# Introduction to Computer Vision

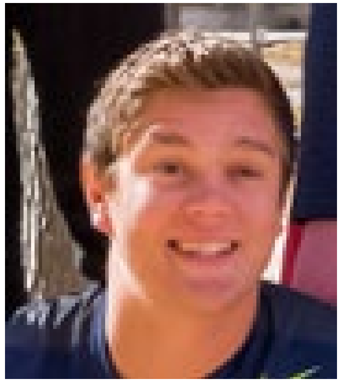
**Kaveh Fathian**
Assistant Professor
Computer Science Department
Colorado School of Mines

**Lecture 17**

# Face Detection in Images


- Slide windows of different sizes across image
- At each location, match window to face model

# Face Detection Framework



Extract Features

$\begin{bmatrix} f \end{bmatrix}$

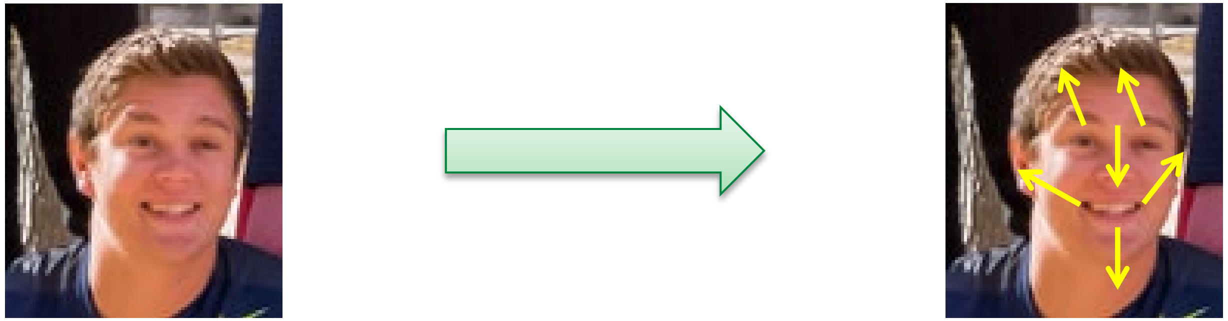
Match Face Model

YES / NO

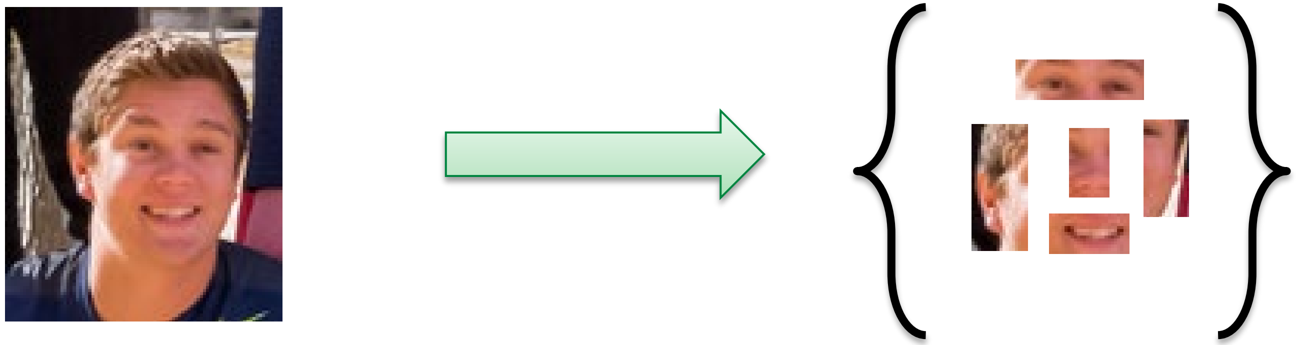
- **Features**: which features represent faces well?
- **Classifier:** How to construct a face model & efficiently classify features as face or not?

# What Are Good Features

- **Interest points:** edges, corners, SIFT, HOG?



- **Facial components:** templates?



# Characteristics of Good Features

- **Discriminate face & non-face**

≠

- **Extremely fast to compute**
  Need to evaluate millions of windows in an image

# Haar Features


Input Image

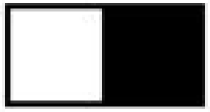
$$\otimes \begin{bmatrix} H_A \\ H_B \\ H_C \\ H_D \\ \vdots \end{bmatrix} = \begin{bmatrix} V_A[i,j] \\ V_B[i,j] \\ V_C[i,j] \\ V_D[i,j] \\ \vdots \end{bmatrix}$$

Haar Filters

Harr Features
$f[i,j]$

# Computing a Haar Feature


$\otimes$



$H_A$

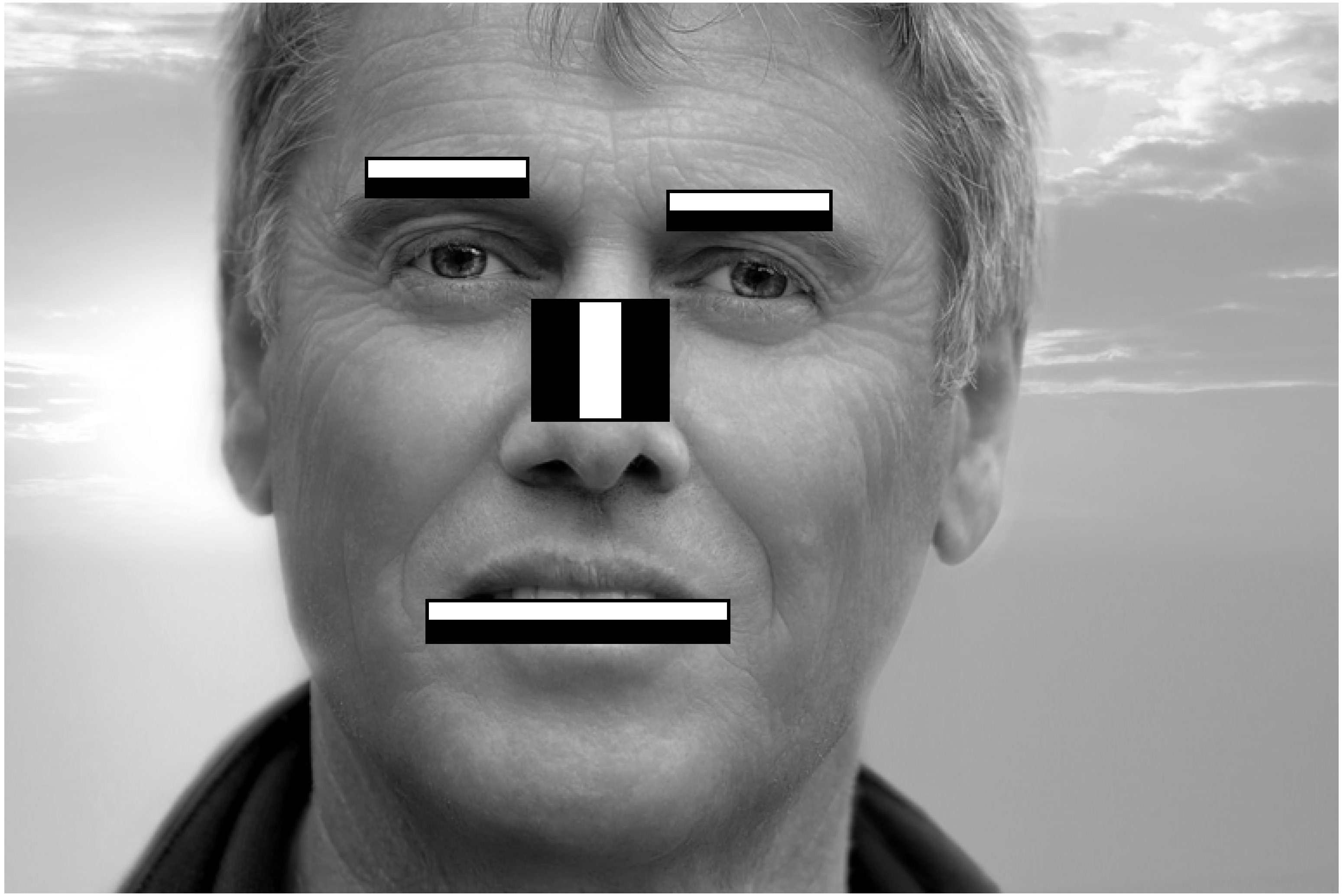
White = 1
Black = -1

Response to filter $H_A$ at location $(i, j)$:

$$V_A[i,j] = \sum_m \sum_n I[m-i, n-j]\, H_A[m,n]$$

$$V_A[i,j] = \sum(\text{pixel intensities in white area}) - \sum(\text{pixel intensities in black area})$$

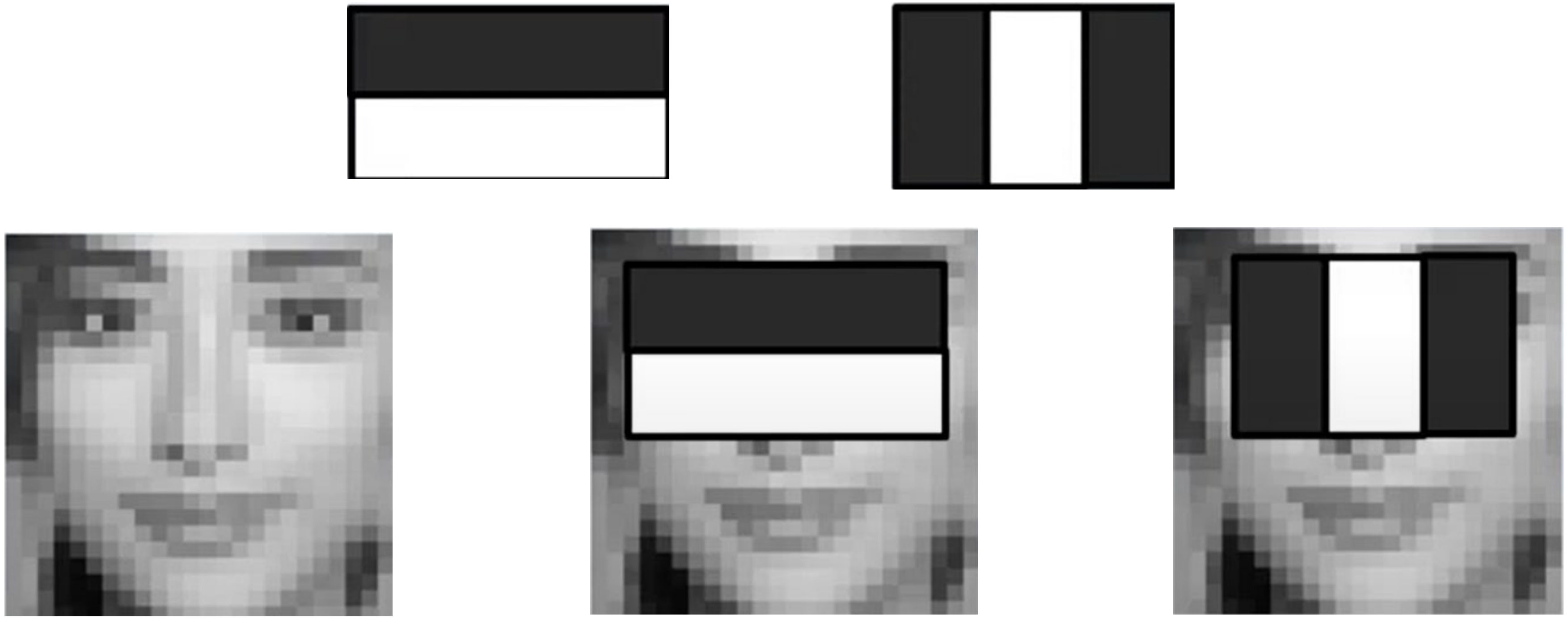
# Discriminative Ability of Haar Features



- Haar features are sensitive to the *directionality* of patterns

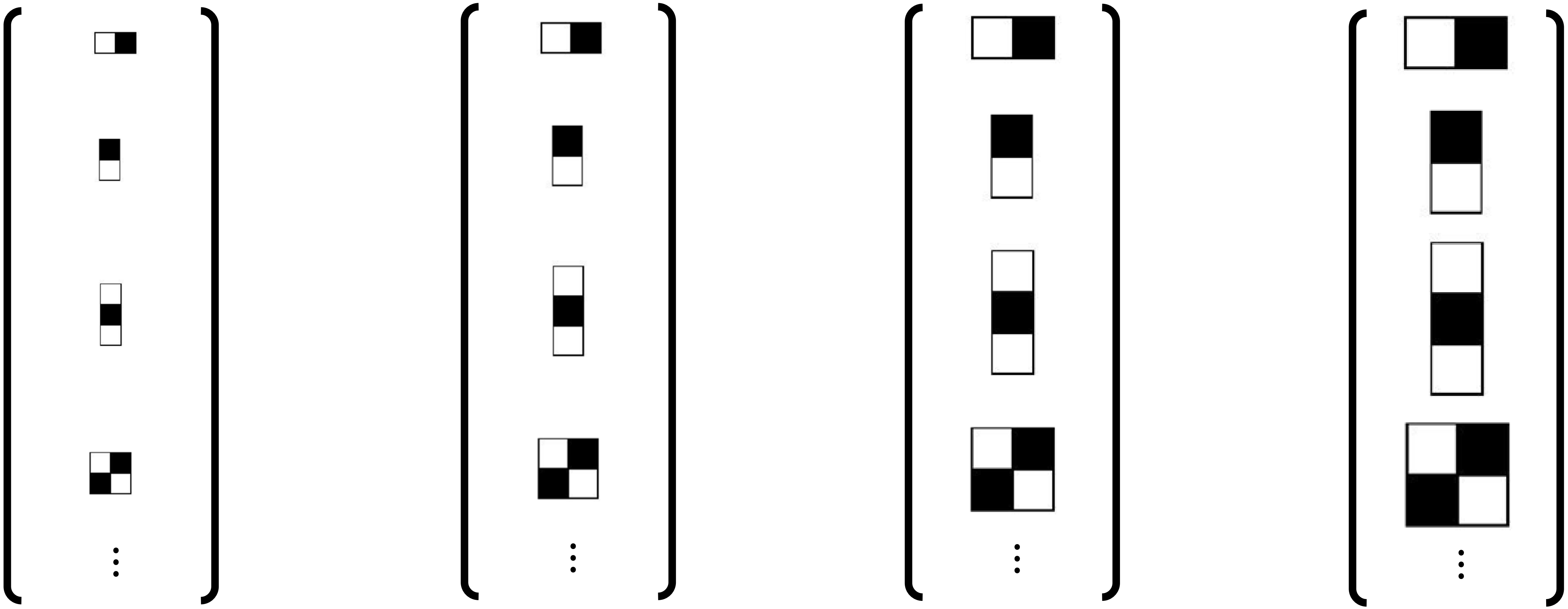
# Discriminative Ability of Haar Features

- Using only 2 Haar features on a 24x24 pixel window:



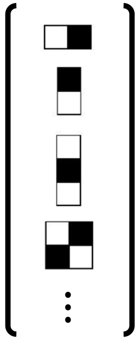
- Can yield 100% face detection rate, and 50% false positive rate

# Detecting Faces at Different Scales



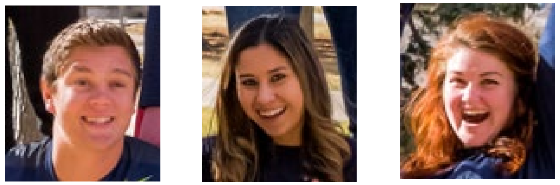
- Compute Haar features at different scales to detect faces of different sizes

# Classifier for Face Detection

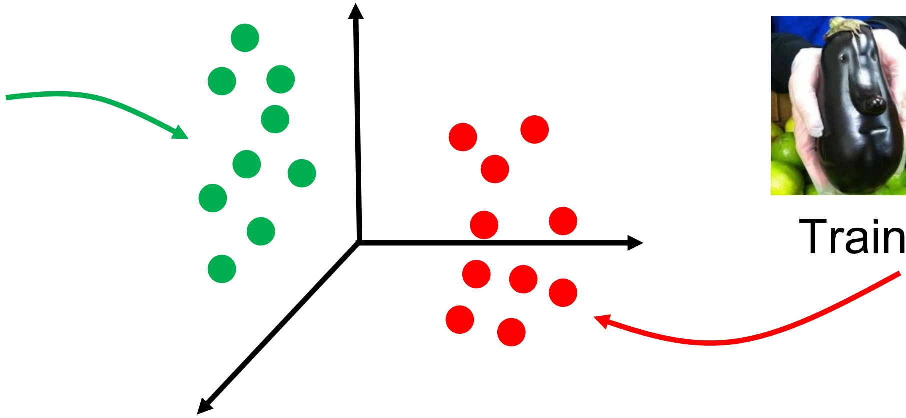


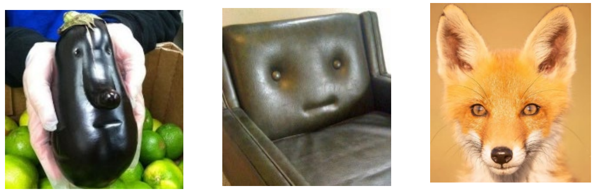
$$f = \begin{bmatrix} V_A[i,j] \\ V_B[i,j] \\ V_C[i,j] \\ V_D[i,j] \\ \vdots \end{bmatrix}$$

- Given features for a window, how to decide whether it contains a face or not?



Training data for face

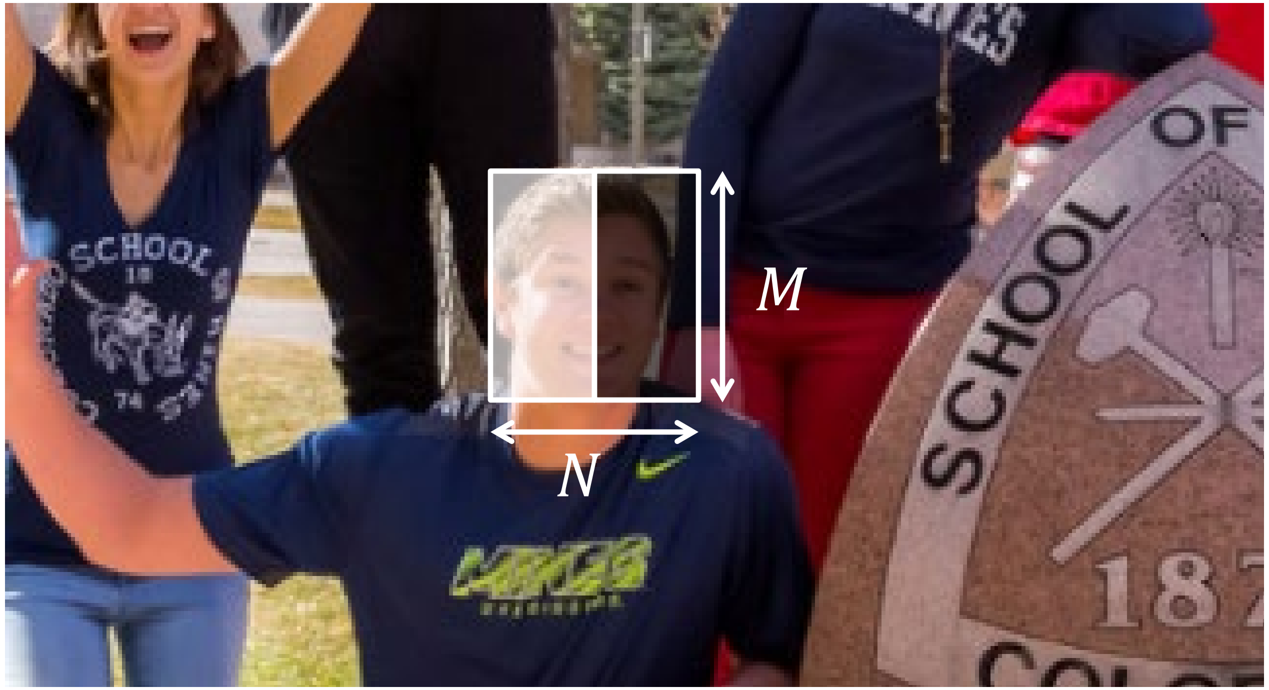




Training data for non-face

- Haar feature $f$ (a vector) at a pixel is a point in n-D space, $f \in \mathbb{R}^n$
- Can use any classifier engine: nearest neighbor, SVM, neural network, etc.

# Haar Feature: Computation Cost




$$\text{Value} = \sum(\text{pixel intensities in white area}) - \sum(\text{pixel intensities in black area})$$

Computation cost = $(N \times M - 1)$ additions per pixel, per filter, per scale

Can we do better?

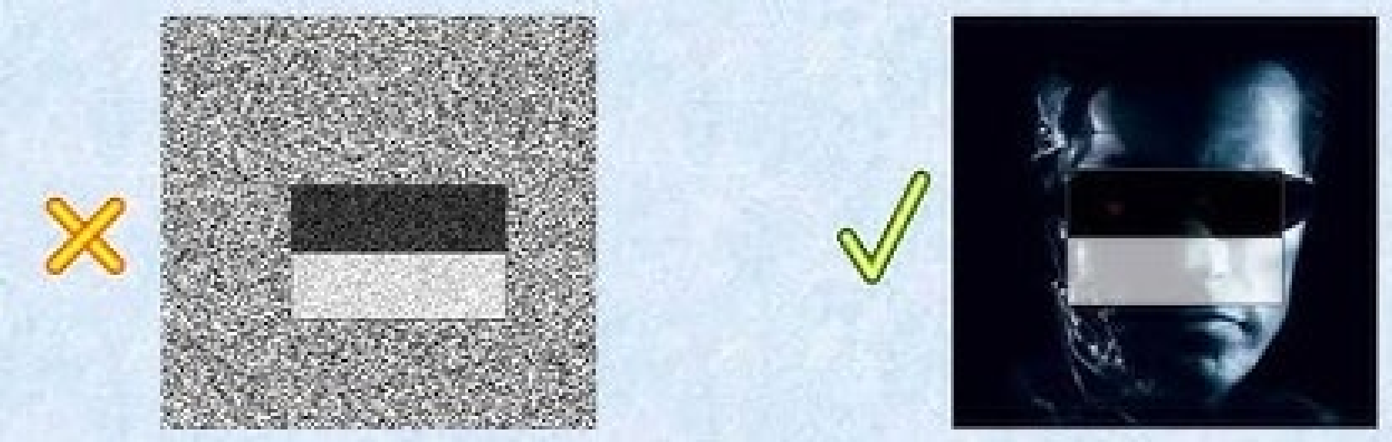
# Requirements for a Face Detector


- Sliding window = tens of thousands of location/scale evaluations
  - One megapixel image has ~$10^6$ pixels, and a comparable number of candidate face locations

- Faces are rare: 0-10 per image
  - For computational efficiency, spend as little time as possible on the non-face windows.

- For 1 Mpix image, to avoid having a false positive in every image, our false positive rate must be less than $10^{-6}$

# Viola-Jones Detector



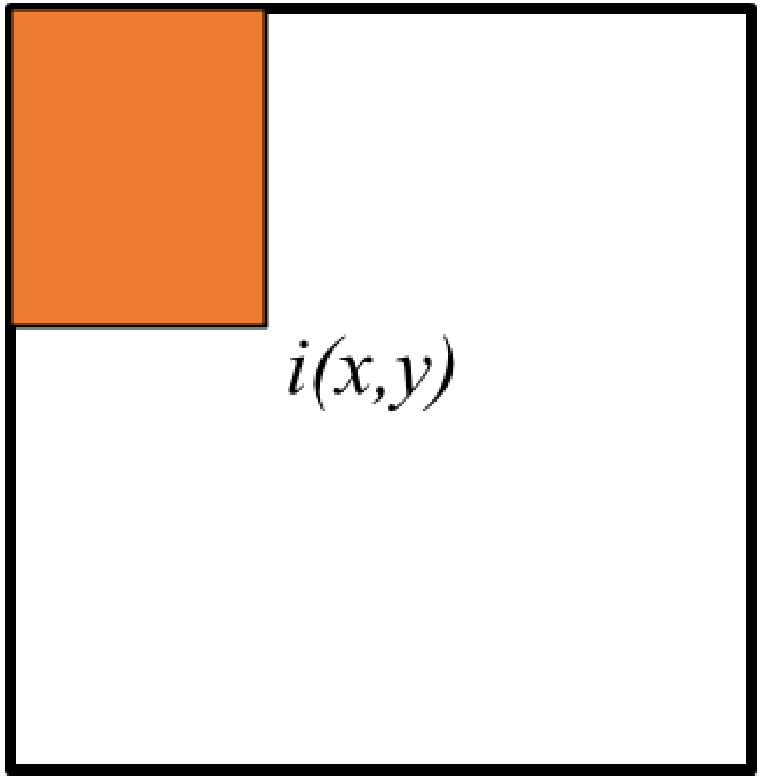
- A seminal approach to real-time object detection
  - P.Viola and M. Jones. “Rapid object detection using a boosted cascade of simple features.” CVPR 2001.  ~30K citations!
  - P.Viola and M. Jones. “Robust real-time face detection.” IJCV 57(2), 2004.
- Training is slow, but detection is very fast

**Key Ideas:**

- **Haar features**
- **Integral images** for fast feature computation
- **Boosting** for feature selection
- Attentional **cascade** for fast non-face window rejection

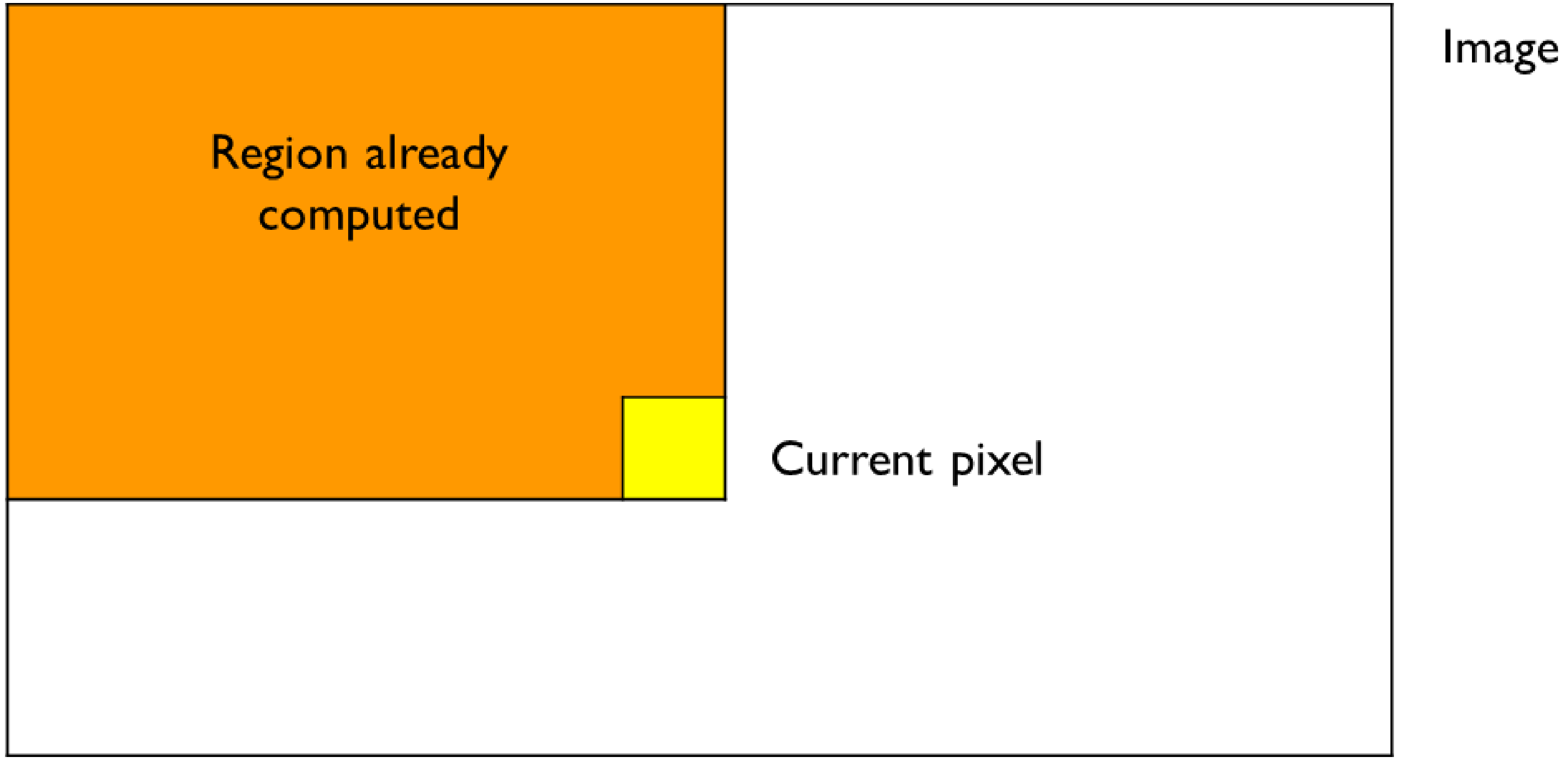
# Integral Images for Fast Feature Evaluation

- The *integral image* computes a value at each pixel ($x$,$y$) that is the sum of *all* pixel values above and to the left of ($x$,$y$), inclusive.
- This can quickly be computed in one pass through the image.
- 'Summed area table'

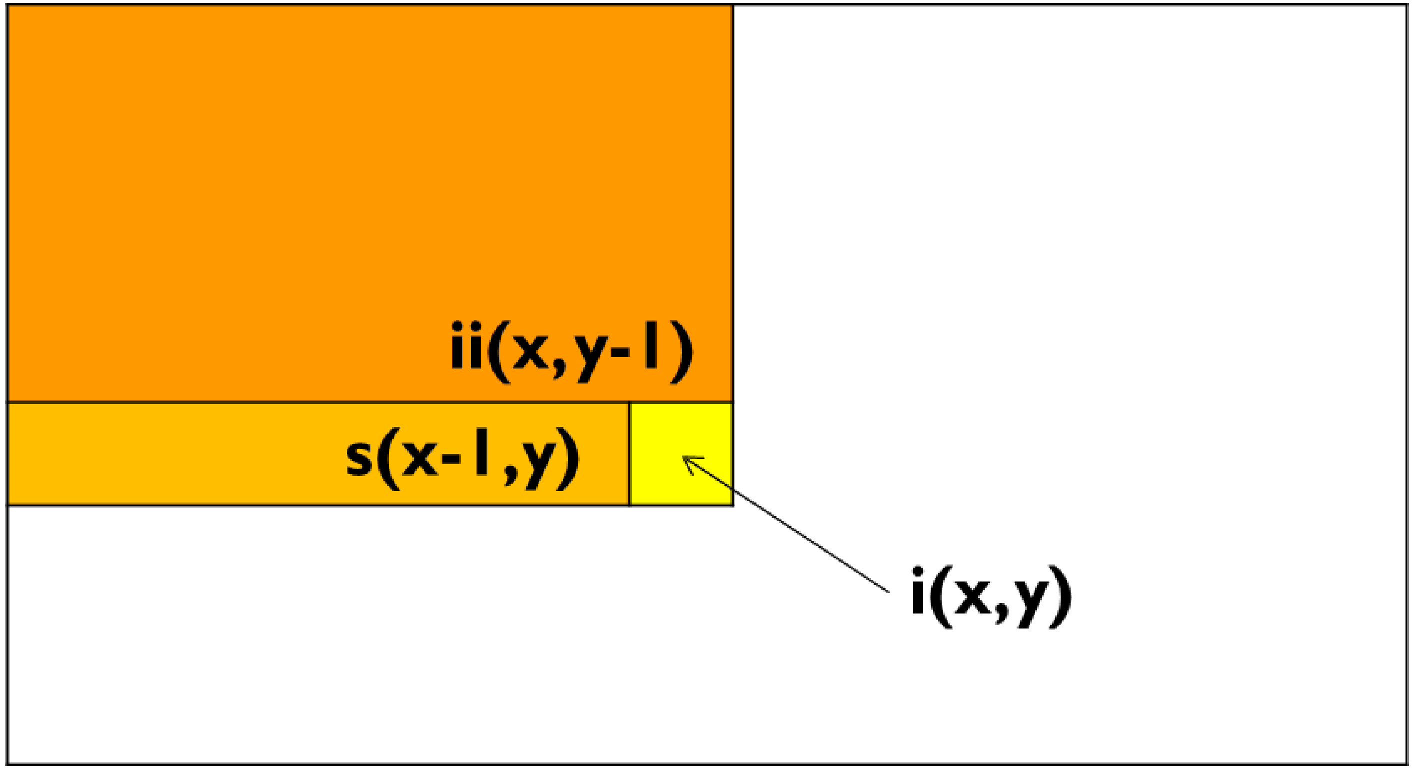



$$I_{\sum}(x,y) = \sum_{\substack{x' \leq x \\ y' \leq y}} i(x', y')$$

# Computing the Integral Image



# Computing the Integral Image



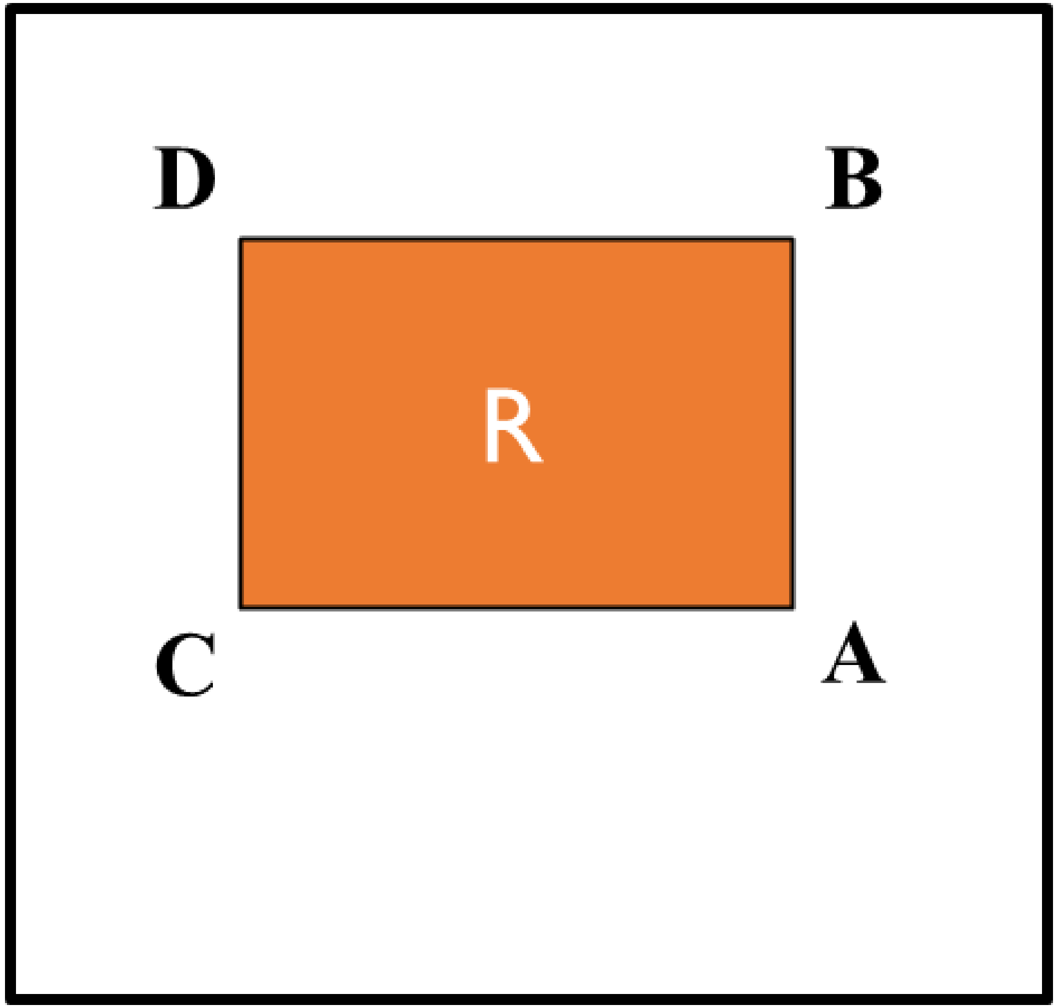

- Cumulative row sum: $s(x, y) = s(x–1, y) + i(x, y)$
- Integral image: $ii(x, y) = ii(x, y−1) + s(x, y)$

Python: ii = np.cumsum(i)

# Computing Sum within a Rectangle

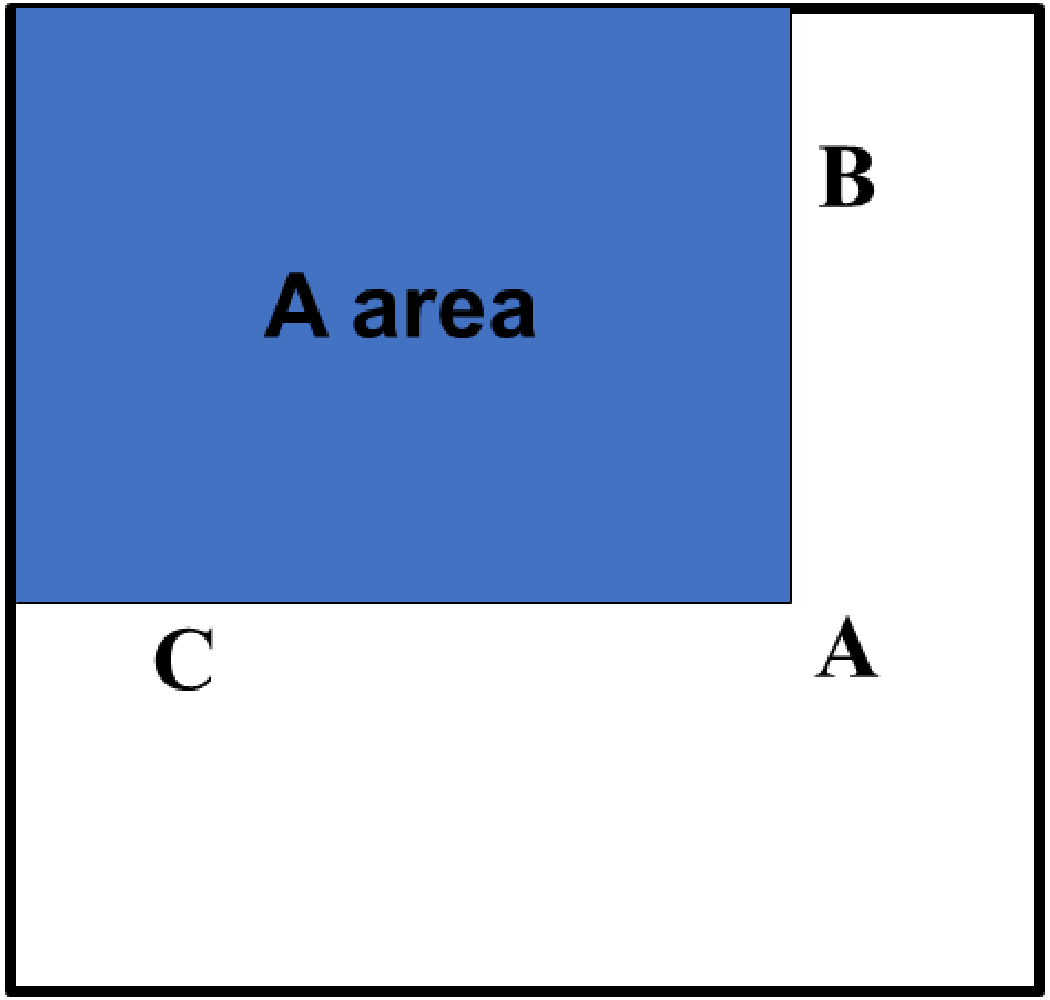
- Let R be a desired rectangle

- A,B,C,D are the values of the integral image at the corners of R



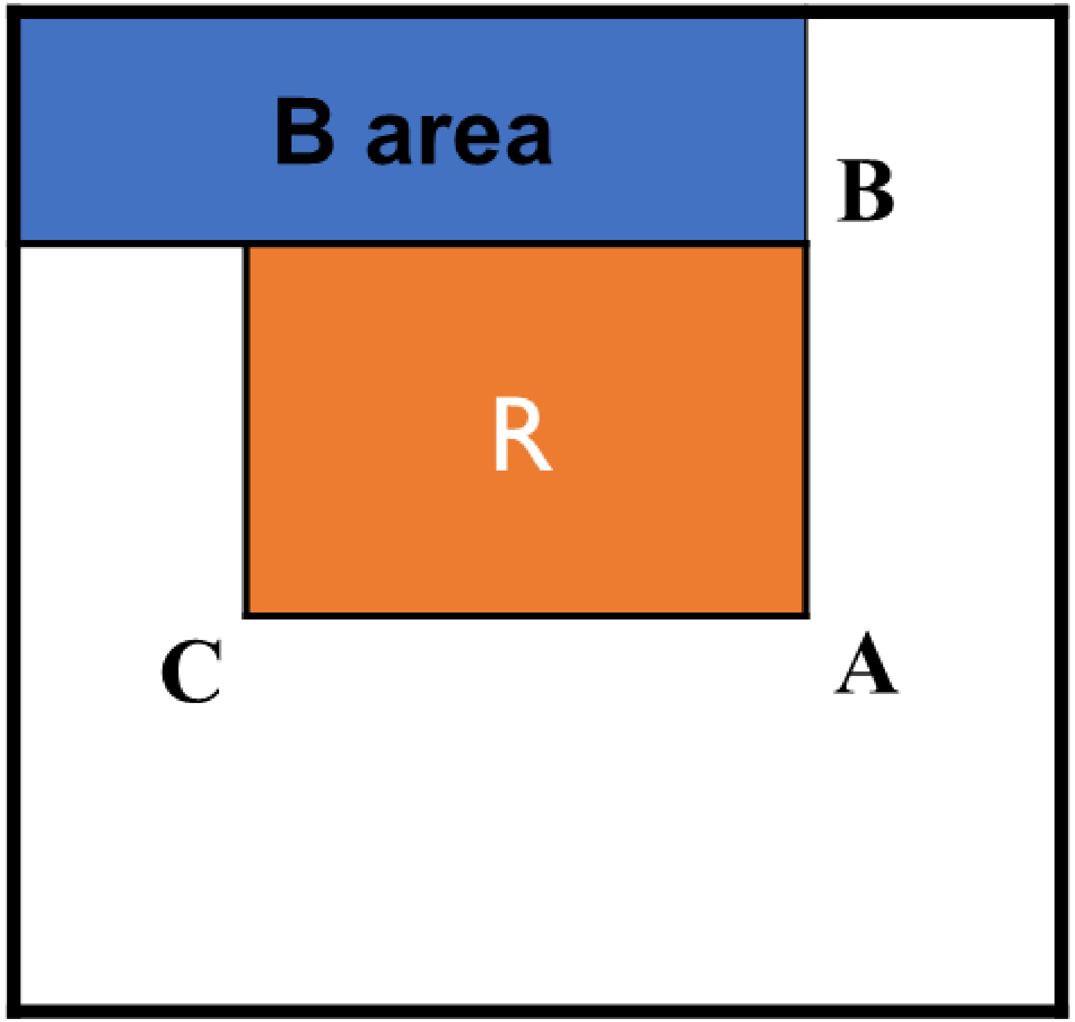
# Computing Sum within a Rectangle

- Let R be a desired rectangle
- A,B,C,D are the values of the integral image at the corners of R



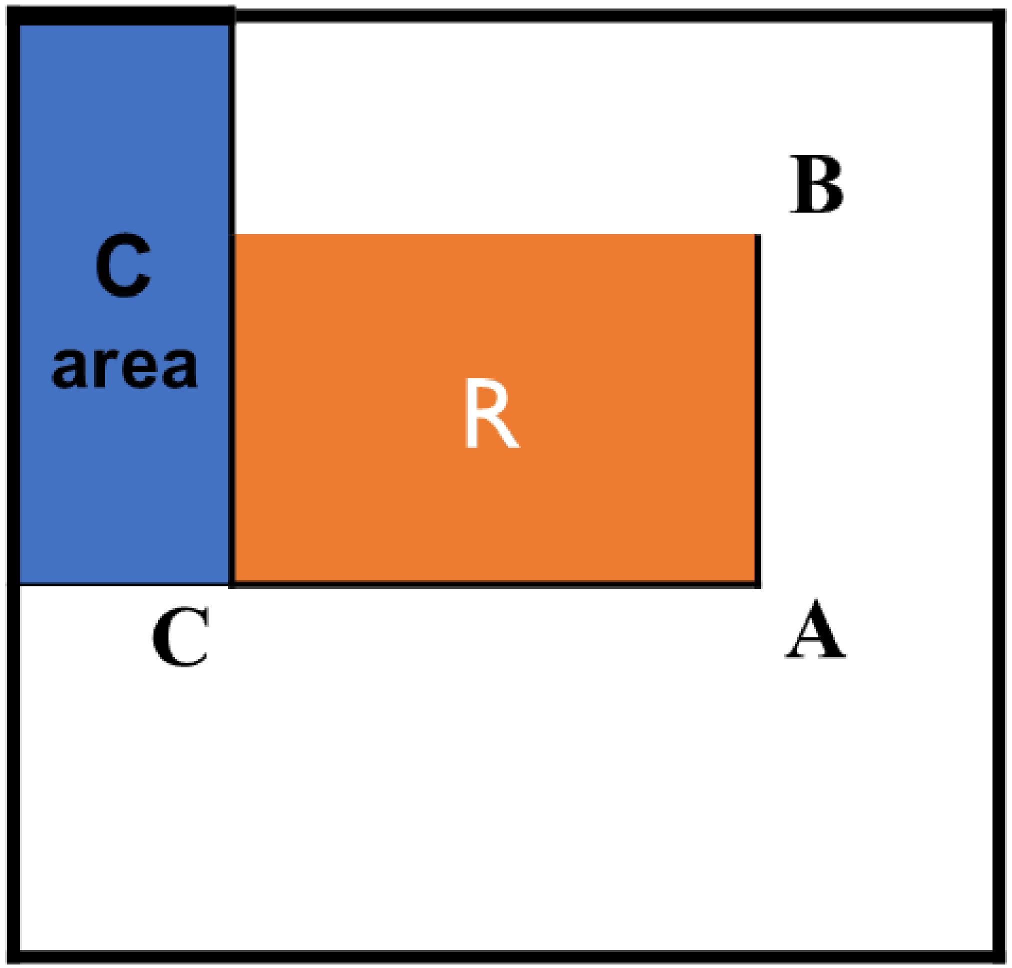
# Computing Sum within a Rectangle

- Let R be a desired rectangle
- A,B,C,D are the values of the integral image at the corners of R



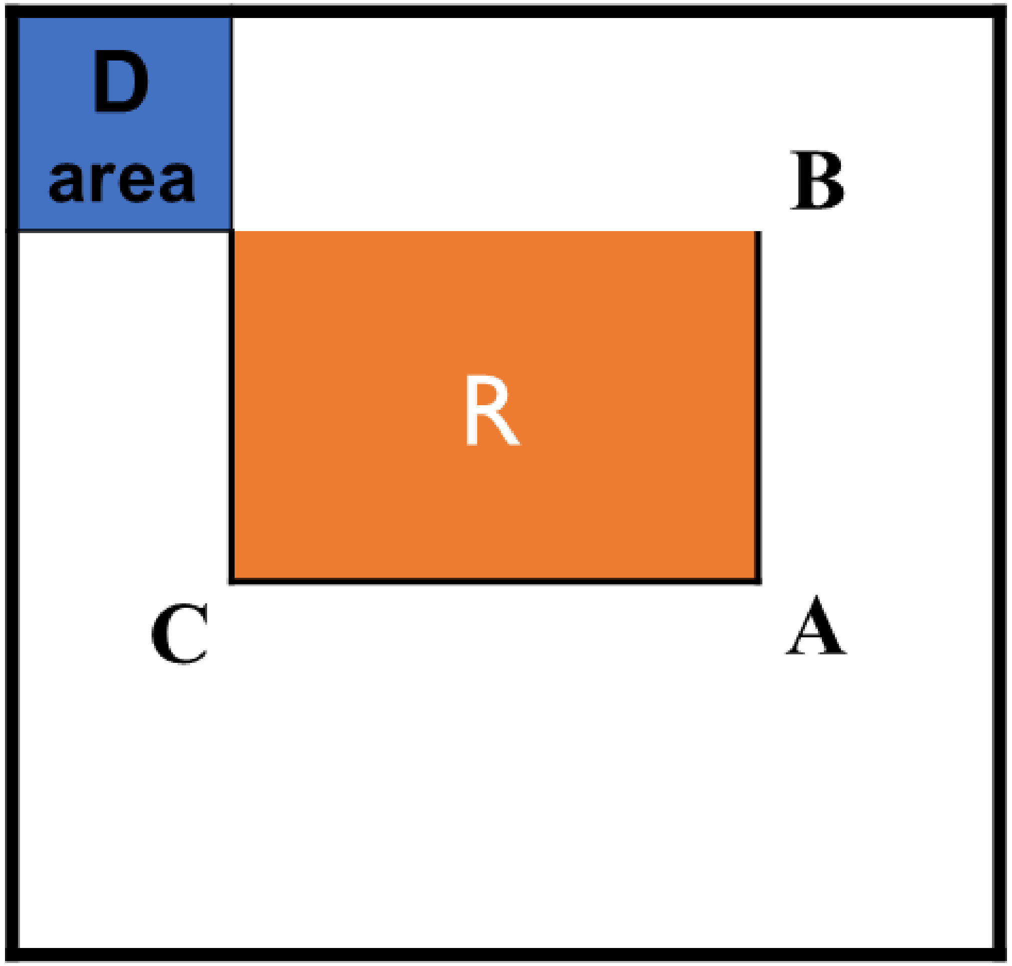
# Computing Sum within a Rectangle

- Let R be a desired rectangle
- A,B,C,D are the values of the integral image at the corners of R



# Computing Sum within a Rectangle

- Let R be a desired rectangle
- A,B,C,D are the values of the integral image at the corners of R

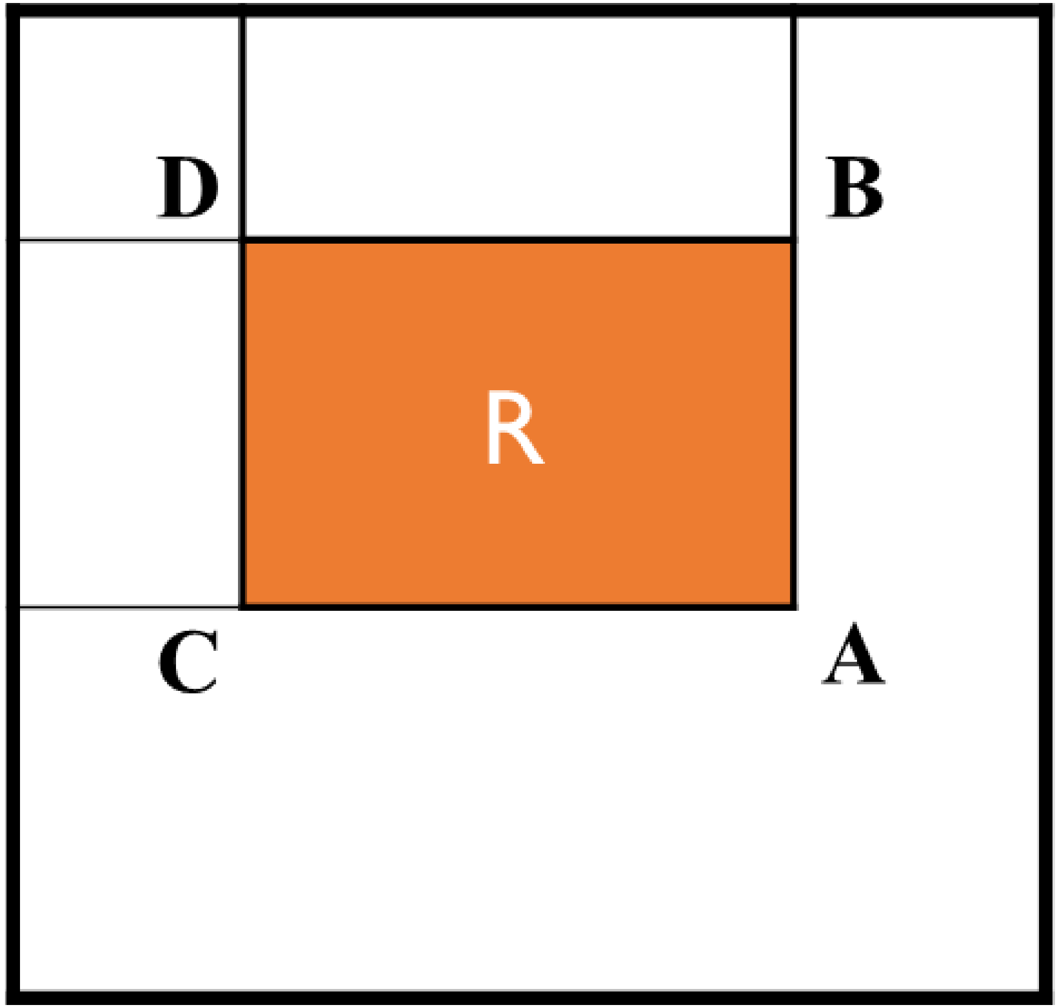


# Computing Sum within a Rectangle

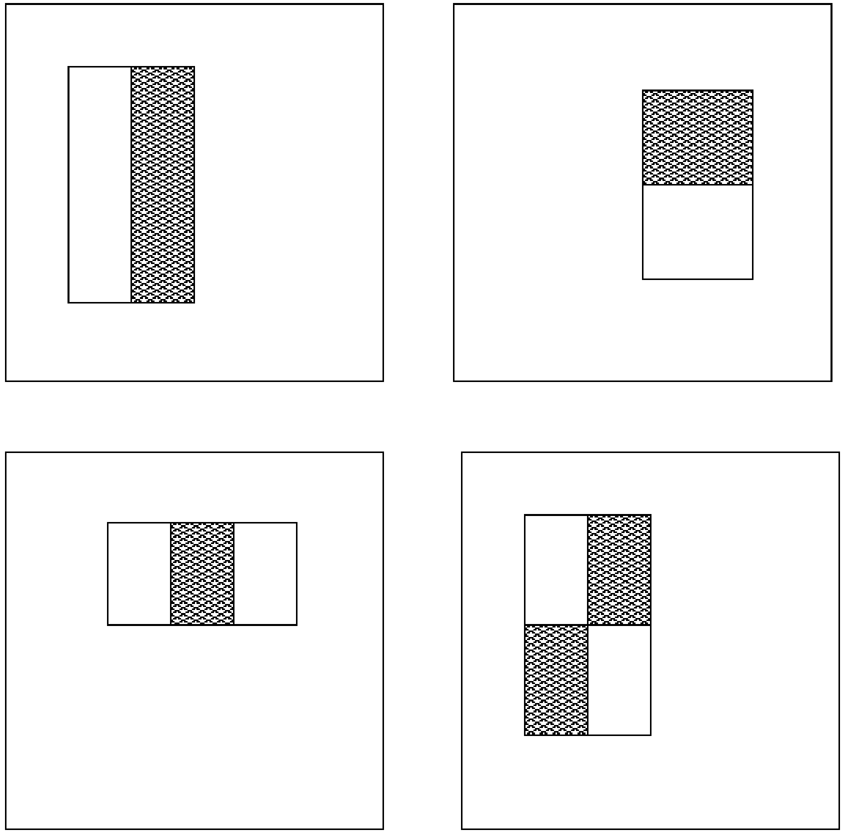
- Let R be a desired rectangle
- A,B,C,D are the values of the integral image at the corners of R
- The sum of original image values within the rectangle can be computed as:

  sum = A – B – C + D

  **Only 3 additions are required for any size of rectangle!**

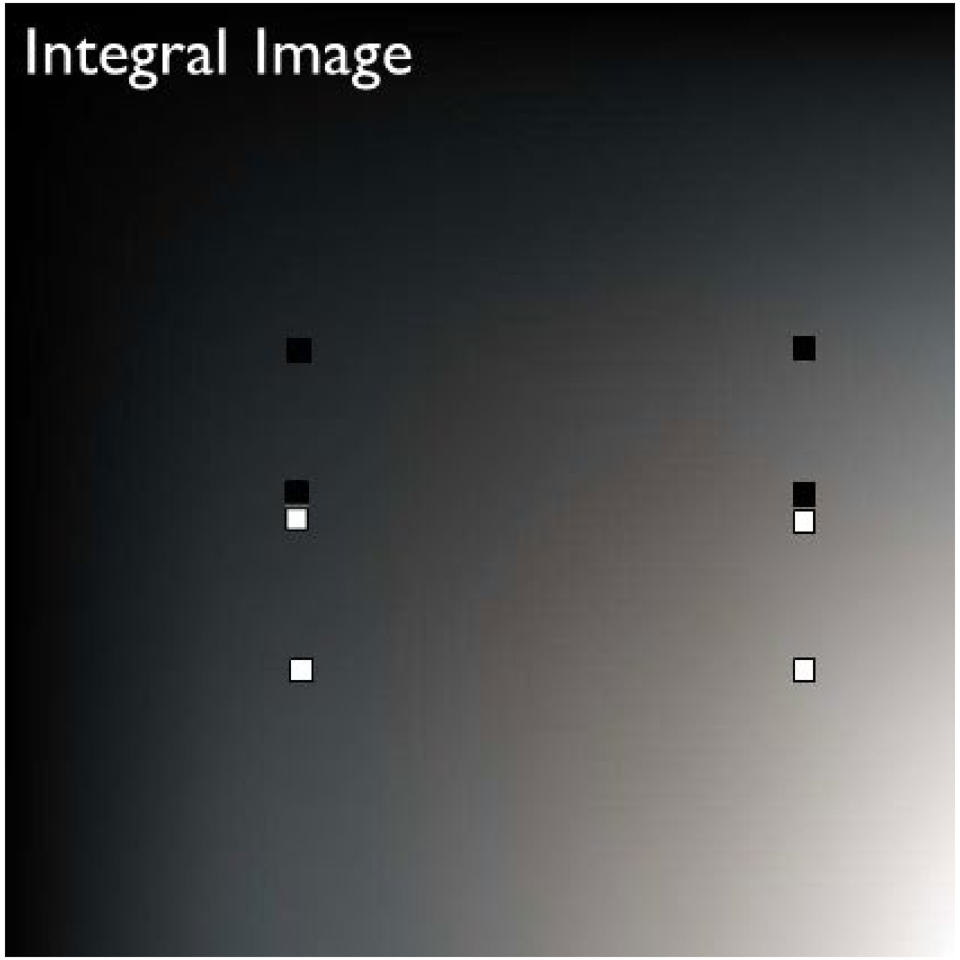


# Computing a Haar Feature



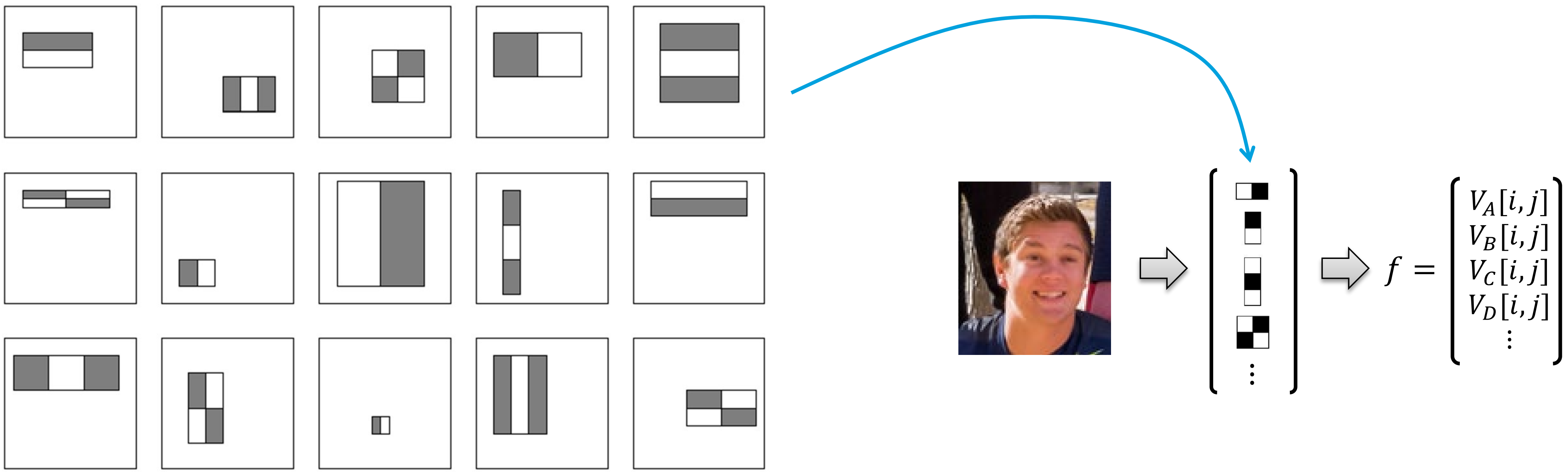
$$V_A[i,j] = \sum(\text{pixel intensities in white area}) - \sum(\text{pixel intensities in black area})$$

# Computing a Haar Feature

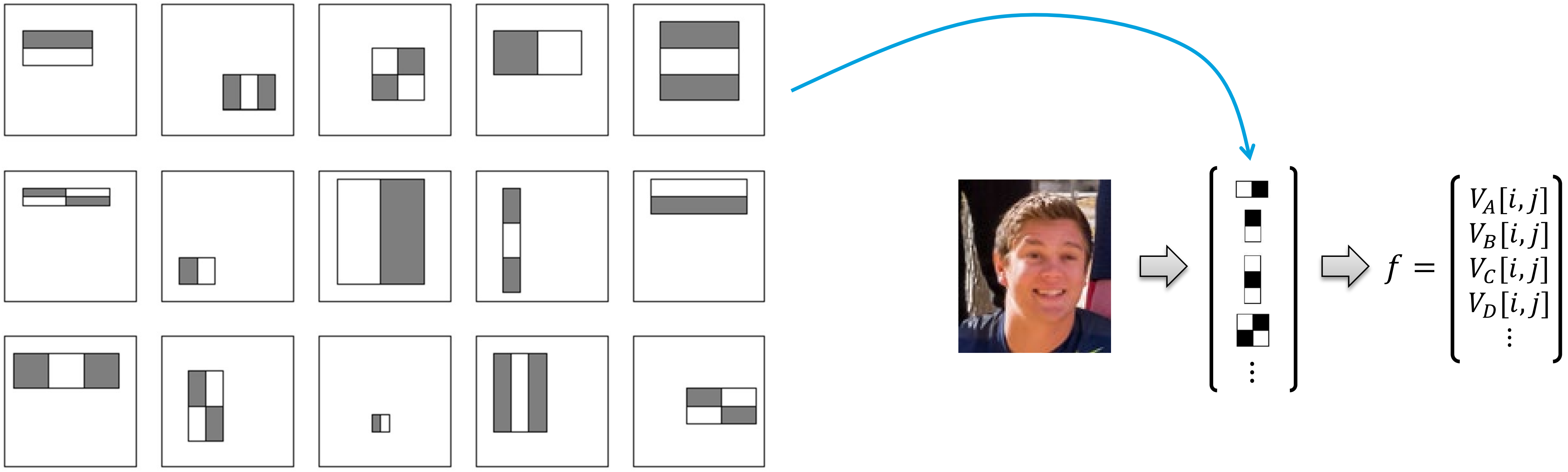


# Number of Features

- Individual Haar features are 'weak classifiers'
  - Jargon: 'feature' and 'classifier' are used interchangeably. Also, 'learner' and 'filter'.
- But, what if we combine thousands of them?



# Number of Features

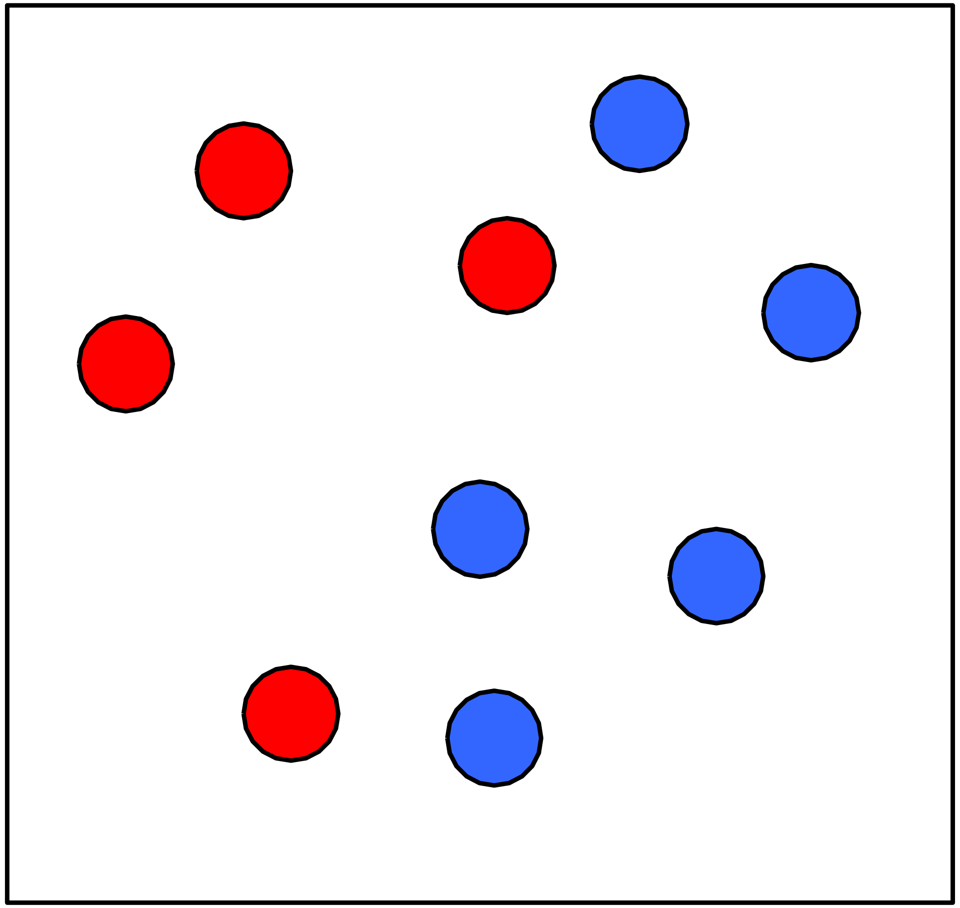



- For a 24x24 detection region, the number of possible rectangle features is ~160,000!
- At test time, it is impractical to evaluate the entire feature set
- Can we learn a ‘strong classifier’ using just a small **subset** of all possible features?

# Boosting for Feature Selection

Initially, weight each training example equally

Weight = size of point

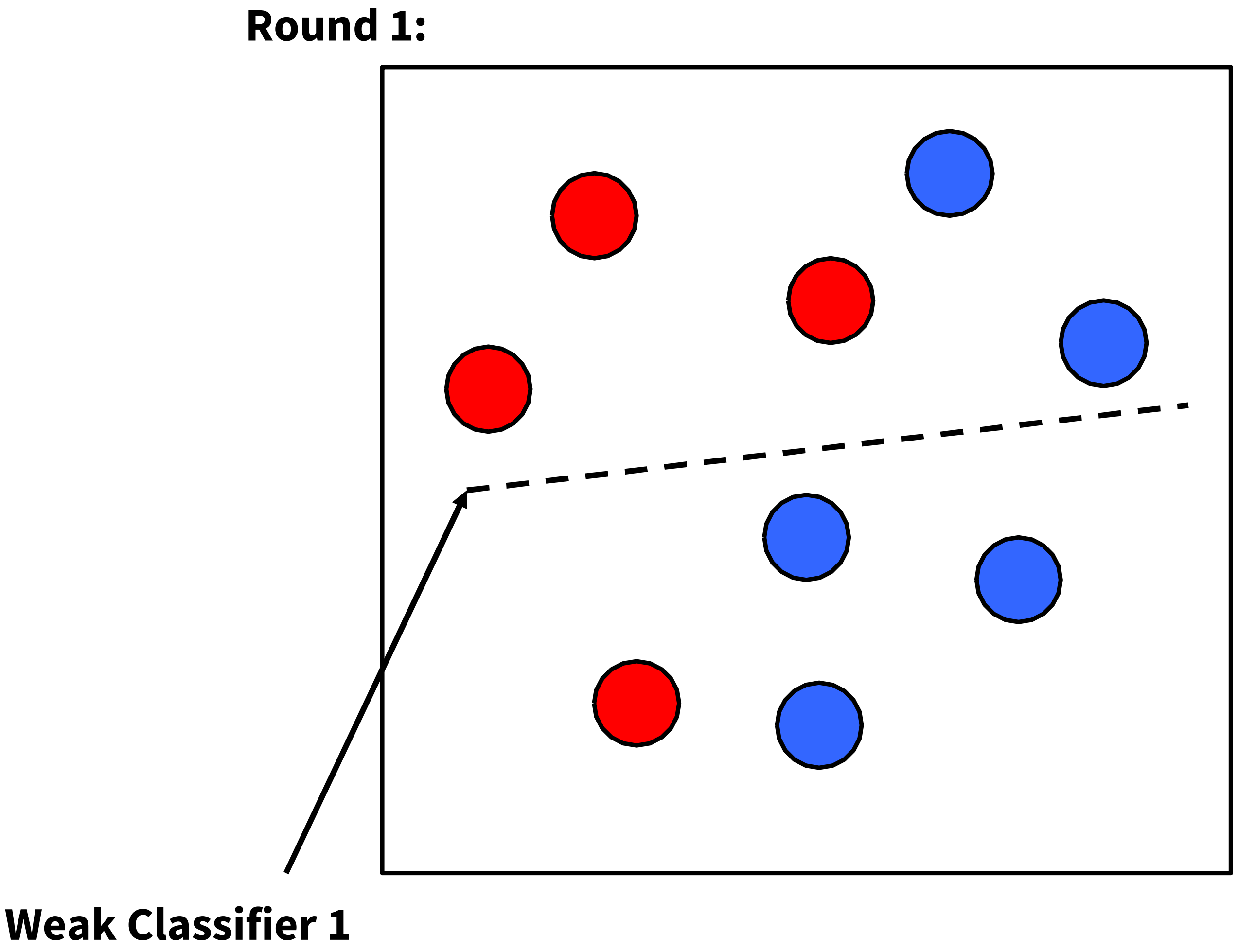


# Boosting for Feature Selection

In each boosting round:

Find the weak classifier that achieves the lowest *weighted* training error.

Raise the weights of training examples misclassified by current weak classifier.



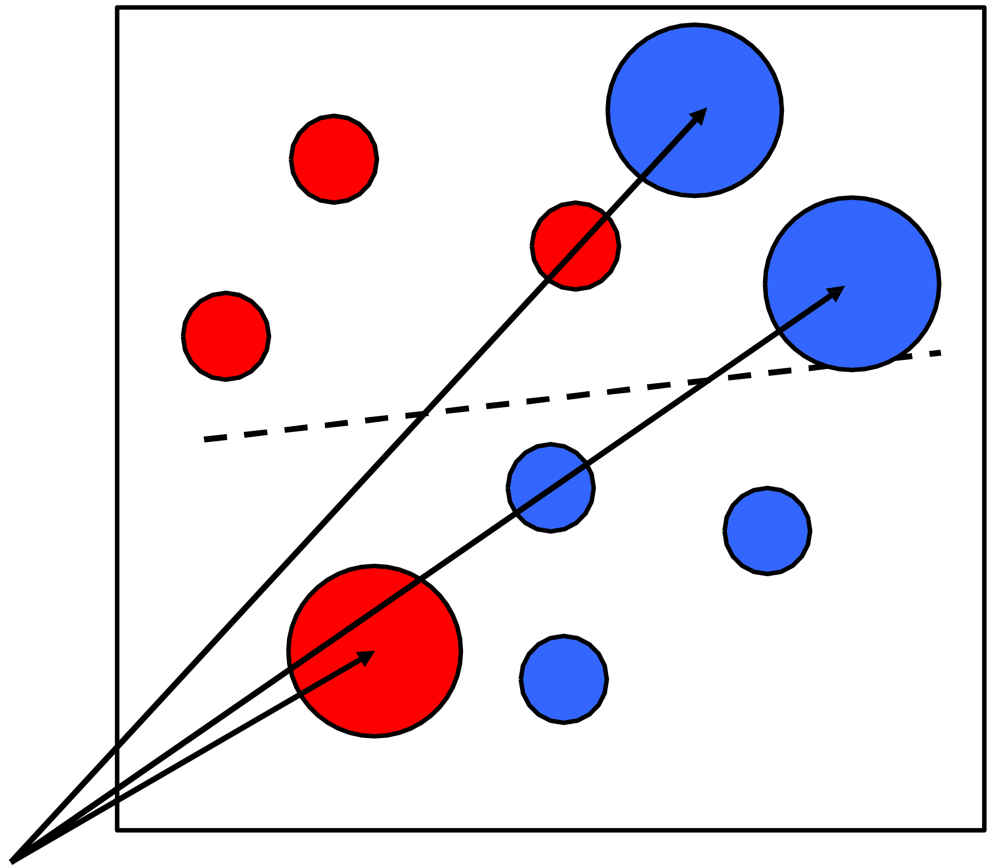
# Boosting Illustration

In each boosting round:

Find the weak classifier that achieves the lowest *weighted* training error.

Raise the weights of training examples misclassified by current weak classifier.

**Round 1:**



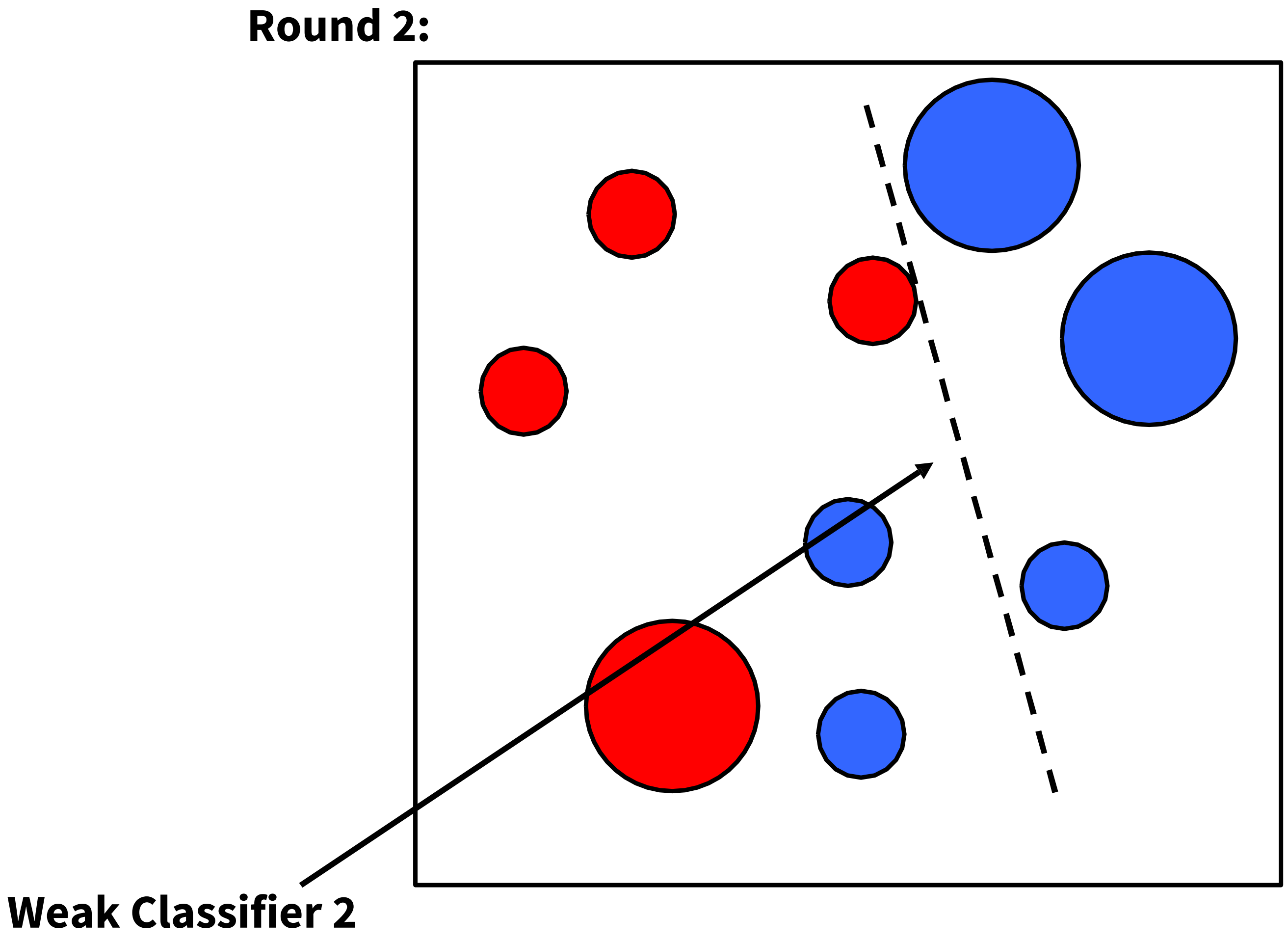
**Weights Increased**

# Boosting Illustration

In each boosting round:

Find the weak classifier that achieves the lowest *weighted* training error.

Raise the weights of training examples misclassified by current weak classifier.



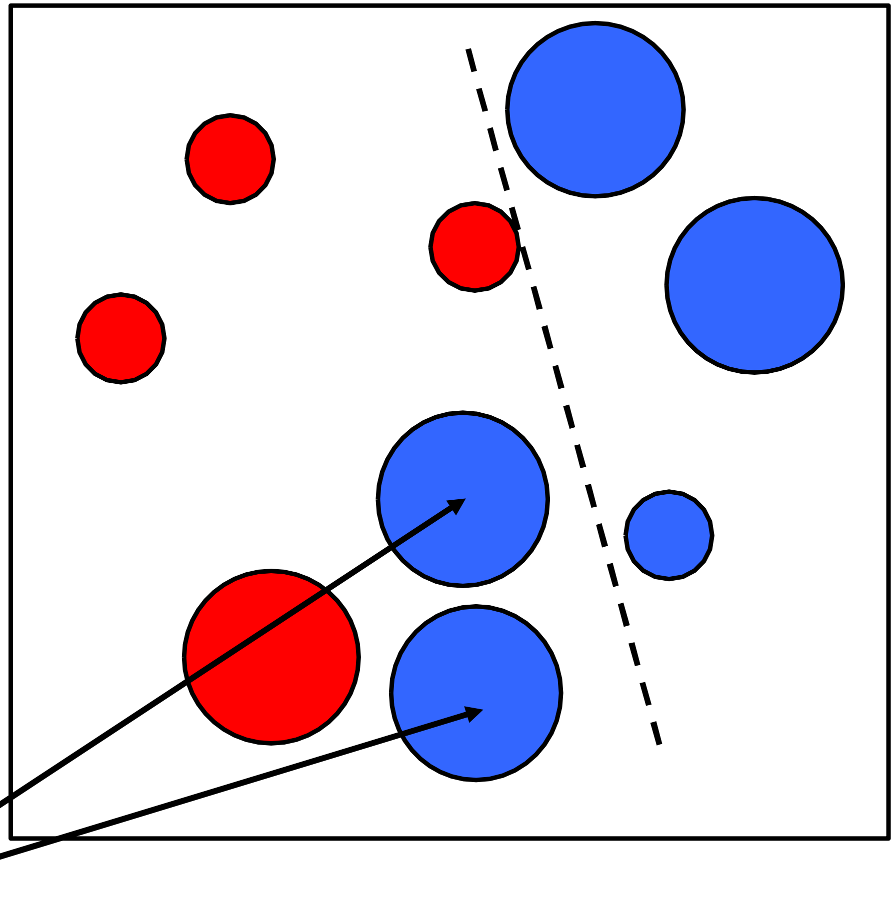
# Boosting Illustration

In each boosting round:

Find the weak classifier that achieves the lowest *weighted* training error.

Raise the weights of training examples misclassified by current weak classifier.

**Round 2:**



**Weights Increased**

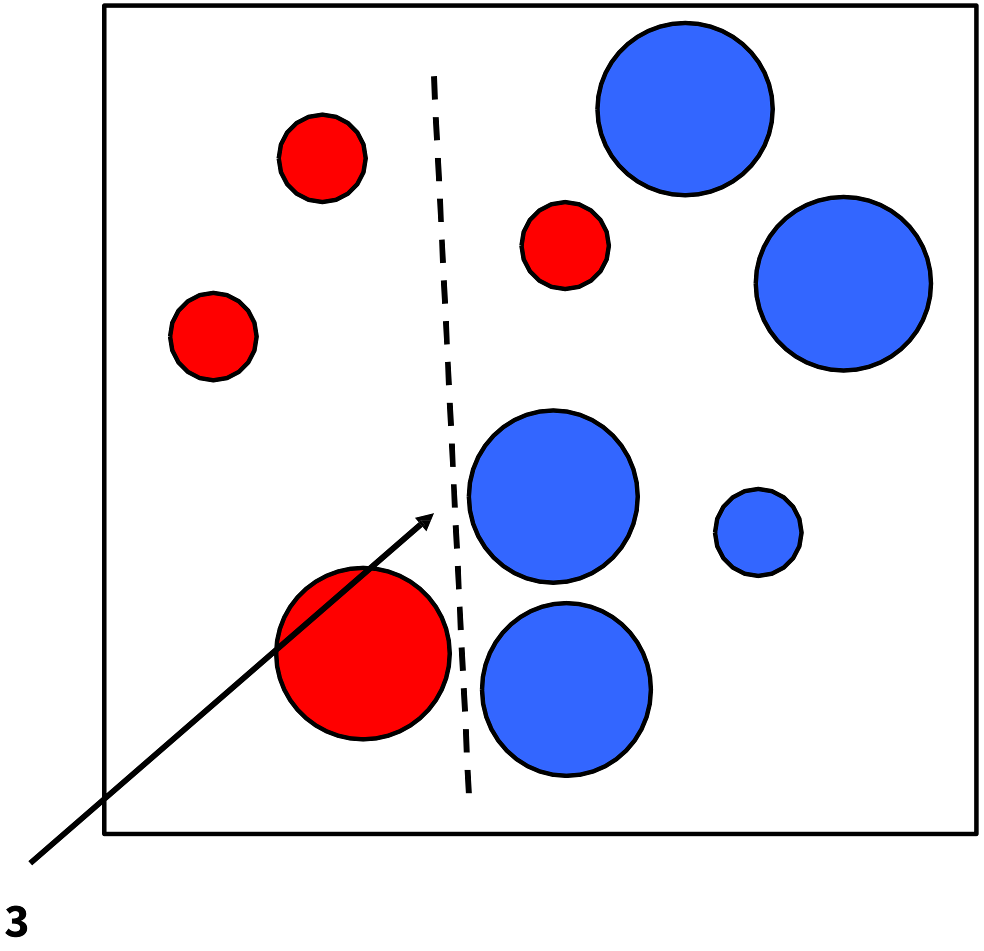
# Boosting Illustration

In each boosting round:

Find the weak classifier that achieves the lowest *weighted* training error.

Raise the weights of training examples misclassified by current weak classifier.

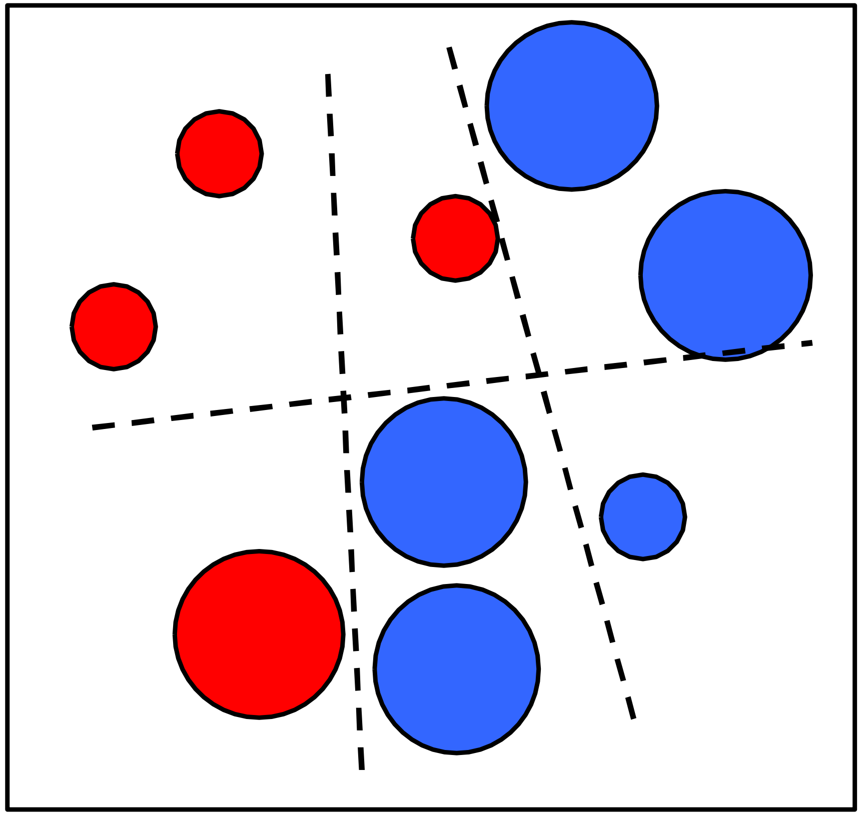
**Round 3:**



# Boosting Illustration

- Compute final classifier as linear combination of all weak classifier.
- Weight of each classifier is directly proportional to its accuracy.
- Exact formulas for re-weighting and combining weak learners depend on the boosting scheme (e.g., AdaBoost)

**Round 3:**



**AdaBoost**: Y. Freund and R. Schapire,A short introduction to boosting, Journal of Japanese Society for Artificial Intelligence, 14(5):771-780, September, 1999.

# Feature Selection with Boosting

- Create a large pool of features (160K)
- Select discriminative features that work well together

Final strong learner

window

$$h(\mathbf{x}) = \text{sign}\left(\sum_{j=1}^{M} \alpha_j h_j(\mathbf{x})\right)$$

Weak learner

Learner weight

- "Weak learner" = feature + threshold + 'polarity'

value of rectangle feature

$$h_j(\mathbf{x}) = \begin{cases} -s_j & \text{if } f_j < \theta_j \\ s_j & \text{otherwise} \end{cases}$$

threshold

'polarity' = black or white region flip $\longrightarrow$ $s_j \in \pm 1$

- Choose weak learner that minimizes error on weighted training set, then reweight

1. Input the positive and negative training examples along with their labels $\{(\boldsymbol{x}_i, y_i)\}$, where $y_i = 1$ for positive (face) examples and $y_i = -1$ for negative examples.
2. Initialize all the weights to $w_{i,1} \leftarrow \frac{1}{N}$, where $N$ is the number of training examples. (Viola and Jones (2004) use a separate $N_1$ and $N_2$ for positive and negative examples.)
3. For each training stage $j = 1 \ldots M$:
   (a) Renormalize the weights so that they sum up to 1 (divide them by their sum).
   (b) Select the best classifier $h_j(\boldsymbol{x}; f_j, \theta_j, s_j)$ by finding the one that minimizes the weighted classification error

$$e_j = \sum_{i=0}^{N-1} w_{i,j} e_{i,j}, \quad (14.3)$$
$$e_{i,j} = 1 - \delta(y_i, h_j(\boldsymbol{x}_i; f_j, \theta_j, s_j)). \quad (14.4)$$

   For any given $f_j$ function, the optimal values of $(\theta_j, s_j)$ can be found in linear time using a variant of weighted median computation (Exercise 14.2).

   (c) Compute the modified error rate $\beta_j$ and classifier weight $\alpha_j$,

$$\beta_j = \frac{e_j}{1 - e_j} \quad \text{and} \quad \alpha_j = -\log \beta_j. \quad (14.5)$$

   (d) Update the weights according to the classification errors $e_{i,j}$

$$w_{i,j+1} \leftarrow w_{i,j} \beta_j^{1 - e_{i,j}}, \quad (14.6)$$

   i.e., downweight the training samples that were correctly classified in proportion to the overall classification error.

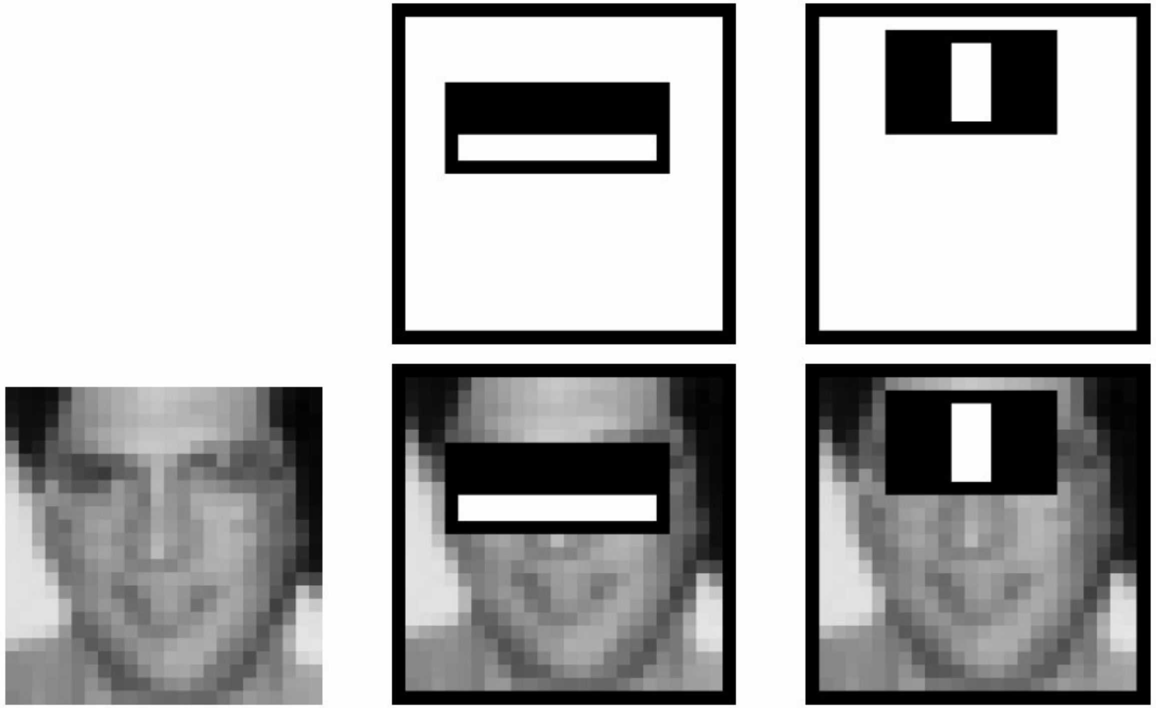
4. Set the final classifier to

$$h(\boldsymbol{x}) = \text{sign}\left[\sum_{j=0}^{m-1} \alpha_j h_j(\boldsymbol{x})\right]. \quad (14.7)$$

AdaBoost pseudocode Szeliski P.665

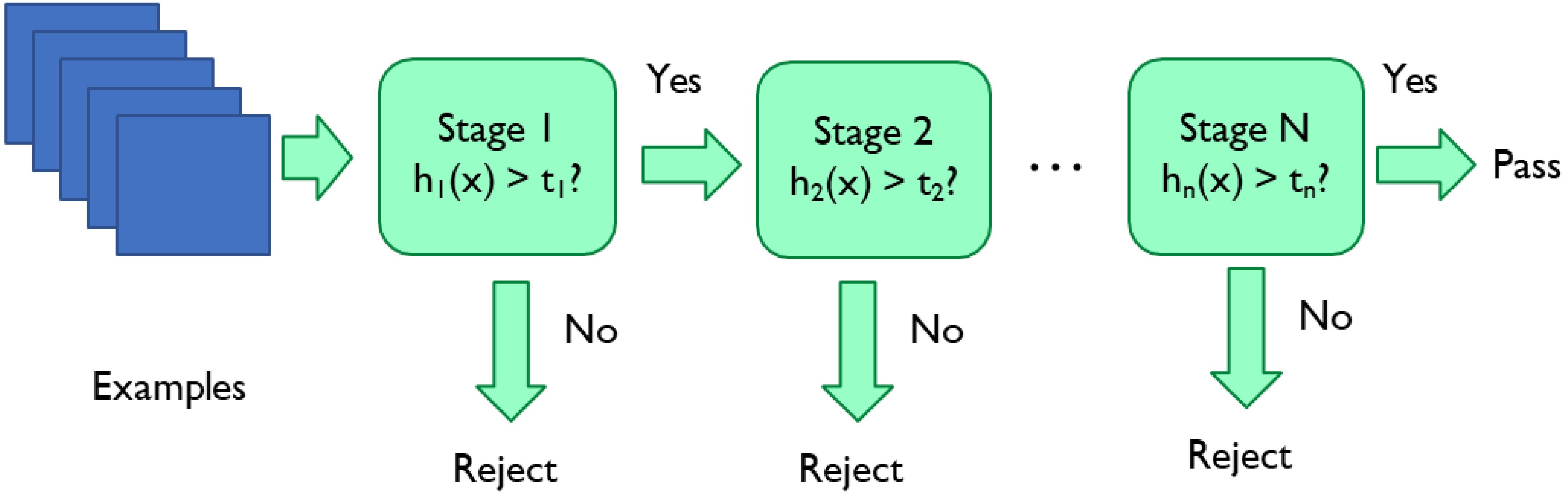
# Boosting for Face Detection

- First two features selected by boosting:



- This feature combination can yield 100% recall and 50% false positive rate

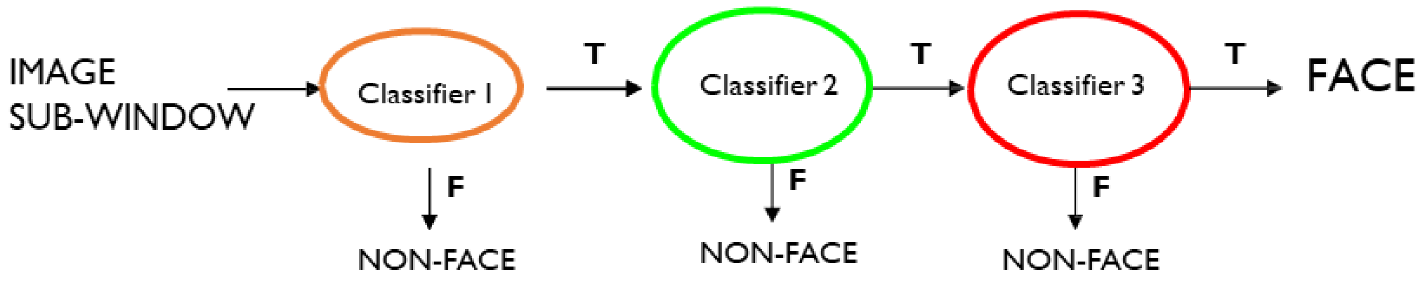
# Attention Cascade for Fast Detection

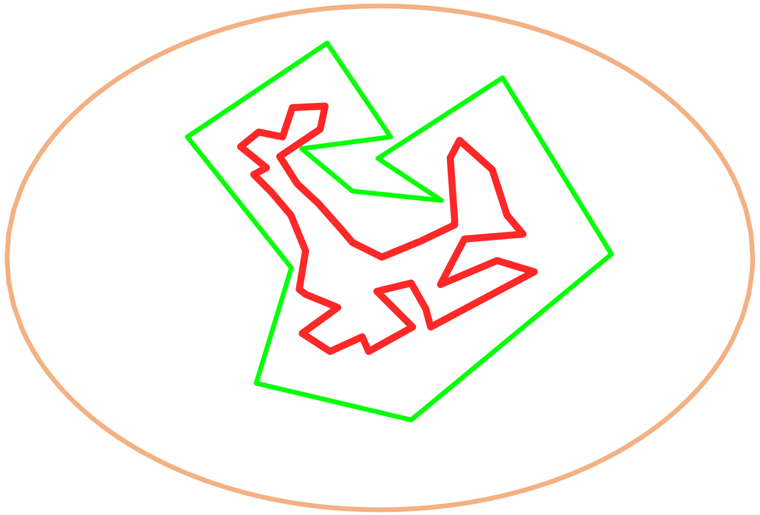


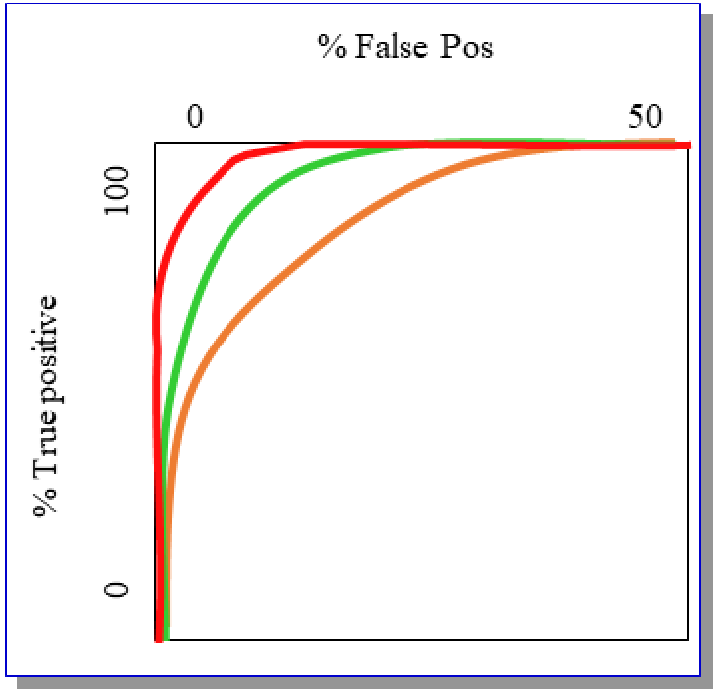

- Fast classifiers early in cascade which reject many negative examples but detect almost all positive examples

- Slow classifiers later, but most examples don’t get there

# Attentional Cascade

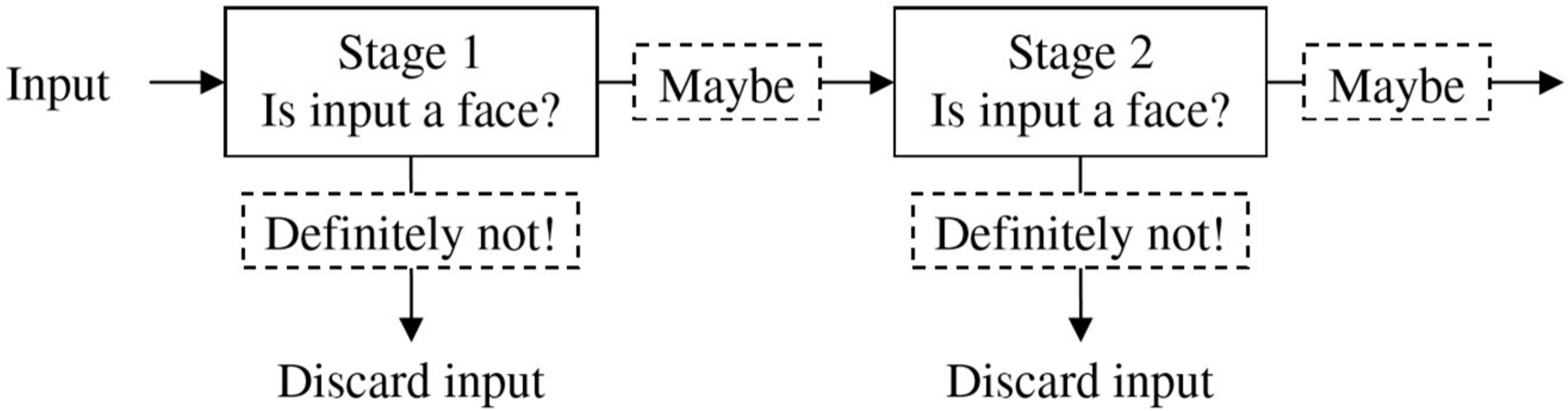





- Chain classifiers that are progressively more complex
- Minimize **false positive** rates at each stage, not absolute error

# Training the Cascade




- Set target detection and false positive rates for each stage
- Keep adding features to the current stage until its target rates have been met
  - Need to lower boosting threshold to maximize detection (as opposed to minimizing total classification error)
  - Test on a *validation set*
- If the overall false positive rate is not low enough, then add another stage
- Use false positives from current stage as the negative training examples for the next stage

# Viola Jones Results

- 38 stages with
  - 6061 total Haar features used out of 160K features candidates
- Training time:“weeks” on 466 MHz Sun workstation
- Average of 10 features evaluated per window on test set
- On 700 Mhz Pentium III processor, process 384x288 pixel image in 0.067 sec
  - 15 Hz
  - 15 times faster than previous detector of comparable accuracy (Rowley et al., 1998)