Overview

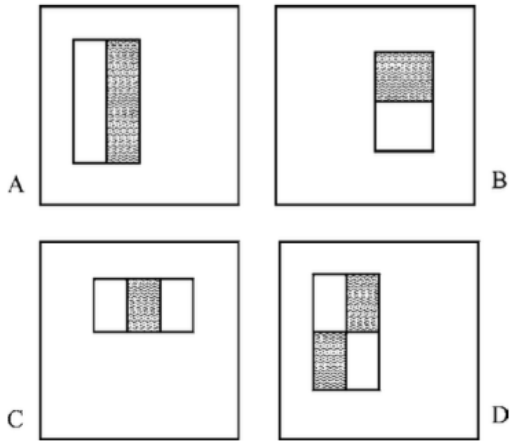
1. Viola-Jones Algorithm

The Viola-Jones algorithm is a robust real-time frontal face detection, competitive in situations where speed is critical. There are three key ideas in this algorithm.

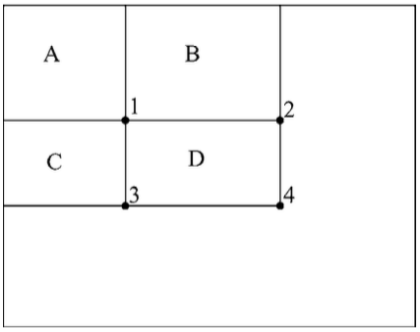
1.1 Feature and Integral Image

- **What is a Viola-Jones feature?**

  The Viola-Jones algorithm introduces the following new simple features.



A and B are two-rectangle features, C is a three-rectangle feature, and D is a 4-rectangle feature. Image taken from original Paper. The sum of the pixels in the unshaded rectangles are subtracted from the sum of the pixels in the shaded rectangles.



The sum of the pixels within rectangle D can be computed with four array references. The value of the integral image at location 1 is the sum of the pixels in rectangle A. The value at location 2 is A+B, at location 3 is A+C, and at location 4 is A+B+C+D. The sum within D can be computed as 4 + 1 – (2 + 3).

- **What is an integral image?**

  The integral image is defined by the following recursive relationship.

$$ii(-1, y) = 0$$

$$s(x, -1) = 0$$

$$s(x, y) = s(x, y - 1) + i(x, y)$$

$$ii(x, y) = ii(x - 1, y) + s(x, y)$$

  where,

  - $s(x, y)$ is the cumulative row sum at point (x, y),
  - $ii(x, y)$ is the integral image value at the same point,
  - $i(x, y)$ is the pixel value at that point.

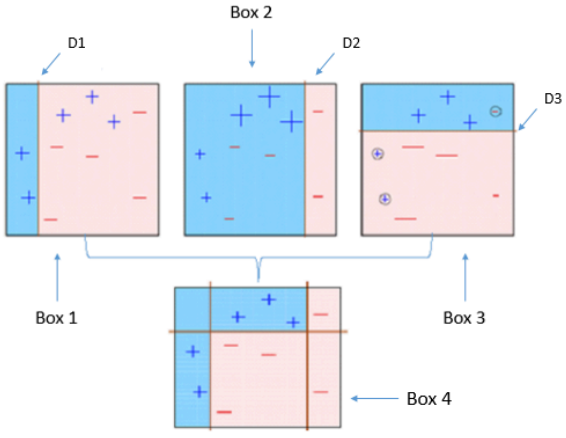
  What this relationship says is that the integral image at a point (x, y) is the sum of all pixels above and left of the current pixel.

- **Why needs the integral image?**

  Since the algorithm requires iterating over all features, even for small images, there are a lot of features, for example, over 160,000 for a 24 x 24 image, so they must be computed very efficiently. The integral image representation allows for very fast feature evaluation.

1.2 AdaBoost Learning Algorithm

For training, the Viola-Jones algorithm uses a variant of Adaboost. The general idea of boosting is for each subsequent weak classifier to correct the mistakes of the previous classifier.

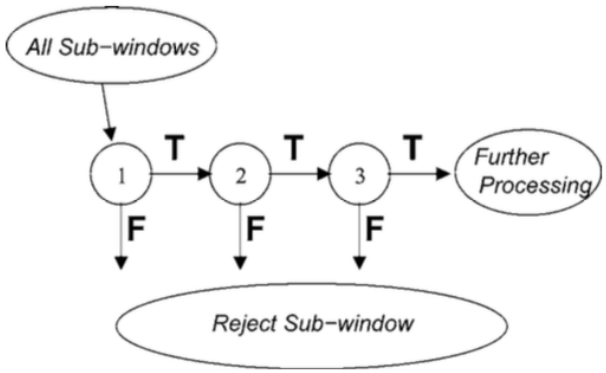


The basic idea of boosting is shown as above, it assigns a weight to each training example, trains the classifiers, chooses the best classifier, and updates the weights according to the error of the classifier. Incorrectly labeled examples will get larger weights so they are correctly classified by the next classifier chosen. This gives the following outline of the algorithm:

1. Initialize the weights
2. Normalize the weights
3. Select the best weak classifier (based on the weighted error)
4. Update the weights based on the chosen classifiers error
5. Repeat steps 2–4 T times where T is the desired number of weak classifiers

1.3 Attentional Cascade

Combining classifiers in a cascade which allows background regions of the image to be quickly discarded while spending more computation on promising face-like regions. A diagram of the attentional cascade borrowed from the original paper is shown below.



The structure of the cascade reflects the fact that within any single image an overwhelming majority of sub-windows are negative. As such, the cascade attempts to reject as many negatives as possible at the earliest stage possible.

1.4 Summary

In summary, Viola-Jones is an ensemble method which uses a series of weak classifiers to create a strong classifier. The output of the algorithm is a weighted combination of the predictions made by each weak classifier. Each weak classifier requires only a small number of parameters, and with a sufficient number of weak classifiers, it has a low rate of false positives. Though it has a very long training period, it classifies images quickly due to the use of cascaded classifiers.

2. Data Preparation

The data preparation includes lots of trivial yet vey important work, below is an overview of the general idea and information, details are to be discussed in the following pages.

- Code file.

```
prepare_data.py --mode pos
```

- Use FDDB dataset to generate positive samples.

```
if args.mode == "pos":
    generate_positive('FDDB-folds','originalPics','test')
```

- Use FDDB dataset as well as another non-face dataset to generate negtive samples.

```
elif args.mode == "neg":
    generate_negative_from_face_dataset('FDDB-folds', 'originalPics', 'test')
elif args.mode == "neg_nf":
    generate_negative_from_non_face_dataset('101_ObjectCategories', 'test')
```

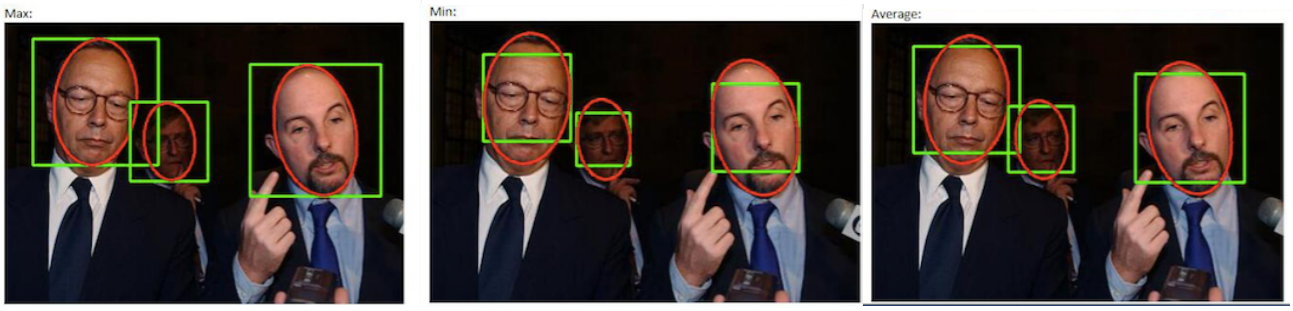
- The size of datasets.

	Positive Samples	Negative Samples	All Samples
Training Set	4214	5518	9732
Validation Set	100	100	200
Testing Set	400	400	800

2.1 Ellipse to Square

The FDDB uses ellipse to decribe the face, first of all, we need to change the ellipse to squares. There are three ways to perform this change:

1. The length of the square is the **longest** diameter of the ellipse, shown as the first image below.
2. The length of the square is the **shortest** diameter of the ellipse, shown as the second image below.
3. The length of the square is the **average** of above two, shown as the third image below.



And I have decided to use the third way to generate the square images, because the results are more similar to what Viola-Jones provides, below is the samples taken from the Viola-Jones paper.

Main code to transform from ellipse to squre. Once we have the coordinates of the square, it is easy to generate samples via *scipy.misc.imsave()*.

```
def get_rectangle_coordiates(width, height, major_axis_radius, minor_axis_radius, angle, center_x, center_y, square=True, square_mode="average"):
    tan_t = -(minor_axis_radius/major_axis_radius) * math.tan(angle)
    t = math.atan(tan_t)
    x1 = center_x + (major_axis_radius * math.cos(t) * math.cos(angle) - minor_axis_radius * math.sin(t) * math.sin(angle))
    x2 = center_x + (major_axis_radius * math.cos(t + math.pi) * math.cos(angle) - minor_axis_radius * math.sin(t + math.pi) * math.sin(angle))
    x_max = filter_coordinate(max(x1, x2), width)
    x_min = filter_coordinate(min(x1, x2), width)

    tan_t = (minor_axis_radius/major_axis_radius) * (1/(math.tan(angle) + 1e-8))
    t = math.atan(tan_t)
    y1 = center_y + (minor_axis_radius * math.sin(t) * math.cos(angle) + major_axis_radius * math.cos(t) * math.sin(angle))
    y2 = center_y + (minor_axis_radius * math.sin(t + math.pi) * math.cos(angle) + major_axis_radius * math.cos(t + math.pi) * math.sin(angle))
    y_max = filter_coordinate(max(y1, y2), height)
    y_min = filter_coordinate(min(y1, y2), height)
```

2.2 Positive Samples

The basic idea to generate positive samples is: obtain squared faces based on the information of ellipse face given by FDDB txt files.

Below is a glimpse of what the samples look like.

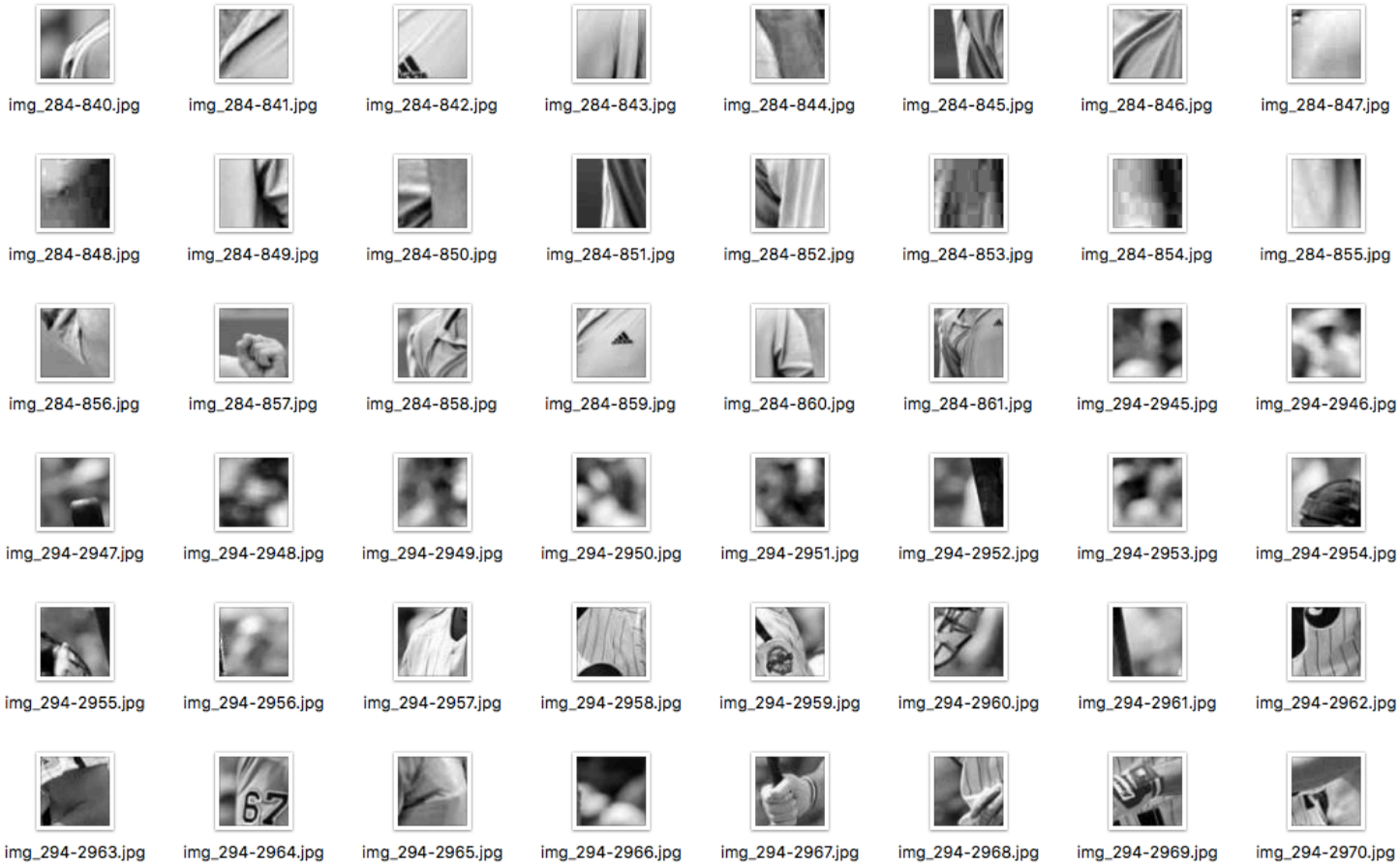
From the dataset above we can see that:

- Most of the faces are frontal and clear, pretty good for training.
- The faces inside red rectangles have two faces overlapped, keep them as well so as to train the model to be able to handle such situations.
- The faces inside blue rectangles are a little blurry, try to keep very few of them in the training set.
- The faces inside green rectangles are side faces, though Viola-Jones is aimed to mainly detect the frontal faces, keep them as well since the testing set from TA has side faces.

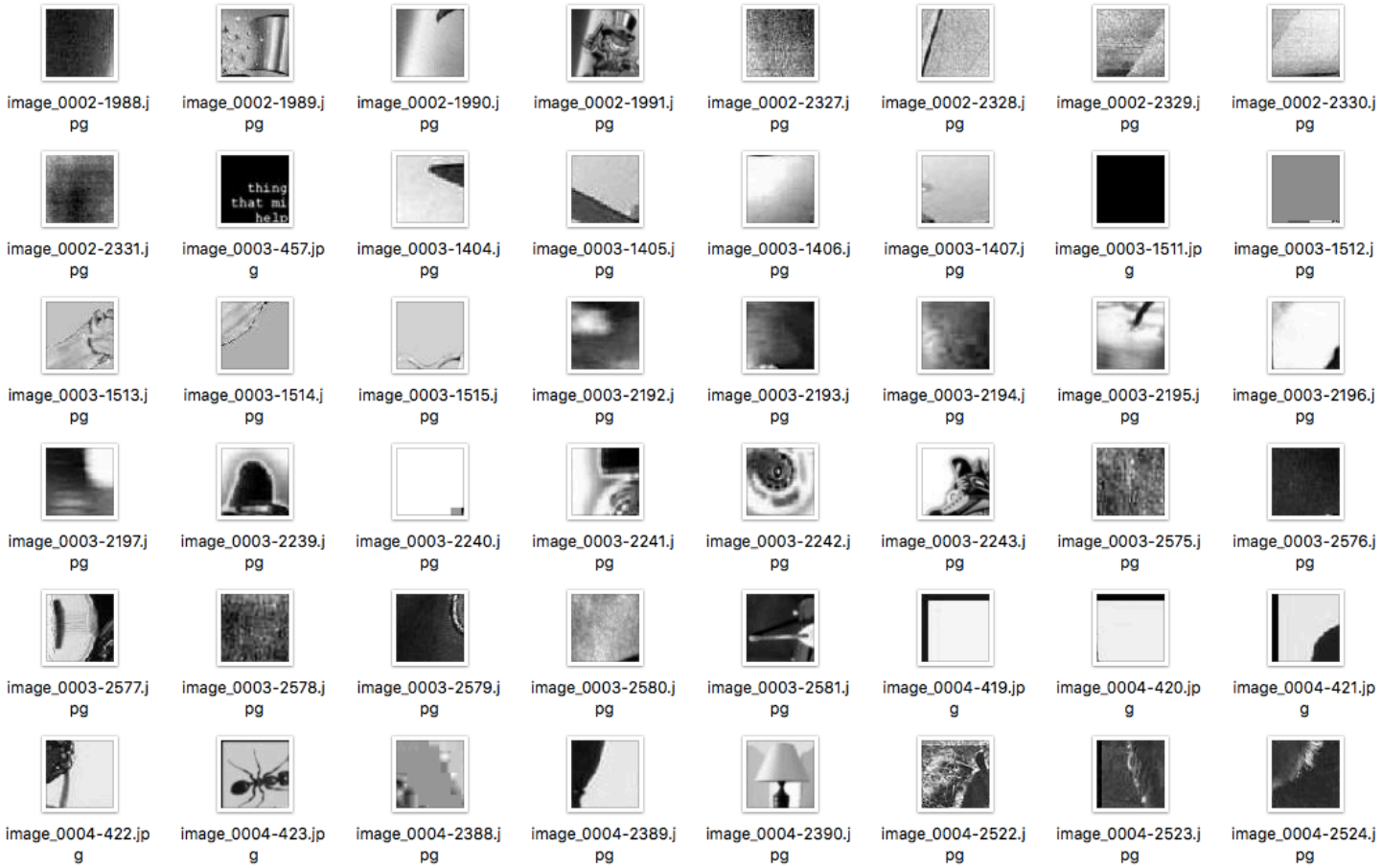
2.3 Negative Samples

The basic idea to generate positive samples is: based on the information of ellipse face given by FDDB txt files, generate samples that has little or no overlap with the faces, I have set the overlap ratio to 0.01 and the results are good to use.

```
def overlap_with_face(region, face_regions, overlap_threshold=0.01):
    region_area = region.width * region.height
    if region_area <= 0:
        rerun False
    for face_region in face_regions:
        intersectin_area = compute_intersection_area(region, face_region)
        overlap_ratio = float(intersectin_area) / region_area
        if overlap_ratio >= overlap_threshold:
            return True
    return False
```



Also use another dataset which has no faces to add more negative samples.



2.4 Check Datasets Manually

Though most of the samples generated automatically are good to use, there are a few samples that are not suitable to use. I manually checked all the samples and removed those unsuitable ones. For example,


img_3-4875.jpg


img_17-824.jpg

This left one has too much overlap, and the right one is too blurry and too small.

3. Train Model

3.1 Build Features

Before training, we need to build all the features at first. The algorithm below loops over all rectangles in the window and checks if it is possible to make a feature with it.

```
def build_features(height, width):
    features = []
    for w in range(1, width+1):
        for h in range(1, height+1):
            i = 0
            while (i + w) <= width:
                j = 0
                while (j + h) <= height:
                    # 2 rectangle features
                    immediate = RectangleRegion(i, j, w, h)
                    right = RectangleRegion(i+w, j, w, h)
                    if (i + 2 * w) <= width: # Horizontally Adjacent
                        features.append(ImageFeature([right], [immediate]))
                    bottom = RectangleRegion(i, j+h, w, h)
                    if (j + 2 * h) <= height: # Vertically Adjacent
                        features.append(ImageFeature([immediate], [bottom]))
                    right_2 = RectangleRegion(i+2*w, j, w, h)
                    # 3 rectangle features
                    if (i + 3 * w) <= width: # Horizontally Adjacent
                        features.append(ImageFeature([right], [right_2, immediate]))
                    bottom_2 = RectangleRegion(i, j+2*h, w, h)
                    if (j + 3 * h) <= height: # Vertically Adjacent
                        features.append(ImageFeature([bottom], [bottom_2, immediate]))
                    # 4 rectangle features
                    bottom_right = RectangleRegion(i+w, j+h, w, h)
                    if (i + 2 * w) <= width and (j + 2 * h) <= height:
                        features.append(ImageFeature([right, bottom], [immediate, bottom_right]))
                    j += 1
                i += 1
    return features
```

3.2 Train Weak Classifier

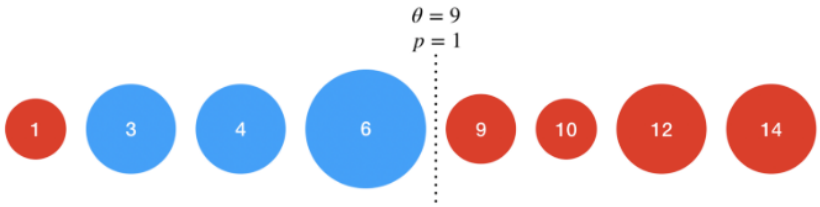
The Viola-Jones uses a series of weak classifiers and weights their results together to produce the final classification. Each weak classifier is weak because by itself, it cannot accurately fulfill the classification task. Each weak classifier looks at a single feature (f). It has both a threshold (θ) and a polarity (p) to determine the classification of a training example.

$$h(x, f, p, \theta) = \begin{cases} 1 & \text{if } pf(x) < p\theta \\ 0 & \text{otherwise} \end{cases}$$

Polarity can be either -1 or 1. When p = 1, the weak classifier outputs a positive result when f(x) < θ, or the feature value is less than the threshold. When p = -1, the weak classifier outputs a positive result when f(x) > θ. And the main code to train a weak classifier is as below.

```
def _train_weak_one_feature(input, y, weights, total_pos, total_neg):
    X_feature_sorted, X_feature_sorted_indices = input

    pos_weights, neg_weights = 0, 0
    min_error, best_threshold, best_polarity = float('inf'), None, None
    for f, idx in zip(X_feature_sorted, X_feature_sorted_indices):
        w = weights[idx]
        label = y[idx]
        error1 = neg_weights + (total_pos - pos_weights) # treat all (f_value < f) as positive
        error2 = pos_weights + (total_neg - neg_weights) # treat all (f_value < f) as negative
        error = min(error1, error2)
        if error < min_error:
            min_error = error
            best_threshold = f
            best_polarity = 1 if (error1 < error2) else -1
        if label == 1:
            pos_weights += w
        else:
            neg_weights += w
    return FeatureResult(min_error, best_threshold, best_polarity)
```

Training the weak classifiers is the most computationally expensive piece of the algorithm because each time a new weak classifier is selected as the best one, all of them have to be retrained since the training examples are weighted differently. However, there is an efficient way to find the optimal threshold and polarity for a single weak classifier using the weights. First, sort the weights according to the feature value that they correspond to. Now iterate through the array of weights, and compute the error if the threshold was chosen to be that feature. Below is an example to demonstrate this method.

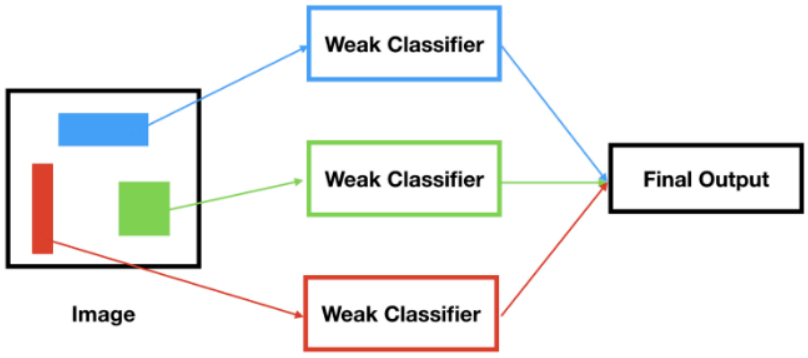


In this example, the numbers indicate the feature values and the size of the bubbles indicates their relative weights. Clearly, the error will be minimized when any feature with a value less than 9 is classified as blue. That corresponds to a threshold of 9 with a polarity of 1.

3.3 Compile Strong Classifier

Each individual classifier is weaker (less accurate, produces more false positives, etc) than the final classifier because it is taking in less information. When the results from each classifier are combined, however, they

produce a strong classifier.



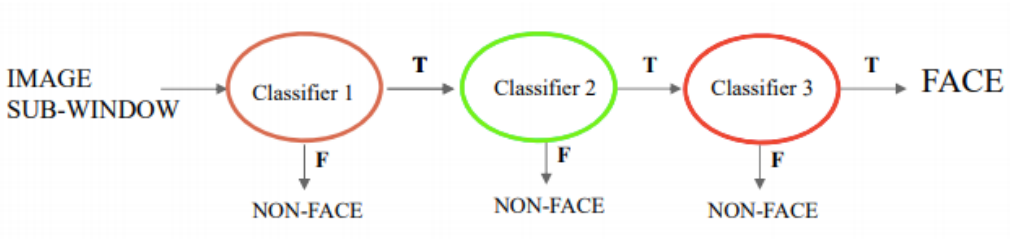
The strong classifier is defined as below.

$$\mathrm{C(x)} = \begin{cases} 1 & \text{if } \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t, \\ 0 & \text{otherwise} \end{cases}$$
$$\alpha_t = ln(\tfrac{1}{\beta_t})$$

The coefficient alpha is how much weight each weak classifier has in the final decision, and it is dependent on the error since it is the natural log of the inverse of beta.

3.4 Cascade

The attentional cascade uses a series of Viola-Jones classifiers, an image is only put through by the nth classifier if the n-1th classifier classifies it as a positive example. If at any point a classifier does not think the image is a positive example, the cascade stops.



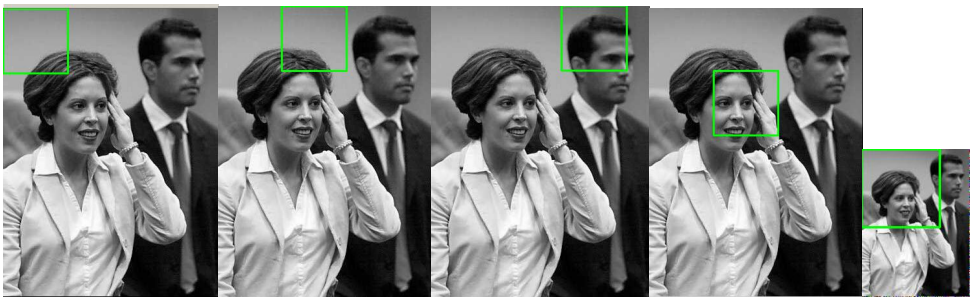
A negative outcome at any point leads to the immediate rejection of the sub-window.

4. Results and Analysis

4.1 Scan and Scale Image

When detecting a face, we need to scan the input image, visualize as below.

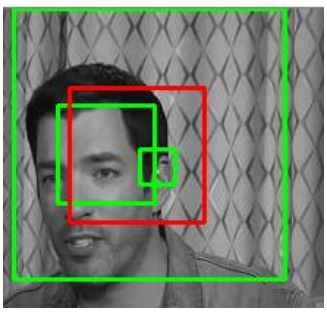
```
def scan_image(self, image_np, current_scale):
    h, w = image_np.shape
    detected_regions = []
    for y in range(0, h - self.window_size, self.step_size):
        for x in range(0, w - self.window_size, self.step_size):
            pred = self.classifier.classify(image_np[y:y+self.window_size, x:x+self.window_size])
            if pred >= 0.5:
                detected_regions.append(RectangleRegion(x / current_scale, y / current_scale, self.window_size / current_scale, self.window_size / current_scale))
    return detected_regions
```



Also, since the size of the input image is unknow, we need to scale the image, as shown in the fifth image, it is scaled down.

4.2 Postprocess Detections

As the paper indicates, in practice it often makes sense to return **one final detection per face**. Toward this end it is useful to postprocess the detected sub-windows in order to combine overlapping detections into a single detection. In these experiments detections are combined in a very simple fashion. The set of detections are first partitioned into disjoint subsets. Two detections are in the same subset if their bounding regions overlap. Each partition yields a single final detection. The corners of the final bounding region are the average of the corners of all detections in the set.



Above is an demonstration, the green squares are multiple overlapping detections, and the red square is the combined final one.

4.3 Training Accuracy

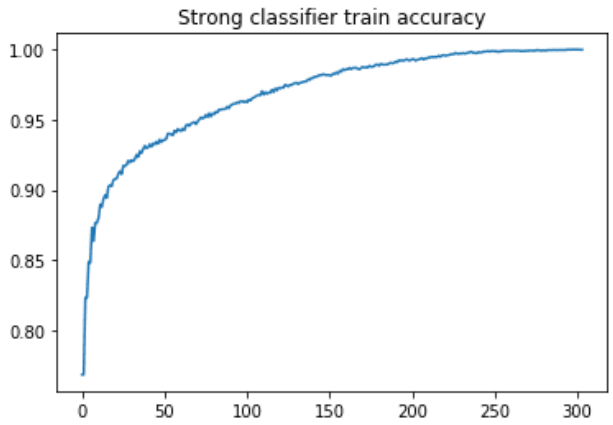
Training starts...

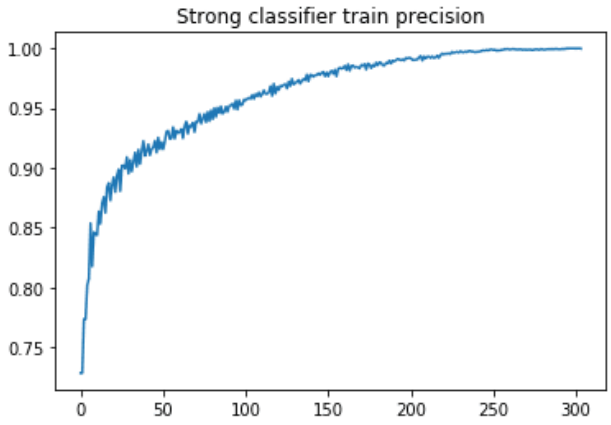
```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Start training 0-th weak classifier...
min_error: 0.23486637252063705, best threshold: -177.64624786376953, best polarity: 1
best feature: (grey_regions=[RectangleRegion(17, 5, 5, 2)], white_regions=[RectangleRegion(12, 5, 5, 2)])

Weak classifier train metrics:
accuracy: 0.7682901767365392, precision: 0.728268468888371, recall: 0.7415757000474609, f1: 0.7348618459729571, dr: 0.7415757000474609, fp_rate: 0.27173153111162895
Strong classifier train metrics:
accuracy: 0.7682901767365392, precision: 0.728268468888371, recall: 0.7415757000474609, f1: 0.7348618459729571, dr: 0.7415757000474609, fp_rate: 0.27173153111162895

Weak classifier valid metrics:
accuracy: 0.76, precision: 0.7549019607843137, recall: 0.77, f1: 0.7623762376237624, dr: 0.77, fp_rate: 0.2450980392156863
Strong classifier valid metrics:
accuracy: 0.76, precision: 0.7549019607843137, recall: 0.77, f1: 0.7623762376237624, dr: 0.77, fp_rate: 0.2450980392156863

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Start training 1-th weak classifier...
min_error: 0.2468972419019148, best threshold: -197.90749502182007, best polarity: 1
best feature: (grey_regions=[RectangleRegion(5, 11, 13, 2)], white_regions=[RectangleRegion(5, 13, 13, 2)])

Weak classifier train metrics:
accuracy: 0.7616111796136457, precision: 0.7207459207459207, recall: 0.7337446606549597, f1: 0.7271872060206961, dr: 0.7337446606549597, fp_rate: 0.2792540792540793
Strong classifier train metrics:
accuracy: 0.7682901767365392, precision: 0.728268468888371, recall: 0.7415757000474609, f1: 0.7348618459729571, dr: 0.7415757000474609, fp_rate: 0.27173153111162895

Weak classifier valid metrics:
accuracy: 0.785, precision: 0.7878787878787878, recall: 0.78, f1: 0.7839195979899497, dr: 0.78, fp_rate: 0.21212121212121215
Strong classifier valid metrics:
accuracy: 0.76, precision: 0.7549019607843137, recall: 0.77, f1: 0.7623762376237624, dr: 0.77, fp_rate: 0.2450980392156863

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Start training 2-th weak classifier...
min_error: 0.2967007989203754, best threshold: 349.1403696537018, best polarity: -1
best feature: (grey_regions=[RectangleRegion(4, 12, 16, 6)], white_regions=[RectangleRegion(4, 18, 16, 6)])

Weak classifier train metrics:
accuracy: 0.6906083025071927, precision: 0.6029789419619929, recall: 0.8357854769814903, f1: 0.7005469915464942, dr: 0.8357854769814903, fp_rate: 0.39702105803800714
Strong classifier train metrics:
accuracy: 0.8232634607480477, precision: 0.7734649122807018, recall: 0.8369719981015662, f1: 0.803966263961705, dr: 0.8369719981015662, fp_rate: 0.2265350877192982

Weak classifier valid metrics:
accuracy: 0.73, precision: 0.6796875, recall: 0.87, f1: 0.763157894736842, dr: 0.87, fp_rate: 0.3203125
Strong classifier valid metrics:
accuracy: 0.845, precision: 0.822429906542056, recall: 0.88, f1: 0.8502415458937198, dr: 0.88, fp_rate: 0.17757009345794394
```

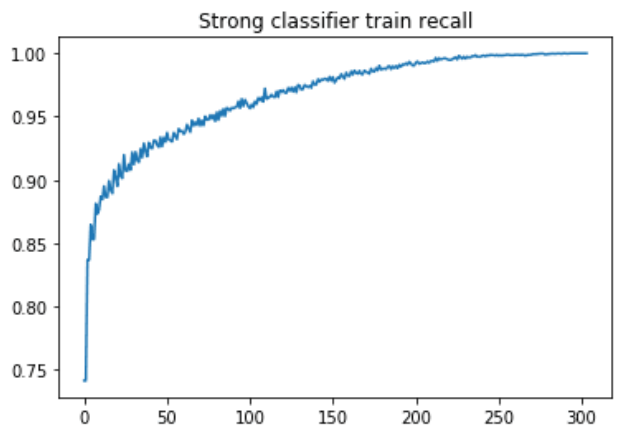
Training ends...

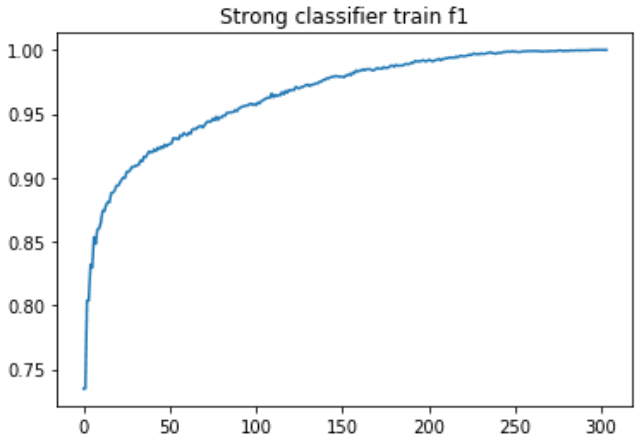
```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Start training 274-th weak classifier...
min_error: 0.43869070694915013, best threshold: -467.1151762008667, best polarity: 1
best feature: (grey_regions=[RectangleRegion(7, 12, 13, 2)], white_regions=[RectangleRegion(7, 14, 13, 2)])

Weak classifier train metrics:
accuracy: 0.6462186600904234, precision: 0.6965833758286588, recall: 0.3241575700047461, f1: 0.44242914979757086, dr: 0.3241575700047461, fp_rate: 0.3034166241713412
Strong classifier train metrics:
accuracy: 0.9991779695848746, precision: 0.9985775248933144, recall: 0.9995253915519696, f1: 0.9990512333965845, dr: 0.9995253915519696, fp_rate: 0.0014224751066855834

Weak classifier valid metrics:
accuracy: 0.6, precision: 0.7272727272727273, recall: 0.32, f1: 0.4444444444444444, dr: 0.32, fp_rate: 0.2727272727272727
Strong classifier valid metrics:
accuracy: 0.95, precision: 0.95, recall: 0.95, f1: 0.9500000000000001, dr: 0.95, fp_rate: 0.050000000000000044

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Start training 275-th weak classifier...
min_error: 0.4422430364233123, best threshold: -480.4933512210846, best polarity: -1
best feature: (grey_regions=[RectangleRegion(12, 9, 4, 5)], white_regions=[RectangleRegion(12, 14, 4, 5)])

Weak classifier train metrics:
accuracy: 0.43413481298808054, precision: 0.4170196380438968, recall: 0.7710014238253441, f1: 0.541274468971262, dr: 0.7710014238253441, fp_rate: 0.5829803619561031
Strong classifier train metrics:
accuracy: 0.9994862309905467, precision: 0.9990514583827366, recall: 0.9997626957759849, f1: 0.999406950539675, dr: 0.9997626957759849, fp_rate: 0.000948541617263432

Weak classifier valid metrics:
accuracy: 0.49, precision: 0.4939759036144578, recall: 0.82, f1: 0.6165413533834586, dr: 0.82, fp_rate: 0.5060240963855422
Strong classifier valid metrics:
accuracy: 0.95, precision: 0.95, recall: 0.95, f1: 0.9500000000000001, dr: 0.95, fp_rate: 0.050000000000000044

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Start training 276-th weak classifier...
min_error: 0.4380104909914768, best threshold: -21.969390869140625, best polarity: 1
best feature: (grey_regions=[RectangleRegion(9, 10, 3, 1)], white_regions=[RectangleRegion(9, 11, 3, 1)])

Weak classifier train metrics:
accuracy: 0.6308055898068229, precision: 0.6208641494745037, recall: 0.378500237304224, f1: 0.4702933805100988, dr: 0.378500237304224, fp_rate: 0.37913585052549625
Strong classifier train metrics:
accuracy: 0.9995889847924373, precision: 0.9992884250474383, recall: 0.9997626957759849, f1: 0.9995255041518387, dr: 0.9997626957759849, fp_rate: 0.0007115749525616888

Weak classifier valid metrics:
accuracy: 0.655, precision: 0.7627118644067796, recall: 0.45, f1: 0.5660377358490566, dr: 0.45, fp_rate: 0.23728813559322037
Strong classifier valid metrics:
accuracy: 0.95, precision: 0.95, recall: 0.95, f1: 0.9500000000000001, dr: 0.95, fp_rate: 0.050000000000000044
```

Below are the graphs for the training accuracy, precision, recall and f1.

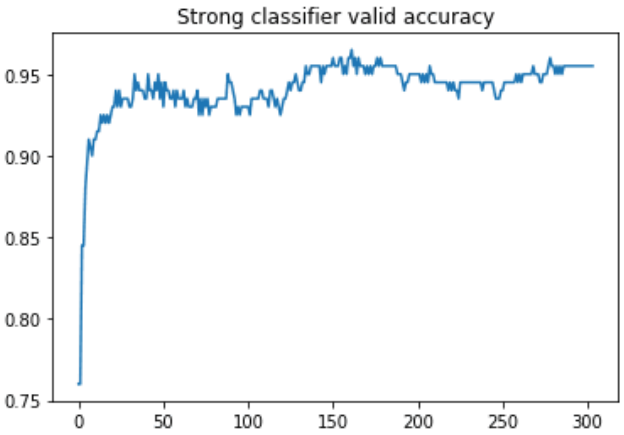


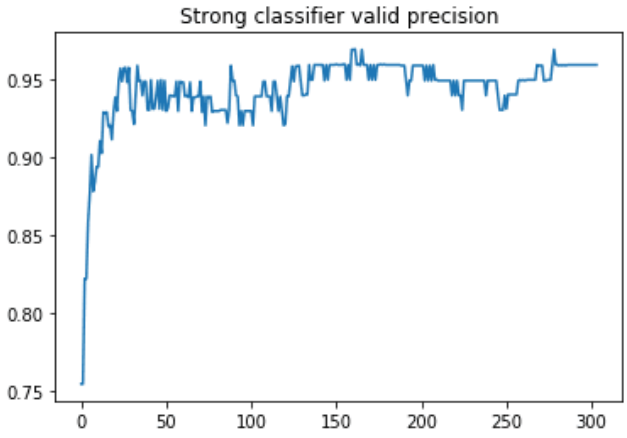


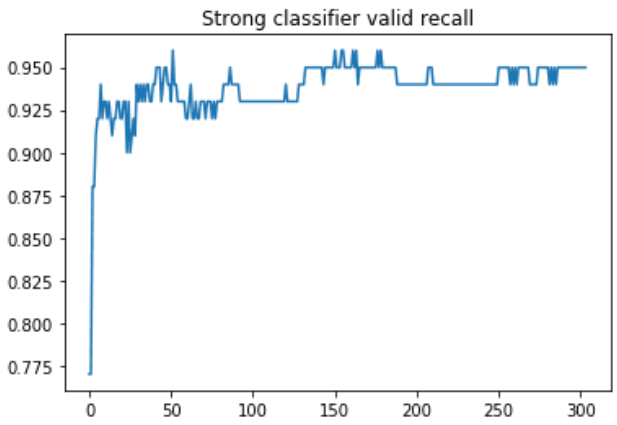


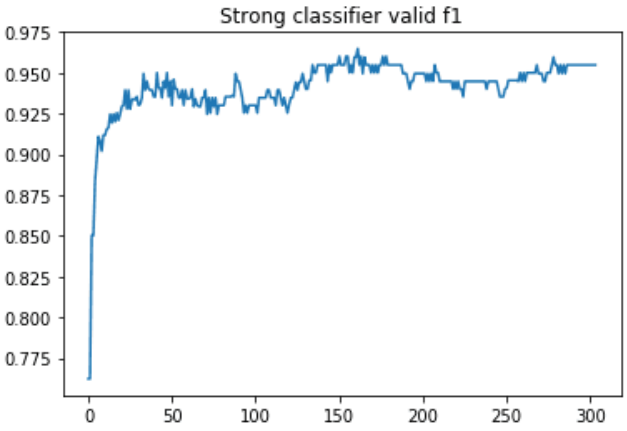


Below are the graphs for the validation accuracy, precision, recall and f1.









The model performs well on the training dataset and validation dataset.

4.4 Testing Accuracy on My Dataset

Run the model on the test dataset I have splitted earlier (repost the table below for your reference), and the result is pretty good.

	Positive Samples	Negative Samples	All Samples
Training Set	4214	5518	9732
Validation Set	100	100	200
Testing Set	400	400	800

- Accuracy: **0.95**625
- Precision: **0.95**28535980148883
- Recall: **0.96**
- F1: **0.95**64134495641344
- DR (detection rate): **0.96**
- FP_rate: **0.047**14640198511166

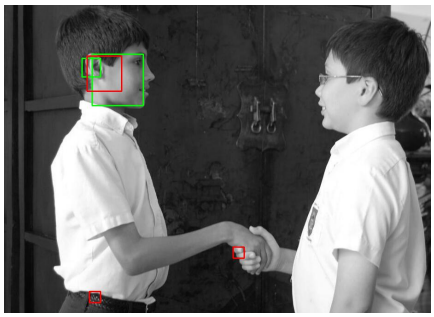
The result on my test dataset is good, however, as examples are not enough, probably my model is just overfitting.

4.5 Testing Accuracy on TA's Dataset

Well, high false positive occurs, wanna cry! Take a look at what my model has recognized.

As discussed in the section *4.2 Postprocess Detections*, the green squares are multiple overlapping detections, and the red square is the combined final one.

- Sample 1:

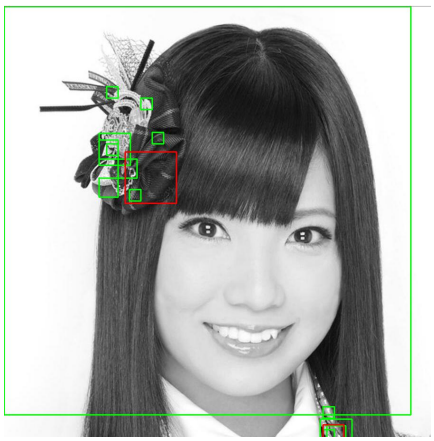


The model recognized one side face, but Viola-Jones aims to detect frontal faces, so this result is OK.

- Sample 2:

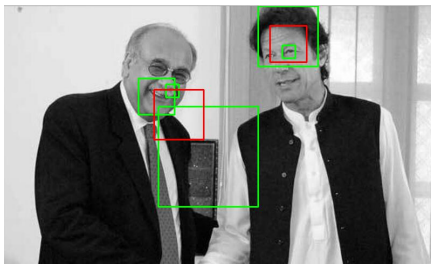
Finally, the model successfully detects the face!

- Sample 3:



The model actually detects the face, if you take a look at the largest green box. It's just there are so many false positives, so the average make the red box not on the face.

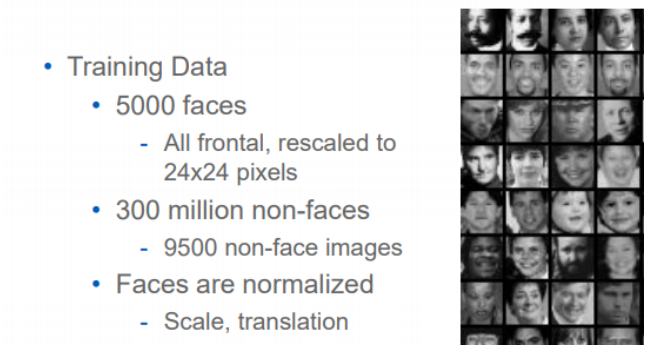
- Sample 4:



The model is close to find faces.

4.6 Analysis

The model performs very good on training dataset, validation dataset and my own test dataset. But the model has high false positive on the dataset that TA has provided, the main reason is that the training dataset, **especially the negative samples, are not enough**, which fails to train the model to learn enough useful information on judging the negative data. Based on the testing accuracy from my test dataset, the model seems overfitting.



Above image comes from the professor's slides, it reveals important information about the size of trainning dataset for an implemented system, compared to it, my training dataset is too small, especially for the negative samples. It uses **300 million non-faces** while I only used **5500**.

THE END, thank you for your time!