Embedded Face Detector System

ENEE 408M, Section 0101, Prof. Shuvra S. Bhattacharyya

May 15th, 2023

Team 3 - Harun Gopal, Thomas Helgesen, Kevin Hermstein, Lucas Kaplan

Table of Contents

2. Team Member Contribution and Signature

Harun Gopal Contribution

I worked on the code responsible for training the strong classifiers alongside Thomas. I primarily worked on helping functions found within the training data flow such as initializing/reweighting weights, and writing the method to write out the final config file at the end of the data flow. Additionally, I was responsible for writing tests for the training code and helping Thomas design the overarching training algorithm.

Harun Gopal Signature

[illegible]

Thomas Helgesen Contribution

I worked on the training program alongside Harun. My main contributions were the top level calculate method in calculate features portion, the feature optimization and select portion of adaboost, the main training and validation dataflows and data structures, makeme script for the training program, scripts to move image files into separate validation/test directories, and debugging the training program.

[illegible]

Thomas Helgesen Signature

Kevin Hermstein Contribution

I developed the strong classifier actor and file sink actor. Along with developing these actors I wrote 11 tests to verify their functionality. I also developed the full configurable graph implementing all of our actors. I also served as the git organizer setting people up with branches and fixing issues with the repo. I also wrote the bash scripts used to convert images and calculate values needed to calculate our F measure. In addition, I helped with writing runme files that required some more advanced bash scripting. I produced all of the tables and plots for Pareto optimization.

Kevin Hermstein Signature

[illegible]

Lucas Kaplan Contribution

I worked on the graph portion along with Kevin. My primary contributions were to develop and test the image read actor and to validate and test the graph code. Within the image read actor, I created code to store the image and the image index in an IIS struct for use in future actors. Additionally, I created code to output a set of labels for our image dataset which was used in determining the accuracy of our EFDS. I created the files required to compile all of the code on the graph side. Lastly, I generated code to determine the runtime of our EFDS within the tests for the complete graph functionality.

Lucas Kaplan Signature

[illegible]

3. Executive Summary

The objective of this project is to develop an embedded face detection system (EFDS), capable of classifying an image as containing or not containing a face, while operating on a Raspberry Pi. This project consists of two components: a dataflow graph and a training module. The dataflow graph for the EFDS is a sequential connection of actors. The first actor reads in the input image, the next set of actors classify the image, and the last actor outputs the classification of the EFDS. To classify each image, the well-known Viola-Jones algorithm was used as the backbone for each image classifier actor, or strong classifier [1]. The training module produces the optimal configuration of rectangular features that are to be used in the EFDS. To do this we calculate all possible rectangular features and then use the adaboost (adaptive boosting) algorithm to select the optimal arrangement of features. Outside of the adaboost algorithm used to train each stage, we set TPR targets as stop rates for each successive stage to balance throughput and number of classifiers used with accuracy.

Testing plans for this project included a robust test suite for both the graph and training sections of the code. This primarily included testing individual methods as well as integration tests combining multiple code units. In addition to a test suite, the training subsystem had a validation test set associated with it. This contained about 140 images and allowed verification of proper performance near the end of development.

The primary variable considered in the final testing of the EFDS was the number of strong classifiers. The highest true positive rate of classification was achieved with one strong classifier, but the most optimal design in terms of the F-measure was achieved with four strong classifiers.

4.1 Goals and Design Overview

The goal of this project is to implement a face detection system made up of cascaded strong classifiers utilizing dataflow graph design principles. This system must run on board a raspberry pi. In addition to developing a functioning face detection system, the effect of various constraints on the system must be considered. The effect of these constraints will be investigated by observing changes to the accuracy (true positive and true negative rates) and runtime of the face detection system when these constraints are altered. Constraints considered were the number of strong classifiers in the cascade and the number of weak classifiers within each strong classifier.

The full embedded face detection system (EFDS) is implemented as a graph utilizing 3 distinct actors: an image read actor, strong classifier actors, and a file sink actor. The only token passed between these actors is an indexed image sub-window (IIS). This IIS contains a 24x24 grayscale image and an image index so that each image that gets classified can be tracked.

The image read actor reads through an entire directory of images. It takes this image and constructs a proper IIS token to be written onto its output FIFO. Each time it is invoked, it reads the next image in the directory.

The strong classifier performs the actual classification of the image. It reads an IIS on its input FIFO. After performing its classification, the IIS is read in, is either written out to the next strong classifier, or it is written to the file sink actor. If the strong classifier happens to be the last one in the cascade, it will always write to the file sink actor.

The file sink actor takes in an IIS as input. It will write out the image index and whether or not a face was detected in the image to the output file. Outputting this to a file allows us to easily compare this file to a “key” of which indices correspond to face or no face and calculate the accuracy of the face detector.

A challenge that must be addressed is how to tell the file sink actor if a face was detected or not. This is a challenge because the project specification dictates that the classifier actor can only write out an IIS onto its FIFOs. This means we cannot add any sort of indicator of whether or not there is a face in the associated image.

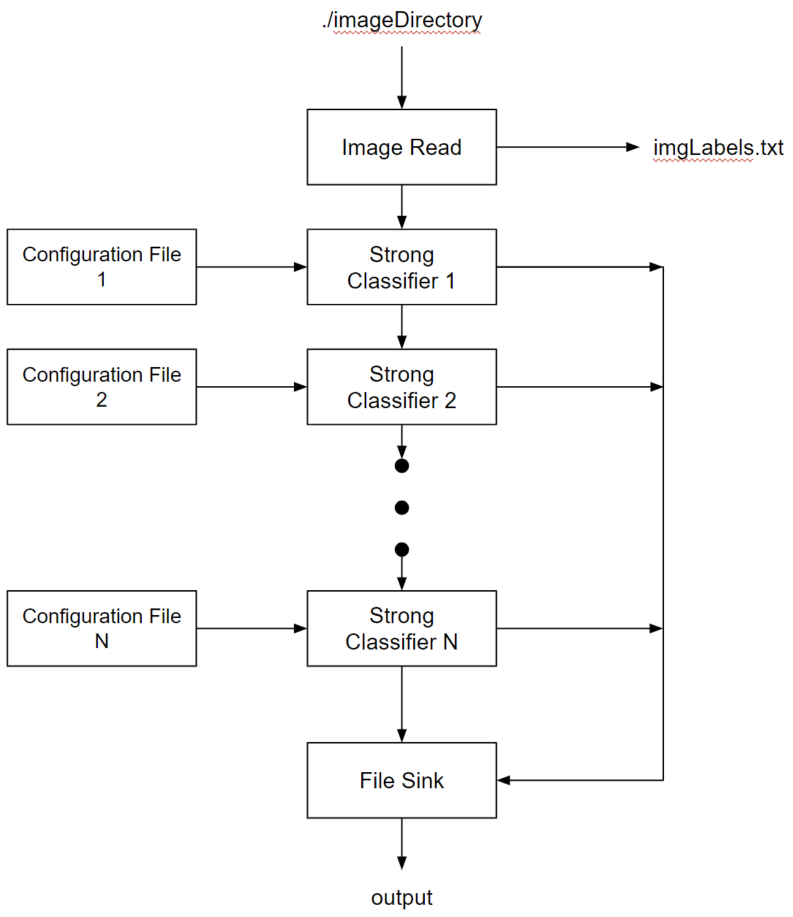


Fig 1: Complete data flow graph.

Figure 1 depicts the full dataflow graph for the EFDS and illustrates its modularity. Through this modularity, the graph will be designed such that the number of strong classifiers can be configured.

Training

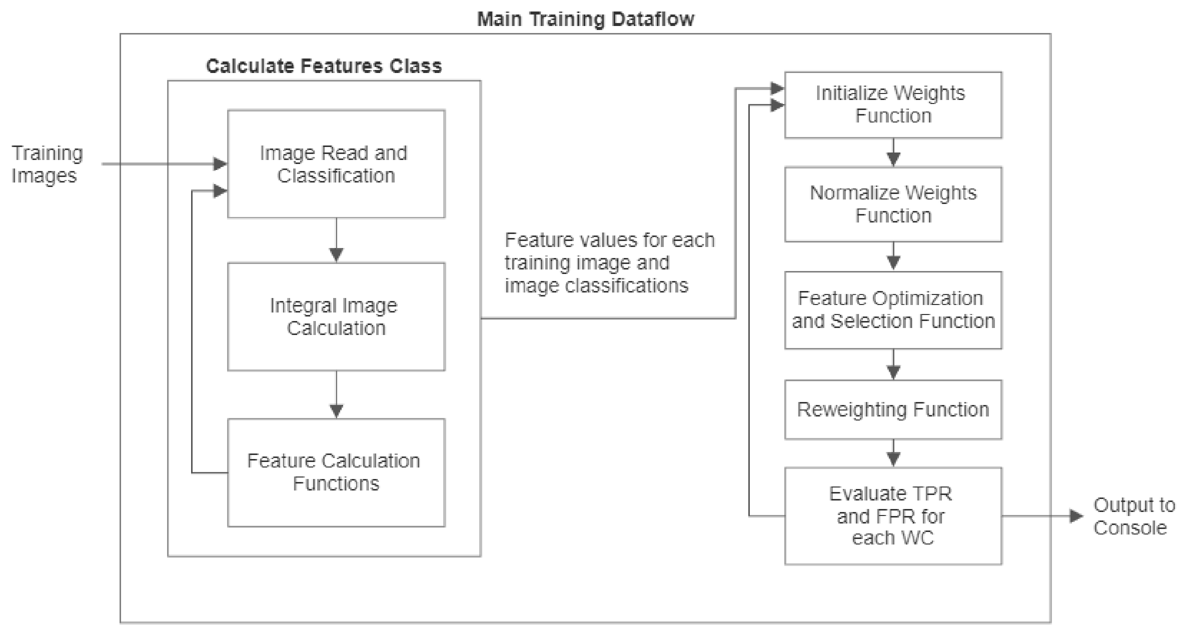

Main Training Dataflow
Calculate Features Class
Training Images
Image Read and Classification
Integral Image Calculation
Feature Calculation Functions
Feature values for each training image and image classifications
Initialize Weights Function
Normalize Weights Function
Feature Optimization and Selection Function
Reweighting Function
Evaluate TPR and FPR for each WC
Output to Console

Fig. 2: Training program structure

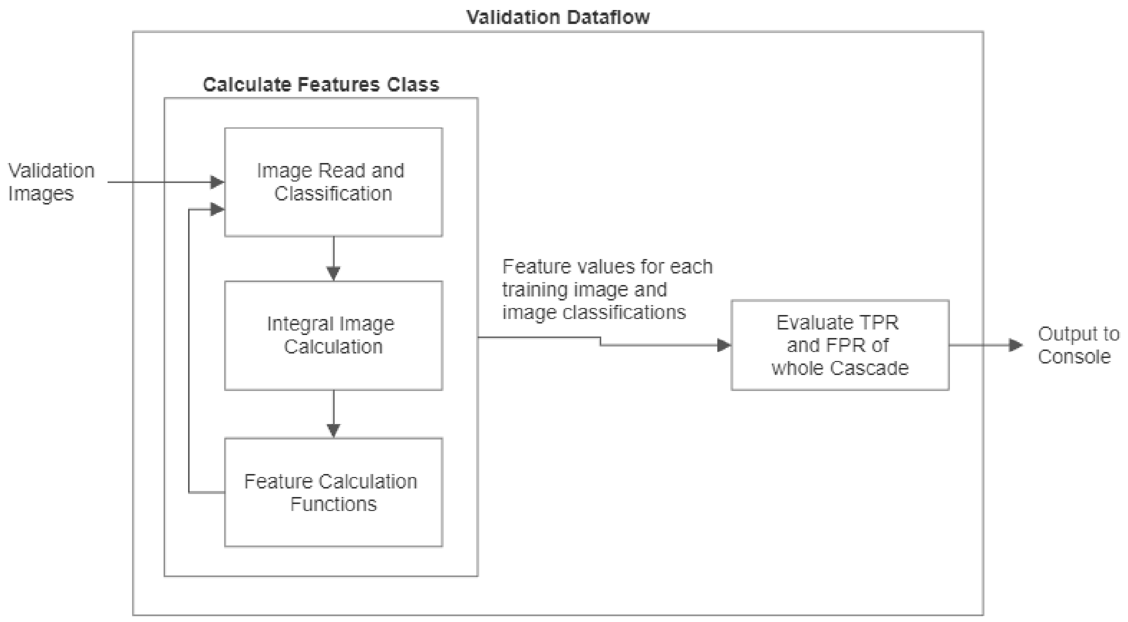

Validation Dataflow
Calculate Features Class
Validation Images
Image Read and Classification
Integral Image Calculation
Feature Calculation Functions
Feature values for each training image and image classifications
Evaluate TPR and FPR of whole Cascade
Output to Console

Fig. 3: Training program structure

The training program is split into three main parts: a calc_features (calculate features) class, the main training dataflow, and the validation dataflow. As seen in the above figure, the calculate features class is used by both the main training dataflow and the validation dataflow. It most notably contains the calculate() method, which is called in order to calculate the features and populate an array of structs of type feature (or more accurately an array of pointers to structs), and a vector of img structs. It does so by iterating through the images in the specified directory and calculating all features for every image.

The feature and img structs are the two main data structures passed between functions. A feature struct corresponds to one feature, and contains its size, position, and the feature values for all training images. The 163,000 feature structs are instantiated via C++'s new operator in order to avoid the memory constraints of static memory. An img struct corresponds to one image, and contains whether that image contains a face and other information relevant to training (image weight and whether to include in training instance). The img structs are implemented in a vector to accommodate varying numbers of images between training sessions.

When called inside the main training dataflow, the calculate() method is used to calculate the features for the main training images. As configured, the user is prompted to pick whether to run the training with the full training set (~6500 image), or a smaller test dataset of 140 images. The test dataset is simply a subset of the full training set that we use to debug the training process. It contains enough images to give approximately correct training results, while still cutting training time down to under 30 seconds for a simple 2 stage cascade.

Once the user has selected the desired training dataset and the corresponding features are calculated, the program begins the AdaBoost selection process. The entire cascade is implemented as an array of vectors, with each element being a struct of type opt_feat (optimized feature). This struct contains the parameters of each optimized feature; threshold, parity, error, alpha, and index, which corresponds to the feature index of the chosen feature. Each step within the AdaBoost algorithm is implemented as a function; initialize_weights(), normalize_weights(), feature_select(), update_weights(), and revise_dataset(). The reweighting functions simply adjust the weights in each img struct.

The feature_select() function follows the feature selection algorithm set forth by Viola, Jones [2]. For every feature being examined the program begins by sorting the training images by their feature values. Then, with sums of the total positive and negative weights and running sums of the same, the optimal threshold and parity can be calculated [2]. The final optimized feature that is found to have the least error among all training images in that training instance is returned to main as an opt_feature struct containing the threshold, parity, error, alpha, and index of the optimized feature. This optimized feature is then added to the cascade and reweighting can begin.

This process is repeated until the TPR target for that stage is hit, indicating a successful stage. This process is founded in the overall goal of keeping the overall TPR relatively high over multiple stages. The overall TPR and FPR is the product of the TPR and FPR of the individual stages, respectively. Meaning that a TPR of .95 over four stages would be .95^4 = .81 overall. Similarly a FPR of .5 on four stages would be .5^4 = .0625. Thus, through balancing the number of classifiers (and thus throughput of the EFDS) with the TPR of each stage, we landed on targets of .93 for the first two stages and .96 for the remaining stages. This, coupled with the adaboost algorithm inherently minimizing the error in each successive training, we were able to reach an FPR of zero in only four stages, which allows our TPR to remain relatively high (~.79).

The third main part of the training program is the validation stage. The validation stage simply tests the trained cascade on a 140 image subset. The features are calculated again for this new set of images using the calculate features calculate() method. The program then calculates TPR and FPR on the subset of images. Finally the program outputs the formatted cascade information into configuration files that are fed into the embedded face detector implementation.

4.2 Realistic Constraints

There were many constraints inherent to this project and in this section we will discuss 5 major ones. The first major set of constraints were those imposed upon the project by virtue of the welter api and fifo buffers. Since this is the library we had available on the pi, we were almost forced to use it to make our code both modular and effective. This included using fifo buffers between actors in our graph, as well as using the invoke and enable functions included.

Had we not had this constraint, we may have had the option to use other libraries for communicating between actors in a graph.

The environment that we developed our code in put a lot of constraints on this project as well. The environment we were programming in is a UMD server and has limited processing power available for us. We really felt this constraint when training our classifiers. Training with our validation test set would take about 30 seconds to process while training with the full dataset would take upwards of 10 minutes. Although we may have been able to optimize this through how we coded our algorithms, a portion of blame belongs to the limited processing power provided to use by our environment.

The programming language (C++ and bash) also resulted in a few constraints placed upon us. Both bash and C++ are low level languages, which permitted us to have a lot of control of our implementation of the project, but also resulted in a few headaches. Due to the complete control C++ provides developers over its memory, any error we made with respect to memory management was difficult to debug. With respect to bash, we utilized it to preprocess some of the images and other minor applications. Some of the constraints imposed by bash include the limited functionality provided, though since we used it for simple scripts we didn’t notice it as a problem as much.

The requirement of cross compilation imposed some constraints on the project as well. Cross compilation inherently means that the environment we develop in is different from the environment we plan on executing the code in. This change in environment results in there being different libraries available in the two libraries. We ran into an issue where one of the file system libraries we relied upon during development were not available on the raspberry pi. This issue was due to the constraint that is inherent with cross compilation.

The last constraint that we would like to address is the constraint testing requirements placed upon the project. We ensured that we unit tested our code during development to ensure that when we integrated, there would be no or at the most, very few issues. However, this testing requirement constrained our project by taking time away from development and other functions. However, not all the constraints were negative. By utilizing tests effectively, our code was constrained in the way that it was guaranteed to hit certain performance metrics. These were five realistic constraints that were either inherent or placed upon our project by the very nature of the project requirements itself.

4.3 Engineering Standards

Many of the key features available in C++, as well as those provided in the DSPCAD libraries, were used to create the dataflow graph for our embedded face detector. In the strong classifier class, for example, each weak classifier is stored in a custom structure, defined as follows:

```
typedef struct weak_class {
     int type;
     int p;
     float thresh;
     float alpha;
     int width;
     int height;
     int x;
     int y;
};
```

The set of weak classifiers which makes up the strong classifier is stored in a vector of these structs. This allows us to easily change the number of weak classifiers in a strong classifier without necessitating the tedious process of reallocating memory. In the strong classifier, image read, and file sink actors, file streams are used to read in information from input files and output information to a file. For instance, a file stream is used in the file sink to output the image index and classification for each image.File streams are a little easier to use than their counterpart in C, and they make the process of writing to and reading from files quite simple.
In terms of the features from DSPCAD, all actors within the dataflow graph use the Welter library to read in and write to their FIFOs. Within the graph itself, each actor and FIFO is defined using the welter library to ensure a simple process to connect each actor to its proper inputs and outputs. Lastly, each part of our code was tested using the *dxtest* functionality. This made it much easier to verify that each subpart of our code functioned as desired and to debug any issues to achieve our desired results.

C++ itself offers many benefits that we took advantage of in the training program. Features like file streams and string streams allow for easy I/O and string manipulation. These operators allow us to use operator overloading to assist in printing formatted struct data at multiple points in our program. Additionally, vectors allow for easy resizing and dynamic memory usage within a neat wrapper. This ease of use of vectors over something using malloc in C or similar functions allowed us to implement them in a variety of applications requiring an unknown or undefined amount of memory. An example of this would be the image data, which is stored in a vector of structs. This allows us to run the program with varying amounts of training and validation images.

Another related feature of C++ we utilized in the training program was dynamic memory allocation using the new operator. Similar to C's malloc, the new operator allows the user to dynamically allocate memory. With the large amount of data being handled, we used this to avoid the memory limitations of stack. C++'s filesystem standard library, new to C++17 was also useful. This allowed us to easily iterate through the directory entries in our training data folder.

Bash was utilized to achieve many peripheral tasks during development. For example, converting all test png files to txt files was achieved via a bash script. This allowed us to only track a single .sh file as opposed to tens of thousands of text files. Additionally, bash scripts were used to compare the output of the EFDS with the key generated by the image read actor. This allowed us to quickly calculate true positive, false positive, and false negative rates very quickly.

4.4 Alternative Designs and Design Choices

The results of our training yielded 4 strong classifiers, each with unique weak classifiers. To explore other designs we chose to investigate what happened to the accuracy and runtime of the EFDS when the number of strong classifiers were changed. The four design points we chose were a graph just the first strong classifier, just the first two, just the first three and lastly all four strong classifiers.

For each run we calculated the true positive, false positive and false negative rates. This way we can calculate the precision, recall, and finally the F measure. Additionally the runtime of the entire EFDS was recorded to allow us to investigate the tradeoff of latency and accuracy. The data collected for each is displayed below.

Num Classifiers	Latency (μs)	TP Rate	FP Rate	FN Rate
1	31254	0.91	0.26	0.4
2	32433	0.89	0.6	0.5
3	33596	0.83	0.4	0.8
4	39209	0.79	0	0.11

Table 1: latency and accuracy of EFDS configurations

From this data we calculated the F measure for each configuration of classifiers by following the formulas for precision and recall [3]:

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \qquad \text{and} \qquad \text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

And finally the formula for the F measure, a measure of the accuracy of our EFDS.

$$F_{\text{measure}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}.$$

The result of these calculations is recorded below:

Num Classifiers	Precision	Recall	F meas
1	0.7777777778	0.6946564885	0.7338709677
2	0.5973154362	0.6402877698	0.6180555556
3	0.674796748	0.509202454	0.5804195804
4	1	0.8777777778	0.9349112426

Table 2: F measure of EFDS configurations

We then plotted the F measure against the number of weak classifiers to observe what would happen to our accuracy.

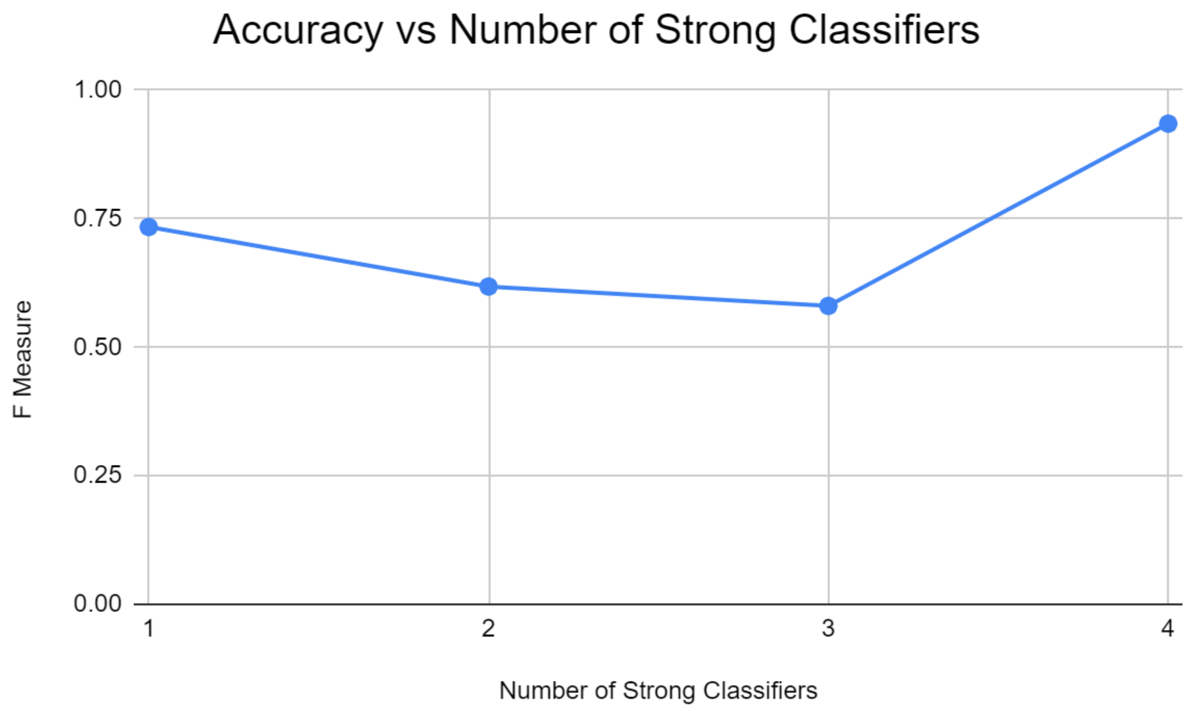


Fig 4: Accuracy as function of strong classifiers.

We can observe in figure 4 that the accuracy initially declines when more strong classifiers are added, but then has a major increase once the fourth and final classifier is added. However, it should be noted that each classifier added does increase our runtime. This is not reflected in the graph but can be observed in table 1.

To be able to interpret our results in regards to the tradeoff of accuracy and latency, we constructed a pareto curve. In this case, the y axis should indicate “worse” performance with higher values. The F measure metric indicates higher accuracy with higher values. In other words, a higher value indicates higher performance. To account for this we can simply plot 1-F on the y axis. Because 1 is the max value F can be, this is essentially the same as taking the inverse. This “inverted” F measure is then plotted against the runtime of each configuration of classifiers.

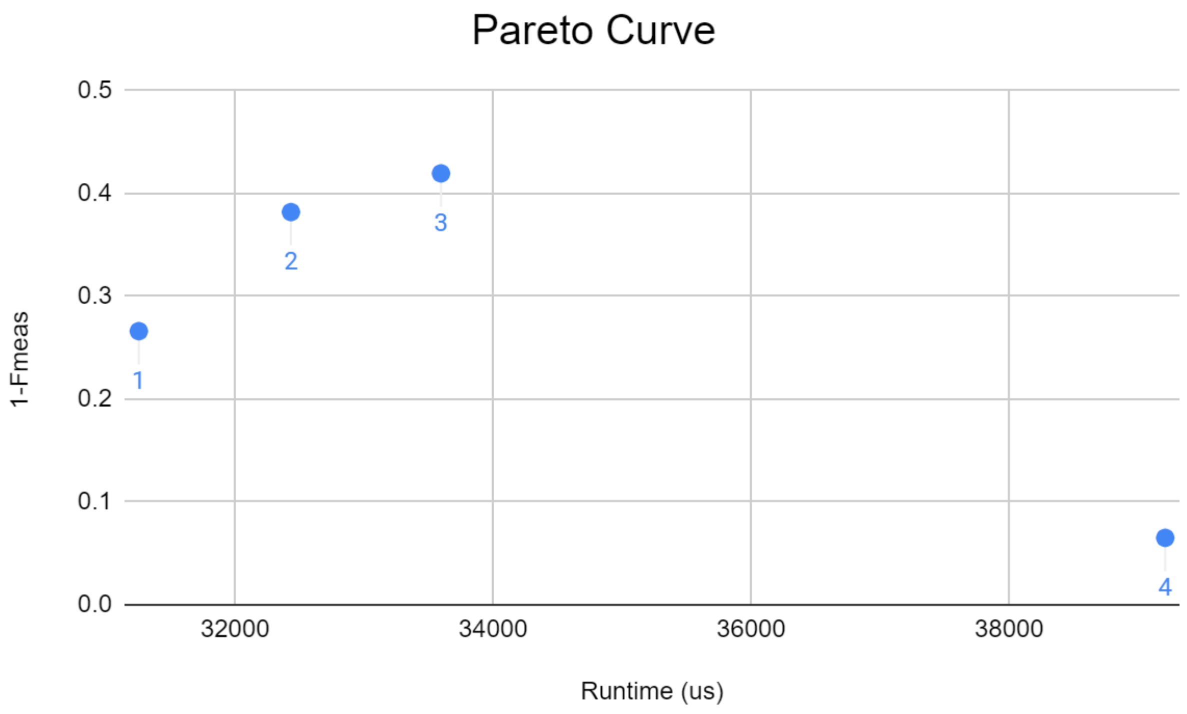


Fig 5: Pareto Curve.

The points are labeled with the number of strong classifiers in the EFDS. We can observe that both the 2 classifier and 3 classifier configurations are dominated by the 1 classifier configuration. We can observe that our Pareto points are then the one classifier and the four classifier configurations.

4.5 Technical Analysis for System and Subsystems

The Indexed Image Sub-window

The IIS contains a pointer to a 24x24 image represented as a 2D integer array and an index in the form of an integer. This index is necessary to relate the output of the system back to an actual image. The IIS was implemented utilizing a struct. This struct contained two fields. A square 2D array of integers of size 24, and an integer to store the index.

Image Read Actor

The image read actor is a fairly simple actor, as displayed in the dataflow graph below. It takes in one input, the image directory, and produces two outputs, an IIS and a file called *imgLabels.txt*. The actor has 2 modes: READ and ERROR. When the actor is invoked, if there is no image directory provided on the input FIFO, the actor will go into the ERROR mode. Otherwise, the actor will go into the READ mode. Within the READ mode, the actor completes four tasks. First, the actor stores the current image index, *k*, in the IIS stored privately within the actor. Next, the actor iterates through the image directory *k* times to get the path of the current image. After this, the actor writes *k*, as well as if the image represents a face or a non-face, to *imgLabels.txt*. Last, the actor reads in each of the 576 pixel values for the image, stores them in the actor's private IIS, and writes this internal IIS to the output FIFO.

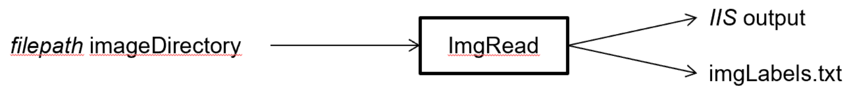


Fig 6: Dataflow graph for the Image Read actor.

There were a few improvements and changes implemented over the course of creating this actor. Originally, we planned to use the name of the image to track its image index, since each image has a number value after its image type. For example, some of the image file names are "graves111_89.png" and "GULF_2.png", so we were planning to make their image indices 89 and 2, respectively. After attempting to implement this, we noticed that some of the images share the same number value, such as "goldwater67_23.png" and "B5_00023.png", signifying that this approach would not work. Next, we planned on storing the image index outside of the image read actor and instead storing it in the graph. Although it would be easy to simply iterate the image index for each iteration of the graph, we realized that the graph itself cannot input anything into the image read actor and therefore cannot track the image index. Both of these changes led to our final implementation for the image read actor.

Strong Classifier Actor

Each strong classifier is made up of weak classifiers which are chosen by the Viola-Jones training algorithm. The strong classifier takes in a single IIS token as input, then it will pass along that same IIS to different actors depending on whether or not a face was detected.

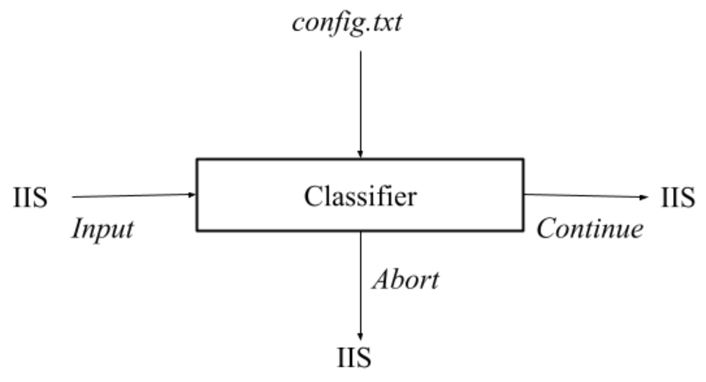


Fig 7: Strong Classifier inputs and outputs

The strong classifier has five states: configure, read, classify, true and false. In the configure mode the actor reads in data from a configuration file. This data includes the number of weak classifiers within the strong classifier and for each weak classifier it includes the feature type, position, size, parity, threshold and alpha value. The strong classifier is only ever invoked once in this mode. After this it transitions to the read mode wherein the actor reads an IIS token. It then moves to the classify mode where it calculates the feature values and determines if there is a face or not. If it detects a face, it moves to the true mode where it writes the IIS to the next classifier. If it does not detect a face, it transitions to the false mode where it writes the IIS to the file sink. It then transitions back to the read mode. It will cycle through these modes for each image provided to the actor.

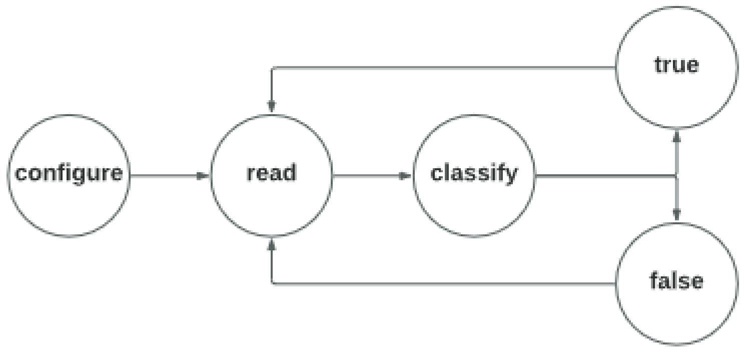


Fig 8: Strong Classifier state diagram

The configure state utilizes C++ data streaming to read in the data from a configuration file designated on the command line. The classifier actor contains a vector of structs where each struct contains fields that store the necessary information about each weak classifier. This vector is then iterated over in the classify stage and the corresponding feature value for each weak classifier is calculated.

The File Sink Actor

The file sink actor is a modified form of a typical file sink actor. To maintain consistency with the rest of the graph, the file sink actor also reads in an IIS token. It then writes information out to a specified output file.

Do determine whether it should write out that a face was found, the actor has 2 input FIFOs. One is an abort FIFO shared by all strong classifiers. Any token read from this FIFO will be written out to the output file with the label “NF” to indicate that a face was not detected. This is a fine approach as it maintains the standard of only IIS tokens being written and we do not have to worry about data races on the FIFO and the graph is iterative. The other input FIFO is the continuous output of the last strong classifier. If an IIS makes it all that way to this FIFO then we have detected a face. The file sink actor outputs this IIS with “F” to indicate a face was detected.

The EFDS Graph

The full EFDS graph implements all of the actors previously described. Initially the number of actors was static, but it was necessary to allow the graph to be configurable in the number of strong classifiers it contains. The graph takes in three parameters: The number of strong classifiers, then input image directory, and the output file name.

From the number of strong classifiers we can calculate the number of total actors and FIFOs. We can then allocate the proper space and instantiate each actor. Because the cascade portion of the graph is so modular, instantiating each strong classifier is done within a for loop.

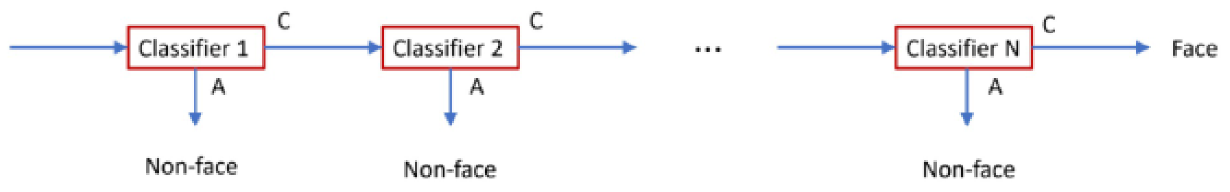
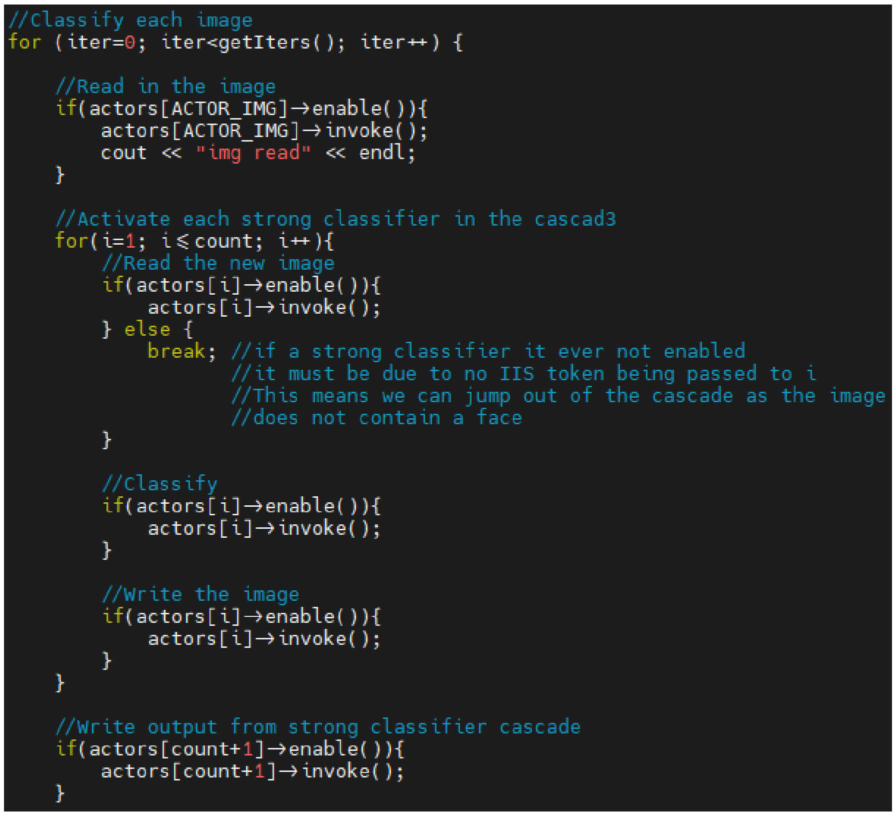


Fig 9: EFDS classifier cascade

The graph scheduler runs the full EFDS for every image in the input directory. The basic flow is as follows: Configure each classifier (This only needs to be done once) The image read actor is invoked to write an IIS token to the first classifier. The first classifier determines if there is a face then passes the token along to the next classifier or writes to the abort FIFO. The file sink actor checks which FIFO contains the token and writes the image index and “F” or “NF.” This can be handled by two simple for loops. An outer loop runs for each iteration. An inner for loop runs the cascade of classifiers and breaks if a token is not passed to the next classifier.



```
//Classify each image
for (iter=0; iter<getIters(); iter++) {

    //Read in the image
    if(actors[ACTOR_IMG]->enable()){
        actors[ACTOR_IMG]->invoke();
        cout << "img read" << endl;
    }

    //Activate each strong classifier in the cascad3
    for(i=1; i<=count; i++){
        //Read the new image
        if(actors[i]->enable()){
            actors[i]->invoke();
        } else {
            break; //if a strong classifier it ever not enabled
                   //it must be due to no IIS token being passed to i
                   //This means we can jump out of the cascade as the image
                   //does not contain a face
        }

        //Classify
        if(actors[i]->enable()){
            actors[i]->invoke();
        }

        //Write the image
        if(actors[i]->enable()){
            actors[i]->invoke();
        }
    }

    //Write output from strong classifier cascade
    if(actors[count+1]->enable()){
        actors[count+1]->invoke();
    }
```

Fig 10: EFDS Scheduler Code Example

This is essentially the entire scheduler. It is extremely compact, easy to follow and can run for any number of strong classifiers. This is even easier to follow than the previous implementation with a fixed number of strong classifiers. Each stage is easily discernible.

4.6 Design Validation for System and Subsystems

To validate our system, we crafted a smaller subset of inputs that we called our validation input set. This subset provided 2 main benefits compared to the full dataset. First, due to its smaller size, it would allow the training code to choose and configure classifiers much faster than would be possible with the full dataset. Classifiers could be chosen and tested for the validation set in about 30 seconds while the full dataset would take about 10 minutes. Second, the size of the dataset also made it easier to identify issues with the code and which specific image was where the problem was occuring. This made debugging code easier compared to only using the full dataset.

Despite the fewer number of images, we ensured that the validation dataset was a good representation of the full dataset by adding a random variety of samples from the full dataset into the validation set. This randomness helps ensure that the results we get from the validation set will also apply to the full dataset when that is run.

To compare designs, we calculated the TPR (True Positive Rate) and FPR (False Positive Rate) resulting from each design. These benchmarks give us a good idea of how effective a particular design is. We tested various designs first with the validation set and then again with the full dataset (but only for promising designs) and calculated the TPR and FPR for each design. This method allowed us to efficiently and effectively determine which design was the best design for our application.

For verifying the design of a full EFDS graph, all configuration files for classifiers were set up such that they always would detect a face. This would ensure a token is passed through the entire cascade without issue. This gives us a good upper bound on the runtime. We also then would test with images whose pixels contained garbage values and as such would never detect a face. This would ensure the functionality still worked.

4.7 Test Plan

Face Detector

The embedded face detector is composed of a multitude of components, and as such, requires a multitude of tests to ensure its proper functionality. We created and ran four groups of tests for this code, one for each actor type within the graph—the classifier actor, the file sink actor, and the image read actor—and one for the entire graph itself. Each of the tests is detailed in the list below.

Tests on the Classifier Actor

1. Check the classifier actor's ability to read in a configuration file with the same format as those produced by the training code.
2. Check the actor's ability to properly read in and store an IIS, presented on its *input* FIFO.
3. Check the actor's ability to calculate an integral image, given a 24x24 image with all its pixel values set to 1.
4. Check the output for each of the 5 weak classifiers computed in the strong classifier.

Tests on the File Sink Actor

1. Check if the file sink actor produces a proper output file, which should take the format of "[image index] [F/NF]" on each line.

Tests on the Image Read Actor

1. Check if the image read actor outputs the correct image index and properly reads in and stores one of the positive sample images provided.

2. Check if the image read actor outputs the correct image index and properly reads in and stores one of the negative sample images provided.

Tests on the Embedded Face Detector Graph

1. Check if the graph produces a sensible output when classifying a directory containing one positive sample image.
2. Check if the graph produces a sensible output when classifying a directory containing three sample images. 2 are negative samples, and 1 is positive.
3. Check if the graph produces a sensible output when classifying a directory containing a subset of 139 of the sample images.

Training

The test plan for Training code consisted of 2 main components. The first component was unit testing individual methods while the second component was integration testing of sections of the dataflow together.

The first component was unit testing individual methods. Throughout this project, we decomposed problems into subproblems to facilitate the construction of the code. This decomposition made it simpler to understand but also allowed for more efficient development and testing. With the training module, we created a test suite for each method created to ensure proper functionality with the code we created. Some of these test suites were simple while others required lots of test and edge case checks.

The second component was integration testing. We combined methods and utilized them in the training dataflow. Throughout dataflow development, checkpoints were established where the state of the code would be checked to ensure it is performing as expected. Although this process was not a formal test suite, it provided some assurance that actual progress was being made. Once the full data flow was constructed, a validation stage was created. This stage would test the classifiers chosen against a subset of images (about 138 images) to ensure that the classifiers are yielding appropriate results. This acted as our final testing suite to showcase that the training module as a whole worked.

4.8 Project Planning and Management

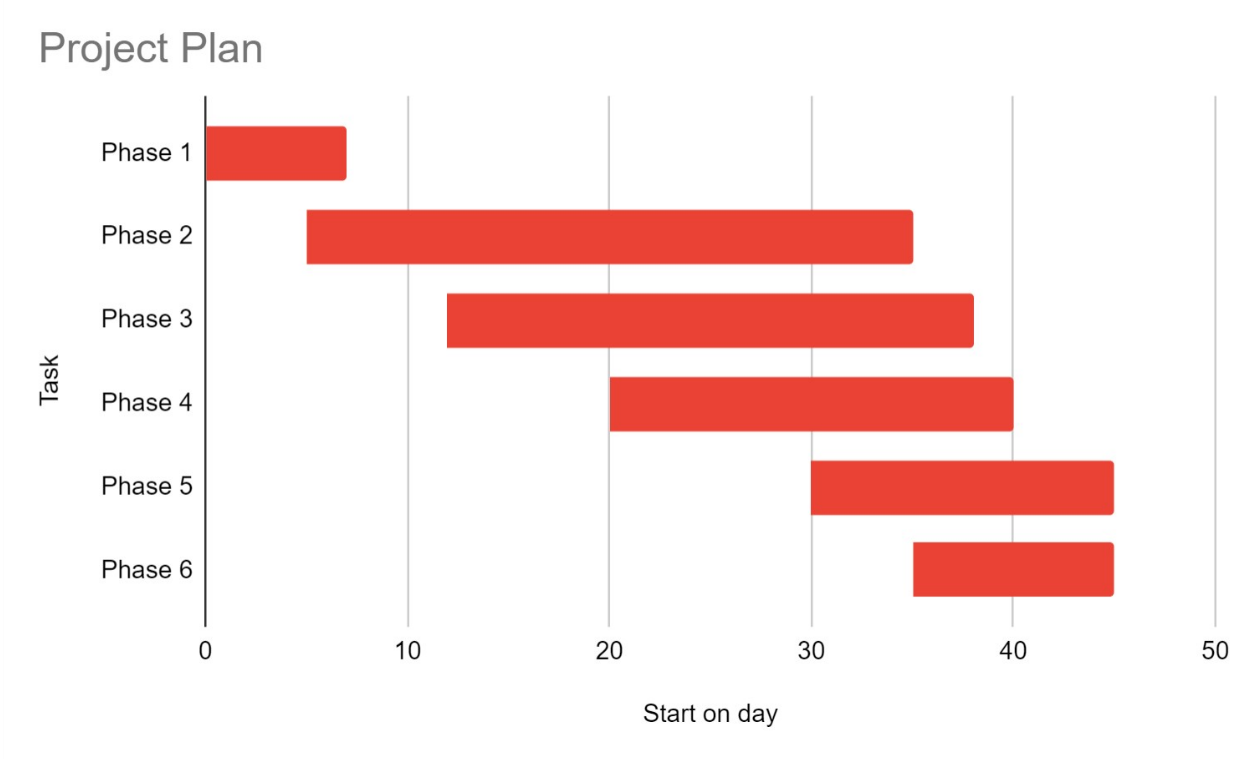


Fig 11: Gantt chart

Key:

Phase 1 - Understanding Specifications and Project Decomposition

Phase 2 - Code Development

Phase 3 - Unit Testing

Phase 4 - Low-Level Integration

Phase 5 - High Level Integration and Testing

Phase 6 - Write Report and Perform Analysis

At the beginning of the project, we established a rough timeline shown above that dictated when we should be done with certain aspects of the project. This also gave us a reference for how quickly we were making progress. We purposely made this timeline vague in the sense that we didn't provide deadlines for specific tasks, rather just phases of the project. Instead, we decomposed the project such that we assigned each member a subsection of the project. Kevin was responsible for the Classifier Actor; Lucas was responsible for the Image read actor; Thomas was responsible for the Adaboost feature calculation; Harun was responsible for developing supporting training functions. These designations were not strict. There were a lot of instances of collaboration where one team member would help another with their code. This gave individuals flexibility to determine what to work on next and overall made the process smoother.

With respect to adherence to the timeline, we underestimated the amount of time integration and tests would take. Developing code wasn't that big of an issue, but when we had to integrate code between team members, we ran into more bugs which took more time. This resulted in high-level integration and debugging taking away some time allotted to writing the report. Regardless, the timeline served its purpose and allowed team members to accomplish their tasks in a reasonable timeframe.

5 Conclusion

One major issue faced in debugging the training program was memory errors. We are storing and manipulating a large amount of data – data for over 6,000 images and 163,000 features, which ended up causing inconsistent issues with double free or corruption errors, segmentation faults, and size vs previous size errors. We resorted to using Valgrind to find the source of the double free or corruption error. Although once we resorted to Valgrind the issue was resolved fairly quickly the process of debugging before we got to that point seemed endless. A lesson learned is to not be afraid to fall back into debuggers and tools. Although oftentimes these tools are new, as both GDB and Valgrind were to most of us before this project, learning them will pay out ten fold. Our memory issues would have been insurmountable without Valgrind, and surely would have been taken care of much sooner if we'd used it from the beginning.

Another massive roadblock we faced towards the conclusion of the project was poor performance when attempting to train with the full training dataset. TPRs and FPRs were acceptable when training with the 140 test images, but when training with the full dataset we faced extremely low TPRs on stages involving more than one weak classifier. The issue ended up being that only the correctly classified positive samples were being reweighted, rather than all the correctly classified samples. Although quite a simple problem, it ended up taking days to find.

Another large lesson learned on the training portion is to implement tests sooner. Tests came towards the end of our project as a way to double check ourselves, rather than something that was being proactively designed with the program itself. This made certain aspects of testing tougher, even just little things like not being able to use our image size for testing smaller images (2x2, etc.). Although the number of rows and columns in our program is a changeable parameter, implemented as defined in one of our header files, this doesn't lend itself well to testing. Define statements, at least in our implementation, have to be in the low level files rather than the top level main file.

One of the main factors that limited early test implementation was the failure to define the overall architecture and data structures early on. While we had a good idea of how to lay things out and what functions were needed, we could have done a better job at finalizing the overall architecture before diving into our respective roles. This would have made early testing easier as well.

In the development of the EFDS graph and associated actors, the main design method was to adhere to the guidelines specified in the project description. Straying from these guidelines can cause developers to lose direction. Another important method was to test as we developed. The state property of actors makes this extremely easy. Actors were developed one state at a time. For example, for the classifier actor we started with the config state. Then we wrote a test for this state and made sure it worked. Once this test was passing we could move onto the next state a repeat, making sure all previous tests were also passed. Another method was to write general functions to help with the readability of code. For example, for calculating features, we simply developed a function to calculate the value box that could be called for each distinct part of a feature.

One of the biggest challenges faced was cross compiling our code. At first it was easy as we utilized very basic C/C++ libraries in all of our actors. However, to achieve what we thought was ideal for our image read actor we tried to utilize the filesystem library which is not recognized by the version of GNU used to compile for the raspberry pi. Concerns about this were expressed, but no testing was done until it was too late.

A large lesson learned is to always program code to be adaptable and configurable whenever possible. Our initial implementation of the EFDS graph was hard coded in the number of strong classifiers. Manually changing the code and recompiling each time was not sustainable when changing the number of classifiers we were testing. Eventually we developed a graph that was configurable from a single integer on the command line. This made life much easier and changing the number of classifiers is as easy as changing a single argument. Also the graph code was much easier to follow and was a more accurate representation to the dataflow graph.

6 References

[1] 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.

[2] P. Viola and M. Jones. Robust real-time face detection. International Journal of Computer Vision, 57(2):137-154, 2004.

[3] S. Mankad, A Tour of Evaluation Metrics for Machine Learning,
URL: https://www.analyticsvidhya.com/blog/2020/11/a-tour-of-evaluation-metrics-for-machine-learning/