Laboratorio Nº 6- Detección de Rostros			
Asignatura	Tópicos Especiales - Visión Artificial	**Código**	0756
Profesor	José Carlos Rangel Ortiz		

Introducción

En este laboratorio se mostraran lo diferentes algoritmos disponibles en OpenCV para detectar bordes, buscar correspondencias entre imágenes a través de descriptores detectar marcadores, rostros y códigos QR. De igual manera mediante el uso de estos temas se abordará un introducción a los procedimientos de Realidad Aumentada con OpenCV y el uso de Clasificadores en cascada con características de Haar.

Adicional a este documento se facilita una carpeta con los códigos y algunas imágenes y archivos adicionales utilizados en el desarrollo del laboratorio.

La mayor parte de este documento se basa en los ejemplos presentados en el libro : *Mastering OpenCv with Python* de Alberto Fernández Villán [1].

Fundamentos Teóricos

La Visión Artificial busca desde sus inicios imitar el comportamiento de la visión humana. Para ello emplea un conjunto diverso de técnicas que se enfocan en extraer información de imágenes de entrada y así facilitar a las computadoras realizar operaciones como lo hacen los seres humanos.

En Visión Artificial la mayor parte de los algoritmos se basan en investigaciones y descubrimientos anteriores, por lo cual, una gran cantidad de los métodos existentes requieren un pre-procesamiento para poder funcionar. Dentro de estas operaciones podemos mencionar las detecciones tanto de puntos, bordes, líneas, características, rostros o partes del cuerpo. Como podemos ver la detección se puede hacer a diferentes niveles y para cada uno de estos existirán un compendio de técnicas especializadas para lograr su fin.

Adicionalmente, debido a los grandes avances actuales en la Visión Artificial, en los últimos años ha surgido el enfoque de ir más allá de la emulación del humano, sino que se enfoca en enriquecer la percepción visual humana a través de la creación de estructuras virtuales que permitan una interacción del humano con elementos que en realidad no existen en el mundo real.

Como parte de los enfoques de realidad aumentada se pueden incluir también la detección facial con sujetos reales y en la mayoría de los casos en tiempo real. Estos emplean clasificadores en cascada u otros algoritmos de machine learning y en base a la detección se pueden añadir modificaciones a las caras de estas personas, emulando los tan conocidos filtros disponibles en muchas aplicaciones de redes sociales.

Objetivo

Aplicar algoritmos de detección de rostros

Entregables

Al final de cada sección del laboratorio se indicará cuales son acciones que deberá cumplir el estudiante y los cuales serán evaluados en la actividad. El informe final debe contener todo lo solicitado y se debe redactar utilizando la Guía de Estilos para Memorias y Laboratorios Disponible en Moodle. El informe debe ser entregado en formato PDF y siguiendo las indicaciones para nombrar el archivo.

1. Códigos QR

1.1. Detección de Códigos QR

Como parte final del laboratorio se presenta una aplicación que mediante OpenCV permite detectar e interpretar los códigos QR en una imagen. Se utiliza en este caso el código `01_qr_code_scanner.py`. Este ejemplo recibe una imagen leída desde un archivo y la procesa para determinar si contiene un código QR en su interior. Puede utilizar cualquier imagen que contenga un código QR. Reemplace la ruta de la imagen en la línea de código pertinente para que el programa pueda leer su imagen.

1. Como primer paso se crea una instancia del detector de QR de OpenCV

```
# Crear el detector de QR
qr_code_detector = cv2.QRCodeDetector()
```

2. Utilizando la función `detectAndDecode()`, se envía la imagen a procesar.

```
# Detectar y decodificar el código QR
data, vertices, rectified_qr_code_binarized = qr_code_detector.detectAndDecode(image)
```

3. Esta función devuelve los siguientes elementos
   - Un arreglo con los vértices del código encontrado.
   - El código QR rectificado
   - La información contenida en el código QR
4. Si utiliza la consola, para ejecutar se debe escribir el siguiente comando :

```
python 01_qr_code_scanner.py
```

5. Luego de su ejecución nuestra imagen se verá de la siguiente manera (figura 8):

1.2. Generación de Códigos QR

Usando Python también es posible generar nuestros propios códigos QR. En esta sección se usara el código `02_qr_code_generar.py`. Con este podemos generar un Código QR que contenga la información que nosotros definamos.

En este caso se necesitan las librerías PyQrCode y PyPNG, por lo cual se deben instalar antes de ejecutar el código usando el siguiente comando en una consola.

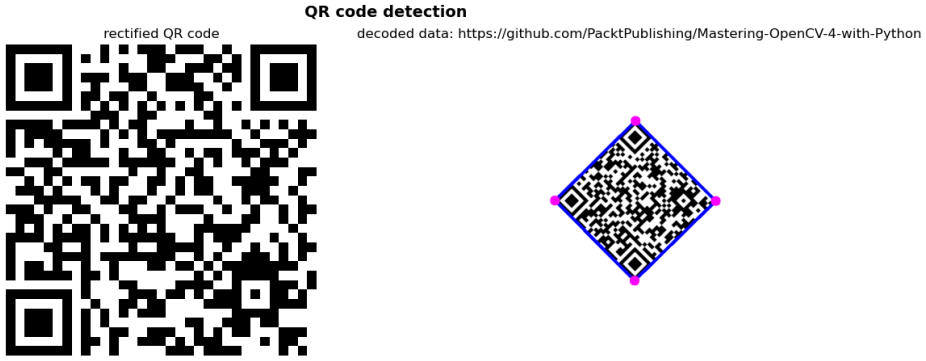

Figura 1: Resultado de la detección del código QR.

```
pip install pyqrcode pypng
```

1. El código recibe una cadena de contenido la cual será agregada al QR generado.

```
link= input("Ingrese la información que desea incluir en el QR: ")
```

2. Posteriormente el siguiente fragmento crea el QR y guarda este en nuestra carpeta.

```
# Generar Código QR
qr_code =pyqrcode.create(link)
qr_code.png("QRCode.png", scale=5)
```

3. Si utiliza la consola, para ejecutar se debe escribir el siguiente comando :

```
python 02_qr_code_generar.py
```

Entregables para la Sección

1. Para esta sección no es necesario realizar ningún entregable.

2. Detección de Rostros y Realidad Aumentada

Dentro de los procedimientos de reconocimiento uno de los más comunes en la actualidad es el reconocimiento facial, el cual es aplicado en temas de seguridad, acceso, control biométrico, superposición de rostros, realidad aumentada, aplicación de filtros, entre otros.

En esta ocasión se creará utilizará una aplicación que en tiempo real detectará el rostro de una persona y colocará una imagen en una posición que se calculará según los elementos detectados en la cara del sujeto, similar a la aplicación de filtros de aplicaciones de mensajería y redes sociales como Snapchat, Facebook o Instagram. Este mismo principio es utilizado en aplicaciones como Deepface y aplicaciones de envejecimiento de rostros.

Para entrenar estos clasificadores se hace necesario contar con cierta cantidad de muestras positivas (objetos, caras, etc) y negativas. A estas se les extraen sus características y posteriormente se realiza un proceso de entrenamiento. Los clasificadores se basan en reconocer patrones presentes en las características o *features* detectadas en las imágenes. En este caso los detectores de características usados serán los filtros Haar.

Para los siguientes ejemplos utilizaremos los conceptos de Cascada de Clasificadores y los filtros de Haar. El método de cascada de clasificadores propuesto en el artículo [3] consiste en la aplicación sucesiva de varios clasificadores a una imagen. En este caso la imagen anteriormente ha sido analizada a través de sub-regiones de menor tamaño, cada una de estas regiones es evaluada por un clasificador y si el clasificador determina que una de esas sub-regiones corresponde con una cara (o cualquier objeto buscado), se envía dicha sub-región al siguiente nivel o clasificador, junto con todas las demás sub-regiones identificadas positivamente. Al final el resultado o la categoría determinada por la cascada se basa en la premisa de que esa sub-región ha sido identificada positivamente por todos los clasificadores dentro de la cascada. Este proceso se puede ver en la figura 2. Para más información se puede consultar el siguiente enlace[1].

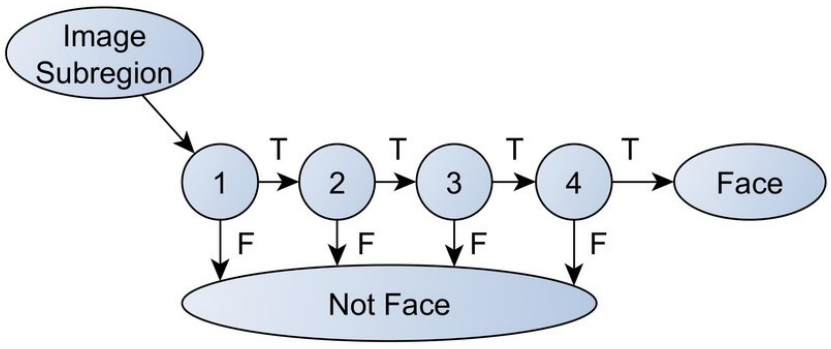

Figura 2: Esquema de una cascada de clasificadores.

[1] Cascada de Clasificadores

OpenCV posee entonces el clasificador HaarCascade[2]. Estos clasificadores permiten reconocer diferentes partes del cuerpo de humanos y animales y algunos objetos. Estos son modelos pre-entrenados y se encuentran realizando una búsqueda en las páginas oficiales y de igual manera realizados por terceras personas. En este caso se han incluido dentro de los archivos del laboratorio ya que se utilizarán el detector de caras, nariz y ojos.

En este caso utilizaremos los códigos `01_snapchat_augmeted_reality_glasses.py` y `02_snapchat_augmeted_reality_moustache.py`. Los cuales agregan unas gafas y un bigote respectivamente a la imagen de entrada, siempre que se detecte un rostro en la misma.

A continuación se presenta la explicación de algunos fragmentos de códigos que se utilizan en estos códigos.

1. Iniciaremos con el código `01_snapchat_augmeted_reality_glasses.py`. Utilizando la cámara web para añadir unas gafas a la imagen de entrada.

2. El fragmento de código siguiente carga los modelos de clasificación de harrcascade para la detección de rostros y los ojos.

```
# Cargar clasificadores para la detección de rostro y ojos
face_cascade = cv2.CascadeClassifier("haarcascade_frontalface_default.xml")
eyepair_cascade = cv2.CascadeClassifier("haarcascade_mcs_eyepair_big.xml")
```

3. Como siguiente paso, las gafas se corresponden con una imagen que debe ser cargada en el código fuente. En este código utiliza el -1 como argumento al leer la imagen para indicar que se debe leer el canal Alfa(transparencia) si la imagen cuenta con uno.

```
# Carga la imagen de las gafas.
# El argumento -1 lee el canal alfa si este existe.
img_glasses = cv2.imread('glasses.png', -1)
```

4. En esta ocasión nuestra aplicación permite utilizar la cámara web o enviar constantemente una imagen que emula la entrada de la cámara o definir un vídeo. La cual se carga con el siguiente código.

```
# Definir una imagen para usar en pruebas y ajustar las ROIs:
test_face = cv2.imread("lena2.jpg")
```

5. En caso que deseemos utilizar esta imagen y no la de la cámara debemos comentar las líneas donde se inicia la captura de vídeo y descomentar la línea donde se asigna al frame nuestra imagen de prueba, tal como se muestra a continuación.

```
# Captura de vídeo desde la cámara
#ret, frame = video_capture.read()

# Utilizar la imagen de prueba como entrada
frame = test_face.copy()
```

[2] HaarCascade

6. Una vez terminada nuestra configuración inicial, procedemos a aplicar el detector de rostros en nuestra imagen de entrada, más precisamente en una copia en escala de grises de la misma. Esta función devuelve un conjunto de coordenadas(x, y), el alto y ancho para cada una de las caras detectadas.

```
# Detectar las caras utilizando 'detectMultiScale()'
faces = face_cascade.detectMultiScale(gray, 1.3, 5)
```

7. A continuación debemos detectar la ubicación de los ojos en cada una de la caras, lo cual se realiza con el detector Haar para los ojos. Para ello el primer paso es extraer las regiones donde se han detectado los rostros, tomando en cuenta su ubicación y dimensiones.

```
# Crear las ROISs basado en el tamaño
roi_gray = gray[y:y + h, x:x + w]
roi_color = frame[y:y + h, x:x + w]
```

8. Seguidamente, se utiliza el detector para identificar en cada ROI la ubicación de los ojos de en la imagen

```
# Detectar los ojos dentro de los rostros detectados
eyepairs = eyepair_cascade.detectMultiScale(roi_gray)
```

9. En el siguiente paso procedemos a calcular la región donde se ubicarán nuestras gafas, esto cálculos toman en cuenta las dimensiones de la imagen de la gafas a utilizar. Para comprenderlos debemos recordar que la imagen de las gafas es un poco más grande que los ojos que detectemos, por lo tanto, con estos cálculos estamos estimando la ubicación de dicha imagen sobre nuestra imagen de entrada.

```
# Calcular las coordenadas donde se ubicaran las gafas.
x1 = int(ex - ew / 10)
x2 = int((ex + ew) + ew / 10)
y1 = int(ey)
y2 = int(ey + eh + eh / 2)
```

10. Una vez determinadas las coordenadas donde se ubicarán las gafas, se procede a realizar las operaciones necesarias para cambiar el tamaño de la imagen de las gafas, al tamaño de la zona en la cual se superpondrá la imagen de las gafas. De igual manera se crea la máscara de esta mediante la operación `bitwise_not()`[3] la cual colocará en negro (0) los píxeles que tengan un valor blanco (255) y viceversa. En este punto la imagen a partir de la cual se creará la máscara, fue creada solo tomando el valor del canal alfa de la imagen original, por ende, tiene una mayoría de valores negros en el fondo y píxeles con valor mayor que 0 en donde se encuentran las gafas.

[3] bitwise_not()

11. Como resultado nuestra máscara solo tendrá valores bajos (negros y grises) en los píxeles donde están las gafas.

```
 # Calcular las dimensiones de la región
img_glasses_res_width = int(x2 - x1)
img_glasses_res_height = int(y2 - y1)

# Cambiar el tamaño de la máscara al de la región
mask = cv2.resize(img_glasses_mask, (img_glasses_res_width, img_glasses_res_height))

# Crear la máscara de las gafas
mask_inv = cv2.bitwise_not(mask)
```

12. Cuando se han creado las máscaras, se procede a editar los ROIs para las cuales fueron creadas, primero se crea una ROI que tendrá los valores de color originales de la imagen de entrada. Posteriormente usando una operación `bitwise_and()` se modifican los valores de estas regiones.

13. Para el fondo se utiliza la región con colores originales y la máscara inversa (mayormente con píxeles blancos), como resultado de la operación `bitwise_and()` saldrá la imagen con colores originales, pero con un valor negro (0) en los píxeles donde la máscara tenia un valor de 0. Ver figura 3 columna izquierda.

14. Para las gafas en primer plano se utilizará una copia de la imagen original de las gafas redimensionada, aplicando un `bitwise_and()` con la máscara creada a partir de la imagen original, la cual tiene píxeles blancos (255) donde están ubicadas las gafas. Debido al funcionamiento de esta operación el resultado será una imagen de las gafas con valores de píxeles negros donde la máscara tenia un valor negro. Dejando solo con un color diferente los píxeles donde se ubican las gafas, siendo en este caso el valor original de las mismas. Ver figura 3 columna derecha.

```
 # Crear una ROI desde la image de color
roi = roi_color[y1:y2, x1:x2]

# Crear los ROIs de fondo y primer plano:
roi_bakground = cv2.bitwise_and(roi, roi, mask=mask_inv)
roi_foreground = cv2.bitwise_and(img, img, mask=mask)
```

15. Como último paso se realiza una suma de las imágenes generadas a partir de las máscaras, en esta operación los píxeles negros tienen un valor de 0, por lo cual no modifican el píxel en esa coordenada de la otra imagen. Por lo cual, el resultado es una imagen donde los píxeles en negro ahora tienen un valor, terminando así el proceso y permitiendo su visualización. La suma se da entre las imágenes inferiores (izquierda y derecha) de la figura 3.

```
# Sumar las imágenes para producir el resultado
res = cv2.add(roi_bakground, roi_foreground)
```

```
# Colocar el resultado en la ROI original
roi_color[y1:y2, x1:x2] = res
```

16. Las máscaras y los resultados de su aplicación se pueden apreciar en la figura 3:

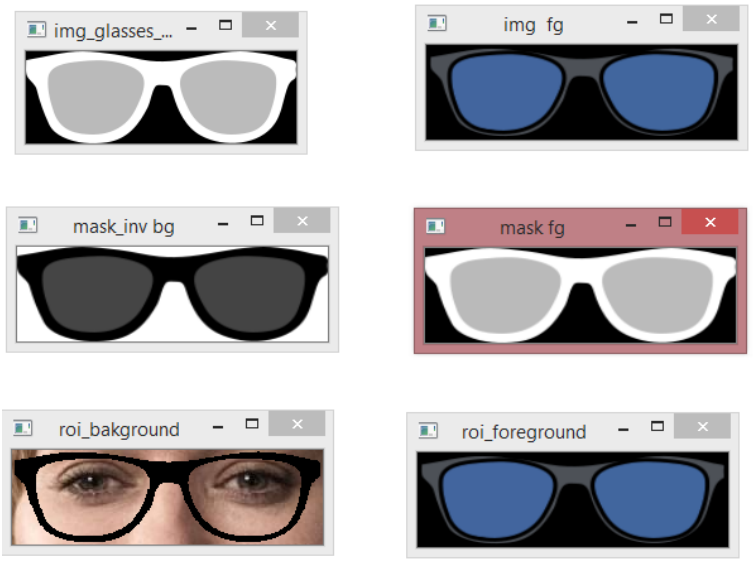

Figura 3: Máscaras y resultados de aplicación.

17. Para ejecutar se debe escribir el siguiente comando en la consola

```
python 01_snapchat_augmeted_reality_glasses.py
```

18. Luego de su ejecución nuestra imagen se verá de la siguiente manera (figura 4):

19. En el caso del segundo ejemplo de esta sección consiste en un código similar pero con ligeras adaptaciones para que en lugar de agregar gafas, agregue un bigote a la imagen de entrada.

20. Para ejecutar se debe escribir el siguiente comando en la consola

```
python 02_snapchat_augmeted_reality_moustache.py
```

21. Luego de su ejecución nuestra imagen se verá de la siguiente manera (figura 5):

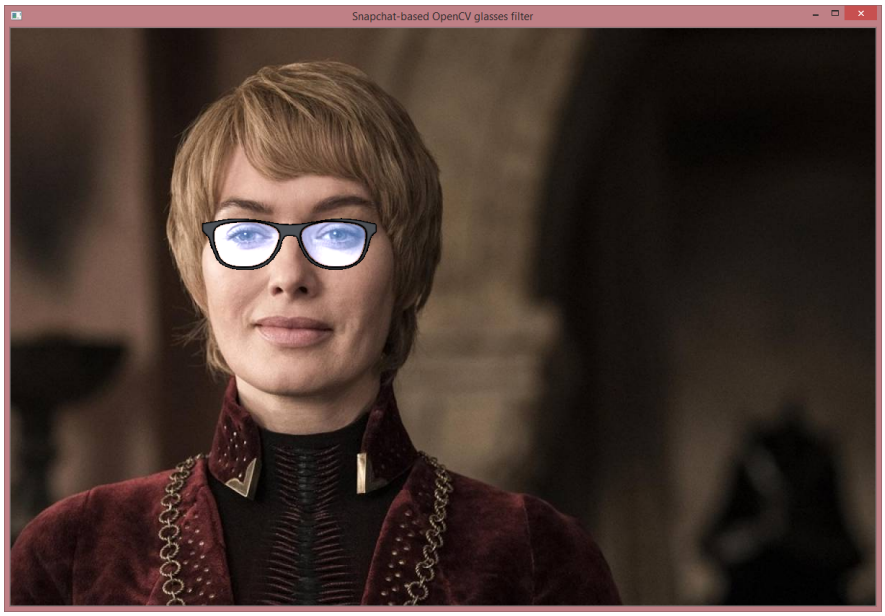

Figura 4: Gafas superpuestas a una imagen de entrada.

Entregables para la Sección

Para esta sección en su informe debe incluir la modificación de una imagen de su propiedad. En dicha imagen se deben realizar las siguientes operaciones:

1. Utilice el código para la detección ojos y modifíquelo para que detecte la boca de una persona y que se reemplace la misma por un bloque negro o algún icono, imagen de su preferencia o aplique algún tipo de filtro que modifique el área detectada. Este ejemplo busca emular los filtros de censura utilizados en la televisión.

Figura 5: Bigote superpuesto a una imagen de entrada.

3. Reconocimiento de Rostros con OpenCV

Uno de los puntos fuertes de OpenCV es la capacidad de detección y las diferentes gamas de procedimientos que se pueden realizar con los rostros. Existen diversas librerías adicionales que permiten trabajar los diversos enfoques, sin embargo en esta ocasión nos centraremos en las opciones que trae OpenCV incluidas en su código. Esta sección del laboratorio ha sido extraída de [4] [5].

Los datos para esta sección se encuentran en un archivo comprimido disponible en Moodle.

En esta sección utilizaremos el enfoque de reconocimiento facial, en el cual, le enseñaremos a un algoritmo a diferenciar entre dos rostros, mediante un proceso de entrenamiento y prueba.

Este ejemplo aplica nociones básicas de aprendizaje automático. Se utiliza un conjunto (*data-set*) de imágenes para entrenar un reconocedor. La salida de este ultimo consistirá en la etiqueta o categoría a la cual pertenece la imagen que se esta evaluando.

El proceso de entrenamiento incluye una fase en la cual se deben procesar las imágenes y extraer las características de las mismas, para nuestro ejemplo dichas características se extraerán y describirán utilizando el enfoque *Local Binary Patterns Histograms* (LBPH).

OpenCV cuenta con 3 reconocedores faciales integrados. Estos se pueden utilizar de manera independiente en este código simplemente cambiando una línea de código. Estos son:

1. EigenFaces Face Recognizer Recognizer `cv2.face.EigenFaceRecognizer_create()`
2. FisherFaces Face Recognizer Recognizer `cv2.face.FisherFaceRecognizer_create()`
3. Local Binary Patterns Histograms (LBPH) Face Recognizer `cv2.face.LBPHFaceRecognizer_create()`

3.1. Local Binary Patterns Histograms (LBPH) Face Recognizer

Una explicación detallada de LBPH puede ser encontrada en Face Detection.

Los reconocedores de Eigen y Fisher son afectados por la luz y esta es una condición que no se puede garantizar en situaciones de la vida real. El reconocedor usando LBPH es una mejora para superar esta desventaja. Su enfoque es utilizar descriptores locales en la imagen. LBPH trata de encontrar una estructura de la imagen y lo hace mediante la comparación de cada píxel con los de su vecindario.

Se toma una ventana de $3x3$ y se mueve a través de la imagen, en cada movimiento se compara el píxel central con los vecinos. Los vecinos con una intensidad menor o igual al del píxel central se marcan utilizando un 1 y los demás con un 0. Estos valores dentro de la ventana se leen en el sentido de las agujas del reloj lo que creará un patrón binario como 11100011 el cual es específico para esta zona de la imagen. Haciendo esto a través de toda la imagen se tendrá una lista de patrones locales binarios.

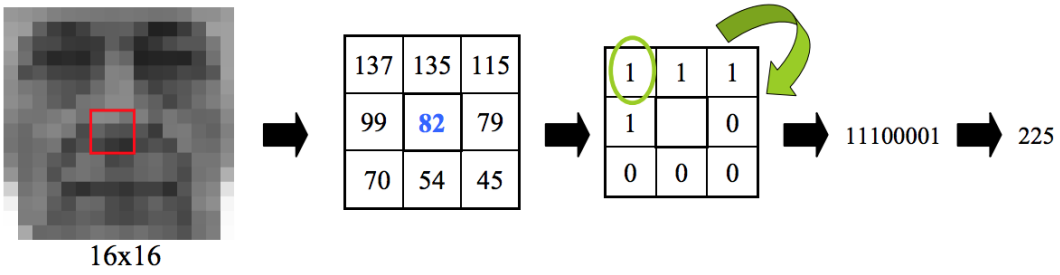

Figura 6: Construcción de Patrón Binario.

Con lo anterior se tiene la parte de los patrones binarios, para la creación del histograma, se convierte cada patrón en un número decimal (binario $\rightarrow$ decimal) y entonces se realiza un histograma de todos los valores decimales.

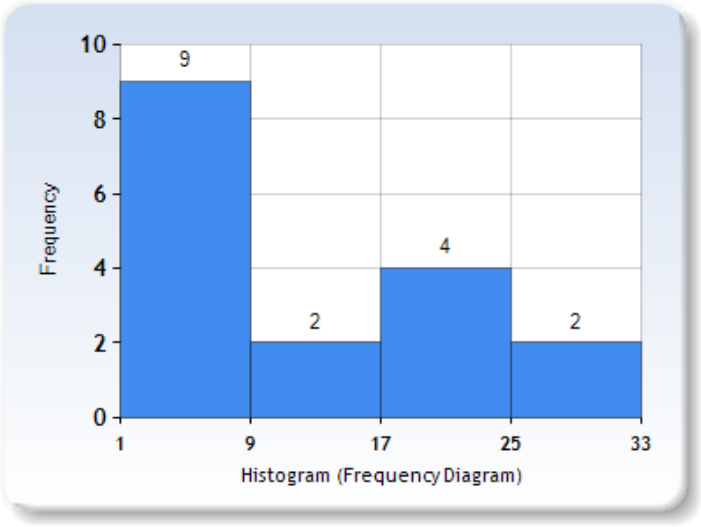

Figura 7: Histograma del Patrón.

Con este enfoque estaremos creando un histograma para cada cara en la imagen. Por lo cual, cuando tenemos un dataset de entrenamiento con 100 caras tendremos 100 histogramas diferentes que se almacenaran para realizar el proceso de reconocimiento posteriormente. El algoritmo sabe que cara pertenece a cada histograma. Durante la etapa de reconocimiento se pasará una imagen al reconocedor, el cual calculará el histograma de la cara detectada en la imagen y lo comparará con los histogramas que tiene almacenados, para devolver la categoría que mejor coincida con la imagen en evaluación.

En esta imagen podemos ver como los descriptores LBPH no son afectados por los factores de iluminación. Fuente.

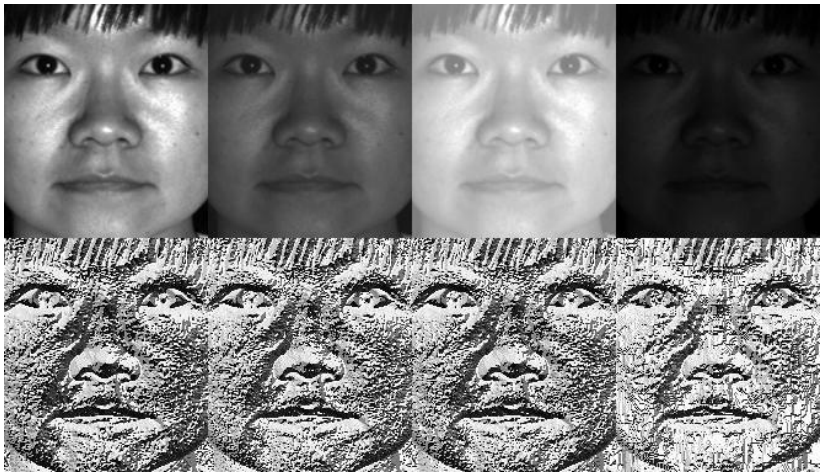

Figura 8: LBPH para diferentes iluminaciones.

3.1.1. Proceso de Reconocimiento Facial en OpenCV

El proceso de reconocimiento facial se puede dividir en 3 etapas:

1. **Preparar los datos de entrenamiento:** En este paso se leerán las imágenes de entrenamiento para cada persona con sus etiquetas, se detectaran las caras en cada imagen y se asignan a una etiqueta o label entero.
2. **Entrenar el reconocedor**: En este paso entrenaremos el reconocedor de caras de LBPH. enviándole/mostrándole la información que se ha preparado en el paso 1.
3. **Testing**: En esta etapa enviaremos algunas imágenes de prueba, para evaluar si la predicción se realiza de manera correcta.

3.2. Datos de entrenamiento

Entre mayor cantidad de imágenes por sujeto, los resultados serán mejores ya que el reconocedor será capaz de aprender datos de la misma persona desde diferentes puntos de vista. En este caso nuestro dataset tiene 12 imágenes de cada sujeto, los cuales se encuentran en el folder `training-data`, este contiene en su interior un folder para cada sujeto que deseamos reconocer cada folder tiene el formato sLabel (e.g. s1, s2)‘ donde el número es la etiqueta entera asignada a cada sujeto.

- training-data
  - s1
    - 1.jpg
    - ...
    - 12.jpg
  - s2
    - 1.jpg
    - ...
    - 12.jpg

El folder `test-data` contiene las imágenes que serán utilizadas para evaluar nuestro reconocedor luego de ser entrenado.

1. Se inicia con la importación de las librerías requeridas.

```
#import OpenCV module
import cv2
#import os module for reading training data directories and paths
import os
#import numpy to convert python lists to numpy arrays as
#it is needed by OpenCV face recognizers
import numpy as np
```

2. Las etiquetas en OpenCV deben ser de tipo entero, por lo cual se establece una forma de mapeado entre los números y los nombres de las personas. En nuestro caso no se utiliza el 0, por lo cual se deja vacío en la lista que contiene los nombres.

```
#there is no label 0 in our training data so subject name for index/label 0 is empty
subjects = ["", "Ruben Blades", "Elvis Presley"]
```

3. Como siguiente paso debemos preparar los datos de entrenamiento. Para entrenar el reconocedor OpenCV necesita dos arreglos, uno con los rostros (histograma de patrones) de los sujetos de en el conjunto de entrenamiento y el segundo vector contiene, en el mismo orden, las etiquetas de cada rostro

4. Por lo cual, si nuestro *dataset* contiene datos en esta forma:

PERSON-1	PERSON-2
img1	img1
img2	img2

5. Las listas producidas tendrán la siguiente estructura.

FACES	LABELS
person1_img1_face	1
person1_img2_face	1
person2_img1_face	2
person2_img2_face	2

6. Esta preparación se puede resumir como:

   a) Procesar la carpeta de entrenamiento, de donde se obtendrán la cantidad de personas que estarán en el reconocedor.

   b) Para cada sujeto se debe extraer la etiqueta que se le asignará y almacenarla en formato de entero.

   c) Leer las imágenes para cada personas y detectar la cara en cada una de estas.

   d) Añadir cada cara a la lista de caras y su etiqueta correspondiente a la lista de etiquetas.

```
#function to detect face using OpenCV
def detect_face(img):
    #convert the test image to gray image as opencv face detector expects gray images
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    #load OpenCV face detector, I am using LBP which is fast
    #there is also a more accurate but slow Haar classifier
    face_cascade = cv2.CascadeClassifier('opencv-files/lbpcascade_frontalface.xml')

    #let's detect multiscale (some images may be closer to camera than others) images
    #result is a list of faces
    faces = face_cascade.detectMultiScale(gray, scaleFactor=1.2, minNeighbors=5);

    #if no faces are detected then return original img
    if (len(faces) == 0):
        return None, None

    #under the assumption that there will be only one face,
    #extract the face area
    (x, y, w, h) = faces[0]

    #return only the face part of the image
    return gray[y:y+w, x:x+h], faces[0]
```

7. El detector LBP necesita trabajar con imágenes en escala de grises, de igual manera se debe realizar como primer paso recorte de la ubicación del rostro dentro de la imagen. En este

fragmento de código se realiza la conversión de la imagen a escala de grises y se utilizan el `cv2.CascadeClassifier` mediante un modelo de detección frontal de rostros, para realizar el recorte de la cara en la imagen. Este método devuelve el (x, y, width, height) de la zona en la cual se encuentra el rostro, para que pueda ser extraída utilizando OpenCV

```
#this function will read all persons' training images, detect face from each image
#and will return two lists of exactly same size, one list
# of faces and another list of labels for each face
def prepare_training_data(data_folder_path):

    #------STEP-1--------
    #get the directories (one directory for each subject) in data folder
    dirs = os.listdir(data_folder_path)

    #list to hold all subject faces
    faces = []
    #list to hold labels for all subjects
    labels = []

    #let's go through each directory and read images within it
    for dir_name in dirs:

        #our subject directories start with letter 's' so
        #ignore any non-relevant directories if any
        if not dir_name.startswith("s"):
            continue;

        #------STEP-2--------
        #extract label number of subject from dir_name
        #format of dir name = slabel
        #, so removing letter 's' from dir_name will give us label
        label = int(dir_name.replace("s", ""))

        #build path of directory containin images for current subject subject
        #sample subject_dir_path = "training-data/s1"
        subject_dir_path = data_folder_path + "/" + dir_name

        #get the images names that are inside the given subject directory
        subject_images_names = os.listdir(subject_dir_path)

        #------STEP-3--------
        #go through each image name, read image,
        #detect face and add face to list of faces
        for image_name in subject_images_names:

            #ignore system files like .DS_Store
            if image_name.startswith("."):
                continue;

            #build image path
```

```
            #sample image path = training-data/s1/1.pgm
            image_path = subject_dir_path + "/" + image_name

            #read image
            image = cv2.imread(image_path)

            #display an image window to show the image
            cv2.imshow("Training on image...", image)
            cv2.waitKey(100)

            #detect face
            face, rect = detect_face(image)

            #------STEP-4--------
            #for the purpose of this tutorial
            #we will ignore faces that are not detected
            if face is not None:
                #add face to list of faces
                faces.append(face)
                #add label for this face
                labels.append(label)

    cv2.destroyAllWindows()
    cv2.waitKey(1)
    cv2.destroyAllWindows()

    return faces, labels
```

8. Esta función cumple la misión de preparar los datos de entrenamiento, recibiendo la ruta de la carpeta de entrenamiento y devolviendo las listas de caras y etiquetas de cada cara.

```
#let's first prepare our training data
#data will be in two lists of same size
#one list will contain all the faces
#and other list will contain respective labels for each face
print("Preparing data...")
faces, labels = prepare_training_data("training-data")
print("Data prepared")

#print total faces and labels
print("Total faces: ", len(faces))
print("Total labels: ", len(labels))
```

3.2.1. Entrenamiento del Reconocedor

9. En este paso procedemos a instanciar el reconocedor y posteriormente se realiza el entrenamiento del mismo utilizando el método train(faces-vector, labels-vector) el cual recibe la

lista de caras y etiquetas del conjunto de entrenamiento.

```
#train our face recognizer of our training faces
face_recognizer.train(faces, np.array(labels))
```

3.2.2. Predicción

10.

```
#function to draw rectangle on image
#according to given (x, y) coordinates and
#given width and heigh
def draw_rectangle(img, rect):
    (x, y, w, h) = rect
    cv2.rectangle(img, (x, y), (x+w, y+h), (0, 255, 0), 2)

#function to draw text on give image starting from
#passed (x, y) coordinates.
def draw_text(img, text, x, y):
    cv2.putText(img, text, (x, y), cv2.FONT_HERSHEY_PLAIN, 1.5, (0, 255, 0), 2)
```

11. Estas funciones dibujarán un rectangulo sobre el rostro detectado y escribiran la etiqueta que ha definido el detector que corresponde con el rostro.

```
#this function recognizes the person in image passed
#and draws a rectangle around detected face with name of the
#subject
def predict(test_img):
    #make a copy of the image as we don't want to chang original image
    img = test_img.copy()
    #detect face from the image
    face, rect = detect_face(img)

    #predict the image using our face recognizer
    label= face_recognizer.predict(face)
    #print(label[0])
    #get name of respective label returned by face recognizer
    label_text = subjects[label[0]]

    #draw a rectangle around face detected
    draw_rectangle(img, rect)
    #draw name of predicted person
    draw_text(img, label_text, rect[0], rect[1]-5)

    return img
```

12. En este caso utilizamos el reconocedor entrenado para definir una etiqueta para un rostro en una imagen de prueba. Para esto se utiliza el método `predict(face)`, este retorna una tupla que contendrá el label(entero) al cual el reconocedor asignó la imagen y también un valor de confianza/probabilidad de dicho resultado.

```
print("Predicting images...")

#load test images
test_img1 = cv2.imread("test-data/test0.jpg")
test_img2 = cv2.imread("test-data/test6.jpg")

#perform a prediction
predicted_img1 = predict(test_img1)
predicted_img2 = predict(test_img2)
print("Prediction complete")

#display both images
cv2.imshow(subjects[1], predicted_img1)
cv2.imshow(subjects[2], predicted_img2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Entregables para la Sección

1. Agregue 2 personas adicionales al *dataset*, se recomienda que estas sean del mismo genero, en otras palabras, 2 mujeres o 2 hombres.
2. Construya un reconocedor utilizando usando `LBPH`.
3. Construya un reconocedor utilizando usando $Fisher$ o $Eigen$. Se recomienda aplicar un `cv2.resize()` a la salida la función que detecta las caras ya que estos métodos requieren que todas las imágenes (crops de caras) tengan el mismo tamaño.
4. Construya un *dataset* de prueba con entre 4-6 imágenes para cada persona en el *dataset*, estas imágenes no deben formar parte del *dataset* de entrenamiento.
5. Imprima el valor de confidencia y etiqueta obtenidos por cada reconocedor, para cada imagen de prueba.
6. Cuantifique los aciertos y fallos para cada reconocedor.

Referencias

[1] A. Fernández Villán, *Mastering OpenCV 4 with Python: a practical guide covering topics from image processing, augmented reality to deep learning with OpenCV 4 and Python 3.7. Mastering Open Source Computer Vision four with Python*. Birmingham: Packt Publishing, 2019. [Online]. Available: https://cds.cern.ch/record/2674578

[2] E. Cruz, S. Orts-Escolano, F. Gomez-Donoso, C. Rizo, J. C. Rangel, H. Mora, and M. Cazorla, "An augmented reality application for improving shopping experience in large retail stores," *Virtual Reality*, vol. 23, no. 3, pp. 281–291, 2019.

[3] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, vol. 1. IEEE, 2001, pp. I–I.

[4] Face recognition using OpenCV and python: A beginner's guide - blogs - SuperDataScience | machine learning | AI | data science career | analytics | success. [Online]. Available: https://www.superdatascience.com/blogs/opencv-face-recognition

[5] R. Raja, "Face recognition with OpenCV and python," original-date: 2017-07-13T13:43:39Z. [Online]. Available: https://github.com/informramiz/opencv-face-recognition-python