CMSC 25400 Homework 4

Ruolin Zheng

February 9, 2019

The submission zip file include:
boosters.txt - a nested list of boosters from 5 trials of cascade
viola_jones.py - the pre-processing and training functions
postprocessing.py - the post-processing image labelling functions
driver.py - driver code
result.png - the prediction result on *test_img.jpg*
hw4.pdf

1 Project Description

Given 2000 faces and 2000 backgrounds each as a 64 * 64 image file, we construct Haar features as our pool of base learners. Then by running the AdaBoost Algorithm, we combine several weak learners into a strong learner which keeps a low False Positive Rate. From one stage of the Cascade to the next, we remove correctly recognized non-face images and feed the rest into the next stage of the Cascade.
All 4000 data are included in training. I used in total **21913 features** as base learners - 8902 two-horizontal-square ones, 8902 two-vertical-square ones and 4109 three-square ones. The step size for both the two-square and three-square ones is 4. As for shape, the two-square ones start at 4 * 8 and increments by 4 * 8; the three-square ones start at 6 * 9 and increments by 6 * 9. **This submission attempts extra credit with the implementation of the three-square feature.**
My implementation includes in total **5 classifiers**, corresponding to the five-stage Cascade. The classifiers consist of 8, 14, 14, 18, 15 weak learners, respectively. The False Positive Rate of the classifiers are 0.242, 0.379, 0.333, 0.457, 0.486, respectively. Training the five layers takes approximately 936.267, 1983.738, 1947.953, 2145.323, 1220.112 seconds (around 3 hours in total). The overall FPR of the Cascade is 0.00678. Please refer to the Comment section for plot and table summary.

2 Pre-processing

This section is a description of the main utility functions used in pre-processing. For helper functions, please also refer to function docstrings included in the code file.

load_data(face_dir, background_dir)

This function opens the 4000 images located in the given directories with PIL Image.open(), and converts them to numpy arrays with np.asarray().

compute_integral_image(imgs)

This function computes the integral image representation of a N * 64 * 64 array of image arrays. Its helper function integral_image(arr) computes the iir for a single image array with the Dynamic Programming approach.

feature_list(N)

This function generates Haar features given a dimension restriction N, which is 64 in this assignment. It calls on helper functions to generate two-vertical-square, two-horizontal-square and three-square features and return them as a numpy array. The two-square features are represented as a tuple of two coordinate tuples *((dark_top_left, dark_bottom_right), (light_top_left, light_bottom_right))*; the three-sqaure ones are represented as a tuple of three coordinate tuples *((dark_top_left, dark_bottom_right), (left_light_top_left, left_light_bottom_right), (right_light_top_left, right_light_bottom_right))*.

preprocess_data()

This function returns *int_img_rep, feat_lst, labels*, loading cached data if they exist and generate anew otherwise. In the latter case it makes calls to int_img_rep(imgs) and feature_list(64), and concatenate numpy arrays of 2000 1's and 2000 -1's into the labels array.

3 Training

This section is a description of the main utility functions used in training. For helper functions, please also refer to function docstrings included in the code file.

compute_feature(int_img_rep, feature)

This function computes a given feature (identified by coordinate and shape) across all images based on int_img_rep. If the feature is a two-square feature, that is, of length 4, the function calls helper compute_two_rect_feature(int_img_rep, feature), or, if the feature is a three-square one with length 6, compute_three_rect_feature(int_img_rep, feature). Both helpers unpack the top-left and bottom-light coordinates of the dark and light rectangles and call pixel_intensity(int_img_rep, top_left, bottom_right) to compute the pixel intensity over the area. The return value, a N * 1 array, is thus $\frac{dark\_pixel\_intensity}{dark\_area} - \frac{light\_pixel\_intensity}{light\_area}$ in the two-square case and $\frac{dark\_pixel\_intensity}{dark\_area} - \sum \frac{light\_pixel\_intensity}{light\_area}$ otherwise.

opt_theta_p(computed_features, labels, weights)

This function finds the optimal θ and p given the evaluated features as a N * 1 array (calculated by compute_feature_() above), labels and weights. It sorts the computed_features, labels and weights based on an increasing ordering of computed_features. It then scans from left to right, keeping track of the faces and backgrounds it has encountered, comparing them

to the faces and backgrounds to be encountered. It finds the computed feature value which minimizes the error and return the corresponding θ and p (θ is set as by averaing feature values at j and $j+1$-th index where j has the minimum error.)

eval_learner(computed_features, theta, p)

This function computes the label predictions given the evaluated feature values (calculated by compute_feature_() above), θ and p. It implements the formula $sgn(p(f-\theta))$, labelling values ≥ 0 as +1 and -1 otherwise. The return value is an N * 1 array of label predictions.

error_rate(labels, hypotheses, weights)

This function calculates the error rate of a given learner / feature given true labels, hypotheses (calculated by eval_learner() above). It adds up the weights where there is a mismatch between a label and a hypothesis by taking the dot product of weights and abs((labels - hypotheses) / 2).

opt_weaklearner(int_img_rep, feat_lst, labels, weights)

This function finds the optimal weak learner among all features by calling the helper functions above. It iterates over the entire feat_lst and for each feature, passes the value of compute_feature() to opt_theta_p() to obtain the optimal θ, p. The particular weak learner is then identified by $(feat\_idx, \theta, p)$. It then calculates and records the weak learner's error with error_rate(). After completing this operation on all features, it returns the weak learner associated with the smallest error rate.

update_weights(weights, learner_weight, labels, hypotheses)

This function returns an updated N * 1 weights array, where learner_weight is computed in advance with math.log((1 - error) / error) / 2.

eval_booster(int_img_rep, weighted_learners, threshold)

This function computes the raw hypotheses values given an array of weak learners and a threshold value, Θ.

eval_cascade(int_img_rep, boosters)

This function computes the raw hypotheses values given an array of boosters from various stages of the cascade.

adaboost(int_img_rep, feat_lst, labels, weights, max_iters=18)

This function initializes an empty array of weighted learners and repeatedly adds the optimal weak learner and its weight to the array. It then updates the weights, finds the minimum value among all images with true label +1 (face) and set Θ as this value such that no face is missed, keeping False Negative Rate to 0. It then calculates the False Positive Rate of the current combined learner. The function returns the combined learner and the updated weights of all datapoints once FPR is lowered to 0.3 or when *max_iters* is reached. (Default value is set to 18 in consideration of run-time, please see the section on Design Choices for details.)

cascade(int_img_rep, feat_lst, labels, max_boosters=7)

This function initializes the weights as a uniform distribution, keeps an array of boosters and repeatedly call adaboost() to fill in the boosters. Once adaboost() returns a new booster, this function calculates and updates the cascade's FPR. The function returns the boosters once

FPR is lower than 0.01 or when *max_iters* is reached. Otherwise, it filters out the correctly labelled non-face images, feeding into the next round the remaining datapoints identified by int_img_rep, labels and updated weights.

4 Post-processing

non_maximum_suppresion(imgarr, boosters)
This function implements the non-maximum suppression technique, evaluating the 64 * 64 areas in each 128 * 128 section, with step size 64. It retains only the maximum score in each 128 * 128 section.
label_image(fname, boosters, outfname=None)
This function calls non_maximum_suppression and labels the area where the maximum score is greater or equal to 0 (predicted to be +1.)
label_image_naive(fname, boosters, int_test_img_rep=None, outfname=None)
This function implements a naive 64 * 64 sliding window with step size 32 over the image. The boosters / cascade filters out coordinates at each stage and the function labels the final set of coordinates that remains.

Suprisingly, the naive labelling technique produces a better figure than the NMS case. Running the detector on *test_img.jpg*, the original 1911 test points that remains after each stage of the cascade are 140, 119, 115, 76, and finally, 70, as shown in the picture below.

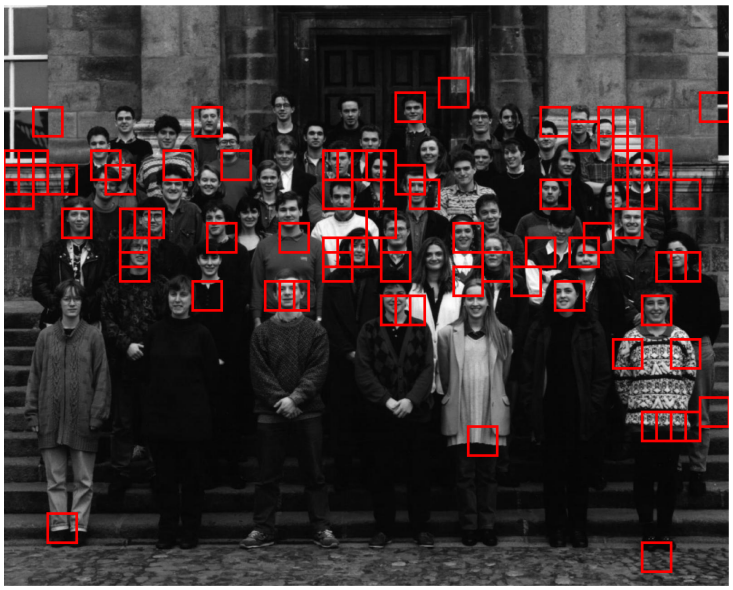

Figure 1: Result of Running Detector on Test Image

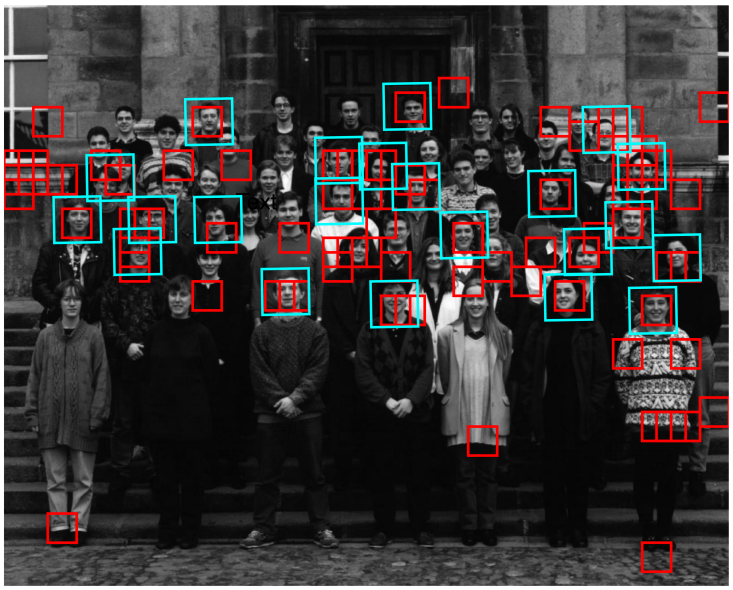

Figure 2: Eyeballing Correct Detection by Detector

As we can see in the picture above, the detector predicted around 70 blocks as human faces (plotted with pyplot in red), 22 out of which (manually marked in light blue) turn out to be actual human faces. The result is not very satisfying as we observe that some faces are missed and some false positives (especially the right-most person's sweater) persist. The detector's errors are actually self-explanatory given how it is trained. Most missed faces are darker than the correctly detected ones and are thus less distinguishable from the background with respect to difference in pixel intensity. Moreover, most false positives are areas of relatively high contrast in pixel intensity. There are also some partially boxed faces which will have a higher chance of being in the correct position if I replace the naive windowing with the full NMS technique.

4.1 Output

boosters.txt contains a nested list of 5 classifiers and their corresponding Θ. The classifiers have respectively 8, 14, 14, 18, 15 weak learners. Each weak learner consists of a feature (two-square or three-square) in the tuple coordinate form, threshold θ, polarity p and weight. Interestingly, there are only 2 three-square features among all weak learners. This may be explained by the fact that I only included 4109 such features in all 21913, and that the increment size of 6 * 9 may cause valuable ones to be missed.

5 Comment

5.1 Design Choices

I chose to implement the three-square features since Professor Kondor mentioned that they are more discriminating.
Run-time plays an important role in my design choices, I shrunk the size of my feature list by starting at slightly larger features (4 * 8 and 6 * 9) and taking strides of 4. Regarding code efficiency, I managed to vectorize compute_feature() to operate on the entire int_img_rep array at once but looping through the entire feature list in opt_weaklearner() is still a performance bottleneck. To prevent indefinite hanging, I set *max_iters* in adaboost() to be 18 and *max_boosters* in cascade() to be 7. This way, the function will terminate eventually even when the FPR fails or takes too long to converge. I do admit this approach may be problematic in the sense that I could be prematurely ending the loop before FPR converges. This sub-optimal FPR may account for the false positives in my test image detection.
Aside from this, to improve code efficiency, I pass around variables to avoid repeated calculations as in computed_features in many of my helper functions for opt_weaklearner(). I also made sure to modularize my code while keeping in mind that function call overhead is not too light for Python.

5.2 Running Record and Summary

Below is a plot of the False Positive Rate when training the classifiers in trials of AdaBoost. As shown in the plot, the first classifer converges with an FPR lower than 0.3. However, subsequent trials did not converge as well and ended with default iteration counts in some cases.

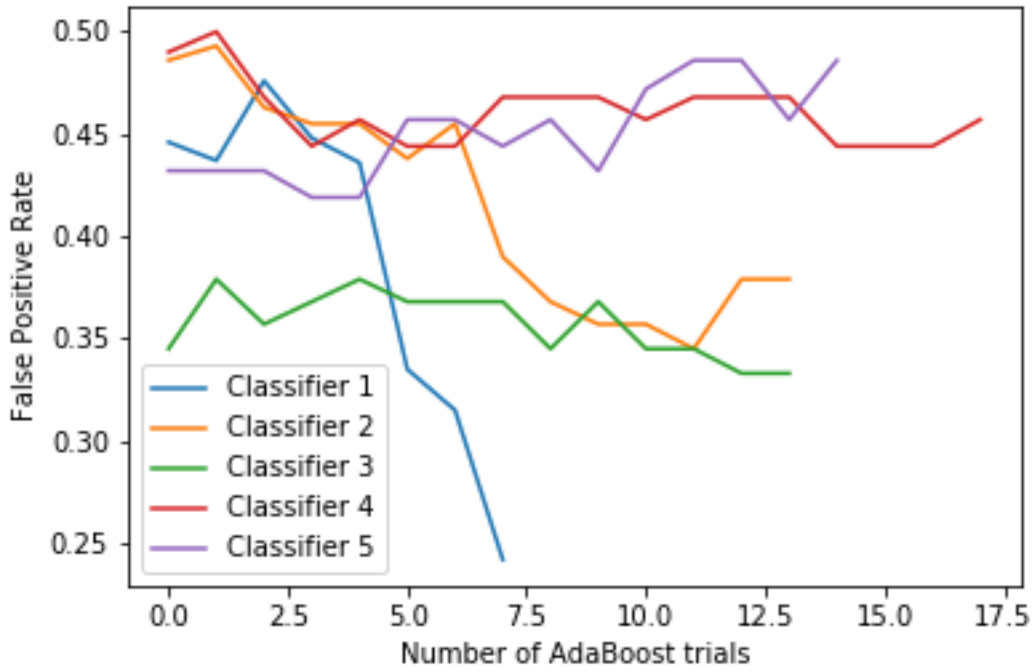

Figure 3: False Positive Rate of 5 Classifiers During Training

Below is a table summary of the statistics of the five layers of the Cascade.

Layer No.	Classifer FPR	Cascade FPR	Training Time (sec)
1	0.242	0.242	936.267
2	0.379	0.0917	1983.738
3	0.333	0.0305	1947.953
4	0.457	0.0140	2145.323
5	0.486	0.0068	1220.112

Table 1: Cascade Statistics

5.3 Run-time, Efficiency and Other Optimization

Running on 4000 data, around 20000 features and 5 layers of Cascade takes around 3 hours. We can greatly improve the run-time by vectorizing the code. **compute_feature()** is one computational bottleneck as we are computing the same feature across all 4000 images. **opt_weaklearner()** is another bottleneck since we call **compute_feature()** on every single one of the 20000 features. In addition, storing feature coordinates in a list seems not very space-efficient. (Luckily we compute the features on the fly.)

As an alternative implementation (per Steven's Piazza Post,) instead of specifying feature coordinates, we can compute a feature of given shape at any location in the image with numpy array slicing and manipulations. This eliminates the use of for loops on the features and will allow more features to be added to the base learner pool.

5.4 Running the Code

Please run *driver.py* in Python 3 and note that library dependencies include *skimage*, *PIL* and so on.