Face Detection system, CV Lab

Implementing Viola & Jones

Daniel Heilper & Natan Silnitsky

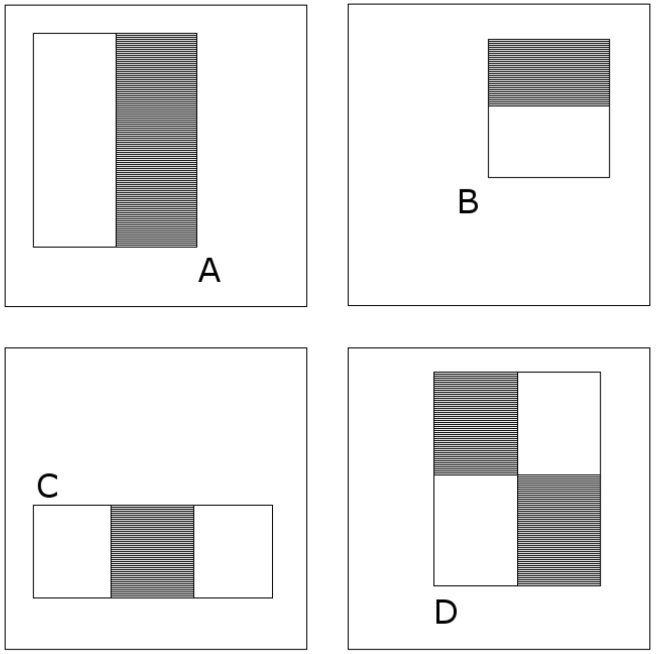
Robust Real-time Object Detection

- Features:

Sum of pixels in dark rectangles minus
Sum of pixels in white rectangles

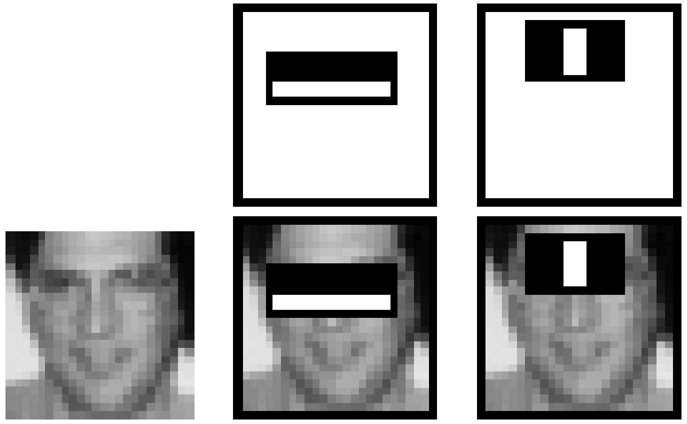
Scaled and shifted
across sub-window

Problem – a huge set of
features



Robust Real-time Object Detection

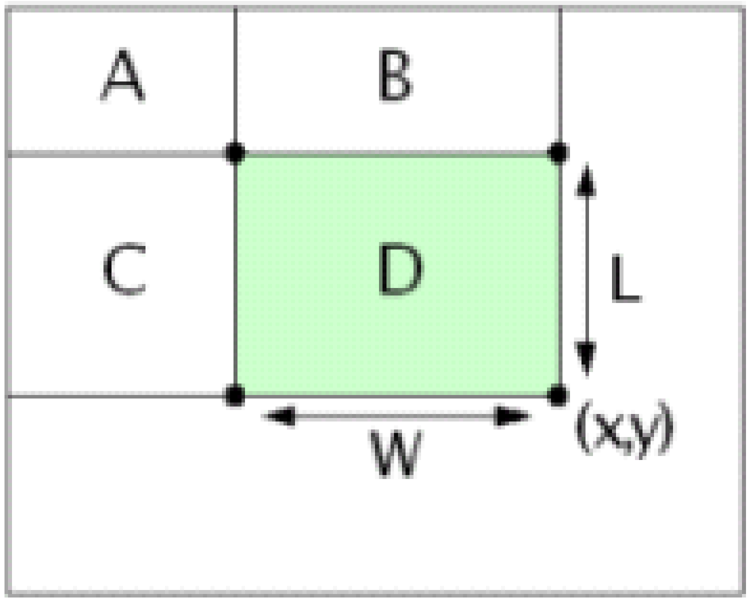
- Selected Features



Robust Real-time Object Detection

- Integral Image

Efficient Computation of features



Robust Real-time Object Detection

- AdaBoost

Boosting weak classifiers (i.e. features) to make 1 strong classifer.

strong classifer = Weighted Combination of weak classifiers

In every iteration the weights of the misclassified examples increase

Problem – Creating a classifier out of a feature.

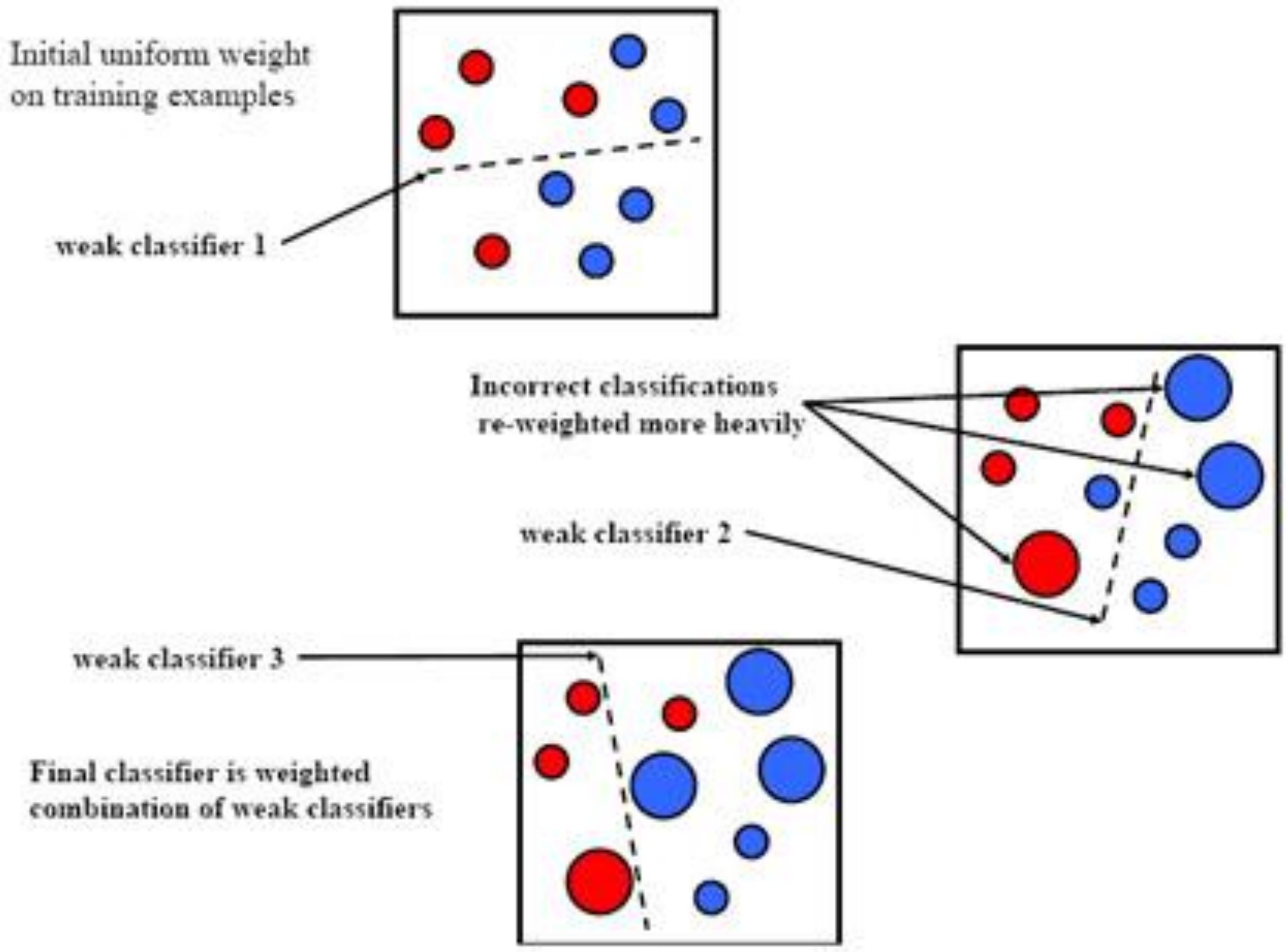
What threshold to take?

Robust Real-time Object Detection

- AdaBoost

Problem –

Threshold

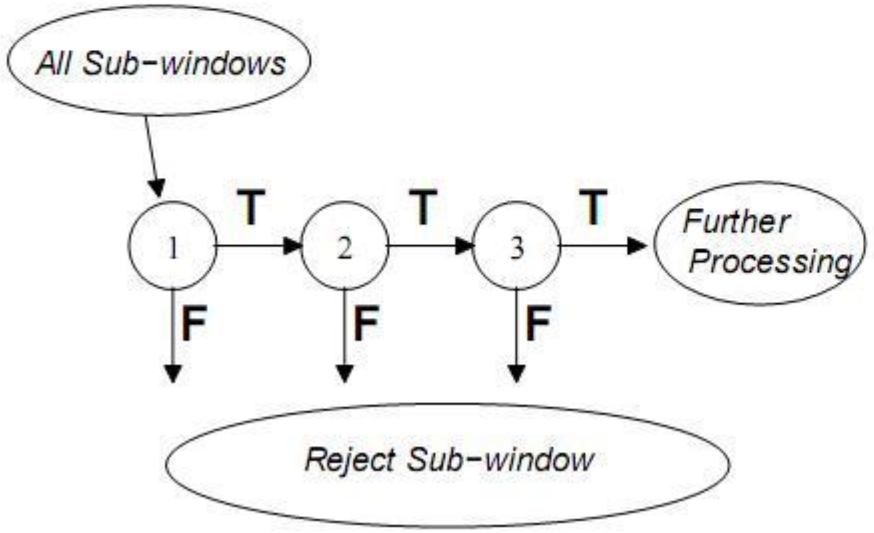

Initial uniform weight on training examples
weak classifier 1
Incorrect classifications re-weighted more heavily
weak classifier 2
weak classifier 3
Final classifier is weighted combination of weak classifiers

$$H(x) = sign(\alpha_1 h_1(x) + \alpha_2 h_2(x) + \alpha_3 h_3(x))$$

Robust Real-time Object Detection

- Cascade

Every (strong, boosted) Classifier only deals with blocks not discarded by previous classifier

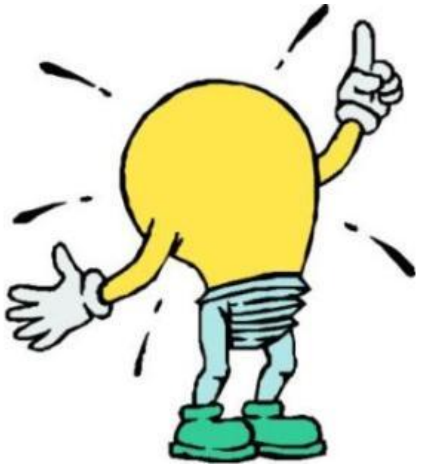


Stop when achieved

Desired detection rate

Problem – Sparse Vector Maintenance

Approach to solving the problem



Use of Matlab

- Vector & matrix operations are basic operation in Matlab.
- Matlab makes redundant the need for memory management (allocation, deallocation).
- it has very good I/O capabilities vis-à-vis Images.
- the Matlab environment is optimized for vector & matrix debugging (easy access to complex structures

Use of Matlab (2)

- Matlab is an interpreted language, and is inherently slow.
- In view of this we decided to migrate time consuming parts of the code to c, if running time turned out to be slow.
- Any high time complexity task will inherently be slow both in Matlab & native languages, therefore we focused more on reducing algorithmic time complexity than lower complexity constants.

Finding Feature threshold – Full Search

- Sort the values array.
- Create 2 additional arrays, positive and negative.
- Scan the retrieved values array. Fill the arrays by Accumulating units of 1 multiplied by the value's weights in the following fashion:
- For a positive value,
  in the positive array, add the previous element's content and add 1 multiplied by the value's weight.
  in the negative array just add the previous element's content
- For a negative value,
  in the negative array, add the previous element's content and add 1 multiplied by the value's weight.
  in the positive array just add the previous element's content

Finding Feature threshold – Full Search (2)

- The value index with the best threshold will be the index where the absolute difference between the arrays content is greatest.
- Polarity is the side of the threshold which has more positive examples,
- meaning polarity is positive if more positive examples occur on the left side of the threshold, and polarity is negative if there is less on that side.
- Quick feature error calculation is achieved by logical vector operation (xor) between the classified vector and the sorted label vector (the “Y”).

Finding Feature threshold – Example

- Pos examples feature application values

Value	9	5	4	10	1
Classification	1	1	1	1	1

- Neg examples feature application values

Value	10	15	12	14	8
Classification	0	0	0	0	0

- Sorted examples feature application values

Value	1	4	5	8	9	10	10	12	14	15
Classification	1	1	1	0	1	1	0	0	0	0

- Positive examples accumulation array

accumulation	1	2	3	3	4	5	5	5	5	5

- Negative examples accumulation array

accumulation	0	0	0	1	1	1	2	3	4	5

- Absolute difference between the arrays

abs-diff	1	2	3	2	3	4	3	2	1	0
threshold	1	4	5	8	9	10	10	12	14	15

- The chosen threshold is **10**, polarity is positive (because more positive examples are on the left of the threshold)

Reducing Time complexity

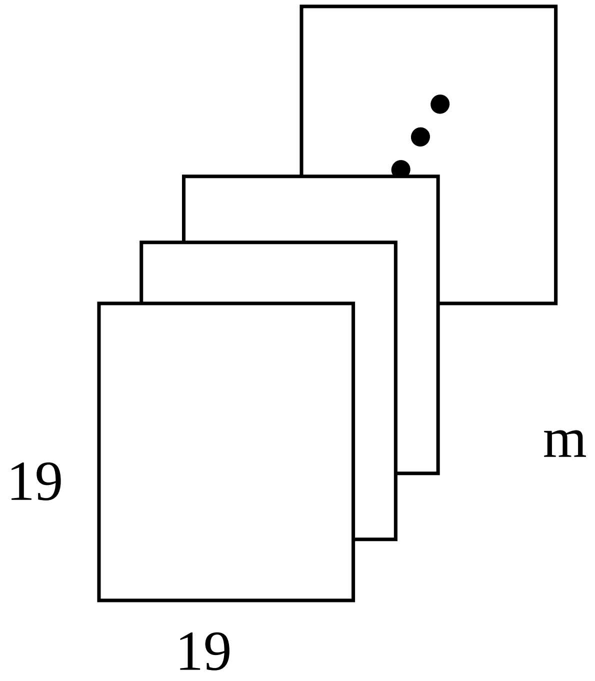
- building data structures such that in Matlab it was is possible to compute operation on them in 1 function call and avoid loops.
- The overhead is an initial construction of these data structures.
- Data Structure Examples follows:

“Boxing” the examples

- Training Examples are 19X19 frames.
- During Training, every features is applied on all integral images of the training examples.
- In order to perform fast Integral Image computation, and to perform fast application of features on the examples, we decide to put all of the example frames in a 3D box.
- We use the 3D box in Testing as well, putting all of the sliding windows of the test image in it.

“Boxing” the examples (2)

- A 3D box with *m* training examples in it



“Boxing” the examples (3)

- Once all of the examples or sliding windows are in the box, simultaneous computation of each frame's Integral Image is as easy as 2 lines of code in matlab (no loops involved):
  box = cumsum(box,1);
  box = cumsum(box,2);
- computing a feature's application simultaneously on all examples or sliding windows becomes much faster, (colon operation in Matlab).
- Padding the image examples with zeros (as the 1st row & 1st column)

Calculating Feature application values only Once

- In training, Feature application values are computed before we start the learning process for all training examples simultaneously.
- The result is a matrix with m rows representing m features & n columns representing n examples. Each matrix element is a feature applied on a training example.
- Because of this data structure we only compute the feature application for each training example once during the entire training process (adaboost + cascade).

Calculating Feature application values only Once (2)

- All iterations of the Adaboost process use the same feature application values, including threshold determination & misclassification calculation.
- Even when cascade training process discards parts of the negative examples that are definitely not considered as a face, there is no need for any new calculations. Instead of discarding the irrelevant negative examples, the corresponding irrelevant feature application values are discarded.

Simultaneous Misclassification Calculation & Examples Weights Update

- Updating Weights Adaboost stage is performed as such that misclassification calculation of the example set & the weights updating of the example set are done at once.
  Pseudo Code:
  - sort examples values
  - apply threshold
  - **xor** with labels
  - update weights for every misclassified example

Reduce correctness testing time

- When we finished writing the code, we decided we will not immediately jump to check if we got nice results on test images, but write unit tests instead, i.e. methodically test for bugs and correctness of each of the functions we wrote.
- In order for these unit tests to be independent of each other (so there will be no "running error") we simulated new inputs for each of the functions.
- Only after we had finished this, we started testing to see if our implementation gives meaningful results.
- The thought behind doing extensive unit testing was to significantly reduce functional testing time and to focus our minds first on simple bugs and in later stage on more complex errors in implementation.

Results

- Our final detector is a cascade of 9 classifiers with a total of 206 features.
- The training set was comprised of Faces & Non-Faces examples of size 19X19.
- The first 5 classifiers were trained on 2429 faces & 4548 non-faces from the *CBCL face database* using the cascade method (we set the detection rate to be 99% and the false positive rate to be 0.25%).

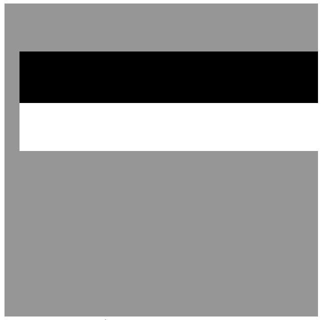
- The next 3 classifiers were trained independently on 4500 Non-faces examples each, that were taken from the previous partial cascade resulting false alarms on random images from the web, which had no faces in them
- (we set the detection rate to be 99.7% and the false positive rate to be 0.1%).

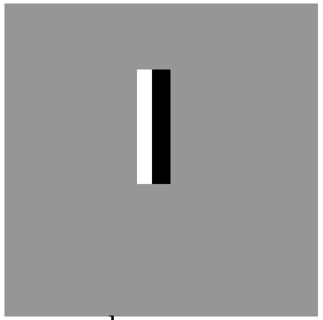
- The final 9th classifier was trained in the same way as the previous three, only this time the non-faces examples were taken from the partial cascade comprising 8 classifiers resulting false alarms
- (we again set the detection rate to be 99.7% and the false positive rate to be 0.1%).

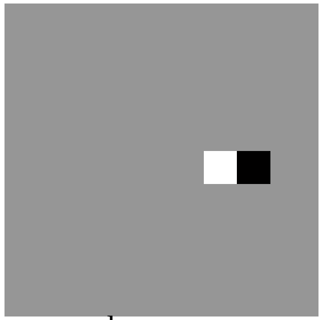
- Detecting is done multiple scales up to a ceiling of 384X288. Images sized above this ceiling are downsized.
- The final cascade:

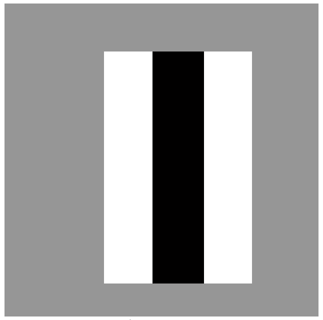
stage	1st cascade					2nd group			Last one
Number of features	5	8	9	10	6	38	35	47	48

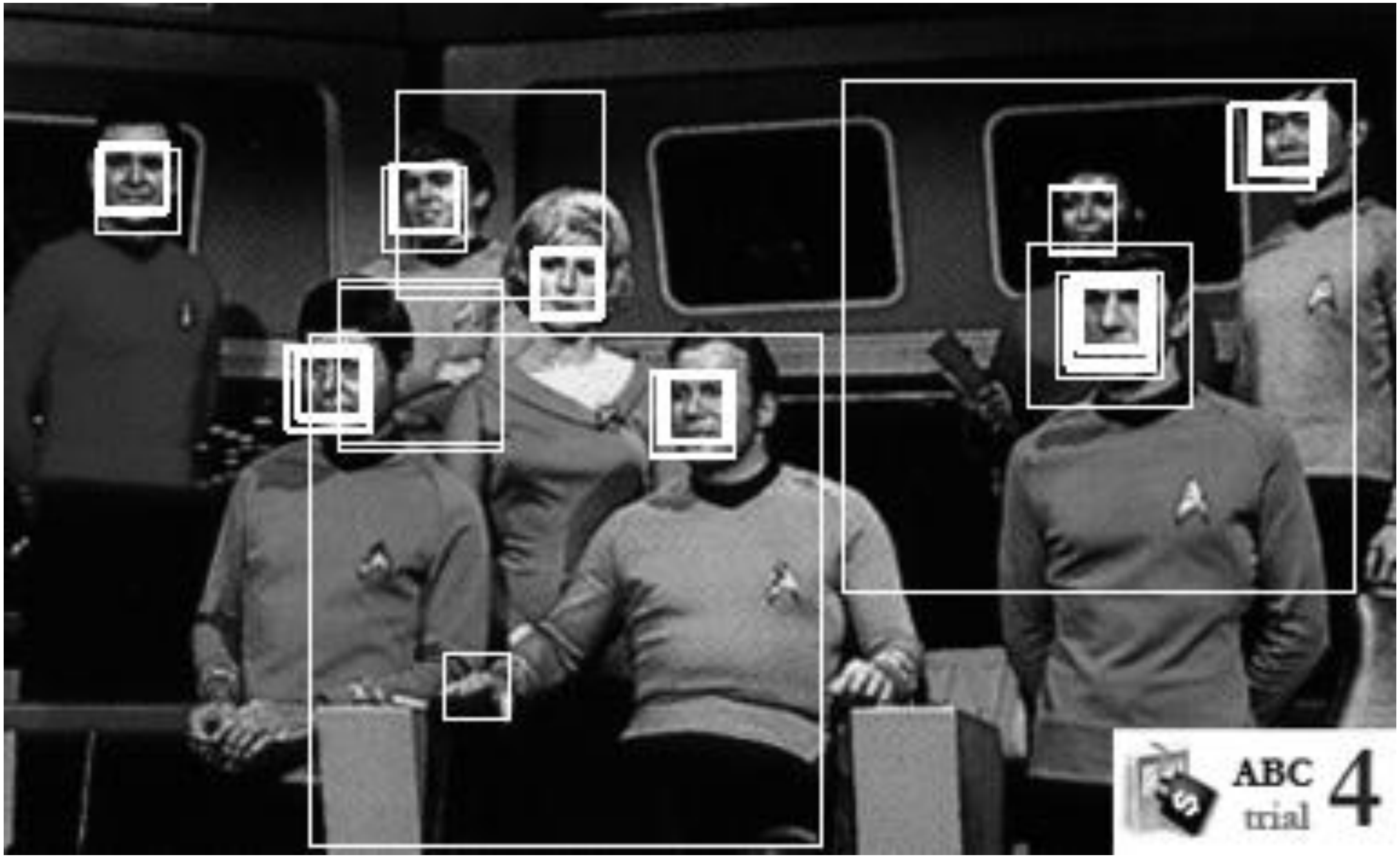
Selected Haar features

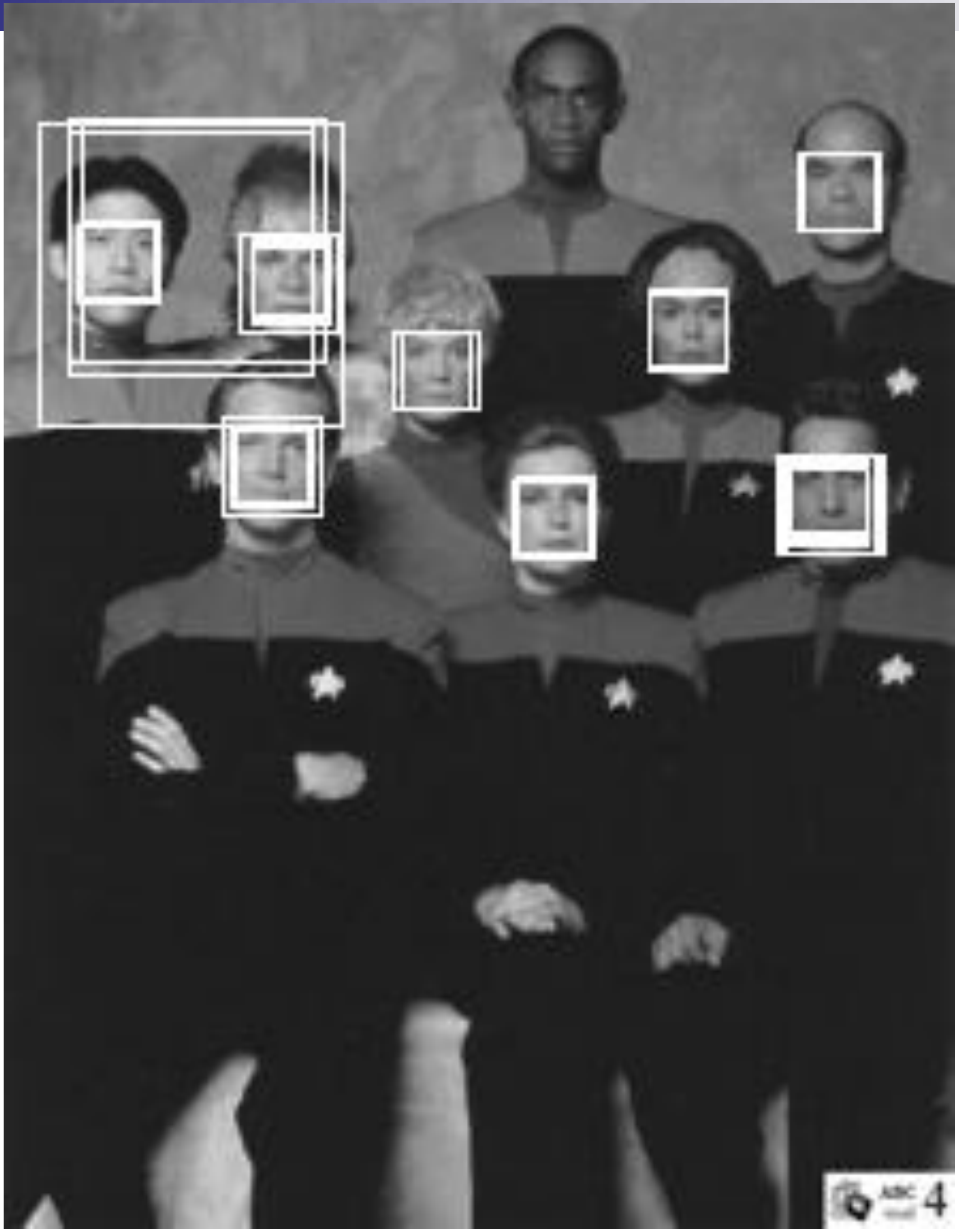

1st feature

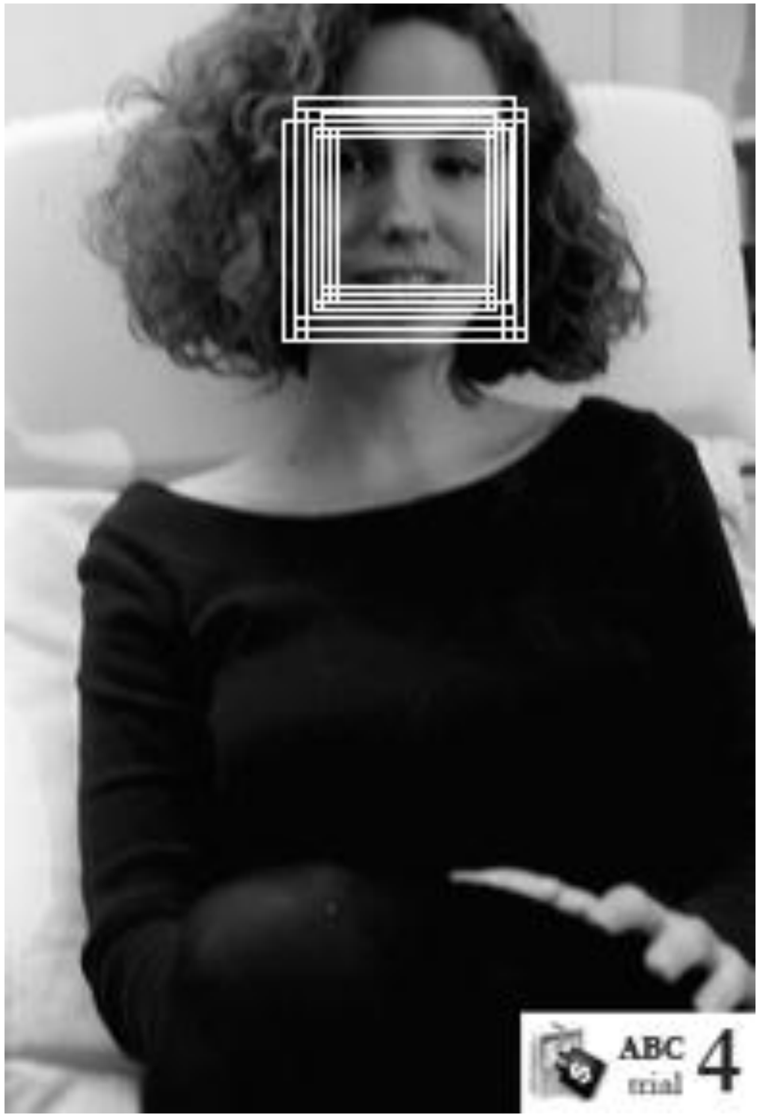

2nd feature

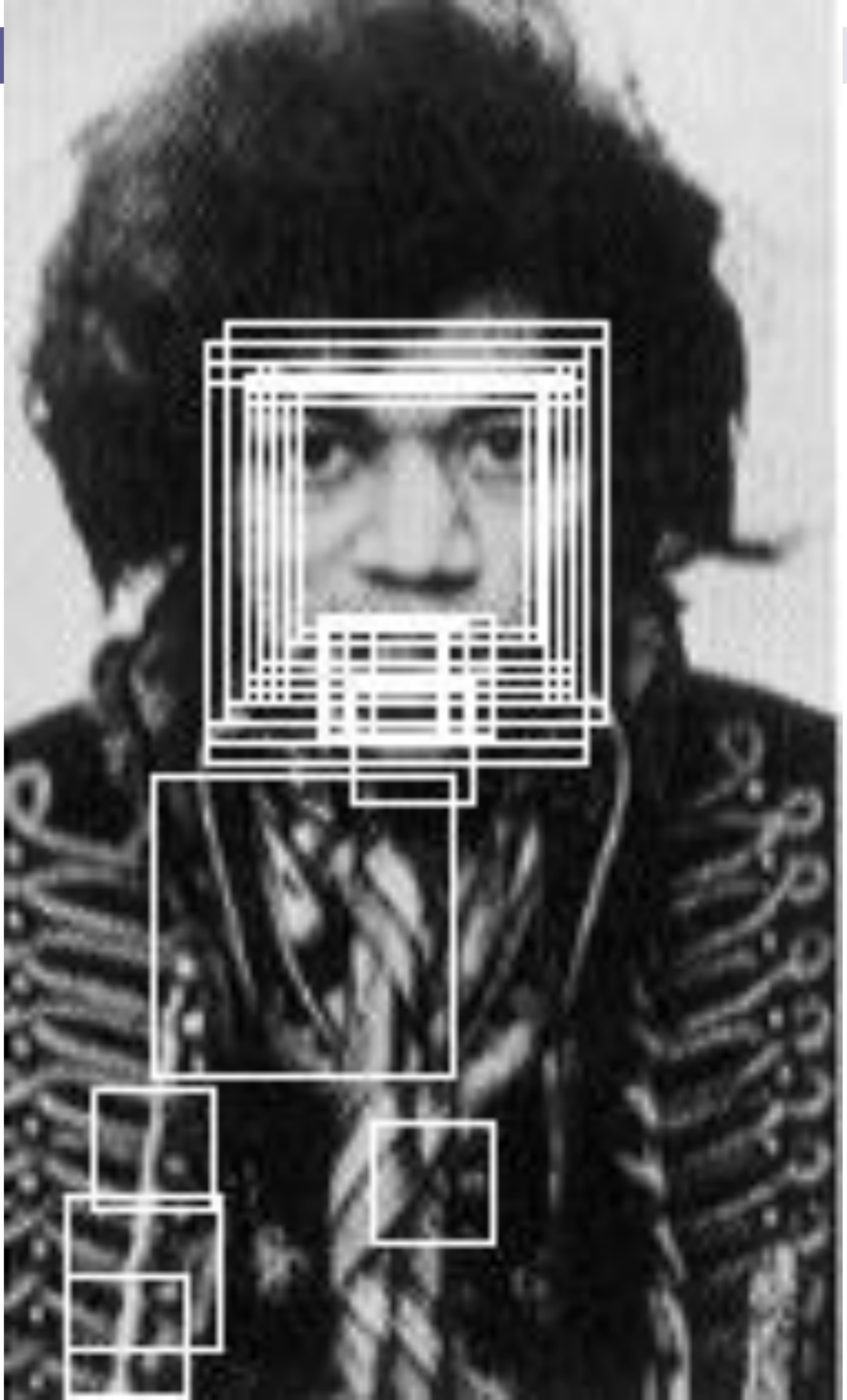

3rd feature


201st feature


ABC
trial
4




ABC
trial
4



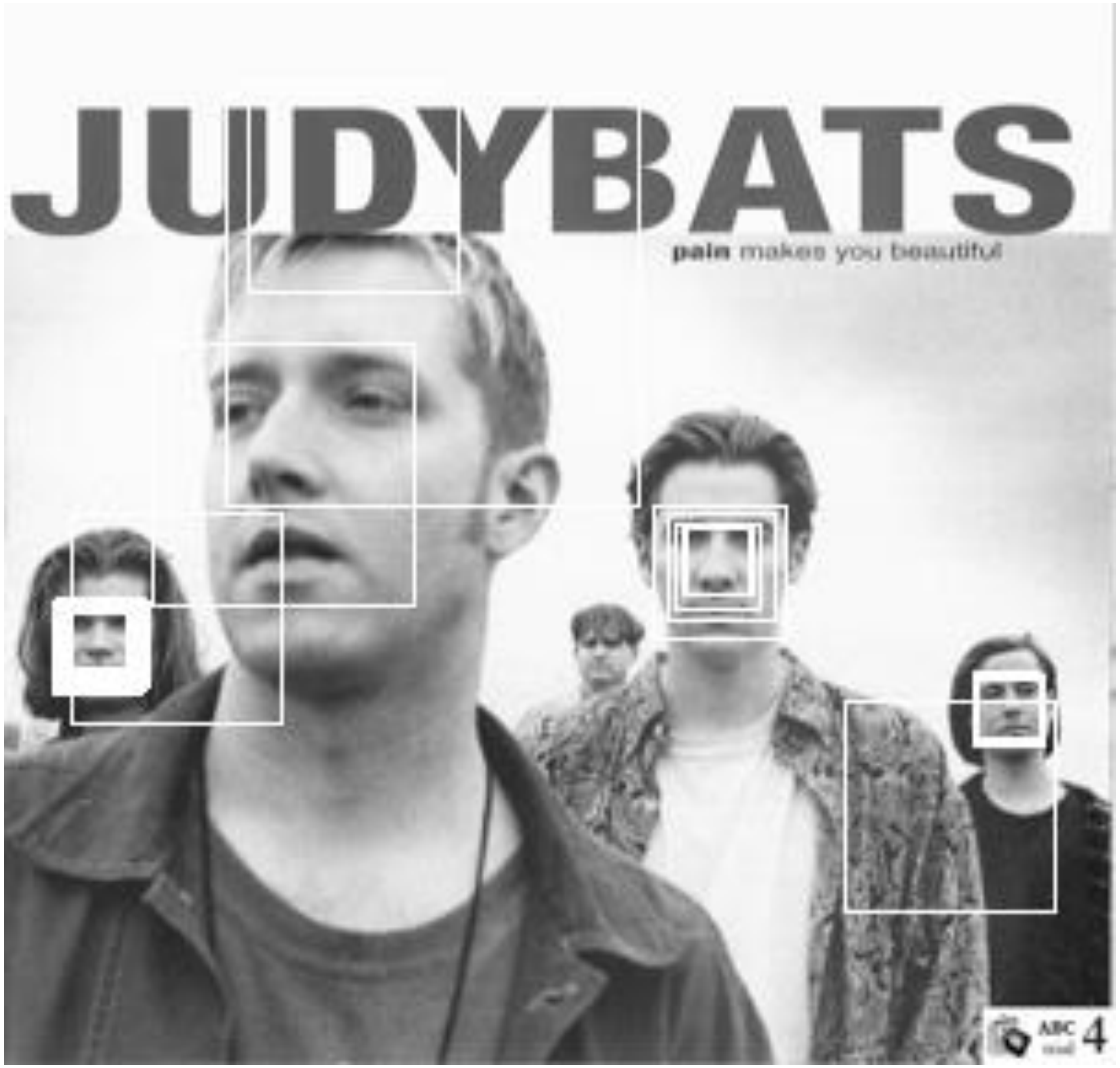

JUDYBATS
pain makes you beautiful

Image Name	Detection Rate(%)	Detected/ Overall	Detection Rate(%) *Frontal, No Facial Hair, 19X19 and above, Closed Mouth, Open Eyes*	Detected/ Overall *Frontal, No Facial Hair, 19X19 and above, Closed Mouth, Open Eyes*	False alarms	Elapsed Time (seconds)
Weighted Average(*)	**49**		**63**		**2.7**	**8.11**

Analysis

Unnecessary recalculation (which we discarded)

- After analyzing initial training results we’ve noticed superfluous recalculation of features in Adaboost.
- Cascade calls Adaboost iteratively with the *numOfFeatures* argument increasing incrementally in order to achieve a lower false alarm rate.
- In our original implementation, in every call to Adaboost, the process found the best features in the amount of *numOfFeatures* all over again, even though it calculated (*numOfFeatures – 1)* in the previous time it was called by cascade.
- The reason for this inefficiency was our original <u>bottom up approach</u> (see “Order of Implementation”) which focused on Adaboost only, not realizing the strong coupling with cascade.

Overfitting

- While experimenting with training parameters, we slowly raised detection rate and lowered false positive rate.
- After some point (99.9% detection rate & 0.01% false positive rate) less faces from the CMU dataset were detected, while not improving false alarm rate.

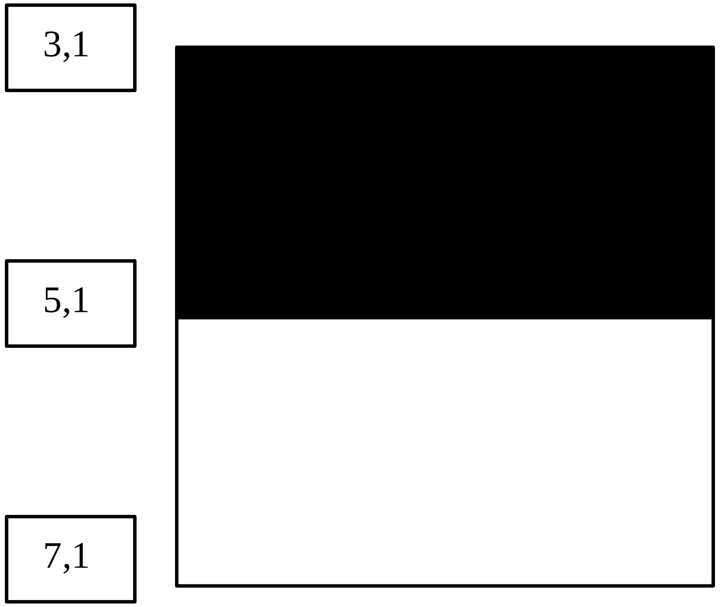
Using partial cascade to find false alarm examples

- We settled on 99% detection rate and 0.25% false positive rate.
- After further examination of the paper we decided to stop the current cascade and start new ones which use false alarms from the old ones. False alarm rate improved significantly

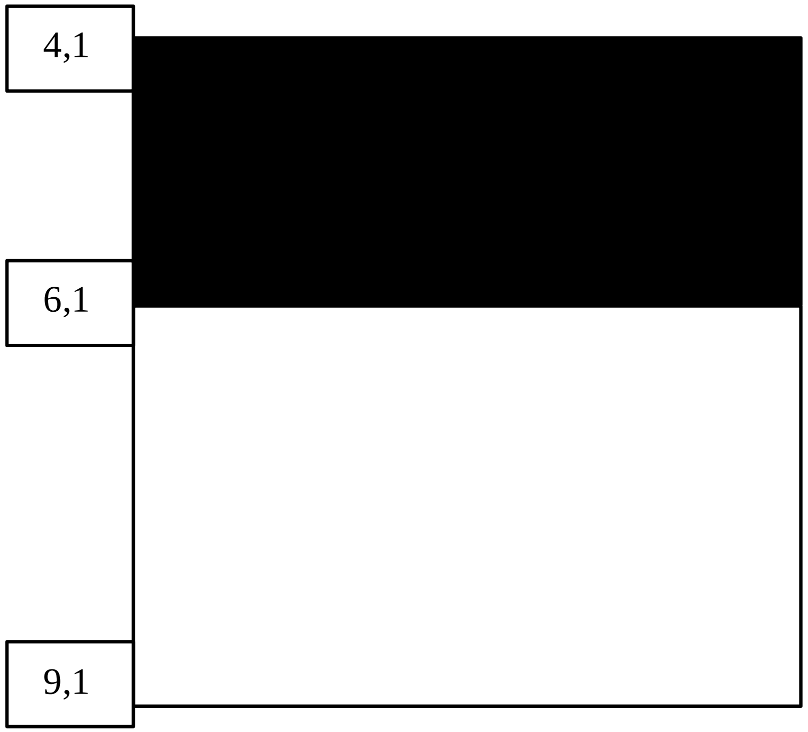
Rounding of Scaling Operation

- After 3 rounds of cascade and another round of negative set collecting from false alarms,
- And another cascade the false alarm rate was sufficient for the base scale of our detector (19X19), but bad for higher scales.
- We came up with the idea the problem lies with the way we scale up the features, particularly with rounding of small rectangle features (rounding effects small number much more proportionally than for larger numbers). As a result rectangles were not congruent.
- We solved this problem by scaling up the size of rectangle sides instead of scaling up integral image indices.
- For example, the following feature's rectangles will not be congruent after scaling up:

Feature in original scale (for 19X19 sub-windows)



Feature after Scaling in 1.25 (for 24X24 sub-windows) and a rounding operation



Memory limitations

- Using a 32-bit machine (and Matlab), We faced serious memory limitation for vector operations on large arrays. In Order to avoid Matlab Memory limitations and still maintain vectorized implementation, we created Chunks of the set of Feature Values of all Available Data
- In Training, FeatureValues data (All possible Feature and their Values for all Training Samples). was stored in chunks files.
- ~3000 faces
- ~10000-15000 Non Faces in each training phase
- 53130 features (all possible..)
- A 64bit Matlab version of + 10 GB of memory, could make "chunking" unnecessary.
- In Testing, we did not store the chunks in files, but processed each featureValuesXSubWin chunk, on at a time (about 200-500 MB per chunk), as we try the cascade fully on each SubWin.
- Testing performance remained good as before.

Downsizing

- Downsizing option is enabled for enhanced performance, default for downsizing is 384X288.

Post Processing

- We have implemented simple overlapping detection merging, of locating and eliminating isolated detections. Does good jobs on well detected images, but can delete true singular detection when the detection subwindows are singular, which can occur for noisy image, big StepSize(deltaX, deltaY),etc.. .

Further Work:

Merging overlapping detections

- In order to determine more precisely detection rate and false positive rate, it is crucial to be able to merge overlapping detections of multiple scales into a single detection window. At this stage, it seems as a non trivial task.

Reducing further discretization errors caused by scaling detectors

- For a single feature, we adjusted the scaling such that they will still preserve a Haar-like shape for each scaling. However, there are at least two more possible fine tuning adjustments:
- Compensating the deficiency of geometrical similarity to the original rectangles that builds the feature in its basic size.
- Compensating the feature change of <u>relative</u> origin within a scaled window.
- The above discretization errors have greater weight in the first few scaling stages, while as the scaling is bigger, it becomes more negligible. Future work on examinations and testing of appropriate linear interpolation methods could be considered.

Easy vector migration to SSE & openCV/IPP

- During the Matlab implementation phase, we worked on the task of migration to C++ (later we dimmed the migration unnecessary), including enabling work with mex & with visual studio compiler.
- As for mex-mechanism, we managed to transfer all types of data that we defined and used in our Matlab implementation.
- As for the C++ coding, further work can be done on how to use OpenCV and IPP data built in structures and API for “Boxing” operations, in order to preserve our code structure.