Machine Learning and Computer Vision Assignment 4

Huihuang Zheng, huihuang@utexas.edu, CS logID: huihuang

hz4674 Fall 2015

1 Short answer questions

1. Parallel to x direction

2. First, You are approaching (or going away) from the object, so the scale of image is changing. Within a window matching may fail because it's not scale invariant.

   Second, when the light of two views are different, the dense stereo matching may not work because light causes intensity changes.

3. SIFT feature divides a patch into 4 * 4 sub-patches. For each sub-patch, compute a histogram of 8 bins (so every bin covers 45 degrees, these orientations are relative to the keypoint's dominant orientation). Finally normalize this 4 * 4 * 8 = 128 dimension vector into unit length. So in a single dimension, it's a normalized of count of histogram covers 45 degree to relative dominant orientation in a sub-patch.

4. x, y, (location) scale, rotation. Because SIFT is invariant to scale and rotation, we need consider that. The main step for general Hough Transform is: in the query image, choose a point and measure distances of SIFT features in query image to the point. Then we vote for the point in matching image:

   For every feature
   For every possible scale
   For every possible rotation
   get location x, y via distance multiplies scale, rotate by rotation
   vote for H[x, y, scale, roation]

2 Programming

To run my program, change the directory at head of *main.m* and then run it. You may also want to test code for every single question. You can add or remove the number in vector RUN_QUESTIONS and RUN_EXTRA in 8th, 9th lines of *main.m*. This number indicates the main function will run which question showing to you. For example, RUN_QUESTIONS = [1, 3, 4]; RUN_EXTRA = [1, 3] will run code of question 1,3,4 and extra credit question 1,3.

Because in this assignment, generate visual words cost a lot of time. So I add precompute data files in submitted folder. You can generate all precompute data files in question 2. To generate, add number 2 in RUN_QUESTIONS mentioned above. It will cost around 1 hour.

Now I will illustrate my code for every question, in all figure results, the matching rank from left to right is top-1, top-2, top-3, ...

2.1 Raw descriptor matching

See *rawDescriptorMatches.m*. Run question 1, you need to select a polygon region in the provided image, then the code will show you the matching in another provided image.

The matching process is: A descriptor D1 is matched to a descriptor D2 only if the Euclidean distance d(D1,D2) multiplied by threshold is not greater than the second minimum distance of D1 to all other descriptors. The default threshold value is 1.5

The figure 1 and 2 show the polygon region I chose and the result raw matching. It looks better than the one in assignment example :-)

2.2 Visualizing the vocabulary

See *visualizeVocabulary.m*. As mentioned above, the question 2 runs quite slowly. I used 0.1 fraction of frames to generate visual words. It needs about 1 hour to complete generation. I found most of time costs in kmeans cluster. I cluster visual works by K = 1500. You can modify K and PORTION in 7, 8th line of *visualizeVocabulary.m* to change the fraction of frames or number of clusters you want to use for generating visual words.

I chose first two of visual words to show in figure 3. The two lines each contains 10 examples

Figure 1: The Polygon I Chose

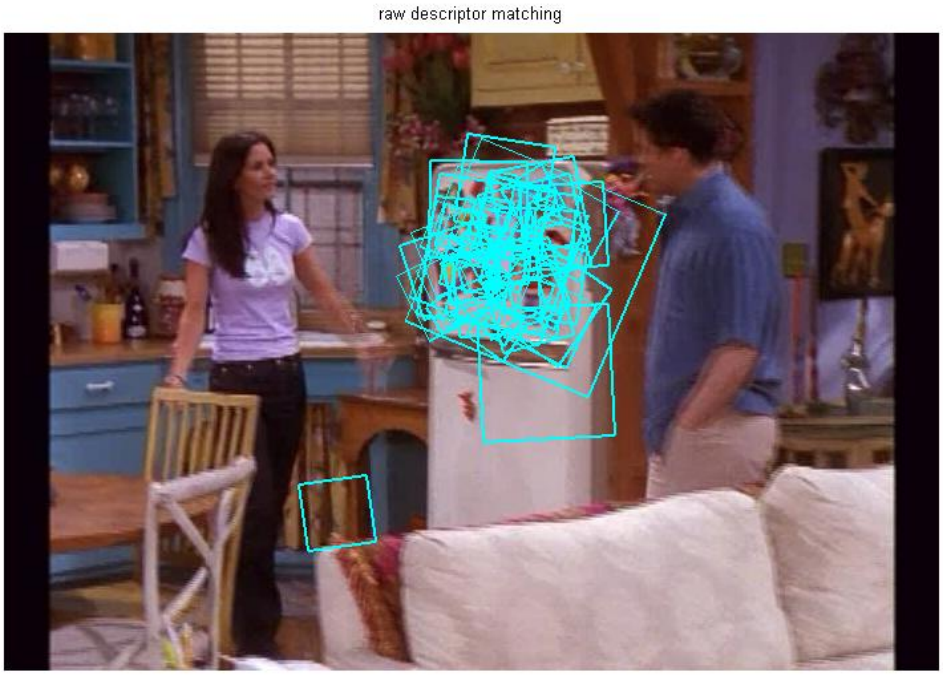

Figure 2: Result of Question 1

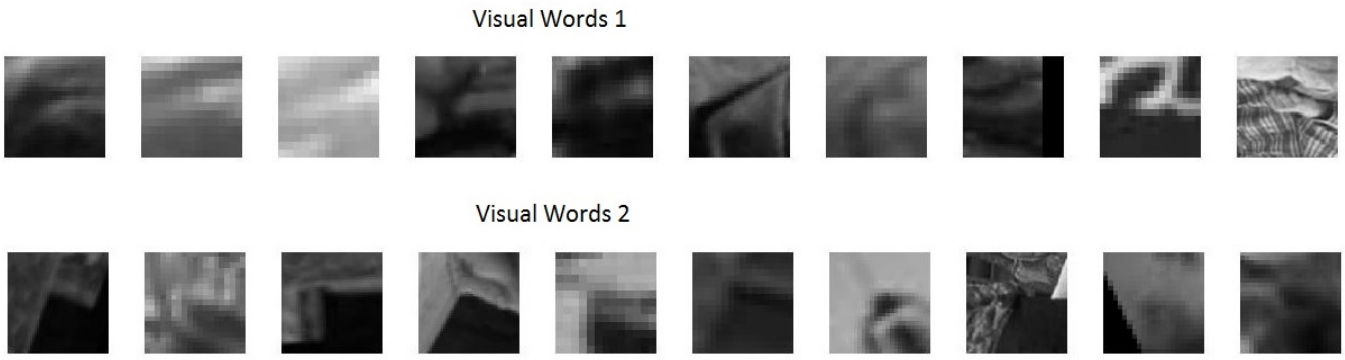

Figure 3: Result of Question 1

Figure 4: Full Frame Query 1

of the visual words. It seems like that the first visual word is some line across the image and the second feature is an angle about 90 degrees.

2.3 Full frame queries

See *fullFrameQueries.m*. But this function just call the *regionQueries.m* by set the region to full frame (I implemented this because this two function do almost same thing). So see question 4 for detail explanation of *regionQueries.m*.

Two successful (figure 4, 5) and one fail (figure 6) examples:

Figure 5: Full Frame Query 2

Figure 6: Fail Full Frame Query

2.4 Region queries

See *regionQueries.m*. This function does bag-of-words matching, using a linear scan through all the database frames. It matches two frames by comparing two bag-of-words histograms using the normalized scalar product. This function has many optional parameters:

- saveToFile, loadFromFile: because compute histograms of all frames cost a lot of time (10 minutes in my experiment). Setting saveToFile true will save the histograms as files in disk. So that you can set loadFromFile true next time you run this function, which makes it outputs result within seconds.
- fullFrameQuery: boolean, if true we don't choose region but query full frame
- onlyCompareNonZeroWords: boolean, if true, we just count histogram of features in the query region.
- useTfidf, ignoreCommon: boolean, if true we will use tfidf, ignore common features. So the extra credit question 1 just set these two true.

From results and other experiments I did. I found two kinds of things are easiest to be find in different scenes: 1. woman's cloth with pattern. 2. multiply objects in region. (e.g. bottles on cabinet, stickers on surface of refrigerator). This is because those have many edges, so will have lots of SFIT features, let they be easier to detect in different views.

Now I show a woman's cloth detected in other scene (figure 7, 8) and sticker of refrigerator (figure 9, 10). Two examples, both can find the region in different scenes.

I found the most common failure case is man's pure color cloth. It doesn't have many edges so few features can be detected. For failure example, figure 11, 12.

3 Extra Credit

3.1 Ignore list and tfidf

To fun this, just set **useTfidf, ignoreCommon** parameters of *regionQueries.m* true. How to determine which visual words to be ignore? I counted how many frames the word has occurred. The result was sorted and represented in figure 13. I ignore those words occurring more than 4000 frames.

The result improves, I randomly sample 20 frames to compare use/without tfidf and ignoring and check whether the result improves manually (by my eyes). 3 results got better

Figure 7: Region Query of Woman's Cloth

Figure 8: Result of Woman's Cloth

Figure 9: Region Query of Stickers

Figure 10: Result of Stickers

Figure 11: Region Query of Man's Cloth

Figure 12: Fail Result of Man's Cloth

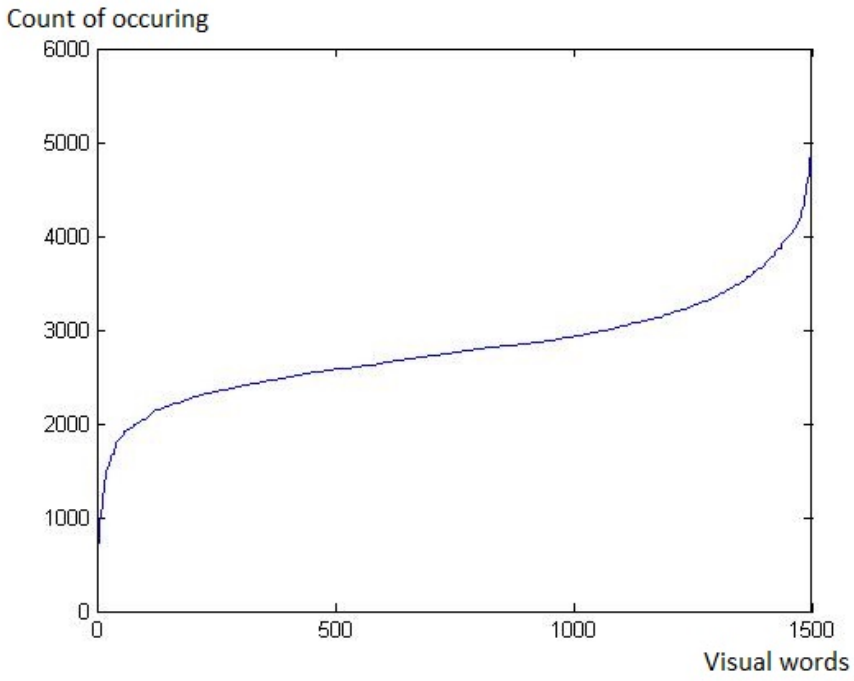

Figure 13: Sorted Count of Occuring

and 1 query got worse. For example, we can see the top 3 rank of using tfidf (figure 14) is better than those without (figure 15).

3.2 Shot break detection

See *shotBreakDetection.m*. I assume that during shot breaking, the bag of words in the scene changes a lot. So I compare histogram of visual words for every adjacent frame pairs by computing normalized Compare two bag-of-words histograms using the normalized scalar product. Set a threshold. If the normalized scalar product less then the threshold (the two frames show less similarity), then it's a shot break.

By setting threshold as 0.2 works well. You can change the threshold at 10th line of *shotBreakDetection.m*. I randomly chose 5 shot break to show in the code, you can see almost every time the 5 pairs of adjacent frames are shot break. See figure 16.

Figure 14: Result using tfidf and ignoring

Figure 15: Result without using tfidf and ignoring

Figure 16: Shot Breaks, every row is a pair of two adjacent frames