Technion
Israel Institute
of Technology

Faculty of Biomedical
Engineering

ML Based Eye Closure Estimation Algorithm, as a Foundation for Bell's Palsy Treatment

Submitted by:

Tom Mendel 204708812

Oren Shapira 204662449

Project Supervisor:

Dr. Dan Rappaport

Project Administrator:

Dr. Janna Tenenbaum-Katan

Submission Date:

February 2019

Table of Contents

0. INTRODUCTION

0.1. Background

Bell's Palsy is characterized by a partial or complete paralysis of one side of the face. Common symptoms include difficulties with eating, drinking and creating certain facial expressions, but the most common ones are related to eye dynamic restrictions, preventing patients from being able to completely close the eye, or blink properly.

Many patients will recover over time, but full recovery sometimes takes up to 12 months. In addition, there may be side effects, most of them are related to eye closure. This is a dangerous situation, since leaving the cornea dry can lead to pain, eye infections and, in extreme cases, blindness.

Every year, about 2.5 million people are getting paralyzed, while 8-10% of them will experience another similar phenomenon in their lifetime. The most vulnerable population is ages 15 to 45, so there is a great value for making the rehabilitation period as quickly as possible.

Existing solutions mainly focus on relieving the symptoms, using eye drops or covering the eye with patch while sleeping to avoid it from being vulnerable to infections. There are currently no non-invasive solutions that are effective in the early stages of paralysis.

0.2. Previous Studies

Many studies from recent years ([1], [2]) attempted to achieve eye full closure by an electrical stimulation to Bell's Palsy patients. The studies discussed the optimal place to locate the relevant electrodes, as well as the intensity and the frequency that will lead to an effective and tolerable stimulus. Studies also show that electrical stimulation does improve the functioning of patients with severe paralysis.

Based on that knowledge, we proposed an idea for a system that uses a glasses-based device that allows blinking detection in the healthy eye, which is used to initiate the electrical stimulation to the paralyzed eye and force it to blink by reaching a full closure.

In addition, the device will estimate the blink quality in the paralyzed eye and will provide an adaptive feedback to the closed loop of the electrical stimulation parameters in order to optimize its closure. We want to activate a minimal effective stimulation of the paralyzed eye that will let it close completely, in order to cause the muscle to "strain" and hopefully achieve faster rehabilitation.

Many studies show different methods for identifying blinking, a common method is the recording of electrical activity (EMG signals) from the Orbicularis Oculi muscle, which has been identified as participating in the blink function ([3], [4]). The method has good results, but involves the adhesion of electrodes to both the healthy and paralyzed eyes, and suffers from cross-talk problems between stimulation and measurement electrodes.

Another innovative method is based on measuring the reflectance of infrared signals [5], which overcomes the disadvantages of the method presented before, but seems to be less precise and consumes much energy.

Additionally, attempts were made to detect blinking by cameras and image processing [6]. Most of the applications are in the field of detecting fatigue and lack of drivers' concentration on the road and therefore do not quantify the eye closure amount [7].

Since there are lots of studies about the stimulation field, our project will focus on detecting eye closure amount for blink detection and adaptive feedback developing. We chose to focus on the research and development of a camera-based solution. This is a non-invasive approach that can detect blinks and estimate their quality by means of eye closure amount, which is essential for the development of an effective adaptive control system.

0.3. The Project's Goal

The most significant and innovative part of the suggested solution is the estimation of eye closure amount and addition of an adaptive control, whose goal is to adjust the paralyzed eye's stimulation level based on the evaluation of its reaction. An optimal stimulation would therefore reach full eye closure, letting the paralyzed eye function normally during the disease period and shortening the rehabilitation period.

The project's goal is to develop an algorithm for eye closure estimation, based on a visual input from cameras and the processing it using advanced techniques in the fields of image processing, computer vision and machine learning.

The project's products will be an essential infrastructure for future development of the complete system, which will include an adaptive control loop that enable optimal electrical stimulation to in order to achieve complete closure of the paralyzed eye.

0.4. The Research Question

Does using advanced methods of **machine learning**, applied on a **visual input** from video cameras, apply for a reliable estimation of **human eye's closure amount**?

0.5. Project's Milestones

The following graph shows the project's milestones:

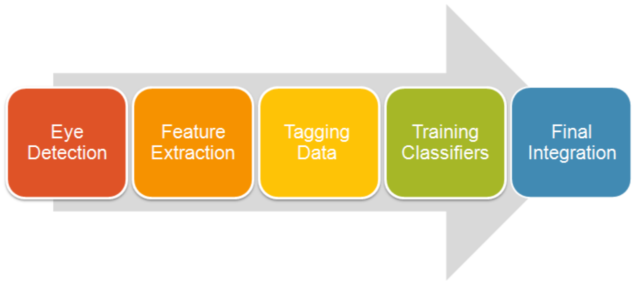


Figure 0-1: The project milestones

We will briefly explain each of the different sections:

1. **Eye Detection:** We will review some state-of-the-art techniques for face and eye detection and will use them to locate faces in a frame and perform further processing to find facial landmarks.

   The desired product of this stage is an array of coordinates that represent the location and shape of each eye.

2. **Feature Extraction:** We will use the facial landmarks' coordinates to design smart features, extract them and analyze their correlation to eye dynamics. We will also develop a method to normalize the features, making them invariant to a variety of angles and distances.

   The desired product of this stage is a set of normalized features that are highly correlative to eye dynamics.

3. **Tagging Data:** We will suggest a dedicated scale for tagging the closure amount and develop a convenient, semi-automatic GUI for the tagging procedure. Using the GUI, we will perform subjective tagging of eye closure amount for each eye in each frame.

   The desired product of this stage is the mentioned GUI, as well as the suggested scale.

4. **Training Classifiers:** We will gather all the data collected from the previous sections to create a labelled dataset which is essential for supervised learning and use machine learning methods to train classifiers and evaluate their performance on a new data in comparison to the ground truth (i.e. labels).

   The desired product of this stage is the generated dataset and a trained classifier which will, hopefully, manage to classify well new data.

5. **Final Integration:** We will integrate the work done in the previous sections, design a comprehensive algorithm and implement it using a graphical user interface (GUI) that allows simple and convenient operation by the user.

   The desired product of this stage is a final product that meets the project's goal.

1. EYE DETECTION

1.1. Introduction

In this section, we review some state-of-the-art techniques for eye detection. During the following sections, we are about to use these methods to extract features that would help us in detecting a blink from the healthy eye and, respectively, the amount of closure of the paralyzed eye.

We use dlib and OpenCV libraries, both embedded in Python. These libraries include a variety of functions which are dedicated to image processing and machine learning.

Our method starts with finding the bounding box of a face, serving as an input for further processing by finding facial landmarks in it.

1.2. Face Detection

We chose to use the standard face detector implemented in dlib. This face detector is made using the now classic Histogram of Oriented Gradients (HOG) feature combined with a linear classifier, an image pyramid and sliding window detection scheme. This type of object detector is fairly general and capable of detecting many types of semi-rigid objects in addition to human faces.

To understand the basic principles and the effectiveness of this detector, we describe each component briefly.

1.2.1. Histogram of Oriented Gradients (HOG)

In the HOG feature descriptor, the distribution (histograms) of directions of gradients (oriented gradients) are used as features. The X and Y derivatives of an image are useful because the magnitude of gradients is large around edges and corners (regions of abrupt intensity changes). Edges and corners pack in a lot more information about object shape than flat regions.

The calculation starts with some preprocessing. The image is cropped into blocks of 64x128 pixels, which then split into 8x8 cells. For each pixel within a cell, the gradient's magnitude and orientation are calculated and then collected into a 9-bin histogram. The histogram is then normalized also using the values from the surrounding 8x8 cells, giving a total of 36x1 normalized vector, describing a region of 16x16 pixels.

An example is shown in Figure 1-1 and Figure 1-2.

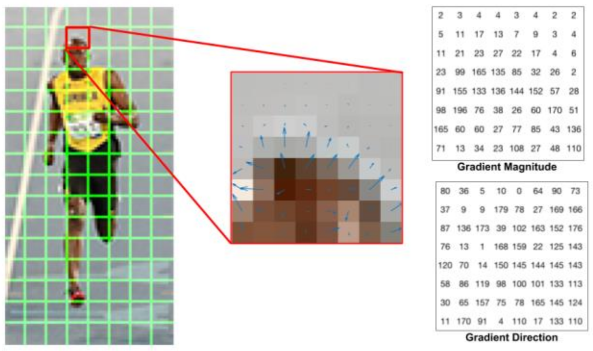
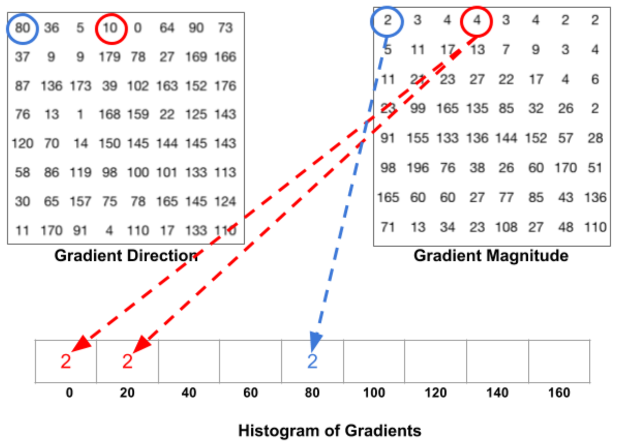


Figure 1-1: Calculation of gradient magnitude and direction using an 8x8 cell



Gradient Direction

80	36	5	10	0	64	90	73
37	9	9	179	78	27	169	166
87	136	173	39	102	163	152	176
76	13	1	168	159	22	125	143
120	70	14	150	145	144	145	143
58	86	119	98	100	101	133	113
30	65	157	75	78	165	145	124
11	170	91	4	110	17	133	110

Gradient Magnitude

2	3	4	4	3	4	2	2
5	11	17	13	7	9	3	4
11	21	23	27	22	17	4	6
23	99	165	135	85	32	26	2
91	155	133	136	144	152	57	28
98	196	76	38	26	60	170	51
165	60	60	27	77	85	43	136
71	13	34	23	108	27	48	110

Figure 1-2: Building up the histogram of gradients

A bin is selected based on the direction, and the vote is selected based on the magnitude. The pixel encircled in blue has an angle of 80 degrees and magnitude of 2. So, it adds 2 to the 5th bin. The gradient at the pixel encircled using red has an angle of 10 degrees and magnitude of 4. Since 10 degrees is half way between 0 and 20, the vote by the pixel splits evenly into the two bins.

1.2.2. Support Vector Machine (SVM)

"Support Vector Machine" (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. However, it is mostly used in classification problems. In this algorithm, we plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane that differentiate the two classes very well.

Support vectors are the data points that lie closest to the decision surface or hyperplane. These are the data points which are the most difficult to classify. They have direct bearing on the optimum location of the decision surface. A two-dimensional demonstration is shown in Figure 1-3.

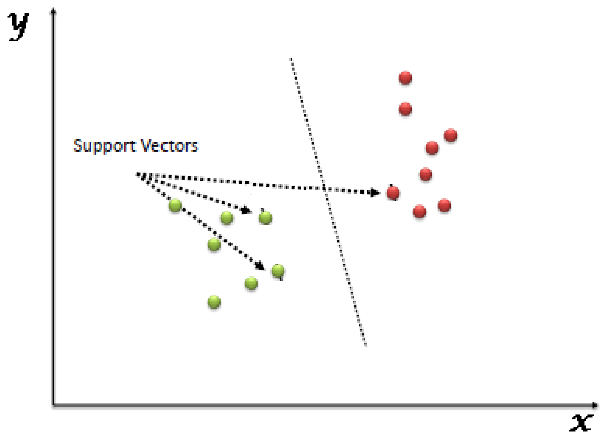


Figure 1-3: Two-dimensional representation of a decision surface and the corresponding support vectors

But how can we identify the "right" hyperplane? In Figure 1-4, we have three hyper-planes (A, B and C) and all are segregating the classes well. Hyper-plane C is the best for the classification task. Here, maximizing the distances between nearest data point (either class) and hyper-plane will help us to decide the right hyper-plane. This distance is called as *Margin*.

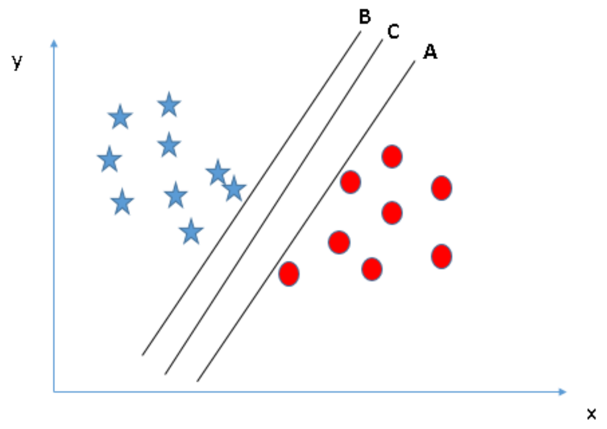


Figure 1-4: Three hyper-planes suitable for classification

The above example shows the problem of finding a linear classifier, but SVM is also suitable in even non-linear cases. In these cases, the cost function isn't a linear combination of the samples, but some kernel (polynomial, exponential etc.)

As for the task of face detection, linear SVM is performed over HOG descriptors extracted from the sample images. The model is trained using pictures of human and non-human objects.

1.2.3. Sliding Window

For each given scale, a sliding window is performed across the image. For each window, the HOG descriptors are computed, and the classifier is applied. If the classifier (incorrectly) classifies a given window as a face, the feature vector is recorded associated with the false-positive patch along with the probability of the classification. This approach is called *hard-negative mining*. In this approach, the false-positive samples are sorted by their confidence and the classifier is re-trained.

The hard-negative mining approach helps reducing the false-positive decisions but detecting multiple bounding boxes around the face you want to detect is very common.

An example is being shown in Figure 1-5. In these cases, non-maximum suppression can be applied to find the bounding box that intersects the most with the other bounding boxes.

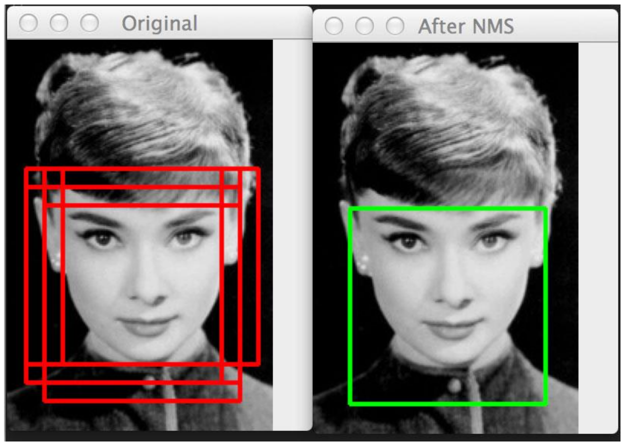


Figure 1-5: (Left) Detecting multiple overlapping bounding boxes around the face we want to detect. (Right) Applying non-maximum suppression to remove the redundant bounding boxes.

1.3. Detecting Facial Landmarks

After the bounding boxes containing faces were extracted from the entire images, we can identify some facial landmarks and regions, including mouth, eyebrows, jaw, nose and, of course, eyes. Once the eyes are located, we can use only the eye regions to detect a blink and the amount of closure of the paralyzed eye. The pre-trained facial landmark detector inside the dlib library is used to estimate the location of 68 (x, y)-coordinates that map to facial structures on the face.

The indexes of the 68 coordinates are visualized in Figure 1-6.

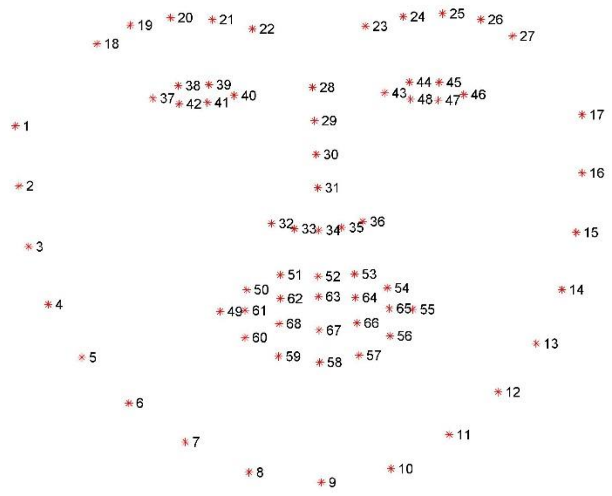


Figure 1-6: The indexes of the 68 coordinates from dlib's facial landmark detector. These annotations are part of the 68-point iBUG 300-W dataset which the dlib facial landmark predictor was trained on. It can be noticed that the indexes 37-42 and 43-48 correspond to the right and left eye, respectively.

The facial landmark detector included in the dlib library is an implementation of *the One Millisecond Face Alignment with an Ensemble of Regression Trees*, paper by Kazemi and Sullivan (2014). [11]

This method starts by using:

1. A training set of labeled facial landmarks on an image. These images are *manually labeled*, specifying specific (x, y)-coordinates of regions surrounding each facial structure.
2. Priors, of more specifically, the probability of distance between pairs of input pixels. Given this training data, an ensemble of regression trees is trained to estimate the facial landmark positions directly from the pixel intensities themselves (i.e., no "feature extraction" is taking place). The result is a facial landmark detector that can be used to detect facial landmarks in real-time with high quality predictions.

Decision tree is a type of supervised learning algorithm (having a pre-defined target variable) that is mostly used in classification problems. It works for both categorical and continuous input and output variables. In this technique, we split the population or sample into two or more homogeneous sets (or sub-populations) based on most significant splitter / differentiator in input variables.

Decision trees are easy to understand and don't require any statistical knowledge to read and interpret them. They are also non-parametric, meaning that no assumptions about the space distribution or the classifier structure should be considered. On the other hand, decision trees often suffer from over-fitting.

Selected results of the facial landmarks detection algorithm on HELEN dataset are given in Figure 1-7.

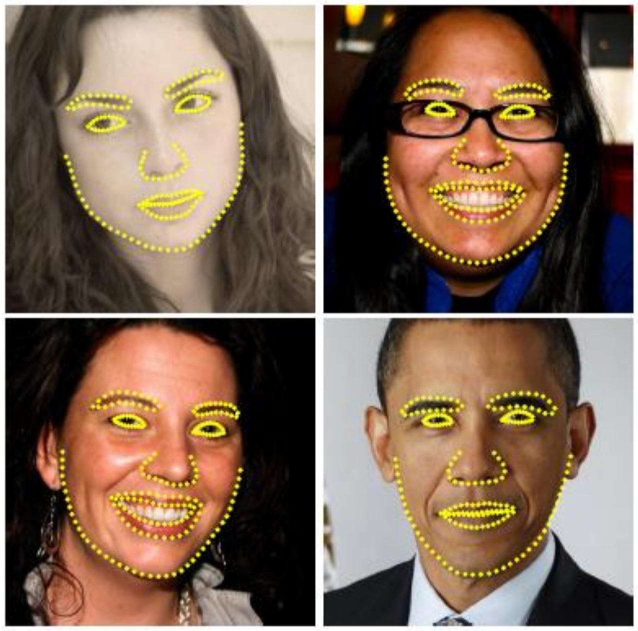


Figure 1-7: Selected results of the landmark algorithm on the HELEN dataset

1.4. Summary

We have demonstrated one way of detecting eye patterns in a convenient and efficient way. The implementation is done in Python, but easy to be adapted for real-time applications.
Now that we have found eye contours, it is easier to find some features that correlate well with eye blink, and specifically, eye amount of closure.

2. FEATURE EXTRACTION

2.1. Introduction

In this section, we try to extract some features that correlate well with the amount of eye closure.

This obstacle is very difficult to overcome. There isn't any work done in this field of research, but many existing researches focus on blink detection.

As for the case of Bell's palsy, only one side of the face is paralyzed, so we can detect blinks from the 'healthy' eye and then calculate the amount of closure of the 'paralyzed' one. Nevertheless, we will show that features extracted for blink detection are sometimes not enough for assessing closure amount and suggest some new features for this task.

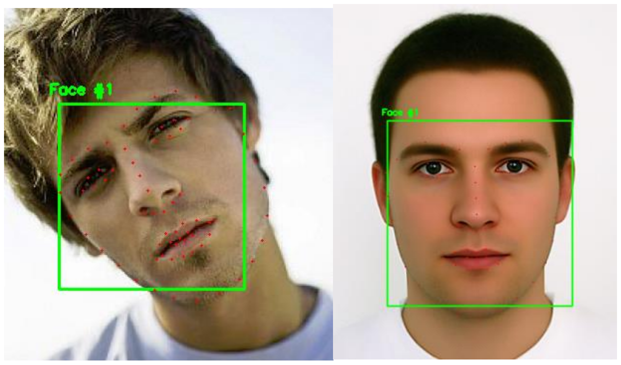


Figure 2-1: Left image shows that the location of the landmarks within the bounding box is variant to changes in orientation (in comparison to the right image)

2.2. The Normalization Problem

We finish stage I by detecting facial landmarks in an image containing face object. Each descriptor contains 68 points, where points 37-42 correspond to the right eye and points 43-48 correspond to the left eye. We are only interested in the eyes, so we start our research with only 12 points.

One can think of using these points as features for future processing. However, these points aren't normalized – crucial for machine learning algorithms.

One problem is the need for rotation invariance. The face detector returns a bounding box parallel to the axes, but in fact the face itself may not be aligned with the image surface. Hence, the relative location of the facial landmarks within the evaluated bounding box could not be determined. This problem is illustrated in Figure 2-1.

To face this problem, we might think of manipulating the coordinates to extract features which are invariant to rotation. One option is to normalize based on scores from previous frames. Assuming that there isn't a lot of motion between adjacent frames, we are able to build a sliding window, saving the results of the latest frames, and then normalize the current score according to the extremum values inside the window.

Before deciding about the normalization method, we might have a look on a patient diagnosed with Bell's palsy

(Figure 2-2).

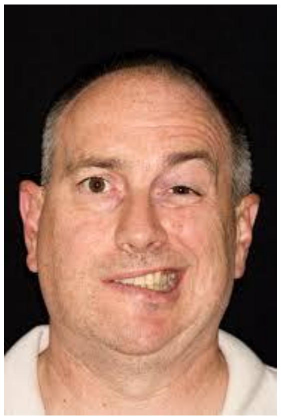


Figure 2-2: A typical patient with Bell's palsy. The 'paralyzed' eye (right) appears to be larger than the healthy, 'blinking' eye (left)

As we can see, even one eye (left eye in this case) manages to close normally, the palsy affects its shape, making it thinner even when the patient isn't blinking.

If we normalize the scores of each eye individually, i.e. based on the previous scores of only one eye, we might have a scale mismatch – as 100% opening of the 'healthy' eye is different than in the case of the 'paralyzed' one. Furthermore, as the 'paralyzed' eye is barely blinking, we would never reach a full closure, i.e., an opening of 0%.

As the 'healthy' eye manages to reach a full closure, we chose to normalize the results of *both* eyes using the minimum value of the 'healthy' eye. The maximum value is still defined individually for both eyes.

To sum up, the normalization formula can be written as:

$$\hat{p} = \frac{p - \mathrm{h}_{min}}{p_{max} - \mathrm{h}_{min}}$$

$$\hat{h} = \frac{h - \mathrm{h}_{min}}{h_{max} - \mathrm{h}_{min}}$$

Where (p, h) and $(\hat{p}, \hat{h})$ correspond to the results of the 'paralyzed' and the 'healthy' eyes before and after normalization, respectively.

2.3. The Sliding Window's Length

Another problem relates to determining the length of the sliding window. As a blink occurs every 2-5 seconds, a window smaller than 5 seconds may not include scores which relate to a full closure. However, a window too long will be more sensitive to scale and orientation.

Therefore, in order to determine whether the sliding window's length should be increased, we must consider two possible conditions:

1) The blink duration is longer than usual; i.e., longer than the sliding window's length (in seconds)
2) The patient isn't blinking for a period that is longer than the sliding window's length (in seconds)

In both cases, to reach a proper normalization we must preserve the extremum values of the sliding window. However, if we reach again an extremum value, the former can be ignored.

To do that, we suggest the following algorithm:

- If the 'earliest' score in the window is higher or lower than most of the scores:
  - If the latest score is also higher of lower than most of the scores:
    - No need to preserve the earliest score; shrink the window's length into the default size
  - Else:
    - The earliest score must be preserved; increase the length by 1
- Else:
  - Do nothing; The window is already in its original size

This method guarantees that the extremum values are considered when needed but ignored when are not necessary.

To make sure that the sliding window doesn't get far larger, we add a hysteresis to the thresholds; i.e., the threshold for increasing is lower than the threshold for shrinking. We chose normalized thresholds of (0.1, 0.9) and (0.2, 0.8) for increasing and shrinking, respectively. Figure 2-3 shows an example of the above technique.

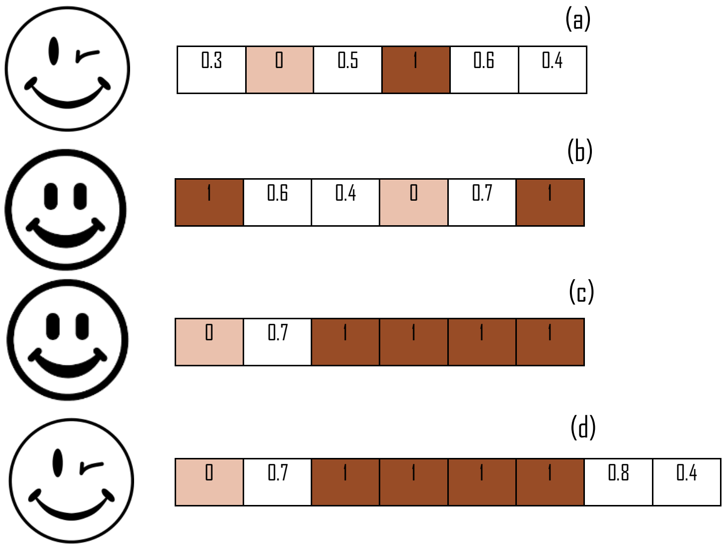


Figure 2-3: An example of a sliding window with an alternating length. The normalized scores relate to the left eye.

(a) The patient blinks – reaching a normalized score of near 0. Because the earliest score isn't close enough to 0 or 1, the length shouldn't be increased.

(b) The patient doesn't blink – reaching a normalized score of near 1. The earliest score is 1 but the latest score is also 1, so the length shouldn't be increased either.

(c) The patient continues to keep his eye open. The earliest score is 0 but the latest score is 1 – the window should be increased to preserve its minimum value.

(d) The patient blinks again. The earliest score is 0 but the latest score is also 0, so the window can be shrunk into its original size.

One clear side effect of using a sliding window is the loss of data extracted from the frames from the beginning of the video, as the normalization starts only when the sliding window reaches its full initial capacity. In order to preserve this data, we added an option for running the video file backwards and stitch the normalized scores to the output file of the original video. Refer to Appendix A for more information and executing instructions.

Now, as we reach a normalization technique, we can suggest some features that might be relevant for the classification of the closure amount.

2.4. Eye Aspect Ratio (EAR)

The term EAR stands for Eye Aspect Ratio. It can be calculated from the six points (from each eye) as the ratio between the major and minor axes:

$$EAR = \frac{||p_2 - p_6|| + ||p_3 - p_5||}{2||p_1 - p_4||}$$

where the points are notated as shown in Figure 2-4. The EAR is mostly constant when an eye is open and is getting close to zero while closing an eye.

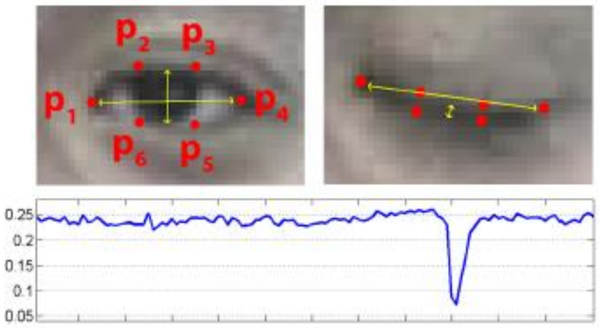
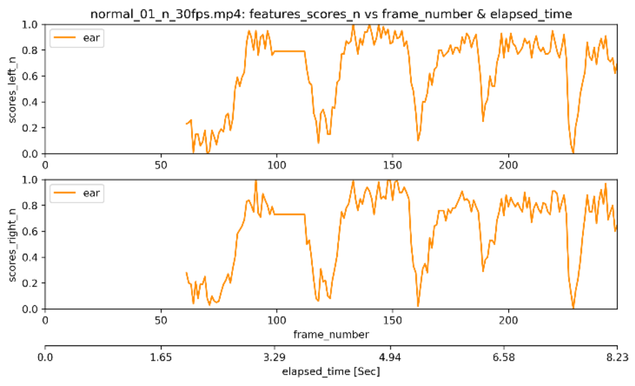


Figure 2-4: Up – notation of the six points for each eye given by the descriptor; Down – EAR score before and after blink detection

As we see, the EAR score correlates well with each blink. However, the scores for an open or closed eye are very similar, so even small amount of noise might affect the results dramatically. Therefore, normalization is necessary.

Figure 2-5 shows the results of the normalized EAR features on normal_01_n_30fps.mp4. This video contains a healthy man blinking. In this case, the results fit subjective tagging.



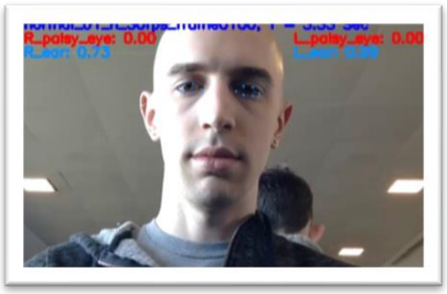


Figure 2-5: Performance of EAR on normal_01_n_30fps.mp4. Subjecting tagging (in seconds): 2.2, 4, 5.5, 6.2, 7.2

Figure 2-6 shows the results of the normalized EAR features on palsy_05_l_25fps.mp4. This video contains a woman with Bell's palsy of the left side of the face. In this case, the results fit subjective tagging.

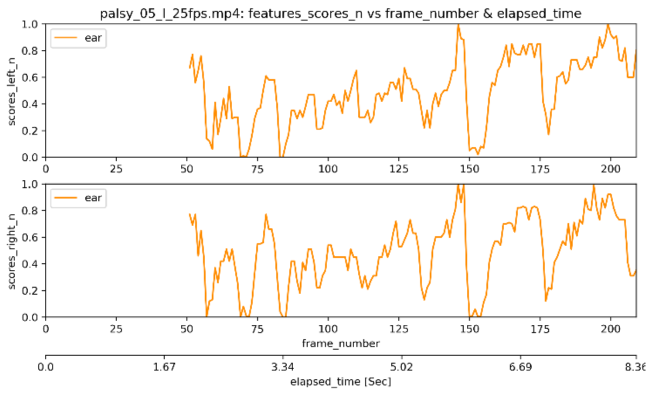


Figure 2-6: Performance of EAR on palsy_05_l_25fps.mp4. Subjecting tagging (in seconds): 2.3, 2.8, 3.3, 3.8, 4.4, 5.4, 6, 7, 8.2

As shown, the 'normalized' EAR metric works well in all cases but might suffer from noise originated in the nature of palsy's symptoms. One clear benefit is the computation simplicity, which is crucial for real-time applications. We continue the research by trying different features.

2.5. Linear Regression

As seen in figure 2-3, when a blink is reached, we expect p1, p2, p3 and p4 to be, approximately, on a straight line. Therefore, we can try fitting a linear curve to these points and then calculate the R^2 metric. Note that high value (close to 1) corresponds to a blink, so the final score is calculated by calculating $1-R^2$. We also apply the same normalization as before. We use the standard LinearRegression from sklearn library, embedded in python. Figure 2-7 and Figure 2-8 show the performance of this metric on the previous videos.

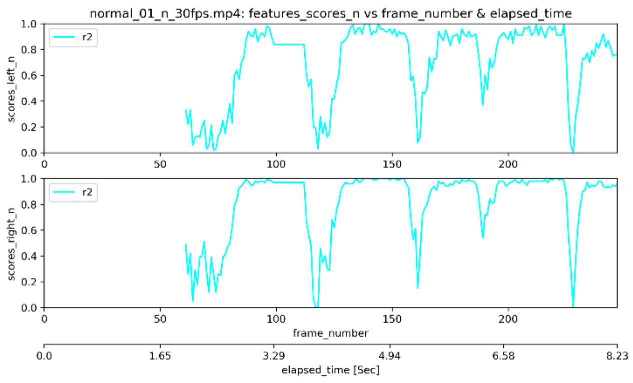


Figure 2-7: Performance of $1-R^2$ on normal_01_n_30fps.mp4. Subjecting tagging (in seconds): 2.2, 4, 5.5, 6.2, 7.2

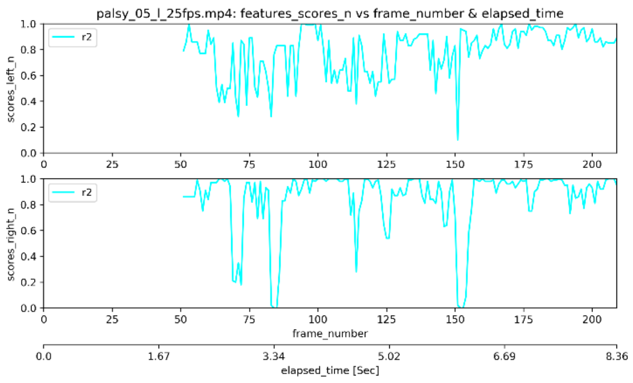


Figure 2-8: Performance of $1-R^2$ on palsy_05_l_25fps.mp4. Subjecting tagging (in seconds): 2.3, 2.8, 3.3, 3.8, 4.4, 5.4, 6, 7, 8.2

As before, this method works very well in the case of 'healthy' eyes. Nevertheless, this method fails on palsy_05_l_25fps.mp4, while having much more sharp 'peaks' than the actual number of blinks. In addition, Figure 2-9 shows that even when a full closure is reached, the detected points often don't fit to a straight line, which might add a significant error to the results.

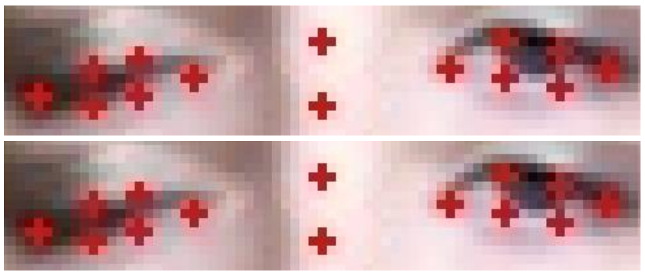


Figure 2-9: Performance of Linear Regression method on normal_01_n_30fps.mp4 (up) and palsy_05_l_25fps.mp4 (down)

Taking that into account, along with the relatively high computation complexity, make this feature inefficient. Therefore, we chose not to continue with that feature.

2.6. Ellipse Area

We continue the research by trying to find an efficient way for assessing the area of the eye. This feature might be very helpful for the task of eye closure estimation – we can calculate the ratio of the current area over the maximum detected area.

We first try to fit the simplest geometric model – an ellipse – to the detected eyes. We can approximate the eye as an ellipse with a major axis (noted as MA) and a minor axis (noted as ma), as shown in Figure 2-10.

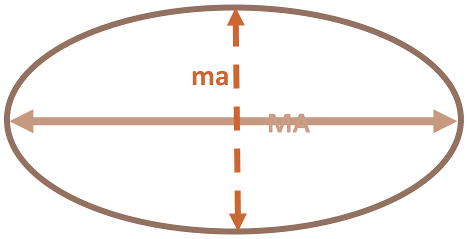


Figure 2-9: Modelling the eye as an ellipse

The area of an ellipse is then easy to calculate:

$$A_{ellipse} = \pi \cdot ma \cdot MA$$

We chose to use the fitEllipse function, embedded in Python as part of OpenCV library. Figure 2-11 and Figure 2-12 show the performance of this metric on the three previous videos.

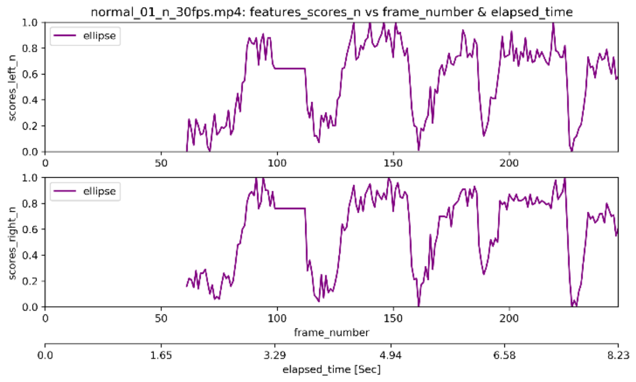


Figure 2-11: Performance of ellipse area on normal_01_n_30fps.mp4. Subjecting tagging (in seconds): 2.2, 4, 5.5, 6.2, 7.2

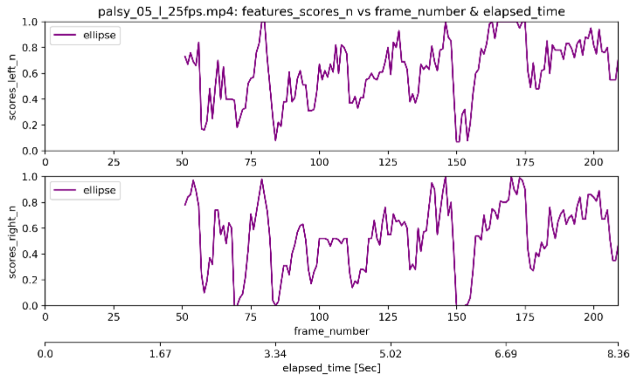


Figure 2-12: Performance of ellipse area on palsy_05_l_25fps.mp4. Subjecting tagging (in seconds): 2.3, 2.8, 3.3, 3.8, 4.4, 5.4, 6, 7, 8.2

Note that the detection is worse than the EAR method. One reason is probably the difficulty in fitting an ellipse when the eye reaches a full closure. A typical case is shown in Figure 2-13.

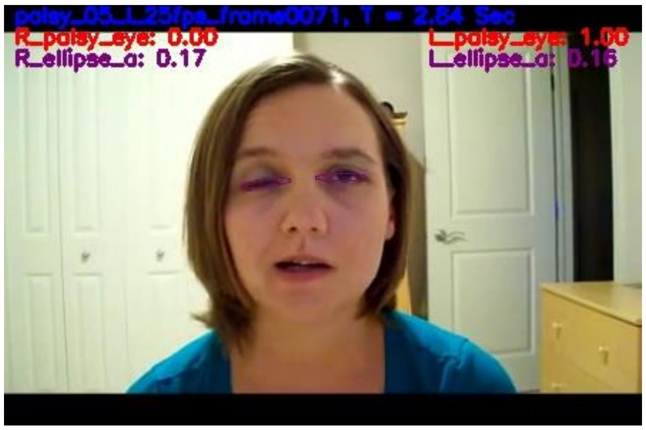


Figure 2-13: An example of ellipse mismatch

2.7. Shoelace Formula

A different approach for assessing the eye's area is treating the eye as a general hexagon, instead of a 'standard', easy-to-calculate shape.

The shoelace formula or shoelace algorithm is a mathematical algorithm to determine the area of a simple polygon whose vertices are described by their Cartesian coordinates in the plane. The user cross-multiplies corresponding coordinates to find the area encompassing the polygon and subtracts it from the surrounding polygon to find the area of the polygon within. The formula can be represented by the expression:

$$A = \frac{1}{2}\left|\sum_{i=1}^{n-1} x_i y_{i+1} + x_n y_1 - \sum_{i=1}^{n-1} x_{i+1} y_i - x_1 y_n\right|$$

In the case of a hexagon, the formula can be simplified as:

$$A = \frac{1}{2}|x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + x_3y_4 - x_4y_3 + x_4y_5 - x_5y_4 + x_5y_6 - x_6y_5|$$

Or as a vector multiplication:

$$A = \frac{1}{2}\cdot\left|\begin{bmatrix}x_1\\x_2\\x_3\\x_4\\x_5\\x_6\end{bmatrix}\cdot[y_2 \quad y_3 \quad y_4 \quad y_5 \quad y_6 \quad y_1] - \begin{bmatrix}y_1\\y_2\\y_3\\y_4\\y_5\\y_6\end{bmatrix}\cdot[x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad x_1]\right|$$

$$= \frac{1}{2}\cdot|\vec{x}\cdot roll(\vec{y})^t - \vec{y}\cdot roll(\vec{x})^t|$$

Where $roll(\vec{x})$ stands for a cyclic transformation performed on vector $\vec{x}$. Python's numpy library has special and efficient functions (roll, dot etc.) for dealing with matrices and vectors. Figure 2-14 provides a graphical illustration of the calculation process.

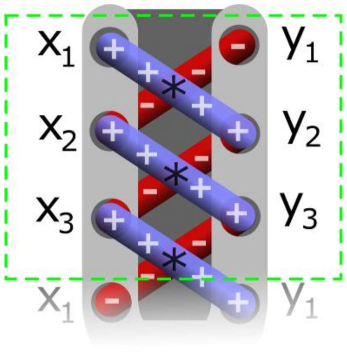


Figure 2-14: Shoelace Algorithm – a graphical illustration

Figure 2-15 and Figure 2-16 shows the performance of this metric on the three previous videos.

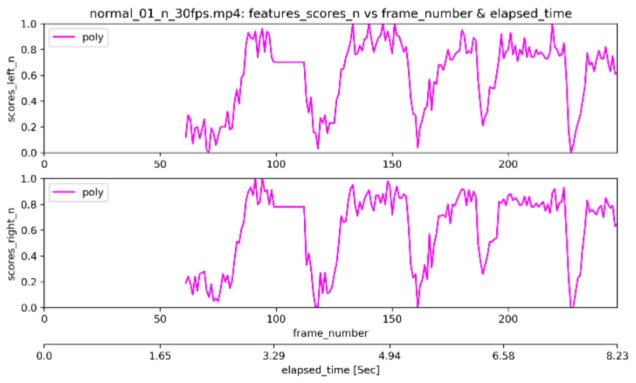


Figure 2-15: Performance of Shoelace formula on normal_01_n_30fps.mp4. Subjecting tagging (in seconds): 2.2, 4, 5.5, 6.2, 7.2

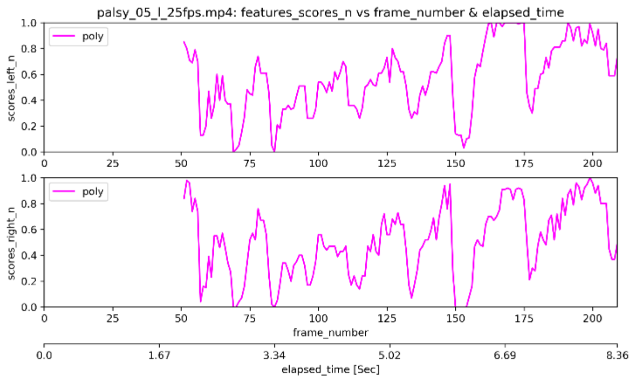


Figure 2-16: Performance of Shoelace formula on palsy_05_l_25fps.mp4. Subjecting tagging (in seconds): 2.3, 2.8, 3.3, 3.8, 4.4, 5.4, 6, 7, 8.2

We can see that the results are quite better than the previous case. Figure 2-17 shows that Shoelace formula, while modeling the eye as a general hexagon and not an ellipse, is more accurate than the ellipse case. It is, however, hard-to-calculate (by means of computational complexity). As the blinks occur every few seconds, we do not have harsh requirements on real-time performance. Hence, we chose to continue with the Shoelace formula feature.

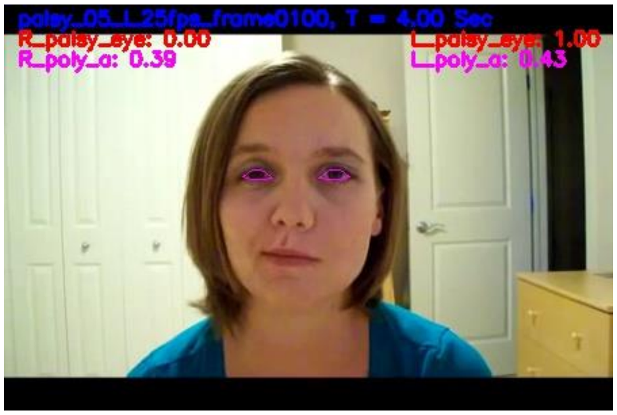


Figure 2-17: Fitting a hexagon to the facial landmarks describing the eyes

2.8. Performance Evaluation

During the work on Section 2, much of the work done was dedicated to improving running times, making the source code appropriate for real-time applications. The code is based on the open-source code from "Learn OpenCV" website, containing basic code for face detection and facial landmarks extraction.

An initial optimization was done, mainly by:

- Avoiding loops
- Taking advantage of Numpy's and OpenCV's built-in functions, including a variety of signal processing and vector manipulation functions that are highly optimized for real-time applications
- Finding more efficient way for frame grabbing (OpenCV's VideoCapture versus Imutils' FileVideoStream)
- Using only the results inside the current sliding window for normalization and shrinking it if possible
- Finding more efficient way for storing data (CSV files versus DataFrames)

And so on. Table 2-1 contains running time statistics (per frame) of the source code after an initial optimization. The code was tested on a laptop with the following specifications:

- Operating System: Windows 10 Home, 64-bit
- CPU: Intel i7-7537U @ 2.5 GHz
- RAM: 8 GB

The tested video was palsy_02_r_30fps.mp4, chosen because it used, on average, the largest sliding window length, making it reach the worst-case results.

Function	Min. time (ms)	Max. time (ms)	Avg. time (ms)	Std.
Frame Grabbing	5.14	50.89	6.73	3.089
Face Detection	201.94	241.98	216.99	19.03
Landmarks Detection	10.98	19.73	12.74	1.871
EAR	0.15	0.70	0.20	0.007
Linear Regression	5.18	11.13	6.15	0.967
Ellipse Area	0.10	0.39	0.12	0.032
Shoelace Formula	0.15	0.40	0.17	0.032
Normalization	0.08	5.35	0.88	0.981
Saving Results	0.02	0.11	0.04	0.010
Total	225.59	458.65	240.70	20.892

Table 2-1: Running times after an initial normalization, tested on palsy_02_r_30fps.mp4

One can clearly notice that the bottleneck of the total running time is the face detection function. This function uses histograms of oriented gradients (HOGs) with multiple scales, making it last much longer than any other part of the source code. To face this problem, we chose to detect faces only at the first frame. The detection of landmarks from the other frames will be based on the bounding box extracted from the frame. Because the videos contain patients who speak straight into the camera, the assumption that the patient's head doesn't move a lot between adjacent frames is quite enough.

In addition, among the four extracted features, we can see that the calculation of the R^2 feature takes about 10 times longer than any other feature. Along with the fact that this feature reached the worst and the 'noisiest' results, we decided not to take this feature into account. Table 2-2 contains running time statistics of the final application.

Function	Min. time (ms)	Max. time (ms)	Avg. time (ms)	Std.
Frame Grabbing	5.14	50.89	6.73	3.089
Landmarks Detection	2.50	11.25	4.26	1.871
EAR	0.15	0.70	0.20	0.007
Ellipse Area	0.10	0.39	0.12	0.032
Shoelace Formula	0.15	0.40	0.17	0.032
Normalization	0.08	5.35	0.88	0.981
Saving Results	0.02	0.11	0.04	0.010
Total	8.14	66.68	12.40	4.781

Table 2-1: Running times of the final application, tested on palsy_02_r_30fps.mp4

We now see the huge improvement to the results, reaching an average processing time of less than 13 milliseconds per frame. As we design the system to deal with cameras with a frame rate of 30 fps or less, and every blink lasts about 300 to 400 milliseconds – much longer than the maximum processing time – we can say that the application can be performed in real-time. We will continue to analyze the source code's performance in the following sections, mainly in Section 5.

2.9. HOG Features

So far, we suggested various features, all of them based on the facial landmarks' classifier.

However, there are cases where this classifier will fail to describe the eye's boundaries, as well as all our suggested features.

For example, let's have a look at palsy_02_r_30fps.mp4. Figure 2-18 illustrates the case where the facial landmarks themselves fail to detect the eyes. Because the features are only based on the coordinates extracted from the facial landmarks classifier, none of them would give accurate results. Figure 2-19 shows the EAR normalized scores according to the extracted landmarks.

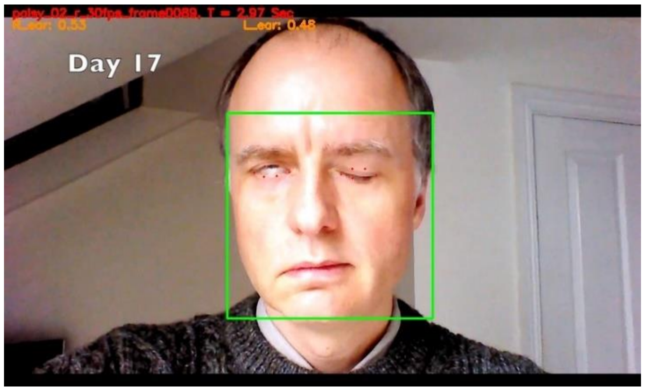


Figure 2-18: Frame taken from palsy_02_r_30fps.mp4; Facial landmarks fail to describe the eyes

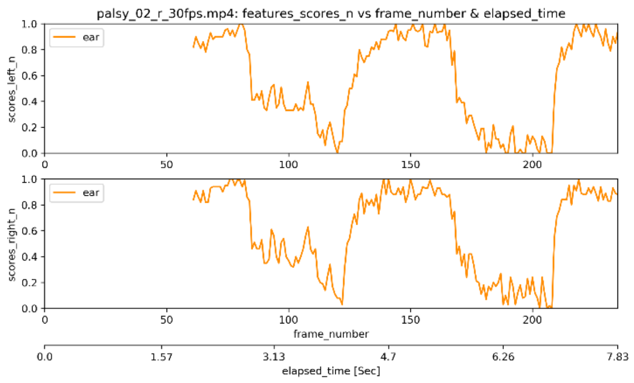


Figure 2-19: Performance of EAR on palsy_02_r_30fps.mp4. The normalized scores of both eyes seem identical though there is a clear visual difference between them

To face this problem, we suggest another type of features, which isn't base directly on the six points returned by the facial landmarks detector – but use them to define the <u>bounding box</u> of the eye, which is used to 'crop' the eyes from the entire picture and use its pixels to establish a feature vector.

The method for cropping the eyes for each frame is given in the following pseudo-code:

1. Detect faces in the given frame
2. Detect facial landmarks for the generated bounding box containing faces
3. For each detected eye:
   3.1. Estimate eye width as $w = |p_{1,x} - p_{4,x}|$ (see Figure 2-4)

3.2. Estimate eye height as $h = \left|\max(p_{2,y}, p_{3,y}) - \min(p_{5,y}, p_{6,y})\right|$ (see Figure 2-4)

3.3. Determine the final eye picture dimensions – use width of a and height of b, reaching the final dimensions of (a x b) pixels

3.4. Calculate N as $N = max\left(\left\lceil\frac{w}{a}\right\rceil, \left\lceil\frac{h}{b}\right\rceil\right)$

3.5. Add pixels from left and right to the bounding box. The number of pixels to be added is calculated by $\left\lceil\frac{Na-w}{2}\right\rceil$ from left side and $Na - w - \left\lceil\frac{Na-w}{2}\right\rceil$ from the right side

3.6. Add pixels from top and bottom to the bounding box. The number of pixels to be added is calculated by $\left\lceil\frac{Nb-h}{2}\right\rceil$ from the top and $Nb - h - \left\lceil\frac{Nb-h}{2}\right\rceil$ from the bottom

3.7. Now as the bounding box size is (Na x Nb), resize by N for each axis to reach the final cropped eye image, with dimensions of (a x b)

3.8. If the current eye is the right eye – flip the given image

3.9. Convert image into grayscale to reach one-dimensional representation.

Note that we scale each eye picture to the same size, in order to extract the same number of features from each picture, to be used for learning. Stage 3.8 is necessary for simplicity – treating both eyes as they were left eyes, in order to avoid biases which are direction-dependent. We use a = 34 and b = 26 for this project, reaching images of 34 x 26 pixels.

We chose to use Histograms of Oriented Gradients (HOGs), as described in detail in the previous section. Song et al [15] used HOG features with SVM to detect blinks and reached accuracies of more than 93%. We hope that those features would be highly effective also for the case of multi-class classification. Note that Song used eye alignment prior to feature extraction, which wasn't done in our case.

The main drawback using this approach is the processing times. Without any feature extraction or machine learning, the cropping of the eye images takes about 150 milliseconds per frame, making it non-relevant for real time applications with high frame rates. However, it is still interesting to evaluate its performance in order to examine the importance of the detector's accuracy. If we got much better results than the other features, we would think of replacing the current detector or improving the performance of the above method.

2.10. Summary

In this section, we dealt with extracting feature which best describe the amount of eye closure. We suggested various methods, all of them based on the facial landmarks' classifier, implemented in Python. We also solved the normalization problem by suggesting an appropriate normalization technique.

However, there are cases where the extracted features will fail to describe the eyes' amount of closure. We suggested a method that is less dependent on the detector's performance; however, it is unlikely to run in real-time.

Therefore, while calculating closure amount, for more accurate estimation we cannot rely on the landmarks only. It might be useful to get some prior information, i.e. some manual tagging, in order to reach an appropriate decision. This is a strong motivation to the aid of machine learning tools.

Figure 2-20 and Figure 2-21 show the scores of all the features mentioned above on palsy_05_l_25fps.mp4 and palsy_08_r_30fps.mp4, respectively. As we can see, the features which are the most accurate are EAR (orange) and Shoelace Formula (pink) methods, Therefore, we will first try these features while fitting a classifier.

`

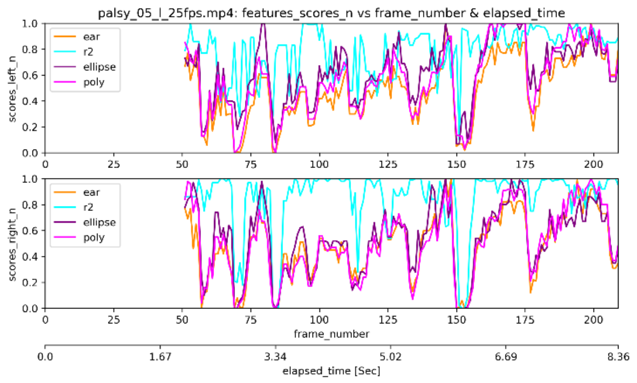


Figure 2-20: Performance of all features on palsy_05_l_25fps.mp4. Subjecting tagging (in seconds): 2.3, 2.8, 3.3, 3.8, 4.4, 5.4, 6, 7, 8.2

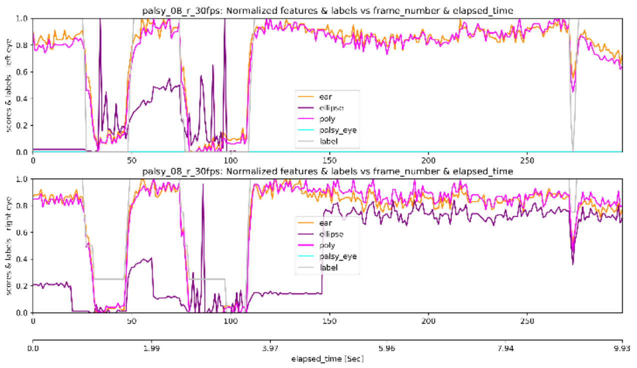


Figure 2-21: Performance of all features on palsy_08_r_30fps.mp4. Subjecting tagging is shown in gray. Here the ellipse area feature fails to describe well the eye's condition.

We end this section by creating Data Frames, data structures embedded in "pandas" library that are very useful in data analysis and machine learning. Each row relates to one frame; each column relates to one feature extracted from that frame.

During the next section, we are about to add one last column to the data frames – the score of subjective tagging. As we apply supervised learning, the information provided to the classifier has to be tagged. We are about to suggest a scale and build a semi-automatic graphic user interface to make the tagging work easier. Once we have the extracted features along with the corresponding tagging, we can apply classification techniques to define the amount of eye closure.

3. DATABASE TAGGING

3.1. Introduction

In this section, we are about to discuss data subjective tagging. As we apply supervised learning, the information provided to the classifier has to be tagged.

We are about to suggest a scale and build a semi-automatic graphic user interface (GUI) to make the tagging work easier. Once we have the extracted features along with the corresponding label, we can apply classification techniques to define the amount of eye closure.

3.2. Scale Suggestion

As mentioned above, we want to estimate the eye closure amount by means of classification, so a discrete scale should be chosen in order to tag the information.

In order to produce qualitative and useful data tagging, the following considerations must be taken into account:

- A sufficiently **spacious** scale is required to achieve a clear separation between consecutive levels. So that, subjective unambiguous labeling could be obtained.
- A sufficiently **dense** scale is required to achieve a good estimation of eye dynamics. So that, a reliable estimation of eye closure amount could be obtained.

In order to balance the considerations presented, the input data will be taken into account alongside the biological data:

- Videos input is at an average 25fps (i.e frames every 40ms)
- Blinking duration is 200-400ms, every 2-10s [16]

In conclusion, we expect each blink to be represented by 5-10 frames in a periodic range of 50-250 frames.

The optimal scale will represent all the input possible levels. According to the described above, we chose a 5-level scale. The chosen scale represents the **opening level** of the eye and is normalized in the range between full closing and full opening.

Full closing - **(0.0, 0.25, 0.50, 0.75, 1.0)** - Full opening.

An illustration of the tagging labels is given in Table 3-1.

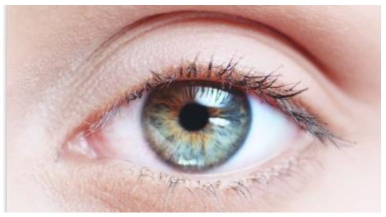
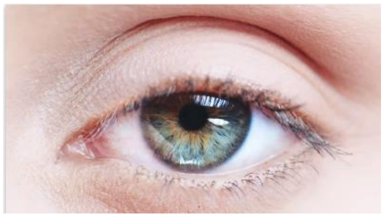
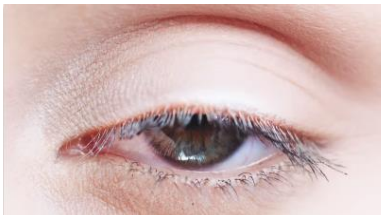
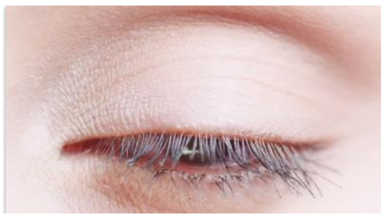
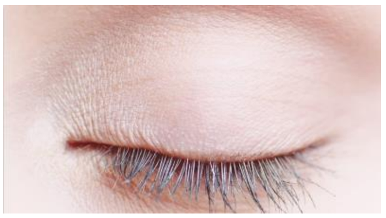
Example	Tagging Label
	**1.0 (100%)** Full opening
	**0.75 (75%)** Between full opening and half opening
	**0.50 (50%)** Half opening
	**0.25 (25%)** Between half opening and full closing
	**0.0 (0%)** Full closing

Table 3-1: Tagging labels illustration for all possible labels

One should note that:

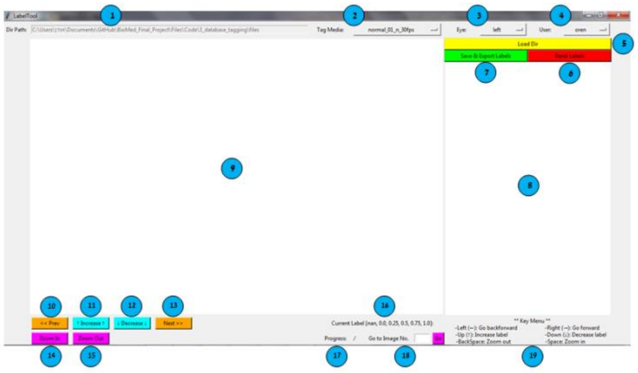
- For videos at a higher frame rate, each blink will be represented by more frames and we can choose a denser scale.
- In this project, we developed a software infrastructure which allows us to modify the scale as needed.

3.3. Tagging Data

To make the tagging process efficient and convenient, we built a semi-automatic graphic user interface (GUI). In this section, we will discuss the GUI structure and the tagging process using it.

3.3.1. GUI structure

The following figure shows the GUI divided into indexed regions.



LabelTool

Dir Path: C:\Users\[illegible]\Documents\GitHub\BioMed_Final_Project\Files\Code\3_database_tagging\files

Tag Media: normal_01_n_30fps

Eye: left

User: oren

Current Label [nan, 0.0, 0.25, 0.5, 0.75, 1.0]:

Progress: /

Go to Image No.

** Key Menu **
-Left (←): Go backforward
-Up (↑): Increase label
-BackSpace: Zoom out
-Right (→): Go forward
-Down (↓): Decrease label
-Space: Zoom in

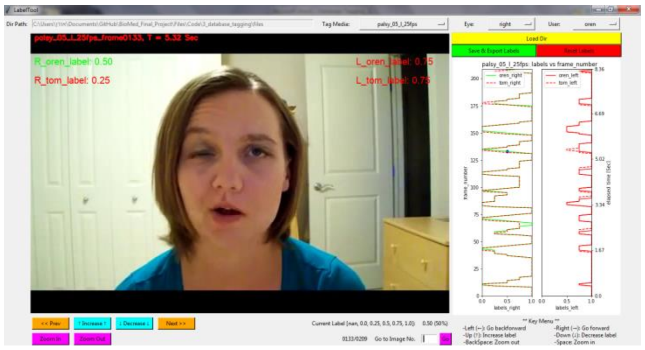
Figure 3-1: The tagging GUI, divided into indexed regions

We explain the GUI structure using the indexed regions:

1. **Dir Path:** entire directory path for tagging database, automatically set considering project folder hierarchy.
2. **Tag Media:** a list containing videos available for tagging.
3. **Eye:** a list containing eyes available for tagging (left, right).
4. **User:** a list containing the users available for tagging.
5. **Load Dir:** a button used for loading frames for tag, loading is according to the selected settings (dir, media, eye, user).
6. **Reset Labels:** a button used for initialize all labels values for the current settings. (= NaN, means - unlabeled frame).
7. **Save & Export Labels:** a button used for export 2 output files (for all users and for both eyes):
   a. An Excel file containing the labeled frames.
   b. A graph of labels vs. frame number and elapsed time.
8. **Graph Canvas:** displays a graph of labels vs. frame number and elapsed time (for all users and for both eyes). The graph is updated during the tagging process. Current plot (suitable for selected eye and user) is marked green while all others are marked red. Current frame is marked on the current plot using a blue dot.
9. **Frame Canvas:** displays the current frame for tag. Textual captions appear on the frame to indicate to labels (for all users and for both eyes). Current caption (suitable for selected eye and user) is marked green while all others are marked red.
10. **<< Prev:** a button used for moving back forward to previous frame (Key = left arrow ⟵).
11. **↑ Increase ↑:** a button used for increase tag label according to available levels. (Key = up arrow ↑).
12. **↓ Decrease ↓:** a button used for decrease tag label according to available levels. (Key = down arrow ↓).
13. **<< Next:** a button used for moving forward to next frame (Key = right arrow ⟶).
14. **Zoom In:** a button used for zoom in (Key = Space).
15. **Zoom Out:** a button used for zoom out (Key = Back Space).
16. **Current Label:** display the available and the current label.
17. **Progress:** display the current and the total frame number.
18. **Go to Image:** Allows moving to selected frame.
19. **Key Menu:** display all keys shortcut (for buttons 10. – 15.)

3.3.2. Tagging Process

The following figure shows the illustrates the tagging process.



LabelTool

Dir Path: C:\Users\[illegible]\Documents\GitHub\BioMed_Final_Project\Files\Code\3_database_tagging\files

Tag Media: palsy_05_l_25fps

Eye: right

User: oren

palsy_05_l_25fps_frame0133, T = 5.32 Sec

R_oren_label: 0.50

R_tom_label: 0.25

L_oren_label: 0.75

L_tom_label: 0.75

Current Label [nan, 0.0, 0.25, 0.5, 0.75, 1.0]: 0.50 (50%)

0133/0209 Go to Image No.

** Key Menu **
-Left (←): Go backforward
-Up (↑): Increase label
-BackSpace: Zoom out
-Right (→): Go forward
-Down (↓): Decrease label
-Space: Zoom in

Figure 3-2: Tagging process using the tagging GUI

The tagging process consists of the following steps:

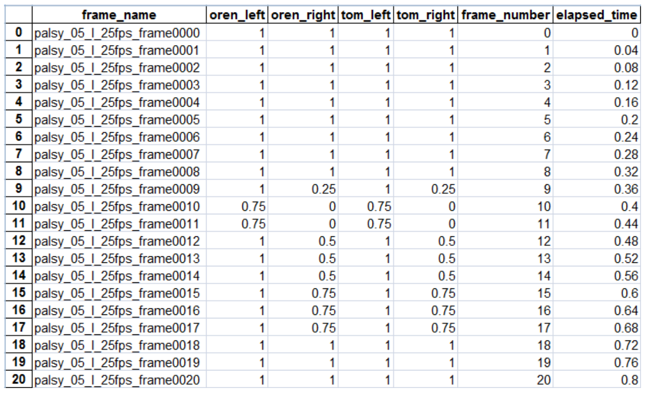
a. **Set tagging settings:** use the drop down lists to select: Dir Path (is set automatically), Tag Media, Eye, User.

b. **Load Dir:** click on Load Dir button, frames for tag will be loaded. If there are existing labels they will also be loaded.

c. **Tag frames:** tag the frames using the relevant buttons (Prev, Increase, Decrease, Next), and the aid buttons (Reset Labels, Zoom In, Zoom Out, Go to Image). The canvases (Graph Canvas, Frame Canvas) help monitoring the entire process.

d. **Save & Export labels:** user Save & Export button to create output files

   a. An Excel file containing the labeled frames.
   b. A graph of labels vs. frame number and elapsed time.



	frame_name	oren_left	oren_right	tom_left	tom_right	frame_number	elapsed_time
0	palsy_05_l_25fps_frame0000	1	1	1	1	0	0
1	palsy_05_l_25fps_frame0001	1	1	1	1	1	0.04
2	palsy_05_l_25fps_frame0002	1	1	1	1	2	0.08
3	palsy_05_l_25fps_frame0003	1	1	1	1	3	0.12
4	palsy_05_l_25fps_frame0004	1	1	1	1	4	0.16
5	palsy_05_l_25fps_frame0005	1	1	1	1	5	0.2
6	palsy_05_l_25fps_frame0006	1	1	1	1	6	0.24
7	palsy_05_l_25fps_frame0007	1	1	1	1	7	0.28
8	palsy_05_l_25fps_frame0008	1	1	1	1	8	0.32
9	palsy_05_l_25fps_frame0009	1	0.25	1	0.25	9	0.36
10	palsy_05_l_25fps_frame0010	0.75	0	0.75	0	10	0.4
11	palsy_05_l_25fps_frame0011	0.75	0	0.75	0	11	0.44
12	palsy_05_l_25fps_frame0012	1	0.5	1	0.5	12	0.48
13	palsy_05_l_25fps_frame0013	1	0.5	1	0.5	13	0.52
14	palsy_05_l_25fps_frame0014	1	0.5	1	0.5	14	0.56
15	palsy_05_l_25fps_frame0015	1	0.75	1	0.75	15	0.6
16	palsy_05_l_25fps_frame0016	1	0.75	1	0.75	16	0.64
17	palsy_05_l_25fps_frame0017	1	0.75	1	0.75	17	0.68
18	palsy_05_l_25fps_frame0018	1	1	1	1	18	0.72
19	palsy_05_l_25fps_frame0019	1	1	1	1	19	0.76
20	palsy_05_l_25fps_frame0020	1	1	1	1	20	0.8

Figure 3-3: Example of an Excel file containing the labeled frames, for all users and for both eyes

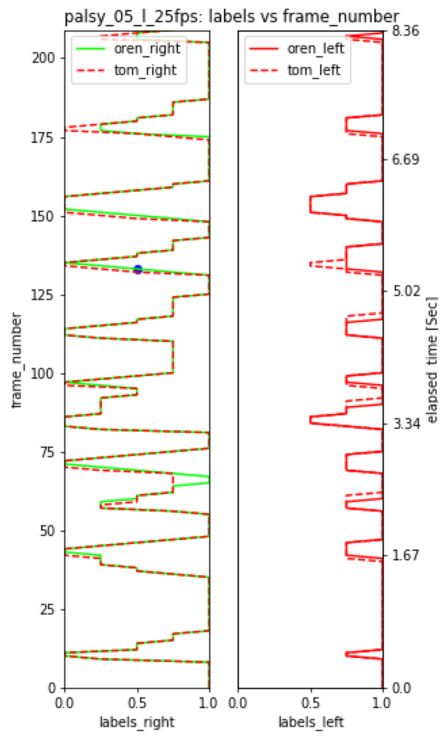


Figure 3-4: Example of tagging result, a graph of labels vs. frame number and elapsed time, for all users and for both eyes

3.3.3. Several Notes and Emphases

- Each eye is labeled individually (independent tagging).
- Value for unlabeled frame = NaN.
- As part of the efficiency, the tagging is done relatively between successive frames. That is, if any frame is not labeled (= NaN), in the tagging process it will get an initial value equal to the previous frame.
- Tagging can be performed by more than one user, the final score will be the selected weighting of the scores.

3.4. Summary

We have demonstrated a way of data tagging in a convenient and efficient way. The process is managed by a dedicated GUI implemented in Python. Now, when we have the extracted features along with the corresponding label, we can apply classification techniques to define the amount of eye closure.

4. TRAINING CLASSIFIERS

4.1. Introduction

In Section 2 , we dealt with the extraction of features which are highly correlative with a human eye's amount of closure. In Section 3, we built a semi-automatic labelling graphic user interface, allowing us to subjectively tag consecutive frames easily and efficiently.

During Section 4, we gather all the data collected from the previous sections, and use Machine Learning methods to train classifiers which will, hopefully, manage to classify well new data into 5 distinct levels of closure.
Prior to choosing a classifier, we need some data to train on. Therefore, we start with an establishment of a single database using the results of Sections 2 & 3.
As we reach such database, we will offer some state-of-the-art classifiers and evaluate their performance on a new data in comparison to the ground truth (i.e. labels).

4.2. Database Generation

4.2.1. Video Selection

The database includes an overall number of 38 videos (.mp4 files), each of 8-10 seconds, describing 436 distinct blinks, with frame rates within the range of 25 to 30 frames-per-second. The people filmed include 13 men and 14 women. In case of patients diagnosed with Bell's palsy, the videos that were selected for the database consist of videos from YouTube. In case of "healthy" patients, videos were taken using the 300-VW dataset [17], a dataset containing a collection of 300 short videos showing mainly faces (collected from YouTube; around 1 minute long)
As can be observed in Figure 2-2, one should pay attention to the Bell's palsy affect the shape of both patient eyes, causing the 'paralyzed' one to look wider and the 'healthy' one to look thinner than the normal case. Therefore, the database also includes videos from healthy (16) and 'paralyzed' (22) patients. Among the latter group, 12 videos depict a paralysis of the right side, while 10 videos depict a paralysis of the left side. These videos were chosen to cover various stages of Bell's Palsy – from complete paralysis to almost complete recovery.
The videos are annotated as follows:

normal/palsy_xx_y_zzfps

Where xx is the video number, y is the 'paralyzed' eye ('l' for left, 'r' for right or 'n' for none) and zz is the number of frames per second. Figure 4-1 shows selected frames from different videos in the database.

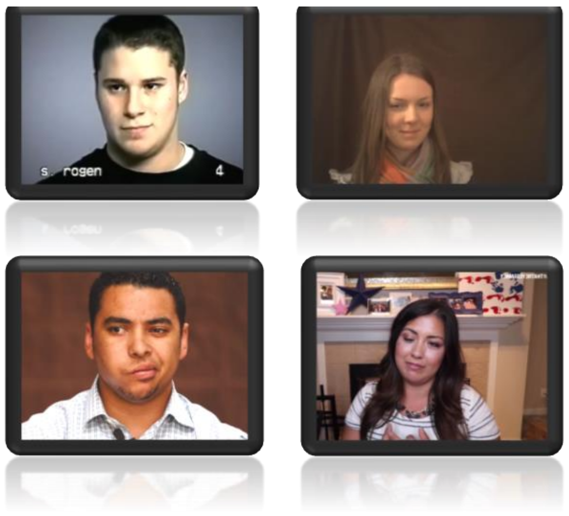


Figure 4-1: Videos integrated in the database. The upper pictures are taken from 300-VW dataset ('healthy' patients), whereas the lower pictures were taken from YouTube ('palsy' patients).

4.2.2. Initial Construction

The selected videos were used for feature extraction and subjective labeling. Their corresponding frames were added to the database only if they had non-null extracted features <u>and</u> non-null labels. We ended up with 9,591 frames and a total number of 19,182 rows (one row for each eye). Figure 4-2 shows graphs of extracted features and manually-tagged labels as a function of the elapsed time, using a video from the database (palsy_04_l_25fps.mp4). We can observe a high correlation th labels and features. This approves our choice of features to serve as descriptors for classification.

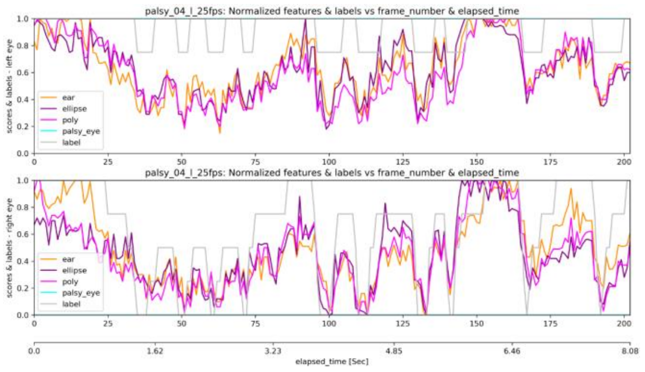


Figure 4-2: Extracted features and labels vs frame number and time, using palsy_04_l_25fps. Note the high correlation between the measures. This match is worse for the 'paralyzed' eye, rather than the 'healthy' one. This can be solved using Machine Learning tools.

Naïve Bayes classifier was trained on the data (using cross-validation) and got an average surprising score of 72%! Using K-Nearest Neighbor (k=10) improves it even more, reaching an average score of above 77%.

One might think that these results are quite good and can have minor adjustments in order to reach an optimal classifier. However, as seen in Figure 4-3, the situation is much complicated – as the labels corresponding to a full eye opening constitute a significant part (>75%!) of the labels. This is because in most of the time the patient tends to keep his eye open and isn't blinking at all. As a result, any suggested classifier would be highly biased to a full eye opening. Even a classifier that will classify all the data to a full eye opening will be right most of the times! An example of such situation is shown in Figure 4-4.

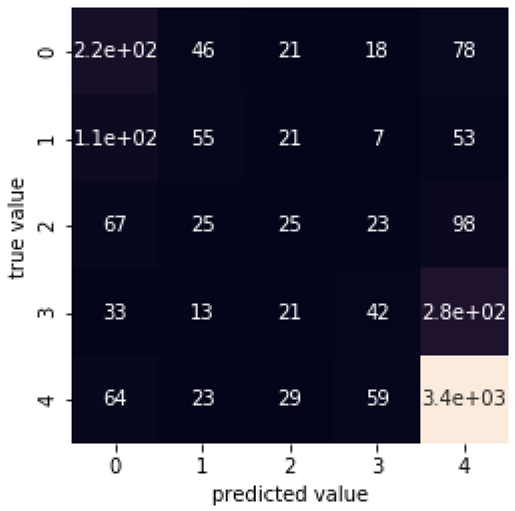


Figure 4-3: A confusion matrix of a K-Nearest Neighbor classifier (k=10), with accuracy score of 0.772. The accuracy score was calculated using a stratified cross-validation, setting 20% of the samples (4,796) as test set. The confusion matrix shows true value labelling versus their predicted values. As can be easily seen, the classifier does well with the label of full opening, but fails with the other labels, as the label of full opening is much dominant (>75%)

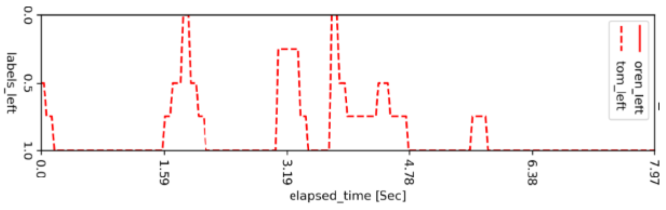


Figure 4-4: An example of subjecting tagging, using palsy_01_r_30fps.mp4. Although the patient tends to blink several times during the video, he keeps his eye open for long periods.

To solve this, we need to decrease the amount of 'widely-opened' samples dramatically. One way of doing this is extracting only the regions which have 'abnormal' activity, i.e. blinks or partial closure.

4.2.3. Blink Extraction

We continue the establishment of our final database by reducing the amount of 'widely-opened' labels. To do this, we need a clear definition of a blink. First, only the labels that relate to a full or partial closure are taken into account.

These samples are then subdivided into different 'blinks', based on their relative location in the database. The blink number is then added as a new column to database – and from now on, we treat the data as a collection of different *blinks*, rather than just different video files. This observation might be important for the advanced stages of this section. An example of the process described above is shown in Figure 4-5.

Label	.75	.25	.0	.5	.5	.0	.5	.75	.25	.0
#Row	19	20	21	22	67	68	69	111	112	113
#Blink	1	1	1	1	2	2	2	3	3	3

Figure 4-5: Samples, based on their row index in the database, are subdivided into bundles of blinks. Each group of adjacent indices now represent a single 'blink'.

Now that we added blink information, we need to have some 'widely-opened' labels re-added to the database. We chose to add a number of samples that is equal to the longest *sequence* of the same label in the current blink. The samples are added equally before the start and after the end of the blink. For example, if the longest sequence is 4, 2 samples are taken into account before and after the samples of the blink.

However, as we reduce the amount of 'widely-opened' labels, we didn't solve the opposite case, i.e., when the patient shuts his eyes for a long period of time. Such situation is demonstrated in Figure 4-6. To prevent a situation where the 'widely-closed' label is highly dominant, we chose to *limit* the longest sequence to a number given as a parameter (4 by default). This technique is explained in Figure 4-7.

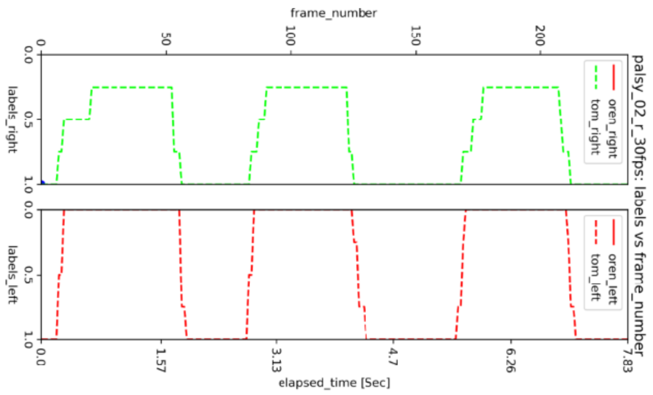


Figure 4-6: Example taken from subjective labelling of palsy_02_r_30fps. Note that the patient keeps his eyes shut for long periods of time, creating a lot of 0-labels in the case of the 'healthy' eye and 0.25-labels in the case of the 'paralyzed' eye.

			.75	.75	.5	.5	.5	.5	.5	.0	.5	.75
1	1	.75	.75	.5	.5	.5	.5	.0	.5	.75	1	1

Figure 4-7: An example of the processing done for each blink. The first and second rows relate to the blink before and after the processing, respectively. Note that the longest sequence in the first row is 5 repetitions of 0.5-label, which is longer than the maximum allowable sequence (4). Therefore, the sample with the fifth 0.5-label (marked in gray) is omitted from the database. In addition, four 1-label samples are added, 2 for each edge.

The final database properties are given in Table 4-1, Table 4-2 and Table 4-3. As can be seen, the data is well distributed for 'healthy' and 'palsy' patients, as well as left and right eyes.

# Blinks	Men	Women
Healthy patients	107	107
Patients diagnosed with Bell's Palsy	111	111

Table 4-1: 'Blinks' extracted from the database's input files, with respect to health status and gender

Label	0	0.25	0.5	0.75	1
Healthy patients	464	388	419	570	610
Patients diagnosed with Bell's Palsy	354	351	461	730	768
Total	818	739	880	1300	1378

Table 4-2: Number of database's samples with respect to health status and label

Label	0	0.25	0.5	0.75	1
Left Eye	408	343	431	664	718
Right Eye	410	396	449	636	660
Total	818	739	880	1300	1378

Table 4-3: Number of database's samples with respect to eye location and label

4.3. The Feature Vectors

Prior to classification, we chose to use two different kinds of feature vectors. One feature vector is based on the features extracted directly from the landmarks – EAR and Shoelace formula.

One should notice that neighboring frames have a lot in common, especially when the eye is opening or shutting, as they all describe well the eye's dynamics. This information might be helpful in classification.

There are tradeoffs for choosing the right number of neighboring frames to account for learning. Too few frames fail to describe the dynamics of an entire blink; however, choosing too many frames make the prediction more sensitive to feature incoherence and might end up in an over-fitting of the training data. Before choosing the right number of neighboring frames, we need to understand how many frames take part in each blink. Figure 4-8 shows a histogram of blink durations (in frames). It can be inferred that most blinks are around 5 to 15 frames in length.

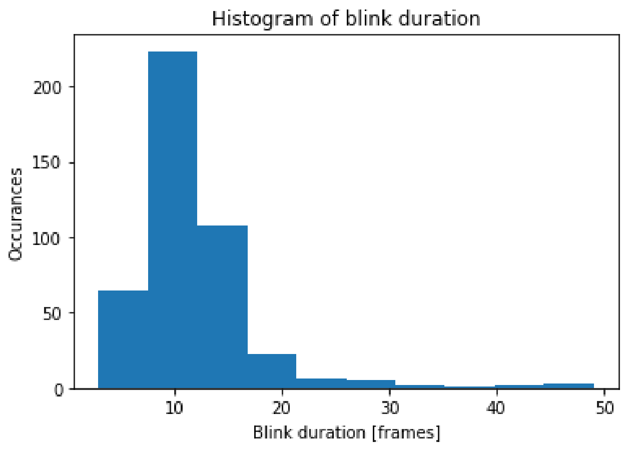


Figure 4-8: A histogram of blink duration for each blink in the database. Most blinks have lengths of 5 to 15 frames.

For searching the best number of frames, we use SVM with RBF kernel (which will be proven later as the best classification method). We evaluate the results using 'blink-wise' classification, as will be explained later. We start by using only previous frames (Figure 4-9) and then by using both past and future frames (Figure 4-10). It can be clearly seen that the best results (~75%) are obtained using features from 6 frames before and after the current frame. Using only past frames yields worse results (<60%).

However, using future frames make the estimation non-suitable for real-time applications. This is one reason for applying 'blink-wise' classification – i.e., classify all the frames in the current blink and treat the minimum predicted label as the next blink 'score', assuming that adjacent blinks have the same closure amounts. This will be explained in detail later.

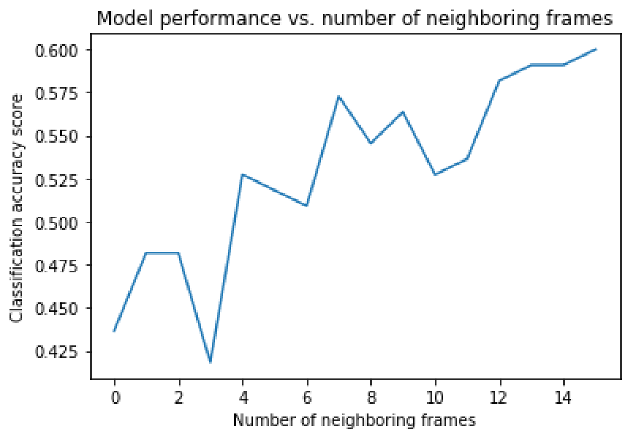


Figure 4-9: Accuracy scores for 'blink-wise' classification with respect to the number of past frames taken into account.

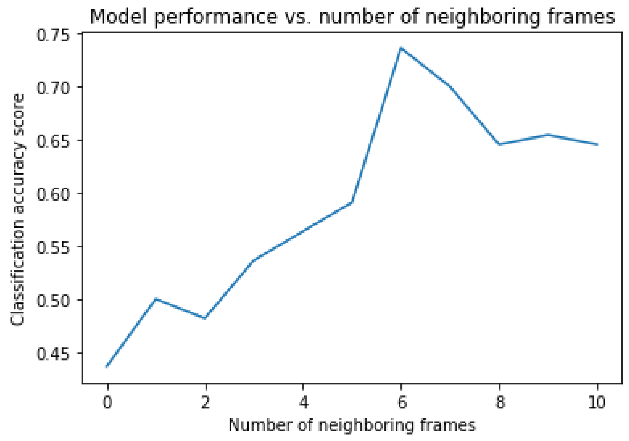


Figure 4-10: Accuracy scores for 'blink-wise' classification with respect to the number of past and future frames taken into account.

In addition, we chose to use the indication for eye status (healthy or paralyzed), as we saw different blink patterns between them. For example, a blink of a healthy eye tends to be longer (in terms of number of frames) than a blink of a paralyzed eye. This difference is demonstrated in Table 4-4.

To sum up, the first feature vector is made of **27** features – landmark features of 13 frames (the current frame and six frames before and after it), along with the eye status indication.

tag_left	tag_right	frame_number	elapsed_time
1	1	32	1.28
1	1	33	1.32
1	1	34	1.36
1	1	35	1.4
1	0.75	36	1.44
1	0.5	37	1.48
1	0.5	38	1.52
1	0.5	39	1.56
1	0.25	40	1.6
0.75	0.25	41	1.64
0.75	0	42	1.68
0.75	0	43	1.72
0.75	0	44	1.76
0.75	0.25	45	1.8
0.75	0.5	46	1.84
1	0.75	47	1.88
1	1	48	1.92
1	1	49	1.96
1	1	50	2
1	1	51	2.04

Table 4-4: Manual tagging results of palsy_05_l_25fps.mp4. The healthy eye (right) tends to blink 'slower' than the paralyzed one (left). The left eye starts blinking after the right eye and end blinking before the right eye.

Another feature vector is based on graphical features, extracted using 'cropped' images containing only one eye, which we generate through the bounding boxes determined by the facial landmarks. As discussed in Section 2, we use Histograms of Oriented Gradients (HOG) as the graphical features. For each frame, two images for both eyes were cropped and then normalized to a constant size of 26 x 34 pixels. These images were further cropped into sizes of 24 x 33 pixels. Using 3x3 cells and 6x6 blocks, each cell containing a 9-bin histogram, we ended up in 36 features for each block, 10 blocks in the horizontal axis and 7 blocks in the vertical axis, giving a total vector length of 36 x 10 x 7 = **2,520** features.

4.4. Scoring Methods

In order to evaluate the performance of the various classifiers on our generated database, we are using two different approaches:

4.4.1. 'Frame-Wise' Scoring

Using the 'frame-wise' approach, we evaluate the score for <u>each frame</u> in the dataset. The evaluation is done by using a variant of a 5-fold cross-validation process, with the following procedure:

1. Run for 5 iterations:
    - **1.1.** Randomly select 20% of the blinks from healthy patients and 20% of the blinks from paralyzed patients
    - **1.2.** Remove the selected blinks from the available blinks to be randomized in the next iteration
    - **1.3.** Use all the frames that belong to the selected blinks as the test set, and consider the rest as train set
    - **1.4.** Train the given classifier on the train set
    - **1.5.** Evaluate the results using the test set

The selection of whole blinks is done for analysis convenience. The separation between healthy and 'palsy' blinks is done to avoid biasing to any health status.
As we are about to witness, the evaluation using 'frame-wise' approach returns a maximum accuracy score of ~68%, which can be improved by applying the 'blink-wise' approach.

4.4.2. 'Blink-Wise' Scoring

By applying the 'blink-wise' approach, we evaluate the score for <u>each blink</u> in the dataset, i.e., considering the data as a collection of blinks rather than individual frames. The motivation for acting this way is our final target – trying to assess the <u>minimal</u> amount of closure of the paralyzed eye with respect to the healthy eye, <u>for each blink</u>, and use it for proper electrical stimulation of the paralyzed eye. By looking at Table 4-5, comparing the system's predictions versus the ground truth reveals that the estimation tends to be inaccurate by means of 'frame-wise' scoring, but the minimum opening rate is correct (50%).

#Frame	Time [s]	EAR	Poly	Label	Prediction
81	3.24	0.61	0.6	1	1
82	3.28	0.33	0.37	1	1
83	3.32	0.06	0.22	0.75	0.75
84	3.36	0.08	0.11	0.5	0.75
85	3.4	0.08	0.13	0.5	0.75
86	3.44	0.18	0.23	0.5	0.5
87	3.48	0.26	0.34	0.75	0.5
88	3.52	0.31	0.38	0.75	0.75
89	3.56	0.26	0.34	0.75	1
90	3.6	0.23	0.35	1	1
91	3.64	0.26	0.34	1	1

Table 4-5: Predictions versus ground-truth labels for a specific blink. The evaluated video is palsy_05_l_25fps.mp4. The results are shown for the paralyzed eye, i.e., the left eye. The accuracy score using the 'frame-wise' approach is 63.6%. However, by using the 'blink-wise' approach, i.e., comparing the minimal prediction with the minimal ground-truth label, we get 100% accuracy for this blink.

The process for 'blink-wise' scoring is done as follows:

1. Apply 'frame-wise' scoring for each frame in the dataset

2. For each blink from a 'paralyzed' eye:

- **2.1.** Find the minimum ground-truth label
- **2.2.** Find the minimum prediction, not including the first frame and the last frame
- **2.3.** Find all the frame indices whose predictions equal to the minimum value
- **2.4.** Sort the indices and get the index in the middle
- **2.5.** Look at the prediction of the frame with the middle index, along with the predictions of one frame before and one frame after it
- **2.6.** Calculate the median of these 3 scores and consider it as the blink score

Note that we are calculating the accuracy score for 'paralyzed' eyes only, and not for all the blinks in the dataset, because the final application only estimates the amount of closure for this particular eye.

Stage 2.2 helps us filtering the outliers, as we assume that every blink starts and ends with a 100% eye opening. Stage 2.4 is used for increasing the model's precision, assuming that each blink is almost symmetrical so that the minimum value should be found somewhere in the middle of the blink frames. Stage 2.6 adds an extra layer of noise immunity by filtering rapid changes that might stem in the incoherence of the extracted features.

4.5. Classification Methods

In this work, we compare classification results using the state-of-the-art classifiers, which are highly suitable for multi-class classification and yet have a satisfying computational complexity.

4.5.1. Naïve Bayes

Naïve Bayes classifiers are a family of simple "probabilistic classifiers" based on applying Bayes' theorem with strong (naive) independence assumptions between the features.
Bayes' theorem simply says that:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Using Bayes theorem, we can find the probability of A happening, given that B has occurred. Here, B is the evidence and A is the hypothesis. The assumption made here is that the predictors/features are independent. That is presence of one particular feature does not affect the other. Hence it is called naive.
Recall A as the label y and B as the feature vector **X**, Bayes' theorem can be rewritten as:

$$P(y|x_1, x_2, \dots, x_n) = \frac{\prod_{i=1}^{n} P(x_i|y)}{\prod_{i=1}^{n} P(x_i)}$$

Where:

$$\boldsymbol{X} = (x_1, x_2, \dots, x_n)$$

As the denominator does not change for all entries in the dataset, it can be removed, and a proportionality can be introduced:

$$P(y|x_1, x_2, \dots, x_n) \propto p(y) \prod_{i=1}^{n} P(x_i|y)$$

In case of multi-class classification, we want to find the class y with the maximum probability:

$$y = argmax_y\left(p(y)\prod_{i=1}^{n} P(x_i|y)\right)$$

In this project, we examine the use of Gaussian Naïve Bayes. We assume that the features are sampled from a Gaussian distribution. Hence, the formula for conditional probability changes to:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y{}^2}} exp\left(-\frac{\left(x_i - \mu_y\right)^2}{2\sigma_y{}^2}\right)$$

Naïve Bayes classifiers are fast and easy to implement but their biggest disadvantage is that the requirement of predictors to be independent. In most of the real-life cases, the predictors are dependent, this hinders the performance of the classifier.

As for our data, we get poor result using the Gaussian Naïve Bayes classifier. This means that probably our features are not statistically independent, or their probability distribution cannot be assumed as Gaussian. Figure 4-11 and Figure 4-12 shows confusion matrices for Gaussian Naïve Bayes classification using the first and second feature vector, respectively.

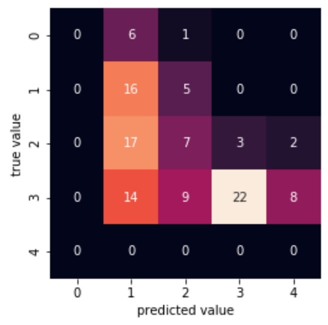


Figure 4-11: 'Blink-wise' confusion matrix using Gaussian Naïve Bayes with the first feature vector. The accuracy score is about 40.9%

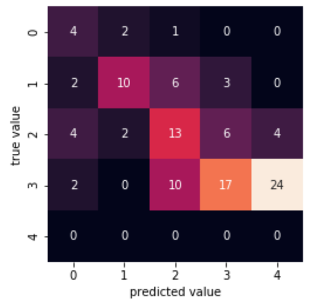


Figure 4-12: 'Blink-wise' confusion matrix using Gaussian Naïve Bayes with the second feature vector. The accuracy score is about 40%

4.5.2. K-Nearest Neighbors (K-NN)

The KNN algorithm assumes that similar things exist in close proximity. In other words, similar things are near to each other. KNN captures the idea of similarity (sometimes called distance, proximity, or closeness) with some basic mathematics – calculating the distance between points on a graph. There are other ways of calculating distance, and one way might be preferable depending on the problem we are solving. However, the straight-line distance (also called the Euclidean distance) is a popular and familiar choice.

The stages of the KNN algorithm for classification can be simplified using the following pseudo-code:

```
1. Load the data
2. Initialize K to your chosen number of neighbors
3. For each example in the data
        3.1. Calculate the distance between the query example
             and the current example from the data.
        3.2. Add the distance and the index of the example to
             an ordered collection
4. Sort the ordered collection of distances and indices from
   smallest to largest (in ascending order) by the distances
5. Pick the first K entries from the sorted collection
6. Get the labels of the selected K entries
7. Return the mode of the K labels
```

To select the K that's right for our data, we run the KNN algorithm several times with different values of K and choose the K that reduces the number of errors we encounter while maintaining the algorithm's ability to accurately make predictions when it's given data it hasn't seen before.
As we saw before, there are tradeoffs in choosing the appropriate value of K. Smaller values would make our predictions become less stable. Inversely, larger values would apply extra averaging on the data, until we begin to witness an increasing number of errors. It is at this point we know we have pushed the value of K too far. Moreover, in cases where we are taking a majority vote (e.g. picking the mode in a classification problem) among labels, we usually make K an odd number to have a tiebreaker.

To evaluate the right value of K, we run for different values of K (odd numbers only). We chose the range between 1 to 41 neighbors. Figure 4-13 and Figure 4-15 shows accuracy scores for different K's while using the first and the second feature vectors, respectively. Figure 4-14 and Figure 4-16 shows confusion matrices for the best K value. The scores were evaluated using the 'blink-wise' approach. We can see that using K-NN brings us the results with the highest accuracy scores for the second feature vector.

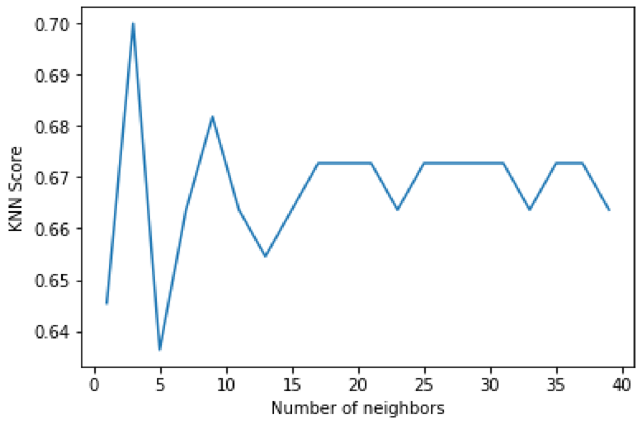
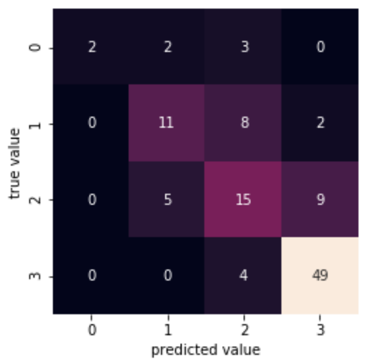


Figure 4-13: 'Blink-wise' accuracy score using K-Nearest neighbors with the first feature vector, for different K values. The best score is given for K=3, reaching 70% accuracy



	0	1	2	3
0	2	2	3	0
1	0	11	8	2
2	0	5	15	9
3	0	0	4	49

Figure 4-14: 'Blink-wise' confusion matrix using K-Nearest neighbors (K=3) with the first feature vector. The accuracy score is about 70%

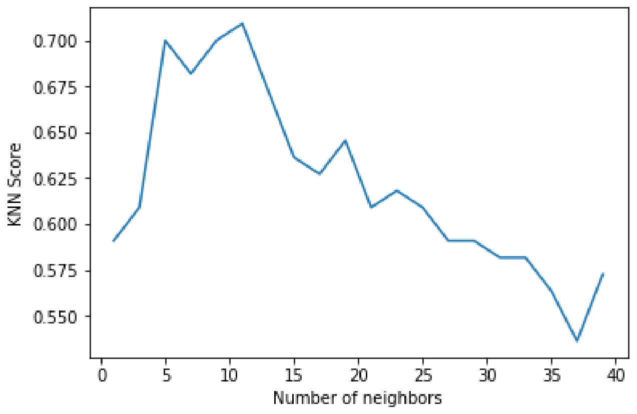
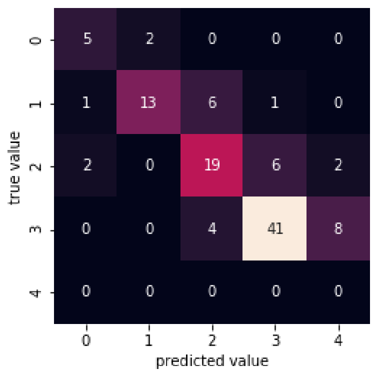


Figure 4-15: 'Blink-wise' accuracy score using K-Nearest neighbors with the second feature vector, for different K values. The best score is given for K=11, reaching 70.9% accuracy



	0	1	2	3	4
0	5	2	0	0	0
1	1	13	6	1	0
2	2	0	19	6	2
3	0	0	4	41	8
4	0	0	0	0	0

Figure 4-16: 'Blink-wise' confusion matrix using K-Nearest neighbors (K=11) with the second feature vector. The accuracy score is about 70.9%

4.5.3. Support Vector Machine (SVM)

We are using the Support Vector Machine algorithm, which is described in detail in section 1.2.2. We assess the model performance using two different kernels: the well-known linear kernel, and the Radial Basis Function (RBF) kernel.

The RBF kernel on two samples, x and x', represented as feature vectors in some input space, is defined as:

$$K(\boldsymbol{x},\boldsymbol{x}') = exp\left(-\frac{||\boldsymbol{x}-\boldsymbol{x}'||^2}{2\sigma^2}\right)$$

Where $||\boldsymbol{x}-\boldsymbol{x}'||^2$ is the squared Euclidean distance between the two feature vectors and σ is a free parameter. An equivalent definition involves a parameter $\gamma = \frac{1}{2\sigma^2}$, so that the kernel function becomes:

$$K(\boldsymbol{x},\boldsymbol{x}') = exp(-\gamma||\boldsymbol{x}-\boldsymbol{x}'||^2)$$

The γ parameter controls the amount of data fitting. Higher the value of γ, will try to exact fit the as per training data set i.e. generalization error and cause over-fitting problem. Figure 4-17 illustrates the decision boundaries for 3-class classification problem, using different values of γ.

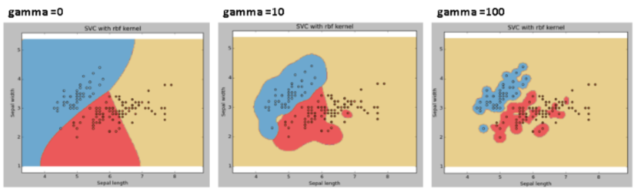


Figure 4-17: An illustration of decision boundaries for 3-class classification problem, using different values of γ. Larger values of γ might suffer from over-fitting.

Another parameter that can be tuned in SVM classification if the penalty parameter C, or the error term. It also controls the trade-off between smooth decision boundary and classifying the training points correctly. Figure 4-18 illustrates the decision boundaries for 3-class classification problem, using different values of C.

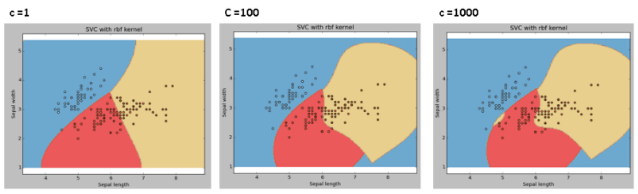


Figure 4-18: An illustration of decision boundaries for 3-class classification problem, using different values of C. Larger values of C might classify the training points correctly, but will have sharp and 'peaky' decision boundaries.

Using grid search, we tried to find the best combination of C and γ. We found out that the best results were while using C=1000 and γ equals to the normalized standard deviation of the data, i.e., $\gamma = \frac{\sigma_x}{N_{features}}$. This configuration of γ is possible by setting gamma='scale' while using scikit-learn's SVC() function. We start with a linear kernel. Figure 4-19 and Figure 4-20 shows confusion matrices of using linear SVM (C=1000) for the first and the second feature vectors, respectively. It can be inferred from the poor accuracy scores that the problem is not linearly separable.

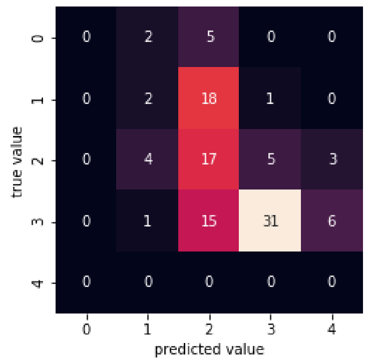


Figure 4-19: 'Blink-wise' confusion matrix using Linear SVM with the first feature vector. The accuracy score is about 45.4%

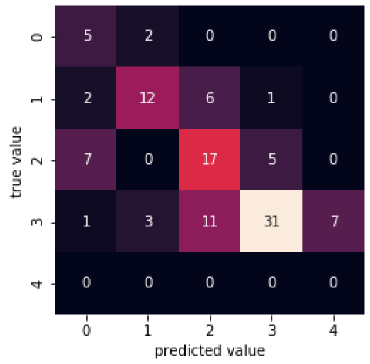


Figure 4-20: 'Blink-wise' confusion matrix using Linear SVM with the second feature vector. The accuracy score is about 59.1%

As for the RBF kernel, Figure 4-21 and Figure 4-22 shows confusion matrices of using RBF SVM (C=1000, γ='scale') for the first and the second feature vectors, respectively. We can see that this approach brings us the results with the highest accuracy scores for the first feature vector.

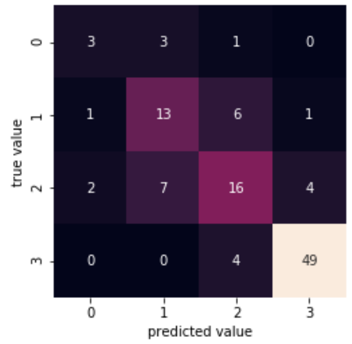


Figure 4-21: 'Blink-wise' confusion matrix using RBF SVM with the first feature vector. The accuracy score is about 73.6%

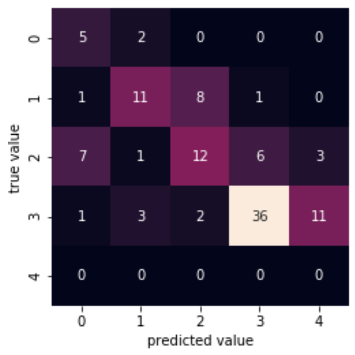


Figure 4-22: 'Blink-wise' confusion matrix using RBF SVM with the second feature vector. The accuracy score is about 58.2%

4.5.4. Random Forest

Random Forest is a flexible, easy to use machine learning algorithm that produces, even without hyper-parameter tuning, a great result most of the time.

Random Forest builds multiple decision trees and merges them together to get a more accurate and stable prediction. Random Forest adds additional randomness to the model, while growing the trees. Instead of searching for the most important feature while splitting a node, it searches for the best feature among a random subset of features. This results in a wide diversity that generally results in a better model.

The Hyperparameters in random forest are either used to increase the predictive power of the model or to make the model faster. The relevant hyperparameters in Python's sklearn Random Forest implementations can be divided into two categories:

A. Parameters that control the predictive power:
   - n_estimators: the number of trees the algorithm builds before taking the maximum voting or taking averages of predictions. In general, a higher number of trees increases the performance and makes the predictions more stable, but it also slows down the computation.
   - max_features: the maximum number of features Random Forest considers splitting a node.

B. Parameters that control the model speed:
   - n_jobs: tells the engine how many processors it is allowed to use. If it has a value of 1, it can only use one processor. A value of "-1" means that there is no limit.
   - max_depth: controls the maximum depth allowed for each decision tree in the forest, hence the computational complexity.

We used grid search for tuning the above parameters in order to find the best combination that will give the best accuracy score. The best parameters are given for n_estimators=500, max_features=log2, n_jobs=-1 and max_depth=7. These parameters were chosen as the minimum ones that will lead to significant results.

Figure 4-23 and Figure 4-24 show confusion matrices of using Random Forest with the mentioned parameters for the first and second feature vectors, respectively. As can be seen, we get identical results with respect to RBF SVM; however, the complexity of Random Forest is much higher than RBF SVM.

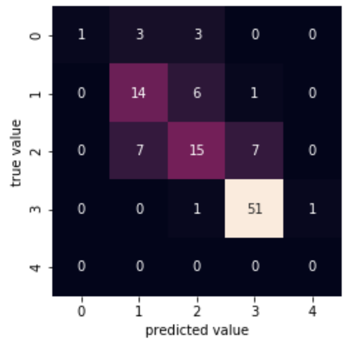


Figure 4-23: 'Blink-wise' confusion matrix using Random Forest with the first feature vector. The accuracy score is about 73.6%

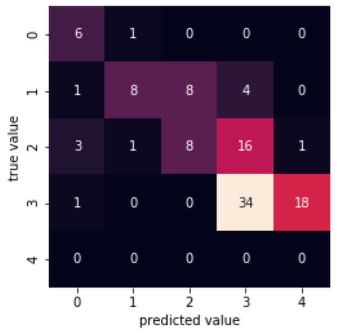


Figure 4-24: 'Blink-wise' confusion matrix using Random Forest with the second feature vector. The accuracy score is about 50.9%

4.6. Dimensionality Reduction

In order to handle "curse of dimensionality" and avoid issues like over-fitting in high dimensional space, methods like Principal Component analysis is used. This is more important while using the second feature vector because of its high dimensionality (>2500!) In addition, lowering the dimensionality also helps by reducing the learning rate, using only relevant data without redundancies.

PCA is a method used to reduce number of variables in your data by extracting important one from a large pool. It reduces the dimension of the data with the aim of retaining as much information as possible. In other words, this method combines highly correlated variables together to form a smaller number of an artificial set of variables which is called "principal components" that account for most variance in the data.

PCA starts by normalizing the predictors by subtracting the mean from each data point. It is important to normalize the predictor as original predictors can be on the different scale and can contribute significantly towards variance.
Next, we calculate the covariance matrix for the data which would measure how two predictors move together. For reference covariance formula is:

$$COV_{(x,y)} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{n-1}$$

Next, we calculate Eigen values and Eigen vector of the above matrix. This helps in finding underlying patterns in the data. Now we can perform reorientation - converting the data into new axes by multiplying original data with eigenvectors, which suggests the direction of new axes. We now can choose to leave out smaller eigen vector or use both and decide how many sets of features to keep based on which set accounts for 95% or more variance.

Figure 4-25 and Figure 4-26 shows graphs of data's explained variance for different numbers of principal components, using both feature vectors. As can be seen, there is a lot of redundancy in the dimensionality of the second feature vector.

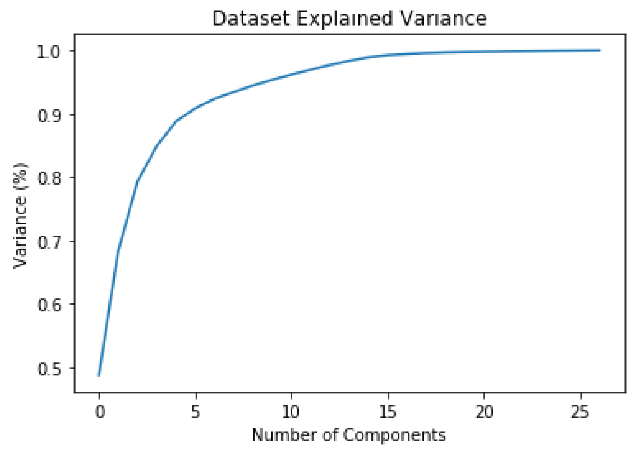


Figure 4-25: Dataset explained variance for the first feature vector. Almost 100% of the variance can be described using only 15 principal components

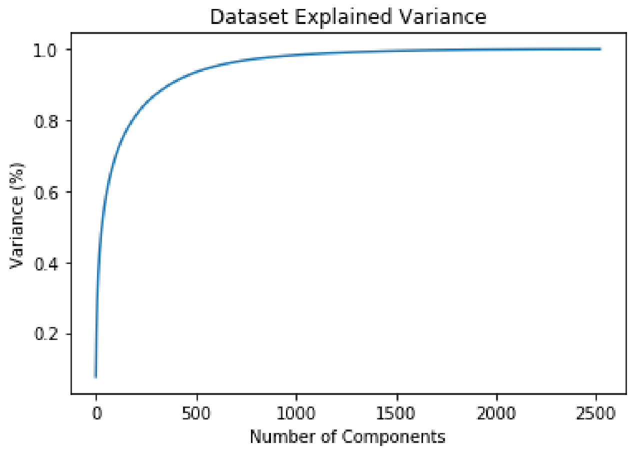


Figure 4-26: Dataset explained variance for the second feature vector. Almost 100% of the variance can be described using only 1500 principal components

We evaluated the performance of the PCA while using the classifiers that brings us the best accuracy scores. We used RBF SVM for the first feature vector and K-NN (K=11) for the second feature vector. Applying PCA on both feature vectors returns worse results for the first feature vector (66.1% frame-wise; 58.2% blink-wise), and the same results for the second feature vector (58.5% frame-wise; 70.9% blink-wise). Therefore, applying PCA is only relevant for the second feature vector. A detailed calculation reveals that using PCA improves processing times by almost 33%.

4.7. Performance Analysis

Using all the above information, Table 4-6 summaries the performance of each tested method, using both suggested feature vectors.

Feature vector	Method	Frame-wise accuracy	Blink-wise accuracy
1	Gaussian Naïve Bayes	49.9%	40.9%
	K-NN (K=3)	64.5%	70.0%
	Linear SVM	54.7%	45.4%
	RBF SVM	**66.7%**	**73.6%**
	Random Forest	68.9%	73.6%
2	Gaussian Naïve Bayes	45.1%	40.0%
	K-NN (K=11)	58.5%	70.9%
	Linear SVM	47.0%	59.1%
	RBF SVM	52.6%	58.1%
	Random Forest	54.0%	50.9%

Table 4-6: Accuracy scores for each tested method, using both suggested feature vectors. The selected method is shown in bold

According to this table, there are three methods that works best for our data:

(a) Feature vector 1 + RBF SVM
(b) Feature vector I + Random Forest
(c) Feature vector II + K-NN (K=11)

In comparison to (b), method (a) runs about 3 times faster while giving the same results using 'blink-wise' approach (~30ms vs ~100ms; values are calculated as the mean processing time for 100 independent runs, using the same testing equipment).

As being said in Section 2, the second feature vector is offered in case of the first feature vector, based directly on the six points given for each eye, fails to describe the eyes' boundaries. Only extracting of the features takes about 150ms before any machine learning tool is being applied on the data. Since our predictions are not getting better, the use of the second feature vector, for this constellation, is not necessary. It can be explained by using videos with poor resolution or slower frame rates. Moreover, the eyes' images extracted from the frames are assumed to be mean-shifted and normalized, which isn't the case in our data. Perhaps shifting and scaling the images prior to learning or using pyramidal (e.g. Gaussian) representation to acquire more features, would have maximized the accuracy.

As being said, the selected method lasts about 30 milliseconds on average. The code was tested on a laptop with the following specifications:

- Operating System: Windows 10 Home, 64-bit
- CPU: Intel 3rd Generation i7-7537U @ 2.5 GHz
- RAM: 8 GB

We conclude that the estimation of the amount of closure for a single blink lasts a period that resembles the gap between consecutive frames, using the highest frame rate from the dataset (30 fps). We believe that these results might get even better using embedded programming for the final application. In addition, as we will see in Section 5, the estimation isn't done after extracting features from each frame, but is performed on a complete blink, to be used as the trigger level for the next blink.

As for the maximum accuracy, we got almost 75% success rate using 5-level manual labelling. These are quite good results; however, they might be improved by changing one or more of the following:

- Using more accurate eye detectors: As we saw in Section 2 (especially Figure 2-19), the chosen solution for detecting the outlines of the eyes isn't perfect. It has higher sensitivity to the lower levels (i.e., 0% and 25%) than the higher ones (75% to 100%), as demonstrated in the above confusion matrices.
- Using high resolution videos: The entire database is made using YouTube videos, which aren't necessarily taken in high resolutions. We could choose different videos for healthy patients; however, we found a very limited number of videos showing patients dealing with Bell's palsy - most of them are of poor quality. Perhaps the accuracy of the eye detector using high-definition videos would be higher, improving the results.

- Using higher frame rates: Higher frame rates will allow us to get more samples for each blink, making the prediction less sensitive to feature noise and other artifacts.
- Increasing the database: We eventually reached a database containing 436 blinks, only half of them are taken from patients diagnosed with Bell's palsy. A larger number of recorded blinks may increase the reliability of the model and reduce over-fitting. We believe that recording more videos of patients will have a direct positive influence on the results.
- Applying much complexed and sophisticated classifiers: The use of advanced learning methods, i.e. deep learning, might reach excellent results. They definitely worth the try, but still have to be suitable for real-time applications.
- Using different number of levels for tagging: The choice of 5 levels for manual tagging was random, and maybe choosing a different number of levels would be more suitable for our dataset. Figure 4-27 and Figure 4-28 show confusion matrices of the first feature vector with RBF SVM, while using only 3 levels for tagging and classification – meaning that the eye can be classified into on of three modes: opened, closed or half closed. Figure 4-27 accounts for 'frame-wise' estimation while Figure 4-28 accounts for 'blink-wise' estimation. We clearly see that reducing the number of levels will lead to a great improvement – almost 92% success rate!
  Note that increasing the frame rate will allow us to increase to number of available levels for tagging because there are more samples within each blink.

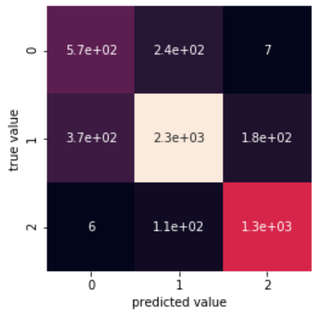


Figure 4-27: 'Frame-wise' confusion matrix using RBF SVM with the first feature vector & 3-level tagging. The accuracy score is about 81.9%

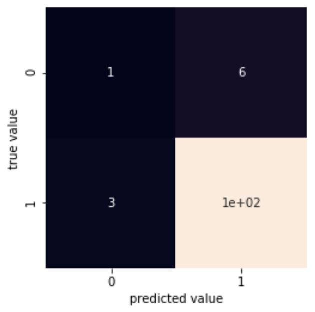


Figure 4-28: 'Blink-wise' confusion matrix using RBF SVM with the first feature vector & 3-level tagging. The accuracy score is about 91.8%

4.8. Summary

During this section, we combined the work done in the previous sections and established the final dataset, which we use to train and test various classifiers and compare their performance.
Our final database includes almost the same number of blinks for healthy and Bell's palsy patients, as well as men and women. In addition, we got rid of many samples which are not describing eye movements, preventing a clear bias to the higher levels.
We used two different sets of features, one is also based on the features of the surrounding frames. We then defined two evaluation methods, 'frame-wise' and 'blink-wise' and used them to compare the performance of the various classifiers. We reached a maximum accuracy of almost 75%, which is quite good but can be improved by increasing the database, using 'smarter' machine learning tools or use videos of improved quality and frame rate. We can also reduce the number of levels to 3, using success rates of almost 92%.

We are now facing the last obstacle – taking all the gathered information and using it to build a prototype, which is enough accurate and suitable for real-time applications.
The final system and algorithm, as well as timing analysis, will be discussed in detail in Section 5.

5. FINAL INTEGRATION

5.1. Introduction

In this section, we are about to integrate the research and tools that we have developed in the previous sections into a comprehensive algorithm.

We will describe the design of the algorithm and demonstrate its implementation using a dedicated GUI that we have built.

We will also discuss the design and implementation considerations and analyze its performance on various inputs.

5.2. The Algorithm

As mentioned before, the comprehensive algorithm combines aspects from the previous sections into a final product that meets the requirements of the project.

The purpose of the algorithm is as follows:

- Receive visual input containing **human face** that includes **eyes** with predefined functionality (healthy/palsy eye).
- Perform an estimation of the eyes closure amount and **detect blinks** from the **healthy eye**
- When a start of a blink is detected, **stimulation the paralyzed eye** for improving its closure
- When end of blink is detected, **estimate blinks quality** of the **palsy eye** in order to update the stimulation properties and manage an **adaptive control** of the stimulation.

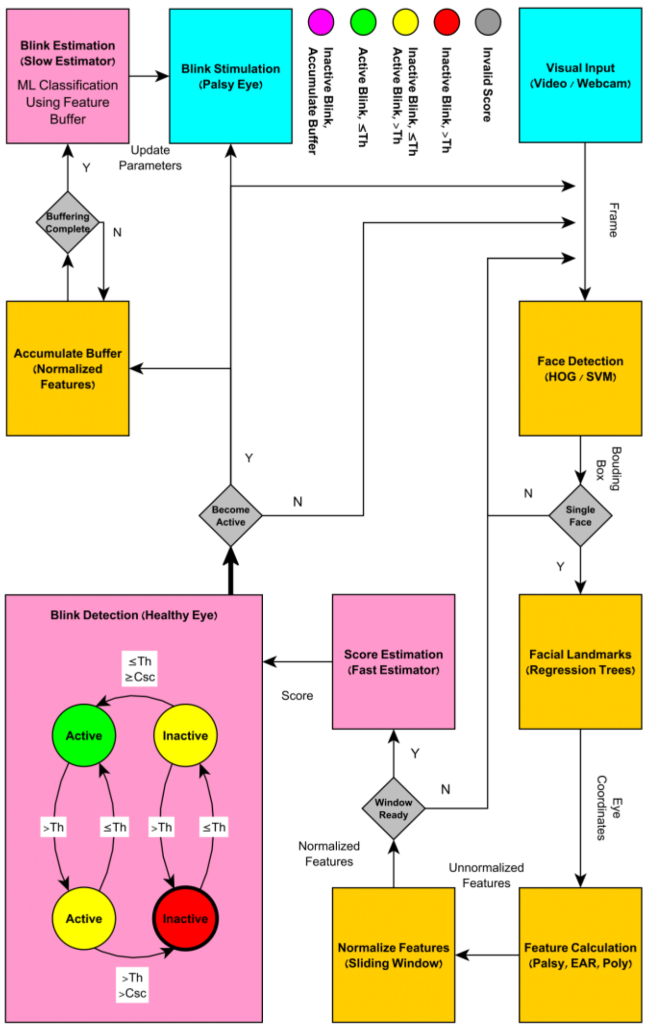
The algorithm is designed to support real-time operation, since the essential processing operations for each frame are faster than the time between grabbing two consecutive frames

- Offline Support: Input video from file
- Real Time Support: Input video from webcam

The described implementation works in an open loop, since the indication for stimulation is generated by the algorithm, but the stimulation itself is outside the project boundaries

5.2.1. Algorithm Schematic Description

A schematic description of the algorithm is provided below



Blink Estimation (Slow Estimator)
ML Classification Using Feature Buffer
Blink Stimulation (Palsy Eye)
Inactive Blink, Accumulate Buffer
Active Blink, ≤Th
Active Blink, >Th
Inactive Blink, ≤Th
Inactive Blink, >Th
Invalid Score
Visual Input (Video / Webcam)
Update Parameters
Y
Buffering Complete
N
Frame
Accumulate Buffer (Normalized Features)
Face Detection (HOG / SVM)
Y
Bouding Box
N
Become Active
N
Single Face
Y
Blink Detection (Healthy Eye)
≤Th ≥Csc
Active
Inactive
>Th
≤Th
>Th
≤Th
Active
Inactive
>Th >Csc
Score Estimation (Fast Estimator)
Score
Facial Landmarks (Regression Trees)
Y
N
Window Ready
Eye Coordinates
Normalized Features
Unnormalized Features
Normalize Features (Sliding Window)
Feature Calculation (Palsy, EAR, Poly)

Figure 5-1: Schematic description of the algorithm

5.2.2. The Algorithm Steps

The algorithm consists of the following steps:

1. **Visual Input:** grab a frame from the visual input (video from file or from webcam)
2. **Face Detection:** perform face detection on the frame and return a bounding box for each face (discussed in section 1)

3. **Facial Landmarks:** if the frame contains a single face, extract the facial landmarks and return the coordinates that describe the eyes (discussed in section I)
4. **Feature Calculation:** extract features for each eye based on the eye coordinates, the following features are used:
    - Palsy: An indication for paralyzed eye
    - EAR: The score using EAR metric
    - Polygon: Area calculation using Shoelace Formula
5. **Normalize Features:** normalize features using a dynamic sliding window. The buffer is not used until it is fully populated with features from previous frames. Therefore, the rest of the algorithm flow continues only once the buffer has reached its full capacity (discussed in section II)
6. **Score Estimation:** if the sliding window is ready, perform deterministic calculation based on the current normalized features, the result considered as the fast estimation of the healthy eye's closure amount
7. **Blink Detection:** use the fast estimation results to perform blink detection for the healthy eye, the blink detection algorithm will be discussed next
8. **Accumulate Buffer:** if a start of a blink is detected, accumulate the following frames into a dedicated features buffer in order to represent the entire blink duration
9. **Blink Estimation:** if the features buffer is ready, use it to perform a ML classification, the result considered as the slow estimation of the palsy eye blink quality. Use the estimation result to update blink stimulation properties
10. **Blink Stimulation:** if a start of a blink is detected, the algorithm generates a stimulus indication for the palsy eye

Steps 6-10 will be discussed in detail next.

5.3. The Estimators

In order to achieve reliable results alongside rapid performance, we chose to implement the algorithm based on two dedicated estimators

5.3.1. Fast Estimator

The fast estimator provides an estimation of the eye closure amount by performing a mathematical deterministic calculation based on the extracted features of the current frame - frame level processing.

The estimator is updated for each frame and is an input for the blinking detection algorithm. It also provides fast performance and relevant for real-time applications.

The estimator, along with the blinking detection algorithm, has two main objectives:

1. Detecting start of blink in the healthy eye, in order to give an indication for stimulating the palsy eye
2. Defining the blink boundaries of the healthy eye, in order to collect features samples representing the palsy eye dynamics during the blink (will be used by the slow estimator to evaluate the blink quality)

5.3.2. Blink Detection

For each frame, the value of the fast estimator is used as an input for the blink detection algorithm.

The algorithm is also influenced by user defined parameters:

- **Th**: A threshold for eye closure amount to consider as a blink
- **Consc**: Number of consecutive frames for which the conditions must be consistently maintained

The algorithm can be described by the following FSM:

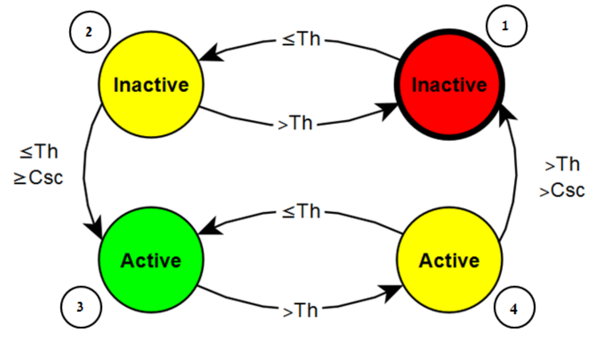


Figure 5-2: FSM describing the blink detection algorithm

1. **Inactive Red (default):** No blinks are detected
    1.1. If value $\leq$ **Th**, go to state 2
2. **Inactive Yellow**: No blinks are detected
    2.1. If value $\leq$ **Th** for at least **Consc** frames, go to state 3
    2.2. If value > **Th**, go to state 1

3. **Active Green:** A blink is detected
   3.1. If value > **Th**, go to state 4
4. **Active Yellow:** A blink is detected
   4.1. If value ≤ **Th**, go to state 3
   4.2. If value > **Th** for at least **Consc** frames, go to state 1

When a start of a blink is detected (switching from state 2 to 3), the following events occur simultaneously:

- A proper stimulation is given for paralyzed eye (based on previous blink's quality estimation by the slow estimator)
- Accumulating features into a dedicated buffer in order to represent the entire blink dynamic, which will be the basis for classification by the slow estimator

5.3.3. Accumulate Buffer

When a start of a blink is detected, features representing eye dynamics during the blink are accumulated into a dedicated buffer. Accumulation is affected by a user-defined parameter:

- **Buffer:** number of samples (row of features for a single frame) to store before the blink starts and after it ends

The features buffer is used as an input for the slow estimator

5.3.4. Slow Estimator

The slow estimator provides an estimation of the eye blink quality by performing a ML based classification (discrete levels) on the features buffer that represents the blink - blink level processing (discussed in section IV). Note that the chosen classifier wasn't trained on the test video; we split the train and test sets so that the train set includes blinks only from different videos.

The estimator is updated after each blink and used for updating the properties of stimulation. The required stimulation properties are based on the gap between the achieved and the required blink quality.

The estimator provides reliable results and relevant for implement an adaptive stimulation for the palsy eye.

5.3.5. Blink Stimulation

Blink stimulation is managed by the fast and slow estimators:

- When a start of a blink is detected in the healthy eye, a proper stimulation indication is given for the paralyzed eye
- After the blink ends, the blink quality of the paralyzed eye is estimated by the slow estimator and the required stimulation properties are updated based on the gap between the achieved and the required blink quality (trigger calculation).

It is important to note that the stimulation itself is outside the project's boundaries. Future developments can use the trigger calculation and map it to appropriate electrical characteristics for generating an actual stimulation.

5.4. Integration and Indication

We chose to encode the algorithm status during its operation, using color-based indication.

The motivation is to illustrate all the processes described before, and allow the user to be impressed by them during the operation of the algorithm:

- **Gray: fast estimation is not available**
  - Multiple faces were detected / face wasn't detected
  - The feature normalization sliding window is not ready
- **Red: Blink inactive**
  - No blink detection, fast estimation value > **Th**
- **Yellow: Blink status decision point**
  - Blink active, fast estimation value > **Th**
  - Blink inactive, fast estimation value is ≤ **Th**
- **Green: blink active**
  - Blink detected, fast estimator value ≤ **Th**
- **Violet: Accumulate Buffer**
  - Blink is finished, accumulate buffer for slow estimator

The indications will be expressed as an indication LED embedded in the GUI, we will discussed next

- The LED color represents the algorithm's status

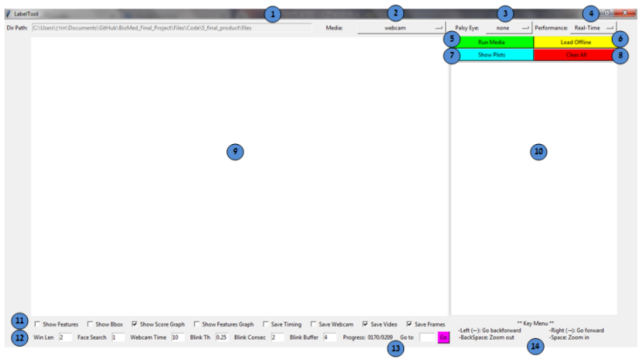
- The number appears in the LED's center represents the number of blink detected so far
- Values beneath the LED represent blink quality estimation for both eyes, and the trigger calculation for palsy eye stimulation, Those values are updated after each blink

5.5. The GUI

In order to let the user run the algorithm on various inputs, provide convenient result viewing, and support debugging tools for analyzing it, we built a dedicated graphic user interface (GUI).

5.5.1. GUI Structure

The following figure shows the GUI divided into indexed regions.



Dir Path: C:\Users\יוחי\Documents\GitHub\BioMed_Final_Project\Files\Code\5_final_product\files
Media: webcam
Palsy Eye: none
Performance: Real-Time

☐ Show Features ☐ Show Bbox ☑ Show Score Graph ☐ Show Features Graph ☐ Save Timing ☐ Save Webcam ☑ Save Video ☑ Save Frames

Win Len 2 Face Search 1 Webcam Time 10 Blink Th 0.25 Blink Consec 2 Blink Buffer 4 Progress: 0170/0209 Go to

** Key Menu **
-Left (←): Go backforward
-BackSpace: Zoom out
-Right (→): Go forward
-Space: Zoom in

Figure 5-3: The final GUI, divided into indexed regions

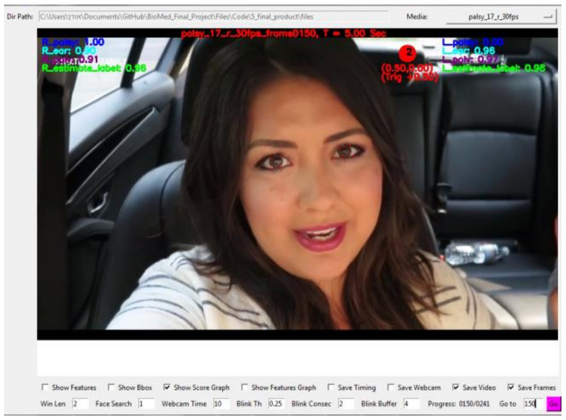
We explain the GUI structure using the indexed regions:

1. **Dir Path:** entire directory path for saving the results, automatically set considering project folder hierarchy.
2. **Media:** a list containing available inputs for running, first choice is video from webcam, other choice are video files.
3. **Palsy Eye:** setting palsy eye for the input (none, left, right), chosen automatically for video file, according to its name:

   normal/palsy_xx_y_zzfps

   Where xx is the video number, y is the 'paralyzed' eye ('l' for left, 'r' for right or 'n' for none), zz is the fps.
4. **Performance:** chose if to run the input for offline or for real time application (affects output display at runtime).
5. **Run Media:** a button used for running the algorithm on the selected media (running mode will be discussed later).
6. **Load Offline:** a button used for loading offline results for debug and analysis (debug mode will be discussed later).
7. **Show Plots:** a button used for display result graphs of running mode (after it was successfully completed).
8. **Clear All:** a button used to clear all canvases.
9. **Primary Canvas:** a canvas used for present different items depending on the working mode:

   a. **Running mode**: display the current frame of the currently running video (Runtime view) with the indication LED (discussed in section 5.4) located next to the healthy eye.



Dir Path: C:\Users\יוחי\Documents\GitHub\BioMed_Final_Project\Files\Code\5_final_product\files
Media: palsy_17_r_30fps

☐ Show Features ☐ Show Bbox ☑ Show Score Graph ☐ Show Features Graph ☐ Save Timing ☐ Save Webcam ☑ Save Video ☑ Save Frames

Win Len 2 Face Search 1 Webcam Time 10 Blink Th 0.25 Blink Consec 2 Blink Buffer 4 Progress: 0150/0241 Go to 150

Figure 5-4: Running mode view of the primary canvas during video runtime (for palsy_17_r_30fps.mp4)

After video processing is successfully finished, result graph is displayed. The graph shows the fast estimation score vs. frame number and elapsed time (for both eyes). Blinks boundaries are marked in the graph (yellow mark for palsy eye, green mark for healthy eye).

Slow estimation is also displayed (red mark) within the blink boundaries of the palsy eye.

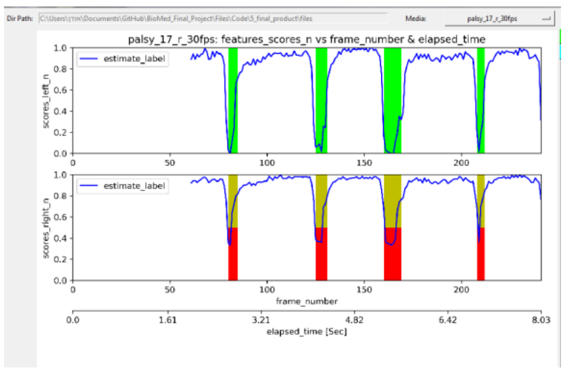


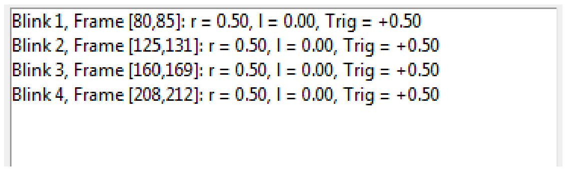
Figure 5-5: Running mode view of the primary canvas after video processing is finished (for palsy_17_r_30fps.mp4)

b. **Debug mode**: display the selected frame of the analyzed video (Offline view).

See **Figure 5-4** for illustration.

10. **Secondary Canvas:** a canvas used for present different items depending on the working mode:

a. **Running mode**: display blinks log, after each end of blink, a unique row is created. The row includes details about: blink number, blink boundaries (in terms of frame numbers), blink quality estimation for each eye and stimulation trigger calculation for the palsy eye



```
Blink 1, Frame [80,85]: r = 0.50, l = 0.00, Trig = +0.50
Blink 2, Frame [125,131]: r = 0.50, l = 0.00, Trig = +0.50
Blink 3, Frame [160,169]: r = 0.50, l = 0.00, Trig = +0.50
Blink 4, Frame [208,212]: r = 0.50, l = 0.00, Trig = +0.50
```

Figure 5-6: Running mode view of the secondary canvas during video runtime (for palsy_17_r_30fps.mp4)

b. **Debug mode**: display the result graph. The graph shows the fast estimation score vs. frame number and elapsed time (for both eyes). Blinks boundaries are marked in the graph (yellow mark for palsy eye, green mark for healthy eye). Slow estimation is also displayed (red mark) within the blink boundaries of the palsy eye.

Current frame is marked on the plot using a black dot, enabling a dynamic debugging.

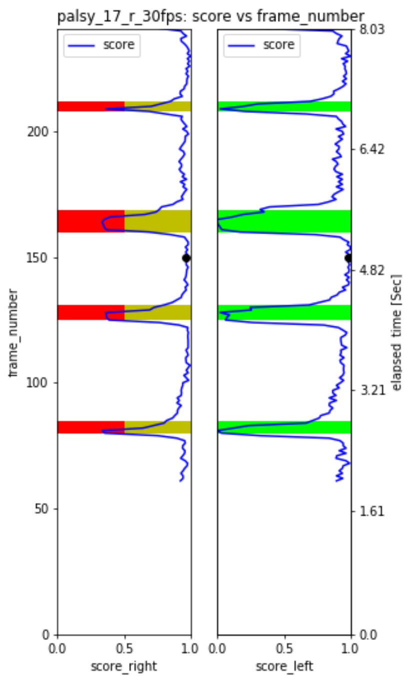


Figure 5-7: Debug mode view of the secondary canvas, frame number 150 (for palsy_17_r_30fps.mp4)

11. **Check Box Parameters:** list of ON-OFF user defined parameters, affects media processing and output products (have to be defined before clicking on "Run Media" Button)

a. **Show Features**: allows for visualization of features on the current frame

b. **Show BBox**: show face bounding box on the current frame

c. **Show Score Graph**: show result graph

d. **Show Features Graph:** show features in result graph

e. **Save Timing:** generate a graph of timing analysis

f. **Save Webcam**: save raw webcam video to database

g. **Save Video**: save processed frames to video

h. **Save Frames**: save processed frames to folder (essential for offline debugging)

Parameter	Default	Parameter	Default
Show Features	OFF	Save Timing	OFF
Show BBox	OFF	Save Webcam	OFF
Show Score Graph	ON	Save Video	ON
Show Features Graph	OFF	Save Frames	ON

Table 5-1: default values for Check Box Parameters

12. **Entry Parameters:** list of numeric user defined parameters, affects media processing and output products (have to be defined before clicking on "Run Media" Button)

   a. **Win Len**: sliding window length (in seconds), used for features normalization

   b. **Face Search**: number of frames between consecutive face detection

   c. **Webcam Time**: length of webcam input (in seconds)

   d. **Blink Th**: threshold for eye closure amount to consider as a blink (discussed in 5.3.2)

   e. **Blink Consec**: Number of consecutive frames for which the conditions must be consistently maintained. Discussed in 5.3.2.

   f. **Blink Buffer**: number of samples to store before blink starts and after blink ends (sample = row of features for single frame). Discussed in 5.3.3.

Parameter	Default	Parameter	Default
Win Len	2	Blink Th	0.25
Face Search	1	Blink Consec	2
Webcam Time	10	Blink Buffer	4

Table 5-2: default values for Entry Parameters

13. **Debug mode navigation:** navigation tools for debug mode (after the user click on "Load Offline" button).

   a. **Progress**: display the current frame index and the total frame number

   b. **Go to Image:** enable moving to selected frame, set frame index and press "Go".

14. **Key Menu:** keys shortcut for debug mode

   a. **<< Prev:** moving to previous frame (left arrow ⟵).

   b. **>> Next:** moving to next frame (right arrow ⟶).

   c. **Zoom In:** zoom in (Key = Space).

   d. **Zoom Out:** zoom out (Key = Back Space).

5.6. Operating The GUI

We will now sum up the main steps of the GUI operation, which let the user run the comprehensive algorithm on various inputs, provide convenient result viewing, and support debugging tools for analyzing it.

5.6.1. Select Input Media

- Use "Media" list to choose input media (from webcam/file)
- Use "Palsy Eye" list to define the palsy eye in the input media (will be defined automatically for files)
- Use "Performance" list to choose if to run the input for offline or for real time application (affects output display)

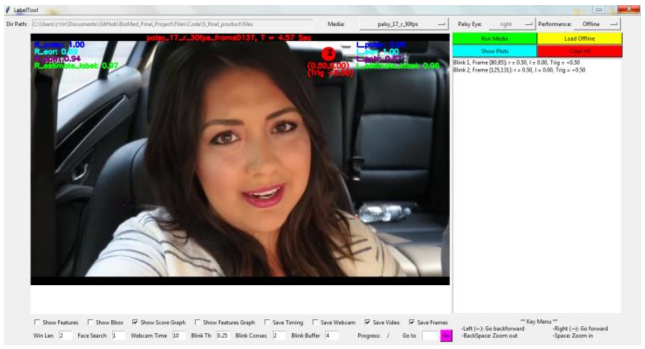
5.6.2. Define Parameters

- Define Check Box Parameters (ON/OFF) which affects media processing and output products
- Define Entry Parameters (Numeric) which affects media processing and output products

5.6.3. Running Mode

Press "Run Media" button to start the running mode.

- Primary canvas displays current frame of running video with indication LED located next to the healthy eye. In addition, the values of the features and the quick revaluation are also shown for each eye.
- Secondary canvas display blinks log, includes unique row for each blink

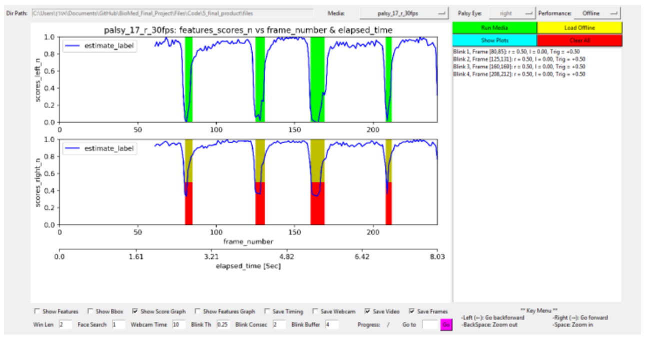


Blink 1, Frame [80,85]: r = 0.50, l = 0.00, Trig = +0.50
Blink 2, Frame [125,131]: r = 0.50, l = 0.00, Trig = +0.50

Figure 5-6: Running mode view of the GUI during video runtime (for palsy_17_r_30fps.mp4)

The running mode continues until the video is finished.

- After video processing is successfully finished, result graph is displayed in the primary canvas



```
Blink 1, Frame [80,85]: r = 0.50, l = 0.00, Trig = +0.50
Blink 2, Frame [125,131]: r = 0.50, l = 0.00, Trig = +0.50
Blink 3, Frame [160,169]: r = 0.50, l = 0.00, Trig = +0.50
Blink 4, Frame [208,212]: r = 0.50, l = 0.00, Trig = +0.50
```

Figure 5-7: Running mode view of the GUI canvas after video processing is finished (for palsy_17_r_30fps.mp4)

Result graph and blinks log will be also displayed by pressing "Show Plots" button

5.6.4. Clear Canvases

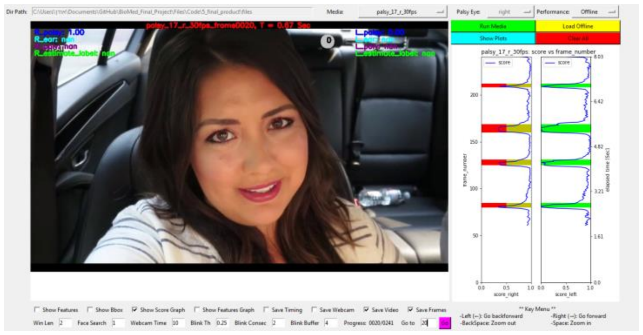
Canvases will be cleared by pressing "Clear All" button

5.6.5. Debug Mode

Press "Load Offline" button to start the debug mode.

This mode will be possible only if the selected media has been successfully processed in running mode and "save frames" parameter was ON.

- Primary canvas displays selected frame of analyzed video.
- Secondary canvas displays the result graph, current frame is marked on the plot using a black dot, enable a dynamic debugging.



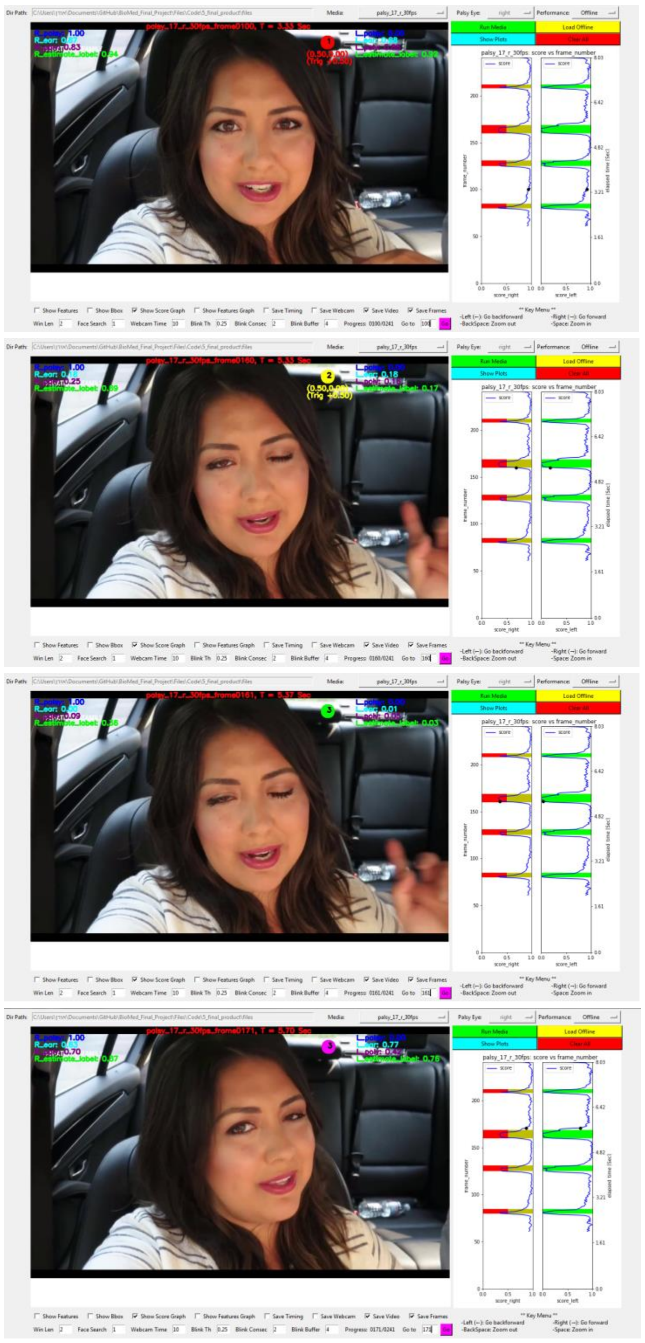


Figure 5-8: Debug mode view of the GUI during video analyze, all possible indication led states are shown (for palsy_17_r_30fps.mp4)

5.7. Results

We will review the results for healthy people and palsy people

5.7.1. A Healthy Woman (Webcam_03_n_15fps)

This video was taken from a webcam, blinks were subjectively observed at seconds: 1.80, 3.00, 4.53, 5.60, 7.00, 8.33, 9.53, 11.13, 12.47, 14.20.

The observed blinks were normal with full opening and closing of the eyes.

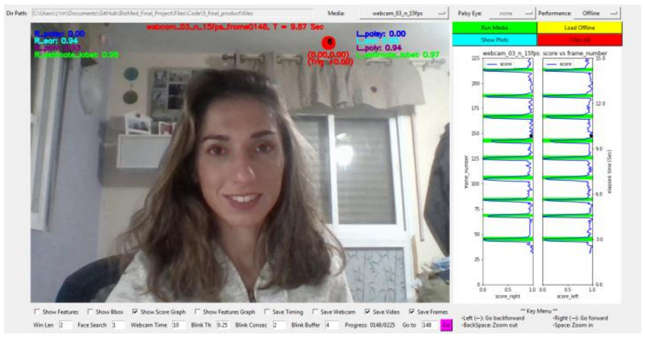


Figure 5-9: Processing results for webcam_03_n_15fps.mp4

As we can see, the results shown in the graph are very well matched to the observed dynamic. As expected, the first blink (at 1.8Sec) does not appear because the normalization sliding window was not yet ready (window length = 2Sec)

5.7.2. A Healthy Man (Noraml_07_n_30fps)

This video was taken from a file, blinks were subjectively observed at seconds: 0.93, 2.50, 3.40, 4.23, 5.87, 7.17, 7.90, 9.03

The observed blinks were normal with full opening and closing of the eyes.

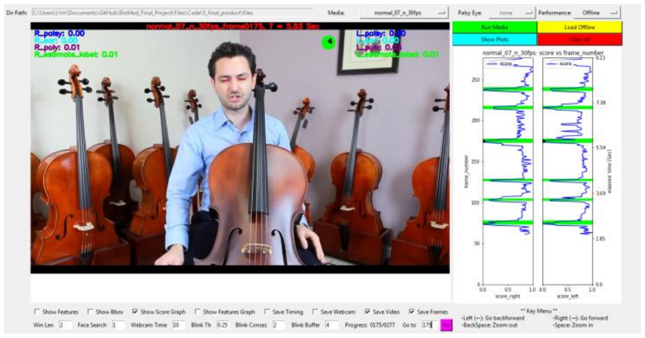


Figure 5-10: Processing results for noraml_07_n_30fps.mp4

As we can see, the results shown in the graph are very well matched to the observed dynamic. As expected, the first blink (at 0.93Sec) does not appear because the normalization sliding window was not yet ready (window length = 2Sec), and the last blink does not appear because it is near video's end and the blink buffer filling requirement is not met for it.

5.7.3. A Paralyzed Woman (Plasy_05_l_25fps)

This video was taken from a file, blinks were subjectively observed in the healthy eye at seconds: 0.44, 1.71, 2.28, 2.80, 3.40, 3.88, 4.48, 5.36, 6.12, 7.12

The observed blinks of the healthy eye were normal with full opening and closing of the eyes, partial blinks of the palsy eye were observed simultaneously and was ranged between a quarter to half of eye's full closure.

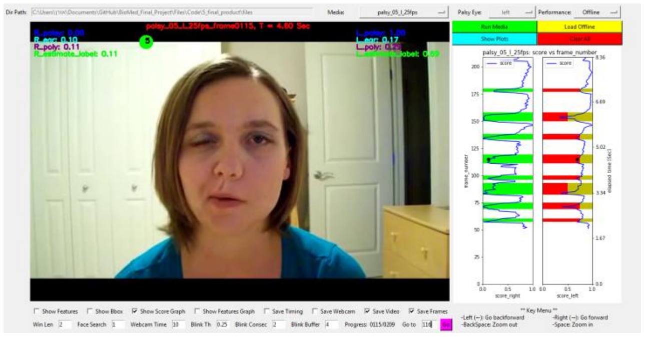


Figure 5-11: Processing results for palsy_05_l_25fps.mp4

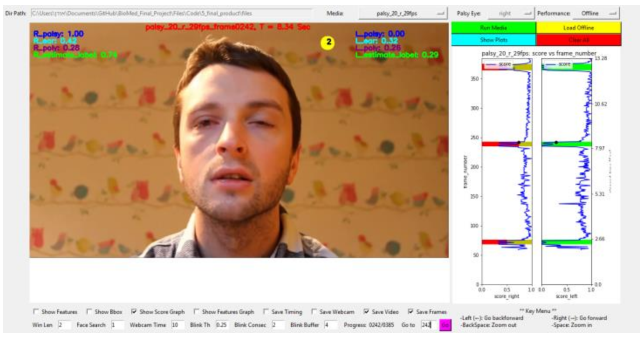
As we can see, the results shown in the graph are very well matched to the observed dynamic, for the healthy eye and for the palsy eye. We can also see that the palsy eye blink quality was well estimated (partial closure).

As expected, the first two blinks (at 0.44Sec, 1.71Sec) does not appear because the normalization sliding window was not yet ready (window length = 2Sec).

5.7.4. A Paralyzed Man (Palsy_20_r_29fps)

This video was taken from a file, blinks were subjectively observed in the healthy eye at seconds: 2.52, 8.24, 12.76.

The observed blinks of the healthy eye were normal with full opening and closing of the eyes, partial blinks of the palsy eye were observed simultaneously and was ranged between a quarter to half of eye's full closure.



Dir Path: C:\Users\ןתיי\Documents\GitHub\BioMed_Final_Project\Files\Code\5_final_product\files Media: palsy_20_r_29fps Palsy Eye: right Performance: Offline

palsy_20_r_29fps_frame0242, T = 8.34 Sec

R_palsy: 1.00
R_ear: 0.42
R_poly: 0.26
R_estimate_label: 0.74

L_palsy: 0.00
L_ear: 0.32
L_poly: 0.26
L_estimate_label: 0.29

☐ Show Features ☐ Show Bbox ☑ Show Score Graph ☐ Show Features Graph ☐ Save Timing ☐ Save Webcam ☑ Save Video ☑ Save Frames

Win Len 2 Face Search 1 Webcam Time 10 Blink Th 0.25 Blink Consec 2 Blink Buffer 4 Progress: 0242/0385 Go to 242

** Key Menu **
-Left (←): Go backforward -Right (→): Go forward
-BackSpace: Zoom out -Space: Zoom in

Figure 5-12: Processing results for palsy_20_r_29fps.mp4

As we can see, the results shown in the graph are very well matched to the observed dynamic, for the healthy eye and for the palsy eye. We can also see that the palsy eye blink quality was well estimated (partial closure).

5.8. Performance

5.8.1. Performance Review

Average running times of relevant processing stages:

Stage	Average Time [ms]	Frequency
grab_frame	6.94	Every frame
detect_face	23.89	For non-static face
facial_landmarks	4.18	Every frame
features_calculation	0.16	Every frame
normalize_features	0.03	Every frame
fast_estimation	0.15	Every frame
blink_detection	0.69	Every frame
slow_estimation	17.58	Every Blink
save_to_dataframe	0.02	Every frame
display_frame	14.50	For debug only

Table 5-3: average running times of relevant processing stages. The last column shows the execution frequency of each stage

The running times were recorded in "Real-Time" performance mode (affects mainly display_frame time).

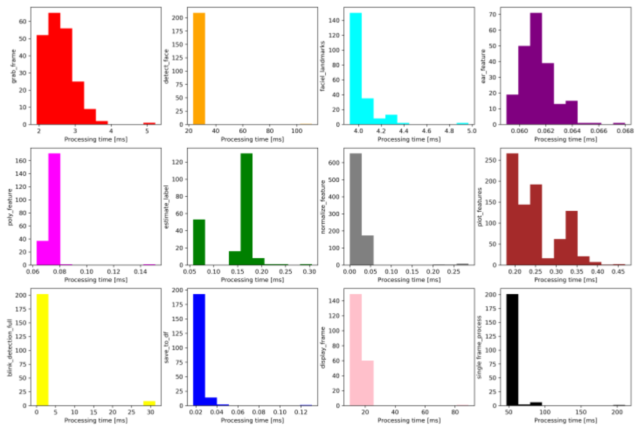


Figure 5-13: Running time histogram for palsy_05_l_25fps.mp4

There is a relatively large variance in the times calculation since they depend also on the computer's hardware and background processes. Therefore, we averaged the running times on the input videos presented in the sub-section 5.7.

The code was tested on a laptop with the following specifications:

- **Operation System**: Windows 7 Ultimate, 64-bit
- **Processor**: Intel Pentium CPU 2020M @ 2.4GHz
- **RAM**: 4 GB

We divided the processing stages into three types, according to the colors in table 5-3:

1. **Required every frame:** Average time of 12.16 ms
2. **Required every blink:** Average time of 17.58 ms

Therefore, based on the required stages, we see that the implementation supports real-time performance at a typical frame rate of 30fps regardless the blinking frequency.

3. **Not required:** Average time of 39.39 ms

Displaying frame is necessary for debugging only. Face detection is not required after the initialization stage, because face movement expected to be negligible, especially when the system is fixed to the user's face.

5.8.2. Runtime Improvement

During development, we improved the running time in the following ways:

- Conducting massive and one-time processing stages before and after the video processing (e.g. initializations, save output products, plot graphs, and more)
- Face detection is done relative to the current face location
- Using cost-effective data structures that require minimal processing time
- Functional separation between the fast estimator (blink detection, stimulation indication) which is calculated for each frame with fast processing time, and the slow estimator (blink quality estimation) which is calculated after each blink with slow processing time

5.9. Summary

In this section, we took all the work done in the previous sections and integrate into a final product that meets the project requirements.

We designed a comprehensive algorithm, based on two dedicated estimators, and implemented it using a graphical user interface that allows for:

- Input from video file or camera
- Extraction of features
- Estimation using the chosen classifier
- Displaying the results on the screen

Finally, we presented an analysis of results and performance and saw that we've achieved reliable results alongside real-time performance.

6. CONCLUSIONS

6.1. Project Summary

During this project, we tried to find a solution to the most common case of facial paralysis – Bell's palsy disease.

Patients with Bell's palsy suffer from a complete paralysis of one side of the face. In most cases, patients will recover from Bell's palsy; however:

- Full recovery may take up to 12 months to complete
- Peak ages are 15 - 45 years, so rapid rehabilitation is crucial!
- The paralysis might have long-term effects, most of them are related to problems with blinking or closing the eye. This would leave the cornea dry, which cause pain, infections and even lead to blindness.

Moreover, there are no available solutions which are not invasive and effective in advanced stages.
Therefore, we saw a great potential in designing a system that might relieve patients' symptoms, as well as decreasing recovery times. We came up with the idea of a feedback system, based on standard cameras, that will use the normal blinking template of the healthy, try to estimate the paralyzed eye's amount of closure and use the information of the healthy eye to generate an adequate electrical pulse that will bring to a full closure. We are based on the following assumptions:

1. The muscle responsible for blinking, the orbicularis oculi, is known and traceable. Stimulating it will force the paralyzed eye to close. We found relevant data from previous researches.
2. The weaker the stimulation is, the more work needs to be done by the paralyzed muscle in order to reach a full closure. Therefore, finding the minimal trigger level is important.
3. Applying smart electrical stimulation might lead to a rapid recovery and to a relief in symptoms.

For the scope of this project, we only deal with the estimation and not with the electrical stimulation.
We divided the project into 5 individual sections, each section is based on its predecessors.

6.1.1. Summary of Section 1

In Section 1, we looked for methods for face and eye extraction from image. We used the extraction results to find features relevant for classification. We chose to use dlib library, embedded in Python, for the task of face detection. The embedded detector is based on Histograms of Oriented Gradients (HOGs), which we also used as features for Section 2. We use the bounding box containing a face, which is the output of the face detector, for the task of eye detection. We chose to use the facial landmarks detector, an accurate detector that was trained on thousands of face images. This detector returns 63 points that represent different parts in the face. Among them, there are 12 points (6 for each eye) that were the basis for features extraction, done in Section 2.

6.1.2. Summary of Section 2

In Section 2, we used the outputs of the landmarks detector to extract feature that might be useful in training and have a strong correlation with eye dynamics. We explored 5 different features – 4 of them are based directly on the 6 points returned by the detector (EAR, ellipse area, Shoelace formula and linear regression), and one is based on the pixels within the bounding box defined by those six points (HOG). We chose to continue with EAR and Shoelace formula as they both perform well with respect to subjective tagging and are suitable for real-time application. Nevertheless, we continued to examine HOG features as we noticed some cases where the 6 points fail to describe well the eye's boundaries.

6.1.3. Summary of Section 3

In Section 3, we add labels to the extracted features, as the first next step for database creation. We chose a scale of 5 levels, separated equally between the range of 0% to 100%. Next, we used this scale to build a semi-automatic graphic user interface that is used for manual tagging. This tool allows for frame-by-frame tagging in a convenient way. It also shows graphs containing the already-labelled frames, as well as the labels given by other users. With this tool we managed to tag almost 20,000 frames!

6.1.4. Summary of Section 4

In Section 4, we combined features and labels into one large database. We then filtered out sequences longer than 4 consecutive frames that had identical labels, in order to avoid bias to a specific label. By doing that, we left with 5,115 samples that are almost equally split throughout the dataset. The final steps were assigning each sample to a specific blink number and defining two approaches for accuracy estimation: 'frame-wise' and 'blink-wise'. We applied various classifier using two different feature vectors, while accounting also the features of the neighboring frames to maximize the results, we found out that SVM with RBF kernel, using the first feature vector, gave the best results of ~74% for 'blink-wise' scoring. If we reduce the number of levels to 3, we will get ~92% for 'blink-wise', which are excellent results for this task.

6.1.5. Summary of Section 5

In Section 5, we took all the work done in the previous sections and create a graphical user interface that allows for the input from video file or camera, the extraction of features, the estimation using the chosen classifier and the display of the results on the screen. As the classification is done 'blink-wise', it is done for an entire blink – so we need a tool that is responsible for separating the video into sequences of blinks. This is where the 'fast' estimator comes in handy, using simple calculations which are relevant for real-time applications. Each estimation sets the amount of electrical stimulation to be applied for the next blink. This application reaches great accuracies for different videos, including healthy patients and patients with Bell's palsy.

6.2. Future Work

We got classification results that are quite satisfying, but can be improved using more accurate eye detectors, higher resolution videos or higher frame rates. By increasing those, we can reach more samples for each level, augmenting the database while improving accuracy. Moreover, more samples apply for classification using higher number of levels.

Future work might include increasing also the number of blinks, by using more videos of patients. As our database is based entirely on YouTube, we are limited by image resolution and frame rate. The problem worsens in the case of Bell's palsy patients, as there are relatively small number of YouTube videos to use for learning. In order to reach more reliable results, we might take our own videos.

Although we proved the system concept as suitable for real-time processing, some work yet needs to be done in order to reach a prototype. The final system may be assembled on glasses, so the cameras 'see' only the eyes and not the whole face. Therefore, the landmarks detector, that assumes an input of an entire face, may not be relevant for the final application. In addition, the code is written in Python, and has to be optimized for embedded implementation. We hope that this will improve processing times, maybe allowing us to use advanced learning techniques.

Of course, a lot of work has to be done on the electrical stimulation that has to be applied to reach a full closure. One should consider amplitudes, frequencies and exposure times that will make the best results, while guaranteeing safety. Would this architecture be relevant as a cure for Bell's palsy patients? Only time will tell.

7. BIBLIOGRAPHY

[1] Frigerio A, Heaton JT, Cavallari P et al. (2015) "Electrical stimulation of eye blink in individuals with acute facial palsy: progress toward a bionic blink." PlastReconstrSurg 136:515e–523e DOI 10.1097/PRS.0000000000001639

[2] Rantanen V, Vehkaoja A, Verho J et al. (2016) "Prosthetic pacing device for unilateral facial paralysis." XIV Mediterranean Conference on Medical and Biological Engineering and Computing (MEDICON 2016), pp. 647-652 DOI 10.1007/978-3-319-32703-7_126

[3] Frigerio A, Cavallari P, Frigeni M, Pedrocchi A, Sarasola A, Ferrante S. (2014) Surface EMG mapping of the orbicularis oculi muscle for real-time blink detection. JAMA Facial Plast Surg .

[4] Frigerio A, Brenna S, Cavallari P. (2013) "Surface electromyography recording of spontaneous eyeblinks: Applications in neuroprosthetics". Otolaryngol Head Neck Surg. 148:209–214.

[5] Frigerio A, Hadlock TA, Murray EH, Heaton JT (2014) "Infrared-based blink-detecting glasses for facial pacing: toward a bionic blink", JAMA Facial PlastSurg 16(3):211–218 DOI 10.1001/jamafacial.2014.1

[6] Hoang, L., Thanh, D. & Feng, L. (2013) "Eye Blink Detection for Smart Glasses" IEEE International Symposium on Multimedia (ISM 2013), pp.306-308, IEEE, Anaheim, CA, 2013.

[7] Nakaso S., Güttler J., Mita A. and Bock T. (2015) "A Fatigue Detection System implemented in an Office Deployable Gateway based on Eye Movement". In Proceedings of the CIB*IAARC W119 CIC 2015 Workshop, pages 39-46, Munich, Germany, 2015.

[8] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In CVPR, 2005.

[9] https://www.learnopencv.com/histogram-of-oriented-gradients/

[10] https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/

[11] V. Kazemi and S. Josephine. One millisecond face alignment with an ensemble of regression trees. In CVPR, 2014.

[12] T. Soukupova and J. Cech, "Real-time eye blink detection using facial landmarks," in 21st Computer Vision Winter Workshop (CVWW2016), 2016, pp. 1–8.

[13] Shoelace Formula in Wikipedia: https://en.wikipedia.org/wiki/Shoelace_formula

[14] https://www.learnopencv.com

[15] F. Song, X. Tan, X. Liu, S. Chen, Eyes closeness detection from still images with multi-scale histograms of principal oriented gradients, Pattern Recognit. 47 (9) (2014) 2825–2838.

[16] Wikipedia - Blinking (https://en.wikipedia.org/wiki/Blinking)

[17] S. Zafeiriou, G. Tzimiropoulos, and M. Pantic. The 300 videos in the wild (300-VW) facial landmark tracking in-the-wild challenge. In ICCV Workshop, 2015. http://ibug.doc.ic.ac.uk/resources/300-VW./

[18] https://towardsdatascience.com

[19] https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/

8. APPENDIX: RUNNING PYTHON SOURCE CODE

8.1. Introduction

In this section, we will provide steps that allows for execution of our source code and code examples.

8.2. Installing Visual Studio 2015

Download and install Visual Studio 2015 community edition from https://www.visualstudio.com/vs/older-downloads/ or via your Visual Studio Developer Essentials account. Please note that we switch back to 2015 version since Visual Studio 2017 fails to compile dlib – one of the two fundamental Python libraries for image processing and machine learning.
Run installer, select “Custom” in “type of installation”. In next screen within Programming Languages, select Visual C++ and Python tools for Visual Studio. Click next. Click next again and wait until the installation completes.

8.3. Installing CMake

Download and install the latest version of CMake from https://cmake.org/download/. During installation select "Add CMake to installation PATH".

8.4. Installing Anaconda

Download and install Anaconda 64-bit version from https://www.continuum.io/downloads. Choose and install Anaconda 3. Please note that dlib ships only a prebuild binary for Python 3 and not Python 2, so Python 3 is necessary. While installing Anaconda make sure that you check both options:

1. Add Anaconda to my PATH environment variable
2. Register Anaconda as my default Python

8.4.1. Create Virtual Environment

Open the Anaconda prompt and execute the following command:

```
conda create --name opencv-envpython=3.6
```

Press enter and make sure that the creation completes without errors.

8.4.2. Activate the Environment

Execute the following command:

```
activate opencv-env
```

The (opencv-env) now appears before the prompt.

8.4.3. Install OpenCV and Supportive Packages

Using (opencv-env) as environment, run the following line for download and install Python's supportive packages:

```
conda install -c conda-forgexxxx or
pip install xxxx
```

Where xxxx stands for the library to be installed.
A list of the necessary libraries is given in Table A-1.

Name	Description
numpy	The fundamental package for scientific computing with Python
scipy	A Python-based ecosystem of open-source software for mathematics, science, and engineering
matplotlib	A Python 2D plotting library
scikit-learn	Machine Learning for Python
jupyter	An open-source web application that contains live code, equations, visualizations and narrative text.
opencv-python	An open source computer vision and machine learning software library
dlib	A modern C++ toolkit containing machine learning algorithms and tools for creating complex software in C++ to solve real world problems
imutils	A python package containing a series of OpenCV convenience functions
pandas	A Python package providing fast, flexible, and expressive data structures designed to make working with “relational” or “labeled” data both easy and intuitive.

openpyxl	A Python library to read/write Excel 2010 xlsx/xlsm/xltx/xltm files
pillow (PIL)	The friendly PIL fork by Alex Clark and Contributors. PIL is the Python Imaging Library by Fredrik Lundh and Contributors.
xlrd	A library for reading data and formatting information from Excel files, whether they are .xls or .xlsx files.
tkinter	Python's standard GUI (Graphical User Interface) package

Table 8-1: Description of the necessary Python libraries needed for running the project's source codes

8.4.4. Test Your Installation

Open the Python prompt by typing python on the Anaconda prompt. Make sure that these commands are executed without errors:

- `import cv2`
- `cv2.__version__`
- `import dlib`
- `dlib.__version__`

8.5. Running Section 2 Source Code

Navigate to the project folder, then change directory to Files/Code/2_feature_extraction. This folder consists of the following:

- files – a directory containing part of the output files of Section 2
- feature_extraction_webcam (/video) –Python source codes for Section 2 (for input being generated via web camera or from file, respectively)
- feature_extraction_video_full – Extension of feature_extraction_video, allows for feature extraction also for the first (wlen x fps) frames (works by running the video file backwards)
- crop_eyes – Python source codes for cropping eye images, to be used in Section 4
- run_xxx (xxx for user name) – batch file for running feature_extraction_video.py, feature_extraction_video_full.py or feature_extraction_webcam.py
- run_xxx_eyes (xxx for user name) – batch file for running crop_eyes.py
- run_xxx_eyes_db (xxx for user name) – batch file for running crop_eyes.py on each video in the database

8.5.1. Running xxx.bat

To run the batch file, change the following lines according to your system setup:

- Line 13: Change the path to the path where Anaconda3 is installed
- Line 22: In case of Windows 10 user, change the path to your project path (/Files/Code/2_feature_extraction)
- Line 24: In case of any other operating system, change 3.6 to the current installed version of Python
- Line 25: In case of any other operating system, change the path to your project path (/Files/Code/2_feature_extraction)
- Line 38: Change to your desired configuration. The parameters that can be modified are given in Table A-2.

Name	Description
--ear	Calculate the EAR feature. '0' to disable (default: 1)
--r2	Calculate the linear regression feature. '0' to disable (default: 0)
--ellipse	Calculate the ellipse area feature. '0' to disable (default: 1)
--poly	Calculate the Shoelace formula feature. '0' to disable (default: 1)
--wlen	The initial length of the sliding window in seconds (default: 2)
--face_frame	Detect faces every x frames (default: 1)
--pictures	Save a picture of each frame with an illustration of the extracted features, and create mp4 file containing the results (default: 0)
--tag	Whether to save frames for tag (default: 0)
--graphs	Plot graphs containing normalized scores of all the features for both eyes (default: 0)
--times	Plot histograms of performance analysis (default: 0)

Table 8-2: Description of section 2 source code settings

While running the batch file, one can choose between the following configurations:

- Input source ([w]ebcam or [v]ideo)
- Location of the palsy eye ([l]eft, [r]ight or [n]one)
- For video source: the video index (palsy/normal_xx where xx is the video number). The relevant input files are located under Files/Database directory
- For video source: an additional option for running the video backwards (_full), in order to fill the missing scores of the frames from the beginning of the video (note that as we started normalizing only once the sliding window reached its initial capacity)

The outputs of this script are:

- ***<video_name>*_scores_n.xlsx** – An Excel file containing the normalized scores of all the features and for both eyes. To be used in Section 4.
- **frames_for_tag** - a directory containing the frames extracted from the input source, including their elapsed time. These files are saved in Section 3 directory (Files/Code/3_database_tagging/files) and are very useful while using the semi-automatic labeling tool.
- **frames_features** - a directory containing the frames extracted from the input source, including their elapsed time and an illustration of the extracted features. This is an optional output and can be set by changing –pictures parameter.
- ***<video_name>*_out.mp4** – video file containing the frame extracted from the input source, including their elapsed time and an illustration of the extracted features.This is an optional output and can be set by changing –pictures parameter.
- ***<video_name>*_scores_graphs.png -** graphs containing normalized scores of all the features for both eyes.
- For webcam input: mp4 file containing the frames extracted from the camera (the amount of FPS is estimated automatically). This file is saved in Database directory

8.5.2. Running run_xxx_eyes.bat

This script is suitable for video files only. To run this batch file, follow instructions on A.5.1.
While running the batch file, choose v for video input source, and then specify the video index (palsy/normal_xx where xx is the video number. The relevant input files are located under Files/Database directory.

The output of this script is **eye_pics**, a directory containing the pictures of eyes which were cropped out of the frame. These pictures are the input for Section 4.

8.6. Running Section 3 source code

Navigate to the project folder, then change directory to Files/Code/3_database_tagging. This folder consists of the following:

- files – a directory containing the generated files
- database_tagging – Python source code for Section 3, responsible for opening the tagging GUI.
- run_xxx (xxx for user name) – batch file for running the source code

To run the batch file, change the appropriate lines according to your system's setup (see instructions for Section 2).

See section 3 document (database tagging.docx) to learn how to use the tagging GUI.

The outputs of Section 3 are:

- ***<video_name>*_labels.xlsx** – An Excel file containing the labels given to each frame from each user. To be used in Section 4.
- ***<video_name>*_labels_graphs.png** – graphs containing the given labels for each frame and user.

8.7. Running Section 4 Source Code

Navigate to the project folder, then change directory to Files/Code/4_training_classifiers. This folder consists of the following:

- files – a directory containing the generated database
- generate_database – Python source code for Section 4, responsible for creating one database, merging the output files of Sections 2 and 3.
- run_xxx (xxx for user name) – batch file for running generate_database.py
- 4_training_classifiers.ipynb – Jupyter Notebook containing explanations and process of all the work done for the learning section

8.7.1. Running generate_database.py

To run the batch file, change the appropriate lines according to your system's setup (see instructions for Section 2). Also change the following parameters given in Table A-3:

Name	Description
--ear	Use the EAR feature. '0' to disable (default: 1)
--r2	Use the linear regression feature. '0' to disable (default: 0)
--ellipse	Use the ellipse area feature. '0' to disable (default: 1)
--poly	Calculate the Shoelace formula feature. '0' to disable (default: 1)
--exclude	Videos to be discarded from learning (default:[]). Usage: --exclude normal: don't take healthy patients into account --exclude normal a b palsy c: ignore normal_a, normal_b and palsy_c
--weights	The importance of the tags of each user (default: [0.5 0.5]). Usage: --weights w1 w2 … wN : for N users (default: N=2). All weights must sum into 1
--levels	The number of levels used for tagging (default: 5)
--graphs	Save graphs of features and labels vs frame time, for each video in the database (default: 1)

Table 8-3: Description of section 4's generate_database source code settings

The outputs of generate_database are:

- **database.xlsx** – An Excel file containing the normalized scores the features, extracted for each frame,and their respective label
- **database_reduced.xlsx** – An Excel file containing the full database after reducing of consecutive frames with identical labels

8.7.2. Running training_classifiers.ipynb

Note that in contrary to the source codes of the previous sections, which are written in a 'standard' Python format, this code is written using Jupyter notebook, allows for great data visualization, optimized for usage for data science and machine learning. To open and run this source code:

- Make sure that Jupyter Notebook is installed
- Run Anaconda prompt
- Activate opencv-env environment variable
- Navigate to Section 4 folder
- Type jupyter notebook.
- A web application containing the files in Section 4 directory should appear, containing the mentioned ipynb file
- Follow instructions and run each code section by pressing Shift + Enter

Note: This code is responsible for generating the classifier which we work with during Section 5 (called 'model.joblib').

8.8. Running Section 5 Source Code

Navigate to the project folder, then change directory to Files/Code/5_final_product. This folder consists of the following:

- files – a directory containing Section 5 output files
- final_product – Python source code for Section 5. Combining all the work done in the previous sections and runs in a dedicated GUI.
- run_xxx_gui (xxx for user name) – batch file for running final_product.py

To run Section 5 code, simply run the batch file and follow instructions on Section 5 documentation.