Vision and image processing
Assignment 4: Content Based Image Retrieval

University of Copenhagen

January 13, 2021

Mark Ambrodji
qpf823@alumni.ku.dk

Francesco Done'
qwg586@alumni.ku.dk
Exchange Student

Michelle Low
mdc577@alumni.ku.dk

Polina Olander
wdb694@alumni.ku.dk

Introduction

For the assignment, we have used the $Caltech$ 101 image dataset that includes 101 categories, each one ranging from 40 up to 800 images with roughly size 300 x 200 pixels. The number of categories to choose has been set to 20 and the number of images per category has been set to 40 and maximum (the categories that we have chosen, have only color and excluded grayscale images), We used the following 20 categories: barrel, bonsai, anchor, binocular, ant, bass, BACKGROUND_Google, airplanes, accordion, beaver, ceiling_fan, brain, butterfly, chair, cellphone, brontosaurus, car_side, cannon, camera and buddha.

The report is divided into three main sections, the first one concerning the generation of a codebook; the second one related to context indexing; finally, in the last one, the retrieving of images is explained.

1 Codebook Generation

Firstly the images have been loaded by the program and divided equally for each category into a unique $training$ set and a $test$ set. After that, iteratively for each image, the $SIFT$ features have been calculated through $OpenCV$ $SIFT$ function `cv.xfeatures2d.SIFT_create()`. In particular, the descriptors for each image have been computed by the function `sift.detectAndCompute(...)`. The script used for this purpose is as follows:

```
def compute_des_list(data_set):
  #initialize output list
  des_list = []
  #generate SIFT
  sift = cv.xfeatures2d.SIFT_create()
  for img in data_set:
    for i in range(len(img)):
    #extract the SIFT descriptors for
        each image
    _, des =
        sift.detectAndCompute(img[i],
        None)
    des_list.append((img[i], des))
  return des_list
```

Thanks to that, it has been possible to run the K-means algorithm on the retrieved descriptors. For this purpose the $sklearn.cluster.KMeans$ has been used, our code is:

```
#prepare the data for using in kmeans
    algorithm
descriptors = train_des_list[0][1]
for _, descriptor in train_des_list[1:]:
  descriptors = np.vstack((descriptors,
      descriptor))

#calculate kmeans on train images
    descriptors
kmeans = KMeans(n_clusters=k,
    random_state=0,
    verbose=1).fit(descriptors)
```

Since that function requires, in input the number of clusters, k, we have used different values such as k = 500, 1000 and 2000 since the assignment constraint was to set k between 500 and 2000. Furthermore, we used different amount images to test the impact of this on the final result. We used 20, 40 and all images from the 20 first categories for this.

We found the clusters using sklearn's KMeans algorithm we found it to be simple, more precisely because this implementation can fit and predict. With this we were practically skipping the calculation of computing the distance from each image to the cluster, to find the closest clusters, thereby which cluster the image likely belongs to.
With the function `kmeans.predict(img)`, a prediction on the closest cluster has been calculated for every image inside the training set, so then it has been possible to calculate the BoW with our script:

```
def compute_bow(des_list):
  #initialize output list
  bow_list = []
  #generate SIFT
  sift = cv.xfeatures2d.SIFT_create()
  for img in des_list:
    #extract key point and descriptors
        for each image
    kp, des =
        sift.detectAndCompute(img[0],
        None)
    bow = np.zeros(k)
    nkp = np.size(kp)

    #calculate the prediction with the
        i-th image
    pred = kmeans.predict(img[1])

    for clust in pred:
      bow[clust] += 1/nkp # percentage BOW
      # bow[clust] += 1 # actual BOW

    bow_list.append(bow)
  return bow_list
```

The histogram of an image of a barrel from the training set is shown in the Figure 1 and the image of the barrel can be seen in Figure 2

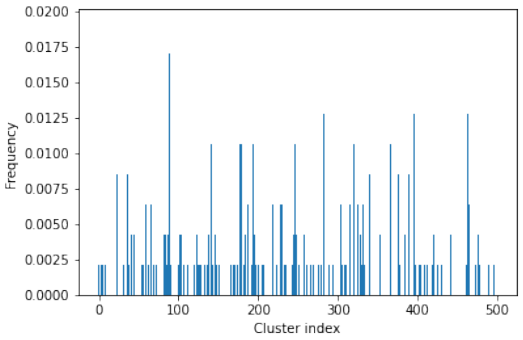

Figure 1: BoW histogram of the a image of a barrel.

2 Indexing

For this task, the same codebook as before has been used, so we succeeded in doing the following steps: first, we computed the descriptors list taking into account the test set by calling the function `compute_des_list(test)`; secondly, we projected these descriptors onto the codebook by running `compute_bow(test_des_list)`; finally, we generated the histogram as before, but regarding the test set.

3 Retrieving

To implement retrieving of images, we used the $Kullback-Leibler$ and $Bhattacharyya$ distance similarity measure for normalized histograms as

Figure 2: Image (image_0041.jpg) of a barrel from the Caltech 101 image dataset.

described in the lecture slides, the formulas are:

$$Bhattacharyya = d(v_1, v_2) = \sum_{i=1}^{K} \sqrt{v_{1i} v_{2i}}$$

$$Kullback - Leibler = D_{kl}(v_1 || v_2) = \sum_{i=1}^{K} v_{1i}(\ln\ v_{1i} - ln\ v_{2i})$$

We transformed the $Kullback - Leibler$ using log Quotient rules to avoid division by zero error. We used these as they essentially both are distance measures, except one of them is non-symmetric. Having the same output type makes it easier to make general functions for the errors.

To accomplish this task, we first implemented the function `compute_similarities`, which receives three parameters; a subset of the dataset (`df_sub`), the full dataset (`df_full`) and lastly a function (`fun`). `df_sub` should either be the training set or the test set, i.e. a subset of `df_full`. `df_full` should be the full dataset, which was computed in the previous task and `fun` should be the desired similarity measure function. In the function, we compute the similarity between each

BOW from the subset and each BOW from the entire dataset. For each image in the subset, we will get a list of similarities between the image and the rest of the images in the entire dataset. These lists are sorted going from most similar to least similarity. Lastly, the lists are added to our pandas dataframe in a new column called "distance list".

To compute the mean reciprocal rank and how often the correct category is in top-3 in percent, we implemented the functions `reciprocal_rank` and `is_top3` and applied these to our dataframe. The function `reciprocal_rank` simply finds the rank (index of the first occurrence of an image's true category) and returns the reciprocal rank. `is_top3` returns either 1 or 0 for an image to indicate if the image's true category is in top-3 of its distance list (1) or not (0). After applying these functions to our dataframe, we get two new columns "reciprocal_rank" and "is_top3". To compute the mean reciprocal rank and the percentage of how often the correct category is in top-3, we simply compute the average of their respective columns. In table 1 and table 2 the results of the experiments can be seen.

The NAN values below are due to how long it took to run the code. We did not make the last calculation in time.

Conclusion

We see from section 3 Table 1 and Table 2 that given the number of images for the training and test set, the k value affects the result substantially. As k rises, the accuracy of our retrieval models falls. This is likely due to the relatively small amount of images. It seems that the k value should strongly depend on the amount of images and/or categories as too many cluster would dilute the correct predictions and too few clusters would fail to capture the unique features of each category.

From the tables we have a result for using all images from each category with a rather small k value of 500. This increases the results substantially; partially because we have more images to train on, to learn the features better, and partially because the amount of data and the k value are balanced better.

We realize that the numbers for our Kullback distance are bad, and this is likely because we did not implement the conversion from non-symmetric to symmetric. Though as this would essentially average the error between the 2, then pull in one direction or the other is not as substantial. Meaning that the result would likely be just as bad or not far from it either way.

	Train		Test	
Parameters	Mean reciprocal rank	Top-3 (%)	Mean reciprocal rank	Top-3 (%)
#Images = 40, k = 500	0.4471	51.62	0.3916	43.06
#Images = 40, k = 1000	0.3242	41.15	0.3515	42.53
#Images = 40, k = 2000	0.2810	26.99	0.3318	31.92
#Images = all, k = 500	0.5050	58.53	0.4418	51.08

Table 1: An overview of the performance using Bhattacharyya distance

	Train		Test	
Parameters	Mean reciprocal rank	Top-3 (%)	Mean reciprocal rank	Top-3 (%)
#Images = 40, k = 500	0.0883	7.32	0.0134	0.0
#Images = 40, k = 1000	0.0805	6.80	0.0412	2.92
#Images = 40, k = 2000	NAN	NAN	NAN	NAN
#Images = all, k = 500	0.0263	1.25	0.0406	2.62

Table 2: An overview of the performance using Kullback distance