**CHAPTER 4**

**IMPLEMENTATION OF THE SYSTEM**

This chapter describes implementation of Driver Drowsiness Detection and Alert System Using Ensemble of Regression Trees.

**4.1. Block Diagram of Training and Testing Datasets**

Figure 4.1 shows the block diagram of training and testing datasets of driver drowsiness detection and alert system using ensemble of regression tress algorithm step by step. Firstly, some percentages of collected dataset are trained with ensemble of regression trees algorithm to build a model. After building model with ensemble of regression trees algorithm, building model can be used to predict each instance in test dataset, which produces facial landmarks location.

Training Datasets

Test Data

Predict Landmarks with Generated Model

**ERT** Algorithm

Trained Model

Facial Landmarks

Figure 4.1. Block Diagram of Training and Testing

**4.2. Flow Chart of the System**

Firstly, read video source from camera when start program and then covert the source into numpy array. Then predict these numpy array data with pre-trained model that produces facial landmarks location. And then, extract eye’s landmark location from facial landmarks location because the system only needs eye’s landmark location and then if close the system, the system will be terminated; if not, calculate the EAR (Eye Aspect Ratio) using EAR formula. Then, the program checks the EAR for “drowsiness” or “not drowsiness”. If the result is “drowsiness”, the system alerts the driver; if not, the system will be started again. The flow chart of the system is as shown in figure 4.2.

Start

Read Video from Camera

Covert into numpy array

Predict with Pre-Trained Model

Calculate EAR

Is Drowsy?

Alert

Is Close?

End

Yes

No

No

Yes

Extract Eye’s Landmark

Figure 4.2 Flow chart of the system design

**4.3. Implementation of the system**

Before implementing the system, the platform has already been installed some editor (note++, visual studio, sublime text, etc..) or jupyter (web based editor) and python 3.0 and upper, and packages.

Some necessary packages in python for this system are as following include:

* Imutils
* Dlib
* Opencv-python
* Time
* Playsound
* Multiprocessing

‘imutils’ package is used for reading video stream from camera, a series of convenience functions to make basic image processing functions such as translation, rotation, resizing, skeletonization, displaying Matplotlib images, sorting contours, detecting edges, and much more easier with OpenCV and both Python 2.7 and Python 3.x, ‘dlib’ package is used for training, predicting and generating model and it also includes ensemble of regression trees algorithm. Dlib is a modern C++ toolkit containing machine learning algorithms and tools for creating complex software in C++ to solve real world problems. It is used in both industry and academia in a wide range of domains including robotics, embedded devices, mobile phones, and large high performance computing environments. Dlib's open source licensing allows you to use it in any application, free of charge.

OpenCV is used for all sorts of image and video analysis, like facial recognition and detection, license plate reading, photo editing, advanced robotic vision, optical character recognition, and a whole lot more. OpenCV is a cross-platform library using which we can develop real-time computer vision applications. It mainly focuses on image processing, video capture and analysis including features like face detection and object detection. ‘opencv-python’ package is only used to display drawed eye’s landmark location and to covert video to numpy array in this system. ‘playsound’ package is used for alert with emergency sound. ‘time’ package is used to wait the camera is opening or closing. ‘mutiprocessing’ packages is used to train data with multi tasks that reduces the processing time. Figure 4.3 shows the installation of necessary packages.

Figure 4.3 shows the installation of necessary packages


Figure 4.3 Installation of the necessary packages

Datasets for this system have been collected from iBUG (Intelligent Behavior Understanding Group) at Imperial College London. There are 35600 instances and 2 xml files (train.xml and test.xml) in the collected datasets. XML files include path of images and feature maps. Extensible Markup Language (XML) is a markup language which encodes documents by defining a set of rules in both machine-readable and human-readable format. Extended from SGML (Standard Generalized Markup Language), it lets us describe the structure of the document. In XML, can define custom tags and can also use XML as a standard format to exchange information. Figure 4.4 shows files structures in the collected datasets and figure 4.5 shows list of images path and its feature maps in xml files.

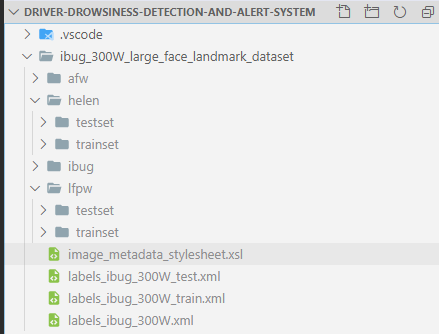


Figure 4.4 File structures in collected datasets

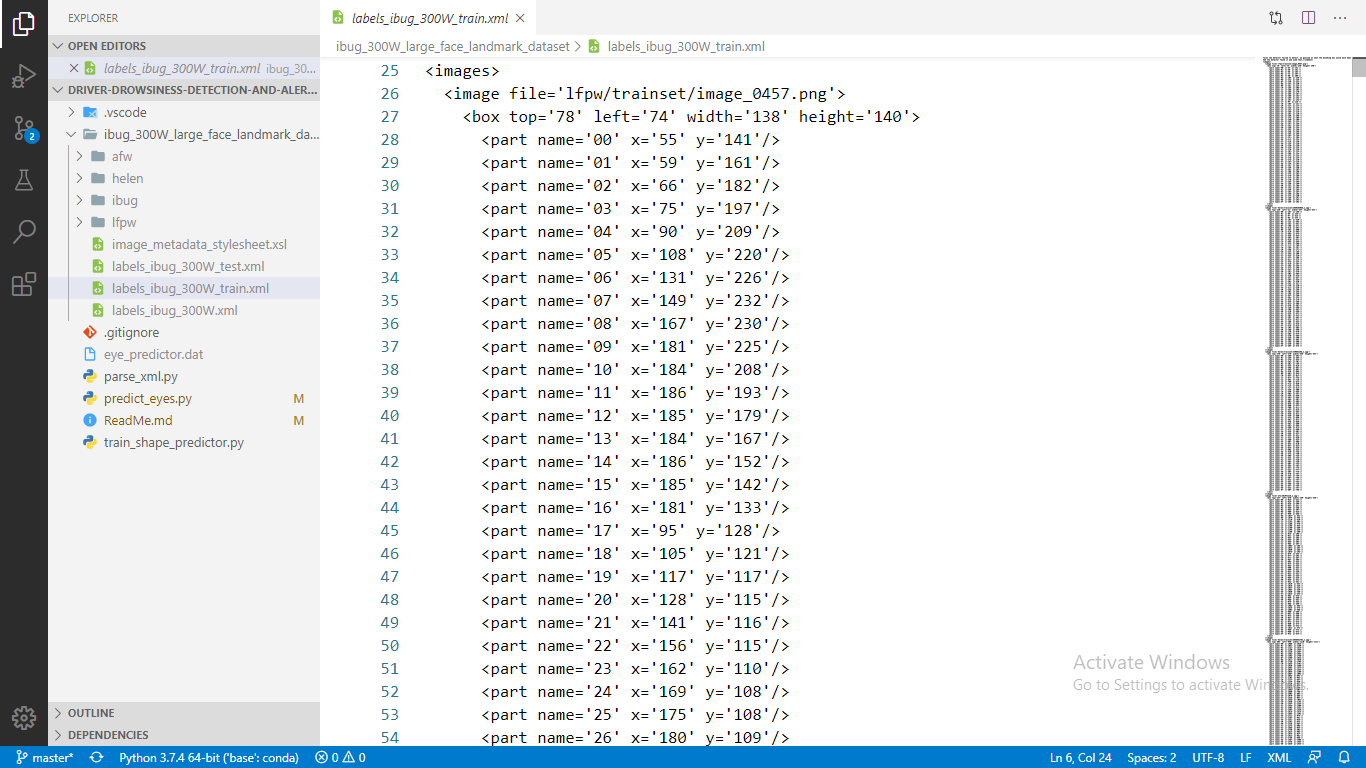


Figure 4.5 List of images’ paths and its feature maps

There are 68 landmarks in each face of collected datasets. In this system doesn’t need all landmarks but only need eye landmarks to train left eye and right eye that is started from 37 to 48. These landmarks already defined by iBUG. Figure 4.6 shows the coordinates of facial landmarks.

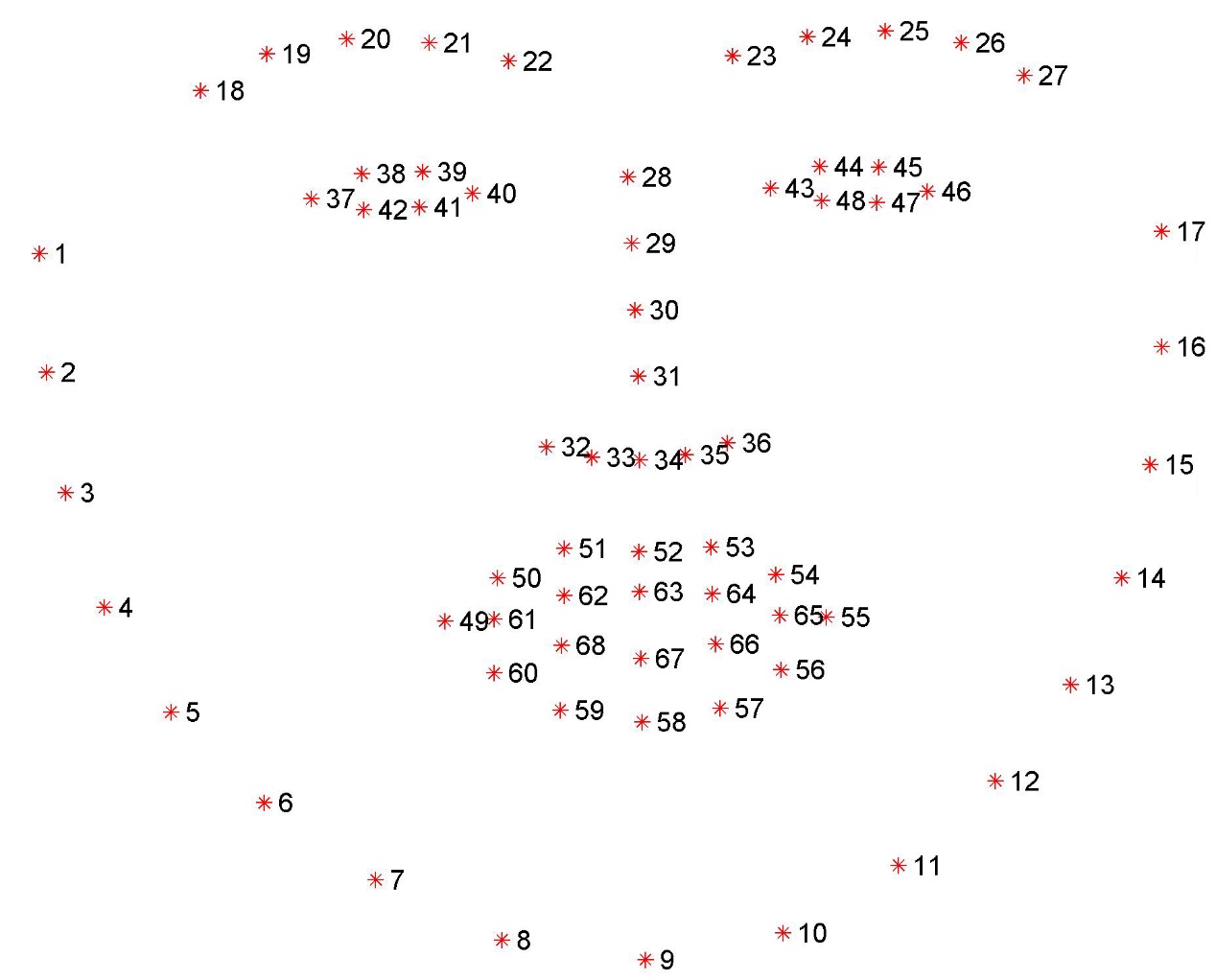


Figure 4.6 Samples of X-Y Coordinates of landmark locations in datasets

iBUG datasets have 68 landmarks for each trainable image. In this system, only needs eye landmarks location. Eye landmarks location start from 37 to 48 according to datasets images. So, extract these locations from 68 landmarks datasets. It can be implemented in python as follow:

# import the necessary packages

# re is regular expression package for search strings with

# regular expression

import re

# integer indexes that belong to the eyes

LANDMARKS = set(list(range(36, 48)))

# to easily parse out the eye locations from the XML file

# utilize regular expressions to determine if there is a 'part'

PART = re.compile("part name='[0-9]+'")

# path of input datasets it have 68 landmarks

# output and input path for training

inputPath4Train =  "datasets/labels\_ibug\_300W\_train.xml"

outputPath4Train = "datasets/labels\_ibug\_300W\_train\_eyes.xml"

# output and input path for testing

inputPath4Test =  "datasets/labels\_ibug\_300W\_test.xml"

outputPath4Test = "datasets/labels\_ibug\_300W\_test\_eyes.xml"

# load 68 landmarks XML file  and then split into array like line  by line

rows4Train = open(inputPath4Train).read().strip().split("\n")

rows4Test = open(inputPath4Test).read().strip().split("\n")

# create or update data for inserting eye locations into xml file

# outputs of eye landmarks xml for training

output4Train = open(outputPath4Train,"w")

# outputs of eye landmarks xml for testing

output4Test = open(outputPath4Test,"w")

# loop over the rows of the data split file for training

print("[INFO] Parsing eye landmarks from 68 facial landmarks for training")

for row in rows4Train:

    # check to see if the current line has the (x, y)-coordinates for

    # the facial landmarks we are interested in

    parts = re.findall(PART, row)

    # if there is no information related to the (x, y)-coordinates of

    # the facial landmarks, we can write the current line out to disk

    # with no further modifications

    if len(parts) == 0:

        output4Train.write("{}\n".format(row))

    # otherwise, there is annotation information that we must process

    else:

        # parse out the name of the attribute from the row

        attr = "name='"

        i = row.find(attr)

        j = row.find("'", i + len(attr) + 1)

        name = int(row[i + len(attr):j])

        # if the facial landmark name exists within the range of our

        # indexes, write it to our output file

        if name in LANDMARKS:

            output4Train.write("{}\n".format(row))

print("[INFO] Parsed eye landmarks for training")

# close the output file

output4Train.close()

# loop over the rows of the data split file for testing

print("[INFO] Parsing eye landmarks from 68 facial landmarks for testing")

for row in rows4Test:

    # check to see if the current line has the (x, y)-coordinates for

    # the facial landmarks we are interested in

    parts = re.findall(PART, row)

    # if there is no information related to the (x, y)-coordinates of

    # the facial landmarks, we can write the current line out to disk

    # with no further modifications

    if len(parts) == 0:

        output4Test.write("{}\n".format(row))

    # otherwise, there is annotation information that we must process

    else:

        # parse out the name of the attribute from the row

        attr = "name='"

        i = row.find(attr)

        j = row.find("'", i + len(attr) + 1)

        name = int(row[i + len(attr):j])

        # if the facial landmark name exists within the range of our

        # indexes, write it to our output file

        if name in LANDMARKS:

            output4Test.write("{}\n".format(row))

print("[INFO] Parsed eye landmarks for testing")

# close the output file

output4Test.close()

After running this program, it generates the output file named ‘labels\_ibug\_300W\_test\_eyes.xml’ and ‘labels\_ibug\_300W\_train\_eyes.xml’ that include eye locations from 36 to 47. The output of eye location file is used for training and testing with ERT algorithm. For training eye landmark, import necessary packages and configure options for ERT algorithm. Before training the eye landmark locations, defines the options for Ensemble of Regression Trees algorithm such as tree depth (depth of each regression trees), cascade depth (the number of cascade used to train), feature pool size (number of pixels used to generate features for the random trees at each cascade), num of test splits (selects best features at each cascade when training), oversampling amount (controls amount of jitter), number of threads (the num of CPU threads), oversampling translation jitter (amount of translation jitter to apply) and verbose (print out status). It can be implemented in python as follow:

import multiprocessing

import argparse

import dlib

#setting options

options = dlib.shape\_predictor\_training\_options()

# define the depth of each regression, [2^0 to 2^3]

options.tree\_depth = 4

# regularization parameter in the range [0, 1]

options.nu = 0.1

# the number of cascades used to train the shape predictor

options.cascade\_depth = 15

# number of pixels used to generate features for the random trees at

# each cascade -- larger pixel values will make your shape predictor

options.feature\_pool\_size = 400

# selects best features at each cascade when training

options.num\_test\_splits = 50

# controls amount of "jitter"

# ability of our model to generalize

options.oversampling\_amount = 5

# amount of translation jitter to apply the dlib docs

# recommend values in the range [0, 0.5]

options.oversampling\_translation\_jitter = 0.1

# tell the dlib shape predictor to be verbose and print out status

# messages our model trains

options.be\_verbose = True

# number of threads/CPU cores to be used when training

# this value to the number of available cores on the system

options.num\_threads = multiprocessing.cpu\_count()

# train the shape predictor

print("[INFO] training shape predictor...")

dlib.train\_shape\_predictor("D:\Thesis\Project\datasets\labels\_ibug\_300W\_train\_eyes.xml", "eye\_predictor.dat", options)

print("[INFO] trained model")

“labels\_ibug\_300W\_train\_eyes.xml” file include path of images and coordinate of eye locations; and “eye\_predictor.dat” is output file name that is used to save the trained model into a file. The training time is about 2 hours taken in CPU i7 4510U @2.0GHz , RAM 8GB and GPU 2GB.

After training the datasets, the progarm generates the model. The model is used to predict eye landmark location for drowsiness that is produced eye landmarks in numpy shape arrary according to input image. The output numpy array has 12 lengths that array index starts from 0 to 11; and index for right eye starts from 0 to 5 and left eye starts from 6 to 11. According to left eye and right eye landmarks, eye aspect ration (EAR) can be calculated with EAR formular. For prediction, import necessary packages and compose the code to predict the eye landmark location and EAR calculation. It can be implemented as follow:

import imutils

# for reading video source from camera

from imutils.video import VideoStream

# for finding face

from imutils import face\_utils

# for define any changes

import imutils

# for define during eye blink

import time

# for ERT algorithm

import dlib

# for display and convert into numpy array

import cv2

# for convert image into numpy array

import numpy as np

#for play emegency sound

import playsound

# initialize dlib's face detector (HOG-based) and then load our

# trained shape predictor

print("[INFO] loading facial landmark predictor...")

detector = dlib.get\_frontal\_face\_detector()

predictor = dlib.shape\_predictor("eye\_predictor.dat")

# initialize the video stream and allow the cammera sensor to warmup

print("[INFO] camera sensor warming up...")

vs = VideoStream(src=0).start()

time.sleep(2.0)

def euclidean\_dist(ptA, ptB):

    # compute and return the euclidean distance between the two points

    return np.linalg.norm(ptA - ptB)

def eye\_aspect\_ratio(eye):

    # compute the euclidean distances between the two sets of

    # vertical eye landmarks (x, y)-coordinates

    A = euclidean\_dist(eye[1], eye[5])

    B = euclidean\_dist(eye[2], eye[4])

    # compute the euclidean distance between the horizontal

    # eye landmark (x, y)-coordinates

    C = euclidean\_dist(eye[0], eye[3])

    # compute the eye aspect ratio

    ear = (A + B) / (2.0 \* C)

    # return the eye aspect ratio

    return ear

# loop over the frames from the video stream

EYE\_AR\_THRESH = 0.2

EYE\_AR\_CONSEC\_FRAMES = 16

# initialize the frame counter as well as a boolean used to

# indicate if the alarm is going off

COUNTER = 0

ALARM\_ON = False

while True:

    # grab the frame from the video stream, resize it to have a

    # maximum width of 400 pixels, and convert it to grayscale

    frame = vs.read()

    frame = imutils.resize(frame, width=400)

    frame = cv2.flip(frame, 1)

    gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

    # detect faces in the grayscale frame

    rects = detector(gray, 0)

    # loop over the face detections

    for rect in rects:

        # convert the dlib rectangle into an OpenCV bounding box and

        # draw a bounding box surrounding the face

        (x, y, w, h) = face\_utils.rect\_to\_bb(rect)

        # cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)

        # use our custom dlib shape predictor to predict the location

        # of our landmark coordinates, then convert the prediction to

        # an easily parsable NumPy array

        shape = predictor(gray, rect)

        shape = face\_utils.shape\_to\_np(shape)

        rightEye = shape[0:6]

        leftEye = shape[6:]

        rightEAR = eye\_aspect\_ratio(rightEye)

        leftEAR = eye\_aspect\_ratio(leftEye)

        ear = (leftEAR + rightEAR) / 2.0

        leftEyeHull = cv2.convexHull(leftEye)

        rightEyeHull = cv2.convexHull(rightEye)

        cv2.drawContours(frame, [leftEyeHull], -1, (0, 255, 0), 1)

        cv2.drawContours(frame, [rightEyeHull], -1, (0, 255, 0), 1)

        if ear < EYE\_AR\_THRESH:

            COUNTER += 1

            if COUNTER >= EYE\_AR\_CONSEC\_FRAMES:

                if not ALARM\_ON:

                    ALARM\_ON = True

                # cv2.putText(frame, "DROWSINESS ALERT!", (10, 30),cv2.FONT\_HERSHEY\_SIMPLEX, 0.7, (0, 0, 255), 2)

                playsound.playsound('alarm.mp3', True)

        else:

            COUNTER = 0

            ALARM\_ON = False

        # loop over the (x, y)-coordinates from our dlib shape

        # predictor model draw them on the image

        # for (sX, sY) in shape[6:]:

        #   cv2.circle(frame, (sX, sY), 1, (0, 0, 255), -1)

        cv2.putText(frame, "EAR: {:.3f}".format(ear), (250, 30),cv2.FONT\_HERSHEY\_SIMPLEX, 0.7, (0, 0, 255), 2)

    # show the frame

    cv2.imshow("Drowsiness Detection And Alert System by CYA", frame)

    key = cv2.waitKey(1) & 0xFF

    # if the `q` key was pressed, break from the loop

    if key == ord("q"):

        break

# do a bit of cleanup

cv2.destroyAllWindows()

vs.stop()

**4.6. Summary**

In this chapter, design and implementation of driver drowsiness detection and alert system using ensemble of regression tress algorithm has been described with respective figures.