect the filter responses, and run k-means. A list of training images can be found in `data/train_files.txt`.

Finally, execute `compute_dictionary()`, and go do some push-ups while you wait for it to complete. If all goes well, you will have a file named `dictionary.npy` that contains the dictionary of visual words. If the clustering takes too long, reduce the number of clusters and samples. You can start with a tiny subset of training images for debugging.

Sol:

```
def compute_dictionary_one_image(args):
    '''
    Extracts filter responses for a single image.

    [input]
    * args     : tuple containing (opts, img_file)

    [output]
    * filter_responses: numpy.ndarray of shape (H*W, 3F)
    '''
    opts, img_file = args
    img = np.array(Image.open(join(opts.data_dir, img_file)).convert('RGB')).astype(np.
        float32) / 255
    filter_responses = extract_filter_responses(opts, img)
    # reshape the filter responses to a 2D array and select an alpha random subset of
        responses
```

```
    filter_responses = filter_responses.reshape(-1, filter_responses.shape[-1])
    return filter_responses
```

Listing 1: Function to Compute Dictionary for One Image

```
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()

    # Use multiprocessing to parallelize the computation of filter responses
    with multiprocessing.Pool(n_worker) as pool:
        all_responses_ = list(tqdm(pool.imap(compute_dictionary_one_image, [(opts,
            img_file) for img_file in train_files]), total=len(train_files), desc="
            Processing images"))

    # select alpha points at random
    alpha = opts.alpha
    all_responses = [response[np.random.choice(response.shape[0], alpha, replace=False),
        :] for response in all_responses_ if response is not None]

    # Stack all responses into a single 2D array
    all_responses = np.vstack(all_responses)


    # Perform k-means clustering to create the dictionary of visual words
    kmeans = KMeans(n_clusters=K, verbose=True).fit(all_responses)
    dictionary = kmeans.cluster_centers_

    # Save the dictionary to disk
    np.save(join(out_dir, 'dictionary.npy'), dictionary)
    return
```

Listing 2: Function to Compute Visual Words Dictionary

Q1.3 (10 points): We want to map each pixel in the image to its closest word in the dictionary. Complete the following function to do this:

`visual_words.get_visual_words(opts, img, dictionary)`

and return `wordmap`, a matrix with the same width and height as `img`, where each pixel in `wordmap` is assigned the closest visual word of the filter response at the respective pixel in `img`. We will use the standard Euclidean distance to do this; to do this efficiently, use the function `scipy.spatial.distance.cdist()`. Some sample results are shown in Fig. 5.

Visualize wordmaps for three images. Include these in your write-up, along with the original RGB images. Include some comments on these visualizations: do the "word" boundaries make sense to you? The visualizations should look similar to the ones in Fig. 5. Don't worry if the colors don't look the same, newer `matplotlib` might use a different color map.

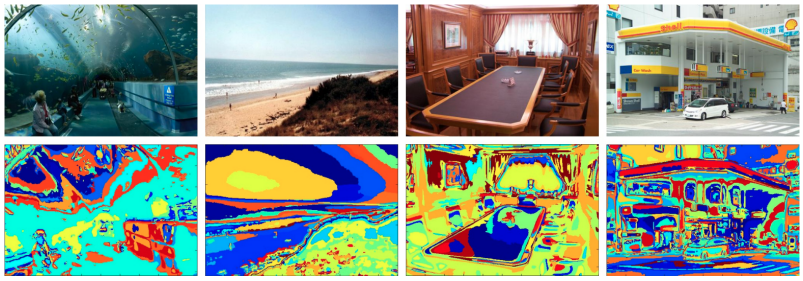


Figure 3: Visual words over images. You will use the spatially unordered distribution of visual words in a region (a bag of visual words) as a feature for scene classification, with some coarse information provided by spatial pyramid matching [4].

Sol: The method captures smooth regions as blobs, indicating areas of uniform intensity or gradual variation. Textures, on the other hand like in the waters of waterfall have a pattern which is nicely captured by the yellow blob that correspond to vertical edges (probably). Additionally, it seems that color plays a role in clustering, influencing the way different regions are grouped based on their hue and intensity distribution.

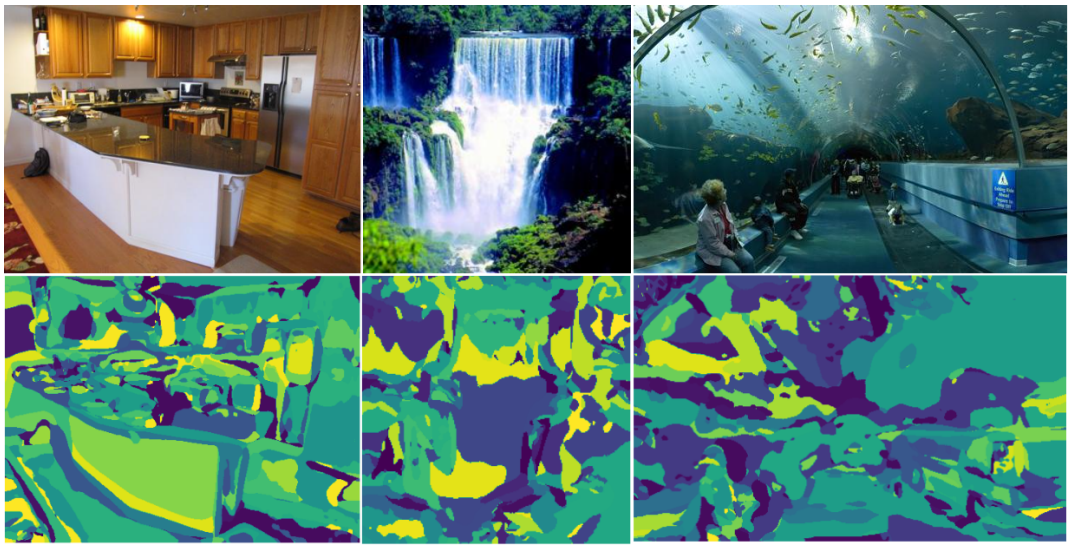


Figure 4: Visualisation of visual words over images

```python
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)

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

    # ----- TODO -----
    filter_response = extract_filter_responses(opts, img)
    word_map = np.zeros((img.shape[0], img.shape[1]))
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            word_map[i, j] = np.argmin(np.linalg.norm(filter_response[i, j] - dictionary
                , axis=1))
    return word_map
```

Listing 3: Python code for computing visual words mapping

Q2.1 (10 points) Write the function:

`visual_recog.get_feature_from_wordmap(opts, wordmap)`

that extracts the histogram using `numpy.histogram()` of visual words within the given image (i.e., the bag of visual words). The function should return `hist`, an L_1-normalized histogram of length equal to the dictionary size. The L_1 normalization ensures that the sum of the histogram values equals 1. You may wish to load a single visual word map, visualize it, and verify that your function is working correctly before proceeding.

Sol:

```python
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 -----
    hist = np.zeros(K)
    for i in range(K):
        hist[i] = np.sum(wordmap == i)
    # hist = hist / np.sum(hist)
    return hist
```

Listing 4: Function to compute histogram of visual words

2.2 Multi-resolution: Spatial Pyramid Matching

A bag of words is simple and efficient, but it discards information about the spatial structure of the image, which is often valuable. One way to alleviate this issue is to use spatial pyramid matching [**?**]. The general idea is to divide the image into a small number of cells, and concatenate the histogram of each of these cells to the histogram of the original image, with a suitable weight.

Here we will implement a popular scheme that divides the image into $2^l \times 2^l$ cells where l is the layer number. We treat each cell as a small image and count how often each visual word appears. This results in a histogram for every single cell in every layer. To represent the entire image, we concatenate all the histograms together after normalization by the total number of features in the image. If there are $L+1$ layers and K visual words, the resulting vector has dimension $K \sum_{l=0}^{L} 4^l = K\frac{4^{L+1}-1}{3}$.

Now comes the weighting scheme. Note that when concatenating all the histograms, histograms from different levels are assigned different weights. Typically (and in the original work [**?**]), a histogram from layer l gets half the weight of a histogram from layer $l+1$, with the exception of layer 0, which is assigned a weight equal to layer 1. A popular choice is to set the weight of layers 0 and 1 to 2^{-L}, and set the rest of the weights to 2^{l-L-1}. For example, in a three-layer spatial pyramid, $L = 2$ and weights are set to 1/4, 1/4, and 1/2 for layers 0, 1, and 2, respectively (see Fig. 6 for an illustration of a spatial pyramid). Note that the L1 norm (absolute values of all dimensions summed together) for the final vector is 1.

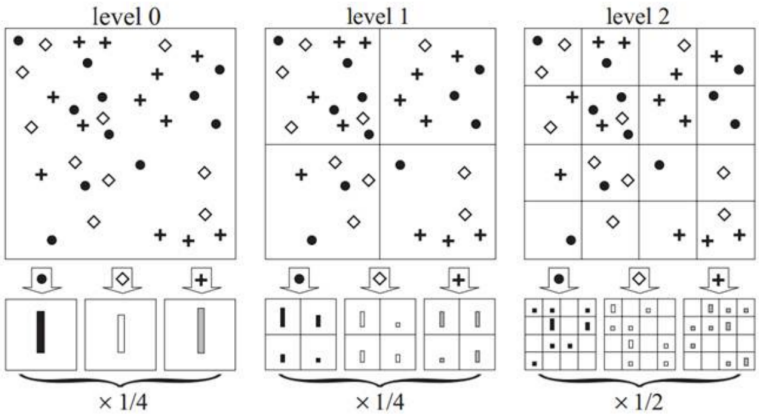


Figure 5: Spatial Pyramid Matching: From [4]. Toy example of a pyramid for L = 2. The image has three visual words, indicated by circles, diamonds, and crosses. We subdivide the image at three different levels of resolution. For each level of resolution and each channel, we count the features that fall in each spatial bin. Finally, weight each spatial histogra

Q2.2 (15 points)

Create the following function that forms a multi-resolution representation of the given image:

`visual_recog.get_feature_from_wordmap_SPM(opts, wordmap)`

You need to specify the layers of the pyramid (L) in `opts.L`. As output, the function will return `hist_all`, a vector that is L1 normalized.

One small hint for efficiency: a lot of computation can be saved if you first compute the histograms of the finest layer, because the histograms of coarser layers can then be aggregated from finer ones. Make sure you normalize the histogram after aggregation.

Sol:

```python
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 \((K \cdot \frac{4^L - 1}{3})\)
    '''

    K = opts.K
    L = opts.L
    H, W = wordmap.shape
    hist_all = []

    for l in range(L + 1):
        num_cells = 2 ** l
        cell_H = H // num_cells
        cell_W = W // num_cells
        hist_level = []

        for i in range(num_cells):
            for j in range(num_cells):
                cell_wordmap = wordmap[i * cell_H:(i + 1) * cell_H, j * cell_W:(j + 1) *
                    cell_W]
                hist = get_feature_from_wordmap(opts, cell_wordmap)
                hist_level.append(hist)

        hist_level = np.concatenate(hist_level)
        weight = 2.0 ** (l - L) if l != 0 else 2.0 ** (-L)
        hist_all.append(hist_level * weight)

    hist_all = np.concatenate(hist_all)
    hist_all = hist_all / np.sum(hist_all)  # Normalize the histogram

    return hist_all
```

Listing 5: Function to compute histogram of visual words using spatial pyramid matching

Q2.3 (10 points)

Create the function:

`visual_recog.distance_to_set(word_hist, histograms)`

where `word_hist` is a vector of size $\frac{K \cdot 4^{(L+1)}-1}{3}$ and `histograms` is a matrix of size $T \times \frac{K \cdot 4^{(L+1)}-1}{3}$ containing T features from T training samples concatenated along the rows.

This function computes the histogram intersection similarity between `word_hist` and each training sample, returning a vector of length T. The function should return one minus the above quantity as a distance measure (distance is the inverse of similarity). Since this is called every time you look up a classification, you will want this to be fast! (Doing a for-loop over tens of thousands of histograms is a bad idea.)

Sol:

```python
def distance_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]
    * hist_dist: numpy.ndarray of shape (N)
    '''

    # ----- TODO -----
    # computes the distance between 2 histograms. This function computes the histogram
    # intersection similarity between word_hist and each training sample
    # euclidean distance between word_hist and each training sample
    hist_dist = np.sum(np.minimum(word_hist, histograms), axis=1)
    return hist_dist
```

Listing 6: Function to compute distance between a histogram of visual words and training image histograms

2.4 Building A Model of the Visual World

Now that we've obtained a representation for each image and defined a similarity measure to compare two spatial pyramids, we want to put everything together.

Simple I/O code has been provided in the respective functions, which include loading the training images specified in `data/train_files.txt` and the filter bank and visual word dictionary from `dictionary.npy`, as well as saving the learned model to `trained_system.npz`. Specifically, in `trained_system.npz`, you should have:

1. `dictionary`: your visual word dictionary.
2. `features`: an $N \times \frac{K \cdot 4^{(L+1)}-1}{3}$ matrix containing all of the histograms of the N training images in the data set.
3. `labels`: an N vector containing the labels of each of the training images. (`features[i]` will correspond to label `labels[i]`).
4. `SPM_layer_num`: the number of spatial pyramid layers you used to extract the features for the training images.

Do not use the testing images for training!

The table below lists the class names that correspond to the label indices:

0	1	2	3	4	5	6	7
aquarium	desert	highway	kitchen	laundromat	park	waterfall	windmill

Q2.4 (15 points): Implement the function

```
visual_recog.build_recognition_system()
```

that produces `trained_system.npz`. You may include any helper functions you write in `visual_recog.py`.

Implement

```
visual_recog.get_image_feature(opts, img_path, dictionary)
```

that loads an image, extracts the word map from the image, computes the SPM, and returns the computed feature. Use this function in your `visual_recog.build_recognition_system()`.

Sol:

```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 \((K \cdot \frac{4^L - 1}{3})\)
    '''

    # ----- TODO -----
    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)
    return feature

from concurrent.futures import ProcessPoolExecutor
import numpy as np
from os.path import join
from tqdm import tqdm

# External function to process a single image
def process_image(file_idx, opts, train_files, dictionary, data_dir):
    try:
        return get_image_feature(opts, join(data_dir, train_files[file_idx]), dictionary
            )
    except Exception as e:
        print(f"Error processing image {file_idx}: {e}")
        return None

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)
    '''
    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'))

    # Initialize arrays to store features and labels
    features = []

    # Use ProcessPoolExecutor to parallelize the processing of images
    with ProcessPoolExecutor(max_workers=n_worker) as executor:
        futures = [executor.submit(process_image, i, opts, train_files, dictionary,
            data_dir) for i in range(len(train_files))]
        for future in futures:
            result = future.result()
            if result is not None:
                features.append(result)

    # Convert lists to numpy arrays
    features = np.array(features)
    np.savez_compressed(join(out_dir, 'trained_system.npz'),
        features=features,
        labels=train_labels,
        dictionary=dictionary,
        SPM_layer_num=SPM_layer_num,
    )
    print("Recognition system built and saved.")
```

Listing 7: Function to extract spatial pyramid matching feature

2.5 Qualitative Evaluation

Qualitative evaluation is all well and good (and very important for diagnosing performance gains and losses), but we want some hard numbers.

Load the test images and their labels, and compute the predicted label of each one. That is, compute the test image's distance to every image in the training set, and return the label of the closest training image. To quantify the accuracy, compute a confusion matrix C. In a classification problem, the entry $C(i, j)$ of a confusion matrix counts the number of instances of class i that were predicted as class j. When things are going well, the elements on the diagonal of C are large, and the off-diagonal elements are small. Since there are 8 classes, C will be 8×8. The accuracy, or percent of correctly classified images, is given by the trace of C divided by the sum of C:

Q2.5 (10 points) Implement the function

`visual_recog.evaluate_recognition_system()`

that tests the system and outputs the confusion matrix. Include the confusion matrix and your overall accuracy in your write-up. This does not have to be formatted prettily: if you are using LaTeX, you can simply copy/paste it into a `verbatim` environment.

Sol: The observed accuracy was 58% and the hyperparameters were: $K = 150$, $\alpha = 150$, filter-scales = [1, 2, 4, 8] $L = 3$. Other confusion matrices at different hparams are provided in code repo.
The confusion matrix is as follows:

```
34,1,0,2,4,0,1,8
0,37,3,4,2,0,2,2
0,3,30,1,2,3,1,10
5,0,5,24,8,3,2,3
4,4,3,12,22,3,1,1
3,1,7,1,2,31,3,2
3,1,1,1,3,10,29,2
2,6,9,2,1,4,1,25
```

```
1
2  def process_test_image(params):
3      i, test_file, test_opts, dictionary, features, labels, test_labels, data_dir =
           params
4      img_path = join(data_dir, test_file)
5      img = Image.open(img_path)
6      img = np.array(img).astype(np.float32) / 255
7      wordmap = visual_words.get_visual_words(test_opts, img, dictionary)
8      test_feature = get_feature_from_wordmap_SPM(test_opts, wordmap)
9
10     distances = distance_to_set(test_feature, features)
11     predicted_label = labels[np.argmax(distances)]
12     true_label = test_labels[i]
13
14     return predicted_label, true_label
15
16
17 def evaluate_recognition_system(opts, n_worker=1):
18     '''
19     Evaluates the recognition system for all test images and returns the confusion
           matrix.
20
21     [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.deepcopy(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)

    # Load trained system information
    features = trained_system['features']
    labels = trained_system['labels']
    SPM_layer_num = trained_system['SPM_layer_num']

    # Initialize confusion matrix
    num_classes = len(np.unique(test_labels))
    conf = np.zeros((num_classes, num_classes))

    # Prepare input parameters for each image processing task
    params = [(i, test_files[i], test_opts, dictionary, features, labels, test_labels,
        data_dir)
              for i in range(len(test_files))]

    # Parallel processing of image evaluation
    with ProcessPoolExecutor(max_workers=n_worker) as executor:
        results = list(tqdm(executor.map(process_test_image, params),
                            total=len(test_files),
                            desc="Evaluating images"))

    # Aggregate results and update confusion matrix
    correct_predictions = 0
    for predicted_label, true_label in results:
        conf[true_label, predicted_label] += 1
        if predicted_label == true_label:
            correct_predictions += 1

    # Dump predicted labels, true labels, and test image names to a single text file
    with open(join(out_dir, 'predicted_labels.txt'), 'w') as f:
        for i in range(len(results)):
            f.write(f"{test_files[i]} {results[i][0]} {results[i][1]}\n")

    # Calculate accuracy
```

```
    accuracy = correct_predictions / len(test_labels)

    return conf, accuracy
```

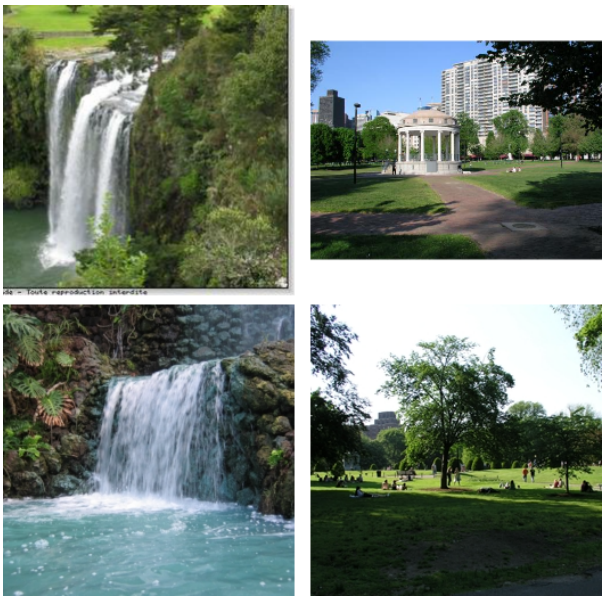
Listing 8: Function to evaluate the recognition system and return the confusion matrix

2.6 Find the Failures

There are some classes or samples that are more difficult to classify than others when using the bags-of-words approach. Consequently, these samples may be misclassified into incorrect categories.

Q2.6 (5 points): In your writeup, list some of these challenging classes or samples, and discuss why they are more difficult to classify compared to the others.

Sol: Distinguishing between parks and waterfalls can be challenging due to the presence of vertical edges in both types of scenes. In park images, vertical edges are often created by structures such as trees, fences. These edges correspond to the trunks of trees and vertical elements within the park. Similarly, in waterfall images, vertical edges are formed by the cascading flow of water, which produces a series of vertical patterns. The similarity in these vertical edges can lead to difficulties in classification. Plus the existence of greenery around waterfalls and also in parks makes them easy to get confused.



(a) Images confused as either park or waterfall

(b) Images confused as kitchen or laundry room

Figure 6: Examples of misclassified images using the bag-of-words approach

The confusion between kitchens and laundromats in image classification usually happens because they look pretty similar in certain ways. Both places tend to have smooth surfaces, like countertops in kitchens or washing machine doors in laundromats, which can be mistaken for each other. Washing machines in laundromats often look like kitchen cabinets, making it easy for a classifier to mix them up. Plus, the color schemes inside kitchens and laundromats are often similar, which just adds to the confusion.

3 Improving Performance

3.1 Hyperparameter Tuning

Now that we have a fully functional recognition system and evaluation framework, it's time to improve its performance. In practice, a model often doesn't perform optimally right out of the box. Therefore, it's crucial to understand how to fine-tune a visual recognition system to suit the task at hand.

Q3.1 (15 points): Tune the system you have built to achieve approximately 65% accuracy on the provided test set (`data/test files.txt`). A list of hyperparameters to be tuned is provided below, all of which can be found in `opts.py`. Include a table of ablation studies showcasing at least 3 major steps, such as: "Changing parameter X to Y results in accuracy Z%". Additionally, describe why you believe altering a particular parameter will increase or decrease overall performance in the table.

- `Filter scales:` A list of filter scales used for extracting filter responses.
- `K:` The number of visual words, which also represents the size of the dictionary.
- `Alpha:` The number of sampled pixels from each image when creating the dictionary.
- `L:` The number of spatial pyramid layers used in feature extraction.

Sol:

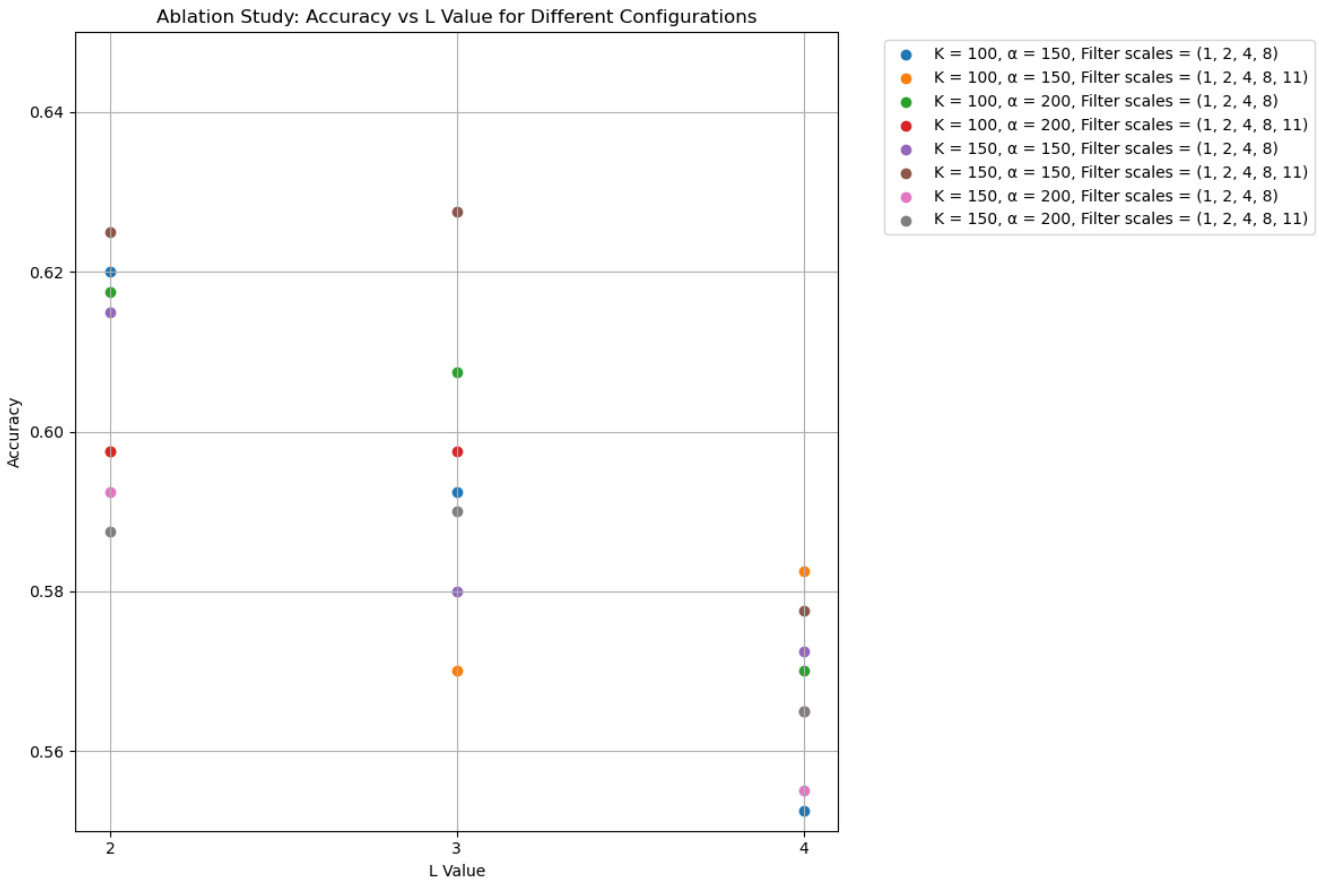


Figure 7: Ablation plot for ablation over K, α, L, filter-size

In this ablation study, we evaluate the impact of various hyperparameters on the performance of our recognition system. The parameters under investigation include the number of visual words K, the number of spatial pyramid layers L, the scaling factor α, and the filter scales, represented as $(1, 2, 4, 8)$. The highest accuracy achieved across all configurations was 0.6275 for the setup with $K = 150$, $L = 3$, and $\alpha = 150$. This suggests that a moderate number of visual words, combined with a well-chosen layer structure and scaling factor, can enhance the model's discriminative capabilities.

Analyzing the trends: - Increasing K generally leads to improved accuracy, particularly evident in configurations where K is set to 150. - The number of layers L appears to have a nuanced effect; while $L = 3$ yielded some of the best results, configurations with $L = 2$ and $L = 4$ also demonstrated competitive performance. - The scaling factor α affects performance, with $\alpha = 150$ consistently yielding better results compared to $\alpha = 200$.

The results indicate that careful tuning of K, L, and α is essential for optimizing performance. Future experiments may explore additional configurations and further analyze the interplay between these parameters.