CS676A Introduction to Computer Vision

Assignment 3

Scalable Vocabulary Recognition Tree

Saurav Kumar, 12641
Shreyash Pandey, 12683

The problem statement for this assignment requires us to implement a simplified image retrieval and verification system, scalable to large databases. This can be done using a scalable vocabulary recognition tree as described in the paper by Nister and Strewenius in their paper published in CVPR, 2006.

A Vocabulary tree based on 'words' generated by extracting SIFT features of images present in the database, is created using hierarchical k-means. Thus, SIFT features undergo hierarchical quantization, defined at each level by k centers and their Voronoi regions.

SIFT features of query images are then passed down the tree, and quantized according to the leaf they end up in. A TF-IDF scoring scheme is then implemented using the following formulation: $(n_{di}/n_d)*\log (N/N_i)$, where N is the total number of images, N_i the number of images that go to the i^{th} leaf, in the whole database; n_{di} is the number of SIFT features of image d that end up in the i^{th} leaf , and n_d is the number of leaves SIFT features from image d go to. It is to be noted that this scoring scheme is different from the one originally proposed in the paper, but is generally used to for efficient implementation.

Part 1: Generation of Vocabulary Tree

For each image in the database, we extract the SIFT descriptors and recursively cluster them in the feature space using K-Means algorithm. This recursion is carried out L times, where L is the depth factor pre-defined by the user. Each node in the tree has an array of 'children' trees, and a cluster center associated with it. The leaf nodes, additionally, also have an array of TF-IDF scores (computed at a later stage), for every image that ends up at their location.

```
def getAllSiftDescriptors():
    descriptors   = []
    for imagePath in imagePathList:
       img = cv2.imread(imagePath, cv2.COLOR_BGR2GRAY)
       (kps, des) = SIFT.detectAndCompute(img, None)
       for d in des:
           descriptors.append((imgIndex, d))
           imgIndex += 1
    return descriptors
```

```
def genVocabTree(descriptors, level=L):
    vtree = VocabTree()
    if level == 0:
        vtree.descriptors = descriptors
        return vtree
    km = KMeans(n_clusters=K)
    clusters = km.fit(descriptors)
    for (i, d) in descriptors:
        C[clusters.predict(d)].append(d)
    for i in range (0, K):
        vtree.children.append(genVocabTree(C[i], level-1))
        vtree.children[i].center = clusters.cluster_centers_[i]
    return vtree
```

Part 2: Generation of Inverse Document Frequency

The score for an image d, at leaf i, is computed according to the formula : $(n_{di}/n_d)*\log (N/N_i)$, where N is the total number of images, N_i the number of images that go to the i^{th} leaf, in the whole database; n_{di} is the number of SIFT features of image d that end up in the i^{th} leaf , and n_d is the number of leaves SIFT features from image d go to.
Thus an array of scores for various database images is stored at each leaf.

```
def computeNDArray(tree):
    global ND
    if len(tree.children) != 0:
        for child in tree.children:
            computeNDArray(child)
    else:
        tree.imageIndices = [d[0] for d in tree.descriptors]
        tree.images       = list(set(tree.imageIndices))
        for img in tree.images:
            ND[img] += 1
```

```
def computeIFIndex(tree):
    global ND
    if len(tree.children) != 0:
        for child in tree.children:
            computeIFIndex(child)
    else:
        Ni = len(tree.images)
        for img in tree.images:
            ndi = tree.imageIndices.count(img)
            nd  = ND[img]
            tree.scores.append(float(ndi)/nd*math.log(N/Ni))
```

Part 3: Query

For any query image, we extract the SIFT descriptors and traverse the tree - the path being decided by an L2 norm distance measure from the cluster center for each node in the tree. Once a descriptor ends up in a leaf, we use the TF-IDF scores of each database image (at that leaf) to generate the best matches for the query image. This is done by sorting the TF-IDF scores and returning the top k% of the results.

```
def bestMatches(imagePath):
    img = cv2.imread(imagePath, cv2.COLOR_BGR2GRAY)
    (kps, des) = SIFT.detectAndCompute(img, None)
    allVotedImages = {}
    for d in des:
        leaf = getLeaf(VTree, d)
        for (score, imgId) in leaf.topImages:
            if imgId in allVotedImages:
                allVotedImages[imgId] += score
            else:
                allVotedImages[imgId] = score
    votes = [(v, k) for k, v in allVotedImages.iteritems()]
    votes.sort(reverse=True)
    return votes[:topK]
```

```
def getLeaf(tree, descriptor):
    if len(tree.children) != 0:
        index = 0
        minDist = sys.maxint
        for i in range(0, K):
            d = distance(tree.children[i].center, descriptor)
            if d < minDist:
                minDist = d
                index = i
        return getLeaf(tree.children[index], descriptor)
    else:
        return tree
```

Experiments

We implement the code in Python, using the SIFT library from OpenCV, and the K-Means library from Scikit Learn. As specified in the problem statement, we judge the performance on the UKY dataset.

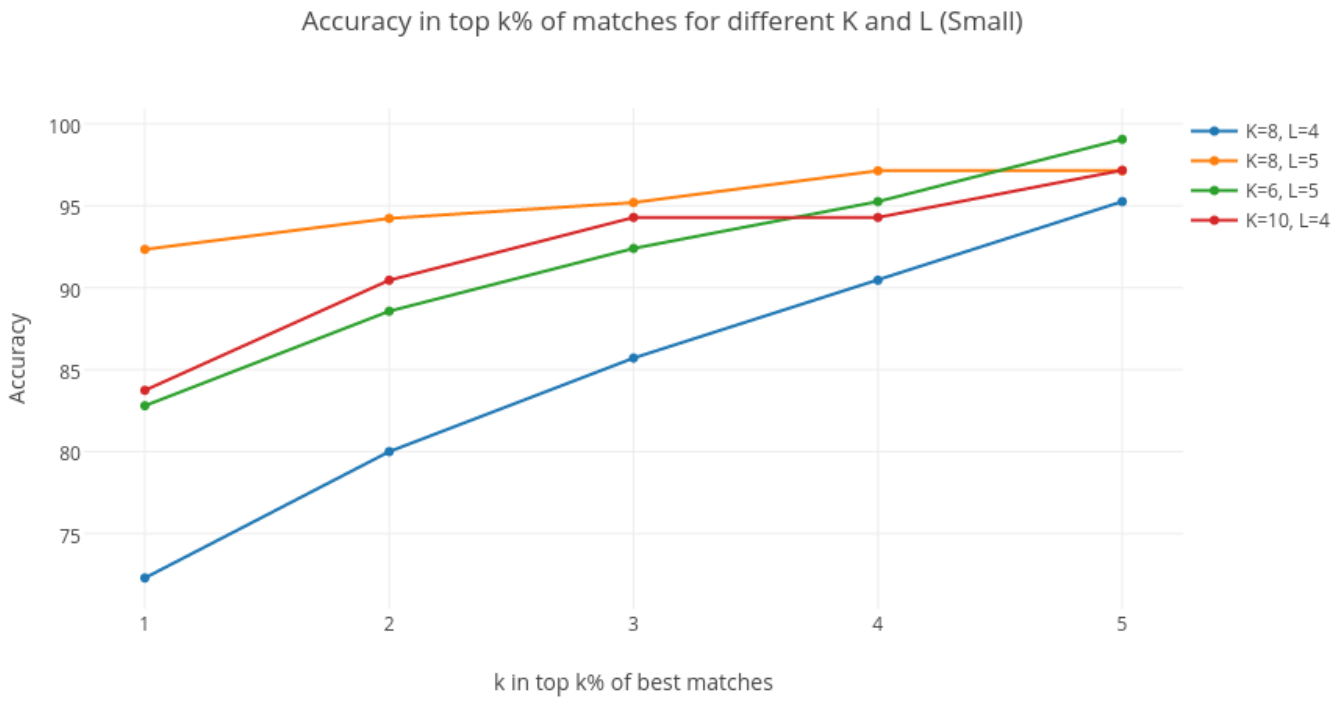
The dataset consists of groups of similar images, with each group consisting of four images. For this assignment, we use three images from each group as database images (for training), and the remaining image from each group as query images (for testing).

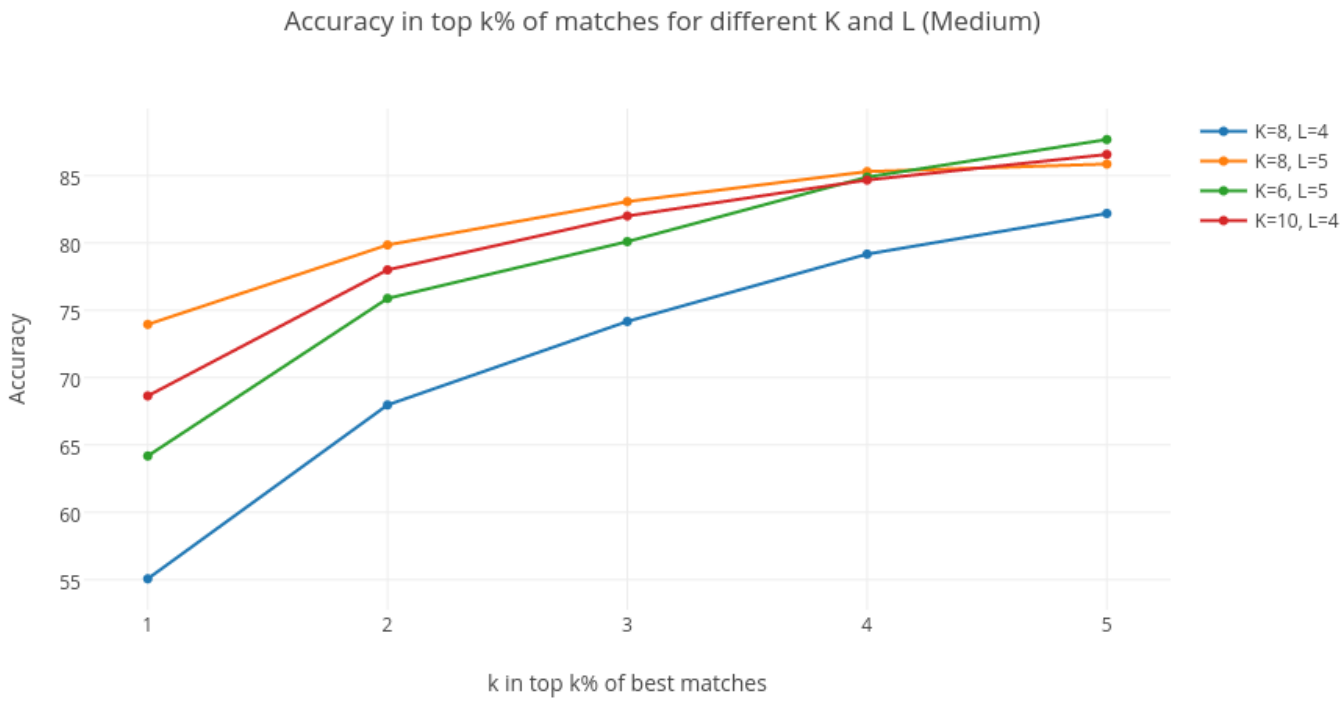
For a measure of accuracy of retrieval, for each query image, we increment the accuracy score by 0.33*n, where n is the number of similar images returned. For example, if all similar images present in the database are returned, it leads to an increment of 1. Finally, this score is averaged over all the query images, and the results are plotted.

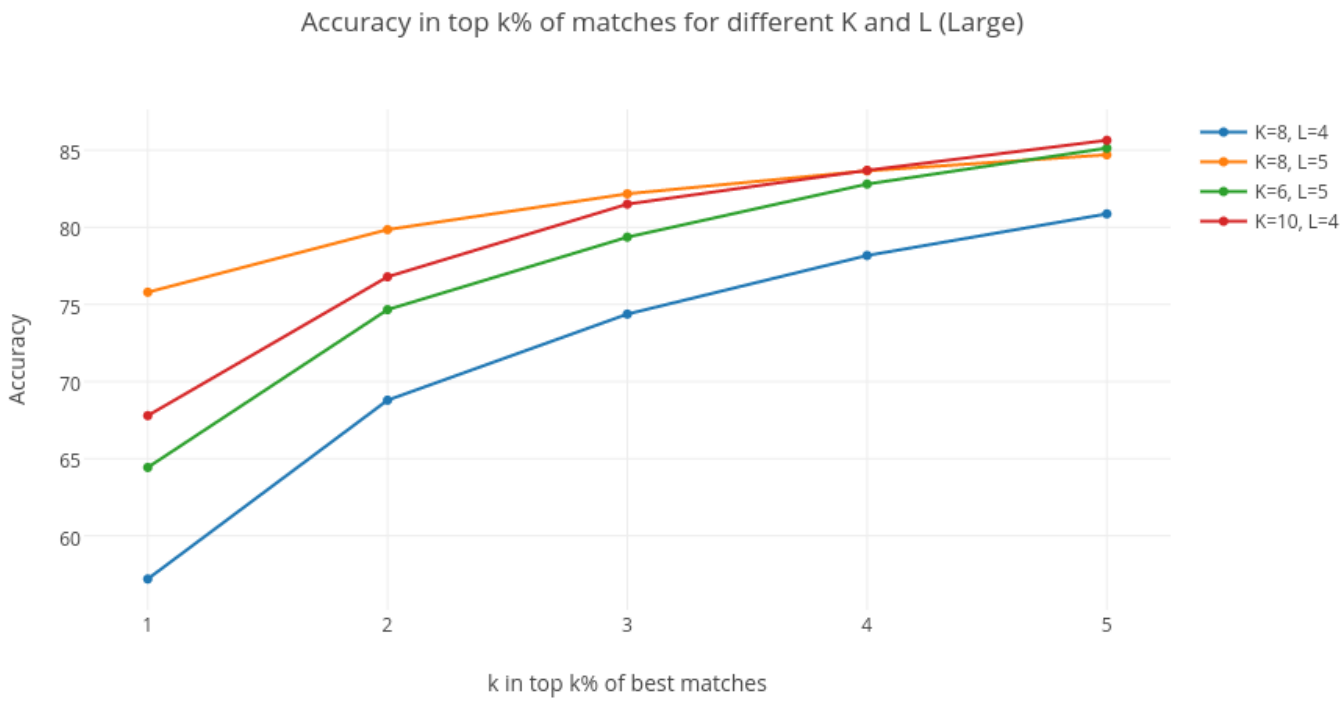
We evaluate the method with different branch factors (K) and depth factors (L) and judge the matching accordingly.

Results and Conclusions

For swifter experimentation, we divided the dataset into small, medium and large sets, with the latter exhausting all the images in the dataset. Small, medium and large set consisted of 35, 300 and 2550 images respectively. The results for each of them are shown below.







As mentioned in the paper, increasing the branching factor(k) and the depth factor(L) leads to higher accuracies. This is expected as it leads to a finer quantization of the feature space.

Also, as number of best matches returned increases, accuracy of retrieval increases. This is also expected as it leads to matching over a larger set of images.

To conclude, we have to appreciate the fact that this is indeed a scalable image retrieval process, as the retrieval times for a query image are of the order of a second even on a database of size greater than 7500 images from UKY dataset. It is to be noted that the retrieval time generally decreases as the branching factor and depth factor are increased. This is again a result of finer quantization of the feature space.

References and Acknowledgments

1. Scalable Recognition with a Vocabulary Tree
   http://www-inst.eecs.berkeley.edu/~cs294-6/fa06/papers/nister_stewenius_cvpr2006.pdf
2. https://www.mpi-inf.mpg.de/fileadmin/inf/d2/HLCV/HLCV_2014/sb_exercise4_0.pdf
3. OpeCV and OpenCV Contribution (github.com/opencv_contrib)