

# CSCI-B657

## Assignment 3: Object detection

Raghuveer Krishnamurthy Kanchibail  
Supreeth Keragodu Suryaprakash  
Suhas Jagadish

March 29, 2016

### Part 1: A simple baseline

1. Subsample the images to a fixed size. Then convert each image into a vector by concatenating the rows of the image, e.g. converting a 40x40 pixel image into a 1600-dimensional vector.

Solution: In order to subsample the images, we use the "extract\_feature" function defined in "Nearest-Neighbor.h" header.

- This function will extract the features of the input image by resizing the image to 40\*40 pixel and unrolls the instance of the image on x-axis.
- Results are stored in a vector "sample" of type CImg<double>.

2. Call an SVM library or program to train an SVM on this task.

Solution: We downloaded the SVM\_multiclass classifier from: [https://www.cs.cornell.edu/people/tj/svm\\_light/svm\\_multiclass.html](https://www.cs.cornell.edu/people/tj/svm_light/svm_multiclass.html). We tested the classifier using the example problem provided in the website and studied the input file format.

- The vector generated above is written to a output file such that the format is compatible with the input format of SVM(Class1 Pixel1:Feature1 Pixel2:Feature2 ...).
- We use the below system() call to execute the program directly on the command line and parse the output files to the SVM classifier.

```
system("cd SVM/ && ./svm_multiclass_learn -c 5000 train.dat model");
```

3. Then, given a new image, your testing program should apply the SVM to estimate the correct class for each test image. How well does your program work, both quantitatively and qualitatively? Does it make a difference if you use color or not?

Solution: Once the SVM model is generated, we test our program on the test images using the below system call-

```
system("cd SVM/ && ./svm_multiclass_classify test.dat model prediction");
```

- We perform the entire operation(obtaining the feature points and calling SVM classifier) for both grayscale and color images. The runtime and accuracy changes each time quantitatively and qualitatively.
- For color images, we get an accuracy of 19% with a runtime of 20 seconds. For grayscale images, we get an accuracy of 11% with a runtime of 10 seconds. Hence we can say that the program works well

quantitatively and qualitatively for color images as compared to grayscale.

- Entire code for this implementation is stored in "Baseline.h" and we call the train and test functions using "a3.cpp".
- In order to run the grayscale algorithm, please use the algo name as "baseline\_b" and to run the color algorithm, use the algo name as "baseline\_c".

```
Final epsilon on KKT-Conditions: 0.09840
Upper bound on duality gap: 492.02429
Dual objective value: dval=1.37281
Primal objective value: pval=493.39710
Total number of constraints in final working set: 601 (of 873)
Number of iterations: 874
Number of calls to 'find_most_violated_constraint': 43750
Number of SV: 576
Norm of weight vector: |w|=1.65723
Value of slack variable (on working set): xi=0.02104
Value of slack variable (global): xi=0.09840
Norm of longest difference vector: ||Psi(x,y)-Psi(x,ybar)||=4422.10102
Runtime in cpu-seconds: 427.08
Final number of constraints in cache: 6250
Compacting linear model...done
Writing learned model...done
```

Figure a. Train screenshot for Baseline color

```
Reading model...done.
Reading test examples... (250 examples) done.
Classifying test examples...done
Runtime (without IO) in cpu-seconds: 0.18
Average loss on test set: 82.4000
Zero/one-error on test set: 82.40% (44 correct, 206 incorrect, 250 total)
```

Figure b. Test screenshot for Baseline color

```
Final epsilon on KKT-Conditions: 0.09932
Upper bound on duality gap: 496.61985
Dual objective value: dval=8.21211
Primal objective value: pval=504.83196
Total number of constraints in final working set: 1030 (of 2234)
Number of iterations: 2235
Number of calls to 'find_most_violated_constraint': 85000
Number of SV: 996
Norm of weight vector: |w|=4.05370
Value of slack variable (on working set): xi=0.02770
Value of slack variable (global): xi=0.09932
Norm of longest difference vector: ||Psi(x,y)-Psi(x,ybar)||=4941.41495
Runtime in cpu-seconds: 1510.79
Final number of constraints in cache: 12500
Compacting linear model...done
Writing learned model...done
```

Figure c. Train screenshot for Baseline grayscale

```
Reading model...done.
Reading test examples... (500 examples) done.
Classifying test examples...done
Runtime (without IO) in cpu-seconds: 0.24
Average loss on test set: 89.0000
Zero/one-error on test set: 89.00% (55 correct, 445 incorrect, 500 total)
```

Figure d. Test screenshot for Baseline grayscale

## Part 2: Traditional features

1. Eigenfood. Apply Principal Component Analysis (PCA) to the training set of grayscale feature vectors extracted above. What do the top few Eigenvectors look like, when plotted as images? How quickly do the Eigenvalues decrease? Using the top k eigenvectors, represent each image as a k-dimensional feature vector by projecting the image into this lower-dimensional space. Then use an SVM similar to the one above.

Solution: We start by storing all our images into a matrix of dimension  $1600 \times 1250$ . 1600 since each image is reduced to  $40 \times 40$  pixel image and 1250 is for total number of images.

- Next we will compute the mean matrix which contains the mean value for each row of the above generated matrix.
- After the mean is calculated, we subtract the original matrix with the mean matrix and take the transpose of the result.
- The resultant matrix will then be multiplied with the transpose of itself, giving us the covariance matrix.
- Now we use the symmetric-eigen function on the covariance matrix which gives us a matrix of eigenvectors.
- We then select top 'k' columns from the above matrix, resulting in a reduce matrix of dimensions  $1250 \times k$ .
- The vector is written to a file(train.dat) which is compatible with the input format of SVM. SVM classifier is run on both train and test images and the accuracy is recorded.
- Entire code for this implementation is stored in "Eigen.h" and we call the train and test functions using "a3.cpp".

```
Reading model...done.
Reading test examples... (237 examples) done.
Classifying test examples...done
Runtime (without IO) in cpu-seconds: 0.01
Average loss on test set: 0.0000
Zero/one-error on test set: 0.00% (237 correct, 0 incorrect, 237 total)
Correct: 0 Incorrect: 250 Total: 250
Precision of the model = 0
```

Figure a. Test screenshot for PCA

2. Haar-like features. Similar to Viola and Jones, define a set of many of sums and differences of rectangular regions at different positions and sizes in different configurations. Use Integral Images to compute these efficiently. Instead of Adaboost or the cascaded classifier used in Viola-Jones, simply compute each feature for each image, put them in a feature vector, and use an SVM to do the classification.  
Solution: First step in generating Haar-like features is to define thousands of rectangular regions at different positions and sizes.

- We have defined a 2D vector of size  $1000 \times 4$  filled with random generated values containing x, y, width and height, such that the symmetry is maintained.
- We then resize each of the image as  $40 \times 40$  pixels image. Now for each of the x and y points from the above vector, we compute the sum of all pixels(using w and h values) in this region and we call this as the white space sum. Similarly we will form a rectangle of the same size just below the white region and compute sum of all pixels to form a gray space sum.
- We will compute the absolute difference of both the summations(white space - gray space) and pass it to a vector. The difference is computed for each image in a similar way and stored in the vector. Size of this vector is "number of images \* 1000".
- This vector is then passed to SVM classifier with the input compatible format. We train the model first and compute the accuracy for test images.
- Entire code for this implementation is stored in "haar.h" and we call the train and test functions using "a3.cpp".
- NOTE that we did last minute changes to our "haar.h" function but couldn't get time to upload the latest model. We have attached the previous model file and screenshot which gave just 1% result. But we are sure that modified function gives us more accuracy(we have put it to run but unfortunately it is running for long). Hence please re-run "haar.h" with "train" and "test" and check the accuracy.

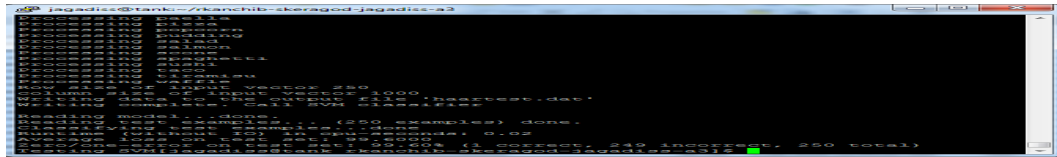


Figure a. Test screenshot for haar-like features

3. Bags-of-words. Run SIFT on the training images, and then cluster the 128-d SIFT vectors into  $k$  visual words. Represent each training image as a histogram over these  $k$  words, with a  $k$ -dimensional vector. Learn an SVM similar to the one above.

Solution: We start by computing SIFT descriptors for each image in the "train" directory and store the results in a vector of vectors.

- Descriptor values in Vector of vectors is copied to a Matrix object such that the Matrix holds the total number of descriptors across all images(rows) and their corresponding SIFT values(columns). We keep a reference of which image consists of which descriptors using a "ref" array.
- We then cluster the SIFT descriptor values using k-means. We have used opencv implementation of kmeans, which takes the above created Matrix as input and computes  $k$  clusters( $k$  set to 1000 in our case).
- We will then represent each training image as a histogram over the  $k$  visual words generated above and the result is stored in "buf" vector. Essentially the vector consists of  $k$  dimensions for each image and value represents the count of SIFT descriptors for each  $k$ .
- The vector is written to a file(train.dat) which is compatible with the input format of SVM. SVM classifier is run on both train and test images and the accuracy is recorded.
- Entire code for this implementation is stored in "bow.h" and we call the train and test functions using "a3.cpp".

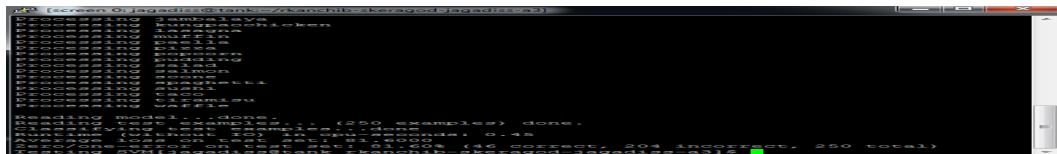


Figure a. Test screenshot for Bag of Words

### Part 3: Deep features

1. Convolutional Neural Networks for image classification. A variation on this approach is to take a pre-trained network, and instead of re-training it on a new problem, simply use the output of one of a deep but not final layer as features for input to another classifier. Your program can then input the file of features, and learn and test an SVM using those feature vectors.

Solution: We used the OverFeat(Convolutional Network-based) feature extractor from <http://cilvr.nyu.edu/doku.php?id=overfeat>. In order to extract the features out of OverFeat, we use the below command-

```
./bin/linux_64/overfeat -f train/bagel/37105.jpg
```

- We first convert the image to grayscale, resize it to 40\*40 pixel image and then pass it to OverFeat. This optimizes the runtime of the program.
- Extracted features of each of the resized image is stored in a intermediate pipe, using which we will read the feature values(non-zero feature values as specified in the SVM classifier description) and write it to a file compatible with the input SVM format.
- Entire code for this implementation is stored in "CNN.h" and we call the train and test functions

using "a3.cpp".

```
Final epsilon on KKT-Conditions: 0.09896
Upper bound on duality gap: 495.49342
Dual objective value: dval=2786.15735
Primal objective value: pval=3281.65077
Total number of constraints in final working set: 388 (of 535)
Number of iterations: 536
Number of calls to 'find_most_violated_constraint': 27500
Number of SV: 379
Norm of weight vector: |w|=74.65719
Value of slack variable (on working set): xi=0.02369
Value of slack variable (global): xi=0.09896
Norm of longest difference vector: ||Psi(x,y)-Psi(x,ybar)||=35.03457
Runtime in cpu-seconds: 178.55
Final number of constraints in cache: 6250
Compacting linear model...done
Writing learned model...done
```

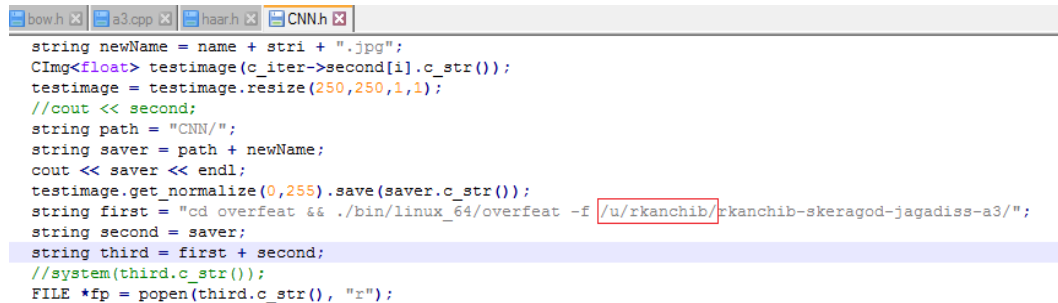
Figure a. Train screenshot for cnn

```
Reading model...done.
Reading test examples... (250 examples) done.
Classifying test examples...done
Runtime (without IO) in cpu-seconds: 0.18
Average loss on test set: 40.0000
Zero/one-error on test set: 40.00% (150 correct, 100 incorrect, 250 total)
```

Figure b. Test screenshot for cnn

Assumption:

1. While running the CNN algorithm, we had to access the images in our path dynamically. Hence we have used the value of our folder name i.e., "rkanchib" as one of the inputs. Screenshot below -



```
string newName = name + stri + ".jpg";
CImg<float> testimage(c_iter->second[i].c_str());
testimage = testimage.resize(250,250,1,1);
//cout << second;
string path = "CNN/";
string saver = path + newName;
cout << saver << endl;
testimage.get_normalize(0,255).save(saver.c_str());
string first = "cd overfeat && ./bin/linux_64/overfeat -f /u/rkanchib/rkanchib-skeragod-jagadiss-a3/";
string second = saver;
string third = first + second;
//system(third.c_str());
FILE *fp = fopen(third.c_str(), "r");
```

Hence you need to modify the highlighted line in "CNN.h" to include your folder name i.e.,

```
string first = "cd overfeat && ./bin/linux_64/overfeat -f /u/"YOUR FOLDER NAME"/rkanchib-skeragod-jagadiss-a3/";
```

You need to make this change for 25 lines in the file "CNN.h"(for 25 classes). We regret for the trouble since we couldn't come up with a better solution in time.

Also as explained in the CNN description above, we are optimizing the process by converting each image to 40\*40 pixel image. Since OverFeat requires image as the parameter, we have created a separate folder called "CNN"(inside our directory) and we store all the resized images in this folder. Hence our code also demands a folder named "CNN" and in order to test CNN, the resized images should be present here.

So please run the "train" command before "test"(If you directly run the "test" against the model we gave, it will give an error since the images are not uploaded to github)

2. We haven't added the OverFeat package in the git directory since the file size was 1.3GB. Please install the package before executing CNN algorithm.
3. We have downloaded SVM classifier in a separate folder named "SVM". All the algorithms assume that SVM is installed in a folder named "SVM" and executes. You need to install SVM classifier in a separate folder named "SVM" inside our github folder in order to run the code.

Experimental comparison:

1. Below table gives us an overview of the accuracies and runtimes of different algorithms. In general, we can see that CNN gives us a better accuracy with considerable runtime as compared to other algorithms.

Table 1: Experimental comparison

	Algorithm	Accuracy	Runtime(approx)
1	Baseline color	17.6%	15 minutes
2	Baseline grayscale	11%	20 minutes
3	PCA	0%	20 minutes
4	Haar	1%	25 minutes
5	BOW	18.4%	30 minutes
6	CNN	40%	20 minutes