B657 Assignment 3: Object detection

Spring 2017
Due: Tuesday April 4, 11:59PM
Late Deadline: Thursday April 6, 11:59PM (with 10% grade penalty)

This assignment will give you experience with several of the object recognition, detection, and localization techniques that we have discussed in class.

We've once again assigned you to a group of other students. You should only submit one copy of the assignment for your team, through GitHub, as described below. After the assignment, we will collect anonymous information from your teammates about everyone's contributions to the project. In general, however, all people on the team will receive the same grade on the assignment. Please read the instructions below carefully; we cannot accept any submissions that do follow the instructions given here. Most importantly: please start early, and ask questions on Piazza so that others can benefit from the answers.

We once again recommend using C/C++ for this assignment, and continue to recommend using CImg and have prepared skeleton code to get you started. For this project, you *may* use the methods of the CImg class, instead of having to write everything from scratch. You may also use additional libraries for routines not related to image processing (e.g. data structures, sorting algorithms, etc.) or to what we expect for you to implement in the assignment below. Please ask if you have questions about this policy.

While you may decide to do your code development on any system you choose, **make sure your code compiles and runs on the SOIC Linux systems** because that is where we will grade them.

Academic integrity. You and your teammates may discuss the assignment with other people at a high level, e.g. discussing general strategies to solve the problem, talking about C/C++ syntax and features, etc. You may also consult printed and/or online references, including books, tutorials, etc., but you must cite these materials in the documentation of your source code. However, the code that you and your teammate submit must be your own work, which you personally designed and wrote. You may not share written code with any other students except your own teammates, nor may you possess code written by another student who is not your teammates, either in whole or in part, regardless of format.

Overview

Perhaps the number one public health crisis in the U.S. today is obesity, with over 70% of American adults obese or overweight (as of 2015). As computer scientists, a natural question is how we can use technology to help address this pressing crisis. Devices like FitBit and Apple Watch have recently had success in encouraging exercise, but the other side of the health equation – encouraging healthy eating – has not seen as much progress and seems to be a much more difficult goal.

Here we imagine a mobile application that would let people take pictures of the food they eat, automatically recognize it, and keep track of their nutritional intake in terms of calories, vitamins, etc. The goal of this assignment is to apply some recognition algorithms to this problem and test how well such an application may work in practice. All of these approaches will share the same machine learning classifier (a Support Vector Machine), so much code can be recycled across the different parts. This will let you test the importance and relative strengths of different computer vision features.

To make the problem more concrete, we've prepared a dataset of training and test images for 25 different food categories. There are a total of about 1,250 training images and 250 test images. In addition, we've held out a separate set of 250 test images which we will use for testing your work, so that you won't be

tempted to train on the test images or try any other unfair tricks. :)

Part 0: Getting started

1. Clone the GitHub repository:

   `git clone https://github.iu.edu/cs-b657/`*your-repo-name*`-a3`

   where *your-repo-name* is the one you found on the GitHub website.

2. Download the training and test data from OnCourse, and then run (inside your repository directory):

   `tar -xzf food-data.tar.gz`

   This should create two directories, called train and test, with the 25 food categories as subdirectories.

3. The skeleton code already implements a very simple nearest neighbor-based classifier, which you can compile and run like this:

   ```
   make
   ./a3 train nn
   ./a3 test nn
   ```

Part 1: A simple baseline

In this part, you'll use raw image pixel data as input to a black box machine learning classifier, a support vector machine (SVM). Your training program should:

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 (where "40" is just an example – you may want to choose something higher or lower).

2. Call an SVM library or program to train an SVM on this task. One good choice is SVM_multiclass, which is fast, easy to use, and freely available: `https://www.cs.cornell.edu/people/tj/svm_light/svm_multiclass.html`. Instead of trying to integrate their code into yours, it's fine to have your program write out a file in the format they expect, and make a `system()` to execute their program on the command line, and then parse the output files it produces.

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? (i.e. if you convert the image into grayscale so that a 40x40 image becomes a 1600-d vector, versus preserving the red, green, and blue channels to produce a 4800-d vector)?

Part 2: Traditional features

Implement the following three types of features we discussed in class:

1. Eigenfood. Apply Principal Component Analysis (PCA) to the training set of grayscale feature vectors extracted above. (Note that CImg includes routines for Eigendecomposition.) What do the top few Eigenvectors look like, when plotted as images? How quickly do the Eigenvalues decrease? Using the top k eigenvectors (where k is a number you'll have to choose), 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.

2. Haar-like features. Similar to Viola and Jones, define a set of many (probably thousands) of sums and differences of rectangular regions at different positions and sizes (e.g. randomly-chosen) in different configurations (see Figure 1 of the Viola and Jones paper). 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 vectopr, and use an SVM to do the classification.

3. Bags-of-words. Run SIFT on the training images (we've include the Sift code again in your repo), 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-dimenional vector. Learn an SVM similar to the one above. (You can either use an existing implementation of k-means, with proper citations, of course, although it is not hard to implement it yourself.)

In your report, include an experimental comparison (quantitative and qualitative) of the two approaches you choose with each other and the baseline from Part 1.

Part 3: Deep features

In class we discussed Convolutional Neural Networks for image classification, including the idea of taking a pre-trained network and "fine-tuning" it on a new problem. 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. For this problem, instead of re-implementing a CNN, you can instead use the OverFeat package to extract features from your images (`http://cilvr.nyu.edu/doku.php?id=software:overfeat:start`). (You don't need to integrate it in your source code; simply call the OverFeat binary to dump features to a file using `system()`.) Your program can then input the file of features, and learn and test an SVM using those feature vectors. Again, compare results to the approaches in Parts 1 and 2.

Specifications

Your programs should assume that there is a subdirectory called train/, that contains training images, and another directory called test/, that contains test images. Your program should be run with the following command line:

```
./a3 mode algo
```

where `mode` is either `train` or `test`, and `algo` is one of `baseline`, `eigen`, `haar`, `bow`, or `deep`. Your training programs will likely generate data files (containing SVM model parameters) that your test program will need to load in. The skeleton code already implements this for you, so just follow its lead!

What to turn in

Make sure to prepare (1) your source code, and (2) a report that explains how your code works, including any problems you faced, and any assumptions, simplifications, and design decisions you made, and answers the questions posed above. To submit, simply put the finished version (of the code and the report) on GitHub (remember to `add`, `commit`, `push`) — we'll grade the version that's there at 11:59PM on the due date.