CMSC 25400 Face Detection

Can "Stella" Liu

1 Data Preprocessing

1. Function <read-images> converts an image to greyscale using PIL and computes the pixel values at each point of the 64×64 picture. The matrices storing pixel values are then saved as csv files for later usage.

2. Function <find-iimages> takes in the matrix of pixel values and compute the integral image of the picture. Here instead of writing recursing call myself, I used slicing and np.sum to compute the integral image. The integral images are also stored as csv files.

3. Function <find-features> find all possible features (given step = 6) within the 64×64 square. The coordinates of the four vertices of a rectangle are stored as a list in the form of $[x_0, y_0, x_1, y_1, x_2, y_2]$. Each rectangle corresponds to two possible "two-rectangles" features - one vertical and one horizontal. The orientation of the feature is stored as "v" or "h". Function <select-features> and <features-subset> can later extract a subset of these features to accommodate to different training and testing needs.

4. Function <compute-feature-table> compute the values for selected features across all samples and turns the results into a dictionary. The key of the dictionary is a unique string indicating the 8 coordinates as well as orientation of a feature. The value corresponding to this key is a list of feature values across all samples.

2 Algorithm

I was following the Viola-Jones algorithm described in the slides and assigned paper. Basically the algorithm investigates "two-rectangles" features across 64×64 photo and compute the pixel value difference between the "black" block and "white" block. While these features are only weak learners, applying Adaboost and cascade can group some relatively significant hypotheses together to boost a stronger learner.

1. Fucntion <compute-feature> computes the feature value at each spots using the method suggested on the slides. I stored some of thees values in feature-table dictionary for future uses. When testing on the photo, I simply computed the feature values on the fly.

2. Function <best-learner> finds the appropriate threshold and polarity for a given feature. After computing the feature values for each sample, the function sorts the results

from low to high, each corresponding to an "is-face" value indicating whether the sample is a face or not. The threshold and polarity are then found through for loop to minimize the error rates. I am mostly using numpy instructions in this function to speed up the program (although it doesn't seem very successful)

3. Function <adaboost-one> performs a single step of Adaboost. That is, it finds the feature with the lowest error when the distribution of data is given. The function also computes α, which is the weight of this learner in the strong learner.

4. Function <adaboost> performs Adaboost. After finding the strongest learner at each stage, it changes the distribution of the data based on

$$D_{t+1}(i) = \frac{D_i(x)exp(-\alpha_t h_t(x_i))y_i}{Z_t},$$

where $Z_t = 2(\epsilon_t(1-\epsilon_t))^t$. The adaboost function terminates either when a number of iterations has been finished, or the false positive rate has dropped below 0.3. In the beginning I used a lot of for loops and list comprehensions so the program was really slow. Now it takes about 5 - 20 minutes to finish one stage of boosting, subject to the number of features and samples.

5. Function <adjust-threshold> modify the threshold (big theta) of a computed adaboost learner so that the false negative rate is minimized (that is to say, faces should all be recognized). This is done by sorting the prediction value of each sample under a given learner. I then picked the smallest threshold that corresponds to a "face" photo and used the prediction value immediately smaller than it as the new threshold. Usually the threshold gets smaller than zero, which means we are more confident towards categorizing a photo as "non-face".

6. Function <cascade-one> takes out the samples that already fails the first cascade learner. It also normalizes the weights.

7. Function *cascade* finishes cascade. It has two types of terminate conditions. Either it reaches a given number of iterations, or the false positive rate falls below 0.01.

3 Parameter Tuning

1. Samples and feature sizes: I started with small features sizes and sample sizes (1000 features and 400 samples). Later on I tried the algorithm on larger scale (8000 features and 3800 samples). I left 200 samples aside for testing purpose.

2. Both adaboost and cascade can be terminated manually by adjusting maximum iteration number, or automatically through computing false positive rate. I started with setting the maximum iteration for adaboost and cascade as 3 and 5, relatively. Then

learner generated behaved fine on the training data set (I used all 4000 photos as testing data as well), but when it comes to photo, the learner gives too many positive results. Later on I modified the condition so that the code would stop adaboost once the false positive rate drops below 0.3, and stop cascade when the same rate is less than 0.01. (The weird thing is that the false positive rate usually drops below 0.3 after only 1 or 2 cycles of adaboost, so I might have done something wrong here...)

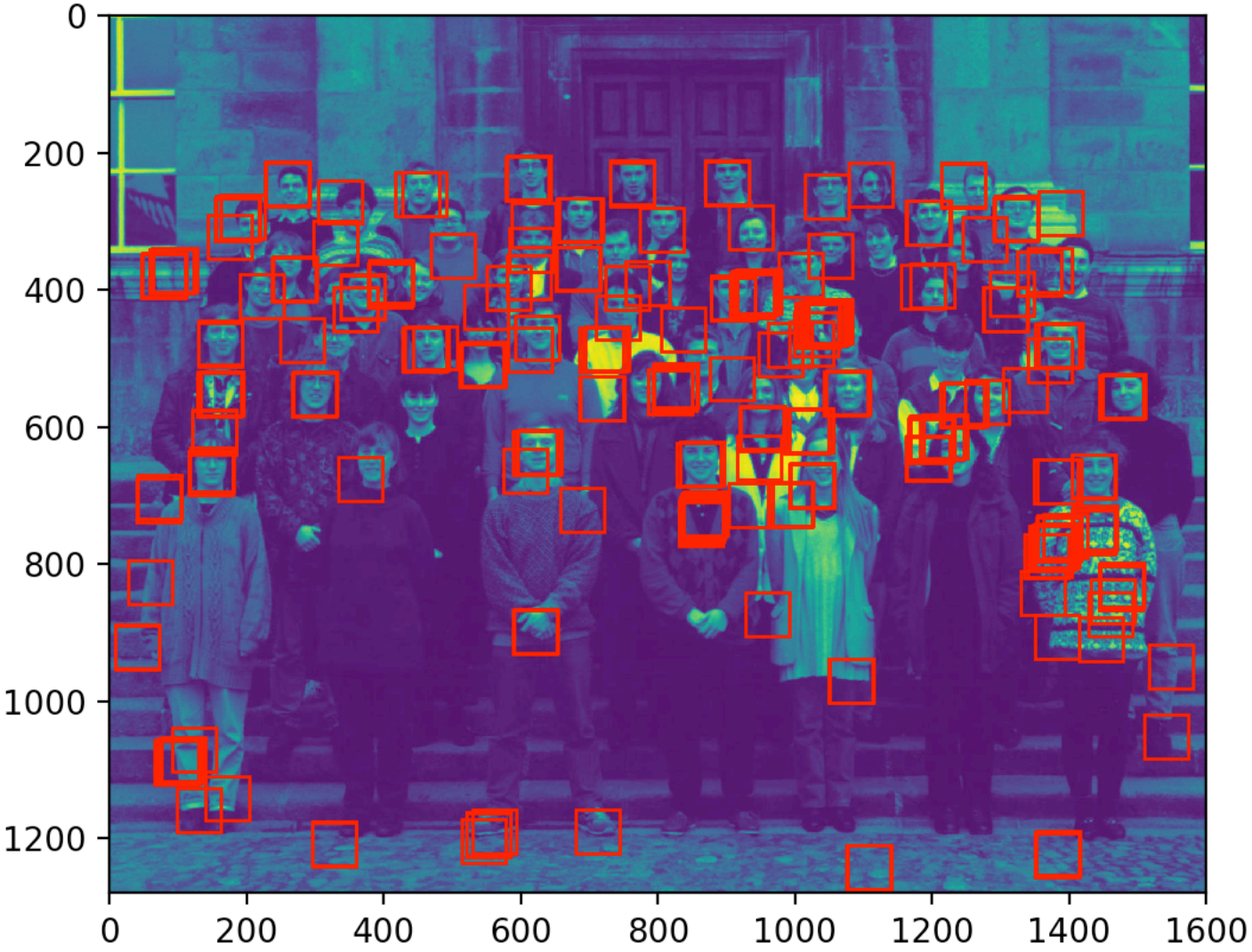
4 Testing and Results

I used both the training data and the photo to test the learners. For the training data sets, the learner generated through 3 iterations of adaboost from 2000 features and 400 samples already behaves well. When tested on faces photo, it can reach a correctness over 0.96, and on background, a correctness around 0.92. The learner using the automatic terminate condition behaves great on background data (false positive rate = 0.003), but is not good enough when it is tested on the faces photos (for some reason, it misses a lot of faces)

On the photo, the smaller training sets tend to detect more faces but generate much more false positives at the same time as well. To find the faces, I used a 64×64 sliding window across every single point on the photo. Since there are lots of overlaps, I then exclude any "faces" that lie within 32 pixels of some previous detected faces.

The error rate for the first learner in adaboost is around 0.07. Later on it gets larger to around 0.16.

The following two pictures indicate the faces detected, respectively, by learner trained with 1000 features using 400 samples, and learner trained with 8000 features using 3800 samples.


0
200
400
600
800
1000
1200
0
200
400
600
800
1000
1200
1400
1600


0
0
0
0
0
0