16-720A Computer Vision: Homework 1
Spatial Pyramid Matching for Scene Classification

Instructors: Srinivasa Narasimhan, David Held
TAs: Brian Okorn, Chen-Hsuan Lin, Yifan Xing, Sree Harsha Kalli, Siddarth Malreddy, Prakhar Pradeep Naval, Khushi Gupta, Shangxuan Wu, Bala Siva Jujjavarapu, Jingyan Wang, Yashasvi Agrawal
Due: September 27 at 11:59pm

Figure 1: **Scene Classification:** Given an image, can a computer program determine where it was taken? In this homework, you will build a representation based on bags of visual words and use spatial pyramid matching for classifying the scene categories.

Instructions/Hints

1. Please pack your system and write-up into a single file named <**AndrewId**>**.zip**, see the complete submission checklist at the end.
2. All questions marked with a **Q** require a submission.
3. **For the implementation part, please stick to the headers, variable names, and file conventions provided. You will lose marks if you don't.**
4. **Start early!** This homework will take a long time to complete.
5. **Attempt to verify your implementation as you proceed:** If you don't verify that your implementation is correct on toy examples, you will risk having a huge mess when you put everything together.
6. Try to use relative paths with respect to the working directory.
7. If you have any questions or need clarifications, please post in Piazza or visit the TAs during the office hours.

Overview

The bag-of-words (BoW) approach, which you learned about in class, has been applied to a myriad of recognition problems in computer vision. For example, two classic ones are object recognition [**?**, **?**] and scene classification [**?**, **?**][1].

Beyond that, the BoW representation has also been the subject of a great deal of study aimed at improving it, and you will see a large number of approaches that remain in the spirit of bag-of-words but improve upon the traditional approach which you will implement here. For example, two important extensions are pyramid matching [**?**, **?**] and feature encoding [**?**].

An illustrative overview of the homework is shown in Figure. 2. In Section. 1, we will build the visual words from the training set images. With the visual words, *i.e.* the dictionary, in Section. 2 we will represent an image as a visual-word vector. Then the comparison between images is realized in the visual-word vector space. Finally, we will build a scene recognition system based on the visual bag-of-words approach to classify a given image into 8 types of scenes.

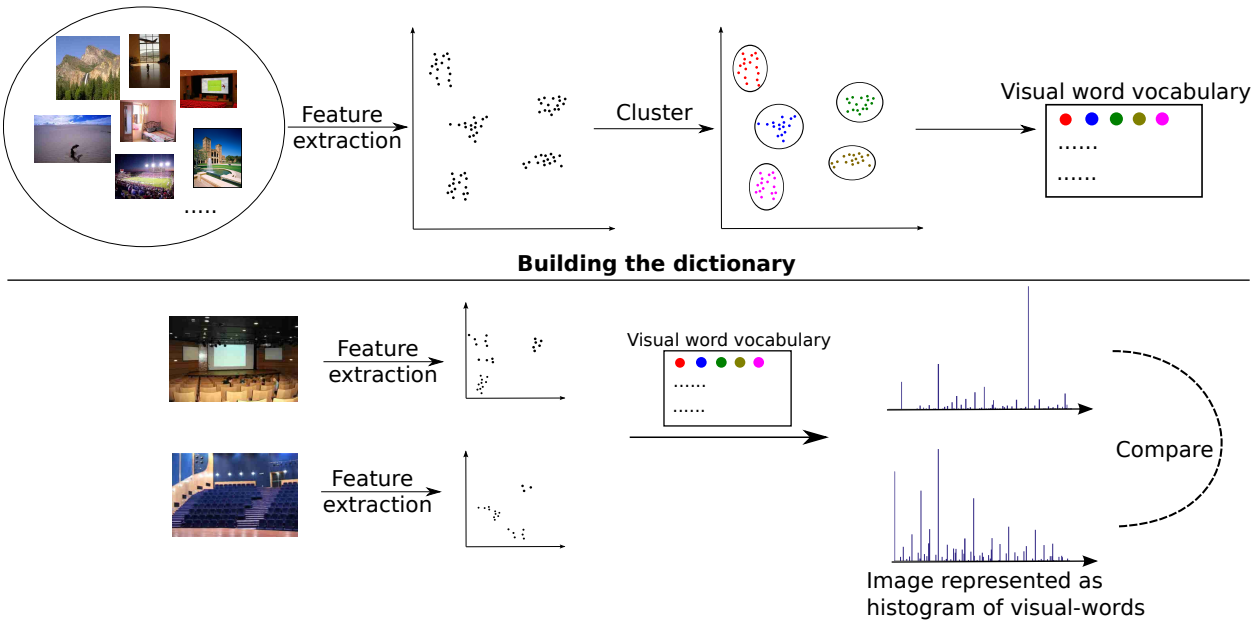

Figure 2: An overview of the bags-of-words approach to be implemented in the homework. Given the training set of images, the visual features of the images are extracted. In our case, we will use the filter responses of the pre-defined filter bank as the visual features. The visual words, *i.e.* dictionary, are built as the centers of clusterings of the visual features. During recognition, the image is first represented as a vector of visual words. Then the comparison between images is realized in the visual-word vector space. Finally, we will build a scene recognition system that classifies the given image into 8 types of scenes

What you will be doing: You will implement a scene classification system that uses the bag-of-words approach with its spatial pyramid extension. The paper that introduced the pyramid matching kernel [**?**] is:

> K. Grauman and T. Darrell. *The Pyramid Match Kernel: Discriminative Classification with Sets of Image Features.* ICCV 2005. `http://www.cs.utexas.edu/~grauman/papers/grauman_darrell_iccv2005.pdf`

Spatial pyramid matching [**?**] is presented at:

[1]This homework aims at being largely self-contained; however, reading the listed papers (even without trying to truly understand them) is likely to be helpful.

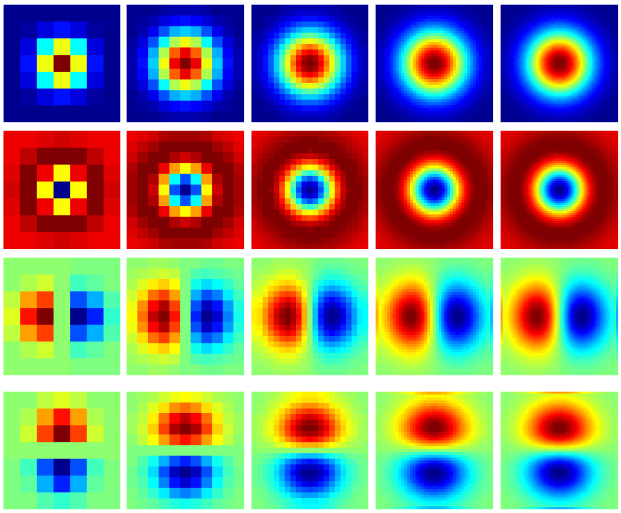

Figure 3: The provided multi-scale filter bank

S. Lazebnik, C. Schmid, and J. Ponce, *Beyond Bags of Features: Spatial Pyramid Matching for Recognizing Natural Scene Categories*, CVPR 2006. `http://www.di.ens.fr/willow/pdfs/cvpr06b.pdf`

You will be working with a subset of the SUN database[2]. The data set contains 1600 images from various scene categories like "auditorium", "desert" and "kitchen". And to build a recognition system, you will:

- first, take responses of a filter bank on images and build a dictionary of visual words;
- then, learn a model for the visual world based on the bag of visual words (with spatial pyramid matching [**?**]), and use nearest-neighbor to predict scene classes in a test set.

In terms of number of lines of code, this assignment is fairly small. However, it may take *a few hours* to finish running the baseline system, so make sure you start early so that you have time to debug things. Also, try **each component** on **a subset of the data set** first before putting everything together.

We provide you with a number of functions and scripts in the hopes of alleviating some tedious or error-prone sections of the implementation. You can find a list of files provided in Section 3.

1 Representing the World with Visual Words

We have provided you with a multi-scale filter bank that you will use to understand the visual world. You can create an instance of it with the following (provided) function:

```
[filterBank] = createFilterBank()
```

`filterBank` is a cell array[3], with the pre-defined filters in its entries. In our example, we are using 20 filters consisting of 4 types of filters in 5 scales.

Q1.0 (5 points): What properties do each of the filter functions (See Figure 3) pick up? You should group the filters into broad categories (*i.e.*, all the Gaussians). Answer in your write-up.

[2] `http://groups.csail.mit.edu/vision/SUN/`

[3] Look at MATLAB's documentation for more details, but `filterBank{i}` is a 2D matrix, and `filterBank{i}` and `filterBank{j}` are not necessarily the same size.

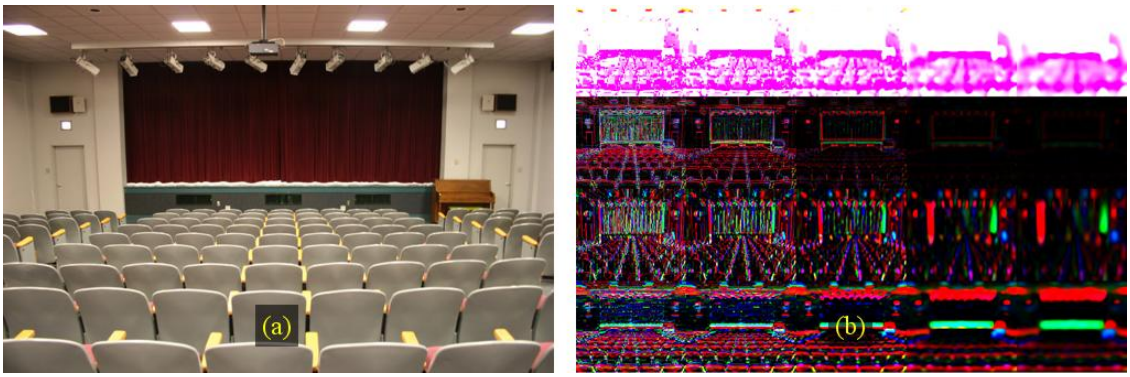

Figure 4: An input image and filter responses for all of the filters in the filter bank. (a) The input image of a gas station (b) The filter responses of Lab image corresponding to the filters in Figure. 3

1.1 Extracting Filter Responses

Q1.1 (10 points): We want to run our filter bank on an image by convolving each filter in the bank with the image and concatenating all the responses into a vector for each pixel. Use the `imfilter` command in a loop to do this. 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 duplicated them into three channels using the command `repmat`. Then output the result as a $3F$ channel image. Follow this function prototype

```
[filter_response] = extractFilterResponses(I, filterBank)
```

We have provided you with a template code with detailed instructions in it. You would be required to input a 3-channel RGB or gray-scale image and filter bank to get the responses of the filters on the image.

Remember to check the input argument `I` to make sure it is a floating point type and convert it if necessary. Use the included helper function `RGB2Lab` to convert your image into `Lab` space before applying the filters. If `I` is an $M \times N \times 3$ matrix, then `filter_response` 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 20 filters on a sample image, and visualize as a image collage (as shown in Figure 4). Submit the collage of 20 images in the write-up. (Hint: Use `montage` matlab function to build the collage).

1.2 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 fixed-size. Instead of using all of the filter responses (**that can exceed the memory capacity of your computer**), you will use responses at α random pixels[4]. If there are T training images, then you should collect a matrix `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, you will cluster the responses with k-means using the built-in MATLAB function `kmeans` as follows:

```
[~, dictionary] = kmeans(filter_responses, K, 'EmptyAction','drop');
```

Note that the output of the function `kmeans` is row-wise, *i.e.*, each row is an sample/cluster. In our following implementations, we will assume the dictionary matrix is column-wise. So you need to transpose `dictionary` after it is estimated from the `kmeans` function.

Q1.2 (10 points): You should write the following functions to generate a dictionary given a list of images.

```
[filterBank, dictionary] = getFilterBankAndDictionary(image_names)
```

[4]Try using `randperm`.

As an input, `getFilterBankAndDictionary` takes a cell array of strings containing the full path to an image (or relative path wrt the working directory). You can load each file by iterating from `1:length(image_names)`, and doing `imread(image_names{i})`. Generate the αT filter responses over the training files and call k-means. A sensible initial value to try for K is between 100 and 300, and for α is between 50 and 200, but they depend on your system configuration and you might want to play with these values.

Once you are done with `getFilterBankAndDictionary`, call the provided script `computeDictionary`, which will pass in the file names, and go get a coffee. If all goes well, you will have a `.mat` file named `dictionary.mat` that contains the filter bank as well as the dictionary of visual words. If all goes poorly, you will have an error message[5]. If the clustering takes too long, reduce the number of clusters and samples. If you have debugging issues, try passing in a small number of training files manually.

1.3 Computing Visual Words

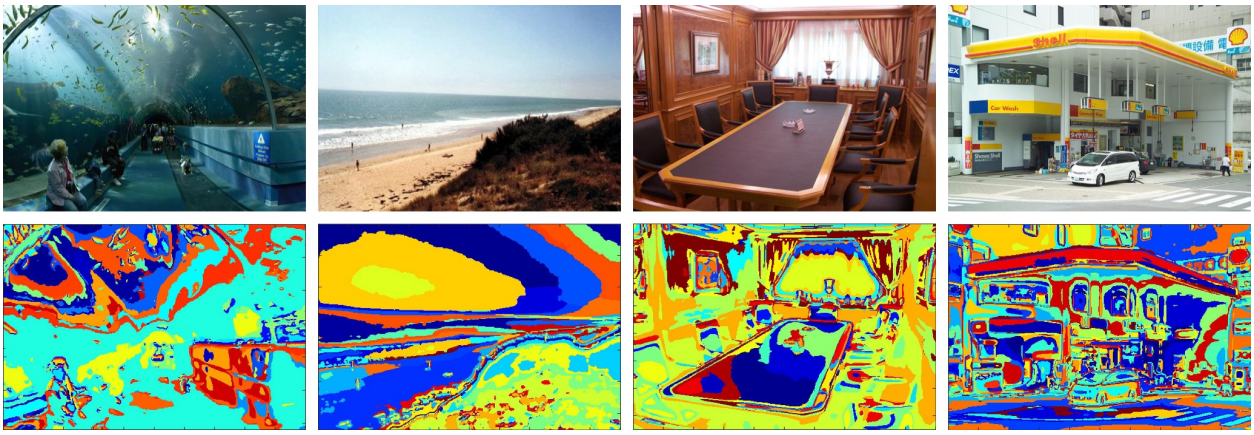

Figure 5: Visual words over images. You will use the spatially un-ordered distribution of visual words in a region (a bag of visual words) as a feature for scene classification, with some coarse information provided by spatial pyramid matching [**?**]

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

```
[wordMap] = getVisualWords(I, filterBank, dictionary)
```

`wordMap` is a matrix with the same width and height as I, where each pixel in `wordMap` is assigned the closest visual word of the filter response at the respective pixel in `I`. We will use the standard Euclidean distance to do this; to do this efficiently, use the MATLAB function `pdist2`. Some sample results are shown in Fig. 5.

Since this can be slow, we have provided a function `batchToVisualWords(numberOfCores)` that will apply your implementation of the function `getVisualWords` to every image in the training and testing set. This function will automatically[6] use as many cores as you tell it to use. For every image "`X.jpg`" in `dat/`, there will be a corresponding file named "`X.mat`" in the same folder containing the variable `wordMap`.

Visualize three wordmaps of three images from any one of the category and submit in the write-up along with their original RGB image. Also, provide your comments about the visualization. They should look similar to the ones in Figure 5. (Hint: use `imagesc` matlab function to visualize wordmap).

2 Building a Recognition System

We have formed a convenient representation for recognition. We will now produce a basic recognition system with spatial pyramid matching. The goal of the system is presented in Fig. 1: given an image, classify (colloquially, "name") the scene where the image was taken.

[5]Don't worry about "did-not-converge" errors.

[6]Interested parties should investigate `batchToVisualWords.m` and the MATLAB commands `matlabpool` and `parfor`.

Traditional classification problems follow two phases: training and testing. During training time, the computer is given a pile of formatted data (*i.e.*, a collection of feature vectors) with corresponding labels (*e.g.*, "desert", "kitchen") and then builds a model of how the data relates to the labels: "if green, then kitchen". At test time, the computer takes features and uses these rules to infer the label: *e.g.*, "this is green, so therefore it is kitchen".

In this assignment, we will use the simplest classification model: nearest neighbor. At test time, we will simply look at the query's nearest neighbor in the training set and transfer that label. In this example, you will be looking at the query image and looking up its nearest neighbor in a collection of training images whose labels are already known. This approach works surprisingly well given a huge amount of data, *e.g.*, a very cool graphics applications from [**?**].

The components of any nearest-neighbor system are: features (how do you represent your instances?) and similarity (how do you compare instances in the feature space?). You will implement both.

2.1 Extracting Features

We will first represent an image with a bag of words approach. In each image, we simply look at how often each word appears.

Q2.1 (10 points): Create a function `getImageFeatures` that extracts the histogram[7] of visual words within the given image (*i.e.*, the bag of visual words).

```
[h] = getImageFeatures(wordMap, dictionarySize)
```

As inputs, the function will take:

- `wordMap` is a `H` $\times$ `W` image containing the IDs of the visual words
- `dictionarySize` is the maximum visual word ID (*i.e.*, the number of visual words, the dictionary size)

As output, the function will return `h`, a `dictionarySize` $\times$ `1` histogram that is L_1 normalized, (*i.e.*, $\sum h_i = 1$). You may wish to load a single visual word map, visualize it, and verify that your function is working correctly before proceeding.

2.2 Multi-resolution: Spatial Pyramid Matching

Bag of words is simple and efficient, but it discards information about the spatial structure of the image and this information 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 chops 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. Finally 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 layers and K visual words, the resulting vector has dimensionality $K \sum_{l=0}^{L} 4^l = K\left(4^{(L+1)} - 1\right)/3$.

Now comes the weighting scheme. Note that when concatenating all the histograms, histograms from different levels are assigned different weights. Typically (in [**?**]), 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 for layer 0 and layer 1 the weight is set to 2^{-L}, and for the rest it is set to 2^{l-L-1} (*e.g.*, in a three layer spatial pyramid, $L = 2$ and weights are set to 1/4, 1/4 and 1/2 for layer 0, 1 and 2 respectively, see Fig. 6). Note that the L_1 norm (absolute values of all dimensions summed up together) for the final vector is 1.

Q2.2 (15 points): Create a function `getImageFeaturesSPM` that form a multi-resolution representation of the given image.

```
[h] = getImageFeaturesSPM(layerNum, wordMap, dictionarySize)
```

[7] Look into `hist` in MATLAB

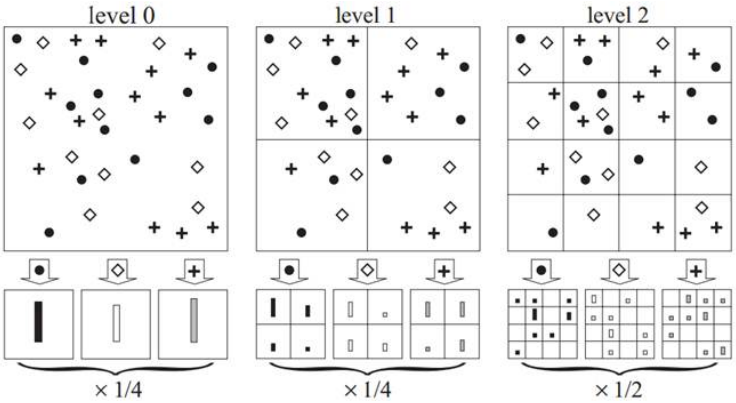

Figure 6: **Spatial Pyramid Matching:** From [**?**]. Toy example of a pyramid for L = 2. The image has three visual words, indicated by circles, diamonds, and crosses. We subdivide the image at three different levels of resolution. For each level of resolution and each channel, we count the features that fall in each spatial bin. Finally, weight each spatial histogram.

As inputs, the function will take:

- `layerNum` the number of layers in the spatial pyramid, *i.e.*, $L + 1$
- `wordMap` is a `H` $\times$ `W` image containing the IDs of the visual words
- `dictionarySize` is the maximum visual word ID (*i.e.*, the number of visual words, the dictionary size)

As output, the function will return `h`, a vector that is L_1 normalized. **Please use a 3-layer spatial pyramid ($L = 2$) for all the following recognition tasks.**

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.

2.3 Comparing images

We will also need a way of comparing images to find the "nearest" instance in the training data. In this assignment, we'll use the histogram intersection similarity. The histogram intersection similarity between two histograms $x_{1:n}$ and $y_{1:n}$ is defined as $\sum_{i=1}^{n} \min(x_i, y_i)$, or `sum(min(x,y))` in MATLAB. Note that since this is a similarity, you want the *largest* value to find the "nearest" instance.

Q2.3 (10 points): Create the function `distanceToSet`

```
[histInter] = distanceToSet(wordHist, histograms)
```

where `wordHist` is a $K\left(4^{(L+1)}-1\right)/3 \times 1$ vector and `histograms` is a $K\left(4^{(L+1)}-1\right)/3 \times T$ matrix containing T features from T training samples concatenated along the columns. This function returns the histogram intersection similarity between `wordHist` and each training sample as a $1 \times T$ vector. Since this is called every time you want to look up a classification, you want this to be fast, and doing a for-loop over tens of thousands of histograms is a very bad idea. Try `repmat` or (even faster) `bsxfun`[8].

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 up to now together.

[8] As a recommendation: unless you're experienced with MATLAB or confident, make sure your optimization works before moving on. Either use a few hand-made examples that you can manually verify or subtract the distances produced by the unoptimized and optimized examples.

You will need to load the training file names from `traintest.mat` and the filter bank and visual word dictionary from `dictionary.mat`. You will save everything to a `.mat` file named `vision.mat`. Included will be:

1. `filterBank`: your filterbank.
2. `dictionary`: your visual word dictionary.
3. `train_features`: a $K\left(4^{(L+1)}-1\right)/3 \times N$ matrix containing all of the histograms of the N training images in the data set. A dictionary with 150 words will make a `train_features` matrix of size 3150×1440.
4. `train_labels`: a $1 \times N$ vector containing the labels of each of the images. (*i.e.*, so that `train_features(:,i)` has label `train_labels(i)`).

We have provided you with the names of the training and testing images in `traintest.mat`. You want to use the cell array of files `train_imagenames` for training, and the cell array of files `test_imagenames` for testing. *You cannot use the testing images for training.* To access the word maps created by `batchToVisualWords.m`, you might need function `strrep` to modify the file names.

You may also wish to convert the labels to meaningful categories. Here is the mapping (a variable named `mapping` is included in `traintest.mat`):

1	2	3	4	5	6	7	8
auditorium	baseball_field	desert	highway	kitchen	laundromat	waterfall	windmill

Q2.4 (15 points): Write a script named `buildRecognitionSystem.m` that produces `vision.mat`, and submit it as well as any helper functions you write.

To qualitatively evaluate what you have done, we have provided a helper function that will let you get the predictions on a new image given the training data. This will give you a visual sanity check that you have implemented things correctly. Use the program as follows:

```
guessImage(absolutePathToImage)
```

The program will load the image, represent it with visual words, and get a prediction based on the histogram. The predictions will appear inside your MATLAB command window as text.

Don't worry if you get a fair amount of wrong answers. Do worry if the program crashes while calling your code or if you get zero correct/all correct/all same answers. If you are getting 0% or 100% performance, go back and verify (visually) that each of your components produces correct output, or check that testing data are accidentally included during training (yet you can pick images from both training and testing set for debugging purposes).

2.5 Quantitative 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 corresponding test images and their labels, and compute the predicted labels of each. To quantify the accuracy, you will compute a confusion matrix `C`: given 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 `trace(C) / sum(C(:))`.

Q2.5 (10 points): Write a script named `evaluateRecognitionSystem.m`

```
[conf] = evaluateRecognitionSystem()
```

that tests the system and outputs the confusion matrix, and submit it as well as any helper functions you write. Report the confusion matrix and accuracy for your results in your write-up. This does not have to be formatted prettily: if you are using LaTeX, you can simply copy/paste it from MATLAB into a `verbatim` environment. Additionally, do not worry if your accuracy is low: with 8 classes, chance is 12.5%. To give you a more sensible number, a reference implementation *with* spatial pyramid matching gives an overall accuracy of $\tilde{5}6\%$.

2.6 Find out the failed cases

There are some classes/samples that are more difficult to classify than the rest using the bags-of-words approach. As a result, they are classified incorrectly into other categories.

Q2.6 (5 points): List some of these classes/samples and discuss why they are more difficult in your write-up.

3 HW1 Distribution Checklist

After unpacking `hw1.zip`, you should have a folder `hw1` containing one folder for the data (`data`) and one for each system you might implement (`matlab` and `custom`). In the `matlab` folder, where you will primarily work, you will find:

- `batchToVisualWords.m`: a provided script that will run your code to convert all the images to visual word maps.
- `computeDictionary.m`: a provided script that will provide input for your visual word dictionary computation.
- `createFilterBank.m`: a provided function that returns a cell array of 20 filters.
- `guessImage.m`: a provided script that will predict the class for given image.
- `RGB2Lab.m`: a provided helper function to convert RGB to Lab image space.

Apart from this, we have also provided the stub codes in `matlab` folder. The data folder contains:

- `data/`: a directory containing `.jpg` images from SUN database.
- `data/traintest.mat`: a `.mat` file with the filenames of the training and testing set.
- `checkA1Submission.m` Script to verify that your submission (<AndrewId>.zip) is in correct structure and contains all required files for submission.

4 HW1 Submission Checklist

The assignment should be submitted to Canvas. The writeup should be submitted as a pdf file named <**AndrewId**>**.pdf**. The code should be submitted as a zip file named <**AndrewId**>**.zip**. By extracting the zip file, it should have the following files in the structure defined below. (Note: Missing to follow the structure will incur huge penalty in scores).

When you submit, remove the folder `data/`, as well as any large temporary files that we did not ask you to create.

- <**andrew_id**>/ # A directory inside .zip file
  - `matlab/`
    - `dictionary.mat`
    - `vision.mat`
    - <!– all .m files inside matlab directory >

Please run the provided script `checkA1Submission('<andrew_id>')` before submission to ensure that all the required files are available. It searches for a '<**andrew_id**>**.zip**' file in the current directory.