הפקולטה להנדסה
ע"ש אלכסנדר קופקין
אוניברסיטת בר־אילן

Anti-Sleeping Car Alarm Mobile App

Computer Engineering
Final Project

Tal Shafir
312499940
Niri Drory
314082843

Guide: Prof. Zvi Lotker

September 2021

Table of Contents

Table of Figures

Section 1: Intro

1.1. Project Background

Today, we see more and more safety and care technologies being implemented in motor vehicles and mobile transportation. Most of them are based on dangers from the outer area of the car, while ignoring the inside occurrences that could cause dangerous driving and car accidents. Driving under the influence of fatigue can happen to anyone and is a major cause of car accidents worldwide. According to a survey among nearly 150,000 adults in 19 states and the District of Columbia, an estimated 1 in 25 adult drivers (aged 18 or older) reported having fallen asleep while driving at least once in the previous 30 days. The National Highway Traffic Safety Administration estimates that drowsy driving was responsible for 72,000 crashes, 44,000 injuries, and 800 deaths in 2013 in the United States. However, these numbers are underestimated, and up to 6,000 fatal crashes each year may be caused by drowsy drivers in the US alone.

Nowadays there are multiple systems implemented in different kinds of cars that identify fatigue and warn the driver but these systems are very expensive and usually come exclusively as a package with the other systems of the car and mostly in luxury vehicles. For all these reasons there is a need for an inexpensive solution for real-time monitoring of the driver's alertness condition and not only monitoring of the road and the drivers performance i.e. staying in the lane, speeding, etc.

1.2. Introduction

In our project we developed a mobile app that will alarm the driver about signs of fatigue. The app will be based on the mobile front camera, while the mobile should be placed in the phone holder and capture the driver's face continuously throughout his driving. Using image processing and machine learning the system will learn the driver's facial expressions and will be able to determine and alarm the driver whenever he shows dangerous levels of fatigue.

During this project we handled many different issues, such as: choosing the most suitable technologies and algorithms, testing the app and tuning the model and algorithm to work optimally, real-time image processing and more.

1.3. Project Goals

Before we started working on the project we set goals for it to be Innovative and successful:

A. The product should be applicable to anyone. Therefore we choose to develop a mobile app that works on android and ios operating systems.
B. The product's code should be secure from hackers or viruses.
C. The product should work in real-time with no sufficient delays.
D. The product should have a high accuracy rate of detection to prevent real life-threatening danger.

1.4. Schedule and Work Flow

As a first step we wrote the project proposal that can be found in the attachments. We sent it to our professor, Zvi Lotker, for approval and scheduled a meeting. In the meeting we discussed the technologies we will be using and agreed that the product should be a mobile app. In addition we made a schedule and set deadlines for the different stages of the project:

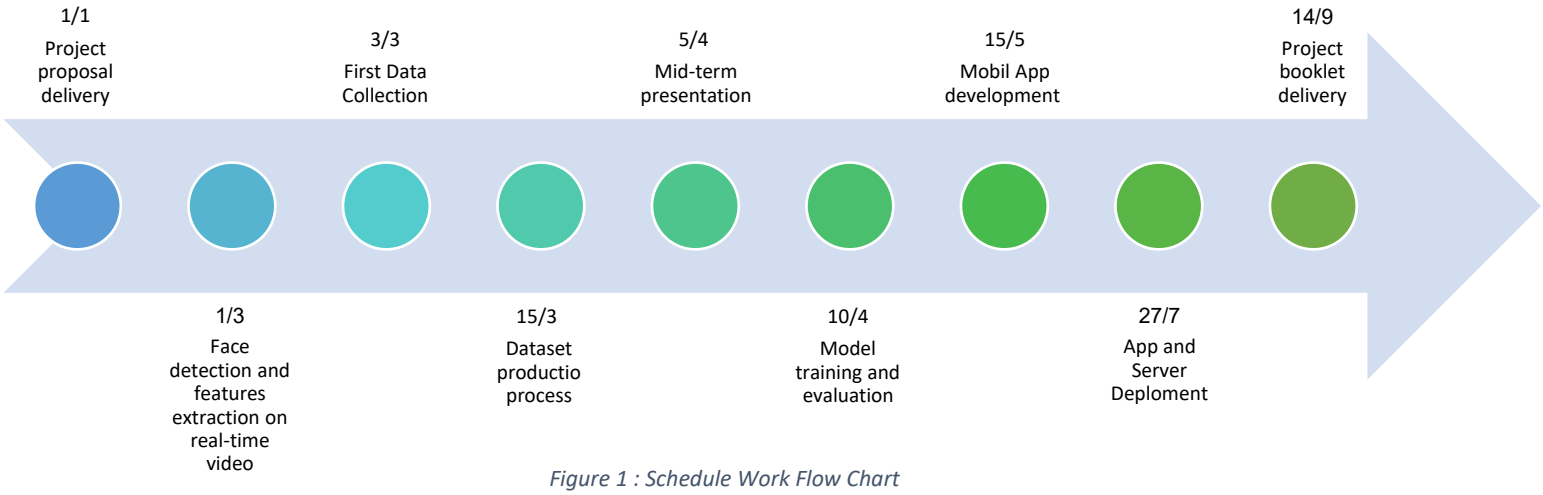


Figure 1 : Schedule Work Flow Chart

Section 2: Project Scope

2.1. Technologies

In our project we developed a cross platform mobile App using React Native UI open source framework that communicate using an API with a backend model located on a global server.

2.1.1. React Native

React Native is an open-source UI software framework created by Facebook, Inc. It is used to develop applications for Android, Android TV, iOS, macOS, tvOS, Web, Windows and UWP by enabling developers to use the React framework along with native platform capabilities. We chose this framework over native app languages like Kotlin and Swift because it is more code efficient, 'write once run anywhere' coding and mainly because our app almost doesn't require using native components.

2.1.2. FastAPI

FastAPI is a modern, high-performance, web framework for building APIs with Python. It is fast to code, intuitive and standards-based.

2.1.3. Git

Git is software for tracking changes in any set of files, usually used for coordinating work among programmers collaboratively developing source code during software development.

GitHub- GitHub, Inc. is a provider of Internet hosting for software development and version control using Git.

2.1.4. Heroku

Heroku is a cloud platform as a service (PaaS) supporting several programming languages. It supports Java, Node.js, Scala, Clojure, Python, PHP, and Go. The Heroku network runs the customer's apps in virtual containers which execute on a reliable runtime environment. Custom buildpacks with which the developer can deploy apps in any other language.

2.2. Architecture and Flow chart

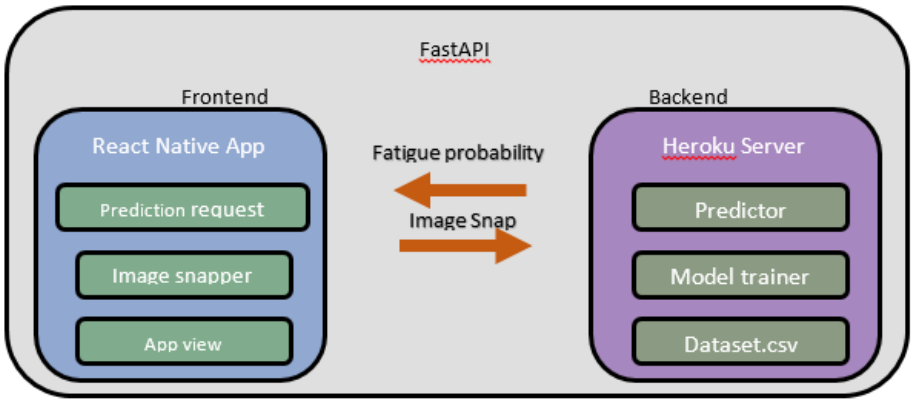


Figure 2: Project architecture diagram

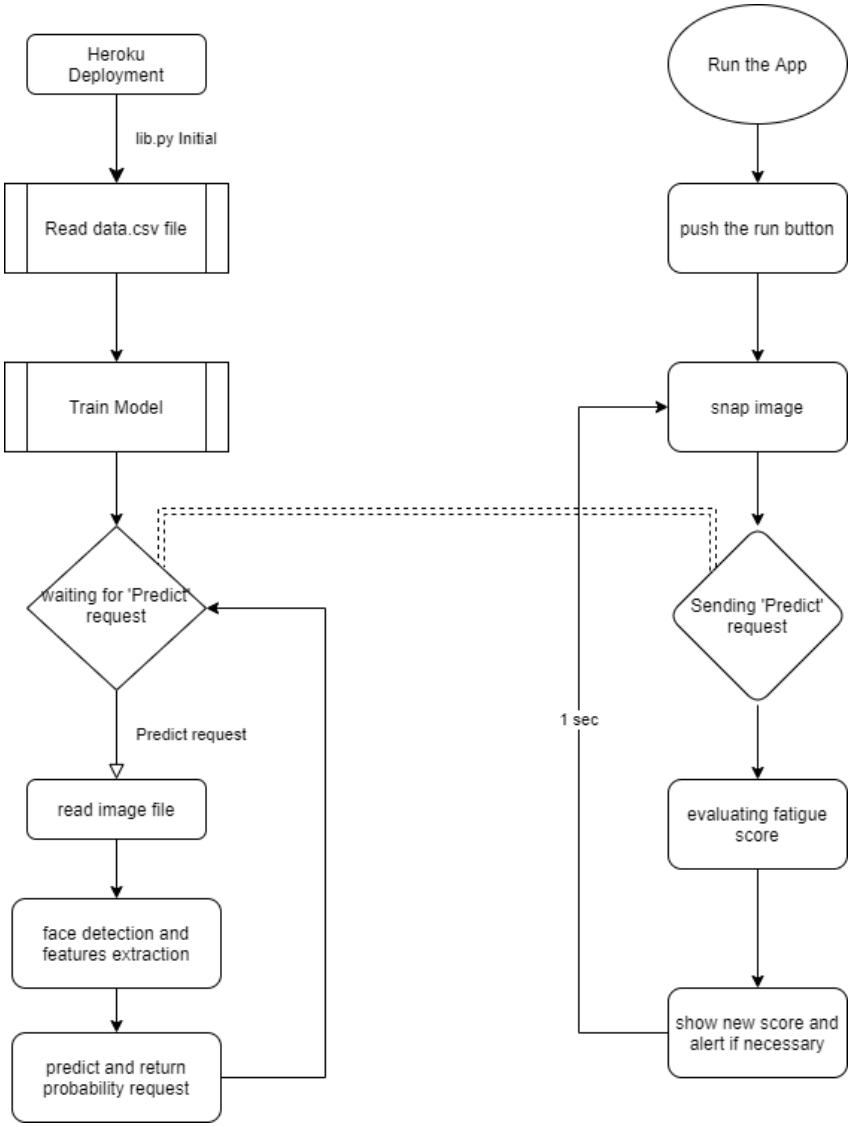


Figure 3: Project flow chart

2.3. Face recognition and Features extraction

After gathering the videos that we will use to generate the training and test datasets, for each video, we used OpenCV to extract 2 frames per second starting at the beginning until the end of the video. We used **mlxtend.image**'s Facial landmark detection function to catch the user's face, there were a total of 68 landmarks per frame but we decided to keep the landmarks for the eyes, nose and mouth only (Points 37–68). These were the important data points we used to extract the features for our model.

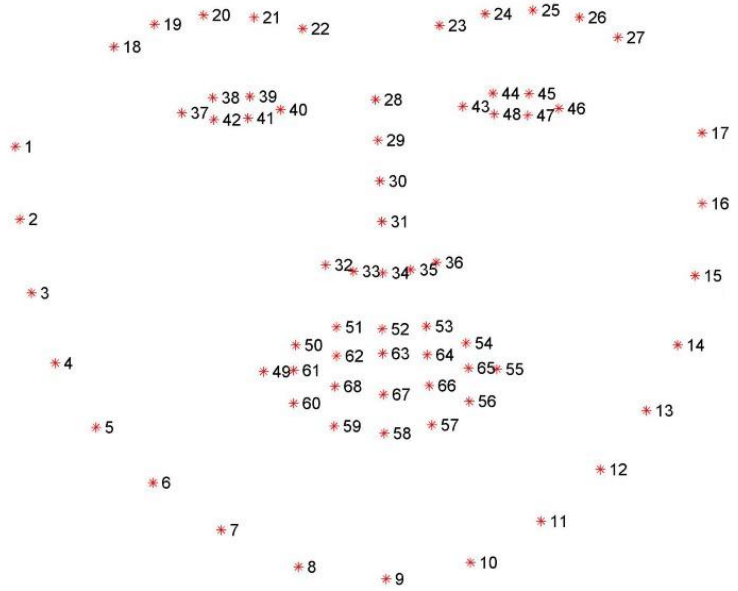


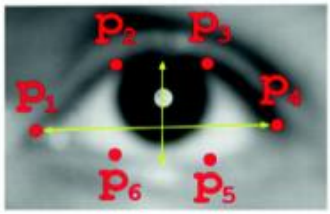
Figure 4: Facial landmarks

Feature Extraction

Based on the facial landmarks that we extracted from the frames of the videos, we ventured into developing suitable features for our classification model. While we hypothesized and tested several features, the five core features that we concluded on for our final models were **eye aspect ratio**, **mouth aspect ratio**, **pupil circularity**, **nose-mouth distance** and finally, **mouth aspect ratio over eye aspect ratio**.

2.3.1. Eye Aspect Ratio (EAR)

EAR, as the name suggests, is the ratio of the length of the eyes to the width of the eyes. The length of the eyes is calculated by averaging over two distinct vertical lines across the eyes as illustrated in the figure below.



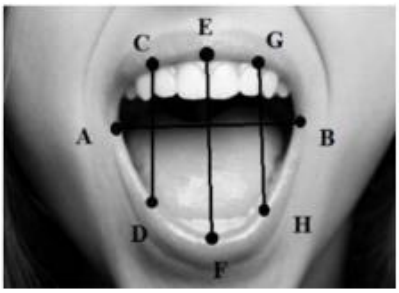
$$\text{EAR} = \frac{\|p_2 - p_6\| + \|p_3 - p_5\|}{2\|p_1 - p_4\|}$$

Figure 5: Eyes Aspect Ratio

Our hypothesis was that when an individual is drowsy, their eyes are likely to get smaller and they are likely to blink more. Based on this hypothesis, we expected our model to predict the class as drowsy if the eye aspect ratio for an individual over successive frames started to decline i.e. their eyes started to be more closed or they were blinking faster.

2.3.2. Mouth Aspect Ratio (MAR)

Computationally similar to the EAR, the MAR, as one would expect, measures the ratio of the length of the mouth to the width of the mouth. Our hypothesis was that as an individual becomes drowsy, they are likely to yawn, making their MAR higher than usual in this state.



$$\text{MAR} = \frac{|\text{EF}|}{|\text{AB}|}$$

Figure 6: Mouth Aspect Ratio

2.3.3. Pupil Circularity (PUC)

PUC is a measure complementary to EAR, but it places a greater emphasis on the pupil instead of the entire eye.

$$Circularity = \frac{4 * \pi * Area}{perimeter^2} \qquad Area = \left(\frac{\text{Distance(p2, p5)}}{2}\right)^2 * \pi$$

$$Perimeter = Distance(p1, p2) + Distance(p2, p3) + Distance(p3, p4) + Distance(p4, p5) + Distance(p5, p6) + Distance(p6, p1)$$

Figure 7: Pupil Circularity

For example, someone who has their eyes half-open or almost closed will have a much lower pupil circularity value compared to someone who has their eyes fully open due to the squared term in the denominator. Similar to the EAR, the expectation was that when an individual is drowsy, their pupil circularity is likely to decline.

2.3.4. Nose-Mouth distance (NMD)

NMD is an hypothesis based on the fact that when people are drowsy while sitting their head tilts down and therefore the distance between the nose and the mouth (front looking) gets smaller. This indication can give us a better perspective of the driver's fatigue levels.

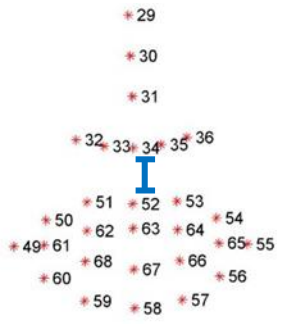


Figure 8: Nose-Mouth Distance

2.3.5. Mouth aspect ratio over Eye aspect ratio (MOE)

Finally, we decided to add MOE as another feature. MOE is simply the ratio of the MAR to the EAR.

$$MOE = \frac{MAR}{EAR}$$

Figure 9: Mouth Over Eyes Aspect Ratio

The benefit of using this feature is that EAR and MAR are expected to move in opposite directions if the state of the individual changes. As opposed to both EAR and MAR, MOE as a measure will be more responsive to these changes as it will capture the subtle changes in both EAR and MAR and will exaggerate the changes as the denominator and numerator move in opposite directions. Because the MOE takes MAR as the numerator and EAR as the denominator, our theory was that as the individual gets drowsy, the MOE will increase.

2.4. Fatigue levels and symptoms

While choosing the features we want to extract in real time we based our selection on common fatigue symptoms known to all:

2.4.1. Eyes blinking levels

When a person reaches certain levels of fatigue his blinking becomes faster and the time passing between blinks becomes shorter, therefore the duration of his **EAR** becoming close is an important detail.

2.4.2. Eyes drop

When a person reaches certain levels of fatigue his eyes can drop slowly and in more frequent occurrences, therefore the frequency of his **EAR** becoming close and his **PUC** becoming more of an elliptical shape is an important detail.

2.4.3. Yawing

Yawing is one of the most obvious symptoms of fatigue, therefore the duration of ones **MAR** becoming wide and open is an important detail.

2.4.4. Eye-Moth combination

Because those symptoms mentioned above are very obvious and important while detecting fatigue levels we focus on finding combinational occurrences of **EAR** and **MAR** together using **MOE**.

2.4.5. Head tilt

When a person reaches certain levels of fatigue his head can tilt forwards or backwards, therefore the distance between his nose and mouth is becomes closer or farther. Using **NMD** is an important detail to indicate that.

2.5. Dataset

The dataset we built for the machine learning model to learn from is attached ("merge.csv"). The process of building the dataset is described in the next section. It consists of 36222 examples of analyzed frames taken from 79 videos.

Video	Frame	EAR	MAR	CIR	ME	HEAD	EAR_N	MAR_N	CIR_N	ME_N	HEAD_N	Label
1	0	0.254456	0.753925	0.410524	2.962887	89.35177	0.139509	1.413868	0.037417	0.972365	0.948243	0
1	1005	0.234888	0.898606	0.361181	3.825676	87.03025	1.149016	0.734003	1.243025	1.375497	0.434518	0
1	1020	0.282137	0.894959	0.455542	3.172068	78.81796	1.288526	0.679865	1.205608	0.403132	1.382761	0
1	1035	0.271223	0.8944	0.412618	3.297657	75.96568	0.725456	0.671566	0.091741	0.061373	2.013936	0
1	105	0.242016	0.689194	0.382492	2.847719	107.8051	0.781279	2.374842	0.690008	1.285765	5.031727	0

Figure 10: data.csv file

There are 3 sources from which we collected the data which we later analyzed:

2.5.1. YouTube videos

We found 2 YouTube playlists of people falling asleep. Many of the videos were not relevant for this project because the quality of the video was not sufficient or because the people's faces were not recognizable enough because of inappropriate angles etc. 25 of the videos used for the dataset are from these YouTube playlists.

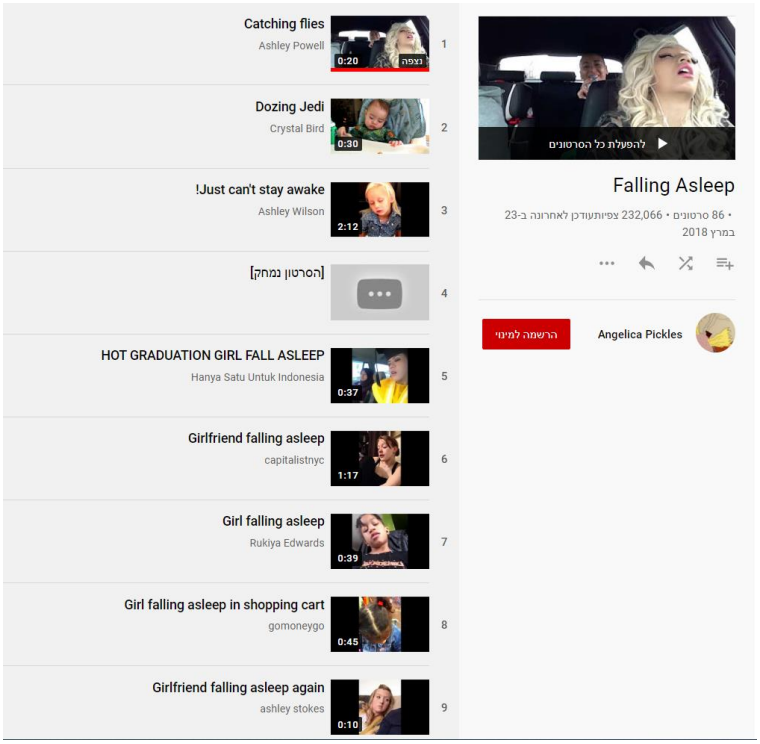


Figure 11: YouTube falling asleep playlist

2.5.2. DROZY Database

The ULg Multi-modality Drowsiness Database (DROZY) provides multi-modal data of 14 healthy people (3 males and 11 females aged 22.7 ± 2.3, without any alcohol dependency, drug addition or sleep disorder) who were asked to perform three 10-minutes psycho-motor vigilance tests (PVT) under conditions of increasing sleep deprivation induced by prolonged wake: first day at 10:00 AM (normal sleep patterns), second day at 3:30 AM (20 hours of sleep deprivation) and second day at 11:00 AM (28 hours of sleep deprivation). The Subjects were instructed to monitor a red rectangular box over a black background on a computer screen, and to press a response button as soon as they noticed the appearance of a yellow stimulus counter within the box. Videos are recorded with an artificial near-infrared illumination using a Microsoft Kinect.

Figure 12: DROZY database

2.5.3. Self-captured videos

Due to lack of data sources we recorded ourselves in a clean environment and performed some staged tiredness symptoms to enlarge our dataset.

Figure 13: Self-captured videos

Section 3: Development

3.1. Face recognition

For the facial recognition part of the project we used several python libraries:

- **OpenCV** (Open Source Computer Vision Library) is a library of programming functions mainly aimed at real-time computer vision. We used several of OpenCV's functions:
    - *VideoCapture*- Creates a video stream class which can read video input frame by frame.
    - *cvtColor* and COLOR_BGR2GRAY- These functions alter the color of the image. We used them to convert the images to grayscale because the face detector works better with grayscale images.
    - *Imshow*- Displays single frame on the screen. We used this function at the end of the while loop to display frame by frame in a video sequence.
    - *waitKey(1)*- Displays each frame for 1ms.
    - *destroyAllWindows*- At the end of the program closes all open output windows.

- **Dlib** is a toolkit for making real world machine learning and data analysis applications in C++. While the library is originally written in C++, it has good, easy to use Python bindings. It is also extensively used for face detection and facial landmark detection. We used 2 functions from dlib:
    - *get_frontal_face_detector*- Creates a class which can receive an image and return an object of all the faces found in the image.
    - *shape_predictor*('shape_predictor_68_face_landmarks.dat')- Creates a class which can receive an image and faces object and return the coordinates of the 68 points of each face in the image as explained above. The parameter we passed to the function is a pre-trained model for facial landmarks detection.

The Facial recognition process:

As seen in the attached code file "Project.py", we begin with createing video stream, predictor and detector objects. Next we start a video stream by opening an infinite while loop and reading a single frame each iteration. Each frame gets resized and converted to grayscale and gets sent to the detector to detect faces in the image. The webcam of the computer which we ran this simulation on works at approximately 20 fps, therefore we analyzed every 20th frame to receive about 1 frame per second to simulate the performance of the future mobile app. For each of the frames analyzed we use the predictor to get the 68 facial points needed for analysis of the face.

3.2. Features extraction

For the features extraction part of the project we used the following python libraries:

Scipy- SciPy is a free and open-source Python library used for scientific computing and technical computing. We used the distance.euclidean function from the scipy.spatial package. This function calculates the Euclidean distance between 2 coordinates passed to it as parameters.

Imutils- Imutils are a series of convenience functions to make basic image processing functions such as translation, rotation, resizing, skeletonization, and displaying Matplotlib images easier with OpenCV. We used face_utils.shape_to_np function which converts the coordinates detected as a face to an array and resize function which resizes the image given.

Math- Math is module with a set of built-in math functions that allow us to perform mathematical tasks on numbers. We used the math.pi constant as the constant π, in order to calculate the circularity of the eye as explained above.

After the facial recognition, explained in the previous sub-section, we extracted the features needed for the prediction of the machine learning models. Each frame analyzed sends the relevant coordinates of the face to 5 functions: *eye_aspect_ratio*, *mouth_aspect_ratio*, *circularity*, *mouth_over_eye* and *head_drop*. These functions compute the features that are sent to the machine learning classifier. The computations for each feature are specified in the previous section.

3.3. Dataset creation and expansion processes

For the Dataset creation and expansion part of the project we used the following python libraries:

OpenCV- For this part we used the functions stated above in addition to a few more:

- CAP_PROP_FPS- returns the fps rate of the given video.
- CAP_PROP_FRAME_COUNT-returns the number of frames in the given video.
- Imwrite- saves image to specified location.
- Imread- reads image from specified location.
- Resize- Resizes image to specified size.

OS- The OS module in Python provides functions for interacting with the operating system. OS comes under Python's standard utility modules. This module provides a portable way of using operating system-dependent functionality. We used the listdir

function to iterate through an entire directory and the path function to read videos from a specific directory.

CSV- The csv module implements classes to read and write tabular data in CSV format.we used the reader and writer classes of this library in order to read our dataset and to write to it in order to expand it.

Math- As explained in the previous sub-section.

Scipy- As explained in the previous sub-section.

Mlxtend- Mlxtend (machine learning extensions) is a Python library of useful tools for the day-to-day data science tasks. We used the extract_face_landmarks function from mlxtend.image which gives the coordinates on the image of the 68 facial points discussed earlier.

Numpy- Numpy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. We used a few of numpy's basic functions, such as:

- *Array*- converts list to array.
- *Mean*- computes the average value of a numerical array.
- *Std*- computes the standard deviation of a numerical array.

The creation and expansion of the dataset can be seen in the attached code files "ml.py", "label_image.py" and "merge_example_label.py". This process is completed in 3 steps:

1. **Manually labeling** frames from the videos in “label_image.py”:
   We iterate through the attached “Videos” folder and check for each video if it has been labeled already by checking if the number of the video exists in the attached “training.csv” dataset. If we have never labeled the video before, we print the video’s fps and number of frames and a window opens in which we manually choose 0 to determine the person is alert or 1 to determine he is drowsy for each frame presented. Since the videos have an average of approximately 30 fps, we didn’t label every single frame, but we labeled 2 frames for every second in the video and saved the frame to the attached “Frames” folder.

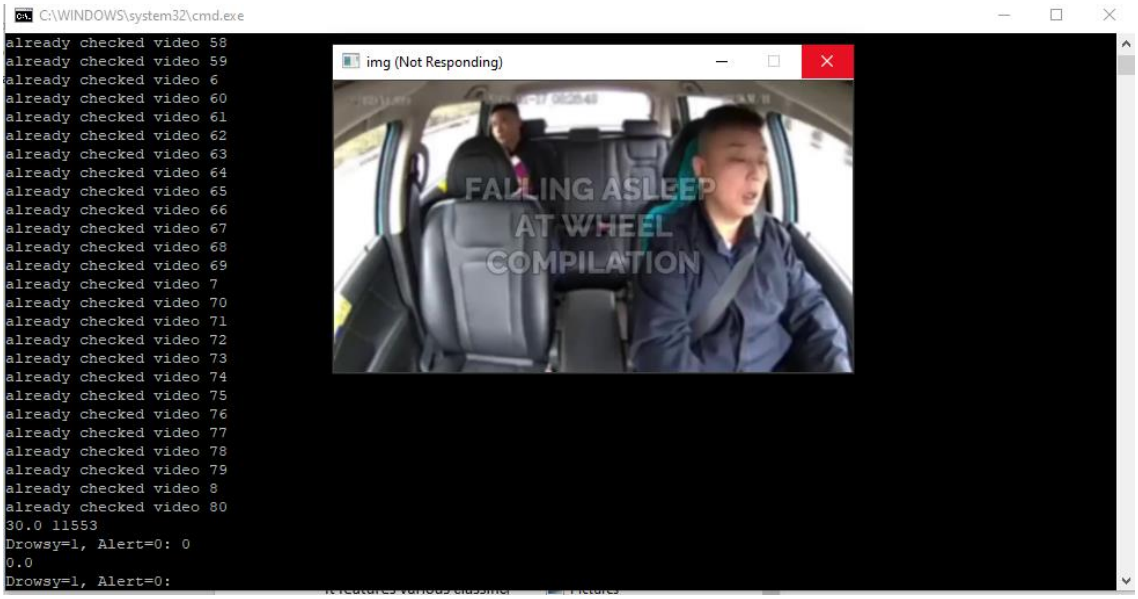


Figure 14: Manually labeling process

Next we save the images and their labels in the attached “training.csv” file in the following format:

Video	Frame	Label
1	0	0
1	15	0
1	30	0
1	45	0
1	60	0
1	75	0
1	90	1
1	105	0
1	120	0

Figure 15: Labled table

.

2. **Features Extracting** from the saved frames in “ml.py”:
   We iterate through the “Frames” folder and check for each frame if it has been analyzed before. If it wasn’t, it’s features are extracted and they are added to a list. The list is sent to the “normalize” function where we first extract the features from the 1st 3 frames of each video and compute the mean value and the standard deviation for each feature. Then we normalize every new frame received by subtracting the mean of the feature from the feature’s value and dividing by the standard deviation. We save the 5 features and 5 normalized features from every frame and save them in the attached “features.csv” file in the following format:

1	0	0.254456	0.753925	0.410524	2.962887	89.35177	0.139509	1.413868	0.037417	0.972365	0.948243
1	1005	0.234888	0.898606	0.361181	3.825676	87.03025	1.149016	0.734003	1.243025	1.375497	0.434518
1	1020	0.282137	0.894959	0.455542	3.172068	78.81796	1.288526	0.679865	1.205608	0.403132	1.382761
1	1035	0.271223	0.8944	0.412618	3.297657	75.96568	0.725456	0.671566	0.091741	0.061373	2.013936
1	105	0.242016	0.689194	0.382492	2.847719	107.8051	0.781279	2.374842	0.690008	1.285765	5.031727
1	1050	0.332021	0.888514	0.497415	2.676078	103.7373	3.86196	0.584181	2.2922	1.752842	4.131575
1	1065	0.334751	0.661725	0.530349	1.976769	67.17263	4.00279	2.782645	3.146835	3.655833	3.959729
1	1080	0.2728	0.396005	0.390971	1.451631	63.90189	0.806817	6.727426	0.469986	5.084862	4.683503
1	1095	0.258226	0.341877	0.376007	1.323941	86.57132	0.054977	7.530996	0.858304	5.432336	0.332962
1	1110	0.245667	0.334937	0.372966	1.363377	80.58383	0.592942	7.634019	0.937215	5.325021	0.991994

Figure 16: Features extraction table

Where the columns are: video number, frame number, 5 features, 5 normalized features.

3. Merging the features and labeling into 1 file in “merge_example_label.py”:
   We iterate through “features.csv” and “training.csv” and find rows where their video and frame columns match. We write the data of the matching rows to the new “merge.csv” file in the following format:

Video	Frame	EAR	MAR	CIR	ME	HEAD	EAR_N	MAR_N	CIR_N	ME_N	HEAD_N	Label
1	0	0.254456	0.753925	0.410524	2.962887	89.35177	0.139509	1.413868	0.037417	0.972365	0.948243	0
1	1005	0.234888	0.898606	0.361181	3.825676	87.03025	1.149016	0.734003	1.243025	1.375497	0.434518	0
1	1020	0.282137	0.894959	0.455542	3.172068	78.81796	1.288526	0.679865	1.205608	0.403132	1.382761	0
1	1035	0.271223	0.8944	0.412618	3.297657	75.96568	0.725456	0.671566	0.091741	0.061373	2.013936	0
1	105	0.242016	0.689194	0.382492	2.847719	107.8051	0.781279	2.374842	0.690008	1.285765	5.031727	0

Figure 17: Merged table

Now we saved the new file. “merge.csv” is our final dataset for the machine learning models to train and test on.

3.4. Models evaluation and selection

For the model evaluation and selection part of the project we used the following python libraries:

Pandas- pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series. We used a few of Pandas' basic functions, such as read_csv and DataFrame to read from the dataset in order to train the machine learning models we used and to display the data conveniently in the form of a table.

Sklearn- Sklearn is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy. We used many of sklearn's functions, mostly classifiers.

- From the ensemble methods we used RandomForestClassifier, BaggingClassifier, ExtraTreesClassifier, AdaBoostClassifier and GradientBoostingClassifier.
- From the neighbors methods we used KNeighborsClassifier.
- From the tree methods we used DecisionTreeClassifier.
- From the linear_model methods we used SGDClassifier and LogisticRegression.
- These are all classifiers, which we used to train the machine learning models.
- Additional sklearn functions used:
- From model_selection methods we used train_test_split to split the dataset to 80% training data and 20% test data for model evaluation.
- From metrics methods we used confusion_matrix to compute the confusion matrix for each model and accuracy_score to compute the accuracy of each model.

After completing our dataset, the next task was to choose the best machine learning model for classification. First we read the “merged.csv” file in the “project.py” file and split the data to 80% training data and 20% test data. Then, for each of the 9 models stated above, we trained the model on the 80% training data, made predictions on the 20% test data and computed the confusion matrix and accuracy of the predictions. The results were as follows:

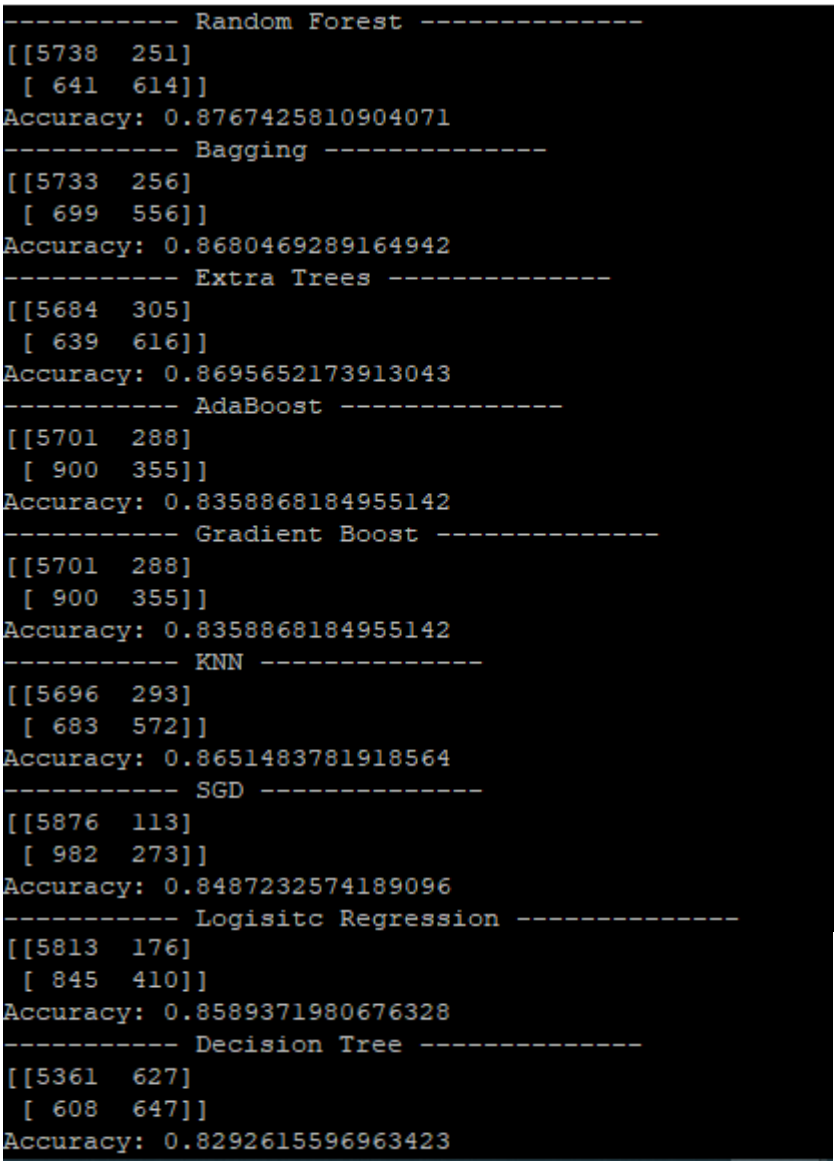


Figure 18: Models evaluation matrixes with normalized features

We repeated the process but with a dataset that didn't include the normalized features, meaning we evaluated the classifiers using only the non-normalized features. The results were as follows:

```
---------- Random Forest --------------
[[5701  288]
 [ 758  497]]
Accuracy: 0.8554865424430642
---------- Bagging --------------
[[5668  321]
 [ 781  474]]
Accuracy: 0.8477570738440303
---------- Extra Trees --------------
[[5638  351]
 [ 743  512]]
Accuracy: 0.8488612836438924
---------- AdaBoost --------------
[[3780 2209]
 [ 329  926]]
Accuracy: 0.6495514147688061
---------- Gradient Boost --------------
[[3780 2209]
 [ 329  926]]
Accuracy: 0.6495514147688061
---------- KNN --------------
[[5639  350]
 [ 781  474]]
Accuracy: 0.8437543133195307
---------- SGD --------------
[[5701  288]
 [ 808  447]]
Accuracy: 0.8485852311939268
---------- Logisitc Regression --------------
[[5828  161]
 [ 897  358]]
Accuracy: 0.8538302277432712
---------- Decision Tree --------------
[[5234  754]
 [ 712  543]]
Accuracy: 0.7973775017253278
```

Figure 19: Models evaluation matrixes without normalized features

The accuracy of the Random Forest classifier (85.5%) is slightly better than the accuracy of the Logistic Regression classifier (85.3%) but since the Random Forest classifier has a higher True negative value, this is the classifier we chose for the project. For this kind of project a high True negative is more important than a high True positive because it is more important to correctly detect that the driver is drowsy to prevent an accident and not as important to correctly detect that the driver is alert and risk hearing the alarm when the driver is completely alert.

In addition, we checked the importance of each feature to the model's classification and got the following results:

```
----- Feature Importance -----
[0.2913218199816846, 0.01998878551657718, 0.08728829316922482, 0.3357547786246195, 0.09636435127196798, 0.02878063755795
4115, 0.004080639395507619, 0.024383772411168993, 0.04172584711308871, 0.0703110749582065]
5 Feature's importance: 0.8307180285640741
5 Normalized feature's importance: 0.16928197143592594
```

Figure 20: Featues Importence calculation

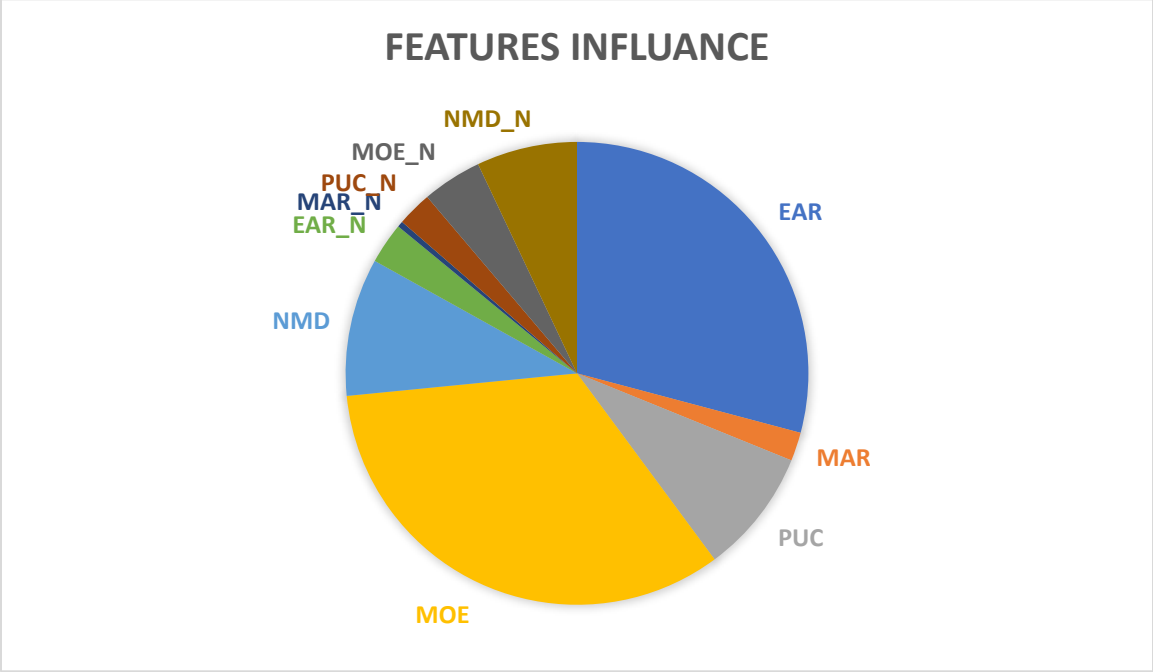


Figure 21: Featues Importence Chart

It is clear that the normalized features contribute a lot less to the classification process than the features themselves. Moreover, the EAR and ME features are significantly more important for the model prediction.

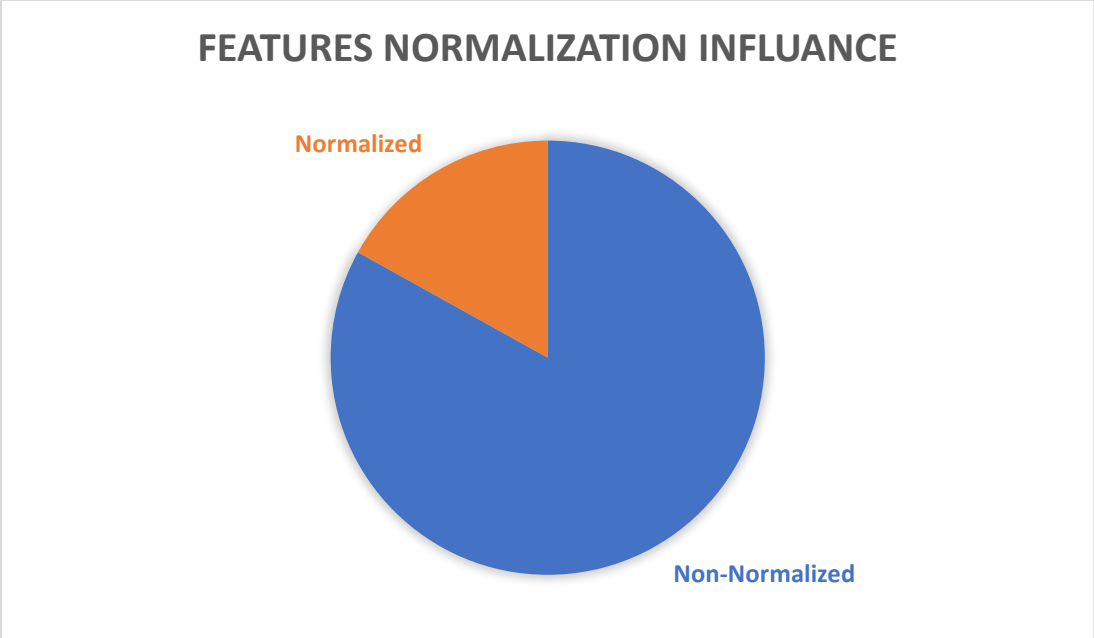


Figure 22: Normalized and non-normalized features comparison

For these reasons we chose the Random Forest classifier for the project and eventually we chose to use the dataset which didn't include the normalized features. Although adding the normalization increased accuracy by about 2% but later, when we ran the code on the mobile app and used a sever to send images and receive predictions we experienced long delays receiving the normalized features. This delay is unacceptable in a real-time app, so taking in to consideration that the gain in accuracy and feature importance were not very big we decided to drop the normalization.

3.5. Model optimization

For the model optimization part of the project we used the **sklearn** python library. The functions we used are:

- *make_scorer* to adjust the scoring of GridSearchCV to evaluate the hyperparameters based on the accuracy they give the model.
- *GridSearchCV* to perform hyperparameters tuning on the selected model to optimize the prediction.

As stated above the classifier we used for this project is **Random Forest** classifier. Random forests are an ensemble learning method for classification tasks that operates by constructing a multitude of decision trees at training time. For classification tasks, the output of the random forest is the class selected by most trees. In order to optimize the accuracy of this classifier we used GridSearchCV to find the set of hyperparameters which gives the best accuracy for the model. The scoring function notifies GridSearchCV to look for the highest accuracy by returning the accuracy of the classifier given different permutations of the hyperparameters.

The hyperparameters we chose to tune are:

- ***N_estimators***- The number of trees in the forest. The range checked was 50-200.
- Criterion- The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain. The functions checked were “gini” and “entropy”.
- ***Max_depth***- The maximum depth of the tree. The range checked was 5-20.
- ***Min_samples_split***- The minimum number of samples required to split an internal node. The range checked was 10-20.
- ***Max_features***- The number of features to consider when looking for the best split.
    - if “auto”, then max_features=sqrt(n_features).
    - if “sqrt”, then max_features=sqrt(n_features) (same as “auto”).
    - If “log2”, then max_features=log2(n_features). The options checked were “sqrt” and “log2” because “auto” is the same as “sqrt”, therefore there is no reason to check them as 2 separate options.
- ***Ccp_alpha***- Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ccp_alpha will be chosen. By default, no pruning is performed. The range checked was 0-1.
- ***Max_samples***- The number of samples to draw from X to train each base estimator.
    - If int, then draw max_samples samples.
    - If float, then draw max_samples * X.shape[0] samples. Thus, max_samples should be in the interval (0, 1). The range checked was 0.5-10.

We ran ***GridSearchCV*** many times with *CV=5* (using cross validation with 5 folds), slightly changing the tested hyperparameters according to the results. In the following images we can see the results of a few of the iterations:

```
[{'best_score': 0.8205984712961456, 'best_params': {'ccp_alpha': 0.001, 'criterion': 'entropy', 'max_depth': 10, 'max_fe
atures': 'log2', 'max_samples': 0.5, 'min_samples_split': 12, 'n_estimators': 100}, 'best_grid': RandomForestClassifier(
ccp_alpha=0.001, criterion='entropy', max_depth=10,
                       max_features='log2', max_samples=0.5,
                       min_samples_split=12)}]
   model  ...
                                  best_params
0 NaN     ...  {'ccp_alpha': 0.001, 'criterion': 'entropy', 'max_depth': 10, 'max_features': 'log2', 'max_samples': 0.5,
 'min_samples_split': 12, 'n_estimators': 100}
```

```
[{'best_score': 0.8210559982653006, 'best_params': {'ccp_alpha': 0.001, 'criterion': 'entropy', 'max_depth': 16, 'max_fe
atures': 'log2', 'max_samples': 0.8}, 'best_grid': RandomForestClassifier(ccp_alpha=0.001, criterion='entropy', max_dept
h=16,
                       max_features='log2', max_samples=0.8)}]
   model  ...                                                                                                  best_params

0 NaN     ...  {'ccp_alpha': 0.001, 'criterion': 'entropy', 'max_depth': 16, 'max_features': 'log2', 'max_samples': 0.8}
```

```
[{'best_score': 0.8201116712744619, 'best_params': {'ccp_alpha': 0.003, 'criterion': 'entropy', 'max_depth': 15, 'max_fe
atures': 'sqrt', 'max_samples': 0.7, 'n_estimators': 100}, 'best_grid': RandomForestClassifier(ccp_alpha=0.003, criterio
n='entropy', max_depth=15,
                       max_features='sqrt', max_samples=0.7)}]
   model  ...
         best_params
0 NaN     ...  {'ccp_alpha': 0.003, 'criterion': 'entropy', 'max_depth': 15, 'max_features': 'sqrt', 'max_samples': 0.7,
 'n_estimators': 100}
```

```
[{'best_score': 0.8676190412582458, 'best_params': {'ccp_alpha': 0, 'criterion': 'entropy', 'max_depth': 16, 'max_featur
es': 'log2', 'max_samples': 0.5, 'min_samples_split': 20, 'n_estimators': 100}, 'best_grid': RandomForestClassifier(ccp_
alpha=0, criterion='entropy', max_depth=16,
                       max_features='log2', max_samples=0.5,
                       min_samples_split=20)}]
   model  ...
                              best_params
0 NaN     ...  {'ccp_alpha': 0, 'criterion': 'entropy', 'max_depth': 16, 'max_features': 'log2', 'max_samples': 0.5, 'mi
n_samples_split': 20, 'n_estimators': 100}
```

Figure 23: GridSearchCV tests

After receiving the same hyperparameters on 3 consecutive iterations we realized that we have found the optimal hyperparameters and those are the ones we used in the final model. The hyperparameters are:

ccp_alpha=0, criterion='entropy', max_depth=16, max_features='log2', max_samples=0.5, n_estimators=100, min_samples_split=20.

Using these hyperparameters increased the model's accuracy to **86.3%.**

3.6. Test Fatigue Score Thresholds

After choosing the optimal machine learning model we needed to test it on a real-time video stream.

In the “project.py” file we ran an infinite while loop which analyzes approximately 1 frame per second, as explained in section 3.1. to predict the class of each of the analyzed frames we used the predict_proba function which returns the probability of the frame to be classified to each class (alert or drowsy). We summed the probabilities to be drowsy of the last 5 frames and computed our “drowsiness score” based on this sum. The score changes overtime according to the prediction of every new frame received. After many rounds of trial and error we chose the appropriate thresholds.

The final thresholds we settled on are:

Score > 0.8: Sleeping- turn on alarm sound.
0.8 > Score > 0.5: Drowsy- Display visual warning.
Score < 0.5: Alert.

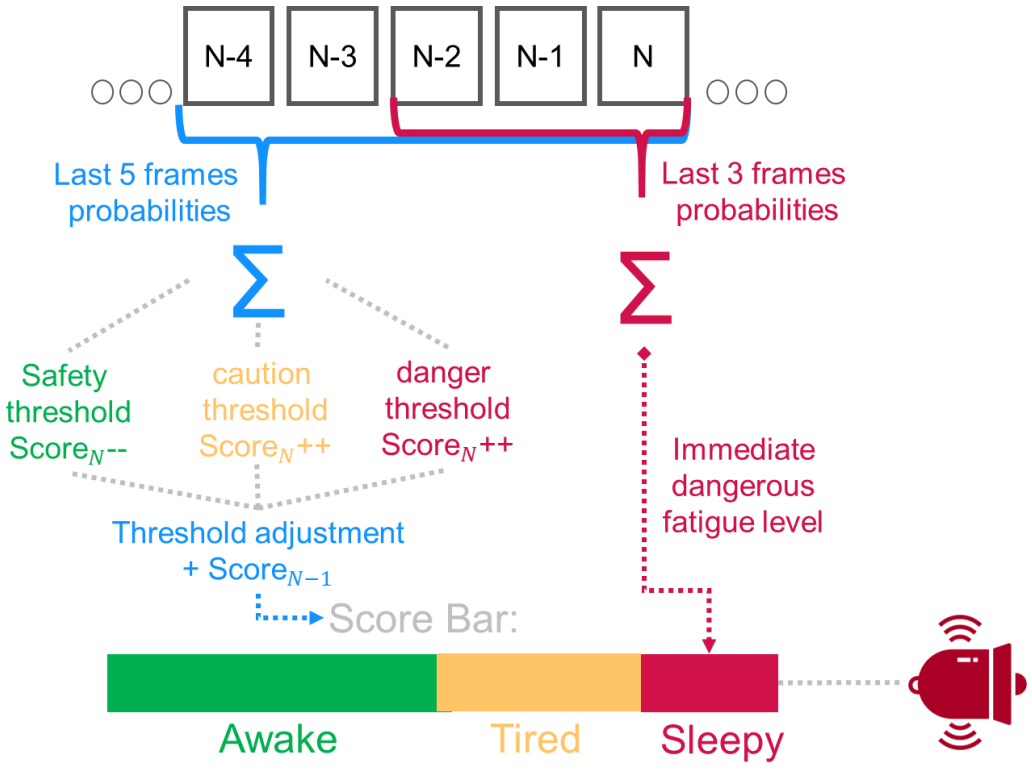


Figure 24: Score updating chart

3.7. API and Backend Server

In our project we needed a backend server to receive an image from the React Native app and return the probability for the driver in the image to be drowsy. In order to implement the 2 directional data sending between the Heroku back-end server and the React Native front-end app we used FastAPI. In the “index.js” file, in the “submitImage” function, in the “Camera” folder we use the fetch(https://safeapp-server.herokuapp.com/predict) function which sends the image to the backend server via the API. In the same function we use response.json function to get the probability prediction from the backend server. In order to implement the backend part of the project we used 3 files: “main.py”, “lib.py” and “merge.csv”. for the implementation of the prediction for the image we altered the python code written for the simulation. We created a class “Lib” in the “lib.py” file which includes all the feature extraction methods, the face predictor and detector and the machine learning model which receives the “merge.csv” dataset to train on. The extraction of the 5 facial features and prediction is the same as in the simulation. In “main.py” we implement the communication to the backend server via the API. We created an instance of the FastAPI framework and an instance of the Lib class. We then define the @app.post function. The predict function receives a frame, encodes it to the appropriate format and returns a prediction which is written to the Heroku backend server under /predict. This way every time the predict function is called from the frontend app, the API writes the prediction to the backend server and the app fetches the result from there. this is how the API handles the communication between backend and frontend parts of the app.

3.8. Mobile App

After testing the model and setting the proper thresholds for the warning signals it was time to develop a mobile app to make an accessible product for clients to use on their smartphones everywhere.

We developed the app using React Native for the reasons described above.

Every React Native project starts with the “App.js” file. React native apps work with multiple screen components and a navigation model to navigate between them. In our project we only needed to use 1 screen with multiple different view components. The view component that is rendered at every time is based on the current status of the app. The status of the app is determined by the user pressing the start/stop button and by the drowsiness score in real-time.

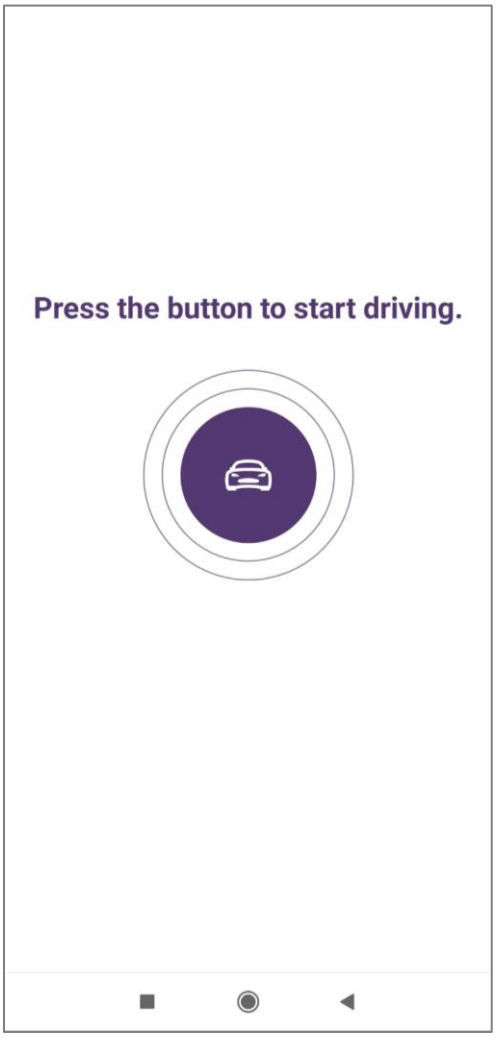


Figure 25: App start screen

“App.js” is the file from which components are rendered to the app. The component we render is the “Scene” component which we can see in the “Scene.js” file. This component returns several things to be rendered:

1. The button which starts and stops the video analysis. The user must click the button to start the app’s functions.
2. The CameraView component which we can see in the “index.js” file in the “Camera” folder or CameraIdle component which we can see in “App.js” according to the app’s status.
3. The Score value which changes color according to the driver’s state (alert, drowsy or sleeping)
4. The warning icon and sound when the score rises above the determined threshold.

The returned components are rendered conditionally. The components are rendered According to the drowsiness score which is computed as described in section 3.6, i.e. if the score is above the sleeping threshold the component rendered is the “You Are Sleeping” text which is colored red, if it is above the drowsy threshold “You are Drowsy” is rendered in orange and if it is under the alert threshold “You Are Alert” is rendered in green. Moreover, if the user pressed the start button then the CameraView component is rendered, otherwise the CameraIdle component is rendered.

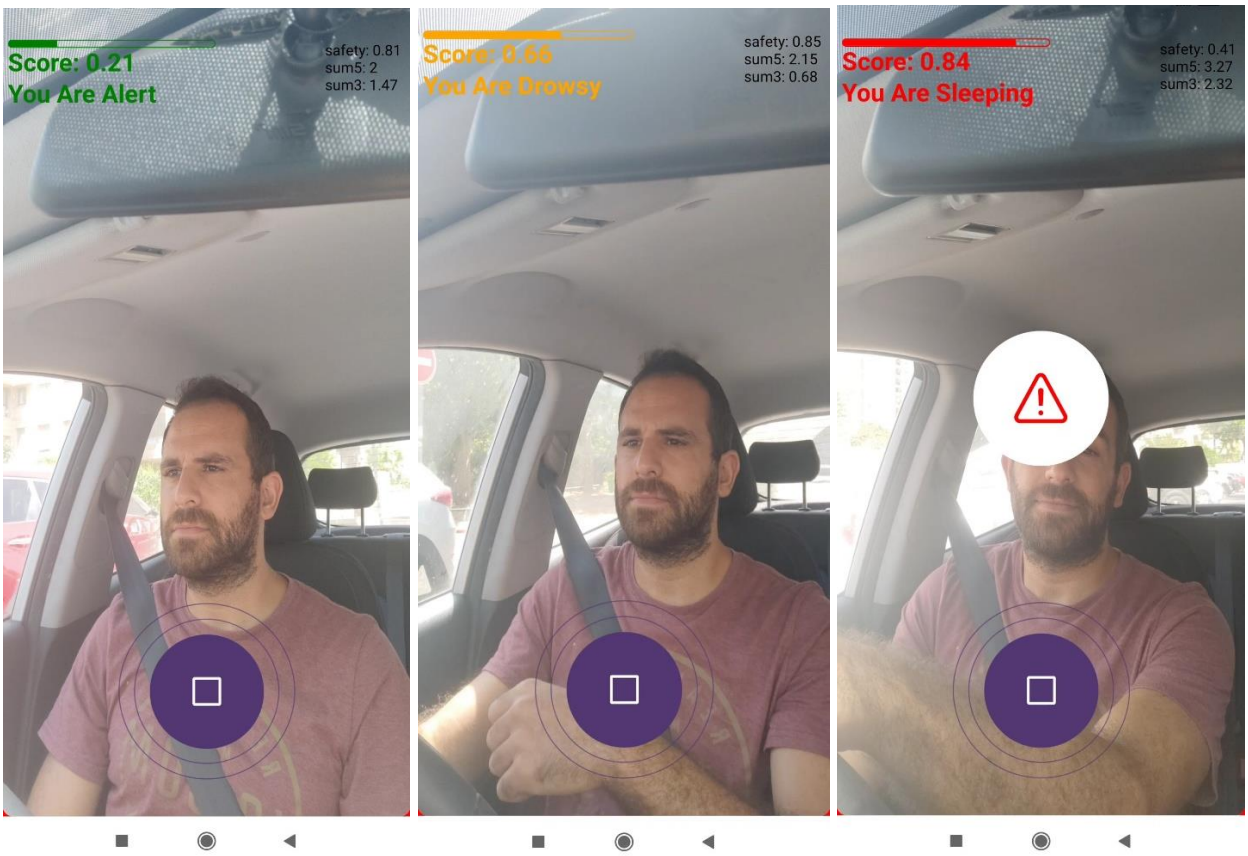


Figure 26: running app fatigue score classification screen

In the CameraView component we first initialize a camera component which connects to the smartphone's front camera. Next in the handleOnCameraReady function we a-synchronically capture a single frame every 1000 milliseconds (1 second) and send it to the backend server using the API in the submitImage function as described in the previous sub-section. A probability prediction for the frame is returned, i.e. the probability that the driver in the image is drowsy. The prediction is returned to the scene component in order to determine the drowsiness score.

3.9. Deployment

3.9.1. Backend Server deployment

To deploy our Project so it can be used anywhere, we used Heroku server to store our backend. Before we can deploy our app to Heroku, you need to initialize a local Git repository and commit our application code to it. We made a connection between our git repository and Heroku server so whenever we push an update to the git repository it will automatically deploy it on Heroku.

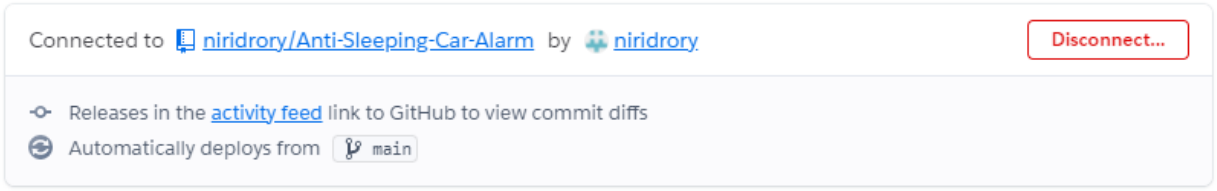


Figure 27: Heroku-Git connection

The deployment process is showed on the following CMD execution:

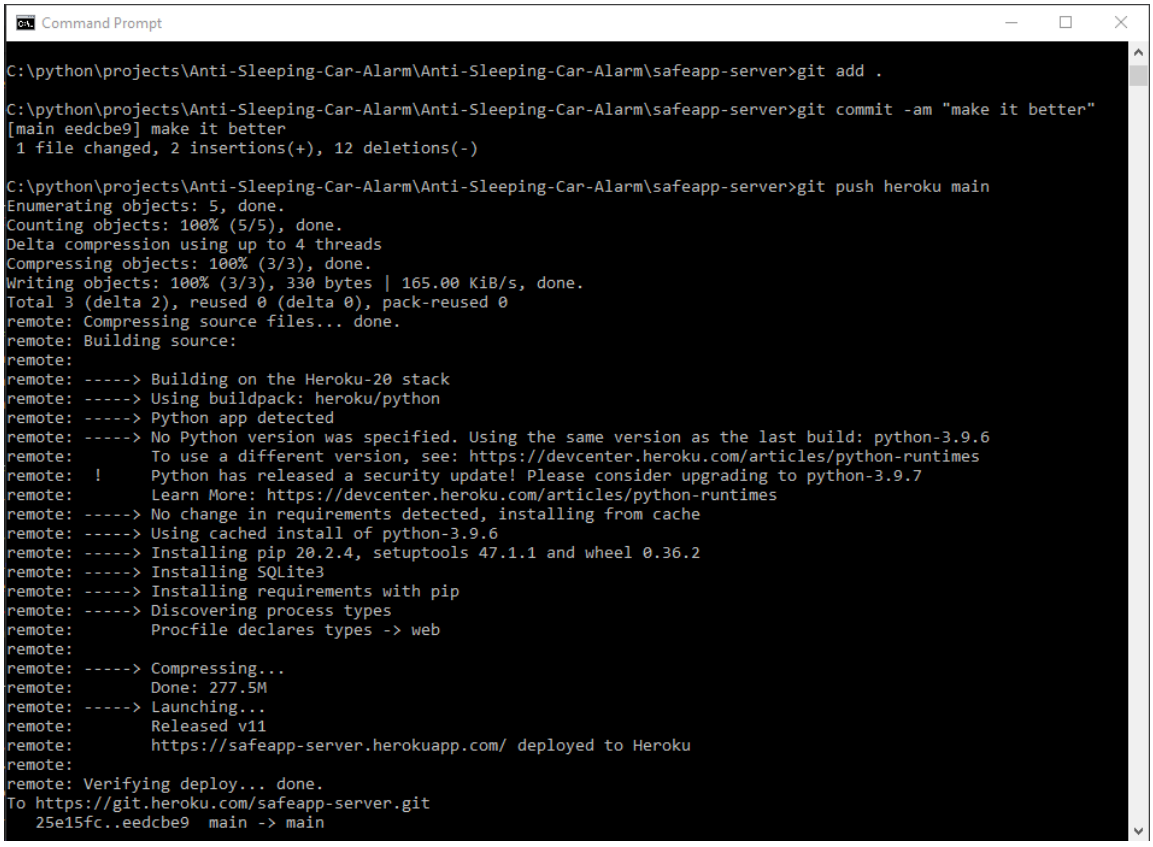


Figure 28: Heroku deployment process

We get confirmation on deploying using the activity panel.

3.9.2. Mobile App deployment

To deploy our App as a standalone .Apk file we use the Expo deployment method for building standalone apps. First we needed to configure the *app.json* file:

```
{
  "expo": {
    "name": "anti-sleep-car-alarm",
    "slug": "anti-sleep",
    "version": "1.0.0",
    "orientation": "portrait",
    // "icon": "./assets/icon.png",
    "icon": "./assets/safeapp-icon.png",
    "splash": {
      "image": "./assets/splash.png",
      "resizeMode": "contain",
      "backgroundColor": "#ffffff"
    },
    "updates": {
      "fallbackToCacheTimeout": 0
    },
    "assetBundlePatterns": [
      "**/*"
    ],
    "ios": {
      "supportsTablet": true
    },
    "android": {
      "adaptiveIcon": {
        "foregroundImage": "./assets/adaptive-icon.png",
        "backgroundColor": "#FFFFFF"
      },
      "package": "antisleep.expo"
    },
    "web": {
      "favicon": "./assets/favicon.png"
    }
  }
}
```

Figure 29: app.json file

In the app.json file we configures the app name, logo and other configurations types. Also in this file we configured ios and android basics configurations so the app can work on both platforms.

Then we continued to building the APK file, there is an option to build an "App Bundle" that helps uploading you app to Google play or Apple apps store, but we choose to create a standalone app for Android devices so it will suit our personal mobile phones. We ran the following command "expo build:android -t apk" that starts the app building process:

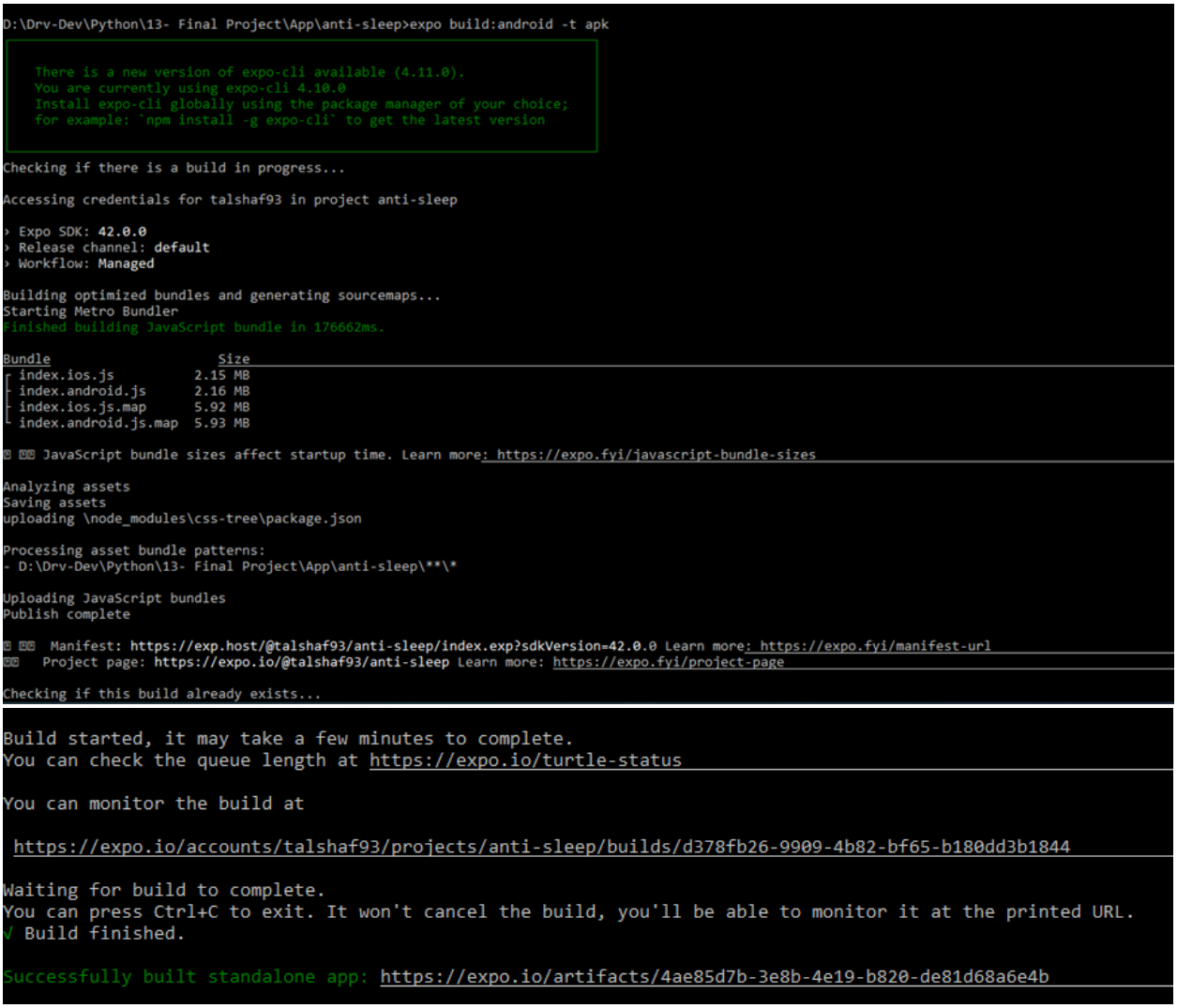


Figure 30: APK building process

While the APK build starts, you app build request is entered to a queue in the Expo app builder server there all the app build process around the world is waiting to be executed. You can watch the queue status in it dashboard:

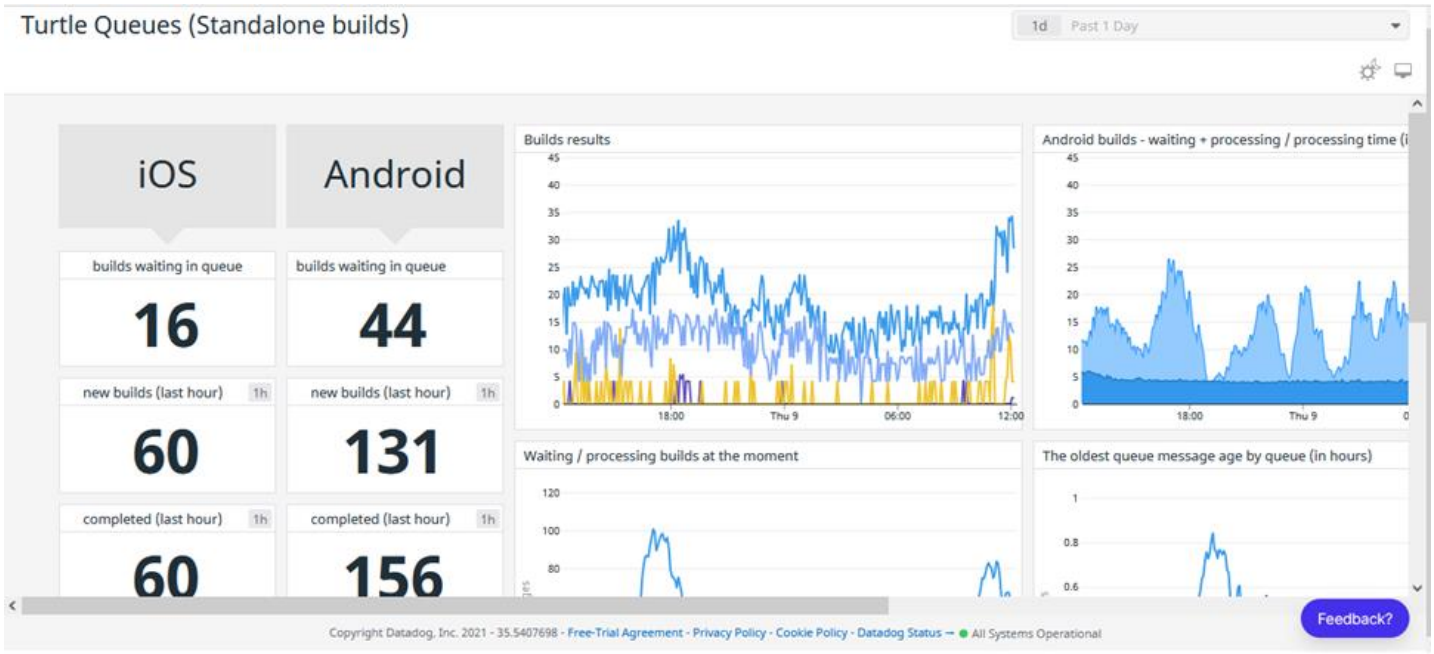


Figure 31: APK builder queues

Section 4: References

1. Project Repository: https://drive.google.com/drive/folders/1ekAtGnRR7vUTkw4ZVM7Kb03muA6DQqF9?usp=sharing

2. "The ULg Multimodality Drowsiness Database (called DROZY) and Examples of Use", by Quentin Massoz, Thomas Langohr, Clémentine François, Jacques G. Verly, Proceedings of the 2016 IEEE Winter Conference on Applications of Computer Vision (WACV 2016), Lake Placid, NY, March 7-10, 2016. [IEEE Xplore] [ORBi] [pdf]

3. Drowsiness Detection with Machine Learning. How our team built a drowsiness detection system in Python. Team Members: Grant Zhong, Rui Ying, He Wang, Aurangzaib Siddiqui, Gaurav Choudhary. https://towardsdatascience.com/drowsiness-detection-with-machine-learning-765a16ca208a

4. OpenCV library, https://opencv.org/

5. Python built-in libraries, https://docs.python.org/3/library.path.html

6. Pandas library, https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html

7. SKlearn library, https://scikit-learn.org/stable/user_guide.html

8. React Native framework, https://reactnative.dev/docs/getting-started

9. FastAPI, https://fastapi.tiangolo.com/tutorial/

10. Dlib c++ library, http://dlib.net/

11. Heroku Server: https://www.heroku.com

12. From Wikipedia, the free encyclopedia:
    a. https://en.wikipedia.org/wiki/OpenCV
    b. https://en.wikipedia.org/wiki/React_Native
    c. https://en.wikipedia.org/wiki/FastAPI
    d. https://en.wikipedia.org/wiki/Git
    e. https://en.wikipedia.org/wiki/GitHub
    f. https://en.wikipedia.org/wiki/Heroku
    g. https://en.wikipedia.org/wiki/SciPy
    h. https://en.wikipedia.org/wiki/NumPy
    i. https://en.wikipedia.org/wiki/Pandas_(software)
    j. https://en.wikipedia.org/wiki/Scikit-learn