1 Introduction

In this document, we will attempt to create a face image classifier using adaptive boosting (AdaBoost) alongside Haar-like features. Images being read are in the JPG format, are 20×20 in size, and are converted to greyscale. We will be using the face database founds here - (http://vis-www.cs.umass.edu/fddb/).

The goal is to quickly recognize faces (potentially in real time) by using attentional cascading.

2 AdaBoost

AdaBoost is implemented so that a strong classifier is produced using a linear combination of weak classifiers (a single Haar feature). Each extracted Haar feature ($\Delta = \{(h_{(t)}(x_i), \alpha_t) : h_{(t)}(x_i) \in \{+1, -1\},\ \alpha_t \in \mathbb{R},\ t = 1, 2, \ldots T\}$, where T is the number of iterations of AdaBoost, and thus the number of **extracted features**) is trained on a set of faces & non-faces ($\{x_i : x_i \in \mathbb{R}^{20\times 20},\ i = 1, 2, \ldots, N\}$ where N is the number of images within the set) and evaluated using equation (10). Labels for the set of images are provided ($\{y_i : y_i \in \{+1, -1\},\ i = 1, 2, \ldots, N\}$) and used in the AdaBoost algorithm.

The way the algorithm works, is initially a set of **all possible** Haar features ($H_j,\ j = 1, 2, \ldots M$) are created given our image resolution. Each possible feature is evaluated for all N images using equation (10) $\forall\ i,\ j$. Each sample image receives a weight ($\{w : w \in \mathbb{R}^{1\times N},\ w(x_i) \in \mathbb{R}\}$) associated with it. After each iteration of the AdaBoost algorithm, weights are updated in a fashion which ensures underrepresented facial features get accounted for. All weights are initialized uniformly

$$w_{(1)}(x_i) = \frac{1}{N} \tag{1}$$

Then, an error term is calculated

$$\epsilon_{(t)}(h_{(t)}) = \frac{1}{N} \sum_{\forall (x_i, y_i)} w_{(t)}(x_i) \times 1_{h_{(t)}(x_i) \neq y_i}, \text{ where } 1_z = \begin{cases} 1, \text{if } z \text{ is true} \\ 0, \text{if } z \text{ is false} \end{cases} \tag{2}$$

where $\epsilon \in \mathbb{R}^{1\times M}$. This essentially says, the more incorrect each feature H_j, ($h_{j,(t)} = H_j$) is, the larger the error term will be. And if the feature mis-classifies a highly weighted image x_i, the larger the error term will be. This is done for all M features. The feature which holds the lowest error term will be selected and appended to Δ (the set of extracted features). This is done T times, so

$$\Delta_t = \underset{h_{(t)} \in H}{\arg\min}\ \epsilon_{(t)}(h_{(t)}) \tag{3}$$

and just to clarify notation, a subscript of t represents a position in a vector space while (t) represents an iteration in time. At the beginning, it was mentioned that the resulting strong classifier is a linear combination of our weak classifiers, so there is another 'weight' term involved (α) for each extracted features Δ_t

$$\alpha_t = \frac{1}{2} \log \frac{1}{\beta_{(t)}} \tag{4}$$

where log is the natural logarithm and

$$\beta_{(t)} = \frac{\epsilon_{j,(t)}(h_{(t)})}{1 - \epsilon_{j,(t)}(h_{(t)})} \tag{5}$$

Computationally, when $\epsilon_{j,(t)}$ is 0 (a perfect classifier $\in H$), calculation for α requires taking the log of infinity. Additionally, if $\epsilon_{j,(t)}$ is 1 (the worst classifier $\in H$), β will be infinity. Therefore, we add a very small value in calculating $\beta_{(t)}$

$$\beta_{(t)} = \frac{\epsilon_{j,(t)}(h_{(t)}) + \varepsilon}{1 - \epsilon_{j,(t)}(h_{(t)}) + \varepsilon} \tag{6}$$

The weights are then updated to account for underrepresented sample images, which were most incorrectly classified by the initial selected feature Δ_t

$$w_{(t+1)}(x_i) \leftarrow w_{(t)}(x_i) \times \beta_{(t)}^{1-1_{h_{j,(t)}(x_i)\neq y_i}}, \ \forall i \tag{7}$$

in the next iteration of AdaBoost, we simply normalize the weights

$$w_{(t)}(x_i) \leftarrow \frac{w_{(t)}(x_i)}{\sum_i^N w(x_i)}, \ \forall i \tag{8}$$

and repeat steps starting with (2). Our final strong classifier after all T extracted features will be

$$Y = sign\left(\sum_n^T \alpha_n \Delta_n\right) \tag{9}$$

where $Y \in \{+1, -1\}$

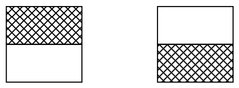
3 Haar features

In this document, the dark regions represent the negative aspects of the feature, while the light regions represent the positive aspects of the feature. Features are generated by inputting the resolution of an image area (in this case, 20×20), and all possible rectangles within the window are tested to see whether a valid feature can fit within it. If so, the coordinates of the negative & positive regions alongside the area of the each selection are stored. These coordinates will be used in evaluating how well the region classifies an area on an integral image. Calculation for the classification is as follows

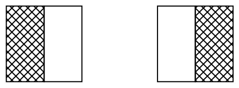
$$h_{(t)}(x_i) = sign\left(\frac{1}{Area_+}\left[\sum_{Region_+} pixels\right] - \frac{1}{Area_-}\left[\sum_{Region_-} pixels\right]\right) \tag{10}$$

where $h \in \{+1, -1\}^{N \times M}$. All possible Haar features from the following selection are utilized in this project

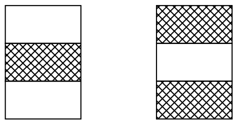
- Horizontal sections (2) - default & inverted



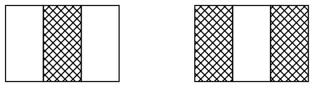
- Vertical sections (2) - default & inverted



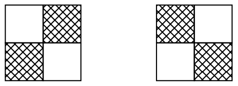
- Horizontal sections (3) - default & inverted



- Vertical sections (3) - default & inverted



- Rectangular grid (2) - default & inverted



4 Results

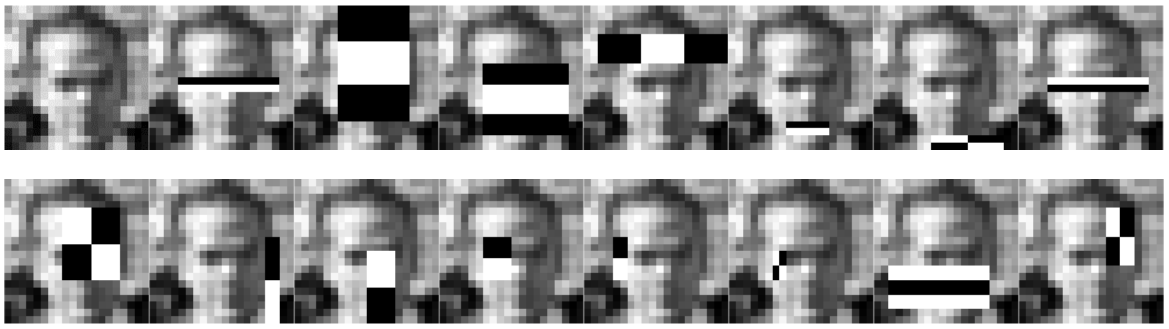


Figure 1: A sample face with the first 15 features with largest α values

Multiple models were created. The one seen in Figure 1 above was one trained using 2000 face and 2000 non-face images. There were a total of 250 extracted features, however not all are prevalent (will talk about this more later). What I found interesting in this model, was you can Adaboost selected the most prevalent feature (first) and the seventh feature to be exact opposites of one another. It seems to be accounting for the fact that the first feature has too much say in the classification, so it self corrected.

If we overlay the features shown above by adding and subtracting the positive and negative regions respectively weighted by these alpha values, we can get a sense of what our image classifier is attempting to do.

As you can see in Figures 2 and 3, the more features added increases the resolution of our model. However, both of these look like 'faces', which is a good sign that this is a promising model.

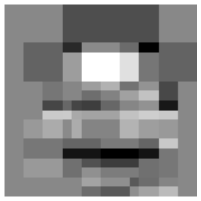


Figure 2: Overlayed features - first 15

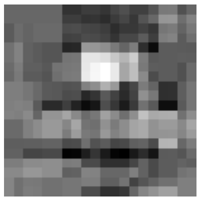


Figure 3: Overlayed features - all 250

Some other models with overlayed features

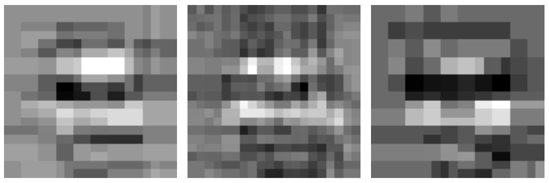


Figure 4: Overlayed features for 3 different models

The first is a model trained on 1000 face and 1000 non-face samples with 25 extracted features. The second is a model trained on a different set of 1000 face and 1000 non-face samples with 150 extracted features. The last is a model trained on 2000 face and 2000 non-face samples with 25 extracted features. It can be observed that they all resemble faces to some degree. For the remainder of this document, the first model with 250 extracted features will be used. However, it will be specified whether or not all 250 features are utilized.

4.1 Tuning & ROC

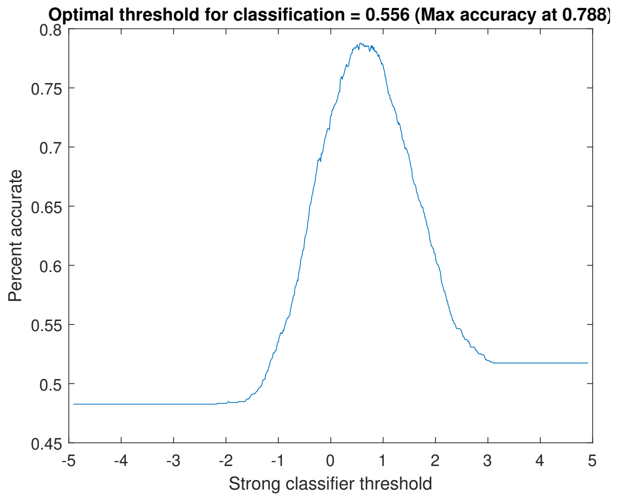


Figure 5: Optimal threshold for the 2000 faces 250 features model

While classifying each image, given by equation (9), the classification essentially centers around 0. However, that may not be the most optimal threshold. If we sweep thresholds δ from $\left(-\sum_t^T \alpha_t \rightarrow \sum_t^T \alpha_t\right)$, we are able to view all levels of classification and tune the threshold to maximize our accuracy. What Figure 5 tells us, is our tuned classifier will not be determined by equation (9), but rather by

$$Y = sign\left(\left[\sum_n^T \alpha_n \Delta_n\right] - \delta_{\text{optimal}}\right) \tag{11}$$

where $\delta_{\text{optimal}} = 0.556$, given by the threshold value which maximizes our accuracy of all test images. The ROC curves for the positives and negatives are plotted below, where the optimal threshold is labeled in red to minimize false rates and maximize true rates.

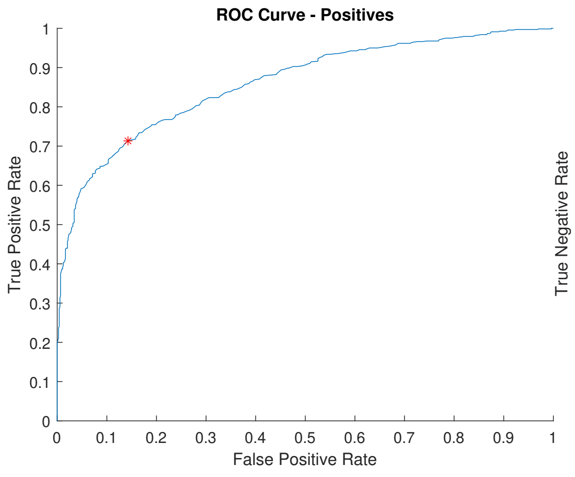


Figure 6: ROC for positives

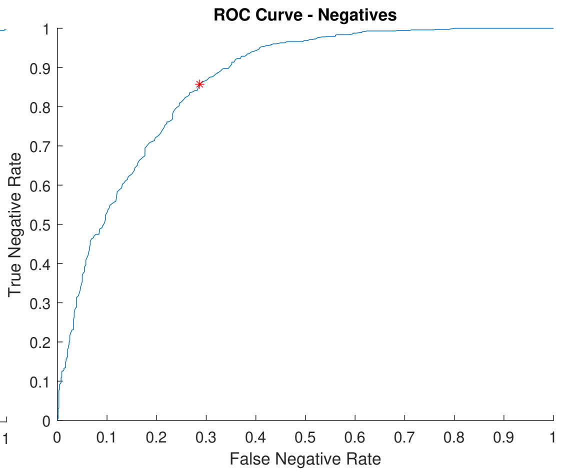


Figure 7: ROC for negatives

It can be observed that the maximum accuracy of this model (0.788, given in Figure 5) is between the true rates shown in these ROC curves (an average of ∼0.71 and ∼0.85)

4.2 Cascading

I tried to implement cascading, where it orders the features from most to least prominent. Then I choose a step-size s, where once the first set of s features are checked on an image, and if it passed then it moves onto the next s set of features until all T features are used. However, my model was too weak and it had major difficulty getting past the first couple of stages so it kept returning that every window within an image was not a face. I have the MATLAB code for it within the directory, which I will turn in.

4.3 Heatmaps

I decided to sweep a window across some images with different models to see how well they evaluate faces, like in the previous project. It can be noted that the optimal threshold value shown above might be optimal for the training & test set of data, but on real life images δ_{optimal} must be increased. Here are some results

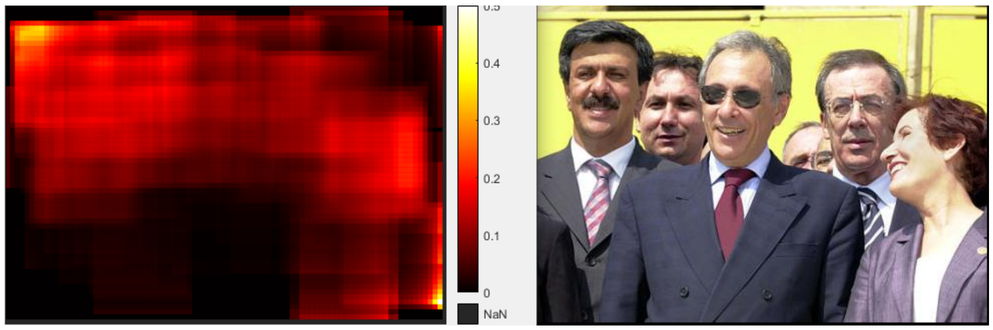


Figure 8: Image heatmap using 2000 face/250 feature model. Low threshold for classification

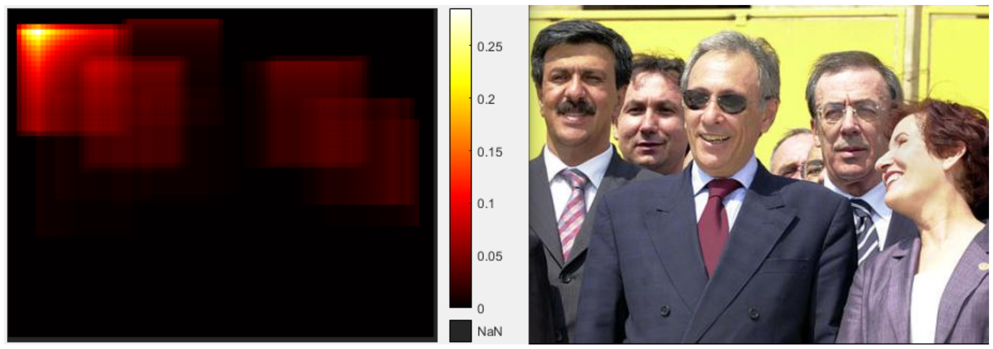


Figure 9: Image heatmap using 2000 face/250 feature model. High threshold for classification

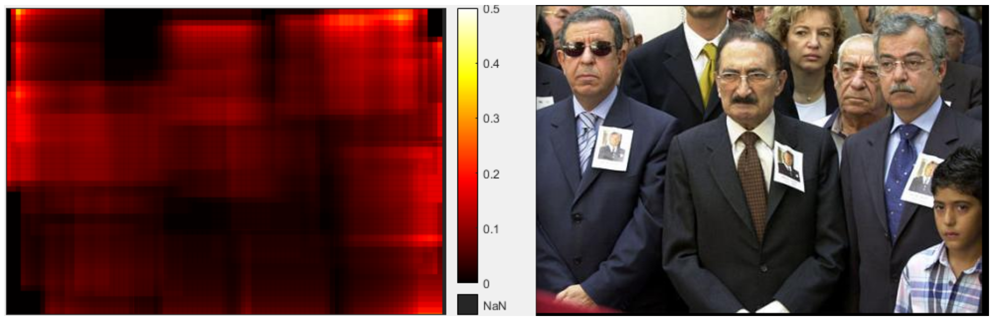


Figure 10: Same model, different image. High threshold

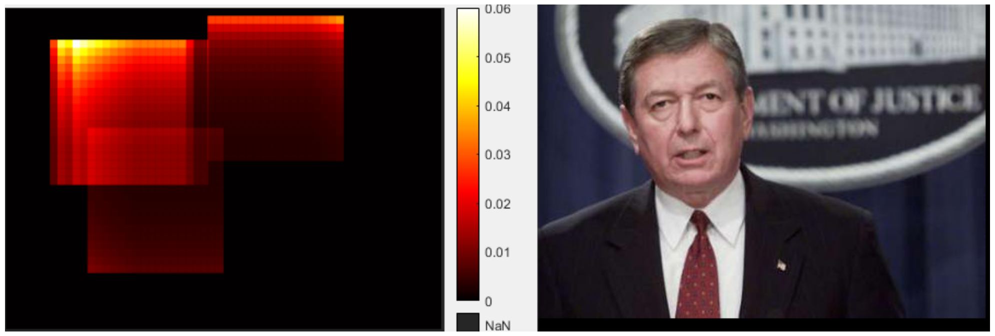


Figure 11: Same model, different image. High threshold

As it can be seen, the model is weak as there are still plenty of region it misclassifies. For the most part, the model works well. Note that in Figures 8, 9, and 10, there is a man wearing sunglasses in each. And notice how for the high threshold Figures, the face is not recognized by either! Despite there being strong classification in all other faces within the set. I found this to be very extraordinary and completely coincidental.

5 Conclusion

Compared to runtime for classification of the Gaussian mixture models from the last project, these Haar features speed up image processing significantly. It is unfortunate that the first feature selected determines the outcome of the rest of the features, as sometimes the first feature it selected was not the best, yet its α value stayed constant. This might have to with the fact that the way I am generating faces, as some low quality faces are being sized to 20×20 which is significantly more grainy than a high quality face with distinct features. In future work, I would like to clean up the data set (and/or the data set generation), and I would also like to add more Haar features (as there are a couple more interesting ones I wanted to implement).

6 Code Refactoring (Project 1)

In Project 1 (as well as this project), I used MATLAB due to my personal ease of use. I went through the Python reference code provided by the professor and noticed a couple of things, which I went ahead changed. Some implementations made their way to this project:

1. `np.diagflat` would have been a useful function to know of. Working with the diagonal covariance matrice in the last project was by far the most difficult aspect of Project 1, where my values continuously collapsed or exploded. In addition, these matricies got extremely large to work with. This function helps take a vector of the diagonal covariances and returns a 2D array with the rest of the covariances being 0. I did NOT set my other covariances to 0 throughout Project 1; I simply increased the magnitude of my diagonal covariances while decreasing the magnitude of the rest.

2. The way FBBD is pulled in is interesting. There is a resize/reshape function which takes the elliptical area, converts it to a rectangle, and then "squeezes" that into a square. The way I was pulling in from the FBBD dataset was taking the major axis values and setting a square around the center of the image that way. This squeezing done by the reference code I feel is more effective, as most of the faces produced have facial features aligned as

the face takes up the entirety of the square window. Whereas the way I did it in Project 1 was take a a square window around the face, which had plenty of background noise in it, and resize to the proper dimensions. This is ineffective due to not only the background noise, but imagine if someone has a ”long” face versus if someone has a ”short” face. Then with my method, the two faces have a high chance of having misaligned features, whereas this ”squeezing/stretching” method would take that into account.

3. I don’t know Python too well when it comes to data science, but it seems like I should learn it more. Reason being, the object oriented aspect of making classes with properties is more appealing than what I have been doing in MATLAB. Though there are classes available in MATLAB, I simply wrote everything into functions and I step through the script as I see fit. This is modular, but not as modular or programmer friendly if I were to share the contents of the project with someone else.