ENEE408M Section 0101 Final Lab Report

Team 3

Team Members: Ryan Koepke, Arden Matikyan, Dante Huynh
Instructor: Shuvra S. Bhattacharyya

Submission Date:

TABLE OF CONTENTS:

EXECUTIVE SUMMARY

The EFDS project was created in order to identify images that contain faces or not. Within the experimentation of the EFDS project, it is important to look at a wide variety of trade-offs, such as processing speed, implementation metrics, detection rate (DR), and false-positive rate (FPR) to understand the benefits and drawbacks of working with this system. Within the EFDS project, it is required to conduct a training module and a testing module, where the former was used to train the EFDS to make calculations properly, and the latter where the EFDS was used to test the quality of precision on a certain number of images.

To achieve these objectives, it was necessary to understand the Viola-Jones algorithm, a powerful algorithm used in many face detection algorithms. Using the Viola-Jones algorithm, the framework for building the training module was established, and building the classifiers required for the EFDS project became significantly easier. From the construction of the classifiers, those same classifiers were used in order to build the entire EFDS project. In terms of considering the number of classifiers for the project, when testing out multiple configurations for the classifiers, it turned out that a majority of training took a significant amount of time, so for design considerations, the best configuration was SC_stages_3_boostRnds_1_featureIncr_50.bin, explained more throughout the lab report.

In terms of results, that configuration of strong classifiers had the best DR and FPR, while compiling and constructing the project within the least amount of time. By running multiple different design choices, it was apparent that the amount of time needed to construct the project was unrealistic. More information about the configuration of strong classifiers in addition to the results is within the Alternative Designs and Design Choices section of the lab report.

GOALS AND DESIGN OVERVIEW

The goal of this project is to implement the Viola-Jones algorithm with an enhanced AdaBoost algorithm to run on a Raspberry Pi platform. This system is designed to be lightweight and fast to implement on hardware with limited performance. The Viola-Jones algorithm is a real-time face-detection process that can quickly determine the features of an image. The project for the embedded face detector system (EFDS) was split into two phases: training and testing modules. To understand how the EFDS system works, it is important to understand the Viola-Jones algorithm and the steps required to construct the algorithm on a training set for implementation on the testing set.

1. Integral Image and Haar-like Feature Generation

The Viola-Jones Algorithm utilizes a process of choosing the best Haar-like features from grayscale faces. In the most basic sense, Haar-like features are specific regions within the grayscale image. During the training, we identify the features that have the highest likelihood of overlaying facial details. These particular regions help to achieve the highest probability that the algorithm correctly identifies a face and includes: the nose bridge, the eyes, and the mouth. In this project, we used a total of five different Haar-like features, a horizontal and vertical two-rectangle feature, a horizontal and vertical three-rectangle feature, and a four-rectangle feature. There are a wide variety of Haar-like features that can be used on face detection systems, yet we decided to go ahead with these five Haar-like features due to the simplicity and efficiency these Haar-like features show when determining facial regions in a grayscale image. The horizontal rectangle and vertical rectangle divide a rectangle into two parts (**Figure 1.a** & **1.b**), a three horizontal and vertical rectangle that splits a rectangle into three parts (**Figure 1.d** & **1.e**),

and lastly a four rectangle that divides a rectangle into a checkered pattern (**Figure 1.c**). **Figure 1** below shows the smallest possible Haar feature (composed of single pixels). Each one of these feature types expands to every possible size configuration and location within an image. Two examples of this are provided in **Figure 2**.

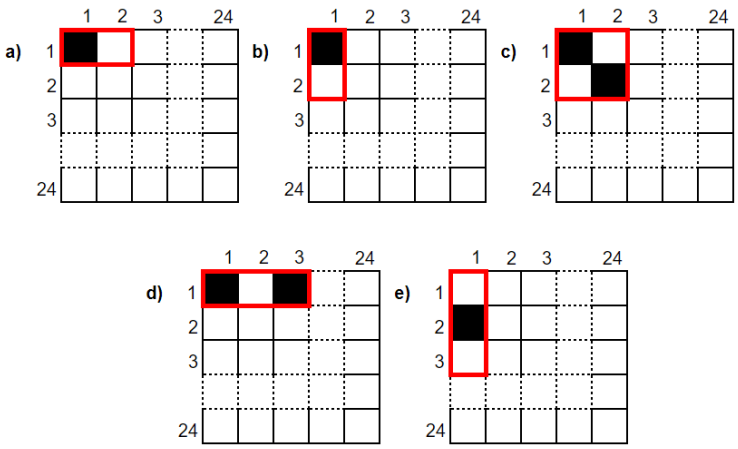


Figure 1: Five Haar-like feature types, **a).** Horizontal Two-Rectangle, **b).** Vertical Two-Rectangle, **c).** Four Square, **d).** Horizontal Three-Rectangle, **e).** Vertical Three-Rectangle

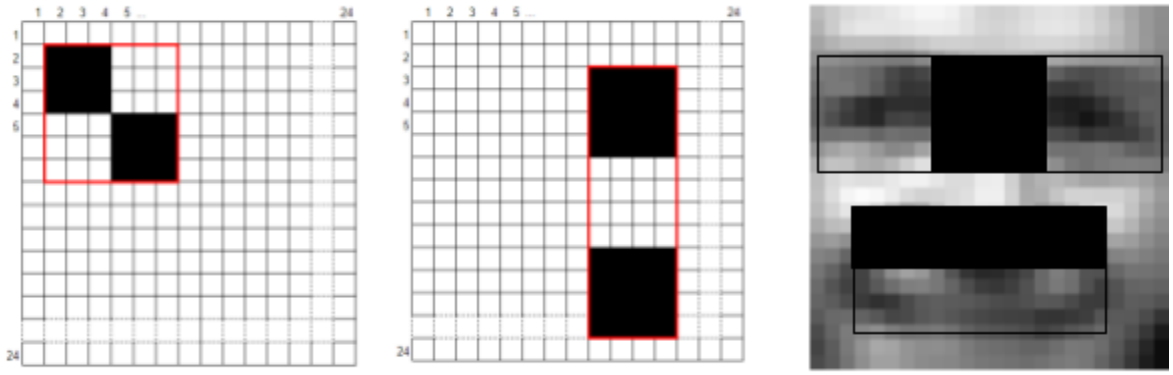


Figure 2: Four possible Haar-like feature configurations **a). and b).** and **c).** two of which are overlaid on a sample image.

The Haar-like features separate sectional areas of pixels of an image, where the two sections are distinguished as a dark and light region. The two sections are then used to calculate the sum of pixels within the light and dark region of the integral image, and the specific Haar-like feature on an image yields a value that is the difference between the sums of the two

areas. Calculating the sum of pixels within a particular region takes a considerable amount of time and memory, due to the high volume of features that exist in a 24x24 pixelated region (+160,000 total Haar-like features) and is highly dependent on the computer's firmware and hardware. In this project, the method of calculating pixels within certain regions is expedited with a concept called the integral image. The integral image is an adaptation of the original image where each pixel in the integral image contains the sum of all pixels above and to the left of it in the original image. An example of an integral image implementation is shown below in **Figure 3**.

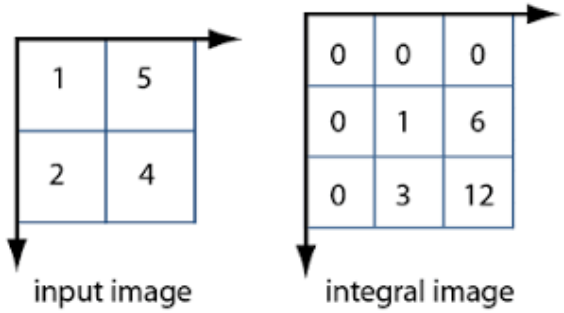


Figure 3. A pixelated input image is transformed into its integral image, sourced from [1].

To calculate the integral image, the image is placed within a 2D vector for processing. Integral image calculation is a single pass through the original image. Starting at the top left corner of the original image, each pixel value at (x,y) is added with the values directly above and to the left of the current pixel. This ensures that each pixel in the subsequent integral image represents the sum of all pixel values above and to the left. This process transforms the original image data into an integral image and stores it in a 2D vector for efficient Haar-like feature value calculation. The integral image allows for the sum of pixel values within any rectangle region in constant time regardless of the size of the rectangle. Integral images are used to efficiently calculate the contrast between various regions within an image according to these predefined Haar-like feature types. The Haar-like features' values are calculated in the featureUtil class for

each training image. Each Haar-like feature value is essentially used to determine the contrast that signifies edges or changes in the texture of an image. The best Haar-like features are what ultimately construct the weak classifiers used in determining if an image has a face.

2. Weak and Strong Classifiers/AdaBoost Algorithm

In the most basic sense, weak classifiers are simple classifiers that are trained to distinguish between a face and a non-face image region based on a singular Haar-like feature. The reason why they are "weak" is because they only perform slightly better than random chance; This is because the pattern they analyze is extremely basic and can be visible on plenty of non-face detail samples.

As each weak classifier has a one-to-one mapping to a Haar classifier, we send one integral image through all weak classifiers before selecting the group of weak. As every image is analyzed through the Haar feature, we collect the corresponding feature values that are normalized with weights and sort them in a list. An example is shown via **Figure 4** below:

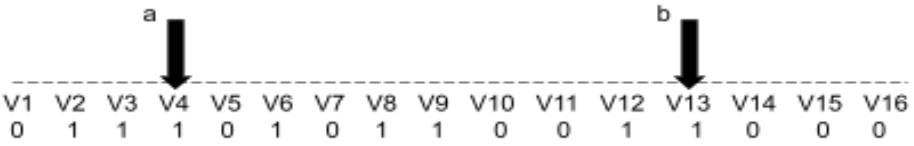


Figure 4: Example of calculating the threshold value of 16 images

Suppose our training is done on 16 images, where each value V1, …, V16 represents the weighted feature value of that image and 0 or 1 below represents a non-face or face sample, respectively. We look at all possible threshold values, in this example, we have "a" and "b". If we take "a" to be our threshold value, we can classify that samples V4 and to the left are 1 (or 0), and our accuracy would be 10/16. We do the same for "b" and we see the base accuracy is 11/16.

Thus, our threshold is the value of V13 which yields the highest accuracy among all possible threshold values. It's also important to store the polarity of the weak classification, signifying what side of the threshold value we classify as 0 or 1.

For each threshold, two types of errors are calculated to determine the effectiveness of that threshold in separating faces from non-faces. The positive error is the sum of the weights of all positive labels that would be incorrectly classified as negative labels if everything to the left of the threshold is classified as negative. This is combined with the weights of all the negative labels correctly classified to the right of the threshold. The converse occurs for the negative error calculation. However, not all weak classifiers are treated equally in the training module. If one weak classifier classifies face images significantly better than other weak classifiers, then the singular weak classifier should have a higher weight in the classification of the images. The calculation of these weights is done by a different algorithm called Adaptive Boosting, otherwise known as AdaBoost. An image exhibiting the steps of the AdaBoost algorithm is shown in **Figure 5** to the right.

- Given example images $(x_1, y_1), \ldots, (x_n, y_n)$ where $y_i = 0, 1$ for negative and positive examples respectively.
- Initialize weights $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$ for $y_i = 0, 1$ respectively, where m and l are the number of negatives and positives respectively.
- For $t = 1, \ldots, T$:
  1. Normalize the weights, $w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$
  2. Select the best weak classifier with respect to the weighted error
     $$\epsilon_t = \min_{f,p,\theta} \sum_i w_i \left| h(x_i, f, p, \theta) - y_i \right|.$$
     See Section 3.1 for a discussion of an efficient implementation.
  3. Define $h_t(x) = h(x, f_t, p_t, \theta_t)$ where f_t, p_t, and θ_t are the minimizers of ϵ_t.
  4. Update the weights:
     $$w_{t+1,i} = w_{t,i} \beta_t^{1-e_t}$$
     where $e_i = 0$ if example x_i is classified correctly, $e_i = 1$ otherwise, and $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$.
- The final strong classifier is:
$$C(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise} \end{cases}$$
where $\alpha_t = \log \frac{1}{\beta_t}$

Figure 5: A table showing the steps of the AdaBoost algorithm, sourced from [2] and [3]

The AdaBoost algorithm is used to transition the weak classifiers into subsequent strong classifiers. Fundamentally, strong classifiers are built of several weak classifiers, trained by the AdaBoost algorithm. The AdaBoost algorithm first initializes and normalizes weights for each of

the training images in the dataset. This makes them equally important at the beginning of the process. The weights of the images are updated based on the predicted performance of the selected classifier. The weights of the misclassified images are increased to ensure the subsequent boosting rounds pay more attention to difficult-to-classify images and the weights of correctly classified images are given lower weights. A “Boosting Round” indicated how many weak classifiers are cascaded within a single strong classifier. Each weak classifier is then given a classification error which is the weighted sum of misclassifications that the weak classifier exhibited on the set. The chosen weak classifier that has the lowest misclassification error (performed the best) in the classification process is then added to the strong classifier and a new boosting round will take place ([4]). Each boosting round adds a weak classifier to the strong classifiers to use in determining whether or not an image has a face. The number of weak classifiers added to a strong classifier is pre-determined by the users.

3. Training

After the boosting rounds are performed and the strong classifiers are determined they are evaluated with a set of testing images. Each image in this set will be marked with a label indicating whether it contains a face or a non-face. The cascade classifier will attempt to use the strong classifier to categorize the test image and compare its results with the image label. This process determines the true positive or false positive and false positive percent and how effective the strong classifier is. Each feature index is first checked to see if it has been used as a classifier in previous boosting rounds and if so, it is skipped. If the current feature has the lowest error it is selected to be the best weak classifier out of the total feature set. The weak classifier is then added to the strong classifier which contributes to its overall performance.

Training a strong classifier begins in the cascade classifier class. The cascade classifier is initially set with parameters such as the number of stages, boosting rounds, and feature increment. The number of stages determines how many strong classifiers will be added to the cascade. The entire training process is accomplished by first determining a set of Haar-like features and integral images from the image data. This is then passed through the AdaBoost algorithm to determine the weak classifiers and subsequent strong classifiers. The weak classifier is selected by iterating over all selected features in the feature set. For every feature the image set is sorted according to the feature value at that iteration. Then the minimum error is selected as the sum of positive or negative weights below the current image example as shown in **Figure 6**.

$selectedFeature = \emptyset$
$selectedFeatureError = \infty$
$selectedThreshold = 0$
$T^+ \leftarrow$ *total sum of positive examples weights* **(1)**
$T^- \leftarrow$ *total sum of negative examples weights* **(2)**
for *all* $f \in$ *features* ***do***
  $X^{(f)} \leftarrow$ *training images sorted by* f *value* **(3)**
  $e_f = \infty$
  $\theta_f = 0$
  for $i = 1$ ***to*** N
    $S_i^+ \leftarrow$ *the sum of positive weights below the current example* x_i^f **(4)**
    $S_i^- \leftarrow$ *the sum of negative weights below the current example* x_i^f **(5)**
    $e_i = \min\left(S_i^+ - (T^- + S_i^-), S_i^- - (T^+ - S_i^+)\right)$ **(6)**
    if $(e_i - e_f)$ ***then***
      $e_f = e_i$
      $\theta_f = f(x_i^f)$
    end if
  end for
  if $selectedFeatureError > e_f$ ***then***
    $selectedFeature = f$
    $selectedFeatureError = e_f$
    $selectedThreshold = \theta_f$
  end if
end for

Figure 6: Training a weak classifier, sourced from [4].

Each stage of the cascade is processed via the AdaBoost algorithm, which trains a series of weak classifiers into a single strong classifier. As subjects make their way through the cascade, only those images that have been classified as positive are sent to the successive strong classifier. This sequential training ensures that later strong classifiers have better performance than the earlier ones, progressively improving the overall accuracy and effectiveness of the model. Each strong classifier and its components are then saved and written to a configuration file for the EFDS testing stages. A graph showing the overall training procedure is shown in **Figure 7**, on the subsequent page.

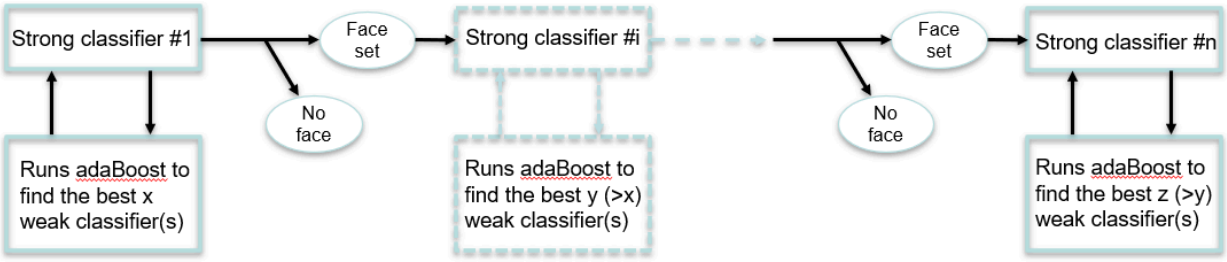


Figure 7: Graph depicting the overall training algorithm.

4. EFDS Welter Graph

The EFDS is accomplished using a Welter graph to optimize the processing and resource use of the Raspberry Pi. The Welter graph structure enhances the efficiency of the system by ensuring a smooth and orderly flow of data that might occur with other processing methods.

Figure 8: The EFDS is built using the Welter graph class.

In the "txt_img_read" actor, we read in a singular image as a text file and convert it to an integral image, assign it a key, package it as an Index Image Sub-Window (IIS) and send it to the strong classifier. Within the "classifier actor", we use the same strong classifier object type as the training module and initialize the strong classifiers using a configuration file carrying all the needed parameters. We then use the "predict" method of said classifier, and if it yields a positive classification we transfer the IIS to the next one. In the end, the "file_sink" actor compares the keys received to a pre-known list of positive and negative sample keys, computes metrics, and

writes to a file. Through the use of modes, FIFOs, and the “enable” function, we can handle when a subject is classified as negative, as it is not sent to the next classifier, and the actor is ready to read in the next IIS. The “enable” function allows us to check if there are any contents in the FIFO, if there are not, we do not invoke the actor, and it is ready for the next possible IIS.

Realistic Constraints

Despite the brevity and simplicity of the EFDS project, there are a few constraints that must be considered in the greater context of globalization. Some of these constraints entail communication restraints regarding the FIFO usage required by the design, specific functionality constraints, the restriction of different programming languages to construct the EFDS, testing requirements, and platform constraints. It is because of these constraints that it is significantly difficult to globalize the project for commercial usage and therefore easier to consider in an ideal scenario.

One major constraint executed by the design process are communication restraints required by the design. One aspect of the design process that is necessary for the completion of this project is the implementation of Welter functionality. In the EFDS project, it was necessary to utilize the graph and actor portions of the Welter code to communicate the data from different parts of the graph smoothly. The way these actors transfer data with each other is through the use of FIFOs, otherwise known as First-In, First-Out buffers. Some major constraints that occur when working with FIFOs are the quality of our data when transporting information from actor to actor and the volume of data a FIFO can transport at a single time. One major drawback in the initialization of FIFOs is that FIFOs can only receive a certain amount of data from an initial actor onto the FIFO. Once the maximum number of data is reached on the FIFO, it is impossible

to allocate more data to the FIFO unless the user fundamentally changes the size of the FIFO. In the grand scheme of things, this means that actors must follow a check to ensure there's enough data before firing, which ultimately restricts the construction of the Welter graph. At first glance, this may seem to be inconsequential, however, when debugging and monitoring data through the FIFOs, it is incredibly difficult to track data through the FIFOs in addition to debugging the entire graph. If any actor misfires, then the entire graph fails.

This leads to the next constraint being the Welter implementation through the "enable" and "invoke" functionality of the actors. The purpose of the "enable" functions of the Welter actor is to ensure that the correct amount of data is being inputted into the actor for usage. However, the amount of data can never go above or below the given amount of data being inputted for the actor. This means that the troubleshooting method for the Welter graph is significantly harder for testing multiple iterations of the EFDS system, which is extremely important for gaining the detection rate (DR) and the false positive rate (FPR) which are important parameters for determining the efficiency of our EFDS system.

One particular aspect of this project that was restrictive are the programming languages that were used in order to create the project which were C++ and Bash Scripts. C++ was used in order to create the framework of the project for the whole of the Viola-Jones algorithm and Bash Scripts was used to input functions that could not be done in C++ and in a faster time method as they are in the memory of the shell. However, in terms of efficiency for our project, many hardware and computers must download a C++ compiler to run the EFDS project outlined here. Most computers in today's day and age don't rely on C++, rather they use more modern languages such as Python and JavaScript to compute certain functions. Because of the sole reliance on an older programming language, it becomes difficult for new hardware and

programmers to use the EFDS project on their computers. This ties to platform constraints where the EFDS project can only be run on C++ compilers and compilers that have the Welter configurations downloaded onto the computer. However, most computers do not have the Welter configurations within the database, as the Welter database is confined to the University of Maryland, College Park students. As mentioned earlier in the paragraph, without proper hardware requirements for the EFDS project, it becomes nearly impossible for anyone to run the EFDS project efficiently and effectively.

Testing requirements also impose minimum constraints that must be met for the design to be validated. This means that the outcome of the EFDS project is essentially more important than the overall simplicity and efficiency of the design, as long as the criteria for meeting the testing requirements is met. However, this is very inefficient as an ineffective design and convoluted design will only hinder the processing time and use a larger memory space to compute the project. This detracts from the overall purpose of creating the EFDS project, where the EFDS must be able to classify faces independently and do more processing with that information. If the EFDS project takes too long in classifying, and cannot classify a wide variety of faces within its database, then the project is ineffective and unable to be used in general scenarios.

Engineering Standards

A key C++ feature utilized in our implementation is support for Object-Oriented Programming (OOP). OOP allowed us to grow our design effectively, for example, we used weak and strong classifier objects. We were then able to rapidly create any number of weak classifiers as needed, all with the same parameters and functionality, the same for the strong classifiers. The standard Welter graph implementation is used for our detection system section of

the project. The use of actors and FIFOS improves the productivity of the design process and allows us to integrate and experiment with flexibility and data flow.

Library standards such as separating our utils, source, and data helped us remain organized when working as a group and allowed useful bash scripting. We used bash scripting to facilitate the automation of system configuration and setup, such as installing dependencies, setting paths, and configuring parameters.

For dynamic data storage, the use of vectors gave us efficient access and manipulation of data elements. Vectors not only handle their memory allocation but also resize themselves as needed. Like arrays, they allow direct access to their elements for efficient reading and updating. However, they are memory efficient as they can allocate and deallocate memory contiguously to optimize memory caching. This enables the quick storage and retrieval of the large data sets required by the AdaBoost training algorithm.

Broader Considerations

In the evolving world of Artificial Intelligence, there arise many concerns and ethical considerations such as data privacy and security, algorithmic bias and fairness, and transparency and explainability. In the development of our Embedded Face Detector System, we are conscientious about public health, safety, and welfare. All of the training and testing data used is from an open-source database, where no images or faces are linked to an individual. Because of this, our system can't compromise an individual's privacy or health. The system functionality cannot pose a physical risk to users. In regards to welfare, since the implementation of the detection system is purely for educational purposes, and won’t be integrated into another system, it will only have positive outcomes. It’s also worth noting that the system does not retain any information about specific images, instead, it is built on finding patterns across a vast dataset.

There was little influence on our design process when it came to broader impacts such as environmental or economic impacts. The only possible concern would be time and data storage, we are restrained in our implementation due to time and storage constraints. During the configuration process, when improving the accuracy of our system we require larger memory space and a longer runtime. However the actual detection algorithm, after the configuration, does not have these limitations. As for the global, cultural, and social impacts, our design is mindful of such sensitivities. Our design is transparent in its functionality, in both its operation and use of input data. Our implementation is mindful of cultural norms and biases about features such as skin tone bias, glasses, facial hair, and facial expressions. The foremost has been a concern since the normalization of facial detection and sensor technology. We used a diverse training data set in hopes of accounting for these ranges but our accuracy is confined to the limits of our algorithm, computing power, and time.

Alternative Designs and Design Choices

This table encompasses the three different design choices during the time frame given to us. They all have a total of three strong classifiers (SCs), with a variable amount of boosting rounds and feature increments, tested over 2000 different images. More information regarding the initialization, the process of choosing these configurations, and explanations regarding the results are explained thoroughly in the Design Validation for Systems and Subsystems section. All of the data found from the configurations is shown within **Table 1** and **Figure 9** on the following page.

	Boost Rds: 10 Feature Inc: 50	Boost Rds: 1 Feature Inc: 50	Boost Rds: 1 Feature Inc: 5

True Positives (TP):	664	841	816
False Positives (FP):	0	1	0
True Negatives (TN):	1000	999	1000
False Negatives (FN):	336	159	184
Accuracy ($\frac{TP+TN}{\#\ of\ Samples}$):	83.2%	92%	90.8%
Avg. Computational Times (microseconds):	SC1: 115.423 SC2: 169.623 SC3: 318.592	SC1: 10.0755 SC2: 65.6789 SC3: 175.578	SC1: 10.1141 SC2: 60.4426 SC3: 170.684

Table 1: Data associated with three different configurations of the EFDS project

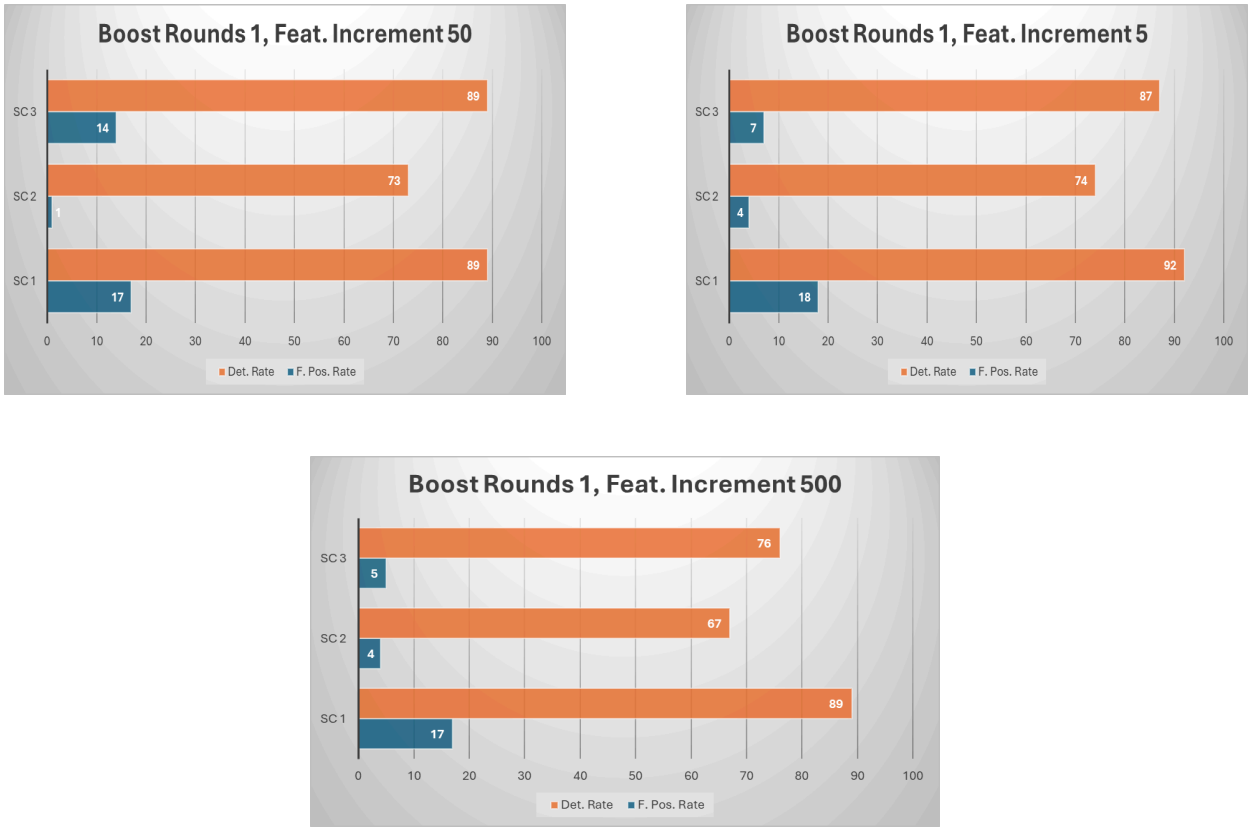


Figure 9: Three different implementations of the EFDS project with one boosting round and different feature increments, a). 5 features, b). 50 features, c). 500 features, shown in a bar graph format

Technical Analysis for Systems and Subsystems

One technical system we used was a Bash script to convert our sample images to text. The bash script runs the enee408mim2txt command on all .png files within the directory. This seems to be the most effective implementation because it is a very quick computation and eliminates the need for our EFDS to hold a massive amount of data as the .txt in grayscale files is much smaller than the .png. Secondly, we used the built-in file renaming tool on Windows to ensure all positive and negative samples have the same respective file name prefix, this made our txt_img_read actor much simpler to implement. With this there are some considerations, we must know how many positive and negative samples we are testing with our design, so we read in the files correctly. This consideration is also within our file_sink actor, it allows us to determine if the IISs that reach the file_sink actor were correctly classified or not. This allowed us to disregard our earlier design of using a fork actor, which would read from txt_img_read and write to the first classifier and the file_sink. We also eliminated our compare actor and did computations in the file_sink actor. Finally, we eliminated our file_source actors, which were used to read multiple configuration files. Instead, we read in a single configuration file and used our cascade classifier object type from the training module to create distinct strong classifiers. Each of these design decisions helped cut down on our development time, but more importantly, decreased the opportunity for bugs and complications to arise.

Design Validation for Systems and Subsystems

To achieve the best results for the strong classifier cascade prediction, different training evolutions were constructed. This was achieved by changing the number of boosting rounds,

feature increment, and number of cascade classifier stages. The goal was to minimize the amount of training time required while maintaining classification performance.

To have consistency across all the cases, only three stages were performed since the Welter graph was designed to only support three strong classifiers. This allowed the comparison of only the number of boosting rounds and feature increments. Also, the number of boosting rounds begins with an initial value and scales with the number of stages involved, following the formula, “5 * stage + 1”. The boosting rounds for training only started with one or ten for testing purposes. For instance, starting with a single boosting round, the sequence of boosting rounds would be 1, 6, and 16. Similarly, if the initial count was 10, the sequence progresses to 10, 15, and 25.

The feature increment was set at various levels (5, 50, 100, 500, and 1000) to evaluate the effectiveness of using a minimum set of features on the speed of testing. This was to determine the most efficient balance between the computational speed of training and the effectiveness of strong classifiers. The effectiveness of classifiers can be shown by comparing two designs, each starting with an initial 1 boosting round but differing in their feature increment. One of the classifiers was set at 5 and the other at 50, with varying results.

The configuration using five feature increments showed the best results in detection rate and false positive rate. The first strong classifier achieved a high detection rate of 92% and > 72% across the other two strong classifiers. This high detection rate came with a high false positive rate of 18%, suggesting that although it is great at detecting positive faces, it also incorrectly classifies a high number of negatives as positives. This will have the effect of detecting a higher volume of false alarms.

With a configuration of an initial 1 boosting round and 50 feature increments, the detection rate decreased from 89% to only 72% across all three classifiers and had the lowest false positive rate of 1% compared with using 5 feature increments while achieving similar false positive rates with 5 feature increments. This classifier showed a greater balance of detection rate to false positive rate and had the benefit of increased training computation time.

Overall, the training configuration of 1 boosting round and 50 feature increments has the best to offer with a balanced detection and false positive rate. However, for situations requiring a higher detection rate, an increased feature set might be required.

Across these tests, a heavy majority of the negative samples did not even make it to the second strong classifier and not a single sample made it to the third strong classifier. This is something we would have explored given more time, it could be possible to completely remove the third strong classifier without sacrificing accuracy.

Figure 10: Examples of positive samples that were misclassified.

Test Plan

The majority of our testing was an iterative process. To verify the functionality of our test suite we relied heavily on the diagnostics.txt of our tests, this contains the standard output of our EFDS during runtime. One of the controls we had was to limit the number of testing images, this allowed us to analyze all the steps that a file went through in the EFDS. With this, we were able to verify the functionality of the data flow of our EFDS. We were able to see every IIS travel from strong classifier to strong classifier, at which point the IIS was discarded as a negative

sample. Our test plan for the EFDS was to implement aspects one by one and check their functionality with the diagnostics. The first feature we incorporated was properly transferring data through the Welter graph. We then moved on to properly reading images in text format. Finally, we implemented using our strong classifiers predict methods on the samples. Because we used the same object types for strong and weak classifiers as our training module, we didn't have to test this functionality. Another control we implemented was certain images with the same values.

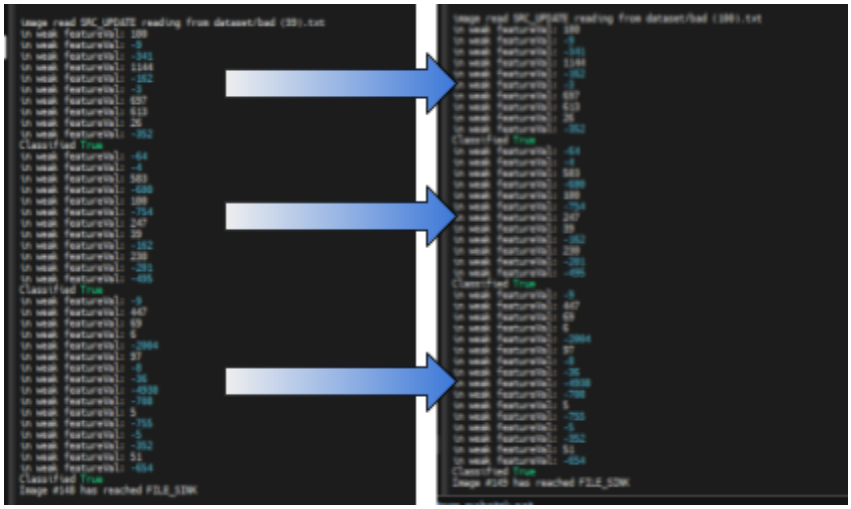


Figure 11: Iterative testing of false samples.

Here we have two samples with the same data attempting to be classified but forced through the entire Welter graph. It can be seen that all the weak classifiers obtain the same feature value for both the same images.

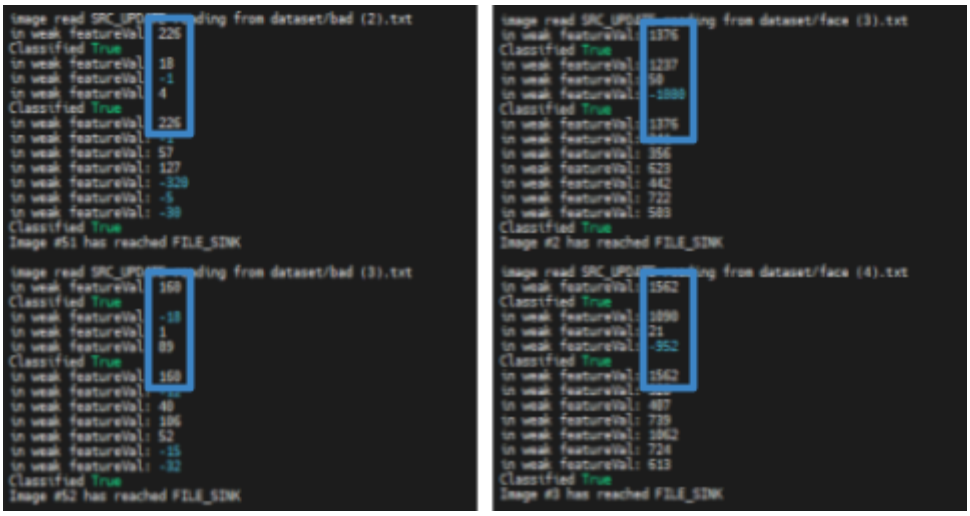
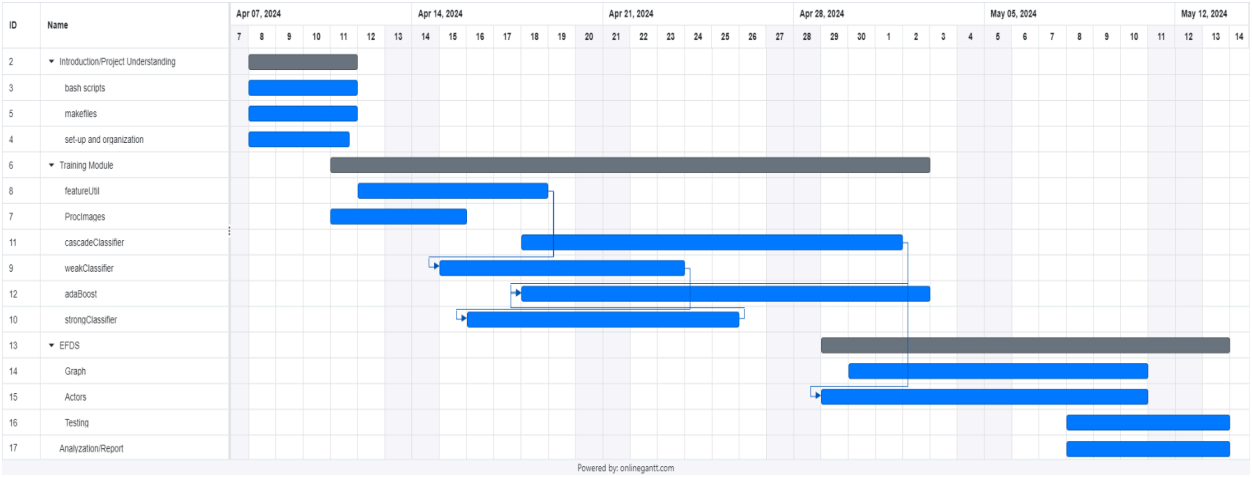


Figure 12: Iterative testing of negative (left) and positive samples (right).

Here we can see the difference in classifications, the negative samples all yield feature values within the weak classification of almost less than 200, whereas the positive samples are much higher (most significant or first values are into the thousands). Because of this large range, the threshold value is likely in between these values, and with the impacts of the weights, likely, these samples were correctly classified after development.

Project Planning and Management



Conclusion

In conclusion, much of the EFDS project was an implementation of C++ and Bash Script programming language in addition to the Welter implementation in C++. The EFDS project was built via training and testing modules, created by the Viola-Jones algorithm. The Viola-Jones algorithm utilized Haar-like features, to construct the weak classifiers, used to predict if an image had a face or not. Although using the Viola-Jones algorithm was a necessity for the EFDS project, there was a wide range of approaches and implementations of classifiers that could be used for the EFDS project, ranging from the number of weak classifiers used within strong classifiers, the number of strong classifiers necessary to classify a face within the DR and FPR, and ultimately the DR and FPR required to call the EFDS project a success.

Some of the many challenges that arose from working on the EFDS project were fully understanding the Viola-Jones algorithm and ensuring that all of the team members were on the same page, so the work on the code could be concise and constructive, rather than erroneous and destructive. This was solved by consistently communicating questions about the Viola-Jones algorithm, in addition to adding comments to the parts of the code that were confusing to other team members. Another weakness

Some valuable lessons learned from working on the EFDS project would be the sheer difficulty of building a face detection system. At first glance, it appears easy to the human eye to identify faces based on our own valuable experiences, however, the computer and processor do not have that kind of valuable experience. It made the project significantly harder to implement as we had to “teach” the learner how to properly understand if the image has a face and it caused a considerable amount of work to build that framework for the processor to start considering what was a face or not.

References

[1] “Integralimage.” MathWorks, www.mathworks.com/help/images/integral-image.html. Accessed 12 May 2024.

[2] P. Viola and M. Jones. Robust Real-Time Face Detection. International Journal of Computer Vision, 57(2):137-154, 2004.

[3] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2001.

[4] Lescano, G., Mansilla, P.S., & Costaguta, R. (2016). Experiences accelerating features selection in Viola-Jones algorithm.