Embedded Face Detector System Capstone Project Report

Team #5: Pranav Puritipati, Dongyeong Kim, Prem Chandrasekhar

ENEE408M: Section 0101

Shuvra Bhattacharyya

05/11/2022

Table of Contents

Cover Page **1**

Table of Contents **2**

Signature (Approval) of Each Team Member **2**

Pranav Puritipati 2

Dongyeong Kim 3

Prem Chandrasekhar 5

Executive Summary **6**

Main Body **6**

Goals and Design Overview 7

Realistic Constraints 9

Engineering Standards 11

Alternative Designs and Design Choices 12

Technical Analysis for System and Subsystems 14

Design Validation for System and Subsystems 15

Test Plan 16

Project Planning and Management 17

References **20**

Signature (Approval) of Each Team Member

Pranav Puritipati

- Created and implemented the strong classifier actor
- Summarized the Viola-Jones algorithm in the main README file
- Created the classifiers directory
- Worked on the naming conventions for both the weak classifier and strong classifier files
- Worked on the scripts directory
- Worked on writing the final report

"I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination."

Pranav Puritipati

Dongyeong Kim

- Explained Viola Jones training systems and its algorithm of AdaBoost
- Created integrator for the integrated image
- Explained the machine learning algorithm to team
- Implemented utils based on skeleton code of Prem
    - Weight updates
    - Threshold updates
    - WeakClassifiers
- Debugged program in classifiers
- Final Report

"I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination."

Dongyeong Kim

Prem Chandrasekhar

- Created and maintained directory structure for Github repository
- Wrote Bash scripts for generating random features, downloading dataset, performing classification, training classifiers, and measuring accuracy of classifiers
- Designed, Implemented, and Debugged Welter Actors/Graph programs, Driver program, and Training/Adaboost program
- CMake Compilation for Graph, Driver, and Training projects
    - Allowed for debugging using gdb
    - Cross-compilation for Raspberry Pi
- Trained classifiers and measured performance
- Created test cases for classification using various classifiers and images, training, and accuracy metrics

"I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination."

Prem Chandrasekhar

Executive Summary

The main goal of this project was to use what we learned about the Viola-Jones algorithm to develop our own embedded face detector system (EFDS) that identifies human faces in any image or image regions. To achieve this, our team implemented a features/classifiers configuration subdirectory defining each feature type, created the structure for the image subwindow, implemented the weak classifier actor, implemented the strong classifier actor using the Viola-Jones algorithm as a template, created the image integrate and file write actors, designed and implemented the EFDS graph, and wrote a test suite to test various images.

The main design that we considered for our project was the dataflow graph implementation for the EDFS graph, which we created to visualize how our driver program read in the different actors in our project. In our design, we used four different actor types to read in the classifiers and images, convert the images into 2D vectors, integrate the images, classify the images, and then write the results to the files.

Similarly, in each of our test suites, which uses the driver program, builds the executable by loading in the C and C++ compiler and debuggers, scanning the target dependencies from CMake build, and then creating the object and executable files. Then, the executable is run, which reads in the classifier file, goes into the scheduler, and then goes through the different classifier modes (configure, read, classify, and continue). Then, the results of the classification will be written to the results text file, and compared and validated to our expected-output text file. Additionally, the entire process of the test is documented in our diagnostics text file, and any errors that we expected in the test will be documented in the expected errors text file.

Main Body

Goals and Design Overview

The goal of this project was to design and implement a face detector system targeting the Raspberry Pi using the Viola-Jones algorithm. This involved designing Haar-like features to identify faces, creating a dataflow graph that allowed 24x24 grayscale images to be classified using a cascade of classifiers, and creating scripts to test and automate several steps in the process. Lastly, the entire application would be cross-compiled to run on a Raspberry Pi.

Originally, the goal was to be able to apply the face detection algorithm to an image of any size, and the features and image subwindow would be rescaled and translated across the image to detect faces. However, this is significantly more complicated than we anticipated, so we focused only on 24x24 images. Additionally, we intended to train a cascade of strong classifiers, and although our graph program can support a variable number of strong classifiers, we were only able to train one strong classifier due to long training times.

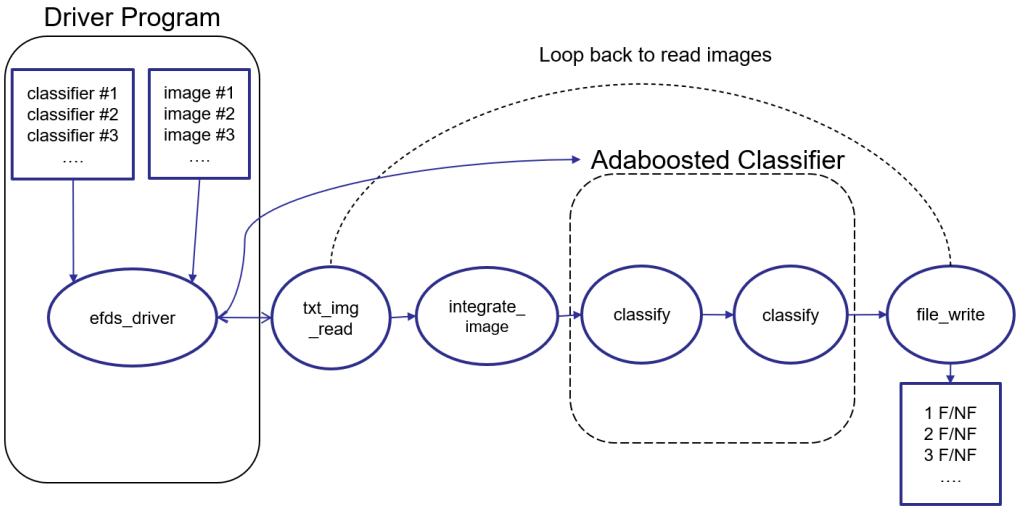


Figure 1. Dataflow for Embedded Face Detector System Graph Implementation.

As illustrated in Figure 1, the driver program reads in the classifiers. The dataflow graph contains four types of actors: txt_img_read, integrate_image, classify, and file_write. In each iteration, the graph reads in a 24x24 grayscale image as a text file, converts it to a 2D vector, integrates the image, passes the image through the classifier cascade, and writes the result to a file.

The classify actor represents a "strong classifier" which holds several weighted weak classifiers. The mode design of this actor is shown below in Figure 2. It can hold an arbitrary number of weak classifiers. After the classification is made, the result is written to the image subwindow and passed to the next actor. If the received image subwindow is already rejected, the classifier does not perform any computations and just passes along the image subwindow.

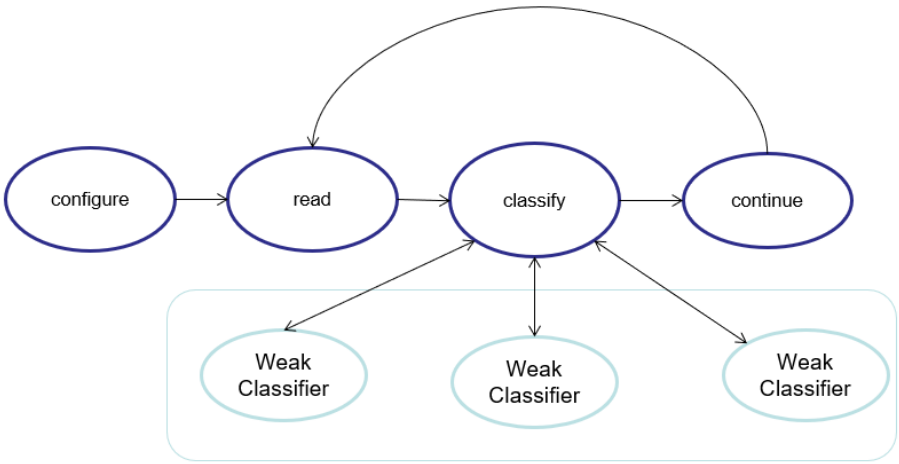


Figure 2. Design of the Classify Actor Modes. The read classify mode relies on the WeakClassifier class to perform the classification.

One of the key functions of the Viola-Jones algorithm is its classifier. Once the classifier classifies its feature value based on all input training integral images and updated threshold, it

can update the threshold and its weight. As described in Figure 3 below, the point is that with the updated value, the updated weight can be used as a strong classifier with error values.

4. Update the weights:

$$w_{t+1,i} = w_{t,i}\beta_t^{1-e_i}$$

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:

$$h(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise} \end{cases}$$

where $\alpha_t = \log \frac{1}{\beta_t}$

Figure 3. Weight update and its usage as a strong classifier.

The fundamental challenges to this are 1) implementing the classification design that allows for a variable number of classify actors, 2) training classifiers through the AdaBoost process described above, and 3) collecting performance metrics about the accuracy of said classifiers.

Realistic Constraints

The main constraint on implementing this project was that the face detection application needed to be implemented using Welter and FIFO buffers. This required the program to be divided into several actors that performed an individual task when invoked. This made several of the components uniform and easy to work with each other.

The programming languages used for this project were limited to C++ and Bash. External libraries like OpenCV could not be used. For example, developing the EFDS graph using Welter-C++ required us to use the simple scheduler, which is inefficient due to run-time overhead, its operation is difficult to predict, and it can deadlock if there isn't sufficient space for an actor's input data.

The minimum requirements we need for the system are the ability to select which images to classify, specify features and threshold values, and select which classifiers to use. The set of images used was limited to 24x24 grayscale images, and the features were the 5 Haar-like features described in the Viola-Jones algorithm.

The intended hardware platform for the system is the Raspberry Pi, despite being developed on the Linux server. This requires the project to be cross-compiled so the executable can be run on the Raspberry Pi. Additionally, since the Raspberry Pi is a relatively low-resource device, the face detector algorithm must be leaner and more efficient than some image algorithms, such as Deep Learning models.

Time and the number of people on our team was also a constraint for this project. We were given a limited amount of time to develop, test, and measure our system, and we only had three people in our team to accomplish all of this. This affected how we explored the problem and prioritized tasks, and we worked through the tasks together rather than dividing into subteams.

Engineering Standards

In our project requirements, the training module was to be implemented as a C++ program, the EFDS and parts of the training were to be implemented using Welter-C++, the scripts subdirectory was to be implemented using Bash, and the overall project would also include text files and CMake code.

C++ is one of the standard programming languages used throughout our project. For each actor, we had a header file, which listed the libraries and other header files we wanted to include, define directives, function headers to be implemented in the .cpp files, and the use of both public and private members. One C++ standard that we incorporated and guided our design was keeping the class variables private and the functions public so that only the functions can be accessed by other actors. If class variables needed to be accessed or modified, getter and setter methods were used. Additionally, we attempted to enforce a common style for all of the C++ programs we wrote for this project.

Regarding Welter, the define directives are used to indicate the modes of the actor. Additionally, the enable and invoke functions are used to implement the actor firings, the reset function is used to return the actor to its original state, and the connect function is used to connect an actor to its input and output edges. Using Welter required us to create each element to switch between modes. With specific regard to the graphs, the simple scheduler was used, which is responsible for enabling each of the actors before they are fired.

Bash was also another standard programming language we used throughout our project. For each script, we included *#!/usr/bin/env bash* as the first line in our scripts to use the default version of Bash, and then we wrote the commands we wanted to run sequentially to perform the actions we wanted.

For our classifiers directory, we chose our file formats for the weak and strong classifiers so that the files properly describe each classifier type. Additionally, we also used CMake for our test suites as a way to generate build tools for cross-compilation on the Raspberry Pi.

Alternative Designs and Design Choices

Our first design choice came in the form of choosing between ease of implementation and efficiency. In the threshold update, there were two options, one was to update its values from large size to small size instead of increasing its values in small size 0.1 or 1. And the other option is to sort the feature values and regard their values as a single threshold value. There are some pros and cons for the two solutions. The first one has less runtime, but it needs more scripts and loops, so when the program has errors, the debugging time takes longer. And the second option has a longer runtime, but the sorting function is already in the standard C++ library, and it only has one loop, so it is easier to implement, and debug. We observed the feature values can vary with thousand or hundreds of values, so the first option is normally better, but for the time conflict, we chose the second option for threshold update.

Training strong classifiers required the classification system to be functional, which we did not finish until relatively late into our timeline. We were very limited on time to explore how training classifiers could be optimized, so our approach was to try multiple parameters and combine the classifiers into a cascade. We ultimately had trained 5 strong classifiers. Our training strategy was very inconsistent: the first 3 classifiers, which consisted of 5 weak classifiers, were trained from around 200 weak classifiers on a subset of the training images (1000 images). Additionally, there was a bug in the way the features were calculated, so the training was likely flawed. The last two classifiers were trained from 1024 weak classifiers on all the training images. Classifier 4 is the same as classifier 3, but with only the best 5 weak classifiers instead of 20.

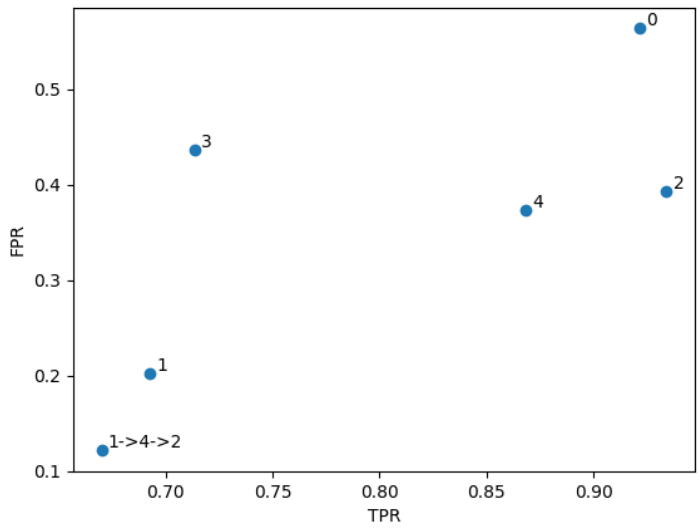


Figure 4. Pareto Diagram of the Trained Strong Classifiers, concerning TPR and FPR.

The two metrics that we used to evaluate our classifiers were the True Positivity Rate (TPR) and False Positivity Rate (FPR). As the names imply, the optimal model has the highest

TPR and the lowest FPR. As shown in Figure 4, the Pareto optimal classifiers (in order of TPR, from highest to lowest) were 2, 4, 1, and the cascade. The full testing metrics for each classifier are shown in Table 1 and the README file in the classifiers/strong/ directory in the repository.

Technical Analysis for System and Subsystems

There were a few deviations in our implementation from the original specification document. For example, instead of having two FIFOs for the classify actor, we decided to only use one output FIFO. Instead of the continue/abort separation, we created a structure that holds the image subwindow as well as a flag marking the image as a face/nonface. Thus, regardless of the classification, the output is always a pointer to this structure. This greatly simplified the graph design, especially when using multiple strong classifiers, by reducing the number of FIFOs in the graph and making the data flow more linear (as opposed to having multiple paths for the image).

We separated parts of the algorithm into separate classes in order to simplify both the design and readability of the code. The feature area sum calculation was encapsulated within a WeakClassifier class, which also stored the feature type, threshold, and polarity values. We created an IntegrateImage actor that performs the image integration once per image, as opposed to within each strong classifier. This made the code both more efficient and cleaner to work with. Throughout the implementation, we prioritized readability and maintainability over pure runtime performance. We also used command line and file outputs liberally throughout our classification and training programs to have an easier time debugging and understanding the flow of values

within the classification graph. This negatively affected our runtime performance, but it made finding issues with pointers and actors not working properly significantly easier.

Continuing with the idea of separating by function, we separated our training code from the classification code. The classification graph driver is called from the training Bash script, but the actual Adaboost procedure relies only on the files generated by the graph. Although it may seem that introducing a separate program complicates it (admittedly, it does in some ways), having a separate training module allowed us to isolate issues with our system to either the classification or the training components.

Design Validation for System and Subsystems

Once we had trained our strong classifiers using the AdaBoost process described in the Viola-Jones algorithm, we tested the performance on the entire testing dataset and recorded the number of false positives and false negatives. We used the precision, recall, and F measure to evaluate the performance of the different classifiers, each of which is defined below.

$$F_{measure} = 2 \times \frac{precision \times recall}{precision + recall}, \quad precision = \frac{TP}{TP+FP}, \quad recall = \frac{TP}{TP+FN}$$

By choosing the classifier with the highest F measure, we were able to optimize between false positive rate and false negative rate and produce the best possible classifier. As mentioned previously, we were not able to put much time into exploring how to optimize these values, but our classifier metrics are shown in the table below.

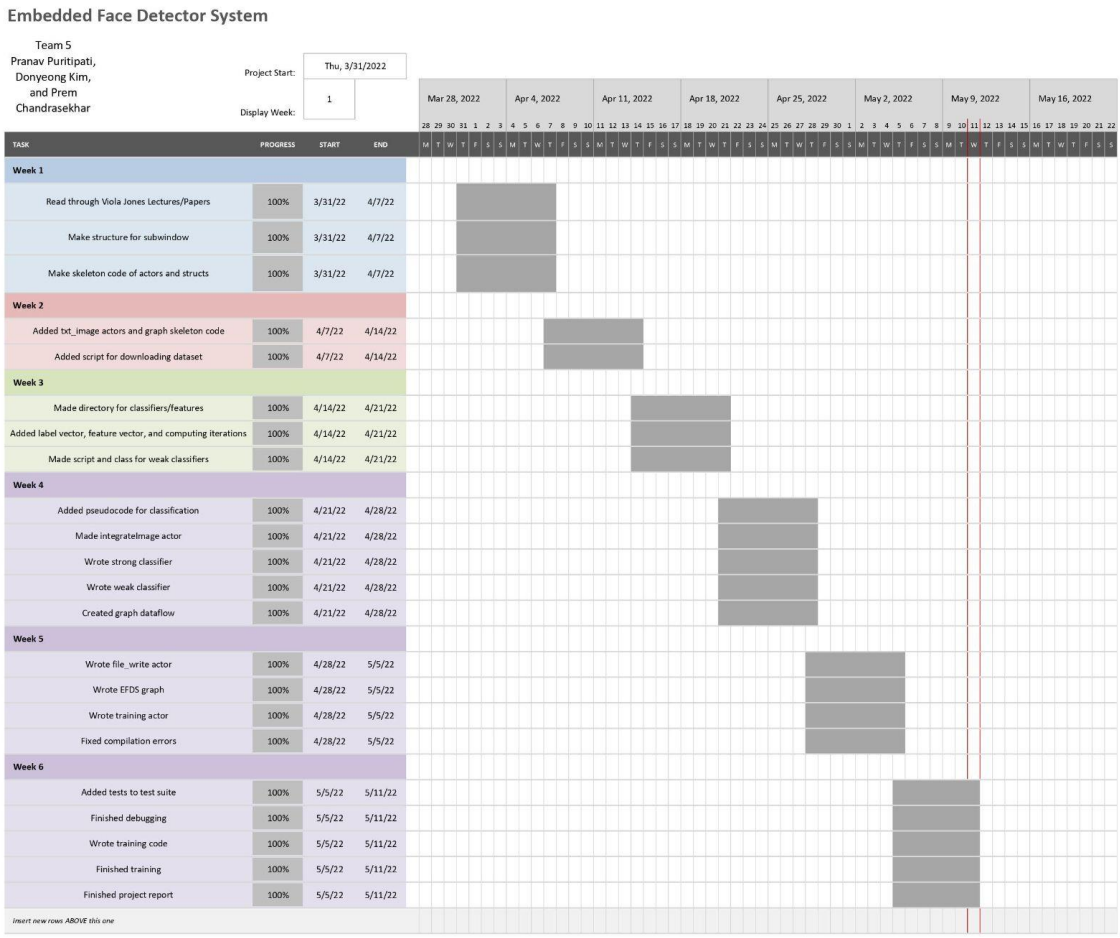
Classifier	Precision	Recall	Fmeasure
0	0.436	0.922	0.592
1	0.798	0.693	0.742
2	0.606	0.934	0.735
3	0.564	0.713	0.630
4	0.626	0.868	0.728
1=>4=>2	0.878	0.670	0.760

Table 1. Table of classifier performance metrics.

Test Plan

We used our test suite as more of a debugging tool than as validation. Test cases were written as needed to test the functionality of the program. The first two test cases were meant to simply read in a set of classifiers and a set of images, which were specified in text files, perform classification on them, and check if the results were as expected. The third test case was made to perform classification on a set of images, and store the feature sums of each image in a file for training. The last test case was for evaluating classifiers on the testing set of images and returning the metrics described in the previous section. It calculated the number of misclassified images for both the faces and non-faces. These test cases were very useful for finding runtime and logic errors in our code, and for determining the optimal classifier to use.

Project Planning and Management



Embedded Face Detector System

Team 5
Pranav Puritipati, Donyeong Kim, and Prem Chandrasekhar

Project Start: Thu, 3/31/2022
Display Week: 1

TASK	PROGRESS	START	END
Week 1			
Read through Viola Jones Lectures/Papers	100%	3/31/22	4/7/22
Make structure for subwindow	100%	3/31/22	4/7/22
Make skeleton code of actors and structs	100%	3/31/22	4/7/22
Week 2			
Added txt_image actors and graph skeleton code	100%	4/7/22	4/14/22
Added script for downloading dataset	100%	4/7/22	4/14/22
Week 3			
Made directory for classifiers/features	100%	4/14/22	4/21/22
Added label vector, feature vector, and computing iterations	100%	4/14/22	4/21/22
Made script and class for weak classifiers	100%	4/14/22	4/21/22
Week 4			
Added pseudocode for classification	100%	4/21/22	4/28/22
Made integratelmage actor	100%	4/21/22	4/28/22
Wrote strong classifier	100%	4/21/22	4/28/22
Wrote weak classifier	100%	4/21/22	4/28/22
Created graph dataflow	100%	4/21/22	4/28/22
Week 5			
Wrote file_write actor	100%	4/28/22	5/5/22
Wrote EFDS graph	100%	4/28/22	5/5/22
Wrote training actor	100%	4/28/22	5/5/22
Fixed compilation errors	100%	4/28/22	5/5/22
Week 6			
Added tests to test suite	100%	5/5/22	5/11/22
Finished debugging	100%	5/5/22	5/11/22
Wrote training code	100%	5/5/22	5/11/22
Finished training	100%	5/5/22	5/11/22
Finished project report	100%	5/5/22	5/11/22
Insert new rows ABOVE this one			

Figure 5. Gantt chart specifying task breakdown and scheduling.

Throughout this project, we used a Google Drive and GroupMe chat to keep track of the tasks that each of us was working on each week, as well as planning for tasks to do in the future. We used Github to keep track of changes in the project codes, and used the issues feature for reviewing changes as needed. Additionally, we held semi-weekly Zoom meetings to work together when needed and to keep each other informed about the progress being made on the project.

As illustrated in Figure 5, we split the project into individual tasks to complete each week over the six weeks allotted. In week 1, we mostly focused on reading through the lecture slides and the research papers to understand the Viola-Jones algorithm as effectively as possible. We also started working on the structure of the 24x24 image subwindow and skeleton code for each of the subdirectories. In week 2, we added the text to image actors and added the script to download the dataset in the scripts subdirectory. In week 3, we made a subdirectory to hold the configuration files for the features, added the label and features vectors to compute iterations, and started work on the weak classifier actor.

The following week, we started by creating pseudocode for the classifier actors and wrote the integrate image actor. Then, we used the pseudocode to write the weak and strong classifier actors while also creating the EFDS graph dataflow. Then, in week 5, we wrote the file write actor, the EFDS graph, the training module, and fixed any compilation errors regarding the source code and the utility code. In the final week, we mostly focused on testing and training as well as debugging and writing the project report.

Conclusions

The classification system was implemented with readability and development in mind. Each aspect of the system was separated into its own class, calling other classes as needed to delegate tasks. This allowed us to write clear code for easier debugging.

Fully comprehending the Viola-Jones algorithm was a challenge. Initially, we interpreted the features to only specify feature type, width, and height, and the position on the image

subwindow would be determined by the image. The Viola-Jones training algorithm was implemented nearly directly from the original paper. Implementing the algorithm into C++ elucidated many parts of it, such as the training and AdaBoost process.

One unique approach we took was using a random subset of the Haar-like features instead of the entire set of 160,000. Using a Bash script, we generated a random feature of any size and position to fit in the 24x24 image. This allowed us to create strong classifiers without needing to classify all the training images for each possible feature, saving weeks of computation.

This project developed our understanding of C++ and Bash much more than anything in our past experiences. We became more familiar with making classes and defining header files and working with structures from the standard library like vectors, iterators, and pairs. Since we dealt with files a lot, managing file streams and writing/reading structures from files was challenging but very educational. We also developed our Bash skills since we used script heavily in our testing and training process. This involved using command line arguments (including default values), performing arithmetic and program logic, and iterating over files in directories. Version control using Git was a challenge, but we managed to learn more about merge conflicts because of it. We are all now familiar with the basic Git commands. It was insightful to develop software working with other people, which presented challenges and benefits. Debugging some logic errors became much easier when we had other people to talk with about potential causes and fixes.

References

[1] L. Shapiro, “Face Detection via AdaBoost,” *University of Washington Computer Science & Engineering*, 21-Feb-2017. [Online]. Available: https://courses.cs.washington.edu/courses/cse455/17wi/notes/FaceDetection17.pdf. [Accessed: 07-May-2022].

[2] P. Viola and M. J. Jones, Kluwer Academic Publishers, Cambridge, MA, tech., 2004.

[3] P. Viola and M. Jones, IEEE, Cambridge, MA, tech., 2001.

[4] R. Gupta, “The intuition behind facial detection: The viola-jones algorithm,” *Towards Data Science*, 12-Feb-2020. [Online]. Available: https://towardsdatascience.com/the-intuition-behind-facial-detection-the-viola-jones-algorithm-29d9106b6999. [Accessed: 05-May-2022].

[5] S. B. Lipman, J. Lajoie, and B. E. Moo, *C++ Primer*. Boston, MA: Addison-Wesley Professional, 2012.

[6] University of Maryland, College Park, Face Detection Using the Viola-Jones Algorithm, 2022.

[7] University of Maryland, College Park, Training Process for the Viola-Jones Algorithm, 2022.

[8] “Vocal Technologies,” *VOCAL Technologies*. [Online]. Available: https://www.vocal.com/video/face-detection-using-viola-jones-algorithm/ [Accessed: 05-May-2022].