# PROJECT REPORT ON

Face Detection In The Wild

MOHAMMED ABDUL KAMRAN

------------------------

M37@BUFFALO.EDU

CVIP

Project 3

The project implementation is divided into 5 parts, as follows:

Part 1: Overview of Viola Jones face detection algorithm

Part 2: The first part consists of computing the integral image for fast feature evaluation

Part 3: This part deals with the creation of Haar features for detecting the faces

Part 4: Training the classifier

Part 5: Testing the trained classifier

The remainder of the report will give more details about each of the aforementioned parts.

# VIOLA JONES ALGORITHM FOR FACE DETECTTION

The Viola Jones face detection algorithm comprises of the following main features:

- Integral Images for rapid calculation of features
- Haar-like features for finding faces in the images
- Adaboost for feature selection
- Attentional cascade for early rejection of non-face windows

Details about each of the above features is given throughout the report.

Viola Jones algorithm uses 2,3 and 4 rectangle haar-like features as the human faces share some common similarities such as the region of eyes is darker than the region below it. The basic intuition is that only the parts where the feature is matched correctly will give a high intensity value when the darker region is subtracted from the brighter region.

Even a 24x24 image consists of 160,000+ features and the evaluation of intensities for each feature can take a very long time and therefore, integral images are introduced which require only 3 arithmetic operation to give the value of any region.

Since it is highly probable that initially, many photos will be misclassified by the classifier, a variant of adaboost is used. Adaboost is basically a type of boosted classifier where the wrongly classified images are given a higher weight so that they are correctly classified next time.

Since evaluating all the features for any test image is infeasible as any image is mostly comprised of negative regions (non-face) with only a few faces, it is better to eliminate non-face regions earlier on in the classification. Attentional cascade implements this idea by creating an ensemble of weak classifiers where each weak classifier deals with more and more features. Only the windows that pass through all the weak classifiers will be classified as a face and each weak classifier tries to correct the mistake of the previous classifier.

# PART 1:

Integral Images

## IN THIS SECTION:

- Why Integral Images?
- Approach for calculating Integral Images

## Why Integral Images?

In this projects, thousands of face and non-face images are used for training and testing and it does not take too long to realize that when there are hundred thousands of features involved, the calculation of the positive and negative region values gets too slow. Integral images help us in overcoming that difficulty.

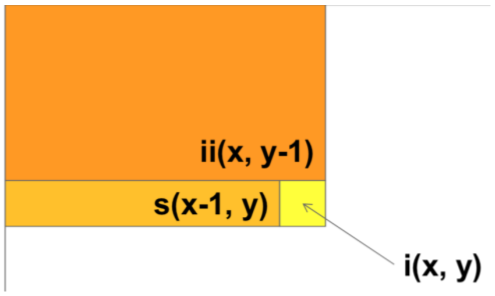
## Approach for calculating integral images:

To calculate the integral images, a simple formula was used as shown below:

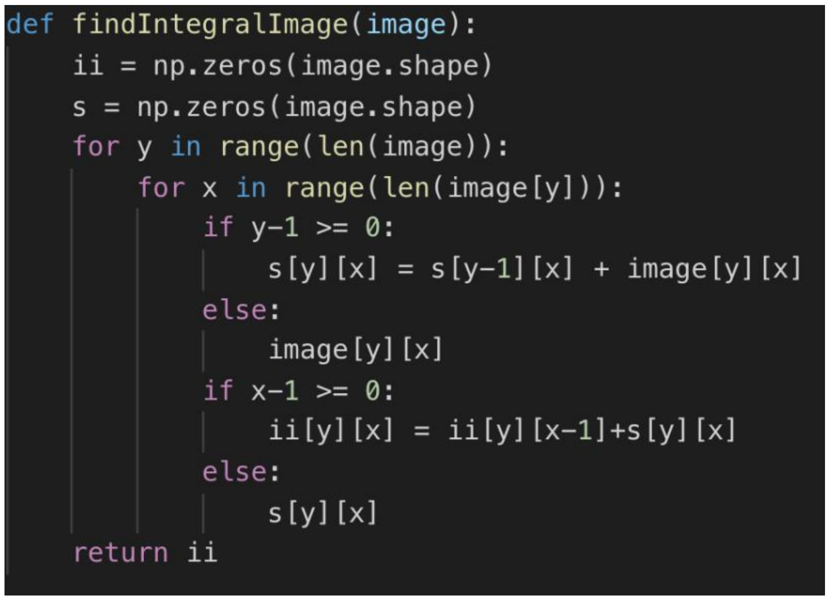
$s(x, y) = s(x-1, y) + i(x, y)$

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

Where s(x,y) is the cumulative sum of the row



The following piece of code is used to calculate the integral image, which also checks for a few corner cases where the value being calculated lies in the first row or first column.



```
def findIntegralImage(image):
    ii = np.zeros(image.shape)
    s = np.zeros(image.shape)
    for y in range(len(image)):
        for x in range(len(image[y])):
            if y-1 >= 0:
                s[y][x] = s[y-1][x] + image[y][x]
            else:
                image[y][x]
            if x-1 >= 0:
                ii[y][x] = ii[y][x-1]+s[y][x]
            else:
                s[y][x]
    return ii
```

## PART 2:

Haar Features

### IN THIS SECTION:

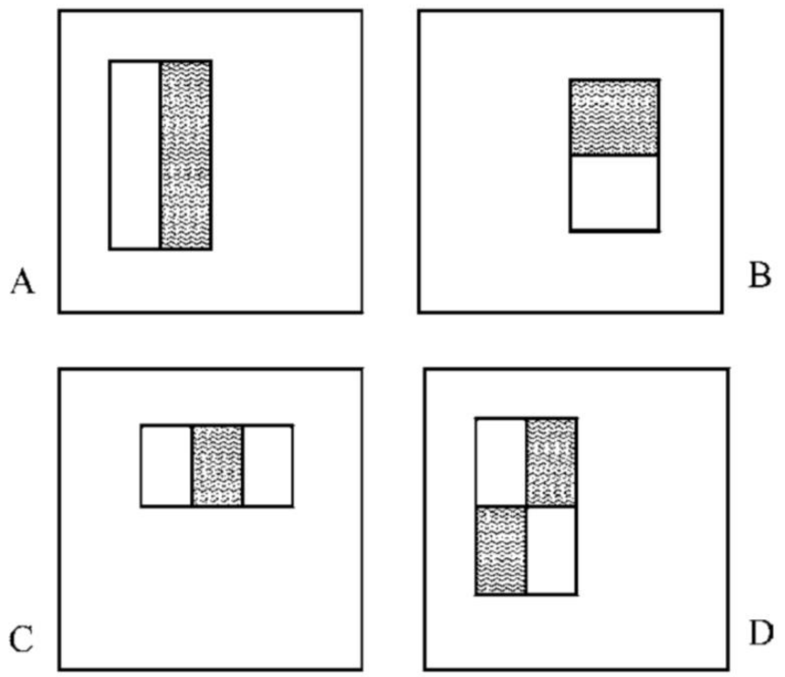
- What are Haar features?
- Few types of Haar features
- Creating Haar features

## What are Haar features?

A simple rectangular Haar-like feature can be defined as the difference of the sum of pixels of areas inside the rectangle, which can be at any position and scale within the original image.1 These type of features are extensively used for human face recognition as every face possesses certain characteristics which can be used to detect it.

## Types of Haar features:

In the Viola Jones face detection algorithm, there are 4 types of features that were used, as shown below. Although, for this project, only the A,B and C types of features were used.



## Creating Haar features:

A Feature class was created to store the coordinates and height, width of the feature. Initially every 2 rectangle feature starts with 1 pixel x 1 pixel (positive and negative region respectively) and is then incremented to the size of the training images (19x19 in this case).

[1] Haar-like feature: https://en.wikipedia.org/wiki/Haar-like_feature

The following piece of code shows the Feature class and the code for creating features:

```
class Features:

  def __init__(self,x, y, width, height):
    self.x = x
    self.y = y
    self.width = width
    self.height = height
def type1_ftrs(self, width, height):
    features = []
    for x in range(1,width):
      w = 0
      h = 0
      for y in range(1, height):
        if (x + w) * 2 < width:
          white_region = Features(x, y, x + w, y)
          black_region = Features(x+w, y, x + w, y)
          features.append(([white_region], [black_region]))
        if y + h < height and x + w < width:
          white_region = Features(x,y,w, y+h)
          black_region = Features(x+w, y,w, y+h)
          features.append(([white_region], [black_region]))
        w+=1
        h+=1
    return features

  def type2_ftrs(self, width, height):
    features = []
    for y in range(1, height):
      w = 0
      h = 0
      for x in range(1,width):
        if x + w < width and y + h < height:
          white_region = Features(x, y, x + w, y + h)
          black_region = Features(x, y + h, (x + w), y + h)
          features.append(([white_region], [black_region]))
```

```
            w+=1
            h+=1
    return features

def type3_ftrs(self, width, height):
    features = []
    for x in range(1,width):
        w = 0
        h = 0
        for y in range(1, height):
            if (x + w) * 2 < width:
                white_region = Features(x, y, x + w, y)
                black_region = Features(x+w, y, x + w, y)
                white_region2 = Features((x+w)*2, y, x+w, y)
                features.append(([white_region], [black_region, white_region2]))
            if (y + h) < height and (x+w)*2 < width:
                white_region = Features(x,y, x, y+h)
                black_region = Features(x+w, y, x, y+h)
                white_region2 = Features((x+w)*2, y, x, y+h)
                features.append(([white_region], [black_region, white_region2]))
            w+=1
            h+=1
    return features
```

## PART 3:

Training the classifier

### IN THIS SECTION:

- Overview
- First effort
- Attentional Cascade

## Overview of the training algorithm:

- Since there are no errors in the beginning based which the weights can be set, initially all the training examples are set equally such that:
    - For each positive example, the weight is give as

      W = 1/(2 * Number_of_positive_examples)

    - Similarly, for each negative example

      W = 1/(2 * Number_of_negative_examples)

For this, the following method is used which finds the integral image of each training example and also creates a list of initial weights for each of the images

```
def compute_integral(self, training, pos_num, neg_num):
    training_data = []
    weights = np.zeros(len(training))
    for x in range(len(training)):
        training_data.append((findIntegralImage(training[x][0]), training[x][1]))
        if training[x][1] == 1:
            weights[x] = 1.0 / pos_num
        else:
            weights[x] = 1.0 / neg_num
    return training_data, weights
```

- In each round of training,
    - Find the weak classifier that achieves the lowest weighted training error
    - Raise the weights of the training examples misclassified by the selected weak classifier.

This is done by the following piece of code:

```
def trainer_stub(self,X,y,features,weights):
    for i in range(self.T):
        weak_classifiers = self.train_weak(X, y, features, weights)
        clf, error, accuracy = self.optimal_classifier(weak_classifiers, weights, training_data)
        for i in range(len(accuracy)):
            weights[i] = weights[i] * ((error / (1.0 - error)) ** (1 - accuracy[i]))
        alpha = math.log(1.0/(error / (1.0 - error)))
        self.alphas.append(alpha)
        self.clfs.append(clf)
```

where, alpha is the value of the learned weight.

- Finally, the above step is performed T times where T is the number of weak classifiers, specified by us.

## First Effort:

- As a first effort, the training algorithm was written without the use of attentional cascades. T was specified as 10 and the resultant classifier was then run on a test image with background and 1 face.

- The result was fairly acceptable but the classifier took a very long time. It was evident that each of the classifier was run on every window, most of which did not even consist of a face.

- Therefore, attentional cascade was implemented for early rejection of non-face region.

## Attentional Cascade:

After realizing that the project was going in the right direction but only needed to speed up, the next step was to implement attentional cascade which increases the number of total classifiers without slowing down that algorithm, as the negative windows are eliminated in the early stages of the classification.

An attentional cascade with the following configuration was then created

```
class CascadeClassifier():
    def __init__(self):
        self.classifiers = [10,15,20,30]
        self.clfs = []
```

Each number in the classifier list corresponds to the number of weak classifiers in that round.

After the implementation of the cascade classifier, there was a significant boost in the detection process without any reduction in the accuracy.

Here, the classifier class looks like:

```
class WeakClassifier:
    def __init__(self, positive_regions, negative_regions, threshold, polarity):
        self.positive_regions = positive_regions
        self.negative_regions = negative_regions
        self.threshold = threshold
        self.polarity = polarity

    def classify(self, x):
        intensity = self.fval(x)
        if self.polarity * intensity < self.polarity * self.threshold:
            return 1
        else:
            return 0

    def fval(self,ii):
        a = sum([pos.compute_feature(ii) for pos in self.positive_regions])
        b = sum([neg.compute_feature(ii) for neg in self.negative_regions])
        return a - b
```

## PART 4:

Testing

## IN THIS SECTION:

- Testing the classifier using sliding window approach
- Non-maximum suppression

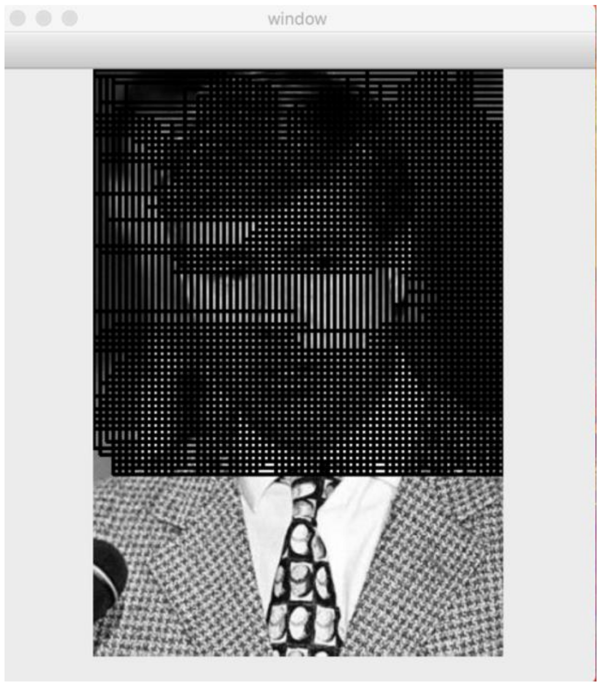
After the all the weak classifiers were trained to form an ensemble of classifiers, which together form a strong classifier, it was time to test it on an actual image.

For the purposes of test, a sliding window approach was used, wherein a window of fixed size is moved all over the image and each window is passed to the classifier to determine whether it is a possible face or not and if a window is classified as a face by all the weak classifiers in the ensemble, then the coordinates of that window are stored in the output file.

```
def faceOrNot(name, image, clf):
    h = w = 100
    stride = 10
    results = []
    json_list = []
    while h < image.shape[0] and w < image.shape[1]:
        for i in range(0, image.shape[0] - h, stride):
            for j in range(0, image.shape[1] - w, stride):
                b = cv2.resize(image[i:i+h, j:j+w], (19,19))
                op, _ = clf.classify(b)
                if op == 1:
                    b = (i,j,i+h,j+w)
                    results.append(b)
        h+=stride
        w+=stride
```

## First effort:

The initial try of detecting a face posed a new problem: there were a lot of overlapping regions which were rightfully classified as face as shown below:



There needed to be a way through which all the overlapping windows can be replaced by one correct window with face.

For this, a variant of non-maximum suppression[2] was used where an overlap threshold was specified such that all the windows whose normalized area was under the specified threshold were removed from the set of windows and only the unique windows were returned.

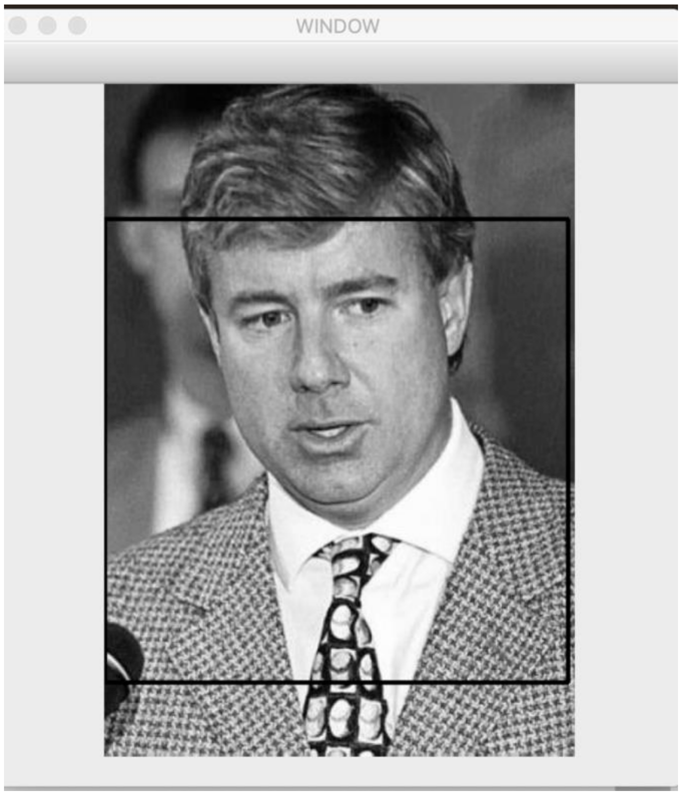
The following code with a threshold of 0.3 was used:

```
def non_max_suppression(boxes, overlapThresh):
    if len(boxes) == 0:
        return []
    if boxes.dtype.kind == "i":
        boxes = boxes.astype("float")
    pick = []
    x1 = boxes[:,0]
    y1 = boxes[:,1]
    x2 = boxes[:,2]
    y2 = boxes[:,3]
    area = (x2 - x1 + 1) * (y2 - y1 + 1)
    idxs = np.argsort(y2)
```

[2] http://www.computervisionblog.com/2011/08/blazing-fast-nmsm-from-exemplar-svm.html

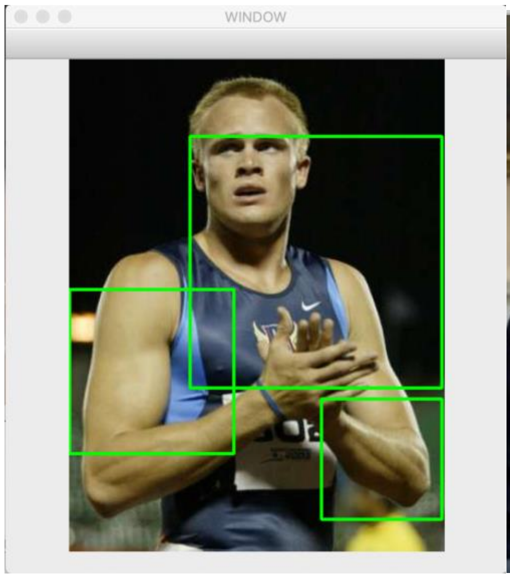
```
while len(idxs) > 0:
    last = len(idxs) - 1
    i = idxs[last]
    pick.append(i)
    xx1 = np.maximum(x1[i], x1[idxs[:last]])
    yy1 = np.maximum(y1[i], y1[idxs[:last]])
    xx2 = np.minimum(x2[i], x2[idxs[:last]])
    yy2 = np.minimum(y2[i], y2[idxs[:last]])
    w = np.maximum(0, xx2 - xx1 + 1)
    h = np.maximum(0, yy2 - yy1 + 1)
    overlap = (w * h) / area[idxs[:last]]
    idxs = np.delete(idxs, np.concatenate(([last],np.where(overlap > overlapThresh)[0])))

return boxes[pick].astype("int")
```

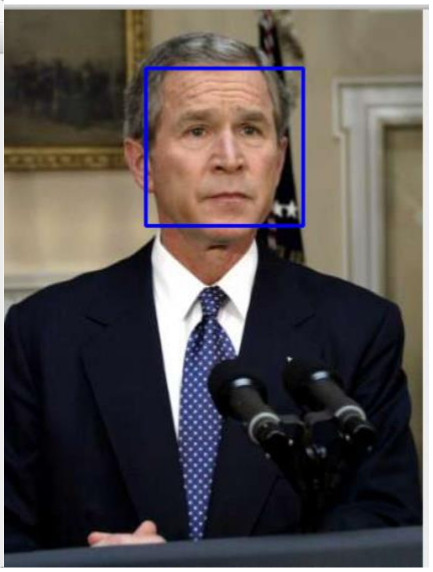
The result was the image shown below:

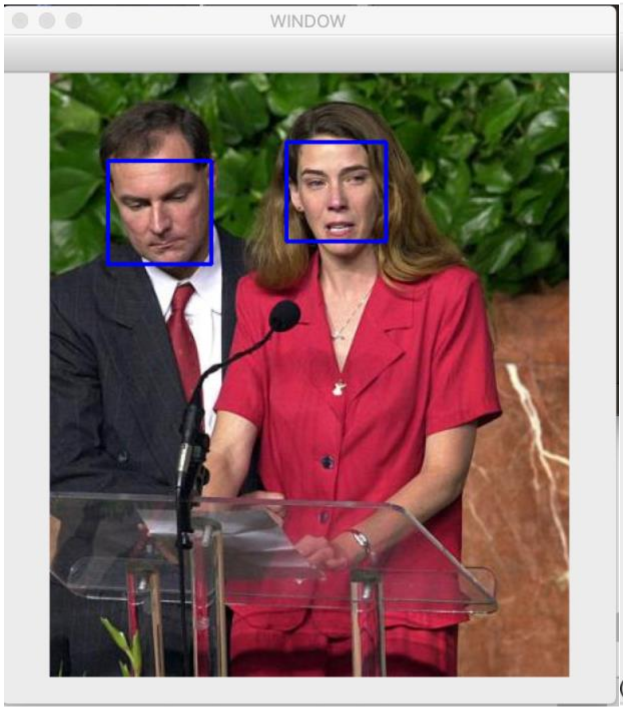


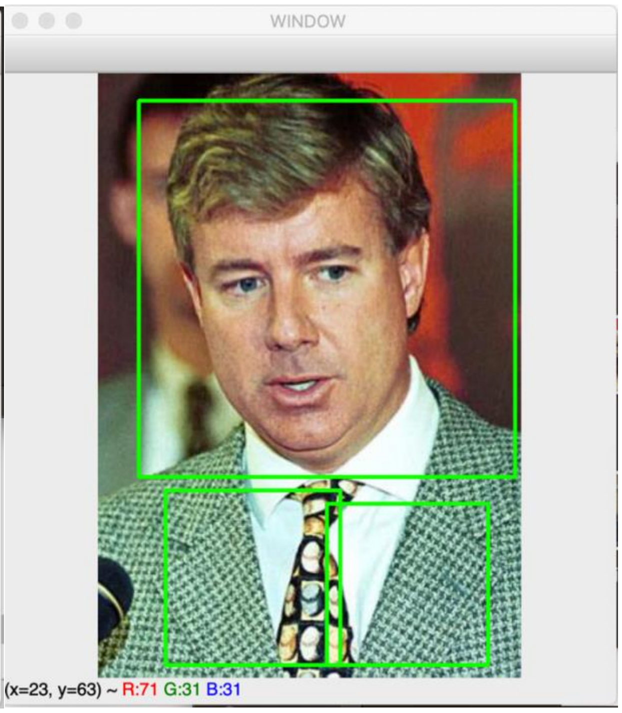
## RESULTS

The following are a few of the final results of the project:









## Analysis of result:

A very evident problem in the implementation of the algorithm was the occurrence of several false positives. This might be solved by one of the following:

- Increasing the size of training data
- Increasing the number of classifiers in the attentional cascade

The implementation fails when there are more than a few faces of different size in the image, or even when there is only one very large face in the image. For some reason, the implementation seems to fail for very large images.

The detection of faces in large images takes a very long time even after increasing the stride at which the window is slid.

## References:

- Understanding and Implementing Viola-Jones Image Classification Algorithm
- Viola Jones Object Detection Framework
- Rapid Object Detection using a Boosted Cascade of Simple Features
- Sliding Window for Object Detection
- Non maximum suppression for object detection