16-385 Computer Vision, Fall 2021

Programming Assignment 4
Scene recognition with bag of words

Due Date: Wed November 3rd, 2021 23:59

Instructions

1. **Integrity and collaboration:** Students are encouraged to work in groups but each student must submit their own work. If you work as a group, include the names of your collaborators in your write-up. Code should **NOT** be shared or copied. Please **DO NOT** use external code unless permitted. Plagiarism is strongly prohibited and may lead to failure of this course.

2. **Start early!** Especially those not familiar with Python. Constructing the dictionary in **Q2.3**, as well as converting all the images to visual words in **Q3.1** can easily take up to 60 mins to run (if implemented well!). If you start late, you will be pressed for time while debugging.

3. **Questions:** If you have any question, please look at Piazza first. Other students may have encountered the same problem, and it might have been solved already. If not, post your question on the discussion board. TAs will respond as soon as possible.

4. **Write-up:** Items to be included in the write-up are mentioned in each question, and summarized in the Writeup section. Please note that we **DO NOT** accept handwritten scans for your write-up in this assignment. Please type your answers to theory questions and discussions for experiments electronically.

5. **Handout:** The handout zip file contains 3 items. `assgn4.pdf` is the assignment handout. `data/` contains 1491 image files from the SUN image database, divided into 8 category folders. it also contains a Python pickle file `traintest.pkl`, which contains all the training and testing image paths, as well as the labels. `python/` contains scripts that you will make use of in this project.

6. **Code:** Stick to the function prototypes mentioned in the handout. This makes verifying code easier for the TAs. If you do want to change a function prototype or add an extra parameter, please talk to the TAs.

7. **Python:** This assignment is based on Python 3. You will need to make use of the following packages: `numpy`, `scipy`, `pickle`, `sklearn`, `skimage`, `opencv`.

8. **Submission:** Your submission for this assignment should be a zip file, `<andrew-id.zip>`, composed of your write-up, your Python implementations (including helper functions), and your implementations, results for extra credit (optional). Do not hand in the image files we distributed in the handout zip, or any of the generated word map files. However you should hand in the dictionary and vision `.pkl` files that you generate.

   Your final upload should have the files arranged in this layout:

   - `<AndrewID>.zip`
     - `<AndrewId>.pdf`
     - ec/
       - `computeIDF.py` (Q5.2, *optional*)
       - `evaluateRecognitionSystem_IDF.py` (Q5.2, *optional*)
       - `evaluateRecognitionSystem_SVM.py` (Q5.1, *optional*)
       - `tryBetterFeatures.py` (Q5.3, *optional*)
       - *Any other helper functions or external code*
     - python/
       - `RGB2Lab.py` (*provided*)
       - `batchToVisualWords.py` (*provided*)
       - `buildRecognitionSystem.py` (Q3.3)
       - `createFilterBank.py` (*provided*)
       - `dictionaryHarris.pkl` (Q2.3)
       - `dictionaryRandom.pkl` (Q2.3)
       - `evaluateRecognitionSystem_NN.py` (Q4.2)
       - `evaluateRecognitionSystem_kNN.py` (Q4.2)
       - `extractFilterResponses.py` (Q2.1)
       - `getDictionary.py` (Q2.3)
       - `getHarrisPoints.py` (Q2.2)
       - `getImageDistance.py` (Q4.1)
       - `getImageFeatures.py` (Q3.2)
       - `getRandomPoints.py` (Q2.2)
       - `getVisualWords.py` (Q3.1)
       - `utils.py` (*provided*)
       - `visionHarris.pkl` (Q3.3)
       - `visionRandom.pkl` (Q3.3)
       - *Any other helper functions you need*

Figure 1: The eight categories you have been given from the SUN Image database

1 Overview

One of the core problems in computer vision is classification. Given an image that comes from a few fixed categories, can you determine which category it belongs to? For this assignment, you will be developing a system for scene classification. A system like this might be used by services that have a lot of user uploaded photos, like Flickr or Instagram, that wish to automatically categorize photos based on the type of scenery. It could also be used by a robotic system to determine what type of environment it is in, and perhaps change how it moves around.

You have been given a subset of the SUN Image database consisting of eight scene categories. See Figure 1. In this assignment, you will build an end to end system that will, given a new scene image, determine which type of scene it is.

This assignment is based on an approach to document classification called **Bag of Words**. This approach to document classification represents a document as a vector or histogram of counts for each word that occurs in the document, as shown in Figure 2. The hope is that different documents in the same class will have a similar collection and distribution of words, and that when we see a new document, we can find out which class it belongs to by comparing it to the histograms already in that class. This approach has been very successful in Natural Language Processing, which is surprising due to its relative simplicity. We will be taking the same approach to image classification. However, one major problem arises. With text documents, we actually have a dictionary of words. But what *words* can we use to represent an image with?

This assignment has 3 major parts. Section 2 involves building a dictionary of *visual*

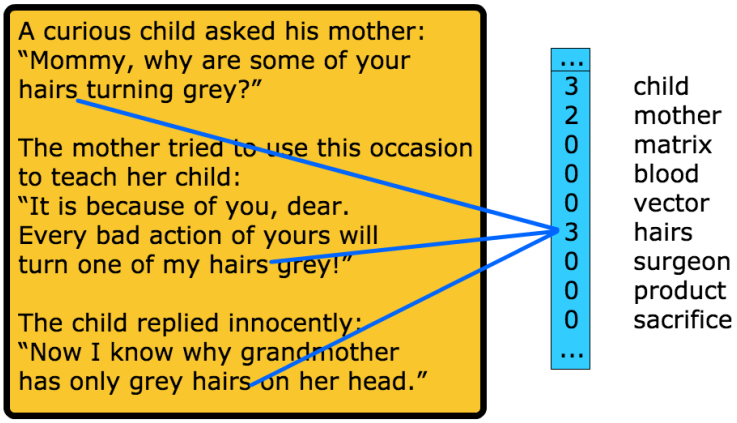

Figure 2: Bag of words representation of a text document

words from the training data. Section 3 involves building the recognition system using our visual word dictionary and our training images. In Section 4, you will evaluate the recognition system using the test images.

In **Section 2**, you will use the provided filter bank to convert each pixel of each image into a high dimensional representation that will capture meaningful information, such as corners, edges etc... This will take each pixel from being a 3D vector of color values, to an nD vector of filter responses. You will then take these nD pixels from all of the training images and and run K-means clustering to find groups of pixels. Each resulting cluster center will become a visual word, and the whole set of cluster centers becomes our dictionary of visual words. In theory, we would like to use all pixels from all training images, but this is very computationally expensive. Instead, we will only take a small sample of pixels from each image. One option is to simply select α pixels from each one uniformly at random. Another option is to use some feature detector (Harris Corners for example), and take α feature points from each image. You will do both to produce two dictionaries, so that we can compare their relative performances. See Figure 3.

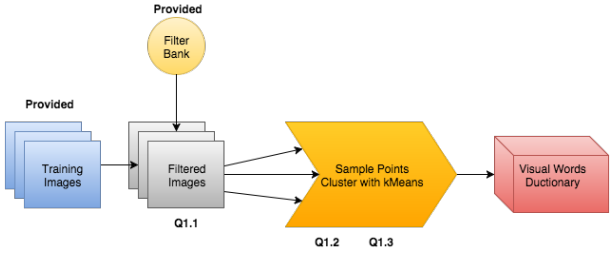

Figure 3: Flow chart of the first part of the project that involves building the dictionary of visual words

In **Section 3**, the dictionary of visual word you produced will be applied to each of the training images to convert them into a word map. This will take each of the nD pixels in all of the filtered training images and assign each one a single integer label, corresponding to the closest cluster center in the visual words dictionary. Then each image will be converted to a "bag of words"; a histogram of the visual words counts. You will then use these to build the classifier (Nearest Neighbors, and SVM for extra credit). See Figure 4.

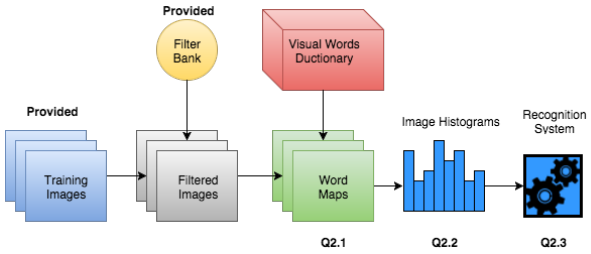

Figure 4: Flow chart of the second part of the project that involves building the recognition system

In **Section 4**, you will evaluate the recognition system that you built. This will involve taking the test images and converting them to image histograms using the visual words dictionary and the function you wrote in Section 3. Next, for nearest neighbor classification, you will use a histogram distance function to compare the new test image histogram to the training image histograms in order to classify the new test image. Doing this for all the test images will give you an idea of how good your recognition system. See Figure 5.

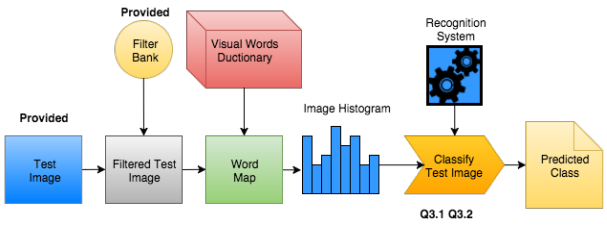

Figure 5: Flow chart of the third part of the project that involves evaluating the recognition system

In your write-up, we will ask you to include various intermediate result images, as well as metrics on the performance of your system.

2 Build Visual Words Dictionary

Q2.1 Extract Filter Responses (20 points)

Write a function to extract filter responses.
`filterResponses = extract_filter_responses(I, filterBank)`

We have provided the function `createFilterBank()`. This function will generate a set of image convolution filters. See Figure 6. There are 4 filters, and each one is made at 5 different scales, for a total of 20 filters. The filters are:

- Gaussian: Responds strongly to constant regions, and suppresses edges, corners and noise
- Laplacian of Gaussian: Responds strongly to blobs of similar size to the filter
- X Gradient of Gaussian: Responds strongly to vertical edges
- Y Gradient of Gaussian: Responds strongly to horizontal edges

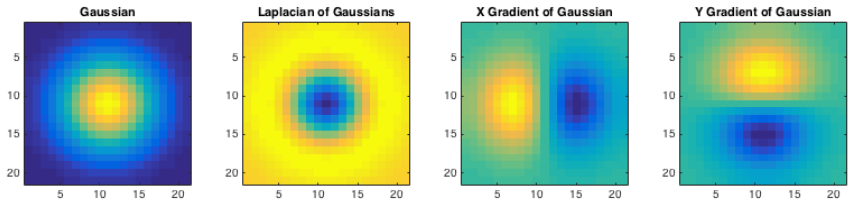

Figure 6: The four types of filters we have provided you.

Pass this array of image convolution filters to your `extract_filter_responses(..)` function, along with an image of size $H \times W \times 3$. Your function should use the provided `rgb2Lab(..)` function to convert the color space of `I` from RGB to Lab. Then it should apply all of the n filters on each of the 3 color channels of the input image. You should end up with $3n$ filter responses for the image. For image filtering, you can use the provided helper function `imfilter(..)` in `utils.py`. The final matrix that you return should have size $H \times W \times 3n$ (Figure 7).

In your write-up: Show an image from the data set and 3 of its filter responses. Explain any artifacts you may notice. Also, briefly describe the CIE Lab color space, and why we would like to use it. We did not cover the CIE Lab color space in class, so you will need to look it up online.

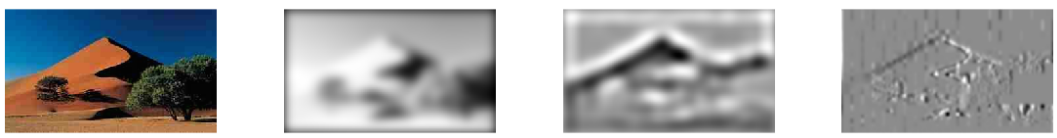

Figure 7: Sample desert image and a few filter responses (in CIE Lab color space)

Q2.2 Collect sample of points from image (20 points)

Write two functions that return a list of points in an image, that will then be used to generate visual words.

First, write a simple function that takes an image and an α, and returns a matrix of size $\alpha \times 2$ of **random pixels locations** inside the image.

```
points = get_random_points(I, alpha)
```

Next, write a function that uses the **Harris corner detection** algorithm to select key points from an input image, using a similar algorithm as shown in class.

```
points = get_harris_points(I, alpha, k)
```

Recall from class that the Harris corner detector finds corners by building a covariance matrix of edge gradients within a region around a point in the image. The eigenvectors of this matrix point in the two directions of greatest change. If they are both large, then this indicates a corner. See class slides for more details.

This function takes the input image `I`. `I` may either be a color or grayscale image, but the following operations should be done on the grayscale representation of the image. For each pixel, you need to to compute the covariance matrix

$$M = \begin{bmatrix} \sum_{p \in P} I_{xx} & \sum_{p \in P} I_{xy} \\ \sum_{p \in P} I_{yx} & \sum_{p \in P} I_{yy} \end{bmatrix}$$

where $I_{ab} = \frac{\partial I}{\partial a} \frac{\partial I}{\partial b}$, and P is a neighborhood of the pixel. You can use 3×3 or 5×5 for P. It is a good idea to precompute the image's X and Y gradients, instead of doing them in every iteration of the loop. For the sum, also think about how you could do it using a convolution filter.

You then want to detect corners by finding pixels whose covariance matrix eigenvalues are large. Since its expensive to compute the eigenvalues explicitly, you should instead compute the response function with

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2 = \det(M) - k \operatorname{tr}(M)^2,$$

where *det* represents the determinant and *tr* denotes the trace of the matrix. Recall that when detecting corners with the Harris corner detector, corners are points where both eigenvalues are large. This is in contrast to edges (one eigenvalue is large, while the other is small), and flat regions (both eigenvalues are small). In the response function, the first term becomes very small if one of the eigenvalues are small, thus making $R < 0$. Larger values of R indicate similarly large eigenvalues.

Instead of thresholding the response function, simply take the top α response as the corners, and return their coordinates. A good value for the k parameter is 0.04 - 0.06.

Note: This Harris can be implemented without any for loops, using basic numpy functions, filtering, and vectorization, particularly the sums in computing M, as well as

the response function R. No marks will be taken off for slow code, but when computing the dictionary of visual words in Q2.3, this can be the difference between minutes and hours. Remember, you will be applying this function to about 1500 images.

These functions will be used in the next part to select α points from every training image.

In your write-up: Show the results of your corner detector on 3 different images.

Figure 8: Possible results for the getPoints functions

Q2.3 Compute Dictionary of Visual Words (20 points)

You will now create the dictionary of visual words. Write the function:

```
dictionary = get_dictionary(imgPaths, alpha, K, method)
```

This function takes in an array of training image paths. For each and every training image, load it and apply the filter bank to it. Then get α points for each image and put them into an array, where each row represents a single n-dimensional pixel (n is the number of filters). If there are T training images total, then you will build a matrix of size $\alpha T \times 3n$, where each row corresponds to a single pixel, and there are αT total pixels collected.

`method` will be a string either `'Random'` or `'Harris'`, and will determine how the α points will be taken from each image. If the method is `'Random'`, then the α points will be selected using `get_random_points(..)`, while method `'Harris'` will tell your function to use `get_harris_points(..)` to select the α points. Once you have done this, pass pixel responses of all the points collected from all training images to `sklearn`'s K-means function to get the dictionary.

The result will be a matrix of size $K \times 3n$, where each row represents the coordinates of a cluster center. This matrix will be your dictionary of visual words.

For testing purposes, use $\alpha = 50$ and $K = 100$. Eventually you will want to use much larger numbers (e.g., $\alpha = 200$ and $K = 500$) to get better results for the final part of the write-up.

This function can take a while to run. Start early and be patient. When it has completed, you will have your dictionary of visual words. Save this in a pickle (`.pkl`) file. This is your visual words dictionary. It may be helpful to write a `computeDictionary.py` which will do the legwork of loading the training image paths, processing them, building the dictionary, and saving the pickle file. This is not required, but will be helpful to you when calling it multiple times.

For this question, you must produce two dictionaries. One named `dictionaryRandom.pkl`, which used the random method to select points and another named `dictionaryHarris.pkl` which used the Harris method. Both must be handed in.

3 Build Visual Scene Recognition System

Q3.1 Convert image to word map (30 points)

Write a function to map each pixel in the image to its closest word in the dictionary.

`wordMap = get_visual_words(I, filterBank, dictionary)`

`I` is the input image of size $H \times W \times 3$. `dictionary` is the dictionary computed previously. `filterBank` is the filter bank that was used to construct the dictionary. `wordMap` is an $H \times W$ matrix of integer labels, where each label corresponds to a word/cluster center in the dictionary.

Use `scipy`'s function `cdist(..)` with Euclidean distance to do this efficiently. You can visualize your results with the `skimage.color` function `label2rgb(..)`.

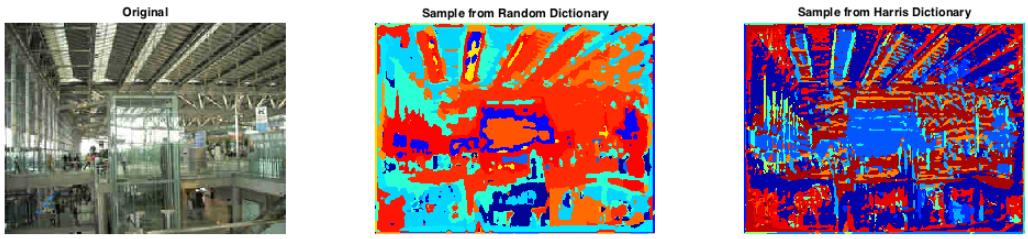

Figure 9: Sample visual words plot, made from dictionary using $\alpha = 50$ and $K = 50$

Once you are done, call the provided script `batchToVisualWords.py`. This function will apply your implementation of `get_visual_words(..)` to every image in the training and testing set. The script will load `traintest.pkl`, which contains the names and labels of all the data images. For each training image `data/<category>/X.jpg`, this

script will save the word map pickle file `data/<category>/X_<point_method>.pkl`. For optimal speed, modify `num_cores` to the number of cores in your computer. This will save you from having to keep rerunning `get_visual_words` in the later parts, unless you decide to change the dictionary.

In your write-up: Show the word maps for 3 different images from two different classes (6 images total). Do this for each of the two dictionary types (random and Harris). Are the visual words capturing semantic meanings? Which dictionary seems to be better in your opinion? Why?

Q3.2 Get Image Features (10 points)

Create a function that extracts the histogram of visual words within the given image (i.e., the bag of visual words).

```
h = get_image_features(wordMap, dictionarySize)
```

`h`, the vector representation of the image, is a histogram of size $1 \times K$, where `dictionarySize` is the number of clusters K in the dictionary. `h(i)` should be equal to the number of times visual word `i` occurred in the word map.

Since the images are of differing sizes, the total count in `h` will vary from image to image. To account for this, L_1 normalize the histogram before returning it from your function.

Q3.3 Build Recognition System - Nearest Neighbors (10 points)

Now that you have build a way to get a vector representation for an input image, you are ready to build the visual scene classification system. This classifier will make use of nearest neighbor classification. Write a script `buildRecognitionSystem.py` that saves `visionRandom.pkl` and `visionHarris.pkl` and in each pickle store a dictionary that contains:

1. `dictionary`: your visual word dictionary, a matrix of size $K \times 3n$.
2. `filterBank`: filter bank used to produce the dictionary. This is an array of image filters.
3. `trainFeatures`: $T \times K$ matrix containing all of the histograms of visual words of the T training images in the data set.
4. `trainLabels`: $T \times 1$ vector containing the labels of each training image.

You will need to load the `train_imagenames` and `train_labels` from `traintest.pkl`. Load `dictionary` from `dictionaryRandom.pkl` and `dictionaryHarris.pkl` you saved in part **Q2.3**.

4 Evaluate Visual Scene Recognition System

Q4.1 Image Feature Distance (15 points)

For nearest neighbor classification you need a function to compute the distance between two image feature vectors. Write a function

```
dist = get_image_distance(hist1, hist2, method)
```

`hist1` and `hist2` are the two image histograms whose distance will be computed (with `hist2` being the target), and returned in `dist`. The idea is that two images that are very similar should have a very small distance, while dissimilar images should have a larger distance.

`method` will control how the distance is computed, and will either be set to `'euclidean'` or `'chi2'`. The first option tells to compute the Euclidean distance between the two histograms. The second uses χ^2 distance. For χ^2 distance, you can use the function `chi2dist(..)` in `utils.py`.

Alternatively, you may also write the function

```
[dist] = get_image_distance(hist1, histSet, method)
```

which, instead of the second histogram, takes in a matrix of histograms, and returns a vector of distances between `hist1` and each histogram in `histSet`. This may make it possible to implement things more efficiently. Of course, you need to modify `chi2dist(..)` to handle the calculation of multiple distances. Choose either one or the other to hand in.

Q4.2 Evaluate Recognition System - NN and kNN (40 points)

Write a script `evaluateRecognitionSystem_NN.py` that evaluates your nearest neighbor recognition system on the test images. Nearest neighbor classification assigns the test image the same class as the "nearest" sample in your training set. "Nearest" is defined by your distance function.

Load `traintest.pkl` and classify each of the `test_imagenames` files. Have the script report both the accuracy ($\frac{\#correct}{\#total}$), as well as the 8×8 *confusion matrix* `C`, where the entry `C(i,j)` records the number of times an image of actual class `i` was classified as class `j`.

The **confusion matrix** can help you identify which classes work better than others and quantify your results on a per-class basis. In an perfect classifier, you would only have entries in the diagonal of `C` (implying, for example, that an `'auditorium'` always got correctly classified as an `'auditorium'`).

For each combination of dictionary (random or Harris) and distance metric (Euclidean and χ^2), have your script print out the confusion metric and the confusion matrix. Use `print(..)` so we can know which is which.

Now take the best combination of dictionary and distance metric, and write a script `evaluateRecognitionSystem_kNN.py` that classifies all the test image using k Nearest Neighbors. Have your script generate a plot of the accuracy for k from 1 to 40. For the best performing k value, print the confusion matrix. Note that this k is different from the dictionary size K. This k is the number of nearby points to consider when classifying the new test image.

In your write-up:

- Include the output of `evaluateRecognitionSystem_NN.py` (4 confusion matrices and accuracies).
- How do the performances of the two dictionaries compare? Is this surprising?
- How do the two distance metrics affect the results? Which performed better? Why do you think this is?
- Also include output of `evaluateRecognionSystem_kNN.py` (plot and confusion matrix). Comment on the best value of k. Is larger k always better? Why or why not? How did you choose to resolve ties?

5 Extra credit

Q5.1 Evaluate Recognition System - Support Vector Machine (extra: 20 points)

In class, we learned about a powerful classifier by the name support vector machine (SVM). Write a script, `evaluateRecognitionSystem_SVM.py` to do classification using SVM. Produce a new `visionSVM.pkl` file with whatever parameters you need, and evaluate it on the test set.

We recommend you to use `sklearn` (https://scikit-learn.org/stable/modules/svm) for SVM. Try at least two types of kernels.

In your write-up: Are the performances of the SVMs better than nearest neighbor? Why or why not? Does one kernel work better than the other? Why?

Q5.2 Inverse Document Frequency (extra: 10 points)

With the bag-of-word model, image recognition is similar to classifying a document with words. In document classification, inverse document frequency (IDF) factor is incorporated which diminishes the weight of terms that occur very frequently in the document set. For example, because the term "the" is so common, this will tend to incorrectly emphasize documents which happen to use the word "the" more frequently, without giving enough weight to the more meaningful terms.

In this project, the histogram we computed only considers the term frequency (TF), i.e., the number of times that word occurs in the word map. Now we want to weight

the word by its inverse document frequency. The IDF of a word is defined as:

$$IDF_w = log \frac{T}{|\{d : w \in d\}|}$$

Here, T is number of all training images, and $|\{d : w \in d\}|$ is the number of images d such that w occurs in that image.

Write a script `computeIDF.py` to compute a vector `IDF` of size $1 \times K$ containing IDF for all visual words, where K is the dictionary size. Save IDF in `idf.pkl`. Then write another script `evaluateRecognitionSystem_IDF.py` that makes use of the `IDF` vector in the recognition process. You can use either nearest neighbor or SVM as your classifier.

In your write-up: How does Inverse Document Frequency affect the performance? Better or worse? Does this make sense?

Q5.3 Better pixel features (extra: 20 points)

The filter bank we provided to you contains some simple image filters, such as Gaussian, derivatives, and Laplacians. However we learned about various others in class, such as Haar wavelets, Gabor filters, SIFT, HOG etc... Try out anything you think might work. Include a script `tryBetterFeatures.py` and any other code you need. Feel free to use any code or packages you find on-line, but be sure to cite it clearly. Experiment with whatever you see fit, just make sure the submission doesn't become too large.

In your write-up: What did you experiment with and how did it perform. What was the accuracy?

6 Writeup Summary

Q2.1 Show an image from the data set and 3 of its filter responses. Explain any artifacts you may notice. Also briefly describe the CIE Lab color space, and why we would like to use it. We did not cover the CIE Lab color space in class, so you will need to look it up online.

Q2.2 Show the results of your corner detector on 3 different images.

Q3.1 Show the word maps for 3 different images from two different classes (6 images total). Do this for each of the two dictionary types (random and Harris). Are the visual words capturing semantic meanings? Which dictionary seems to be better in your opinion? Why?

Q4.2 Comment on whole system:

- Include the output of `evaluateRecognitionSystem_NN.py` (4 confusion matrices and accuracies).
- How do the performances of the two dictionaries compare? Is this surprising?
- How about the two distance metrics? Which performed better? Why do you think this is?
- Also include output of `evaluateRecognionSystem_kNN.py` (plot and confusion matrix). Comment on the best value of k. Is a larger k always better? Why or why not? How did you choose to resolve ties?

Q5.1 (Extra Credit.) Are the performances achieved using SVM classification better than those usng nearest neighbor? Why or why not? Does one kernel work better than the other? Why?

Q5.2 (Extra Credit.) How does Inverse Document Frequency affect the performance? Better or worse? Does this make sense?

Q5.3 (Extra Credit.) What did you experiment with and how did it perform. What was the accuracy?

References

[1] S. Lazebnik, C. Schmid, and J. Ponce, *Beyond Bags of Features: Spatial Pyramid Matching for Recognizing Natural Scene Categories.* CVPR, 2006.