CSCI-B 657

Computer Vision

**A3**

Spring 2016

Indiana University Bloomington

|  |  |  |
| --- | --- | --- |
| Rohit Nair | | | ronair@indiana.edu |
| Karthik Sreenivas | | | kartsree@umail.iu.edu |
| Scott McCaulay | | | smccaula@indiana.edu |

March 27, 2016

# Part 0.

We compiled the code on burrow.soic.indiana.edu using the provided make file.

Execution commands:

* ./a3 train algo
* ./a3 test algo

(algo being baseline, eigen, haar, bow, or deep for corresponding parts)

Each classifier has a header file:

* svm.h
* EigenFood.h
* HaarLike.h
* BagofWords.h
* DeepFeatures.h

# Part 1.

In this part our objective is to pass the training and the test dataset into the svm\_multiclass which has been downloaded from the link mentioned in the pdf. Here we have created a new file svm.h which has been linked to the main file a3.cpp.In the train function we read the training data and have fixed the size to 40 so we obtain a vector having 4800 features. We then read the training data into a text file which is to be passed to the svm\_multiclass\_train function. Here we write the training data into a file called **‘dat.txt’**.We modify the data from the training dataset into a format accepted by the svm\_multiclass. The format is label number followed by the feature number, value pair. We call the the svm\_multiclass\_train through a system call in the void train() .Once the training is complete the svm\_multiclass writes a file called svm\_struct\_model. Then for each test data the string classify() is called where we convert the test data into the format accepted by the svm\_multiclass and write it into a file called **‘testing.txt’** . For each test image we call the the svm\_multiclass\_classify through a system call where we pass the testing.txt and the svm\_struct\_model which was initially generated by the svm\_multiclass.Then svm\_multiclass\_classify generates a file called svm\_predictions which provides a label number and we retrieve that number and map it with the class\_list which contains the label numbers of all the data

# Part 2.

Three traditional types of features were implemented, tested and compared to each other and to the baseline results. These were Principal Component Analysis (PCA), Haar-like features similar to the Viola and Jones paper, and a bag of words algorithm based on a k-means clustering of SIFT descriptors. A brief summary of our implementations of these methods follows:

# Part 2.1. Eigenfoods

We apply the concept of EigenFaces to our food products dataset.

Below are the steps involved in the classifier:

* Average vector: We calculate the average vector from all the images in the entire dataset
* Normalization: subtract average vector from each image
* Covariance: normalized class vector matrix \* its transpose (instead of doing transpose \* vector otherwise it’ll be huge 🡪 1200x1200 for image vector size of 1200)
* Eigendecomposition:
  + Using the symmetric\_eigen function of CImg to calculate eigenvectors and eigenvalues from the covariance matrix. Verified eigenvectors \* its transpose giving an identity matrix. Eigenvalues decrease very slowly, the last one being of the order 10-1 to 10-2 on an average, sometimes going down to 10-6
  + Using the SVD function of CImg to calculate eigenvectors and eigenvalues from the covariance matrix. Verified eigenvectors \* its transpose giving an identity matrix(missing some ones though). Eigenvalues decrease very slowly, the last one being of the order 10-1 to 10-2 on an average
* PCA Dimensionality Reduction: selecting k eigenvectors which have the highest eigenvalues from the previous result (we’ve used 10 eigenvectors with the top 10 highest eigenvalues)

Converting eigenvectors to the image vector dimension, so for 10 eigenvectors and each image vector taken to be of 1200, we get a black and white 10x1200 image as shown below:



We store such a model for the various food products that we have and then apply svm on it

# Part 2.2. Haar-like features.

This method is a simplified version of the Viola Jones object detection framework.  Each image is divided up into adjacent rectangles, and the pixel values within each rectangle are summed.  Each feature is the difference in value of several of these rectangles.  The size of the rectangles, the number of rectangles used in the calculation and their relative position can vary.  Many combinations can be used together, and in our work we did experiment with different sizes and placements of these rectangles.

A key to performance in this implementation is the use of integral images as a shortcut for all the calculation involved in summing pixel values.  An array the size of the original image is populated with sums so that each cell contains the sum of all cells above it and to the left.  When the feature extraction needs to sum pixel values in a rectangle, these pre-summed values can be used rather than doing the full addition at each step.

Each calculation of the difference in pixel sums became one value in an array of such values used to classify the text images.  An attempt was made in our implementation to scale the feature values to the range of approximately 0-255, as results were poor with very large positive and negative values.  Results improved with scaling. The same SVM from part 1 is used for this purpose.  Results varied with the type and size of features selected, but this method was able to improve quite a bit over the baseline results.  More specific results are included in the Part 4 summary and comparison.

Part 2.3 Bag-of-words

This method begins with the extraction of SIFT descriptors from each image. The entire collection of SIFT vectors returned from all the test images should be submitted to a k-means clustering, the result of which will be a set of k vectors represent the center of each cluster.  The next step is for each image to assign each of its SIFT vectors to a cluster, based on the smallest calculated distance to any of the cluster centers.  The feature vector representing each image is an array of k values, the occurrence count of that images SIFT descriptors within each of the k clusters.

Again, the same SVM is used to classify the test images, by creating a vector of the cluster counts of their SIFT vectors, and testing that against the training set.  This method is computationally intensive.  Extracting the SIFT descriptors from the training set alone is a long running process, then hundreds of thousands of these vectors must be iteratively assigned to cluster mean vectors.

Part 3: Deep features

The idea behind implementing convolutional neural networks in this way is to take a pre-trained network and use the output of one of a deep but not final layer as features for input to another classifier, we’re using the 12th layer from the OverFeat package to extract features from our images, followed by learning from our DeepFeatures class and classifying with svm.

Steps in Implementation:

* Store folder names of all the food products in an array, which is used later to calculate the category number required to be passed to the svm classifier
* For each image in each food product, run overfeat and store the result in a separate file. This is the part which takes around 20 minutes

In order to make it compatible to svm, we need to convert these output files

* Extract each file, count number of features, width and height
* Append it to a model file in the correct format
  + <Category\_number> list(<feature\_number>:<feature\_value>) # <filename>

We’ve trained our classifier using 12 layers of overfeat and our own algorithm.

Part 4: Comparison and Summary

* Accuracy: (best to worst)

1. Bag of Words: 10%
2. Baseline: 10%
3. Haar: 8.4%
4. EigenFood: 6%

* Speed: (fastest to slowest)

1. Bag of Words
2. Haar
3. Eigen
4. Baseline
5. Deep – overfeat computing 12 layers takes a lot of time (around 20-30 minutes)

# References.

* Eigen faces: <https://www.youtube.com/watch?v=SaEmG4wcFfg>
* Discussed high level ideas with Arpit Khandelwal for Eigen foods and Deep features