EPFL

CHILI
computer-human interaction
in learning and instruction

LEARN
Center for Learning Sciences

Eye Tracking on iPad

Author:
Victor Schneuwly

Supervisor(s):
Lucas Burget

Professor:
Pierre Dillenbourg

June 9, 2024

Contents

1 Introduction

Dyslexia is defined in the *Diagnostic and Statistical Manual of Mental Disorders (5th ed.; DSM–5; American Psychiatric Association, 2013)* as a difficulty in learning to read, despite otherwise normal intellectual abilities and normal hearing [5]. It affects between 5 and 10% of the population [1]. This learning disorder can lead to difficulties in mastering many skills, such as obviously reading, but also writing or pronouncing words [3]. All of those competences are mandatory to be able to follow in the school system. Thus, it is important to diagnose it as soon as possible.

As we will discuss in section 2, there are studies that show that people with dyslexia have different eye movement patterns than people without. Our goal is to explore how feasible such detection can be done using eye-tracking on iPad. We are going to measure how accurate and precise we can be, and see if we can reach high enough numbers to be able to detect patterns that are specific to people with dyslexia.

If we see that it is possible in some way, we could then integrate this feature into the Dyslexico application. With that, we could have more insight of where the users are looking at during the games.

2 Literature Review

The first paper that we are going to talk about is called *Individuals with dyslexia use a different visual sampling strategy to read text* by Franzen et al.[2]. It investigates the distinct visual sampling strategies from people with dyslexia. The metrics that they used are the fixation duration, the saccade length, and the frequency of atypical jumps. The results show that there is a significant difference between the two groups. One is that people with dyslexia have generally slower reading speed. They also found that they skip fewer words during the first pass, that they fixate on more words and that they stop more frequently. This make reading for them a more laborious and effortful task.

In the second paper, *Eye movements during text reading align with the rate of speech production* by Gagl et al.[4], they highlight the synchronization between eye movements during reading and the rate of speech production. The paper also mentions that reading is influenced by word length, word frequency, and predictability. The first two factors link back to the saccade length and the fixation duration that we talked about in the previous paper. It also mentions how a link between reading and speech rate is only found in people with lower reading skills This can be applied to people with dyslexia.

To link back to our project, we see that what is most important is the saccade length and the fixation duration. This means that the precision we need to reach between two computed points should be the length of a saccade. Those saccades are usually between 7 and 9 characters wide[2]. With this we can say that we need to aim for high precision, so to be precise in the relationship between two points. The accuracy, so being able to precisely place a point, is less essential for dyslexia detection, but is still something to think about to have a better eye tracking system.

3 Material and Methods

The idea of the final application would be to have it integrated with Dyslexico. On this prototype, we simply have a live feed of what the camera sees. On it, we add axes on the user's face and eyes. We then also have a point, which is the computation of the estimated gaze point of the user on the screen.

In the following sections, we will explain how we implemented the different parts of this application, and the different steps we took to get to the final version.

3.1 AR and Swift

First, let's talk about what is available on iOS platforms. The app is coded in Swift[1], and we use the following libraries:

- `ARKit`[2] for the augmented reality part.
- `SceneKit`[3] for the 3D rendering.
- `SpriteKit`[4] for the 2D rendering of the overlay.
- `UIKit`[5] for other helper functions for the UI.

First, we had to choose our coordinate system, known as `WorldAlignment`[6] Since all the computation we are going to do need to be related to the camera, we chose the `camera` alignment. That way, we have the camera as the origin of the world coordinate system.

One of the main classes that we will use is the `ARFaceAnchor`[7] class from `ARKit`. This class enables us to access the user's face transform. The face transform is a matrix that encodes the position, rotation, and scale of the face in the chosen coordinate system. Then, we can use it to get the transform of both eyes, given in the local coordinate system of the face. The first use case of those transform is to add the axes on the user's face and eyes on the scene.

As we will talk about in section 3.2, there is also another object that we can use from the `ARFaceAnchor` class. This object is the `lookAtPoint`[8] property. As the documentation states, it is "a position in face coordinate space estimating the direction of the face's gaze".

3.2 Eye Tracking

In this section, we will talk about the different steps we took to get to where we are now.

[1]Website of the Swift language: https://www.swift.org
[2]https://developer.apple.com/augmented-reality/arkit/
[3]https://developer.apple.com/scenekit/
[4]https://developer.apple.com/spritekit/
[5]https://developer.apple.com/documentation/uikit
[6]https://developer.apple.com/documentation/arkit/arconfiguration/worldalignment
[7]https://developer.apple.com/documentation/arkit/arfaceanchor
[8]https://developer.apple.com/documentation/arkit/arfaceanchor/2968192-lookatpoint

3.2.1 Estimating the gaze point

Using `lookAtPoint`: At first, to estimate the gaze point, we tried to use the `lookAtPoint` property from the `ARFaceAnchor` class. This property is a point in the face coordinate space that estimates the direction of the face's gaze. What we did is to transform this point into the world coordinate space. Once we had that point, we wanted to intersect it with the screen. When testing this property, we used the built-in projection function from the scene, that we will talk about in section 3.2.2. The values that we got were way off from what we expected in the tests, and also, when we looked at the screen, the point was not where we were looking at. From there we realized that the issue could come from either this intersection function, or the projection itself. Our first guess was that the intersection function with the use of `lookAtPoint` was the issue, so we decided to implement our own intersection function.

Using a custom intersection function: For our own intersection function, we decided to use the right eye position and direction. The idea is to find the intersection between where the eye is looking at and the screen. In order to do that, we turn the right eye position and direction into camera coordinates, to then have a line equation that we can use to find the intersection with the screen. We then have the equation of the line:

Here (x_0, y_0, z_0) is the right eye position and (dx, dy, dz) is the direction vector.

$$x = x_0 + t \cdot dx$$

$$y = y_0 + t \cdot dy$$

$$z = z_0 + t \cdot dz$$

Solving for t when $z = 0$:

$$t = -\frac{z_0}{dz}$$

We then simply plug this value into the line equation to get the intersection point. At that point, we had a working version of the gaze point estimation. However, as we will detail it in section 3.2.2, we realized that we still did not have the correct values in terms of screen position. We made the conclusion that the issue was coming from the projection function that we used.

3.2.2 Projection

Once that we got an estimate of the gaze point, we need to project it on the screen. Like we explained in section 3.2.1, we first tried to use the built-in function from the scene.

Using built-in functions: At first, we tried to use the built-in function `projectPoint`[9] from the scene. The problem with this function, is that the projection did not behave as

[9]https://developer.apple.com/documentation/scenekit/scnscenerenderer/1524089-projectpoint

we thought it would. Indeed, what this function does is take a point in the real world and project it on the plane that the camera captures. But in our case, we need to project the point on the physical screen. You can see the differences on figure 1. To solve this issue, we had to make our own projection functions.

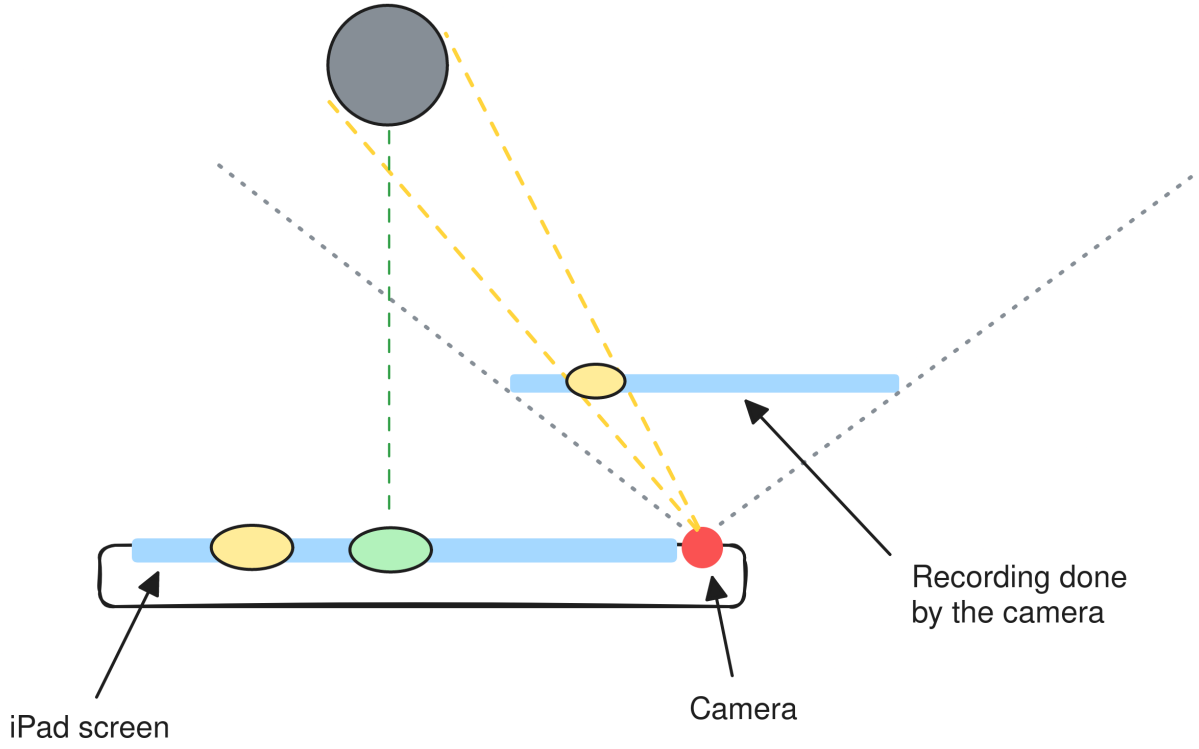

Figure 1: The yellow point is the result of the projection from the `projectPoint` function, while the green point is the type of projection we wanted to achieve.

Using custom functions: When we implemented our own projection function, we already had our own function for the gaze point estimation. This actually had the advantage of having the intersection already being "on the screen". Since this was the case, we had to convert its real-life position into screen coordinates.

The first thing to do is assess the type of measurement that we are working with. For the real-life measurements, `ARKit` gives us the measurements in meters. For the final result, we needed to have the value in points. The way to go from real-life measurements to points is to first go through pixels. To do this, the PPI (pixel per inch) of the device that we are using is needed. This is not something that is built-in within the frameworks that we use. Fortunately, this is something that is easily found on the internet. What we did was then to have a function that checks the device ID, which we can get from the device itself, and then return the PPI corresponding to that device. With this, we were able to turn real-life measurements into pixels. However, since in `ARKit` the measurements are in meters, we first have to convert those into inches. Now we had our real-life point

in pixels, but in order to compute the coordinate of a point on the screen we have to turn it into points. This was also pretty straightforward, as we also have a built-in value that gives us the scale from pixel to points. Finally, we still have to shift the point by half the screen size. The reason for this is that the origin of the screen is in the top left corner, while the camera is placed on the middle left side of the screen.

3.3 Data Gathering

Now that we had an implementation that came relatively close to our unit tests, and that also was showing a point that was in the area of where we were looking at on the prototype, we needed to gather data to calibrate the application.

We decided to separate the screen into nine points: the four corners, the four sides and the middle. When the person is looking at the point, the application takes the measurements. The update of the values is done once per frame, but the application only saves the last ten values. Each point appears one by one, and the user has to look at each of them for a certain amount of time. This process is repeated for five different head positions: middle, top, down, left and right. The user is also required to be at two different distances from the screen: regular and close. At the start, we also wanted to have the user at the maximum arm length, but the values were way off from what we expected (see figure 2). So, at one point, we decided to remove this position from the data gathering.

The measurements that we save are the following:

- The head position, so middle, top, down, left and right.
- The distance from the screen, so regular or close.
- The timestamp of the measurement, to uniquely identify it.
- The target point that the user is looking at.
- The estimated gaze point that we computed.
- The transform of the face.
- The transform of the right eye.
- The transform of the left eye.
- The `lookAtPoint` property (even tough we do not use it, we still added it to the data).

For each point, we also saved a username and the type of device that was used. This information would be useful if we wanted to test the measurement in different settings or other devices with the same user.

What we concluded from this data gathering will be discussed in section 4.

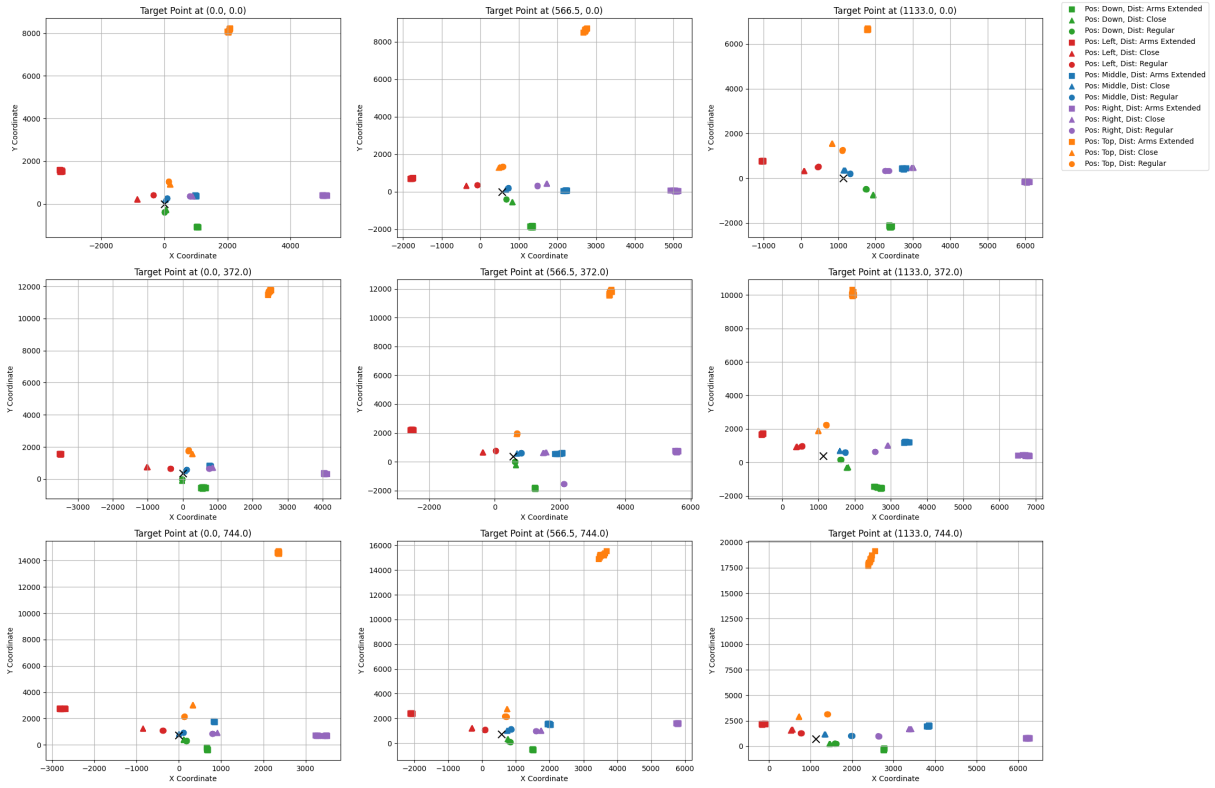

Figure 2: We can see on this picture how much further the values are when the data was gathered with the arms extended.

3.4 Calibration

Now that we had a computation for the gaze point, we needed to calibrate it. In order to do that, we created what we called "calibrators". When they are created, they are given all the calibration data that we gathered. We tried two types of calibrator, one that uses average offsets and one that uses linear regression.

3.4.1 Offset calibration

The first and most straightforward calibration that we tried was to use the average offset. When an instance of the calibrator is created, it is given all the calibration data, computes the offset for each point and then computes the average offset. To then calibrate a given point, this calibrator subtracts the average offset to the estimated gaze point.

Since the application is meant to be used with the face facing forward most of the time, we made another version where we gave more weight to the offset that were taken while facing forward. To compute this "face forwardness", we simply did a dot product between the face normal and the camera direction. This would result in values close to one when looking straight at the camera and close to zero when looking at the sides.

3.4.2 Linear regression calibration

The second type of calibration that we tried was to use linear regression. To train this model, we always gave him our previously computed gaze point. Then we also created different models that would use different features. The features that we used are the following:

- The face transform.
- The right-eye transform.
- The roll, pitch, and yaw of the face (also known as the Euler angles).
- The `lookAtPoint` property.

To then train and evaluate the models, we used two different split methods. The first one is a random split. For this one, we just made sure that, in the train and test splits, the target points were evenly distributed. Since we had nine different target points, we did so that the test split was thirty percent of the data. The second split method that we used was to train on the corners and the middle, and test on the sides.

As we said in section 3.3, we gathered ten gaze points per target points. Once we had to train the data, we did two different versions. The first one was to use all those points as is. This meant we had nine-hundred points of calibration data. The second one was to take the median of those ten points. This would give us ninety points of calibration data. The reason we did this is that we wanted to precompute some data for the model and to remove outliers from the calibration data.

4 Results and Discussion

4.1 Data Gathering

After the first round of data gathering, we plotted the results. The first thing that jumps out is how much the orientation of the face has an impact, see figure 3, despite all this values having the eye actually looking at the same point. This is from this observation that stems the idea to have the calibrators use the face transform is some way during their setup.

Like we mention in section 3.3, the other thing that we realized is how inaccurate the values became when the user had his arm extended. Since the goal of this eye tracking is to be integrated into Dyslexico, we made the decision to remove this position from the data gathering, the reason being that, when using the app, the iPad mostly sit on the table or is held relatively close to the face. As a side effect, by removing it, we were able to reduce the time it took for a user to do the calibration.

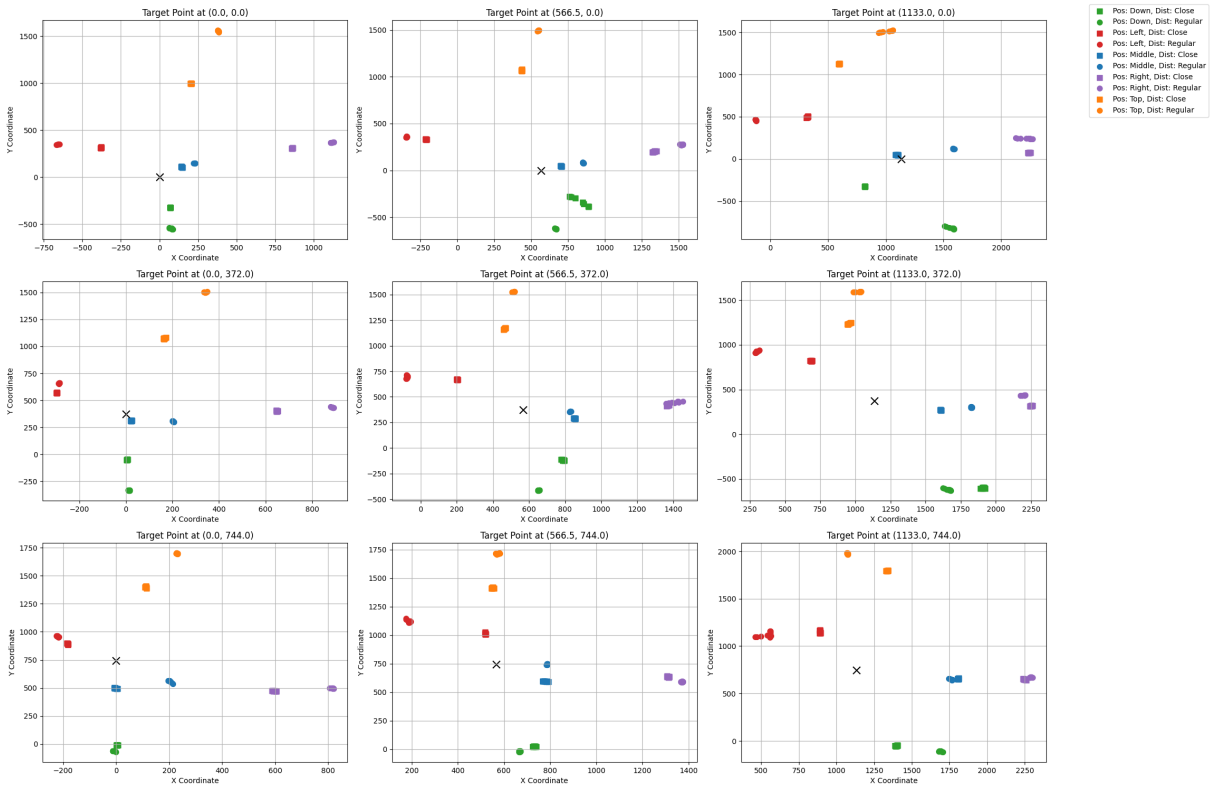

Figure 3: We can see on this figure that the face orientation impacts a lot the position of the estimated gaze point.

4.2 Calibration

The exploration of the calibration was done using Python[10] on Jupyter notebooks[11].

Let's first talk about the metrics we used to compare the different calibrators. We used the Mean Squared Error (MSE), the Root Mean Squared Error (RMSE) and the R-squared value (R^2). The MSE and RMSE are used to see how far off the calibration is from the actual value. For the rest of this section, we will mostly use the RMSE, as it is the most intuitive value, and that the MSE is simply the square of the RMSE. The R^2 value is used to see how well the calibration explains the variance of the data. This value is between 0 and 1, with 1 being the best possible value.

For the following values and graphs, the device that was used is an iPad Mini 6th generation, and its screen size, in points, which is half the size in pixels, is 1133×744.

Let's first talk about those metrics before we did any calibration. What we got was a RMSE of 580.48, which is quite significant. The R^2 value is -1.73. It being negative means that the calibration does not explain the variance of the data at all. This is most likely due to the impact of the face orientation that we talked about in the previous section.

[10]https://www.python.org/
[11]https://jupyter.org/

4.2.1 Offset calibration

For the average offset and the weighted offset, the values are show in table 1. As we can see, the RMSE decreased, but is still quite high, and we still have a negative R^2 value.

Calibrator	RMSE	R^2
Average Offset	556.97	-1.53
Weighted Average Offset	557.41	-1.54

Table 1: Metrics for the offset calibrators.

4.2.2 Linear regression calibration

Here we are going to talk about the result of the different setups from that we mentioned in section 3.4.2.

Random split: At first, let's talk about the result for the random splits.

We ran the split ten times, and we are going to show the average value and the standard deviation. In table 2, we can see the results for the different features using all the data. You can also look at figure 4 to better visualize the results.

Features	RMSE	R^2
Just Computed Gaze Point	309.05 ± 6.27	0.30 ± 0.02
Face Transform	160.30 ± 3.20	0.84 ± 0.01
Roll Pitch Yaw	199.55 ± 3.50	0.70 ± 0.01
Face Transform and Roll Pitch Yaw	155.52 ± 3.40	0.85 ± 0.01
Face and Right Eye Transforms	126.71 ± 3.94	0.89 ± 0.01
All but `lookAtPoint`	114.88 ± 3.35	0.91 ± 0.00
All Features	101.24 ± 3.90	0.93 ± 0.00

Table 2: Comparison of model performances with different features on all data.

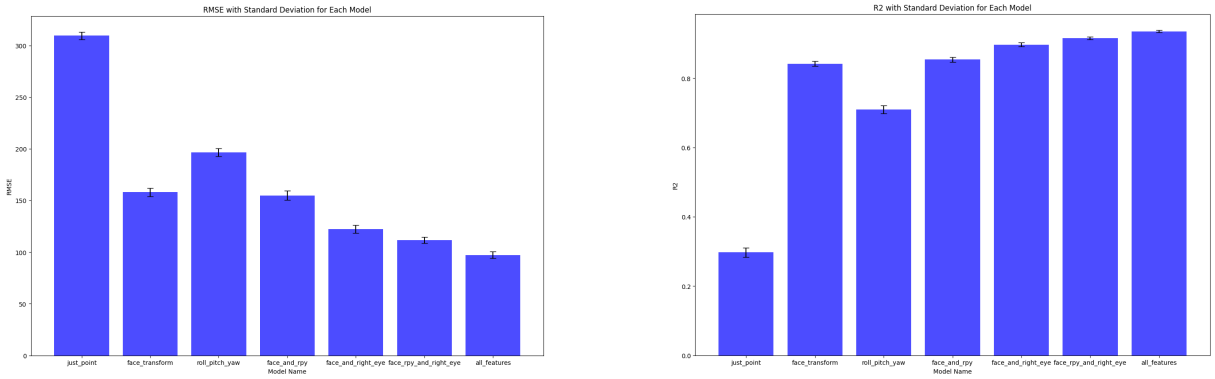

Figure 4: RMSE and R^2 for the different features with random splits on all data.

For the second version of the random split, like we mentioned in section 3.4.2, we used the median of the ten points as the grouped data. You can view the result on table 3, and on its corresponding figure 5. While running this experiment, we noticed that the standard deviation was quite high, and that the values were changing a lot between to runs. So, for this setup, we ran the split a thousand times.

Features	RMSE	R^2
Just Computed Gaze Point	318.16 ± 20.79	0.27 ± 0.08
Face Transform	203.37 ± 49.21	0.74 ± 0.17
Roll Pitch Yaw	213.60 ± 18.01	0.66 ± 0.06
Face Transform and Roll Pitch Yaw	210.41 ± 54.20	0.73 ± 0.17
Face and Right Eye Transforms	197.84 ± 51.99	0.74 ± 0.21
All but `lookAtPoint`	189.79 ± 100.17	0.76 ± 0.60
All Features	184.53 ± 79.26	0.77 ± 0.34

Table 3: Comparison of model performances with different features on the grouped data.

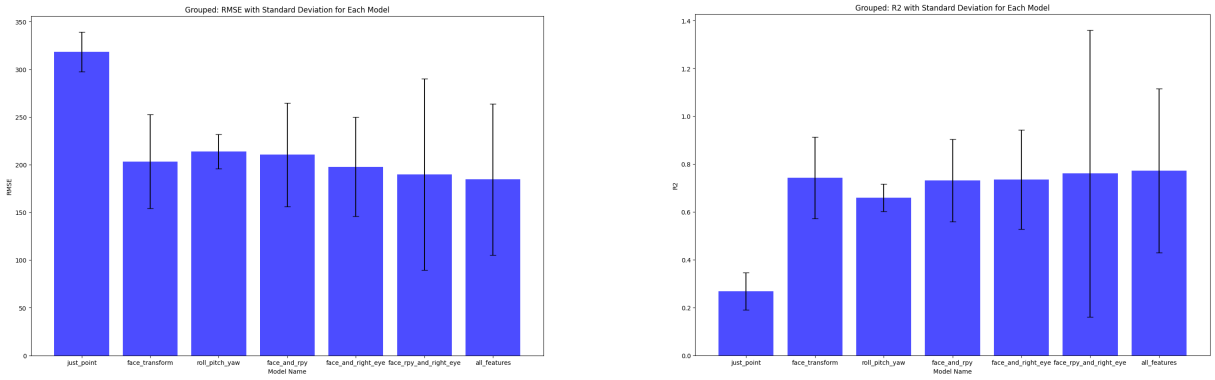

Figure 5: RMSE and R^2 for the different features with random splits on grouped data.

Now let's analyze the results. For both of those setups, we can see a significant improvement compared to the offset calibrators. We can also see that, most of the time, the more features we used, the better the result. Another interesting point is that, despite encoding the same idea, the roll pitch yaw values did not perform as well as the face transform when used alone.

If we compare the two setups, we saw the following. First, the "all data setup" scored better than the grouped data. This actually makes sense since we had a lot more points to train on. One issue, however, is that this group might be prone to more overfitting. This is something we unfortunately did not have time to dive into. We also see that their standard deviation was smaller, so the split did not matter as much. It also makes sense as, since there were more points, the data was more evenly distributed.

Corners and middle: Let's now talk about the results for the setup where we trained on the corners and the middle, and tested on the sides.

Like before, we first ran the test on all the data. The results are shown in table 4, and in figure 6.

Features	RMSE	R^2
Just Computed Gaze Point	272.33	0.28
Face Transform	168.71	0.75
Roll Pitch Yaw	176.41	0.68
Face Transform and Roll Pitch Yaw	174.64	0.72
Face and Right Eye Transforms	170.13	0.75
All but `lookAtPoint`	162.31	0.75
All Features	181.70	0.72

Table 4: Comparison of model performances with the corner and middle for training and the sides for testing, on all data.

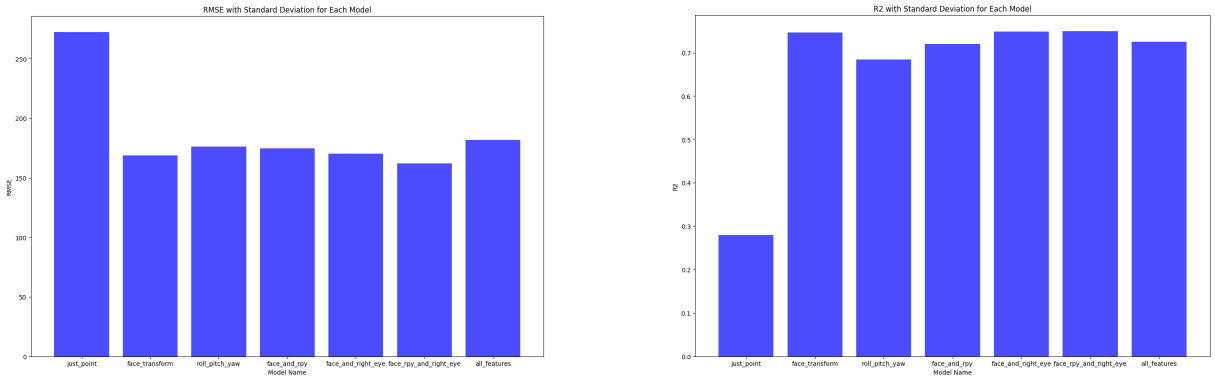

Figure 6: RMSE and R^2 for the different features with the corner and middle for training and the sides for testing, on all data.

Then, for the grouped data, the results are shown in table 5, and in figure 7.

Features	RMSE	R^2
Just Computed Gaze Point	272.23	0.28
Face Transform	172.84	0.74
Roll Pitch Yaw	175.78	0.69
Face Transform and Roll Pitch Yaw	179.18	0.71
Face and Right Eye Transforms	187.04	0.70
All but `lookAtPoint`	168.71	0.73
All Features	181.70	0.72

Table 5: Comparison of model performances with the corner and middle for training and the sides for testing, on grouped data.

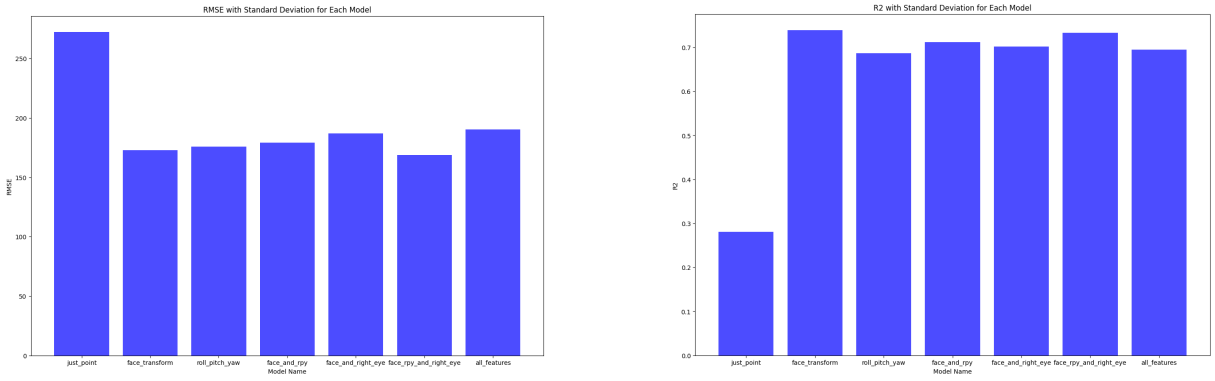

Figure 7: RMSE and R^2 for the different features with the corner and middle for training and the sides for testing, on the grouped data.

Now for the analysis. For both setups, we can see that R^2 is quite high. This means that the calibration explains the variance of the data quite well. For each feature selection, the RMSE is quite close to each other. It is also interesting to see that the best combination of features is the same for both setups, namely the face transform, the roll pitch yaw and the right eye transform (so all but the `lookAtPoint`).

For the difference between the all data and the grouped data, we can see that it is not as significant as for the random split. The reason for this is that, since we only used the corners and the middle, the data was more evenly distributed.

Finally, this split variation performed worse than the random split on all data, but better on the grouped data.

Best model: Based on this values, we can argue that the best model is the one with the lowest RMSE, while still having a high R^2 value. In that case, it would be the random split on all data with all features. Its RMSE was 94.51, and its R^2 was 0.94. You can see the result of the calibration on its train set for each target points on figure 8.

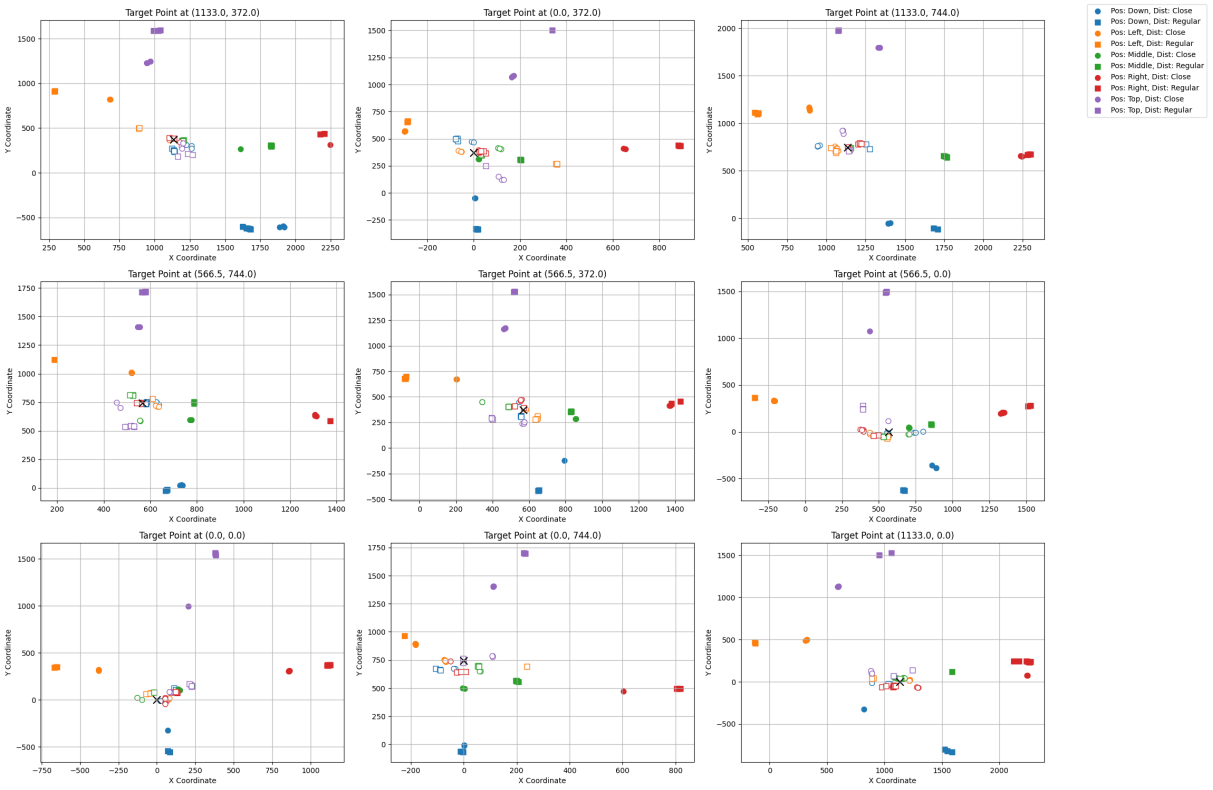

Figure 8: Calibrator that had the lowest RMSE on the test set, using the random split on all data with all features.

4.3 Future Work

More could be done to extend the work of this project. Here are some future work ideas that we could explore. First, we could try, before calibration to find a way to remove the angle from the face. This could help as we saw that the eye tracking was quite dependent on the face orientation.

Also, once we had our own gaze estimation function, we didn't try to use the `lookAtPoint` property (outside the linear regression). Maybe we could try to integrate it again with our custom projection function.

For the calibration, it would be good to have been able to create a random gaze point on the go. That way, we could have trained on the data we gathered, and tested on a random point.

The result of this paper were computed only using the calibration data of one user. We already gathered data from other person, so it would be a good idea to see if the values are different between users. If we were to see that those values are quite similar between one another, we could imagine training a model on a batch of users and use this model as the calibrator. This could remove the need for a user to do the calibration before using the app, or at list be used as a starting point, before fine-tuning the calibration with the user's data. In the same vein, it would also be interesting to see the difference between devices,

or on the same device but in other setups, with varying light conditions for example.

Finally, we could translate the models we did in Python to Swift to be able to test them on the iPad. Then, we could add it the Dyslexico application and see how well it performs in real life.

5 Conclusion

In this project, we explored the feasibility of using iPad-based eye-tracking with the goal of detecting dyslexia. We used AR and eye-tracking methods to estimate the gaze point of the user. Multiple tryouts were done to find the best way to estimate the gaze point. To make our estimation more precise, we tested a variety of calibration methods. This included offset calibration and linear regression calibration. With it, we were able to increase how precise and accurate the estimation was.

We also highlighted the importance of the face orientation in the calibration. The aim of future work would be to reduce its impact even more. This could help to have a better first estimation of the gaze point. Other calibration methods could also be tested to see if we could get better results.

With this project, we created a base for future research on eye-tracking on iPad. We prototyped a way to estimate the gaze point of the user and to calibrate it, that could be extended to other applications. The exploration we made on the different calibration methods could be a staring point to find new possibilities. More work could be done to see if this could be used in a real-world application, such as Dyslexico.

References

[1] Richard K Wagner et al. "The prevalence of dyslexia: A new approach to its estimation". In: *Journal of Learning Disabilities* 53.5 (2020), pp. 354–365.

[2] Leon Franzen, Zoey Stark, and Aaron P Johnson. "Individuals with dyslexia use a different visual sampling strategy to read text". In: *Scientific reports* 11.1 (2021), p. 6449.

[3] Michel Habib. "The Neurological Basis of Developmental Dyslexia and Related Disorders: A Reappraisal of the Temporal Hypothesis, Twenty Years on". In: *Brain Sciences* 11.6 (2021). ISSN: 2076-3425. DOI: 10.3390/brainsci11060708. URL: https://www.mdpi.com/2076-3425/11/6/708.

[4] Benjamin Gagl et al. "Eye movements during text reading align with the rate of speech production". In: *Nature human behaviour* 6.3 (2022), pp. 429–442.

[5] *Diagnostic and statistical manual of mental disorders : DSM-5™.* eng. 5th edition. Washington, DC ; American Psychiatric Publishing, a division of American Psychiatric Association, 2013 - 2013. ISBN: 9780890425541.