MACHINE PERCEPTION

Assignment 2

Submitted By:

Vivek Mehta(MT2016155)

Vijay Agarwal(MT2016152)

Swatantra Pradhan(MT2016140)

1. Segmentation

- Pick 5 images from Berkeley Segmentation dataset
- Run K-means and Meanshift
- Show results along with the ground-truth

Meanshift looks very similar to K-Means, they both move the point closer to the cluster centroids. One may wonder:
How is this different from K-Means? K-Means is faster in terms of runtime complexity!
The key difference is that Meanshift does not require the user to specify the number of clusters. In some cases,
it is not straightforward to guess the right number of clusters to use. In K-Means, the output may end up having too
few clusters or too many clusters to be useful.
At the cost of larger time complexity, Meanshift determines the number of clusters suitable to the dataset provided.

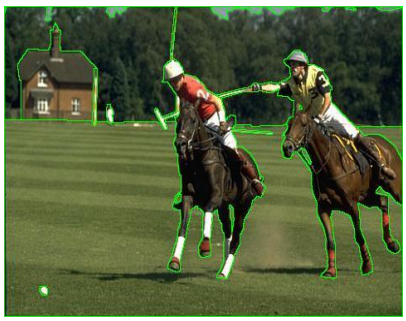
Code:

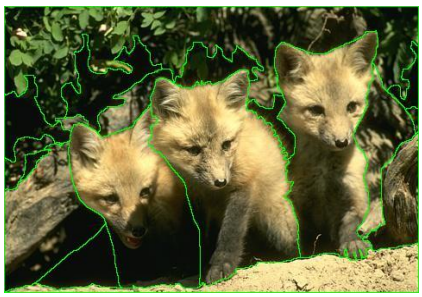
```
# Import numpy and cv2 package
import cv2
import numpy as np
# Importing the library which classifies set of observations into clusters
from scipy.cluster import vq
from sklearn.cluster import MeanShift, estimate_bandwidth

#Segmenting images using the meanShift inbuilt library from sklearn
#here we does not require us to specify the number of clusters.
def calculate_meanshift(image):
    img= cv2.imread(image)
    #flattening the image and converting to array of floats
    vectorized = img.reshape((-1,3))
    vectorized = np.float32(vectorized)
    # Compute clustering with MeanShift
    # bandwidth can be automatically detected using estimate_bandwidth
    bandwidth = estimate_bandwidth(vectorized, quantile=0.1, n_samples=100)
    ms= MeanShift(bandwidth=bandwidth, bin_seeding=True)
    # Perform clustering Samples to cluster.
    ms.fit(vectorized)
    # Labels of each point.
    labels = ms.labels_
    # Coordinates of cluster centers.
    cluster_centers = ms.cluster_centers_
    cluster_centers = np.uint8(cluster_centers)
```

```
    # Assigning each label to one of the clusters
    res= cluster_centers[labels]
    #rebuilding the segmented image from flatten array
    segmented_image= res.reshape((img.shape))
    return segmented_image

#Segmenting images using the kmeans inbuilt library from scipy
# Here we have to explicitly mentions the cluster (5)
def find_kmean(image, k=5):
    img= cv2.imread(image)
    # flattening the image and converting to array of floats
    vectorized= img.reshape((-1,3))
    vectorized = np.float32(vectorized)
    # Performs k-means on a set of observation vectors forming k(5) clusters. This yields a code
book mapping centroids to codes
    center, dist = vq.kmeans(vectorized,k)
    # Converting back to unsigned int
    center = np.uint8(center)
    # Assign codes from a code book to observations.
    # and get length N array holding the code book index for each observation and distance between
the observation and its nearest code.
    code, distance = vq.vq(vectorized, center)
    ## Assigning each label to one of the clusters
    res= center[code]
    # rebuilding the segmented image from flatten array
    f_res= res.reshape((img.shape))
    return f_res

# Find the segmented image  using meanshift  of the image calling calculate_meanshift function and
show the image
result = calculate_meanshift('/home/vivek/PycharmProjects/Assignment2/Input_Images/Seg1.jpg')
cv2.imshow('Mean Shift Seg1 Image', result)

result = calculate_meanshift('/home/vivek/PycharmProjects/Assignment2/Input_Images/Seg2.jpg')
cv2.imshow('Mean Shift Seg2 Image', result)

result = calculate_meanshift('/home/vivek/PycharmProjects/Assignment2/Input_Images/Seg3.jpg')
cv2.imshow('Mean Shift Seg3 Image', result)

result = calculate_meanshift('/home/vivek/PycharmProjects/Assignment2/Input_Images/Seg4.jpg')
cv2.imshow('Mean Shift Seg4 Image', result)

result = calculate_meanshift('/home/vivek/PycharmProjects/Assignment2/Input_Images/Seg5.jpg')
cv2.imshow('Mean Shift Seg5 Image', result)

# Find the Kmean cluster of the image calling find_kmean function and show the image
result = find_kmean('/home/vivek/PycharmProjects/Assignment2/Input_Images/Seg1.jpg')
```

```
cv2.imshow('k=4 kMeans Seg1 image', result)

result = find_kmean('/home/vivek/PycharmProjects/Assignment2/Input_Images/Seg2.jpg')
cv2.imshow('k=4 kMeans Seg2 image', result)

result = find_kmean('/home/vivek/PycharmProjects/Assignment2/Input_Images/Seg3.jpg')
cv2.imshow('k=4 kMeans Seg3 image', result)

result = find_kmean('/home/vivek/PycharmProjects/Assignment2/Input_Images/Seg4.jpg')
cv2.imshow('k=4 kMeans Seg4 image', result)

result = find_kmean('/home/vivek/PycharmProjects/Assignment2/Input_Images/Seg5.jpg')
cv2.imshow('k=4 kMeans Seg5 image', result)


# Display the images and wait till any key is pressed
cv2.waitKey(0)
# Destroy all the windows created by the imshow() function of the OpenCV
cv2.destroyAllWindows()
```

Images:

Ground Truth	MeanShift	K-Means
		
		









2.

- Play with Panorama
- Use this code
- http://ramsrigoutham.com/tag/ransac/
- and stitch together a Panorama
- Explain how SURF is different from SIFT (10 sentences)
- Briefly explain the main principles of FLANN matching (5 sentences)

SURF :-

- SURF takes quite less time for computation.
- SURF is less accurate.
- In SURF we use an integer approximation to the determinant of Hessian blob detector.
- In SURF pyramid, we use different scales of Gaussian masks, while the scale of the image is always unaltered.
- In SURF, we use the sum of the haar wavelet response around the point of interest.

SIFT :-

- SIFT takes more time for computation.
- SIFT takes more time for computation.
- SIFT is more accurate.
- In SIFT we use Difference of Gaussian to build the image pyramid.
- In SIFT pyramids we use different scales of image.
- In SIFT we use an orientation histogram to find the main orientation of the feature descriptor.

Principles of Flann Matcher:

The problem of finding the nearest neighbor is one of the major importance in many of applications. However, solving this problem in high dimensional spaces become very difficult, and there is no algorithm that performs significantly better than the brute force search.

This has led to an increasing interest in a class of algorithms that perform approximate nearest neighbor searches, which have proven to be a good-enough approximation in most practical applications and FLANN in one of them.

FLANN is a library for performing fast approximate nearest neighbor searches in high dimensional spaces. It contains a collection of algorithms which work best for nearest neighbor search and a system for automatically choosing the best algorithm and optimum parameters depending on the dataset.

FLANN based matcher will perform a quick and efficient matching by using the clustering and search in multi-Dimensional Space modules.

This matcher trains a model on a train descriptor collection and calls its nearest search methods to find the best matches.

Images:


LIBRARY


LIBRARY

3.

- Implement Bike vs Horse Classification
- Dataset is available on LMS
- Use Bag-of-visual words approach (SIFT/SURF + K-means + Logistic Regression/KNN)
- Explain the procedure and your approach and observations

Procedure (KNN):
1. Get the path of images in the training set.
2. Extract SIFT features from each and every image in the set.
3. Compute K-Means over the entire set of SIFT features, extracted from the training set.
4. Compute the histogram of features.
5. Train the KNearest classifier with the features (samples) and their corresponding class names (responses).
6. Get the path of image in the test set.
7. Implement step 2 to step 6 for the image in the test set.
8. Now give the Test feature vector and the K value (Number of neighbors. to be considered for classification) to the trained classifier (KNearest).
9. Get the prediction.
10. Print the prediction on to the image in the test data set.

Code:

Features-Extract

```
import cv2
import numpy as np
from scipy.cluster import vq
import os
from os.path import join
import glob
from sklearn.preprocessing import StandardScaler # Importing the library that supports centering and scaling vectors

# Get the path of the training set
train_path =
os.path.abspath('/home/vivek/PycharmProjects/Assignment2/Input_Images/Training_Images')
# Get the training classes names and store them in a list
training_names= os.listdir(train_path)
```

```
print(training_names)

# Get all the path to the images and save them in a list
# image_paths and the corresponding label in image_paths
image_paths = []  # Inilialising the list
image_classes = []  # Inilialising the list
class_id = 0

#Fetching files
def get_imgfiles(path):
                        all_files=[]
                        all_files.extend([join(path,fname)
                                          for fname in glob.glob(path+"/*")])
                        return all_files


for training_name in training_names:
    dir = join(train_path, training_name)
    class_path = get_imgfiles(dir)
    image_paths +=class_path
    image_classes +=[class_id]*len(class_path)
    class_id +=1

feature_detector = cv2.xfeatures2d.SIFT_create()

# List where all the descriptors are stored
descriptor_list = []
# Reading the image and calculating the features and corresponding descriptors
for image_path in image_paths:
    img= cv2.imread(image_path)
    # Computing the key points and the descriptors
    (kps, descs) = feature_detector.detectAndCompute(img, None)
    descriptor_list.append((image_path,descs))  # Appending all the descriptors into the single list

# Stack all the descriptors vertically in a numpy array
descriptors = descriptor_list[0][1]
for image_path, descriptor in descriptor_list[1:]:
    descriptors = np.vstack((descriptors, descriptor))

#Perform k-means clustering
k=500
center, _ = vq.kmeans(descriptors,k,1)

# Calculate the histogram of features
im_features = np.zeros((len(image_paths), k), np.float32 )
for i in range(len(image_paths)):
    code, dist = vq.vq(descriptors, center)
```

```
    for c in code:
        im_features[i][c] +=1

# Perform Tf-Idf vectorization
nbr_occurences = np.sum( (im_features > 0) * 1, axis = 0)
# Calculating the number of occurrences
idf = np.array(np.log((1.0*len(image_paths)+1) / (1.0*nbr_occurences + 1)), 'float32')
# Giving weight to one that occurs more frequently

# Scaling the words
stdSlr = StandardScaler().fit(im_features)
# Scaling the visual words for better Prediction
im_features = stdSlr.transform(im_features)

# Saving the contents into a file
np.savetxt("samples.data",im_features)
np.savetxt("responses.data",np.array(image_classes))
np.save("training_names.data",training_names)
np.save("stdSlr.data",stdSlr)
np.save("voc.data",center)
```

KNN-apply

```
import cv2
import numpy as np
from sklearn import preprocessing, cross_validation, neighbors
from scipy.cluster import vq
import os
from os.path import join
import glob

# Load the classifier, class names, scaler, number of clusters and vocabulary
samples = np.loadtxt('samples.data',np.float32)
responses = np.loadtxt('responses.data',np.float32)
classes_names = np.load('training_names.data.npy')
voc = np.load('voc.data.npy')
k = 50  # Loading the number of cluster

# Training the Knearest classifier with the test descriptors
clf = neighbors.KNeighborsClassifier()
clf.fit(samples,responses)  # Train model using the training samples and corresponding responses

# Get the path of the testing image(s) and store them in a list
test_path =
os.path.abspath('/home/vivek/PycharmProjects/Assignment2/Input_Images/Test_Images')
# Get the training classes names and store them in a list
```

```
image_paths=[]
testing_names= os.listdir(test_path)
#Fetching files
def get_imgfiles(path):
                        all_files=[]
                        all_files.extend([join(path,fname)
                                                for fname in glob.glob(path+"/*")])
                         return all_files

for testing_name in testing_names:
    dir = join(test_path,testing_name)
    class_path = get_imgfiles(dir)
    image_paths +=class_path

feature_detector = cv2.xfeatures2d.SIFT_create()
# List where all the descriptors are stored
descriptor_list = []
# Reading the image and calculating the features and corresponding descriptors
for image_path in image_paths:
    img= cv2.imread(image_path)
     # Computing the key points and the descriptors
    (kps, descs) = feature_detector.detectAndCompute(img, None)
    # Appending all the descriptors into the single list
    descriptor_list.append((image_path,descs))

# Stack all the descriptors vertically in a numpy array
descriptors = descriptor_list[0][1]
for image_path, descriptor in descriptor_list[1:]:
    descriptors = np.vstack((descriptors, descriptor))

# Computing the histogram of features
test_features = np.zeros((len(image_paths), k), np.float32)
for i in range(len(image_paths)):
    words, distance = vq.vq(descriptor_list[i][1],voc)
    for w in words:
        test_features[i][w] += 1  # Calculating the histogram of features


# Perform Tf-Idf vectorization
# Getting the number of occurrences of each word
nbr_occurences = np.sum( (test_features > 0) * 1, axis = 0)
# Assigning weight to one that is occurring more frequently
idf = np.array(np.log((1.0*len(image_paths)+1) / (1.0*nbr_occurences + 1)), 'float32')


# Perform the predictions
```

```
results= clf.predict(test_features)

if results[0][0] == 0:  # results[0][0] will have the predicted class
    prediction = "Horse"
else:
    prediction = "Bike"

accuracy = clf.score(results, responses)
print(accuracy)
# we are getting 89% as we increase the no of clusters our accuracy increases to 94%
```

Observations:

- Observed that this classifier is less efficient in predicting the class.
- SIFT feature extraction is not efficient in case of blur.
- The accuracy is based on the number of clusters as we increase the no of cluster accuracy increases
- The K value is directly proportional to the time complexity of training the classifier i.e. more the K value more is the time to compute the cluster. It took around 45 minutes to get the vocabulary from all the training images with k value of 500 (Machine specs: 4GB RAM, i3 processor).
- The time complexity is reduced if SURF feature extractor is used in place of SIFT
- The prediction also depends on the number of neighbors that I am taking for predicting the class of the test image.
- For predicting a complex feature set, we need more training examples when compared to the number of training examples required to predict a relatively simpler feature set.
- Also, suspect that if we increase the number of training examples for bike, then we can even increase kNN accuracy further.