**A**

**PROJECT REPORT**

**ON**

**CALCULATING THE SCREEN TIME OF ACTORS AND DETECTING**

**AND NAMING THEM IN ANY VIDEO**

*Submitted in partial fulfillment of the*

*Requirements for the award of the degree*

*Of*

**Bachelor of Technology**

in

**Information Technology**

By:

**Arjun Sharma (00813203115/IT1)  
Aryan Veer Singh (01113203115/IT1)**

Under the guidance of

**Mrs. Meenakshi**



**Department of Information Technology**

**Guru Tegh Bahadur Institute of Technology**

**Guru Gobind Singh Indraprastha University, Dwarka, New Delhi**

**Year 2018-2019**

# CERTIFICATE

This is to certify that dissertation entitled **“CALCULATING THE SCREEN TIME OF ACTORS AND DETECTING AND NAMING THEM IN ANY VIDEO”,** which is submitted by **Mr. Arjun Sharma** and **Mr. Aryan Veer Singh** in partial fulfillment of the requirement for the award of the degree of **Bachelor of Technology in Information Technology**, Guru Tegh Bahadur Institute of Technology, New Delhi is an authentic record of the candidate’s own work carried out by them under our guidance. The matter embodied in this thesis is original and has not been submitted for the award of any other degree.

Meenakshi P.S. Bedi

Project Mentor Head of Department

Information Technology

**Date:**

**ACKNOWLEDGEMENT**

We would like to express our great gratitude towards our supervisor, **Mrs. Meenakshi** who has given us support and suggestions. Without their help we could not have presented this dissertation up to the present standard. We also take this opportunity to give thanks to all others who gave us support for the project or in other aspects of our study at Guru Tegh Bahadur Institute of Technology.

Date: Arjun Sharma  
 (08/IT1/2015)  
 arjun\_sharma97@hotmail.com

Aryan Veer Singh  
(11/IT1/2015)

aryanveer310@gmail.com

**ABSTRACT**

The face is one of the easiest ways to distinguish the individual identity of each other.

Face recognition is a personal identification system that uses personal characteristics of a person to identify the person's identity. The face of a human being conveys a lot of information about identity and emotional state of the person. Face recognition is an interesting and challenging problem, and impacts important applications in many areas such as identification for law enforcement, authentication for banking and security system access, and personal identification among others. The prevalent techniques and methodologies for detecting and recognizing face like PCA (principal component analysis)-LDA (Linear Discriminant Analysis), etc. fail to overcome issues such as scaling, pose, illumination, variations, rotation, and occlusions.An effective and real time face recognition system based on OpenCV and the Local Binary Patterns Histograms algorithms is developed in the project. Local Binary Patterns (LBP) is a non-parametric descriptor whose aim is to efficiently summarize the local structures of images. In recent years, it has aroused increasing interest in many areas of image processing and computer vision, and has shown its effectiveness in a number of applications, in particular for facial image analysis, including tasks as diverse as face detection, face recognition, facial expression analysis, demographic classification, etc.

In our project we also calculate the Screen Time of Actors in any Video using Deep Learning and Neural Networks. We will be using the pre trained VGG16 Model. VGG is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper “Very Deep Convolutional Networks for Large-Scale Image Recognition”. The model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes.

**CONTENTS**

**Chapter Page No.**

Certificate i

Acknowledgement ii

Abstract iii

Tables and figures iv

1. Introduction 2
2. Software Requirements Specification 6

2.1. Introduction 6

2.2. Overall Description 7

2.3. Development Requirements 8

2.4. Other Non-Functional Requirements 11

1. System Design 13

3.1. Flowchart 13

3.2. Calculating Screen Time 14

3.3. Face Recognition 21

1. Thesis 31

4.1. Neural Network 31

4.2. Deep Learning 34

4.3. LBPH 35

4.4 Algorithm 37

4.5 Keras 39

4.6 Using Keras 40

5. Conclusions 42

6. Future Scope 43 7. References 44

8. Appendix 45

**TABLES AND FIGURES**

**Figures Description Page No.**

Figure 1. Jupyter IDE Interface 9

Figure 2. Image Plot 16

Figure 3. Summary of our model 18

Figure 4. Epochs 19

Figure 5. Represents all three steps 21

Figure 6. Image plot with face recognized 23

Figure 7. Data gathering 24

Figure 8. Images to train 25

Figure 9. Trainer 25

Figure 10. Recognition 27

Figure 11. A simple perceptron 33

Figure 12. A perceptron with bias 33

Figure 13. Graph (images vs Recognition rate) 36

Figure 14. LBP Detector 37

Figure 15. Neighbors with variable radius 38

Figure 16. LBPH Images 39

**Chapter One**

**INTRODUCTION**

**INTRODUCTION**

Face detection is a fundamental task for applications such as face tracking, red-eyeremoval, face recognition and face expression recognition. Several face recognition algorithms are also used in many different applications apart from biometrics, such as video compressions, indexing’s etc. They can also be used to classify multimedia content, to allow fast and efficient searching for material that is of interest to the user. An efficient face recognition system can be of great help in forensic sciences, identification for law enforcement, surveillance, authentication for banking and security system, and giving preferential access to authorized users i.e. access control for secured areas etc. To build flexible systems which can be executed on mobile products, like handheld PC’s and mobile phones, efficient and robust face detection algorithms arerequired. Most of existing face detection algorithms consider face detection as binary (two-class) classification problem. Even though it looks a simple classification problem, it is very complex to build a good face classifier.Therefore, learning-based approaches, such as neural network-based methods or supports vector machine (SVM) methods, have been proposed to find a good classifier. Most of proposed algorithms use pixel values as features. However, they are very sensitive to illumination conditions and noises.

Existing methods like Eigenfaces and Fisherfaces take a somewhat holistic approach to face recognition. You treat your data as a vector somewhere in a high-dimensional image space. We all know high-dimensionality is bad, so a lower-dimensional subspace is identified, where (probably) useful information is preserved. The Eigenfaces approach maximizes the total scatter, which can lead to problems if the variance is generated by an external source, because components with a maximum variance over all classes aren’t necessarily useful for classification. So to preserve some discriminative information we apply a Linear Discriminant Analysis (LDA) and optimizeit. The Fisherfaces method works greatonly for perfectly constrained scenario which is not the case every time. You simply can’t guarantee perfect light settings in your images or 10 different images of a person. So what if there’s only one image for each person? Our covariance estimates for the subspace may be horribly wrong, so will the recognition.So in order to get good recognition rates you’ll need at least 8(+-1) images for each person and the Fisherfaces method doesn’t really help here.

During the last few years, Local Binary Patterns (LBP) has aroused increasing interest in image processing and computer vision. As a non-parametric method, LBP summarizes local structures of images efficiently by comparing each pixel with its neighboring pixels. The most important properties of LBP are its tolerance regarding monotonic illumination changes and its computational simplicity. LBP was originally proposed for texture analysis, and has proved a simple yet powerful approach to describe local structures. It has been extensively exploited in many applications, for instance, face image analysis, image and video retrieval, environment modeling, visual inspection, motion analysis, biomedical and aerial image analysis, remote sensing, so forth.

Coming on to our second part of project that is “Calculating the on screen time of an Actor in a Video” using deep learning. Now to give you some context on the problem we’ll be solving, keep in mind that screen time is extremely important for an actor. It is directly related to the money he/she gets. Just to give you a sense of this commission, did you know that Robert Downey Jr. Downey picked up $10 million for just 15 minutes of screen time in “Spider-Man Homecoming”? Also many Oscar winning actors appeared in their respective movies for a very short period of time and were still able to achieve the critics appreciation.

* Anthony Hopkins won an Oscar for Silence of the Lambs, but only appeared on-screen for 16 minutes.
* Beatrice Straight won an Oscar for her role in Network. She only appeared on-screen for 5 minutes and 40 seconds.
* Judy Dench won for Shakespeare in Love but only appeared on-screen for about 8 minutes.

We will be using keras and pre trained VGG16 neural network models in this part of our project. Keras is a high level library, used specially for building neural network models. It is written in Python and is compatible with both Python – 2.7 & 3.5. Keras was specifically developed for fast execution of ideas. It has a simple and highly modular interface, which makes it easier to create even complex neural network models. This library abstracts low level libraries, namely Theano and TensorFlow so that, the user is free from “implementation details” of these libraries.

The key features of Keras are:

* **Modularity:** Modules necessary for building a neural network are included in a simple interface so that Keras is easier to use for the end user.
* **Minimalistic:** Implementation is short and concise.
* **Extensibility:** It’s very easy to write a new module for Keras and makes it suitable for advance research.

VGG is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper “Very Deep Convolutional Networks for Large-Scale Image Recognition”. The model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes.

**Chapter Two**

**SOFTWARE REQUIREMENTS SPECIFICATION**

**SOFTWARE REQUIREMENTS SPECIFICATION**

**2.1. INTRODUCTION**

This section gives a scope description and overview of everything included in this SRS document. Also, the purpose for this document is described is provided.

**2.1.1. Purpose**

The purpose of this document is to present a detailed description of the Machine Learning and Deep Learning project on face recognition in a video and calculating the on screen time of an actor in a video. It will explain the importance and features of Machine Learning, Neural networks, Deep Learning and its use for complex algorithms like Local Binary Patterns Histograms, Keras and VGG16. We will also be comparing our results with other various methods that have been used in the past and shows their limitations over ours.It will also present the applications and the interfaces of the project, what will be done in the project and the constraints under which it must operate.

**2.1.2. Document Conventions**

Main Section Titles

* Font: Times New Roman
* Face: Bold
* Size: 14

Sub Section Titles

* Font: Times New Roman
* Face: Bold
* Size: 12

Other Text Explanations

* Font: Times New Roman
* Face: Normal
* Size: 12

**2.1.3. Intended Audience and Reading Suggestions**

This document is intended for students, developers, learners, documentation writers etc.

This document is organized as follows:

* Section 1: Introduction (this section)

This Section provides a brief introduction to this documentation, the purpose, document conventions, intended audience, reading suggestions, project scope, and references.

* Section 2: Overall Description

This Section provides brief general descriptions of the project and its functions, user classes and characteristics, operating environments etc.

* Section 3: Development Environment and Technologies Used

Provides the detailed information of user, hardware and software interfaces.

* Section 4: Other Nonfunctional requirements

Provides information regarding performance requirements, safety requirements, security requirements, software quality attributes and business rules.

**2.1.4. Product Scope**

The overall scope of this product in the industry is huge. Our product is more accurate and precise then the one’s already available in the industry. Our product also provides more flexibility to the user by giving an option to a do a real time recognition.

**2.2. OVERALL DESCRIPTION**

**2.2.1. Product Perspective**

The Face recognitionand screen on time of an actor will be a Python based application. It will be implemented with Jupyter Notebook.

**2.2.2. Product Functions**

The product should have an easy to understand layout. The only pre-requisites required are the thorough understanding of the algorithms. Then further it can be applied to numerous fields.

**2.2. User Classes and Characteristics**

This project hopes to draw on domain specific groups. In whatever industry where face recognition is required we can use it such as video compressions, indexing’s etc. They can also be used to classify multimedia content, to allow fast and efficient searching for material that is of interest to the user. An efficient face recognition system can be of great help in forensic sciences, identification for law enforcement, surveillance, authentication for banking and security system, and giving preferential access to authorized users i.e. access control for secured areas etc. Meanwhile the screen on time application can help in collecting statistics of different actors in a movie.

**2.2.4. Operating Environment**

The application will only be available for systems with Python installed. The user also needs to install various other packages required using pip3.

**2.2.5. Design and Implementation Constraints**

Implementation language restrictions:

The programming language shall be Python for the main project.

The user should have all the required libraries installed.  
Resource limits:

The users‟ device shall have a working data plan or Wi-Fi connection.

The users‟ device shall have sufficient memory storage and CPU & GPU.

**2.3. DEVELOPMENT REQUIREMENTS AND TECHNOLOGIES USED**

For the development of the model, the minimum software requirements should be as follows.

**2.3.1 Software Requirements**

The following Software Requirements should meet:

1. Python Version 3.6 and above
2. Jupyter Notebook
3. Libraries required:
   1. Numpy
   2. Pandas
   3. Keras
   4. Matplotlib
   5. OpenCV
   6. Skimage

(4) TensorFlow

**2.3.2. Technologies Used**

(1)Jupyter Notebook

The Jupyter Notebook is an open-source web application that allows you tocreate and share documents that contain live code, equations, visualizations and explanatory text. Jupyter Notebook allows:

● creation in a standard web browser

● direct sharing

● easy creation and display of beautiful equations

● easy creation and display of interactive visualizations

IPython provides a rich architecture for interactive computing with:

• A powerful interactive shell.

• A kernel for Jupyter.

• Support for interactive data visualization and use of GUI toolkits.

• Flexible, embeddable interpreters to load into your own projects.

• Easy to use, high performance tools for parallel computing.

IPython is a growing project, with increasingly language-agnostic components. IPython 3.x was the last monolithic release of IPython, containing the notebook server, qtconsole, etc. As of IPython 4.0, the language-agnostic parts of the project: the notebook format, message protocol, qtconsole, notebook web application, etc. have moved to new projects under the name Jupyter. IPython itself is focused on interactive Python, part of which is providing a Python kernel for Jupyter.

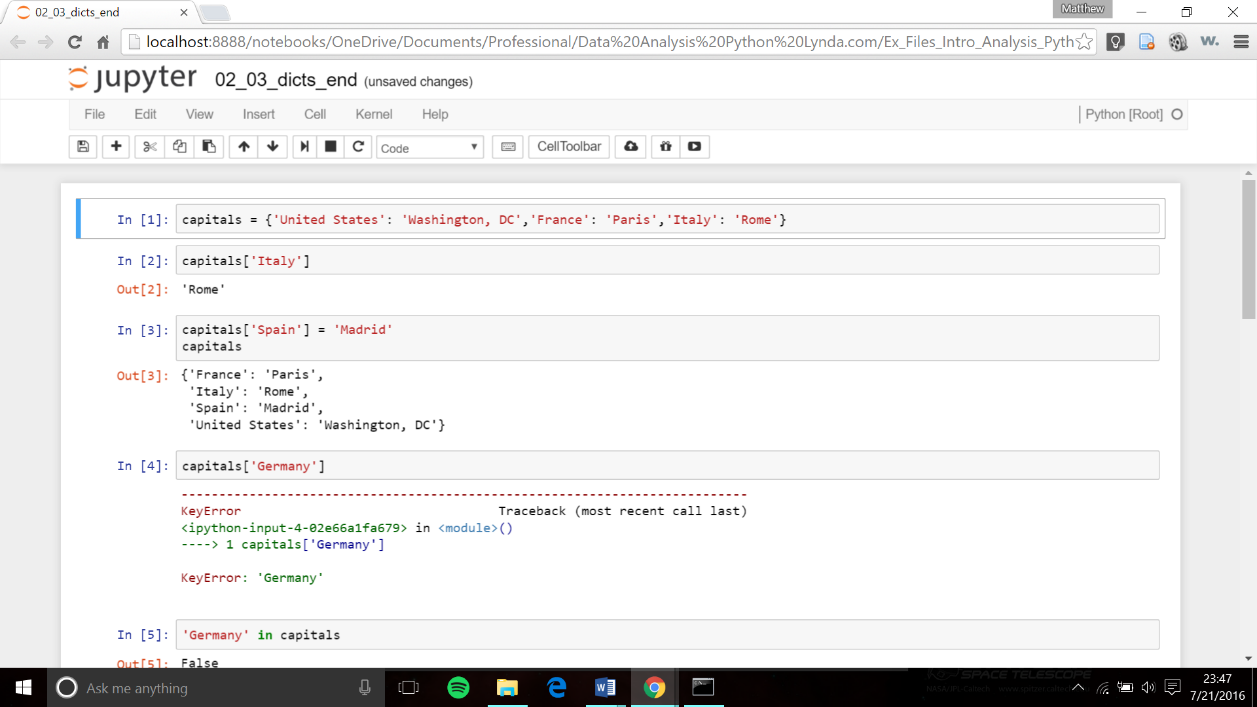


Fig 1. Jupyter IDE Interface

(2) Python

Python is a general-purpose language. It has wide range of applications from Machine Learning, Web development (like: Django and Bottle), scientific and mathematical computing (Orange, SymPy, NumPy) to desktop graphical user Interfaces (Pygame, Panda3D).The syntax of the language is clean and length of the code is relatively short. It is fun to work in Python because it allows you to think about the problem and logic rather than focusing on the syntax.

(3) Deep Learning:  
Deep learning is a set of learning methods attempting to model data with complex architectures combining different non-linear transformations. The elementary bricks of deep learning are the neural networks, that are combined to form the deep neural networks.

These techniques have enabled significant progress in the fields of sound and image processing, including facial recognition, speech recognition, computer vision, automated language processing, text classification (for example spam recognition). Potential applications are very numerous. A spectacularly example is the AlphaGo program, which learned to play the Go game by the deep learning method, and defeated the world champion in 2016.

There exist several types of architectures for neural networks:

• The multilayer perceptron’s, that are the oldest and simplest ones

• The Convolutional Neural Networks (CNN), particularly adapted for image processing

• The recurrent neural networks, used for sequential data such as text or times series.

They are based on deep cascade of layers. They need clever stochastic optimization algorithms, and initialization, and also a clever choice of the structure.

They lead to very impressive results, although very few theoretical foundations are available till now.

(4) Neural Networks:  
An artificial neural network is an application, nonlinear with respect to its parameters θ that associates to an entry x an output y = f (x, θ). For the sake of simplicity, we assume that y is unidimensional, but it could also be multidimensional. This application f has a particular form that we will precise.

The neural networks can be used for regression or classification. As usual in statistical learning, the parameters θ are estimated from a learning sample. The function to minimize is not convex, leading to local minimizers. The success of the method came from a universal approximation theorem due to Cybenko (1989) and Hornik (1991). Moreover, Le Cun (1986) proposed an efficient way to compute the gradient of a neural network, called backpropagation of the gradient, that allows to obtain a local minimizer of the quadratic criterion easily.

**2.4. OTHER NONFUNCTIONAL REQUIREMENTS**

**2.4.1. Performance Requirements**

Resource consumption of this project should not reach an amount that renders the system unusable. The project should be capable of operating in the background should the user wish to utilize other applications.

**2.4.2. Safety Requirements**

● User needs to allow the application to use Wi-Fi.

● User shall not use this project on low configuration machine.

**2.4.3. Reliability**

The application will meet all of the functional requirements without any unexpected behavior. At no time should the gauge output display incorrect or outdated information without alerting the user to potential errors.

**2.4.4. Availability**

The project will work with availability of Development Requirements and Technologies used.

**2.4.5. Security**

The project should never disclose any personal information of any users, and should collect no personal information from its own users.

**2.4.6. Maintainability**

The project can be maintained easily.

**2.4.7. Portability**

This project will be designed to run on any Computer System with Python version 3.6 or higher, Jupyter Notebook.

**Chapter Three**

**SYSTEM DESIGN**

**SYSTEM DESIGN**

**3.1. FLOW CHART**

Reading a video and extracting frames

Gathering Data

Handling video files in Python

Detecting Faces

Recognizing the Face

Training the Model

Calculating the screentime

**3.1.1. CALCULATING THE SCREEN ON TIME OF ACTORS IN A VIDEO**First let’s import the necessary libraries

import cv2 # for capturing videos

import math # for mathematical operations

import matplotlib.pyplot as plt # for plotting the images

%matplotlib inline

import pandas as pd

from keras.preprocessing import image # for preprocessing the images

import numpy as np # for mathematical operations

from keras.utils import np\_utils

from skimage.transform import resize # for resizing images

**Step – 1: Read the video, extract frames from it and save them as images**

First We will load the video and convert it into frames.We will first capture the video from the given directory using the VideoCapture() function, and then we’ll extract frames from the video and save them as an image using the imwrite() function.

count = 0

videoFile = "Train.mp4"

cap = cv2.VideoCapture(videoFile)   # capturing the video from the given path

frameRate = cap.get(5) #frame rate

x=1

while(cap.isOpened()):

frameId = cap.get(1) #current frame number

    ret, frame = cap.read()

if (ret != True):

       break

    if (frameId % math.floor(frameRate) == 0):

       filename ="frame%d.jpg" % count;count+=1

       cv2.imwrite(filename, frame)

cap.release()

print ("Done!")

**Step – 2: Label a few images for training the model**

Now to train the model we need to label a few of images and train the model on them. Once the model has learned the patterns, we can use it to make predictions on a previously unseen set of images. If no actors are present in the frame, then make sure to label it too.

So if there are 3 actors.

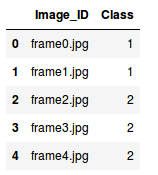
0— Neither Salman Khan or Karan Johar  
1— Karan Johar  
2— Salman Khan

We need to save all the data in a CSV file with the following 2 columns:

* Image\_ID: Contains the name of each image
* Class. Image\_ID: Contains corresponding class for each image

data = pd.read\_csv('mapping.csv') # reading the csv file

data.head() # printing first five rows of the file

****

Our next step is to read the images which we will do based on their names, aka, the Image\_ID column

X = [ ]     # creating an empty array

for img\_name in data.Image\_ID:

    img = plt.imread('' + img\_name)

    X.append(img)  # storing each image in array X

X = np.array(X)    # converting list to array

.

Now we have the images but we need two things to train our model:

* Training images, and
* Their corresponding class

Since there are three classes, we will one hot encode them using the to\_categorical() function of keras.utils.

y = data.Class

dummy\_y = np\_utils.to\_categorical(y) # one hot encoding Classes

We will be using a VGG16 pretrained model which takes an input image of shape (224 X 224 X 3). Since our images are in a different size, we need to reshape all of them. We will use the resize() function of skimage.transform to do this.

image = []

for i in range(0,X.shape[0]):

a = resize(X[i], preserve\_range=True, output\_shape=(224,224)).astype(int) # reshaping to 224\*224\*3

image.append(a)

X = np.array(image)

****

Fig.2. Image Plot

All the images have been reshaped to 224 X 224 X 3. But before passing any input to the model, we must preprocess it as per the model’s requirement. Otherwise, the model will not perform well enough. Use the preprocess\_input() function of keras.applications.vgg16 to perform this step.

from keras.applications.vgg16 import preprocess\_input

X = preprocess\_input(X, mode='tf') # preprocessing the input data

We also need a validation set to check the performance of the model on unseen images. We will make use of the train\_test\_split() function of the sklearn.model\_selection module to randomly divide images into training and validation set.

from sklearn.model\_selection import train\_test\_split

X\_train, X\_valid, y\_train, y\_valid = train\_test\_split(X, dummy\_y, test\_size=0.3, random\_state=42) # preparing the validation set

**Step 3: Building the model**

Now let us start building our model. We will be using VGG16 pre trained model for this task. Let us VGG16 model as base\_model.

base\_model = VGG16(weights='imagenet', include\_top=False, input\_shape=(224, 224, 3)) # include\_top=False to remove the top layer

We will make predictions using this model for X\_train and X\_valid, get the features, and then use those features to retrain the model.

X\_train = base\_model.predict(X\_train)

X\_valid = base\_model.predict(X\_valid)

X\_train.shape, X\_valid.shape

The shape of X\_train and X\_valid is (208, 7, 7, 512), (90, 7, 7, 512) respectively. In order to pass it to our neural network, we have to reshape it to 1-D.

X\_train = X\_train.reshape(208, 7\*7\*512) # converting to 1-D

X\_valid = X\_valid.reshape(90, 7\*7\*512)

We will now preprocess the images and make them zero-centered which helps the model to converge faster.

train = X\_train/X\_train.max() # centering the data

X\_valid = X\_valid/X\_train.max()

Finally, we will build our model. This step can be divided into 3 sub-steps:

1. Building the model
2. Compiling the model
3. Training the model

# i. Building the model

model = Sequential()

model.add(InputLayer((7\*7\*512,))) # input layer

model.add(Dense(units=1024, activation='sigmoid')) # hidden layer

model.add(Dense(3, activation='sigmoid')) # output layer

Let’s check the summary of the model using the summary() function:

model.summary()

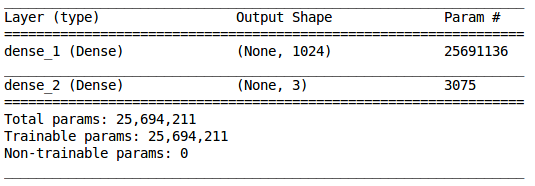


Fig. 3 Summary of our model

We have a hidden layer with 1,024 neurons and an output layer with 3 neurons (since we have 3 classes to predict). Now we will compile our model:

# ii. Compiling the model

model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

In the final step, we will fit the model and simultaneously also check its performance on the unseen images, i.e., validation images:

# iii. Training the model

model.fit(train, y\_train, epochs=100, validation\_data=(X\_valid, y\_valid))

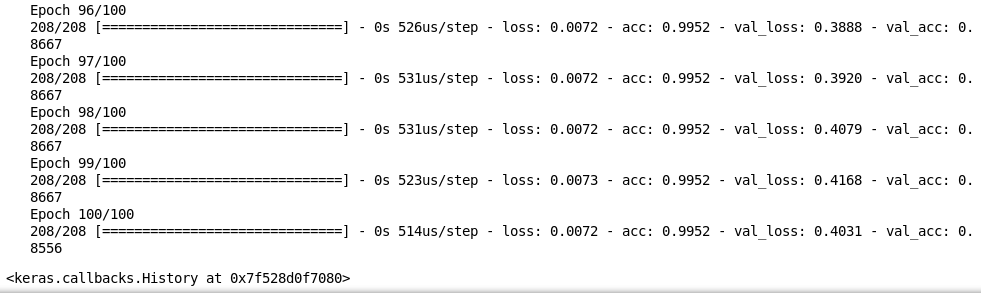


Fig. 4 Epochs

We can see it is performing really well on the training as well as the validation images. We got an accuracy of around 85% on unseen images. And this is how we train a model on video data to get predictions for each frame.

Step 4: Calculating the screen time

Follow the step 1 and extract the frames from the video you want to calculate the screen time.

count = 0

videoFile = "Test.mp4"

cap = cv2.VideoCapture(videoFile)

frameRate = cap.get(5) #frame rate

x=1

while(cap.isOpened()):

frameId = cap.get(1) #current frame number

ret, frame = cap.read()

if (ret != True):

break

if (frameId % math.floor(frameRate) == 0):

filename ="test%d.jpg" % count;count+=1

cv2.imwrite(filename, frame)

cap.release()

print ("Done!")

Next, we will import the images for testing and then reshape them as per the requirements of the aforementioned pretrained model:

test\_image = []

for img\_name in test.Image\_ID:

img = plt.imread('' + img\_name)

test\_image.append(img)

test\_img = np.array(test\_image)

test\_image = []

for i in range(0,test\_img.shape[0]):

a = resize(test\_img[i], preserve\_range=True, output\_shape=(224,224)).astype(int)

test\_image.append(a)

test\_image = np.array(test\_image)

**Step 4: Now let’s make the predictions.**

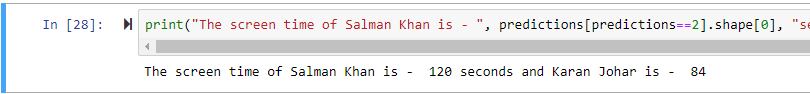
predictions = model.predict\_classes(test\_image)

**Step 5: Calculating the screen time.**

Recall that Class ‘1’ represents the presence of Karan Johar, while Class ‘2’ represents the presence of Salman Khan. We shall make use of the above predictions to calculate the screen time of both these actors:

print("The screen time of Salman Khan is", predictions[predictions==2].shape[0], "seconds")

print("The screen time of Karan Johar is", predictions[predictions==1].shape[0], "seconds")



**3.1.2. FACE RECOGINITION**

We have divided the whole process into 3 steps. The 3 very distinct steps are:

1. Face Detection and Data Gathering
2. Train the Recognizer
3. Face Recognition

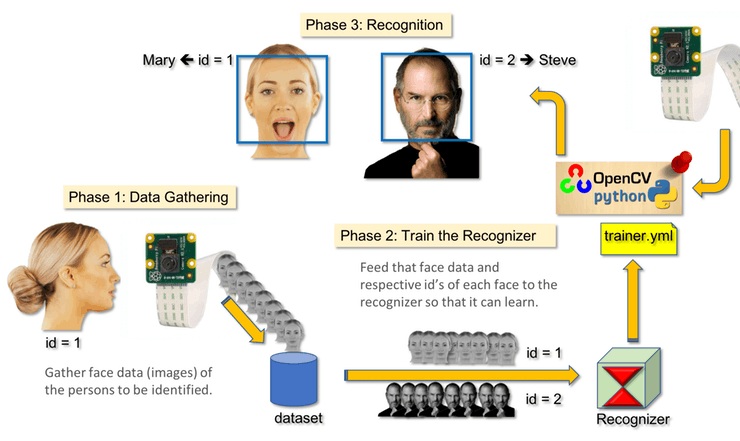


Fig. 5. Represents all the 3 steps

**Step 1a) Face Detection**

The most basic task on Face Recognition is of course, "Face Detecting". Before anything, you must "capture" a face (Step 1) in order to recognize it, when compared with a new face captured on future (Step 3).  
  
We will use Haar Cascade classifier to detect the face.

Object Detection using Haar feature-based cascade classifiers is an effective object detection method proposed by Paul Viola and Michael Jones in their paper, "Rapid Object Detection using a Boosted Cascade of Simple Features" in 2001. It is a machine learning based approach where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images.

Here we will work with face detection. Initially, the algorithm needs a lot of positive images (images of faces) and negative images (images without faces) to train the classifier. Then we need to extract features from it.

import numpy as np

import cv2

faceCascade = cv2.CascadeClassifier('Cascades/haarcascade\_frontalface\_default.xml')

cap = cv2.VideoCapture(0)

cap.set(3,640) # set Width

cap.set(4,480) # set Height

while True:

ret, img = cap.read()

img = cv2.flip(img, -1)

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

faces = faceCascade.detectMultiScale(

gray,

scaleFactor=1.2,

minNeighbors=5,

minSize=(20, 20)

)

for (x,y,w,h) in faces:

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

roi\_gray = gray[y:y+h, x:x+w]

roi\_color = img[y:y+h, x:x+w]

cv2.imshow('video',img)

k = cv2.waitKey(30) & 0xff

if k == 27: # press 'ESC' to quit

break

cap.release()

cv2.destroyAllWindows()

We will set our camera and inside the loop, load our input video in grayscale mode (same we saw before).

Now we must call our classifier function, passing it some very important parameters, as scale factor, number of neighbors and minimum size of the detected face.

faces = faceCascade.detectMultiScale(

gray,

scaleFactor=1.2,

minNeighbors=5,

minSize=(20, 20)

)

Where,

* gray is the input grayscale image.
* scaleFactor is the parameter specifying how much the image size is reduced at each image scale. It is used to create the scale pyramid.
* minNeighbors is a parameter specifying how many neighbors each candidate rectangle should have, to retain it. A higher number gives lower false positives.
* minSize is the minimum rectangle size to be considered a face.

The function will detect faces on the image. Next, we must "mark" the faces in the image, using, for example, a blue rectangle. This is done with this portion of the code:

for (x,y,w,h) in faces:

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

roi\_gray = gray[y:y+h, x:x+w]

roi\_color = img[y:y+h, x:x+w]

If faces are found, it returns the positions of detected faces as a rectangle with the left up corner (x,y) and having "w" as its Width and "h" as its Height ==> (x,y,w,h). Please see the picture.

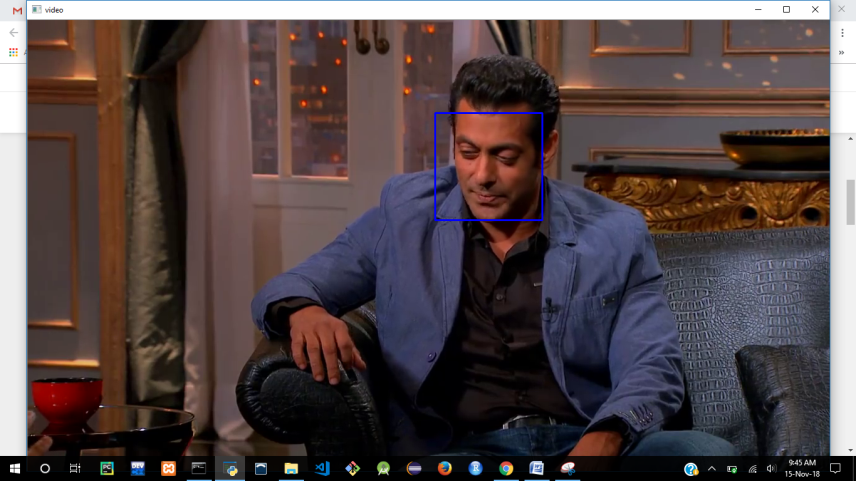


Fig. 6.Image Plot with face recognized

**Step 1b) Data Gathering**

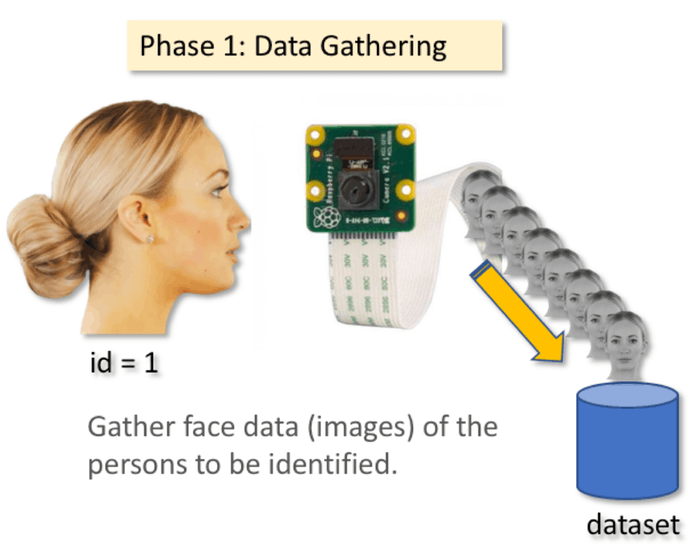


Fig. 7 Data gathering

The following code will store all the images in a sub directory created by you.

**import cv2**

**import os**

**cam = cv2.VideoCapture(0)**

**cam.set(3, 640) # set video width**

**cam.set(4, 480) # set video height**

**face\_detector=cv2.CascadeClassifier('haarcascade\_frontalface\_default.xml')**

**# For each person, enter one numeric face id**

**face\_id = input('\n enter user id end press <return> ==> ')**

**print("\n [INFO] Initializing face capture. Look the camera and wait ...")**

**count = 0**

**while(True):**

**ret, img = cam.read()**

**img = cv2.flip(img, -1) # flip video image vertically**

**gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)**

**faces = face\_detector.detectMultiScale(gray, 1.3, 5)**

**for (x,y,w,h) in faces:**

**cv2.rectangle(img, (x,y), (x+w,y+h), (255,0,0), 2)**

**count += 1**

**# Save the captured image into the datasets folder**

**cv2.imwrite("dataset/User." + str(face\_id) + '.' + str(count) + ".jpg", gray[y:y+h,x:x+w])**

**cv2.imshow('image', img)**

**k = cv2.waitKey(100) & 0xff # Press 'ESC' for exiting video**

**if k == 27:**

**break**

**elif count >= 30: # Take 30 face sample and stop video**

**break**

**print("\n [INFO] Exiting Program and cleanup stuff")**

**cam.release()**

**cv2.destroyAllWindows()**

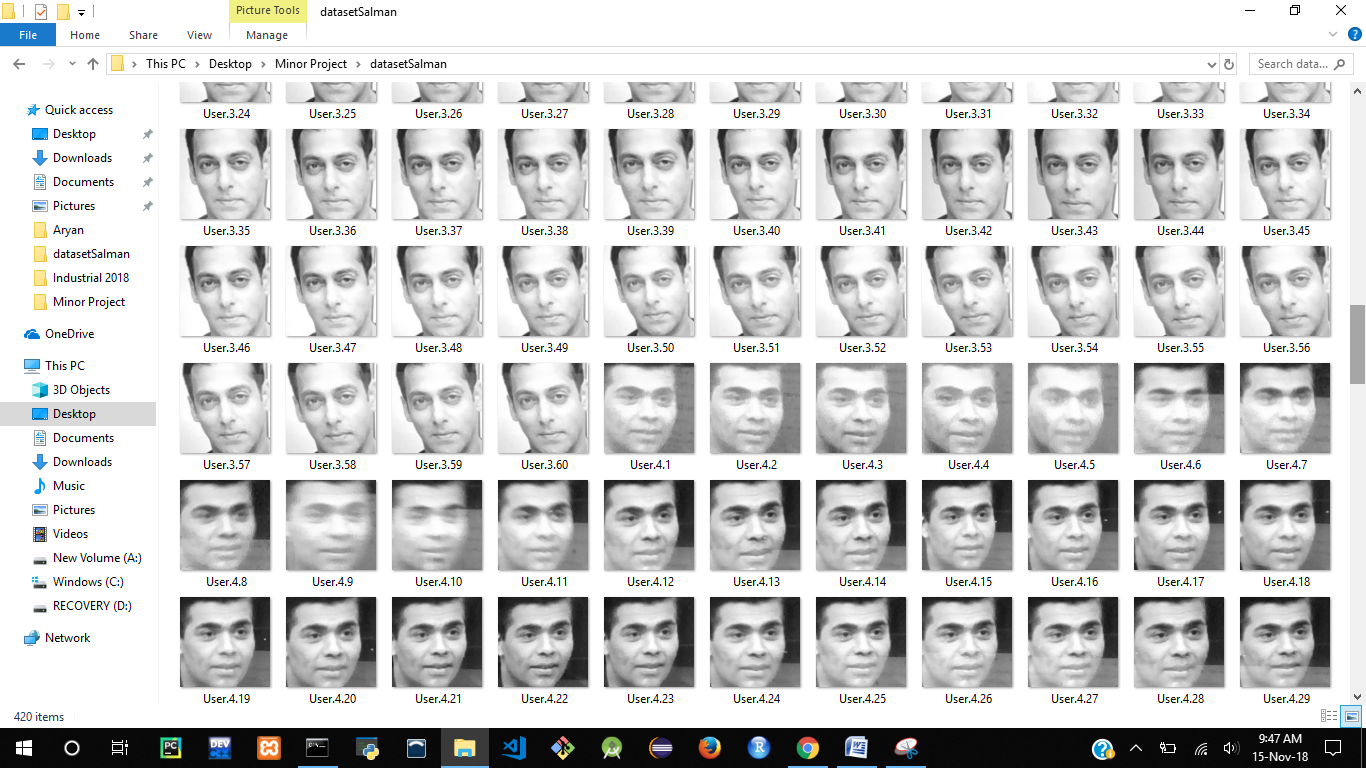


Fig. 8 Images to train

The code is very similar to the code that we saw for face detection. What we added, was an "input command" to capture a user id, that should be an integer number (1, 2, 3, etc)

**face\_id = input('\n enter user id end press ==> ')**

**Step 2) Trainer**

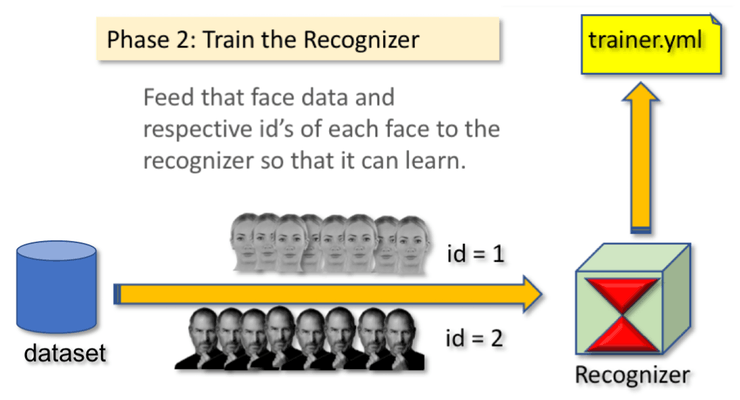


Fig. 9 Trainer

To store the data in the trainer

**import cv2**

**import numpy as np**

**from PIL import Image**

**import os**

**# Path for face image database**

**path = 'dataset'**

**recognizer = cv2.face.LBPHFaceRecognizer\_create()**

**detector = cv2.CascadeClassifier("haarcascade\_frontalface\_default.xml");**

**# function to get the images and label data**

**def getImagesAndLabels(path):**

**imagePaths = [os.path.join(path,f) for f in os.listdir(path)]**

**faceSamples=[]**

**ids = []**

**for imagePath in imagePaths:**

**PIL\_img = Image.open(imagePath).convert('L') # convert it to grayscale**

**img\_numpy = np.array(PIL\_img,'uint8')**

**id = int(os.path.split(imagePath)[-1].split(".")[1])**

**faces = detector.detectMultiScale(img\_numpy)**

**for (x,y,w,h) in faces:**

**faceSamples.append(img\_numpy[y:y+h,x:x+w])**

**ids.append(id)**

**return faceSamples,ids**

**print ("\n [INFO] Training faces. It will take a few seconds. Wait ...")**

**faces,ids = getImagesAndLabels(path)**

**recognizer.train(faces, np.array(ids))**

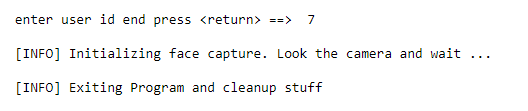
**# Save the model into trainer/trainer.yml**

**recognizer.write('trainer/trainer.yml') # recognizer.save() worked on Mac, but not on Pi**

**# Print the numer of faces trained and end program**

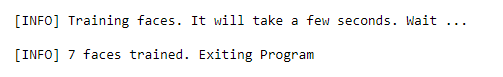
**print("\n [INFO] {0} faces trained. Exiting Program".format(len(np.unique(ids))))**

Figure 5. ROC curve



After getting the data inside the trainer, we will use a recognizer, the LBPH (LOCAL BINARY PATTERNS HISTOGRAMS) Face Recognizer, included on OpenCV package. We do this in the following line:

**recognizer = cv2.face.LBPHFaceRecognizer\_create()**



Now that trainer setup is done we move to the next step and recognize some videos.

**Step 3) Recognition**

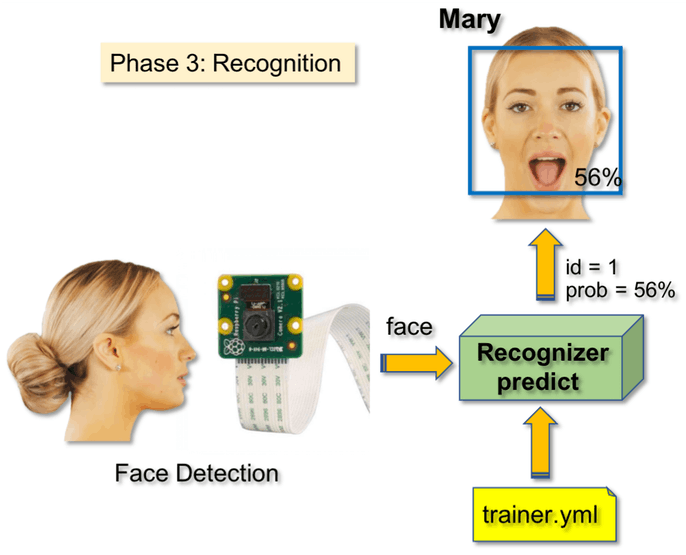


Fig. 10 Recoginition

Now we use the following to recognise the video.

**import cv2**

**import numpy as np**

**import os**

**recognizer = cv2.face.LBPHFaceRecognizer\_create()**

**recognizer.read('trainersalman/trainer.yml')**

**cascadePath = "haarcascade\_frontalface\_default.xml"**

**faceCascade = cv2.CascadeClassifier(cascadePath);**

**font = cv2.FONT\_HERSHEY\_SIMPLEX**

**#iniciate id counter**

**id = 0**

**# names related to ids: example ==> Marcelo: id=1, etc**

**names = ['None','Salman','Salman','Salman','Karan','Karan','Karan','Karan']**

**# Initialize and start realtime video capture**

**videoFile = "Salman Khan Rapid Fire Round.mp4"**

**cam = cv2.VideoCapture(videoFile)**

**cam.set(3, 640) # set video widht**

**cam.set(4, 480) # set video height**

**# Define min window size to be recognized as a face**

**minW = 0.1\*cam.get(3)**

**minH = 0.1\*cam.get(4)**

**while True:**

**ret, img =cam.read()**

**#img = cv2.flip(img, -1) # Flip vertically**

**gray = cv2.cvtColor(img,cv2.COLOR\_BGR2GRAY)**

**faces = faceCascade.detectMultiScale(**

**gray,**

**scaleFactor = 1.2,**

**minNeighbors = 5,**

**minSize = (int(minW), int(minH)),**

**)**

**for(x,y,w,h) in faces:**

**cv2.rectangle(img, (x,y), (x+w,y+h), (0,255,0), 2)**

**id, confidence = recognizer.predict(gray[y:y+h,x:x+w])**

**# Check if confidence is less them 100 ==> "0" is perfect match**

**if (confidence < 100):**

**id = names[id]**

**confidence = " {0}%".format(round(100 - confidence))**

**else:**

**id = "unknown"**

**confidence = " {0}%".format(round(100 - confidence))**

**cv2.putText(img, str(id), (x+5,y-5), font, 1, (255,255,255), 2)**

**cv2.putText(img, str(confidence), (x+5,y+h-5), font, 1, (255,255,0), 1)**

**cv2.imshow('camera',img)**

**k = cv2.waitKey(10) & 0xff # Press 'ESC' for exiting video**

**if k == 27:**

**break**

**# Do a bit of cleanup**

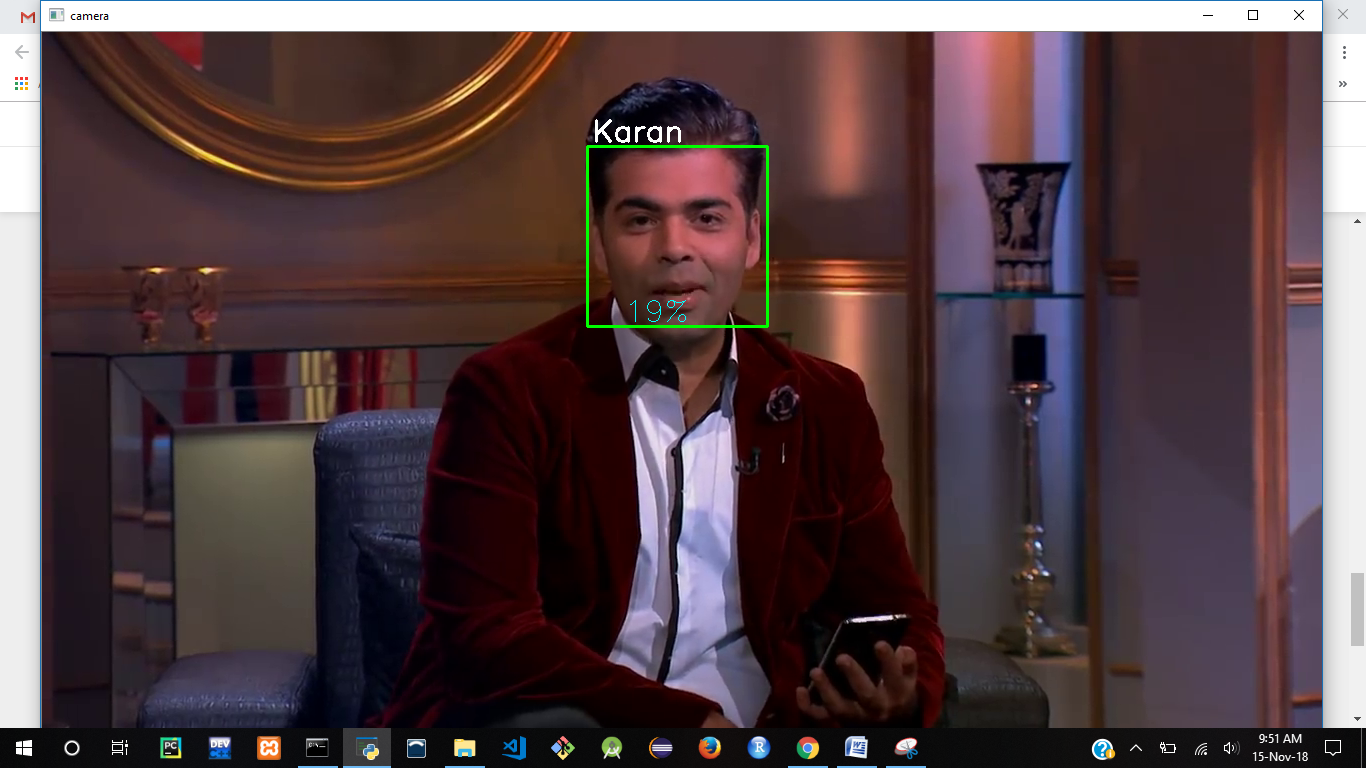
**print("\n [INFO] Exiting Program and cleanup stuff")**

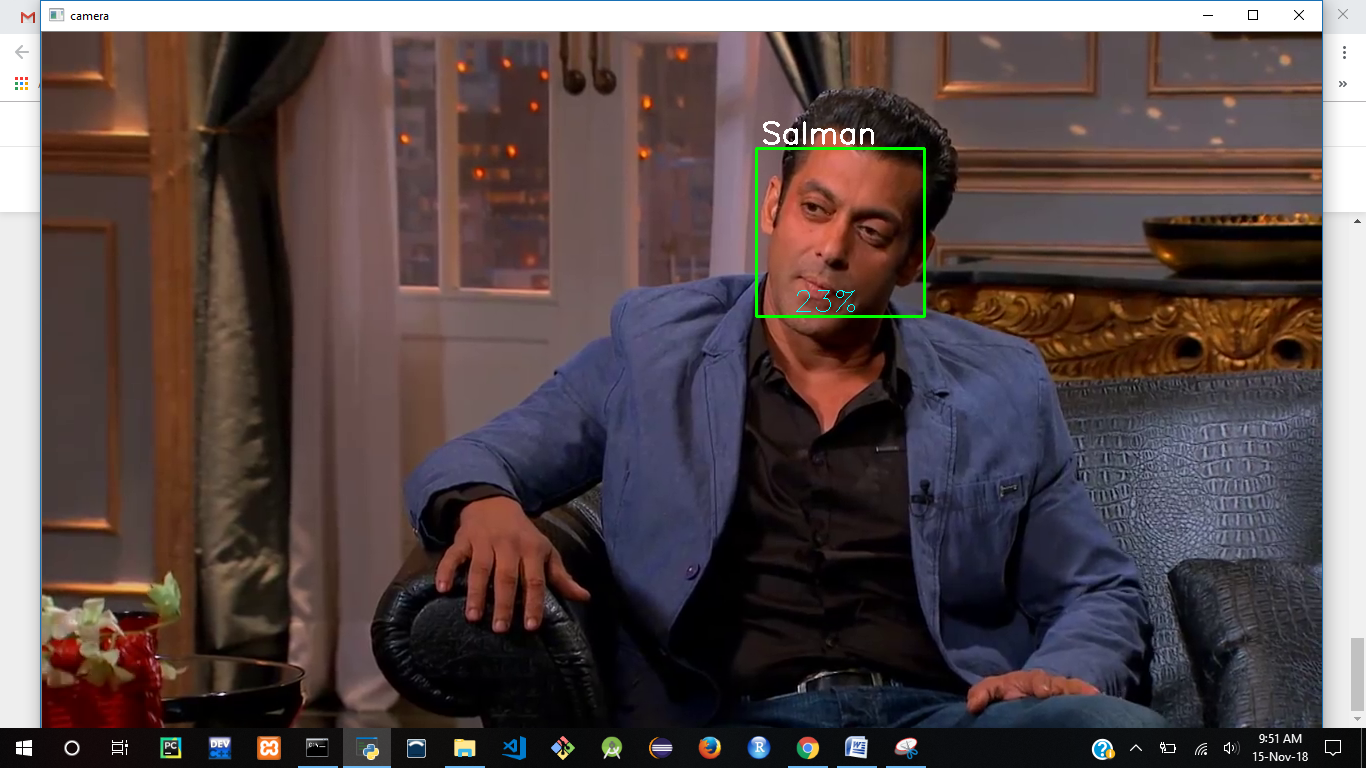
**cam.release()**

**cv2.destroyAllWindows()print("\n [INFO] Exiting Program and cleanup stuff")**

**cam.release()**

**cv2.destroyAllWindows()**





The recognizer.predict (), will take as a parameter a captured portion of the face to be analyzed and will return its probable owner, indicating its id and how much confidence the recognizer is in relation with this match.

**CHAPTER FOUR**

**THESIS**

**4.1 NEURAL NETWORKS FUNDAMENTALS**

Classical neural networks are a type of supervised machine learning deep learning popularity is instead due to the fact that modern deep neural networks can be used in unsupervised learning tasks as well. We will mainly concentrate on classical feed-forward networks that work in a supervised way. Our first question is, what exactly is a neural network? Probably the best way to interpret a neural network is to describe it as a mathematical model for information processing. A neural net is not a fixed program, but rather a model, a system that processes information, or inputs, in a somewhat bland analogy to how information is thought to be processed by biological entities. We can identify three main characteristics for a neural net:

• **The neural net architecture**: This describes the set of connections (feed-forward,recurrent, multi- or single-layered, and so on) between the neurons,the number of layers, and the number of neurons in each layer.

• **The learning**: This describes what is commonly defined as the training.

Whether we use back-propagation or some kind of energy level training, itidentifies how we determine the weights between neurons.

• **The activity function**: This describes the function we use on the activationvalue that is passed onto each neuron, the neuron's internal state, and itdescribes how the neuron works (stochastically, linearly, and so on) and under what conditions.

It should be noted, however, that some researchers would consider the activity function as part of the architecture; it may be easier, however, for a beginner to separate these two aspects for now. It needs to be remarked that artificial neural nets represent only an approximation of how a biological brain works. A biological neural net is a much more complex model; however, this should not be a concern. Artificial neural nets can still perform many useful tasks, in fact, as we will show later, an artificial neural net can indeed approximate to any degree we wish any function of the input onto the output.

The development of neural nets is based on the following assumptions:

• Information processing occurs, in its simplest form, over simple elements, called neurons

• Neurons are connected and exchange signals between them along connection links

• Connection links between neurons can be stronger or weaker, and this determines how information is processed

• Each neuron has an internal state that is determined by all the incoming connections from other neurons

• Each neuron has a different activity function that is calculated on the neuron internal state and determines its output signal

The first example of a neural network was called the perceptron, which was invented by Frank Rosenblatt in 1957. The perceptron is a network comprised of only an input and an output layer. In case of binary classifications, the output layer has only one neuron or unit. The perceptron seemed to be very promising from the start, though it was quickly realized that it could only learn linearly separable patterns. For example,

Marvin Minsky and Seymour Papert showed that it could not learn the XOR logical function. In its most basic representations, perceptrons are just simple representations of one neuron and its input, input that can be comprised of several neurons.

Given different inputs into a neuron, we define an activation value by the formula, where *xi* is the value for the input neuron, while *wi*is the value of the connection between the neuron *i*and the output. We should notice that perceptrons share many similarities with logistic regression algorithms, and are constrained by linear classifiers as well. If the activation value, which should be thought of as the neuron internal state, is greater than a fixed threshold *b*, then the neuron will activate, that is, it will fire, otherwise it will not.



Fig.11.A simple perceptron with three input units (neurons) and one output unit (neuron)

The simple activation defined above can be interpreted as the dot product between the vector w and the vector x. The vector w is fixed, and it defines how the perceptron works, while x represents the input. A vector x is perpendicular to the weight vector w if *<*w,x*> = 0*, therefore all vectors x such that *<*w,x*> = 0* define a hyper-plane in R*3* (where *3* is the dimension of x, but it could be any integer in general). Hence, any vector x satisfying *<*w,x*>> 0* is a vector on the side of the hyper-plane defined by w. This makes it clear how a perceptron just defines a hyper-plane and it works as a classifier. However, rather than keeping track of this value, generally we include a bias unit in our network, which is an always on (*value = 1)* special neuron with connecting weight *-b*. In this case, if the connecting weight has value *–b*, the activation value becomes *a(x) = ∑iwixi* and setting *a(x) > 0* is equivalent to setting *∑iwixi*>*b* .



Fig.12. A perceptron with added a bias unit for the output vector. Bias units are always on.

**4.2 DEEP LEARNING FUNDAMENTALS**

In this section we will introduce deep learning and deep neural networks, that is, neural networks with at least two or more hidden layers. The question may arise what is the point of using more than one hidden layer, given the Universal Approximation Theorem, and this is in no way a naïve question, since for a long period the neural networks used were very shallow, with just one hidden layer.

The answer is that it is true that 2-layer neural networks can approximate any continuous function to any degree, however, it is also true that adding layers adds levels of complexity that may be much harder and may require many more neurons to simulate with shallow networks. There is also another, more important, reason behind the term *deep* of deep learning that refers not just to the depth of the network, or how many layers the neural net has, but to the level of "learning".

In deep learning, the network does not simply learn to predict an output *Y* given an input *X*, but it also understands basic features of the input. In deep learning, the neural network is able to make abstractions of the features that comprise the input examples, to understand the basic characteristics of the examples, and to make predictions based on those characteristics. In deep learning, there is a level of abstraction that is missing in other basic machine learning algorithms or in shallow neural networks.

**4.2.1 Deep Learning Algorithms**

In this class of algorithms, we can generally include:

• **Multi-Layer Perceptrons (MLP)**: A neural network with many hidden layers, with feed-forward propagation. As discussed, this is one of the first examples of deep learning network but not the only possible one.

• **Autoencoders**: A class of unsupervised learning algorithms in which the output shape is the same as the input, that allows the network to better learn basic representations.

• **Convolutional Neural Networks (CNN)**: Convolutional layers apply filters to the input image (or sound) by sliding this filter all across the incoming signal to produce a bi-dimensional activation map. CNNs allow the enhancement of features hidden in the input.

**4.3 LOCAL BINARY PATTERNS HISTOGRAM**

Eigenfaces and Fisherfaces take a somewhat holistic approach to face recognition. You treat your data as a vector somewhere in a high-dimensional image space. We all know high-dimensionality is bad, so a lower-dimensional subspace is identified, where (probably) useful information is preserved. The Eigenfaces approach maximizes the total scatter, which can lead to problems if the variance is generated by an external source, because components with a maximum variance over all classes aren’t necessarily useful for classification. So to preserve some discriminative information we applied a Linear Discriminant Analysis and optimized as described in the Fisherfaces method. The Fisherfaces method worked great... at least for the constrained scenario we’ve assumed in our model.

Now real life isn’t perfect. You simply can’t guarantee perfect light settings in your images or 10 different images of a person. So what if there’s only one image for each person? Our covariance estimates for the subspace *may* be horribly wrong, so will the recognition. Remember the Eigenfaces method had a 96% recognition rate on the AT&T Facedatabase? How many images do we actually need to get such useful estimates? Here are the Rank-1 recognition rates of the Eigenfaces and Fisherfaces method on the AT&T Facedatabase, which is a fairly easy image database:

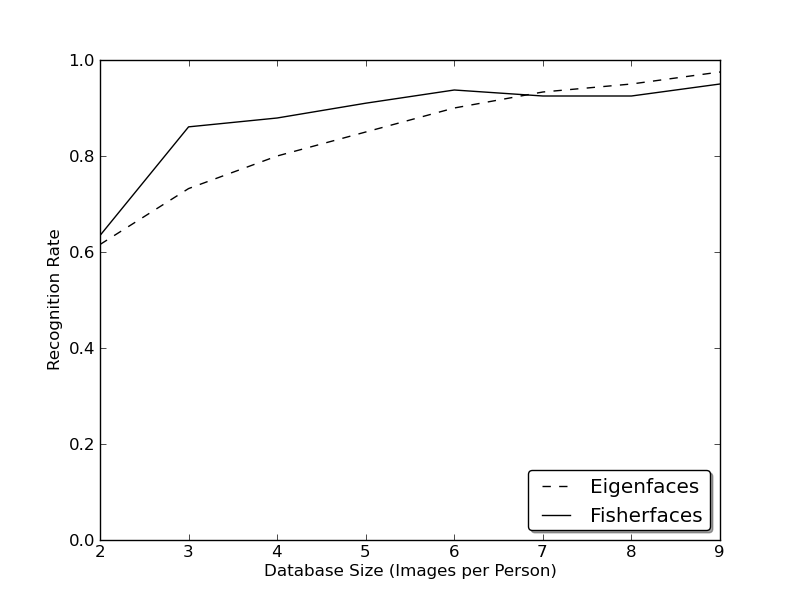


Fig.13 Graph (Images vs Recognition Rate)

So in order to get good recognition rates you’ll need at least 8(+-1) images for each person and the Fisherfaces method doesn’t really help here. The above experiment is a 10-fold cross validated result carried out with the facerec framework at. This is not a publication, so I won’t back these figures with a deep mathematical analysis.

So some research concentrated on extracting local features from images. The idea is to not look at the whole image as a high-dimensional vector, but describe only local features of an object. The features you extract this way will have a low-dimensionality implicitly. A fine idea! But you’ll soon observe the image representation we are given doesn’t only suffer from illumination variations. Think of things like scale, translation or rotation in images - your local description has to be at least a bit robust against those things. Just like SIFT, the Local Binary Patterns methodology has its roots in 2D texture analysis. The basic idea of Local Binary Patterns is to summarize the local structure in an image by comparing each pixel with its neighborhood. Take a pixel as center and threshold its neighbors against. If the intensity of the center pixel is greater-equal its neighbor, then denote it with 1 and 0 if not. You’ll end up with a binary number for each pixel, just like 11001111. So with 8 surrounding pixels you’ll end up with 2^8 possible combinations, called Local Binary Patterns or sometimes referred to as LBP codes. The first LBP operator described in literature actually used a fixed 3 x 3 neighborhood just like this:



Fig.14. LBP Detector

**4.4 ALGORITHMIC DESCRIPTION**

A more formal description of the LBP operator can be given as:

LBP(x_c, y_c) = \sum_{p=0}^{P-1} 2^p s(i_p - i_c)

, with (x_c, y_c) as central pixel with intensity i_c; and i_n being the intensity of the the neighbor pixel. s is the sign function defined as:

\begin{equation}
s(x) =
\begin{cases}
1 & \text{if $x \geq 0$}\\
0 & \text{else}
\end{cases}
\end{equation}

This description enables you to capture very fine grained details in images. In fact the authors were able to compete with state of the art results for texture classification. Soon after the operator was published it was noted, that a fixed neighborhood fails to encode details differing in scale. So the operator was extended to use a variable neighborhood in [AHP04]. The idea is to align an arbitrary number of neighbors on a circle with a variable radius, which enables to capture the following neighborhoods:

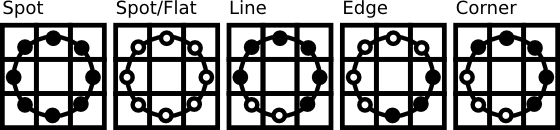


Fig.15 Neighbors with variable radius

For a given Point (x_c,y_c) the position of the neighbor (x_p,y_p), p \in P can be calculated by:

\begin{align*}
x_{p} & = & x_c + R \cos({\frac{2\pi p}{P}})\\
y_{p} & = & y_c - R \sin({\frac{2\pi p}{P}})
\end{align*}

Where R is the radius of the circle and P is the number of sample points.

The operator is an extension to the original LBP codes, so it’s sometimes called Extended LBP (also referred to as Circular LBP) . If a points coordinate on the circle doesn’t correspond to image coordinates, the point get’s interpolated. Computer science has a bunch of clever interpolation schemes, the OpenCV implementation does a bilinear interpolation:

\begin{align*}
f(x,y) \approx \begin{bmatrix}
    1-x & x \end{bmatrix} \begin{bmatrix}
    f(0,0) & f(0,1) \\
    f(1,0) & f(1,1) \end{bmatrix} \begin{bmatrix}
    1-y \\
    y \end{bmatrix}.
\end{align*}

By definition the LBP operator is robust against monotonic gray scale transformations. We can easily verify this by looking at the LBP image of an artificially modified image (so you see what an LBP image looks like!):

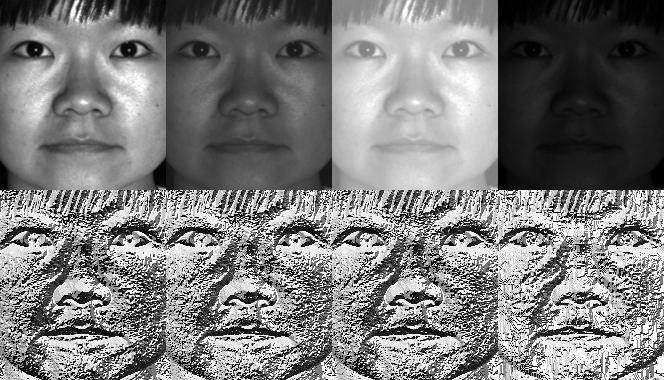


Fig.14 LBPH Image

So what’s left to do is how to incorporate the spatial information in the face recognition model. The representation proposed by Ahonen et. al [AHP04] is to divide the LBP image into m local regions and extract a histogram from each. The spatially enhanced feature vector is then obtained by concatenating the local histograms (**not merging them**). These histograms are called Local Binary Patterns Histograms.

## 4.5 KERAS

Keras is a high-level neural networks API, written in Python and capable of running on top of [TensorFlow](https://github.com/tensorflow/tensorflow), [CNTK](https://github.com/Microsoft/cntk), or [Theano](https://github.com/Theano/Theano). It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.

Use Keras if you need a deep learning library that:

* Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).
* Supports both convolutional networks and recurrent networks, as well as combinations of the two.
* Runs seamlessly on CPU and GPU.

## 4.6 GETTING STARTED WITH KERAS

The core data structure of Keras is a **model**, a way to organize layers. The simplest type of model is the [Sequential](https://keras.io/getting-started/sequential-model-guide) model, a linear stack of layers. For more complex architectures, you should use the [Keras functional API](https://keras.io/getting-started/functional-api-guide), which allows to build arbitrary graphs of layers.

Here is the Sequential model:

**from** keras.models **import** Sequential

model = Sequential()

Stacking layers is as easy as .add():

**from** keras.layers **import** Dense

model.add(Dense(units=64, activation='relu', input\_dim=100))

model.add(Dense(units=10, activation='softmax'))

Once your model looks good, configure its learning process with .compile():

model.compile(loss='categorical\_crossentropy', optimizer='sgd', metrics=['accuracy'])

If you need to, you can further configure your optimizer. A core principle of Keras is to make things reasonably simple, while allowing the user to be fully in control when they need to (the ultimate control being the easy extensibility of the source code).

model.compile(loss=keras.losses.categorical\_crossentropy,

optimizer=keras.optimizers.SGD(lr=0.01, momentum=0.9, nesterov=**True**))

You can now iterate on your training data in batches:

*# x\_train and y\_train are Numpy arrays --just like in the Scikit-Learn API.*

model.fit(x\_train, y\_train, epochs=5, batch\_size=32)

Alternatively, you can feed batches to your model manually:

model.train\_on\_batch(x\_batch, y\_batch)

Evaluate your performance in one line:

loss\_and\_metrics = model.evaluate(x\_test, y\_test, batch\_size=128)

Or generate predictions on new data:

classes = model.predict(x\_test, batch\_size=128)

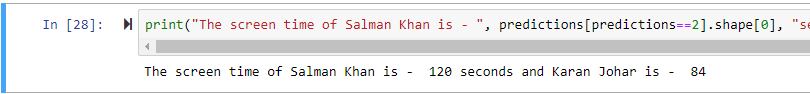
Building a question answering system, an image classification model, a Neural Turing Machine, or any other model is just as fast. The ideas behind deep learning are simple, so why should their implementation be painful?

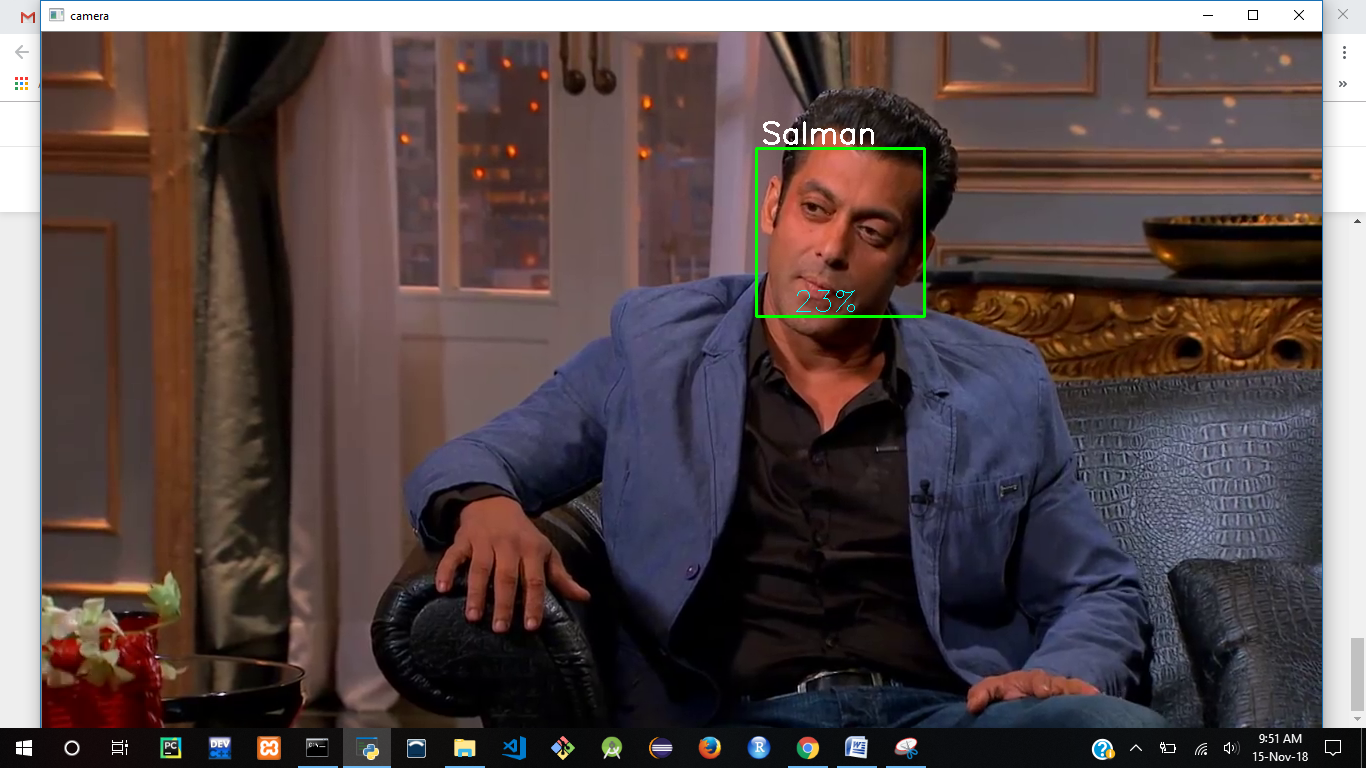
**CONCLUSION**

Such models can help us in various fields:

* We can calculate the screen time of a particular actor in a movie
* Calculate the screen time of your favorite superhero, etc.

These are just a few examples where this technique can be used. We can come up with many more such applications.





### Facial recognition is being used in many businesses:

1. Secure Payments
2. Healthcare
3. Camera Surveillance
4. Criminal Identification
5. Advertising

# FUTURE SCOPE

Face recognition systems used today work very well under constrained conditions, although all systems work much better with frontal mug-shot images and constant lighting. All current face recognition algorithms fail under the vastly varying conditions under which humans need to and are able to identify other people. Next generation person recognition systems will need to recognize people in real-time and in much less constrained situations.

We believe that identification systems that are robust in natural environments, in the presence of noise and illumination changes, cannot rely on a single modality, so that fusion with other modalities is essential. Technology used in smart environments has to be unobtrusive and allow users to act freely. Wearable systems in particular require their sensing technology to be small, low powered and easily integrable with the user's clothing. Considering all the requirements, identification systems that use face recognition and speaker identification seem to us to have the most potential for wide-spread application.

Cameras and microphones today are very small, light-weight and have been successfully integrated with wearable systems. Audio and video based recognition systems have the critical advantage that they use the modalities humans use for recognition. Finally, researchers are beginning to demonstrate that unobtrusive audio-and-video based person identification systems can achieve high recognition rates without requiring the user to be in highly controlled environments.

**REFERNCES**

[1] https://docs.opencv.org/2.4/modules/contrib/doc/facerec/facerec\_tutorial.html

[2] Valentino Zocca, GianmarioSpacagna, Daniel Slater, Peter Roelants. Python Deep Learning. Packt Publishing 2017

[3] Nikhil Buduma, Nicholas Locascio. Fundamentals of Deep Learning. O’Reilly Media, Inc: 2017  
  
[4] https://en.wikipedia.org/wiki/Facial\_recognition\_system

[5] https://keras.io/models/sequential/

**APPENDIX A**

**SOURCE CODE**