Tufts

Class #09:
Ensemble Learning and Boosting

Machine Learning (COMP 135): M. Allen, 19 Feb. 20

1

Ensemble Learning Methods

- An ensemble learning method combines multiple learned functions into a single prediction
- A simple example is the decision forest: build a set of different decision trees using different parts of our data
  - Instead of only keeping one of them (e.g., the one with least error on its test set), we keep them all
  - For any new classification, run it through all of the trees
  - Use the majority classification, breaking ties randomly

2

Boosting Methods

- A more complex method combines weaker, error-prone classifiers in a sequence, getting better as it goes
- Each time we classify the training set, correct/incorrect classifications are used to ***weight the data*** so that next classifier can improve results
- An important version of this is ADABOOST
  - Stands for "Adaptive Boosting"
  - Freund & Schapire, 1999 (Gödel Prize, 2003)
  - Has a very interesting and important convergence property: if each classifier is even ***slightly better*** than random chance, we can eventually boost to a ***perfect classifier*** (in the limit)

3

ADABOOST

Source: Russel & Norvig, *AI: A Modern Approach* (Prentice Hal, 2010)

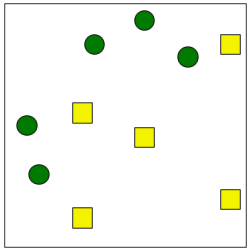
function ADABOOST($examples, L, K$) **returns** a weighted-majority hypothesis
  inputs: $examples$, set of N labeled examples $(x_1, y_1), \ldots, (x_N, y_N)$
    L, a learning algorithm
    K, the number of hypotheses in the ensemble
  local variables: $\mathbf{w}$, a vector of N example weights, initially $1/N$
    $\mathbf{h}$, a vector of K hypotheses
    $\mathbf{z}$, a vector of K hypothesis weights

  for $k = 1$ **to** K **do**
    $\mathbf{h}[k] \leftarrow L(examples, \mathbf{w})$
    $error \leftarrow 0$
    for $j = 1$ **to** N **do**
      if $\mathbf{h}[k](x_j) \neq y_j$ **then** $error \leftarrow error + \mathbf{w}[j]$
    for $j = 1$ **to** N **do**
      if $\mathbf{h}[k](x_j) = y_j$ **then** $\mathbf{w}[j] \leftarrow \mathbf{w}[j] \cdot error/(1 - error)$
    $\mathbf{w} \leftarrow$ NORMALIZE($\mathbf{w}$)
    $\mathbf{z}[k] \leftarrow \log(1 - error)/error$
  return WEIGHTED-MAJORITY($\mathbf{h}, \mathbf{z}$)

- With K runs of some learning algorithm L, we adjust weights on ***both*** training-data ***and*** algorithm classifications themselves until we get to the end
- The final output is based on the weights given to the different classification results

4

An Example of Using ADABOOST

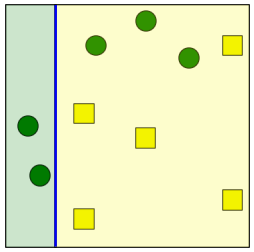


- 10 data-points, in 2 classes
- Run algorithm with $K = 3$ separate, consecutive boosting steps
- Initially, each example in the data-set is given ***equal weight*** = $1/10 = 0.1$

function ADABOOST($examples, L, K$) **returns** a weighted-majority hypothesis
inputs: $examples$, set of N labeled examples $(x_1, y_1), \ldots, (x_N, y_N)$
L, a learning algorithm
K, the number of hypotheses in the ensemble
local variables: $\mathbf{w}$, a vector of N example weights, initially $1/N$
$\mathbf{h}$, a vector of K hypotheses
$\mathbf{z}$, a vector of K hypothesis weights

5

An Example of Using ADABOOST



- At each of the K runs, we classify using an algorithm L that ***uses the weights*** given
  - In this example, we use a linear classifier of some sort, but other examples are possible
  - For example, we could create a new version of the ***decision tree*** algorithm
  - When choosing sets to split, calculate the entropy values (Lecture 03) using the ***weighted sums*** of positive and negative examples, instead of simply counting them

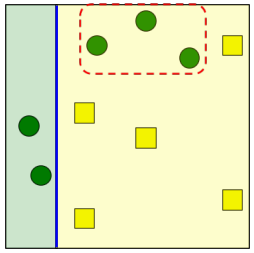
for $k = 1$ **to** K **do**
$\mathbf{h}[k] \leftarrow L(examples, \mathbf{w})$

Initially, since all weights are the ***same***, the algorithm would be identical to the one seen previously.

As we change weights over time, however, getting things right/wrong for heavy-weighted items would ***matter more*** to how the algorithm classified.

6

ADABOOST: Round 1



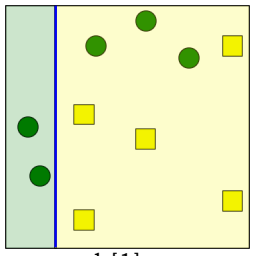
- We sum up the error for any elements that have been mis-classified by the hypothesis

$$error \leftarrow 3 \times 0.1 = 0.3$$

$error \leftarrow 0$
for $j = 1$ **to** N **do**
if $\mathbf{h}[k](x_j) \neq y_j$ **then** $error \leftarrow error + \mathbf{w}[j]$

7

ADABOOST: Round 1



- We sum up the error for any elements that have been mis-classified by the hypothesis

$$error \leftarrow 3 \times 0.1 = 0.3$$

- We adjust weights downwards on the 7 ***correctly classified*** items:

$$w[incorrect] = 0.1$$
$$w[correct] = 0.1 \times 0.3/0.7 \approx 0.043$$

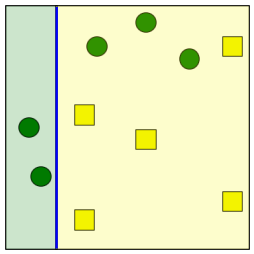
for $j = 1$ **to** N :
if $\mathbf{h}[k](x_j) = y_j$: $\mathbf{w}[j] \leftarrow \mathbf{w}[j] \cdot \dfrac{error}{(1 - error)}$

Correct items are ***less important***, to stress the importance of eliminating classification error.

If we have a ***lot*** of error, then that is far more important to get rid of than getting some correct answers.

8

AdaBoost: Round 1



h[1]

- We normalize all weights:

$$\sum_{w[j]} = (3 \times w[incorrect]) + (7 \times w[correct]) \approx 0.6$$

$$w[incorrect] = 0.1/0.6 \approx 0.167$$

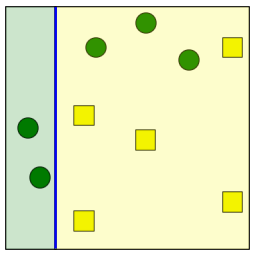
$$w[correct] = 0.043/0.6 \approx 0.0717$$

Note: weight is still highest on the ***incorrect*** items

$\mathbf{w} \leftarrow \text{NORMALIZE}(\mathbf{w})$

9

AdaBoost: Round 1



h[1]

- Last, we calculate the weight for the classification hypothesis h[1] itself:

$$\mathbf{z}[1] \leftarrow \log \frac{0.7}{0.3} \approx 0.847$$

- This weight will be used ***after*** we are done with all the boosting rounds

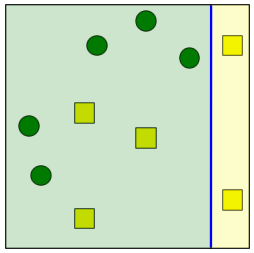
Hypothesis functions with lots of error will have lower weight than those that are more correct.

Using the log ensures that differences between weights aren't too extreme.

$\mathbf{z}[k] \leftarrow \log\,(1 - error)/error$

10

AdaBoost: Round 2



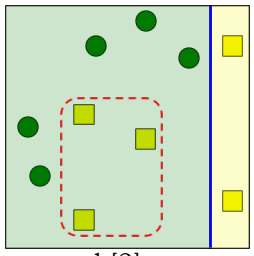
h[2]

- Now we move to next boosting round
- We feed in same data, with new weights ($w[correct]$ or $w[incorrect]$)
- Our algorithm generates a new classification hypothesis, h[2], using the same algorithm L, but results may be different due to the fact that the weights on the data-set have been changed
- This new, adjusted classification is not perfect either

for $k = 1$ **to** K **do**
  $\mathbf{h}[k] \leftarrow L(examples, \mathbf{w})$

11

AdaBoost: Round 2



h[2]

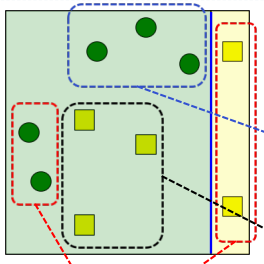
- We now sum up the error for the three mis-classified elements
- ***Note***: we have made the ***same*** number of errors as before, but error value is ***lower*** due to reduced weights on these items
- The incorrect ones were all ***correct*** on previous round, so error now is:

$$error \leftarrow 3 \times w[correct] \approx 0.215$$

$error \leftarrow 0$
for $j = 1$ **to** N **do**
  if $\mathbf{h}[k](x_j) \neq y_j$ **then** $error \leftarrow error + \mathbf{w}[j]$

12

AdaBoost: Round 2



- Now, when we adjust weights down on the correctly classified items, they will get a variety of ***different weights***

Incorrect on 1st, correct on 2nd: 0.046

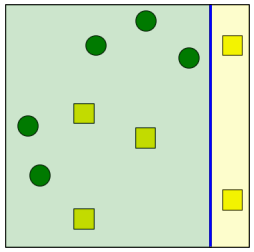
Correct on 1st, incorrect on 2nd: 0.0717 (no change)

Correct on both: 0.02

for $j = 1$ **to** N **do**
  if $\mathbf{h}[k](x_j) = y_j$ **then** $\mathbf{w}[j] \leftarrow \mathbf{w}[j] \cdot \ error/(1 - error)$

13

AdaBoost: Round 2



h[2]

- We normalize these various weights on items, and calculate the weight for the second boosting round:

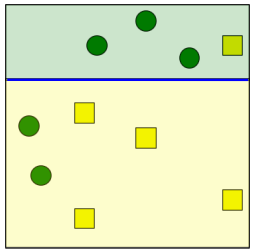
$$\mathbf{z}[2] \leftarrow \log \frac{0.785}{0.215} \approx 1.3$$

Note: this time, the weight is ***higher*** than $\mathbf{z}[1]$, since this classifier is in some ways doing better (because it is based on better information)

$\mathbf{w} \leftarrow$ NORMALIZE($\mathbf{w}$)
$\mathbf{z}[k] \leftarrow \log\ (1 - \ error)/error$

14

AdaBoost: Round 3



h[3]

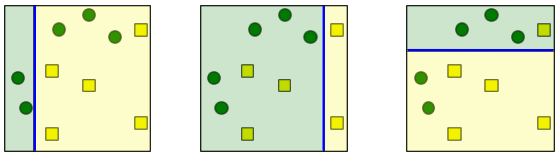
- We do our last boosting round, using weights from prior iteration
- This gives last set of calculated error and hypothesis-weight values:

$$error \leftarrow 0.14$$

$$\mathbf{z}[3] \leftarrow \log \frac{0.86}{0.14} \approx 1.82$$

15

AdaBoost: Final Output



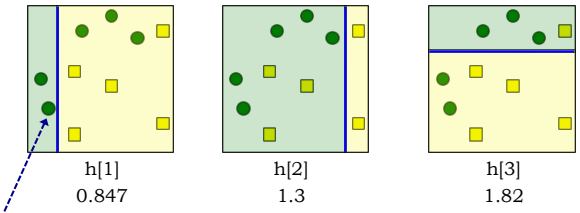
h[1] 0.847 h[2] 1.3 h[3] 1.82

return WEIGHTED-MAJORITY($\mathbf{h}, \mathbf{z}$)

- Now, although ***none*** of our classifiers were perfect, we can combine them in a weighted way to get the final version
- For each object x, and each classification type T, add up weight values for each boosting run that assigned x to T

16

AdaBoost: Final Output

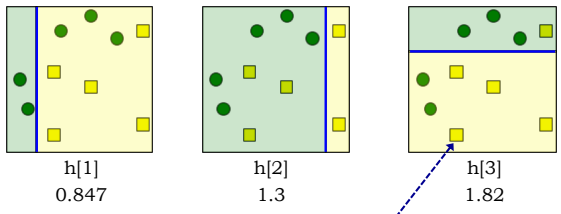



```
w[Green]  = 0.847 + 1.3 = 2.147
w[Yellow] = 1.82
Class = Green
```

- For each object x, and each classification type T, add up weight values for each boosting run that assigned x to T

17

AdaBoost: Final Output

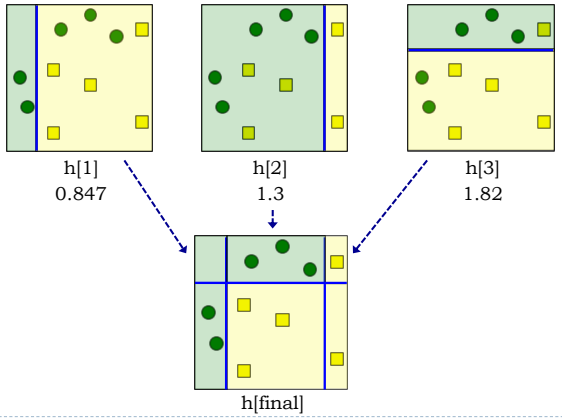



```
w[Green]  = 1.3
w[Yellow] = 0.847 + 1.82 = 2.667
Class = Yellow
```

- For each object x, and each classification type T, add up weight values for each boosting run that assigned x to T

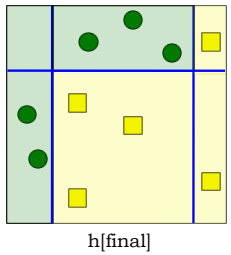
18

AdaBoost: Final Output



19

AdaBoost: Final Output



- The final, weighted-majority classification is ***perfect*** even though ***none*** of the individual components were on their own
- In this example, $K = 3$ rounds were sufficient, but this won't always be the case (although perfection ***always possible***)

20

Ensembles of Ensembles

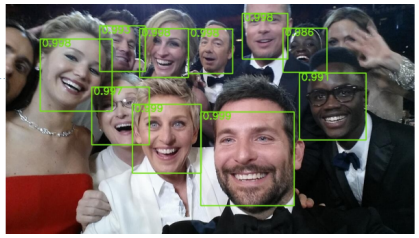
- Once we have used ADABOOST to create a single classifier, we can continue the learning process
- A cascading classifier could, for instance:

1. Build a single strong classifier C_1 using ADABOOST and K_1 boosts of one type of algorithm L_1
2. Create another classifier C_2 using ADABOOST and K_2 boosts of some other algorithm L_2
3. Combine C_1 and C_2 into their own weighted ensemble

- One widely used such method is the Viola-Jones cascade classifier for ***face detection***

Face Detection

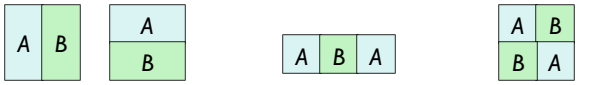
Paul Viola & Michael J. Jones, Robust real-time face detection, *Intl. Journal of Computer Vision*: **57**, 137–154 (2004).



- Data consists of different rectangular regions of a photograph
- Classifiers label each region as "face" or "not face"
  1. If region ***is labeled*** "face" by one level of the classifier, it is passed on to the next, more complex, classifier level
  2. If labeled "not face," the region is ***discarded*** from data-set
- At the end, multiple "face" regions that are very close together are merged into single rectangles

Haar-Like Rectangle Features

- The algorithm evaluates a (24 × 24) region of an image by examining all possible rectangle features
- Each of these divides the region into various sub-regions:

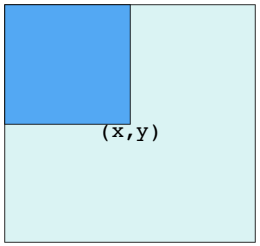


2-rectangle features 3-rectangle feature 4-rectangle feature

- Each ***sums*** the pixel values in each sub-region, and returns the ***difference*** between the ***A*** and ***B*** sub-regions
  - Each rectangle feature-value is calculated for every possible size and starting position for the feature
  - Over 160,000 such features exist for each (24 × 24) region

Efficient Rectangle Features

- To avoid re-calculating pixel-sums over and over, the algorithm first uses dynamic programming to compute an integral image
- An `(m × n)` pixel image is turned into an `(m × n)` integer array, where locatiion `[x][y]` stores sum for entire sub-image from upper-left corner, at `[0][0]`, to that pixel point



Efficient Rectangle Features

- Integral image can be calculated in a single nested `(m × n)` loop:

```
int[][] ii =
    new int[image.length][image[0].length];

for ( int r = 0; r < ii.length; r++ )
{
    int rowSum = 0;
    for ( int c = 0; c < ii[r].length; c++ )
    {
        rowSum += ii[r][c];
        ii[r][c] = rowSum;
        if ( r > 0 )
        ii[r][c] += ii[r - 1][c];
    }
}
```

10	5	0	0
10	5	0	0
10	10	10	10
10	0	0	0

Original image pixels

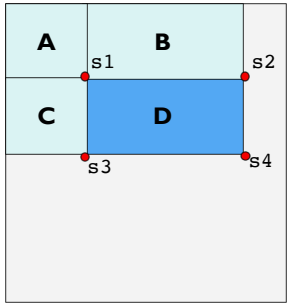
10	15	15	15
20	30	30	30
30	50	60	70
40	60	70	80

Integral image (ii)

25

Efficient Rectangle Features

- We can get the pixel sum value for ***any region we choose***, using the values stored in the integral image array



- To get sum for **D**, look at the other rectangles that lie between it and the origin point `[0][0]`
- Get sum value using only ***four accesses*** to the integral image array
  1. Sum for **A**
  2. Sum for **A** + **B**
  3. Sum for **A** + **C**
  4. Sum for **A** + **B** + **C** + **D**
- We then compute sum for **D**:

  D = `s4 + s1 – (s2 + s3)`

26

Classification Using Rectangle Features

- Set of training examples: image sub-regions $\langle r_1, \ldots, r_N \rangle$, each correctly labeled as containing a face (+) or not (–)
- For rectangle feature F, compute difference of pixel sums, and then ***sort*** all the regions by these difference values
- Loop over regions and compute ***error*** of doing classification using its feature-value as our classification threshold:

$T^+ = \sum_i w(r_i)$, weights of all face regions

$T^- = \sum_j w(r_j)$, weights of all non-face regions

$S_m^+ = \sum_k w(r_k)$, weights of all face regions *before* region r_m in sort

$S_m^- = \sum_l w(r_l)$, weights of all non-face regions *before* region r_m in sort

$$e_m = \min(S_m^+ + (T^- - S_m^-),\ S_m^- + (T^+ - S_m^+))$$

27

Classification Using Rectangle Features

- For region r_m with feature-value v_m, we compute the error that we would get if we make v_m the classification threshold:

$$e_m = \min(\underbrace{S_m^+ + (T^- - S_m^-)}_{\text{Error of labeling every region } \textbf{\textit{after}} \ r_m \text{ (including } r_m \text{ itself) in sorted order as being a face (+), and everything before it as being a non-face (-)}},\ \underbrace{S_m^- + (T^+ - S_m^+)}_{\text{Error of labeling every region } \textbf{\textit{before}} \ r_m \text{ in sorted order as a face (+), and everything after it (including } r_m \text{ itself) as a non-face (-)}})$$

- The algorithm then uses the error value that is smallest overall, and divides on that value for classification using the particular rectangle feature that gave us the sorted ordering

28

Choosing Features by Boosting

- Now we can use ADABOOST to pick set of features:

1. The i-th boosting step chooses the rectangle feature F_i with best error-rate after sorting and thresholding
2. Weights on data and on feature F_i are adjusted according to its error-rate
3. Final output is weighted-majority classifier using the vector of features selected $<F_1, F_2, \ldots, F_K>$

- Viola & Jones trained a single large classifier, using the best 200 rectangle features, and achieved quite good results, but it was quite slow to run classification

Cascading Feature Choices

- Viola & Jones use a different, more efficient cascading approach:

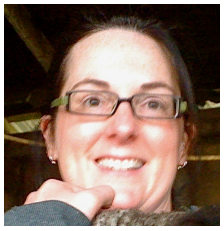
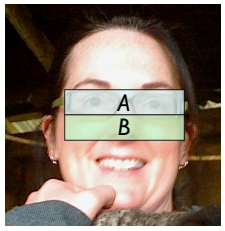
1. The i-th cascading step starts with two limits:
   - Pos_i : ***maximum*** acceptable rate of ***false-positive*** detections
   - Det_i : ***minimum*** acceptable rate of ***correct-positive*** detections
2. Boost as long as false positive rate is not greater than Pos_i:
   a) Choose feature F_i with best classification threshold (as before)
   b) ***Loosen*** threshold value so detection rate ≥ Det_i
   c) Adjust weights and repeat boosting
3. Adjust training-set so that it contains all the actual face regions, along with all those non-face regions that are false positives for current classifier
4. Set new limits Pos_{i+1} and Det_{i+1}; Repeat cascading process until overall detection performance is satisfactory

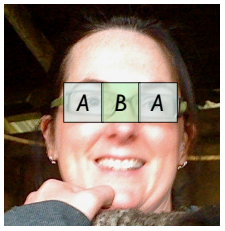
Building the Detector

- Each cascading step starts with two limits:
  - Pos_i : maximum acceptable false-positive rate
  - Det_i : minimum acceptable correct-positive rate
- **Step 1**: Starts with a simple, 2-feature classifier that rejects 50% of non-face regions, while still detecting almost 100% of faces
- **Step 2**: Any regions ***not rejected*** by first classifier are passed to one with 10 features, which rejects about 80% of the non-face regions remaining, while still detecting almost all faces
- ***And so forth…***

Features Chosen

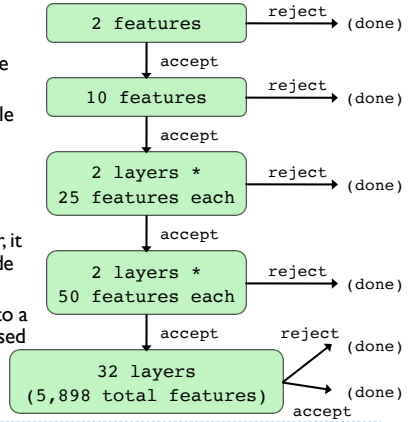
- For the first stage of the classifier, the algorithm chose some basic features of regions that seemed intuitive
- A 2-rectangle feature focusing on contrast between darker eyes and lighter colors typical on upper cheeks (***left***), and a 3-rectangle feature capturing eye contrast with bridge of the nose (***right***)



The Final Classifier

- The final result of the Viola-Jones approach was a 38 layer cascade classifier, with 6,060 total rectangle features used
- On the hardware they had available (a single-core 466-MHz machine), the process took ***weeks***
- They noted that this could be parallelized to speed training
- Once classifier is trained, however, it runs very fast (orders of magnitude faster than others known)
- This concept has been extended to a large number of different Haar-based (and other) cascading classifiers



33

This Week

- **Special schedule**: Class Wednesday & Thursday
- **Topics**: Boosting Classifiers, Feature Engineering
- **Assignments**:
  - Homework 03: due Wednesday, 26 Feb., 9:00 AM
    - Logistic regression & decision trees
  - Project 01: due Monday, 09 March, 5:00 PM
    - Feature engineering and classification for image data
  - Midterm Exam: Wednesday, 11 March
- **Office Hours**: 237 Halligan
  - Tuesday, 9:00 AM – 1:00 PM
  - Thursday, 10:30 AM – Noon
  - TA hours can be found on class website

34