CSCI 1430 Final Project Report:
Online and Offline Improvements to WebGazer

The Eyes of Dr. T.J. Eckleberg: Benjamin Davis, Oliver Hare, Emma Catlin, Shannon Ward.
Brown University
19th December 2017

Abstract

The ability to estimate eye gaze direction is useful in various computer vision contexts like determining users point of attention on a screen. We propose and tested various improvements to WebGazer, an online gaze tracking system. Our improvements are split into two sections; online and offline changes. The online changes consisted of improving WebGazer by making edits to its javascript by fine tuning its machine learning and its data processing. The offline changes consisted of creating a convolutional neural net to run on a set of extracted data and compare its ability to predict gaze direction against WebGazer to show the feasibility to replacing its machine learning method with a CNN. Our online edits to webgazer.js resulted in insignificant changes to the overall accuracy of the unmodified WebGazer code. The CNN outperformed WebGazer for each person in the testing set while processing training frames fairly quickly, showing WebGazer could feasibly be improved by replacing its current machine learning method with a CNN.

1. Introduction

WebGazer attempts to track user's gaze at a screen in real time. Rather than relying upon some geometric model which attempts to identify the pupils position on the eye, WebGazer relies on machine learning. By creating a feature map from images of the user's eyes, it trains itself and creates a model for the user. It then uses the model to estimate user's focus point in real time.

Though there are multiple avenues to potentially improve the current iteration of WebGazer, we focused on adjusting its machine learning process in order to create a better model that would reduce test error.

Our changes and adjustments to reduce error fall into two categories, online changes and offline changes. Online changes refer to changes to WebGazer to reduce error, but preserved its ability to run in real time. The offline changes on the other hand, ran on the Frames Dataset, a given set of training and testing data. Offline changes were more drastic and difficult to implement while maintaining the ability to run WebGazer in real time.

Due to our focus on maintaining real time response for online changes, we directed our efforts towards altering the WebGazer javascript code. The three main things we chose to address were implementing k-means to address potential overfitting and underfitting of the data, improving image contrast in the pupil detection, and investigating the different regressions already implemented in the WebGazer code base.

We also attempted to fine tune some of the parameters in the javascript code. After inspection of the dataset and initial testing though, we realized that it was not feasible to fine tune these parameters by testing on the entire dataset. The Full Dataset contains about 942 minutes of video of 51 participants, which means it takes about 10 hours to test. With our computational constraints this made the amount of time necessary to satisfactorily test and tune parameter unfeasible. We decided in the end to test on three of the participant videos so that we could iterate more rapidly.

For our offline changes we implemented a CNN to replace the current WebGazer machine learning method. CNNs generally provide better performance on computer vision tasks than traditional machine learning methods, but at the cost of increased time complexity. We tested our CNN on the Frames Dataset to explore how much it would reduce error, while also tracking the speed at which it is able to train on the individual video frames.

Based on the results of our CNN we should be able to determine whether it would be worthwhile for WebGazer to replace its current machine learning method with a CNN for better accuracy, or whether it would be in-feasible for such an implementation run in the real time WebGazer application.

2. Related Work

The software used for the online changes included Matlab for testing changes and Javascript for direct changes to the WebGazer code. For offline changes Tensorflow in Python was used to build the CNN. We used the Full Dataset to test

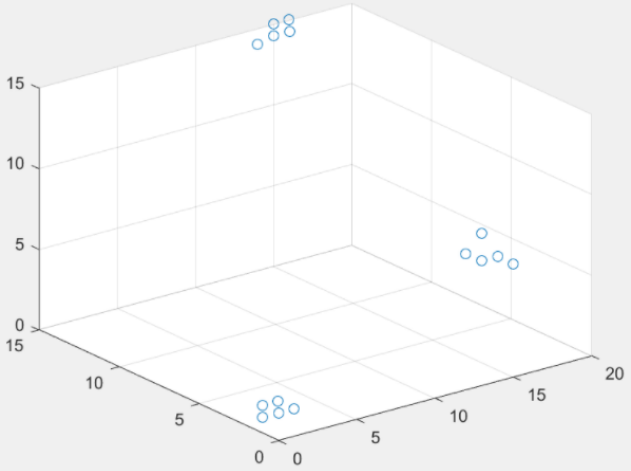

Figure 1. Visualization of K-Means Matlab Implementation

the online changes, and the Frames Dataset to test the offline changes.

There has been work in developing real time CNN based gaze detection by George et al.[2] This shows that it is possible to get real time results, and with the results we found with our CNN, it seems like a very viable path for WebGazer to follow. The architecture and method of our CNN was heavily based on the CNN used in this paper.

We also found some resources which show that keeping the feature maps in RGB instead of converting them to greyscale could keep us from losing valuable information.[1] This inspired to test it out on WebGazer and see what results we could get.

3. Method

We'll split out method section between the various proposals for the online and offline parts of our project.

3.1. K-means (linearReg.js)

To begin implementing K-means, we first implemented the algorithm in Matlab in order to ensure correctness and test the algorithm. Example output from the Matlab implementation is shown below (figure 1).

We continued by implementing the K-means algorithm in javascript, with the goal of comparing the accuracy of various regressions paired with the K-means algorithm. In the end, we only tested the linear regression paired with the K-means algorithm, and it performed significantly poorly that further tests of K-means with other regression models did not seem like a good use of our time. Instead, we focused our efforts elsewhere.

The previous layout of the WebGazer javascript code was to run the regression model on the left and right X and Y datasets. In order to integrate K-means, we modified this to first include passing each dataset into the K-means algorithm, then running the regression model with the output from K-means.


Figure 2. RGB versus Greyscale feature map data

The K-means algorithm in javascript is composed of three functions: initializeMeans, kMeansCluster, and kMeans. In order to run the algorithm, you call kMeans, passing in the number of random clusters to generate and the data to run k-means on.

3.2. RGB Greyscale (blinkDetector.js and util.js)

We decided to implement image processing in the three separate channels, R, G, and B, instead of greyscale, which the current implementation uses. As discussed in The RGB versus Greyscale video[1], using the three separate color channels can improve image contrast, which is used for WebGazer in the pupil detection. See figure 2 for visualization.

The previous code in blinkDetector.js is as follows:

```
1 var grayscaled = webgazer.util.
    grayscale(eye.patch.data, eye.width,
     eye.height);
2 var equalized = webgazer.util.
    equalizeHistogram(grayscaled,
    equalizeStep, grayscaled);
3 var grayscaledThreshold = webgazer.util
    .threshold(equalized, threshold);
```

In order to incorporate RGB instead of greyscale, first we separate the image into the three separate color channels (ignoring the alpha channel), then we perform the equalization step and the threshold step on each channel separately. The only part that was a little unweildly was that in order to keep the general function (extractBlinkData) output the same, we needed to return a one channel image, not a three channel. So after recombining the three equalized and thresholded channels, we then convert them back to greyscale.

To accomplish these changes, we added two functions to util.js: `rgbExtract` and `rgbConstruct`.

`rgbExtract` extracts the red, green, or blue component from an image patch. It can be used for the whole canvas, detected face, detected eye, etc. It takes in an array representing the imageData to extract from and a number representing the rgb component to extract (0 for r, 1 for b, 2 for g). The function returns an array representing the red, green, or blue component of the original image patch.

`rgbConstruct` reconstructs an image patch given the red, green, and blue components. It takes in three arrays representing the red, green, and blue components of the image respectively, and returns an array that interweaves these components.

3.3. Fine Tuning (clmGaze.js)

There were variables in the code which we found to be labeled as "may require some fine tuning." We wanted to investigate how much impact these variables actually had and in turn, how susceptible the code is to small tweaking. We tested a variety of different values for these variables and were able to glean more information about their importance. The specific values we tested in order to fine tune were the tracker parameters Q and R, and the pixel error. We tested numerous values for each of these components, which will be displayed and discussed more in the $Results$ section. The specific tests we tried were modifying the pixel error value in clmGaze.js, which is initially set to 6.5. We tested pixel error values of 1.5, 3.25, 5.5, 7.5, 9.75, and 13. For Q in clmGaze.js, we tested multiplying the original Q value by 2, dividing by 2, and setting the 1/4 values to 1/3. The original Q matrix is shown below:

```
var Q = [ [1/4,  0, 0, 0,  1/2,   0],
        [0, 1/4,  0, 0,    0, 1/2],
        [0, 0,   1/4, 0, 1/2,   0],
        [0, 0,   0,  1/4,  0, 1/2],
        [1/2, 0, 1/2, 0,    1,  0],
        [0, 1/2,  0,  1/2,  0,  1]];
```

3.4. Accuracy Comparison (csvParser.py)

For the accuracy comparison python script, we parse the csv file inputted, assuming it is of the format: P_#... , Accuracy. The script simply adds up all of the distances, and calculates an average, ignoring any 'nan' distances. The script by default calculates the average of all the rows in the csv file, but provides an optional third flag which only calculates the average accuracy over the first three participants (P_1, P_12, P_13). We utilized this flag in order to expedite testing, so that we don't have to run our tests on all of the videos in the full dataset in order to compare results. Instead, we chose to run our tests and compare just the first three participants. (Eq. 1):

$$csvreader.py < file\_path > [First3Participants] \quad (1)$$

3.5. Convolutional Neural Net (gazerCNN.py)

Note for grader: to run both the training and testing of our CNN simply use python to run gazerCNN.py. It will output the results into CNNreults/results.csv.

The CNN was designed to better predict the point of focus of the user than WebGazer on the Frames Dataset. The results of the Tobii Eye Tracker were used as ground truth, meaning that the goal was to get results with a lower average distance to the Tobii Eye Tracker estimates than WebGazer.

This means the Tobii Eye Tracker estimates were used for training labels, and then as the goal for the test data.

A. Data

The frames dataset consisted of 26 unique users, with the training set consisting of 66,031 frames. Each frame included the data of the results of the WebGazer, Tobii Eye Tracker, data from the CLM tracker for different points of each users face, and the actual image from the set. The results of WebGazer had an x coordinate and a y coordinate, whereas the Tobii Eye Tracker had x and y coordinates for each eye, so four total datapoints. Averaging the x and y coordinates of each eye was used to get the final focus point.

B. Models

Our CNN trained on users independently, creating a set of unique models for each user. This was due to the fact that users' eye region are very distinct from each other, so training a models on the entire set of users would heavily bias the models accuracy towards whichever general eye shape was most highly represented. This would result in a simply not very good set of models or very racist ones.

WebGazer's learning algorithm used one feature map containing the two eyes, which was then fed through two models to get independent estimates for the x and y coordinate. Our CNN on the other hand creates x and y models for each eye, thus four models altogether. It then averages the x and the y coordinates to get the final coordinates. This allows the model to be more specific in how it learns both eyes, rather than being overly biased towards the pixels in one part of the feature map, and missing strong signals that could be coming from the less biased eye.

Training a model for either the x or y coordinate of an eye just required us to feel in the relevant eye's x or y labels from the Tobii Eye Tracker data. The CNN would naturally then train for that coordinate based upon the loss.

C. Architecture

The architecture of our CNN was based on CNN used in Anjith and Aurobinda's paper Real-time Eye Gaze Direction Classification Using Convolutional Neural Network.[2]

The input was images of size 42x50. These were fed

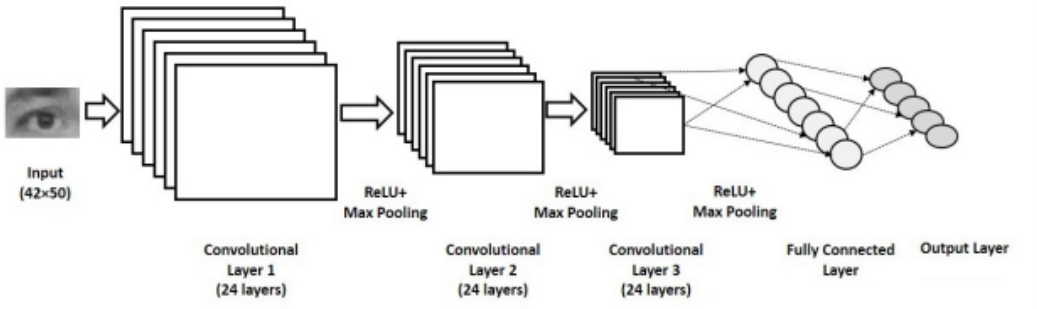

Figure 3. Architecture of the CNN

through three convolutional stages. Each stage consisted of a convolutional layer with 24 filters, followed by a RELU activation function, then a 2x2 max pooling layer to reduce the resolution by half. The first convolutional layer had filters of size 7x7, then the second had filters of size 5x5, then the last had size 3x3. This was followed by a fully connected layer with 4096 units, and then a second fully connected layer with one unit to reduce us down to our final output, either an x or y coordinate. This consists of 9 total layers, 3 convolutional, 3 max-pooling, 2 fully connected, and 1 input.

The loss function was the mean squared error (Eq. 2).

$$L(f(x), y) = (f(x) - y)^2 \tag{2}$$

Where $f(x)$ was the output of our model, and y was the Tobii Eye Tracker estimate for the relevant eye and coordinate. This was chosen as our output is one number, and we just wanted to reduce its euclidean distance to the label as much as possible.

D. Data Processing

The data from frames first needed to be processed before it could be fed into the CNN. This consisted of creating a list of paths from those provided to us by the CS1430 staff for training and testing.

For each user, all paths related to them were collected. Then we went through each path, accessed it's $gazePredictions.csv$, and then for each row in the CSV, created an individual set of frames data.

When training for each epoch the data is shuffled. For every batch in the training set, a feed dict is created by appending images to an array equal to the batch size, and and creating an array of equal length of labels. Then the session was trained using this feed dict.

A similar process was used for testing, except omitting the epochs and shuffling, and retrieving the logits from our model for our results.

E. Parameters

After some tests, the final CNN used a batch size of 1, 1 epoch, and a learning rate of $1e-4$. Reducing batch size greatly increased performance while not sacrificing much runtime, so we opted to have the lowest batch size possible. Increasing the number of epochs increased performance slightly, but the cost of doubling runtime by going from 1 epoch to 2 epochs was not justified by the performance boost. Multiple learning rates are orders of magnitudes around $1e-4$ were tested, but $1e-4$ was found to have the best results.

4. Results

4.1. Online Results

To begin our evaluation of our online changes, we ran the unchanged code numerous times in order to get an accurate representation of the baseline. The baseline average distances though varied significantly (see table 1). The sample standard deviation of the baseline average distances is 0.00664, calculated using the following formula (eq. 3 where N is the number of distances, x_i is a given accuracy, and $mean$ is the sample mean):

$$StandardDeviation = \sqrt{\frac{1}{(N-1)}\sum_{i=1}^{N}(x_i - mean)^2} \tag{3}$$

The average of all of the baseline average distance is 0.23603, which results in the following standard error of the mean, 0.00235. Due to the variance of the baseline, we concluded that our online changes would have to vary significantly from the average baseline in order to conclusively represent improvements or changes. The largest difference between the baseline distances and the overall average baseline is 0.00954, so in order for a change have a significant

Baseline	Average Distance
Baseline 1	0.23571
Baseline 2	0.24557
Baseline 3	0.23313
Baseline 4	0.24041
Baseline 5	0.23034
Baseline 6	0.22773
Baseline 7	0.24413
Baseline 8	0.23123

Table 1. Numerous runs of the unchanged WebGazer online code. Average accuracy is calculated specifically over the first three participants.

impact, the difference in average distance when compared to the average baseline distance would have to be larger than 0.00954, at the very least. Due to this difference, we can effectively ignore any average distances that lie between 0.22650 and 0.24555.

Based on our analysis of the baseline, we came to the conclusion that many of our attempts at fine tuning did not result in significant changes. Looking at the results of fine tuning in figure 4, only a few results lie outside of the range from 0.22650 to 0.24557. Of these outliers, $PixelError = 1.5$, $PixelError = 9.75$, $Q * 2$, and $Q * 1/2$ were all worse than the average. Of those that are better than the average, $PixelError = 5.5$, $PixelError = 3.25, RGB$, $Q * 1/2, RGB$, and $PixelError = 3.25, RGB, Q * 2$, the largest difference of distance was 0.00226 for $PixelError = 3.25, RGB, Q * 2$. Even though $PixelError = 3.25, RGB, Q * 2$ had the best resulting average distance based on these numbers, we cannot conclude that it is actually an improvement over the baseline, because the baseline tests varied at most by 0.01907. In order to prove or disprove that $PixelError = 3.25, RGB, Q * 2$ did any better than the baseline, we would need to conduct more tests to have more data to compare.

Based on the results displayed in figure 5, our k-means algorithm paired with the linear regression model performed significantly worse than the baseline average. Due to the poor performance of k-means with the linear regression model, we decided that further tests of k-means with other regression models would not be a fruitful route to take.

WebGazer had several different regression models built in so we tested each of them to see what kind of performance difference there was. These results can be seen in table 2. Surprisingly, all of the regressions besides the default ridge regression, often returned NaNs, so this required a lot of debugging to track down the issues. But the results showed that the linear regression performed the best by a fair margin. The weighted ridge regression numbers were surprising, but we suspect this could be due to outliers caused by errors

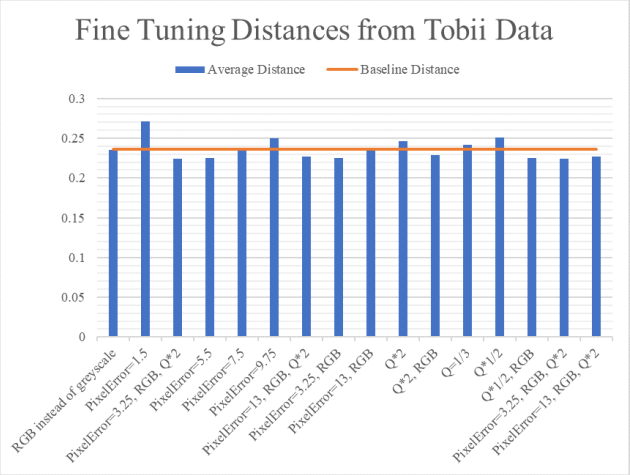

Figure 4. Our fine tuning failed to improve the average distance from the Tobii eye tracking data significantly. Average distance is calculated specifically over the first three participants. RGB meaning separating into the three separate RGB channels instead of greyscale.

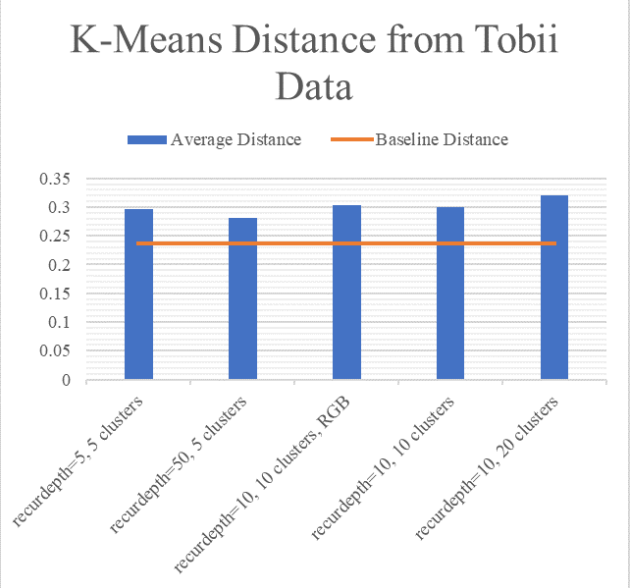

Figure 5. K-Means run with the linear regression failed to improve the average distance from the Tobii eye tracking data significantly. Average distance is calculated specifically over the first three participants. RGB meaning separating into the three separate RGB channels instead of greyscale.

similar to the NaNs for the other regressions. This lead us to pursue implementing the K-Means for the linear regression first.

4.2. Offline Results

The CNN greatly outperformed WebGazer in estimating both the x and y coordinates on average across all users.

Regression Models	Average Accuracy
Baseline Average	0.23603
Linear Average	0.20420
Weighted Ridge	3.59062

Table 2. The results of running the different regressions showed that linear regression was the best at gaze prediction

Method	Average Distance
WebGazer X	0.178
WebGazer Y	0.195
CNN X	0.106
CNN Y	0.115
CNN Left X	0.115
CNN Right X	0.099
CNN Left Y	0.140
CNN Right Y	0.108

Table 3. Both methods perform better on estimating x coordinates than y coordinates, but our CNN outperforms Webgazer's estimates for all outputs and their averages.

Method	Average Distance
WebGazer	0.291
CNN	0.172

Table 4. Our CNN's average distance from the Tobii Eye Tracker was on average more than 0.1 better than WebGazer

Both methods had greater difficulty estimating the y coordinate of the point of focus, likely due to the shape of the eye, less visibility of the pupil when looking along this axis, and inability to use the whites of the eyes unlike the x coordinate. Still our CNN dramatically the distance error of both coordinates (table 3).

Our CNN reduced the x coordinate distance error by 40%, and the y coordiante distance error by 41%. The estimates from each individual eyes outperformed WebGazer on their own, but on average each user's right eye was much more accurate than their left. In the future it may be more accurate to only use the right eye, but this may cause issues with specific individuals whose left eyes are more dominant.

Using Pythagoras's Theorem (Eq. 4) to find the average Euclidean distance error per user from Tobii Eye Tracker's estimates gives us our final results.

$$D(x, y) = \sqrt{(x + y)^2} \tag{4}$$

Our CNN outperformed WebGazer with 41% less distance error averaged across all users (table 4). Further, for each user our CNN had a lower average distance error than WebGazer. This means each of the 26 individual users saw improvement by using our CNN instead of WebGazer.

Additionally the CNN was able to train the model on average 51 frames per second, though it needs to process four frames to build the four CNN models. Thus this could run in real time up to about 13 frames per second.

4.3. Discussion

We found that most of our tweaking and testing made little to no difference in the overall accuracy of WebGazer. With more time and computational time we could potentially implement some parameter optimization machine learning method. The CNN was the only thing to improve the accuracy in a significant way. However the difficulty with this approach is still getting it to run in real time. So the main trade-off is sacrificing accuracy with time.

Based on the runtime of our offline experiments, theoretically the CNN could run in real time at about 13 frames per second, but this doesn't take into account the extra resources that would need to be spent processing incoming video data, as well as the fact that the 51 frames per seconds number from out results was an average. Running in real time the training process could have various slow downs at times.

Running the models after they have trained requires a similar amount of time, so it seems we wouldn't be able to expect a real time Web Gazer with more than 10 frames per second.

There are various alterations we could make to the CNN to possibly improve its speed at the cost of performance such as using a larger batch size, changing the learning rate, and reducing the size of the architecture. Anjith and Aurobinda's paper Real-time Eye Gaze Direction Classification Using Convolutional Neural Network [2] supports this line of thought, as they employ a CNN for gaze direction estimates and is able to support 24 frames per second. Although these improvements would decrease performance, previous tests show it would generally still outperform WebGazer in its current form, although it wouldn't be guaranteed that we would see an improvement for each individual user.

Based on the results and this discussion it seems worthwhile in the future if trying to improve the real time WebGazer to replace its current machine learning methods with a CNN. Although it could reduce the frames per second of the system, the overall improvement in performance, even with a simplified version of our CNN, would be well worth it.

5. Conclusion

In attempting to improve WebGazer we introduced a suite of adjustments to the WebGazer's original javascript code in order to improve its realtime results. Additionally we created a CNN to run on the Frames Dataset to show that changing WebGazer's current machine learning method to a

CNN would yield less error, but perhaps at the cost of time complexity.

The changes to the real time WebGazer code, or the online changes, included a K-means clustering algorithm paired with its linear regression model, RGB data when processing images, and fine tuning the R, Q, and pixel error parameters to increase performance. These tweaks and changes resulted in little to no improvement in error for our real time WebGazer implementation.

The offline changes refer to a CNN that was trained and tested on a premade set of training and testing data. The CNN is comprised of 9 layers, including 3 convolutional stages. Running the neural net independently on each user by creating models for both eyes for both x and y coordinates resulted in a reduction of error for every single user in the test data, as well as a 40% reduction of error on average overall.

The results of our online changes show the difficulty of making small tweaks in order to improve complex systems that use machine learning. Likely a parameter optimizing machine learning method would be necessary to see greater reduction of error.

The offline changes show the current WebGazer could potentially greatly reduce its error by adopting a similar CNN to ours. This may increase time complexity of Web Gazer, but even a simplified version of our CNN would greatly reduce error, while possibly maintaining a reasonable framerate for users.

References

[1] RGB versus Greyscale 2
https://www.youtube.com/watch?v=1-jURfDzP1s

[2] Anjith George and Aurobinda Routray. 2016. Real-time Eye Gaze Direction Classification Using Convolutional Neural Networks. arXiv:1605.05258. Retrieved from http://arxiv.org/abs/1605.05258.

2, 3, 6

Appendix

Team contributions

Oliver Implemented the Convolutional Neural Net that ran over the Frames Dataset to test potential offline changes. This consisted of researching relevant CNN architecture, then writing it in Pythons Tensorflow, as well as code to read in data (gazerCNN.py). Additionally he tested tuning parameters of the CNN to get good results in a reasonable amount of time. Oliver contributed a significant amount to the written report, writing every section about the CNN (Method and Results), as well as the Conclusion and Discussion Sections. He heavily contributed to the Introduction as well.

Shannon Gave our in class presentation. She also figured out how to build the javascript code and test our online changes. She tested different regression models, including ridge, linear, and weighted ridge. This consisted of debugging the different regression code because they were found to often return NaNs in their predicted data due to improper error checking. Shannon worked on data visualization for the final written report. She wrote the Related Work section and wrote the analysis of the different regression model results, and completed the final editing pass.

Ben Implemented the initial k-means algorithm in Matlab.

Emma Implemented and tested the k-means algorithm in javascript. She also tested tuning various variables in clmGaze.js, including Q, the pixel error, and R. She implemented and tested using RGB instead of greyscale in blinkDetector.js. Finally, she also wrote a simple python script to compare accuracies between log files (csvparser.py). Emma wrote sections 3.1, 3.2, 3.3, and 3.4 of the written report, as well as the analysis of the baseline, fine tuning, and k-means results in section 4.1.