
COMP 338

Computer Vision

Assignment 1 Report

Image Classification Based on SIFT Feature Extraction

Programming & Technical Report

Student Group 14

Project Description | 1

This project involves the approach to extract SIFT(Scale-Invariant Feature Transform) features from training data of given dataset with 5 different classes, and clustering the result SIFT descriptors to form an dictionary of Bag-of-words representation. Eventually, generate keywords histograms for both training data and testing data, as well as label them to one of the cluster centers, classify the testing images to its nearest neighbor according to the distances of histograms.

After classification of testing images, the program will run a set of tests to evaluate the classification performance. Moreover, the program also demonstrates the classification progress using varies histogram comparing mechanisms or different clustering parameters.

User Instruction & Requirements Walkthrough | 2

Note that, since the boundary checking is not required, each option/index entered much be from given options or index that within meaningful range.

The project files are with the following directories:

```
SIFT.py
cluster
./features
./COMP338_Assignment1_Dataset
```

Where the SIFT is the project code, cluster is the saved clustered centers variable list of type sklearn.kmeans object. Two folders, features and COMP338_Assignment_Dataset are the folder for saved SIFT features, and given image dataset, respectively.

Step 1. Feature Extraction of Training data

Run the python file SIFT.py, then the following will be presented:

```
Reading from COMP338_Assignment1_Dataset/Training/keyboard/0070.jpg
Reading from COMP338_Assignment1_Dataset/Training/keyboard/0071.jpg
Reading from COMP338_Assignment1_Dataset/Training/keyboard/0072.jpg
Reading from COMP338_Assignment1_Dataset/Training/keyboard/0073.jpg
Reading from COMP338_Assignment1_Dataset/Training/keyboard/0074.jpg
Reading from COMP338_Assignment1_Dataset/Training/keyboard/0076.jpg
Reading from COMP338_Assignment1_Dataset/Training/keyboard/0077.jpg
Reading from COMP338_Assignment1_Dataset/Training/keyboard/0078.jpg
Reading from COMP338_Assignment1_Dataset/Training/keyboard/0079.jpg
Reading from COMP338_Assignment1_Dataset/Training/keyboard/0080.jpg
Images Collected.

_____SIFT Image Classification_____

1.Load existing training datas` pre-extracted features 2.Extract from images
Enter an option with its code: █
```

The program first reading the images from the given dataset in the corresponding directories, then shows the welcoming message with two options to perform the SIFT feature extraction on training data.

Usually, by our testing, the average time taken for the extraction of images will take up to one hour for the given 350 training images, this is however not convenient for code testing during the programming stage, therefore we made the program to be able to save the features ever time it extracted them, to the feature directory, as well as load the saved files when needed. The following list shows the SIFT feature contents saved, with respect to the file name in feature directory:

"SIFT_Features_Train" – SIFT keypoint descriptors from training images

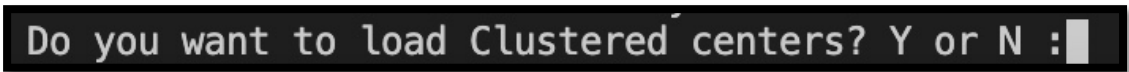
"SIFT_Features_Train_Points" - SIFT keypoints from training images

It is recommended to enter option 2 to directly load the extracted features for time saving purpose. The technical details of SIFT feature extraction is explained in later section.

Step 2. Dictionary Generation

Once successfully obtained the features from training dataset, the program will then cluster the SIFT descriptors of training data to any numbers of cluster centers. The number of centers is given by the variable Num_WORDS at line 14.

Note that this step is time consuming too by our testing, therefore the program also gives an option to load the clustered centers instead of performing clustering, as shown below:



```
Do you want to load Clustered centers? Y or N : █
```

If letter "Y" is entered, the program will load the cluster centers from file *"cluster"*. Otherwise, if the letter "N" is entered, the program will perform k-mean clustering to all descriptors of training dataset and save the resulting clustering center list.

Step 3. Image representation with a histogram of codewords

With the clustering centers ready, the program then ready to represent images from both training and testing dataset as histograms of clustered centers.

Firstly, the program will read the testing images, and then try to extract the features from them as we did to training dataset. The loading per-trained features option and perform feature extraction option are both given, as shown below:

```

Reading From COMP338_Assignment1_Dataset/Test/faces/0016.jpg
Reading From COMP338_Assignment1_Dataset/Test/faces/0031.jpg
Reading From COMP338_Assignment1_Dataset/Test/faces/0021.jpg
Reading From COMP338_Assignment1_Dataset/Test/faces/0035.jpg
Reading From COMP338_Assignment1_Dataset/Test/keyboard/0059.jpg
Reading From COMP338_Assignment1_Dataset/Test/keyboard/0066.jpg
Reading From COMP338_Assignment1_Dataset/Test/keyboard/0060.jpg
Reading From COMP338_Assignment1_Dataset/Test/keyboard/0075.jpg
Reading From COMP338_Assignment1_Dataset/Test/keyboard/0014.jpg
Reading From COMP338_Assignment1_Dataset/Test/keyboard/0016.jpg
Reading From COMP338_Assignment1_Dataset/Test/keyboard/0030.jpg
Reading From COMP338_Assignment1_Dataset/Test/keyboard/0025.jpg
Reading From COMP338_Assignment1_Dataset/Test/keyboard/0037.jpg
Reading From COMP338_Assignment1_Dataset/Test/keyboard/0057.jpg
Testing images collected!

---Extracting Features from Test dataset...
Do you wish to 1.load features from test images or 2.extract features from test images :

```

The following list shows the SIFT feature contents saved, with respect to the file name in feature directory:

"SIFT_Features_Test" – SIFT keypoint descriptors from testing images

"SIFT_Features_Test_Points" - SIFT keypoints from testing images

It is recommended to enter option 2 to directly load the extracted features for time saving purpose.

After obtaining SIFT features of testing images, we now need to label each descriptor from both training and testing images into one of the clustered centers. The following two options is given:

```

---Image representing
- 1.Load labels for Train & Test 2.Label them :

```

It is recommended to enter option 1 to directly load the labels for time saving purpose. Please also note that the labelling is using L2 distance between descriptors, this will be explained in detail in later section.

Once the labeling is done, the program then allows you to view some image patches from a certain class label:

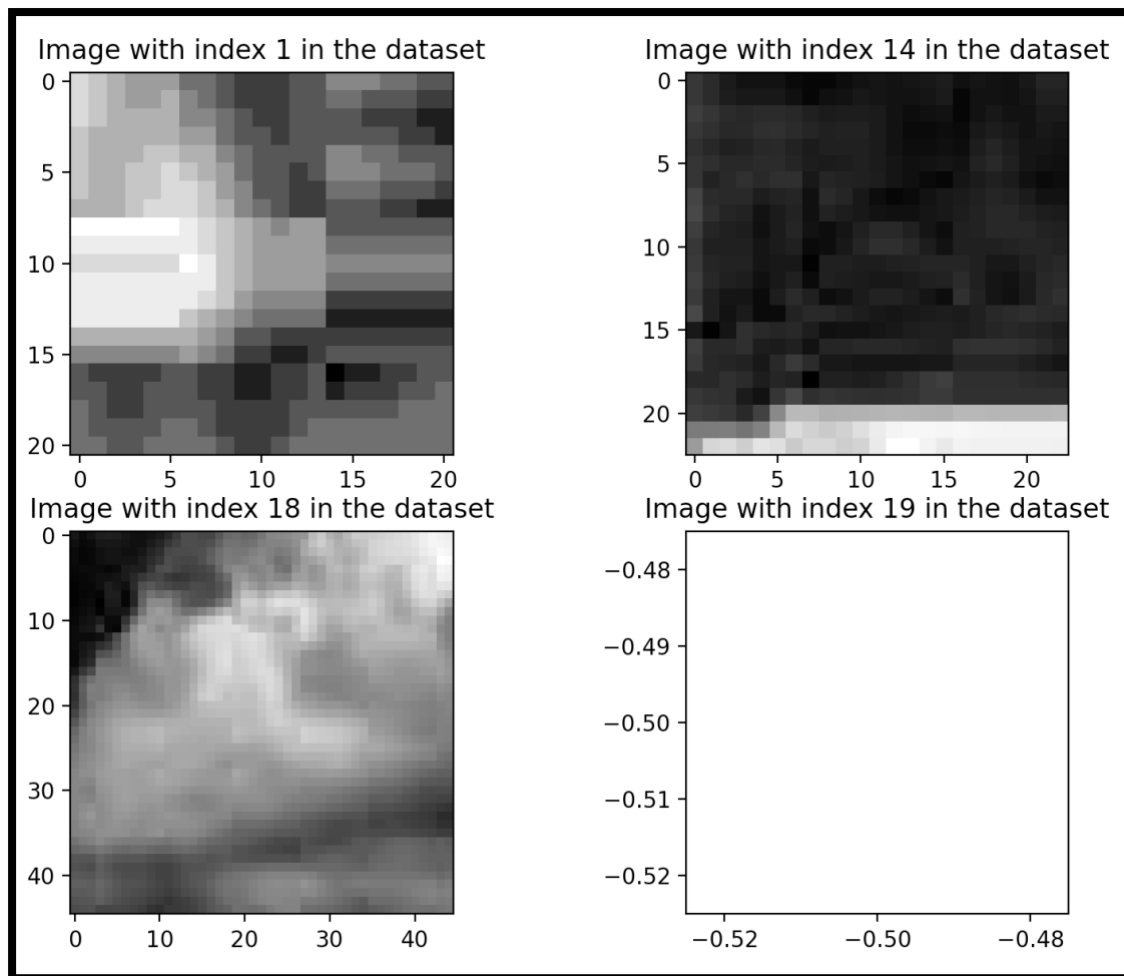
```

Labels are ready.
Enter a number to view some image patch from this word class(-1 if done):

```

Please note that index entered must not exceed (the number of clustered centers - 1), otherwise the program will exist with errors, since the boundary checking is not required.

Please see an example shows following with class label "5" is entered:



Note that if an descriptor and its corresponding keypoint is at the location that on the edge of the image and with high sigma value, the image patch of it might not be drawn correctly since the information is too abstract.

You can enter -1 to continue or enter another class label to view image patches, as suggested by the program output.

Then, if -1 is entered, the program will generate the BOW(Bag-of-Words) histograms for each image in both training and testing dataset. Implementation Details will be explained in later section.

```
Enter a number to view some image patch from this word class(-1 if done): -1
---Creating BOW Histograms for Training images...
Initializing arrays...
Counting for training dataset...
Counting for test dataset...
Counting Done,file saved.
```

Please not that, for testing purpose, the program will also store generated labels and

histograms into “./features” folder for both datasets.

The following list shows the meanings of each file saved, with respect to the file name in feature directory:

“*Training_labels*” – Saved labels for each descriptor of training images with respect to cluster centers

“*Test_labels*” – Saved labels for each descriptor of testing images with respect to cluster centers

“*Training_hist*” – Saved BOW histogram of training dataset

“*Test_hist*” – Saved BOW histogram of testing dataset

Step 4-5. Classification & Evaluation

After obtained the BOW histograms of each image, then program then performs L2 distance on each BOW testing image histograms to compare to all BOW training image histograms, and hence conclude the corresponding testing image to the same image class as a certain BOW training image histogram, which has minimum L2 BOW histogram distance to the current testing BOW histogram.

By comparing the grand truth of all testing images and their conclude(classified) classes, the program evaluate the performance of the classification and showing the error rates.

```
---Evaluating performance...
===Performance Result===
[Overall Error rate :0.78]
Airplane class rate :0.9
Cars Error rate :0.3
Dogs Error rate :0.9
Faces Error rate :0.9
Keyboard Error rate :0.9

Enter 0-4 to see images that are correctly classified(-1 to skip):
```

As shown above, you can also view the images that and correctly & incorrectly classified in each 0-4 different class, where the following list shows the relation between class indexes and class name:

Class 0 – Airplanes

Class 1 – Cars

Class 2 – Dog

Class 3 – Faces

Class 4 – Keyboard

Enter -1 to skip this stage.

The program will also generate a confusion matrix to evaluate the performance, as shown below:

```

---Calculating Confusion Metrics...
[1 7 1 1 0]
[0 7 3 0 0]
[0 7 1 0 2]
[1 6 2 1 0]
[0 8 0 1 1]
Please Note that both row and col here is arranged in the order of airplanes, cars, dog, faces, keyboard.

```

Where both the rows and columns are arranged in the given order.

Step 6. Histogram Intersection

By replacing the L2 method with the histogram intersection method for comparing histograms, the program runs step 4 and 5 again, and present the result, as shown below:

```

---Using Histogram Intersection to comparing histograms...

---Evaluating performance...
===Performance Result===
[Overall Error rate :0.64]
Airplane class rate :1.0
Cars Error rate :0.1
Dogs Error rate :0.8
Faces Error rate :0.5
Keyboard Error rate :0.8

Enter 0-4 to see images that are correctly classified(-1 to skip):

```

The classifying result viewing and confusion matrix generation is also performed in this particular step.

```

Enter 0-4 to see images that are correctly classified(-1 to skip):-1
Enter 0-4 to see images that are incorrectly classified:-1

---Calculating Confusion Metrics...
[0 2 3 3 2]
[0 9 0 1 0]
[0 3 2 1 4]
[0 2 2 5 1]
[1 3 2 2 2]
Please Note that both row and col here is arranged in the order of airplanes, cars, dog, faces, keyboard.

```

Step 7. Reducing the number of clustered centers

As mentioned in the documentation of this assignment, in step 7 we need to perform an experiment with different number of clustering centers.

In our implementation, this is achievable by simply change the variable Num_WORDS

in line 14 of the code. This value is set by default to 500.

```
13
14     Num_WORDS = 20
15
```

Then execute the program, you can get the following results (Please note, since we have changed the number of clustering centers, therefore please cluster a new set of centers instead of load saved file in step 2, this is also the case when comes to labelling stage.):

1. Using L2 distance for histograms comparing

```
---Evaluating performance...
===Performance Result===
[Overall Error rate :0.76]
Airplane class rate :0.8
Cars Error rate :0.6
Dogs Error rate :0.8
Faces Error rate :0.7
Keyboard Error rate :0.9

Enter 0-4 to see images that are correctly classified(-1 to skip):-1

Enter 0-4 to see images that are incorrectly classified:-1

---Calculating Confusion Metrics...
[2 2 3 0 3]
[0 4 3 2 1]
[0 3 2 3 2]
[0 7 0 3 0]
[1 4 3 1 1]
```

2. Using Intersection for histograms comparing

```
===Performance Result===
[Overall Error rate :0.78]
Airplane class rate :0.8
Cars Error rate :0.4
Dogs Error rate :0.8
Faces Error rate :1.0
Keyboard Error rate :0.9

Enter 0-4 to see images that are correctly classified(-1 to skip):-1

Enter 0-4 to see images that are incorrectly classified:-1

---Calculating Confusion Metrics...
[2 1 4 1 2]
[0 6 2 1 1]
[0 4 2 3 1]
[1 9 0 0 0]
[0 4 3 2 1]
```


It is clear to observe a drop in performance when Using Intersection for histograms comparing, whereas the performance improved a bit when using L2 distance.

One of the possible explanations here is, when clustering with a smaller number of clustering centers, with nearly 30,000 descriptors, the actual possible number of cluster centers is likely to be much higher, therefore lots of distinct “words” might be clustered into one single word. Those further effects the labelling stage and hence the classification stage.

Implementation Details | 3

The code of our implementation is mainly consisting of three parts, each has some functions:

Control loop

main_cotrol()

SIFT Feature Extraction

SIFT()	localizeExtremumViaQuadraticFit()
findScaleSpaceExtrema()	computeGradientAtCenterPixel()
remobeDuplicateKeypoints()	ComputeHessianAtCenterPixel()
generateDescriptors()	computeKeypointsWithOrientation()
isPixelAnExtremum()	generateDiscriptors()
unpackOctave()	CompareKeypoints()

Bag-of-Words Classification

L2_distance()	get_class_label()
find_imageIndex_with_index()	histogram_intersection()
clustering()	
labelling_to_mean()	
get_patch()	

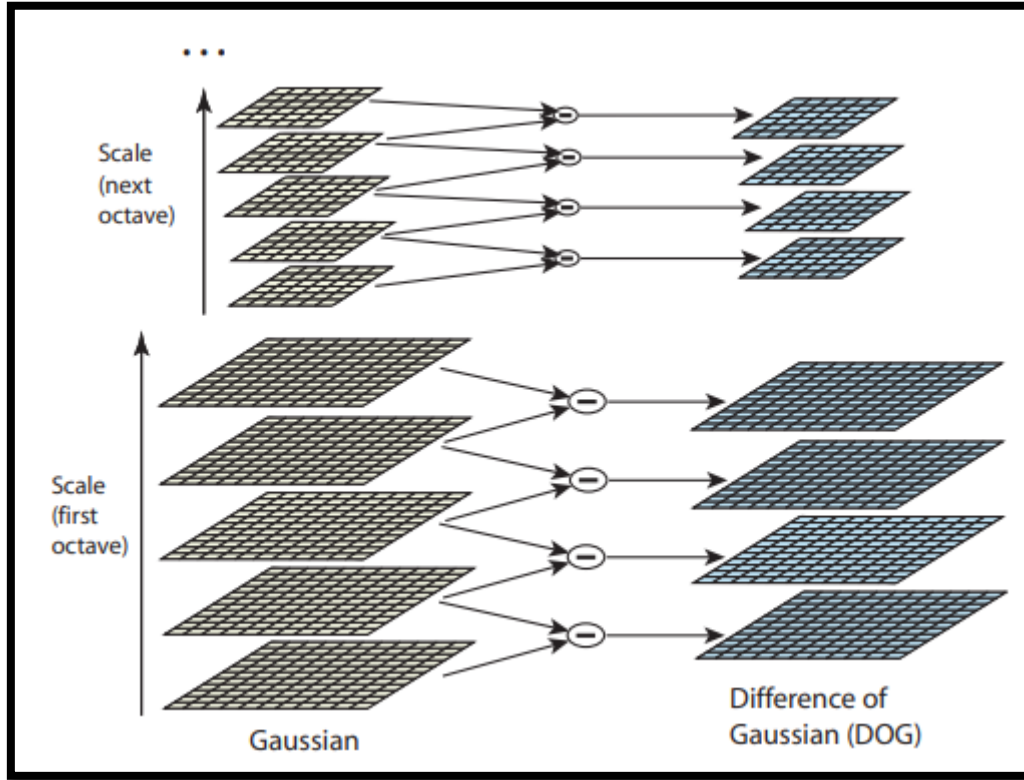
- SIFT Feature Extraction

As mentioned in the project specification, we are not allowed to use the in-build function of SIFT feature extraction but could use some GitHub resources as reference instead. Part of the mathematical implementations are referenced from one of the GitHub resources[1].

Building Gaussians Difference pyramid

This function *SIFT(img)* takes an image matrix as input and returns the extracted SIFT feature descriptors and the corresponding keypoints.

For building the GoD(Gaussian Difference) pyramid, because it is obtained by subtracting from adjacent layers of Gaussian pyramid, as shown below:



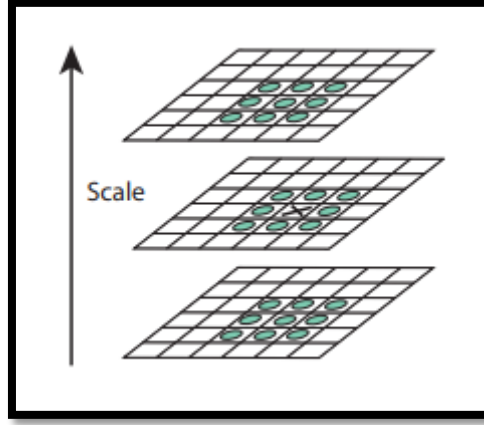
we firstly determine several parameters for later calculations, since we will be going to do the subsampling to deeper octave, the total number of octaves is determined by the size of the image, it can't be too large (causes error when doing subsampling & computationally costly) or too small (likely to have bad performance), therefore it is really a tradeoff.

where according to the David G. Lowe's Paper[2], we take the number of octaves as the follow:

$$\lceil \log_2(\min(M, N)) \rceil - 3$$

The M and N are the number of rows and columns of the image, respectively.

We want the number of layers in each octave as well, this is obtained by determining how many numbers of times we want to later search for to find an extremum. Since we are going to search local extremes using a 3x3x3 volume, it is required that each search takes 3 layers of images, as shown below:



Thus, this number is determined by:

$$S = n + 3$$

S is number of layers; n is the number of layers we want to search for in the Dog pyramid. In our implementation, we have taken $n = 2$.

Furthermore, David G. Lowe's Paper states that we need to determine the so called "Basic Camera Blur" value to represent the value of sigma applied on the image once the image was produced by the camera. We need to use this value and basically undo this effect and apply the different sigma of blur on each image in the layer of each octave. From his paper, it is recommended to use Basic Camera Blur $\sigma = 0.5$. David G. Lowe's Paper also suggested the base blur sigma for the bottom image of pyramid $\sigma = 1.6$.

Therefore, according to successive gaussian blur method, blurring an input image by kernel size σ_1 and then blurring the resulting image by σ_2 is equivalent to blurring the input image just once by σ , where $\sigma^2 = \sigma_1^2 + \sigma_2^2$, so we need to blur the input image with sigma $\sigma=1.52$ and put it at the bottom of our Gaussians pyramid.

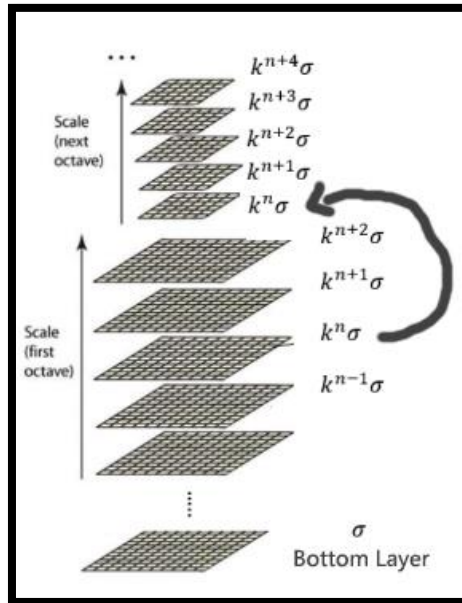
$$\sqrt{1.6^2 + 0.5^2} = 1.52$$

For each layer in the octave, we need to use different values of sigma for blurring the image, David G. Lowe's Paper suggests that we define a constant coefficient K to do so, and K is depending on small "n", which as mentioned above, it is the number of times we want to later search for in the layer.

The coefficient K is defined as:

$$k = 2^{1/n}$$

For each further layer in the octave, we apply this coefficient by multiply it to the previous sigma and use the sigma of reciprocal third image from previous layer as bottom layer for the further(next) octave. Illustrated below:



This is achieved also by successive gaussian blur method, shown below our code for creating a list of sigma values for a certain octave:

```
# Genrate Kernels
gaussian_kernels = np.zeros(numLayers)
gaussian_kernels[0] = baseSigma

for index in range(1,numLayers): # Sigmas to apply within an octave
    last_sigma = (k ** (index - 1)) * baseSigma
    sigma_total = k * last_sigma
    gaussian_kernels[index] = mt.sqrt(sigma_total ** 2 - last_sigma ** 2)
```

Applying above gaussian kernels to each image in the Gaussians pyramid and subtract adjacent layers to get an image of Gaussians Difference pyramid.

Find Local Extremums & Get Keypoints` precise location within continues space & Find Principal Orientation

We need to find the local extremums in GoD pyramid, as mentioned in previous step, we are searching for this value in an 3x3x3 volume, but before that, as stated in David G. Lowe`s Paper, it is important to define a threshold to filtering possible noises(fake extremums), this value is defined as:

$$\text{abs(val)} > 0.5 \cdot T/n$$

Both in OpenCV`s implementation and David G. Lowe`s Paper, T was taken value

T=0.04, this is hence what our implementation used as well.

The function *findScaleSpaceExtrema(gaussian_images ,dog_images ,num_intervals, sigma, image_border_width, contrast_threshold)* together with function *isPixelAnExtremum(first_image, second_image, third_image, threshold)* does the job by searching in the scale space with a 3x3x3 volume, and:

1.then locate the precise location of the points found by function *localizeExtremumViaQuadraticFit(I ,j, image_index, octave_index, num_intervals, dog_image_in_octave, sigma, contrast+threshold, image_boder_width, eigenvalue_ratio=10)*

During the searching process, because for now we only have the location of local extremum in terms of discrete scale space coordinates, it will be unprecise since the idea of using image pyramid is we want scale-invariant, so ideally the scale space should be infinitely big and be continues. David G. Lowe's Paper and the reference we found on GitHub solves the question in a very clever mathematical approximation to locate the exact location of those keypoints found in discrete space and mapping them to continuous space. In other words, we want to find those locations in sub-pixel level.

Quite like the mechanism used in Canny Edge Detection., this function basically performs a ternary quadratic Tylor expression on the discrete location found at:

$$X_0(x_0, y_0, \sigma_0)^T$$

x_0 is the row number of the particular image in GoD scale space, y_0 is the column number, σ_0 is the size(sigma) of that image.

Then the function performs the following Tylor expression:

$$f\left(\begin{bmatrix} x \\ y \\ \sigma \end{bmatrix}\right) = f\left(\begin{bmatrix} x_0 \\ y_0 \\ \sigma_0 \end{bmatrix}\right) + \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial \sigma}\right] \left(\begin{bmatrix} x \\ y \\ \sigma \end{bmatrix} - \begin{bmatrix} x_0 \\ y_0 \\ \sigma_0 \end{bmatrix}\right) + \frac{1}{2} \left(\begin{bmatrix} x \\ y \\ \sigma \end{bmatrix} - \begin{bmatrix} x_0 \\ y_0 \\ \sigma_0 \end{bmatrix}\right)^T \begin{bmatrix} \frac{\partial^2 f}{\partial x \partial x} & \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial x \partial \sigma} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y \partial y} & \frac{\partial^2 f}{\partial y \partial \sigma} \\ \frac{\partial^2 f}{\partial x \partial \sigma} & \frac{\partial^2 f}{\partial y \partial \sigma} & \frac{\partial^2 f}{\partial \sigma \partial \sigma} \end{bmatrix} \left(\begin{bmatrix} x \\ y \\ \sigma \end{bmatrix} - \begin{bmatrix} x_0 \\ y_0 \\ \sigma_0 \end{bmatrix}\right)$$

Which is equivalent to the following scalar form equation:

$$f(X) = f(X_0) + \frac{\partial f^T}{\partial X} \hat{X} + \frac{1}{2} \hat{X}^T \frac{\partial^2 f}{\partial X^2} \hat{X}$$

Moreover, we then need to differentiate the above equation to find the gradient of the current point, according to Tylor expression we have above, we can deduct the following differentiation steps for a certain point

$$\frac{\partial f(X)}{\partial X} = \frac{\partial f^T}{\partial X} + \frac{1}{2} \left(\frac{\partial^2 f}{\partial X^2} + \frac{\partial^2 f^T}{\partial X^2} \right) \hat{X} = \frac{\partial f^T}{\partial X} + \frac{\partial^2 f}{\partial X^2} \hat{X}$$

Let $\frac{\partial f(X)}{\partial X}$ approaching zero, we then have the value of this local extremum:

$$\hat{X} = -\frac{\partial^2 f^{-1}}{\partial X^2} \frac{\partial f}{\partial X}$$

Substitute the above into $f(x)$, we have:

$$\begin{aligned} f(X) &= f(X_0) + \frac{\partial f^T}{\partial X} \hat{X} + \frac{1}{2} \left(-\frac{\partial^2 f^{-1}}{\partial X^2} \frac{\partial f}{\partial X} \right)^T \frac{\partial^2 f}{\partial X^2} \left(-\frac{\partial^2 f^{-1}}{\partial X^2} \frac{\partial f}{\partial X} \right) \\ &= f(X_0) + \frac{\partial f^T}{\partial X} \hat{X} + \frac{1}{2} \frac{\partial f^T}{\partial X} \frac{\partial^2 f^{-1}}{\partial X^2} \frac{\partial^2 f}{\partial X^2} \frac{\partial^2 f^{-1}}{\partial X^2} \frac{\partial f}{\partial X} \\ &= f(X_0) + \frac{\partial f^T}{\partial X} \hat{X} + \frac{1}{2} \frac{\partial f^T}{\partial X} \frac{\partial^2 f^{-1}}{\partial X^2} \frac{\partial f}{\partial X} \\ &= f(X_0) + \frac{\partial f^T}{\partial X} \hat{X} + \frac{1}{2} \frac{\partial f^T}{\partial X} (-\hat{X}) \\ &= f(X_0) + \frac{1}{2} \frac{\partial f^T}{\partial X} \hat{X} \end{aligned}$$

Since we are doing this in discrete space of image matrix, therefore, the differentiations are only achievable by subtracting the pixel values for surrounding pixels, this is done by function `computeGradientAtCenterPixel(pixel_array)`.

Then by comparing the gradient, we can obtain the local extremum points in continuous space.

Except above operations, this function also helps to remove noise keypoints while locating its position in sub-pixel level, it is recommended by David G. Lowe's Paper, that if a point is with low contrast value with its surroundings, then we drop this point, it is calculated by:

$$|f(X)| < \frac{T}{n}$$

If so, drop the point.

Another thing this method has done is to remove so called "Edge Effects", by calling `computeHessianAtCenterPixel(pixel_array)`, one does this by calculating the local curvity of the point via Hessian's Metrix:

$$\mathbf{H}(x, y) = \begin{bmatrix} D_{xx}(x, y) & D_{xy}(x, y) \\ D_{xy}(x, y) & D_{yy}(x, y) \end{bmatrix}$$

$$\text{Tr}(\mathbf{H}) = D_{xx} + D_{yy} = \alpha + \beta$$

$$\text{Det}(\mathbf{H}) = D_{xx}D_{yy} - (D_{xy})^2 = \alpha \beta$$

$$\frac{\text{Tr}(\mathbf{H})^2}{\text{Det}(\mathbf{H})} = \frac{(\alpha + \beta)^2}{\alpha\beta} = \frac{(\gamma\beta + \beta)^2}{\gamma\beta^2} = \frac{(\gamma + 1)^2}{\gamma}$$

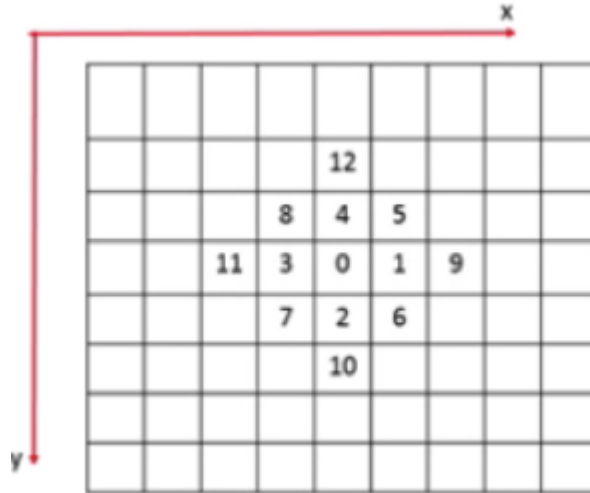
In which, $\alpha > \beta$ & $\alpha = \gamma\beta$.

It is recommended in David G. Lowe's Paper, that $\gamma=10.0$. Then if: **Det(H)** < 0, we drop the point, or if **Tr(H)** / Det(H)** does not hold the following:

$$\frac{\text{Tr}(\mathbf{H})}{\text{Det}(\mathbf{H})} < \frac{(\gamma + 1)^2}{\gamma}$$

Then we also drop the point.

With above being known, finally, the function computeHessianAtCenterPixel uses an approximation for doing differentiations, as mentioned above, which can be illustrated as following:



$$\left(\frac{\partial f}{\partial x}\right) = \frac{f_1 - f_3}{2h} \quad (1)$$

$$\left(\frac{\partial f}{\partial y}\right) = \frac{f_2 - f_4}{2h} \quad (2)$$

$$\left(\frac{\partial^2 f}{\partial x^2}\right) = \frac{f_1 + f_3 - 2f_0}{h^2} \quad (3)$$

$$\left(\frac{\partial^2 f}{\partial y^2}\right) = \frac{f_2 + f_4 - 2f_0}{h^2} \quad (4)$$

$$\left(\frac{\partial^2 f}{\partial x \partial y}\right) = \frac{(f_1 + f_6) - (f_3 + f_7)}{4h^2} \quad (5)$$

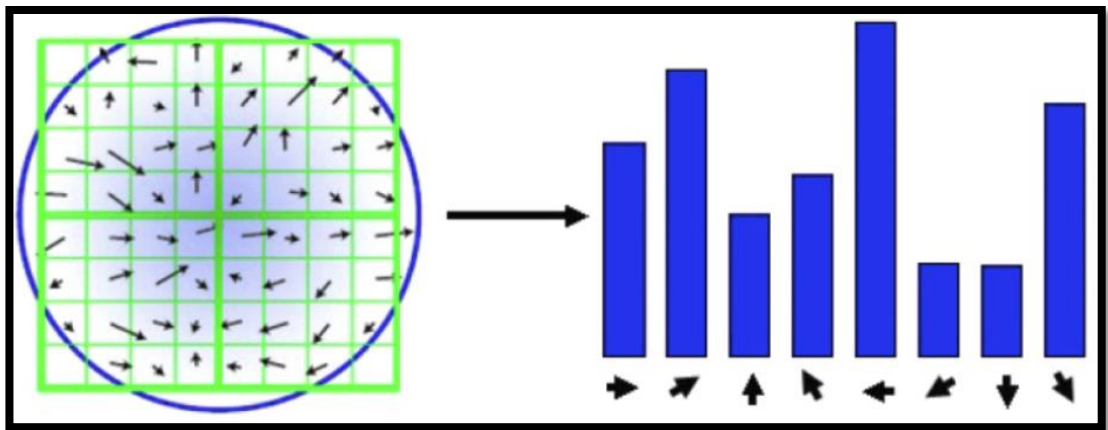
$$\left(\frac{\partial f}{\partial \sigma}\right) = \frac{f_2 - f_4}{2h} \quad (6)$$

$$\left(\frac{\partial^2 f}{\partial \sigma^2}\right) = \frac{f_2 + f_4 - 2f_0}{h^2} \quad (7)$$

$$\left(\frac{\partial^2 f}{\partial x \partial \sigma}\right) = \frac{(f_1 + f_6) - (f_3 + f_7)}{4h^2} \quad (8)$$

$$\left(\frac{\partial^2 f}{\partial y \partial \sigma}\right) = \frac{(f_1 + f_6) - (f_3 + f_7)}{4h^2} \quad (9)$$

2. Find the principal direction of current keypoint by calling function:
computeKeypointsWithOrientation(keypoint, octave_index, gaussian_image,
radius_factor=3, num_bins=36, peak_ratio=0.8, scale_factor=1.5)



The function takes a window around the keypoint with its precise location, then uses radius=3 to further apply a circle, for every pixels in the circle, the function computes the gradient direction and magnitude, use 8 bins to represent each direction boundary, and vote the magnitudes from every pixels in the window to corresponding range to form an

histogram, as shown above. Then assign the point with the direction of the bin with the highest votes.

The function also performs an Gaussians Kernel to the window's pixels, to reduce the effects according to the distance from the center pixel.

Eventually, *findScaleSpaceExtrema()* returns the keypoints with exact continuous pixel location and its orientation.

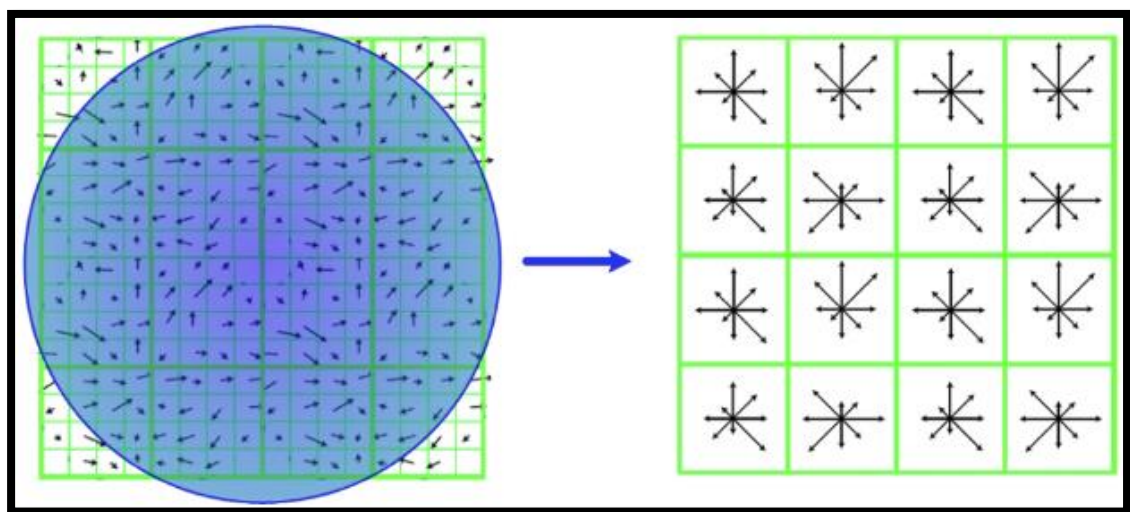
Generating Keypoints Descriptors

With keypoints been found, the *SIFT()* function then calls

generateDescriptor(keypoints, Gaussians_pyramid)

to generate the descriptors if the keypoints.

The function basically takes a window of 4*4 grid around the keypoint, each grid is with shape 4*4. For each grid of 4*4=16 pixels within the window, calculate the orientations(directions) as we did in *computeKeypointsWithOrientation()* , to form a histogram of 4*4*8 = 128 bins, and returns the histogram as descriptor of the keypoints.



- **Bag-of-Words Classification**

We now use BOW to represent each image by processing the extracted features into histograms, and by calculating the distances between histograms we can then performs the classification.

Note that this part is finished all in main control's loop.

K-Means Clustering

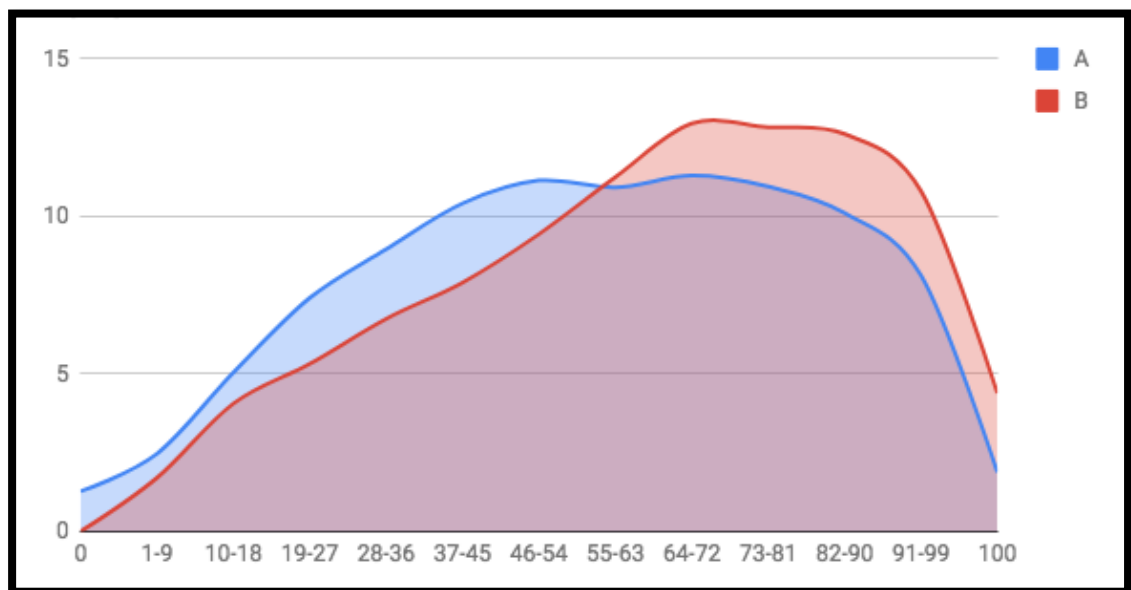
It is worth pointing out that clustering(K-Mean Algorithm) part of the code was initially done by ourselves, however because of the time efficiency, we changed it into one of the external library[3] function of K-Mean algorithm instead. However, our original attempt for the K-Means algorithm can also be found, namely `clustering(descriptors, words=Num_WORDS, CODE="N")`. Where the CODE parameter is used when the user wants to continue his clustering from a certain finished round, since it takes ages to finish one round.

Nearest neighbor classification

We have used both L2 distance and histogram intersection method to apply Nearest Neighbor classifier, where for the previous one we assign the class of minimum L2 distance value to the requesting histogram, for the later one we assign the class of maximum value returned by the histogram intersection method.

Comparing Histograms via Histogram Intersection

The function `histogram_intersection(h1,h2)` deals with this part, and it is following the illustration shown below:



Since the histograms are discrete, we take the minimum of each bins of two histograms, sum them up and returns.

Evaluation Mechanism

The error rate is calculated by:

$$\frac{\textit{Number of Error}}{\textit{Total Number}}$$

Performance Evaluation | 4

In our implementation, the program classifies the images with error rate varies from 75% to 63% according to different histogram comparing methods and number of clustered centers.

Which is very sensible, since the number of images used for training is limited (350), 70 per class. As well as the watermarks in some of the images are effecting the performance as well.

Group Contribution | 5

Yang Zhang: Requirements Walkthrough

Yuhang Song: SIFT & BOW Coding & Implementation Details

Shenpu Zhou: Clustering Algorithm & Evaluation & Testing

Yiming Li: Project Description & References

Shaokun Duan: Control loop Coding

References | 6

Mentioned:

GitHub resources[1]:

<https://github.com/rmislam/PythonSIFT>

David G. Lowe`s Paper[2]:

<https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf>

External library[3]:

<https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html>

Implicit:

Keypoint obj by OpenCV:

https://docs.opencv.org/3.4/d2/d29/classcv_1_1KeyPoint.html

Taylor series:

https://en.wikipedia.org/wiki/Taylor_series

K-Means Algorithm & KNN:

https://en.wikipedia.org/wiki/K-means_clustering

<https://towardsdatascience.com/a-simple-introduction-to-k-nearest-neighbors-algorithm-b3519ed98e>