A Project/Dissertation Review-1 Report

on

DRIVER DROWSINESS DETECTION SYSTEM

Submitted in partial fulfillment of the

requirement for the award of the degree of

BTECH IN COMPUTER SCIENCE ENGINEERING

GALGOTIAS UNIVERSITY

(Established under Galgotias University Uttar Pradesh Act No. 14 of 2011)

Under The Supervision of Name of
Supervisor :Mr.Michael Raj TF
Designation :Professor

Submitted By

Name of Students-:	Enrollment/Admission No.
Mohd Faizan Alam	18SCSE1050032
Hariom Tripathi	18SCSE1010289

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
GALGOTIAS UNIVERSITY, GREATER NOIDA
INDIA

OCTOBER, 2021

ABSTRACT

Driver fatigue is one of the major causes of accidents in the world. Detecting the drowsiness of the driver is one of the surest ways of measuring driver fatigue. In this project we aim to develop a prototype drowsiness detection system

This document is the research conducted and the project made in the field of computer engineering to develop a system for driver drowsiness detection to prevent accidents from happening because of driver fatigue and sleepiness. The report proposed the results and solutions on the limited implementation of the various techniques that are introduced in the project.

Whereas the implementation of the project give the real world idea of how the system works and what changes can be done in order to improve the utility of the overall system. Furthermore, the paper states the overview of the observations made by the authors in order to help further optimization in the mentioned field to achieve the utility at a better efficiency for a safer road.

This system works by monitoring the eyes of the driver and sounding an alarm when he/she is drowsy. The system so designed is a non-intrusive real-time monitoring system. The priority is on improving the safety of the driver without being obtrusive. In this project the eye blink of the driver is detected. If the drivers eyes remain closed for more than a certain period of time, the driver is said to be drowsy and an alarm is sounded.

The programming for this is done in OpenCV using the Haarcascade library for the detection of facial features. The aim of this project is to develop a prototype drowsiness detection system. The focus will be placed on designing a system that will accurately monitor the open or closed state of the driver's eyes in real- time.

TABLE OF CONTENT

INTRODUCTION

Driver fatigue is a significant factor in a large number of vehicle accidents. Recent statistics estimate that annually 1,200 deaths and 76,000 injuries can be attributed to fatigue related crashes. Because of the hazard that drowsiness presents on the road, methods need to be developed for counteracting its affects.

The aim of this project is to develop a prototype drowsiness detection system. The focus will be placed on designing a system that will accurately monitor the open or closed state of the driver's eyes in real-time.
By monitoring the eyes, it is believed that the symptoms of driver fatigue can be detected early enough to avoid a car accident. Detection of fatigue involves the observation of eye movements and blink patterns in a sequence of images of a face.

Initially, we decided to go about detecting eye blink patterns using Matlab. The procedure used was the geometric manipulation of intensity levels. The algorithm used was as follows.
First we input the facial image using a webcam. Preprocessing was first performed by binarizing the image.
Moving down from the top of the face, horizontal averages of the face area were calculated. Large changes in the averages were used to define the eye area. There was little change in the horizontal average when the eyes were closed which was used to detect a blink.

This is where OpenCV came in. OpenCV is an open source computer vision library. It is designed for computational efficiency and with a strong focus on real time applications. It helps to build sophisticated vision applications quickly and easily. OpenCV satisfied the low processing power and high speed requirements of
our application.

We have used the Haartraining applications in OpenCV to detect the face and eyes. This creates a classifier given a set of positive and negative samples. The steps were as follows:-

- Gather a data set of face and eye. These should be stored in one or more directories indexed by a text file. A lot of high quality data is required for the classifier to work well.
- The utility application createsamples() is used to build a vector output file. Using this file we can repeat the training procedure. It extracts the positive samples from images before normalizing and resizing to specified width and height.

- The Viola Jones cascade decides whether or not the object in an image is similar to the training set. Any image that doesn't contain the object of interest can be turned into negative sample. So in order to learn any object it is required to take a sample of negative background image.

- Training of the image is done using boosting. In training we learn the group of classifiers one at a time. Each classifier in the group is a weak classifier. These weak classifiers are typically composed of a single variable decision tree called stumps. In training the decision stump learns its classification decisions from its data and also learns a weight for its vote from its accuracy on the data. Between training each classifier one by one, the data points are reweighted so that more attention is paid to the data points where errors were made. This process continues until the total error over the dataset arising from the combined weighted vote of the decision trees falls below a certain threshold

What Is OpenCV

OpenCV [OpenCV] is an open source computer vision library available from *http://SourceForge.net/ projects/opencvlibrary*.

OpenCV was designed for computational efficiency and having a high focus on real-time image detection. OpenCV is coded with optimized C and can take work with multicore processors. If we desire more automatic optimization using Intel architectures [Intel], you can buy Intel's Integrated Performance Primitives (IPP) libraries [IPP]. These consist of low-level routines in various algorithmic areas which are optimized. OpenCV automatically uses the IPP library, at runtime if that library is installed.

One of OpenCVs goals is to provide a simple-to-use computer vision infrastructure which helps people to build highly sophisticated vision applications fast. The OpenCV library, containing over 500 functions, spans many areas in vision. Because computer vision and machine learning oft en goes hand-in-hand,

OpenCV also has a complete, general-purpose, Machine Learning Library (MLL). This sub library is focused on statistical pattern recognition and clustering. The MLL is very useful for the vision functions that are the basis of OpenCV's usefulness, but is general enough to be used for any machine learning problem.

Computer Vision

Computer vision is the transforming of data from a still, or video camera into either a representation or a new decision. All such transformations are performed to achieve a particular goal. A computer obtains a grid of numbers from a camera or from the disk, and that's that. Usually, there is no built in pattern recognition or automatic control of focus and aperture, no cross-associations with years of experience. For the most part, vision systems are still fairly naïve.

Origin of OpenCV

OpenCV came out of an Intel Research initiative meant to advance CPU-intensive applications. Toward this end, Intel launched various projects that included real-time ray tracing and also 3D display walls. One of the programmers working for Intel at the time was visiting universities. He noticed that a few top university groups, like the MIT Media Lab, used to have well-developed as well as internally open computer vision infrastructures—code that was passed from one student to another and which gave each subsequent student a valuable foundation while developing his own vision application. Instead of having to reinvent the basic
functions from beginning, a new student may start by adding to that which came before.

Why OpenCV

Specific
OpenCV was designed for image processing. Every function and data structure has been designed with an Image Processing application in mind. Meanwhile, Matlab, is quite generic. You can get almost everything in the world by means of toolboxes. It may be financial toolboxes or specialized DNA toolboxes.

Speedy
Matlab is just way too slow. Matlab itself was built upon Java. Also Java was built upon C. So when we run a Matlab program, our computer gets busy trying to interpret and compile all that complicated Matlab code.
Then it is turned into Java, and finally executes the code.

If we use C/C++, we don't waste all that time. We directly provide machine language code to the computer, and it gets executed. So ultimately we get more image processing, and not more interpreting.
After doing some real time image processing with both Matlab and OpenCV, we usually got very low speeds, a maximum of about 4-5 frames being processed per second with Matlab. With OpenCV however, we get actual real time processing at around 30 frames being processed per second.

Sure we pay the price for speed – a more cryptic language to deal with, but it's definitely worth it. We can do a lot more, like perform some really complex mathematics on images using C and still get away with good enough speeds for your application.

Efficient

Matlab uses just way too much system resources. With OpenCV, we can get away with as little as 10mb RAM for a real-time application. Although with today's computers, the RAM factor isn't a big thing to be worried about. However, our drowsiness detection system is to be used inside a car in a way that is non-intrusive and small; so a low processing requirement is vital.

Thus we can see how OpenCV is a better choice than Matlab for a real-time drowsiness detection system.

In this Python project, we will be using OpenCV for gathering the images from webcam and feed them into a ***Deep Learning*** model which will classify whether the person's eyes are 'Open' or 'Closed'. The approach we will be using for this Python project is as follows :

Step 1 – Take image as input from a camera.

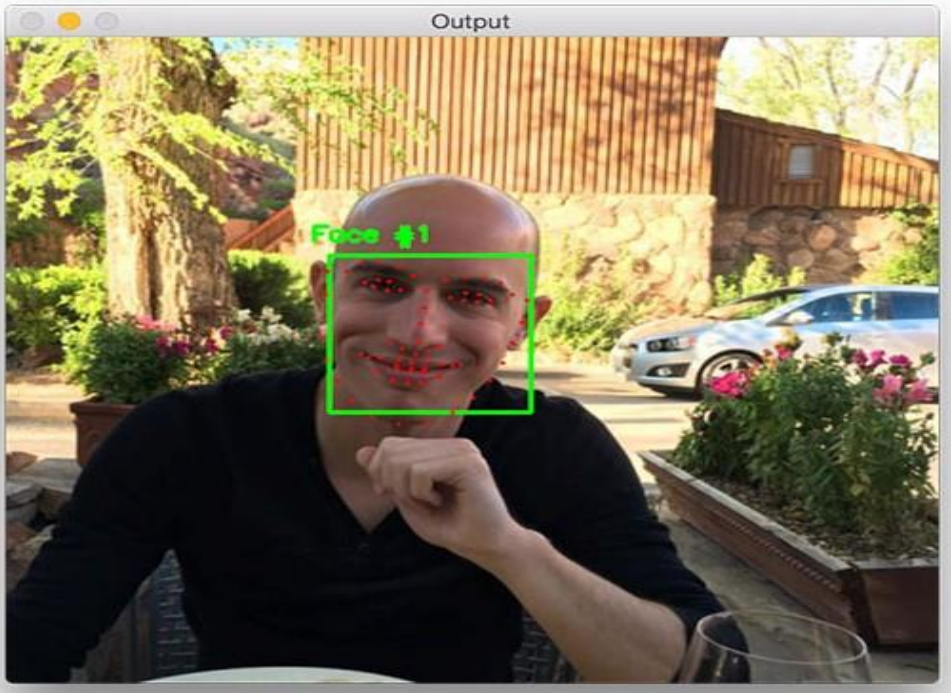
Step 2 – Detect the face in the image and create a Region of Interest (ROI).

Step 3 – Detect the eyes from ROI and feed it to the classifier.

Step 4 – Classifier will categorize whether eyes are open or closed.

Step 5 – Calculate score to check whether the person is drown

FACIAL LANDMARKS



Facial landmarks are used to localize and represent salient regions of the face, such as:

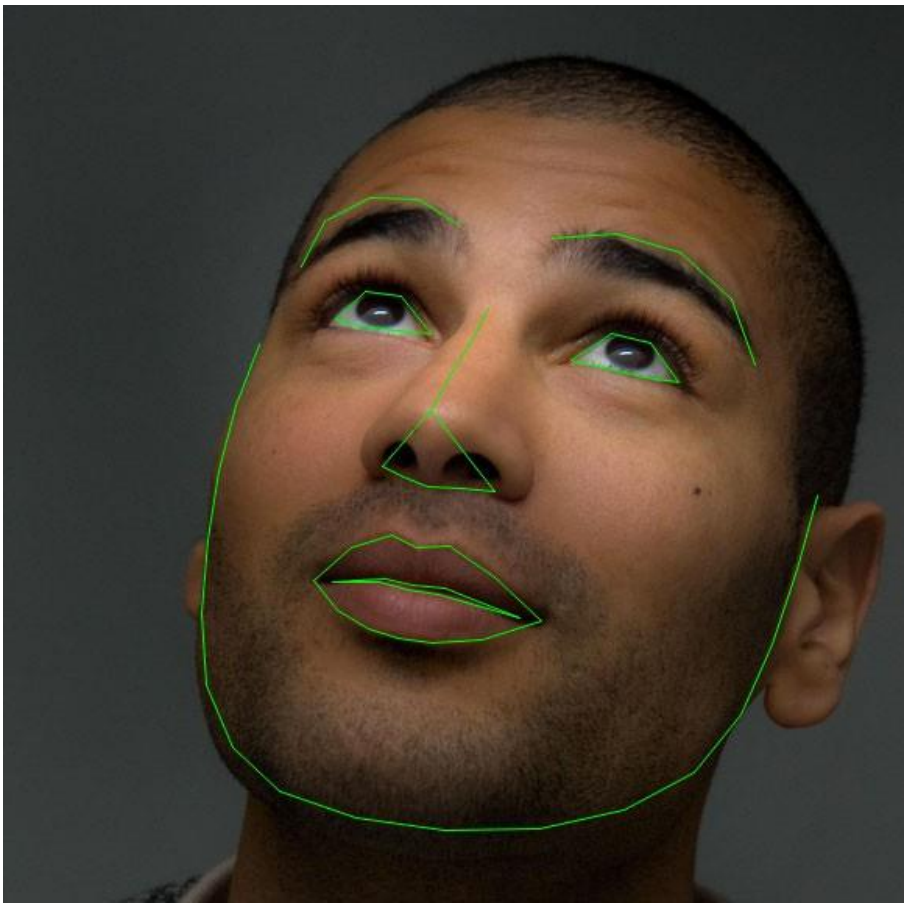
- Eyes
- Eyebrows
- Nose
- Mouth
- Jawline

Facial landmarks have been successfully applied to face alignment, head pose estimation, face swapping, blink detection and much more.
In today's blog post we'll be focusing on the basics of facial landmarks, including:

1. Exactly what facial landmarks are and how they work.
2. How to detect and extract facial landmarks from an image using dlib, OpenCV, and Python.

In the next blog post in this series we'll take a deeper dive into facial landmarks and learn how to extract specific facial regions based on these facial landmarks.

What are facial landmarks



Detecting facial landmarks is a *subset* of the *shape prediction* problem. Given an input image (and normally an ROI that specifies the object of interest), a shape predictor attempts to localize key points of interest along the shape.

In the context of facial landmarks, our goal is detect important facial structures on the face using shape prediction methods.

Detecting facial landmarks is therefore a two step process:

- Step #1: Localize the face in the image.
- Step #2: Detect the key facial structures on the face ROI.

Face detection (Step #1) can be achieved in a number of ways. We could use OpenCV's built-in Haar cascades.

We might apply a pre-trained HOG + Linear SVM object detector specifically for the task of face detection.

Understanding dlib's facial landmark detector

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 can be visualized on the image below:

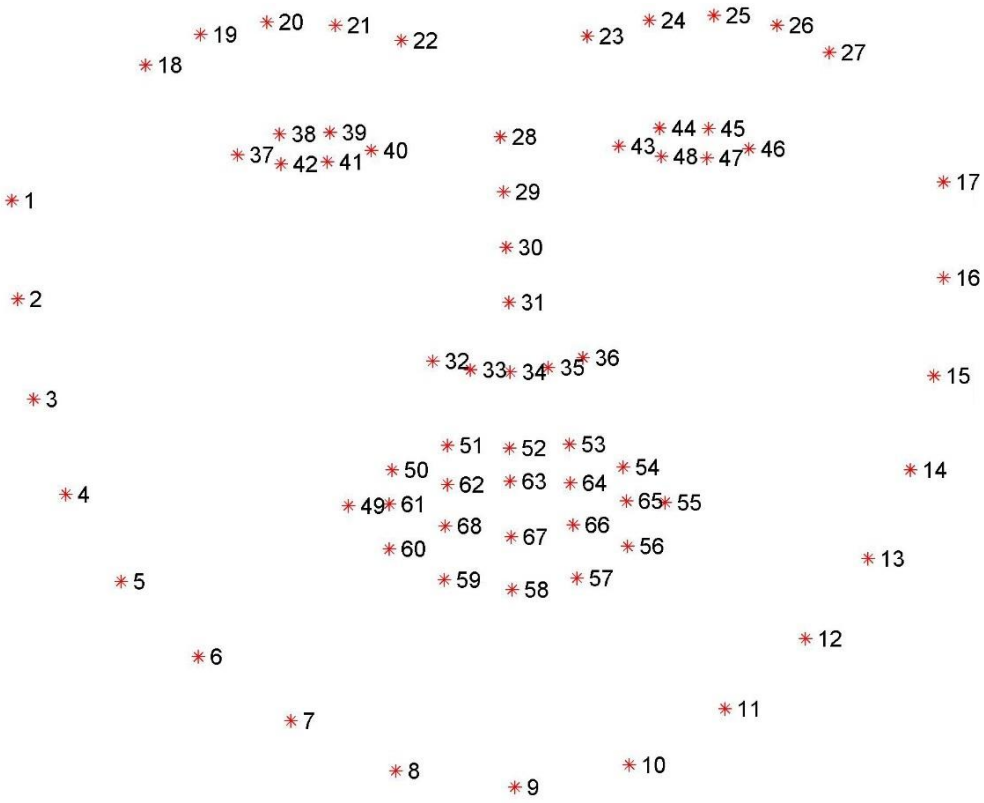


Figure2: Visualizing the 68 facial landmark coordinates from the iBUG 300-W dataset (**higher resolution**).

These annotations are part of the 68 point **iBUG 300-W dataset** which the dlib facial landmark predictor was trained on.

It's important to note that other flavors of facial landmark detectors exist, including the 194 point model that can be trained on the **HELEN dataset**.

Regardless of which dataset is used, the same dlib framework can be leveraged to train a shape predictor on the input training data — this is useful if you would like to train facial landmark detectors or custom shape predictors of your own.

Complete work plan layout

Step 1 – Take Image as Input from a Camera
With a webcam, we will take images as input. So to access the webcam, we made an infinite loop that will capture each frame. We use the method provided by OpenCV, **cv2.VideoCapture(0)** to access the camera and set the capture object (cap). **cap.read()** will read each frame and we store the image in a frame variable.

Step 2 – Detect Face in the Image and Create a Region of Interest (ROI)
To detect the face in the image, we need to first convert the image into grayscale as the OpenCV algorithm for object detection takes gray images in the input. We don't need color information to detect the objects. We will be using haar cascade classifier to detect faces. This line is used to set our classifier **face = cv2.CascadeClassifier(' path to our haar cascade xml file')**. Then we perform the detection using **faces = face.detectMultiScale(gray)**. It returns an array of detections with x,y coordinates, and height, the width of the boundary box of the object. Now we can iterate over the faces and draw boundary boxes for each face.

Step 3 – Detect the eyes from ROI and feed it to the classifier
The same procedure to detect faces is used to detect eyes. First, we set the cascade classifier for eyes in **leye** and **reye** respectively then detect the eyes using **left_eye = leye.detectMultiScale(gray)**. Now we need to extract only the eyes data from the full image. This can be achieved by extracting the boundary box of the eye and then we can pull out the eye image from the frame with this code.

```
1.  l_eye = frame[ y : y+h, x : x+w ]
```

l_eye only contains the image data of the eye. This will be fed into our CNN classifier which will predict if eyes are open or closed. Similarly, we will be extracting the right eye into **r_eye**.

Step 4 – Classifier will Categorize whether Eyes are Open or Closed

We are using CNN classifier for predicting the eye status. To feed our image into the model, we need to perform certain operations because the model needs the correct dimensions to start with. First, we convert the color image into grayscale using **r_eye = cv2.cvtColor(r_eye, cv2.COLOR_BGR2GRAY)**. Then, we resize the image to 24*24 pixels as our model was trained on 24*24 pixel images **cv2.resize(r_eye, (24,24))**. We normalize our data for better convergence **r_eye = r_eye/255** (All values will be between 0-1). Expand the dimensions to feed into our classifier. We loaded our model using **model = load_model('models/cnnCat2.h5')** . Now we predict each eye with our model **lpred = model.predict_classes(l_eye)**. If the value of lpred[0] = 1, it states that eyes are open, if value of lpred[0] = 0 then, it states that eyes are closed.

Step 5 – Calculate Score to Check whether Person is Drowsy

The score is basically a value we will use to determine how long the person has closed his eyes. So if both eyes are closed, we will keep on increasing score and when eyes are open, we decrease the score. We are drawing the result on the screen using cv2.putText() function which will display real time status of the person.

The Dataset:

The dataset used for this model is created by us. To create the dataset, we wrote a script that captures eyes from a camera and stores in our local disk. We separated them into their respective labels 'Open' or 'Closed'. The data was manually cleaned by removing the unwanted images which were not necessary for building the model. The data comprises around 7000 images of people's eyes under different lighting conditions. After training the model on our dataset, we have attached the final weights and model architecture file "models/cnnCat2.h5".

The Model Architecture:

The model we used is built with Keras using Convolutional Neural Networks (CNN). A convolutional neural network is a special type of deep neural network which performs extremely well for image classification purposes. A CNN basically consists of an input layer, an output layer and a hidden layer which can have multiple numbers of layers. A convolution operation is performed on these layers using a filter that performs 2D matrix multiplication on the layer and filter.

The CNN model architecture consists of the following layers:

Convolutional layer; 32 nodes, kernel size 3

Convolutional layer; 32 nodes, kernel size 3

Convolutional layer; 64 nodes, kernel size 3

Fully connected layer; 128 nodes

The final layer is also a fully connected layer with 2 nodes. In all the layers, a Relu activation function is used except the output layer in which we used Softmax.

Classifier will Categorize whether Eyes are Open or Closed:

We are using CNN classifier for predicting the eye status. To feed our image into the model, we need to perform certain operations because the model needs the correct dimensions to start with. First, we convert the color image into grayscale using r_eye = cv2.cvtColor(r_eye, cv2.COLOR_BGR2GRAY). Then, we resize the image to 24*24 pixels as our model was trained on 24*24 pixel images cv2.resize(r_eye, (24,24)). We normalize our data for better convergence r_eye = r_eye/255 (All values will be between 0-1). Expand the dimensions to feed into our classifier. We loaded our model using model = load_model('models/cnnCat2.h5') . Now we predict each eye with our model

lpred = model.predict_classes(l_eye). If the value of lpred[0] = 1, it states that eyes are open, if value of lpred[0] = 0 then, it states that eyes are closed.'

Conclusion

The study has shown promising outcomes in applying the vehicular driver reconnaissance dependent on artificial vision methods and executed in a laptop. The executed framework permits a productive identification of the markers that show up in sleepiness, as long as the estimations are completed under the set up conditions. The right working of the situation relies upon these conditions.

The increment in the handling attributes in cell phones made conceivable to build up a use of counterfeit vision, equipped for recognizing the face and visual markers present in an individual who experiences tiredness, for example, yawning, head developments and the condition of the eyes.

The side effects that individuals present during the change among wakeful and snoozing are showing up as the power of languor increments. The more prominent force of tiredness implies a higher loss of fixation and a lower capacity of driver response. Being developed this work, the usage of 3 degrees of tiredness permits the framework to alarm the driver about their condition, not really at a basic level where it might have genuine repercussions, rather at early levels where sleepiness is simply arising.

A HCI could be actualized utilizing cell phones like appeared in this work, which would permit massify their utilization and accordingly gives more prominent arrangements improving the personal satisfaction of individuals regardless of whether has specials abilities..

1.