

Práctica 4. Eigenfaces.

(Diciembre, 2021)

1st Guerra Silva Erick Iván
Ingeniería en Computación
Universidad Nacional Autónoma de México
 Ciudad de México, México
erickivanguerra0.0@gmail.com

3rd Martínez Gutiérrez Carlos
 Giovanni
Ingeniería en Computación
Universidad Nacional Autónoma de México
 Ciudad de México, México
cgiovanne@comunidad.unam.mx

2nd Lázaro Martínez Abraham Josué
Ingeniería en Computación
Universidad Nacional Autónoma de México
 Ciudad de México, México
abrahamlazaro@comunidad.unam.mx

4th Castillo Sánchez Axel
Ingeniería Mecatrónica
Universidad Nacional Autónoma de México
 Ciudad de México, México
axel_castillo98@hotmail.com

Resumen—Este documento presenta los resultados obtenidos para los ejercicios planteados en la práctica para realizar la caracterización y clasificación de texturas. Toda la práctica se realizó en lenguaje Python utilizando Google Collaboratory y se utilizaron módulos como MultinomialNB de Sklearn para la creación de un clasificador de Bayes, Neighbors también de Sklearn para la implementación de un clasificador K-NN, al igual que el módulo cross_val_score de Sklearn para medir el desempeño de los resultados obtenidos.

Índice de Términos—Procesamiento de Imágenes, caracterización y clasificación, clasificador, segmentación, matriz.

I. INTRODUCCIÓN

Eigenface

Los eigenfaces se refieren a un enfoque sobre el reconocimiento de rostros que busca capturar la variación en una colección de imágenes de rostros y utilizar esta información para codificar y comparar imágenes de rostros individuales de una manera holística (en contraposición a una basada en partes).

Las eigenfaces son los componentes invariables de una serie de caras, o lo que es lo mismo, los vectores propios de la matriz de covarianza del conjunto de imágenes de caras que forman una base de datos.

Esto quiere decir, que extraen la información facial relevante, que puede o no estar directamente relacionada con la intuición humana de rasgos faciales como los ojos, la nariz y los labios. Una forma de hacerlo es capturar la variación estadística entre las imágenes faciales. Los eigenfaces pueden considerarse como un conjunto de rasgos que capturan la variación global entre las imágenes faciales. [1]
<https://proyectoidis.org/eigenface/>

K-Means

K-means es un algoritmo de clasificación no supervisada (clusterización) que agrupa objetos en k grupos basándose en sus características. El agrupamiento se realiza minimizando la suma de distancias entre cada objeto y el centroide de su grupo o cluster. Se suele usar la distancia cuadrática.

El algoritmo consta de tres pasos:

- Inicialización: una vez escogido el número de grupos, k, se establecen k centroides en el espacio de los datos, por ejemplo, escogiendo aleatoriamente.
- Asignación objetos a los centroides: cada objeto de los datos es asignado a su centroide más cercano.
- Actualización centroides: se actualiza la posición del centroide de cada grupo tomando como nuevo

RECONOCIMIENTO DE PATRONES. PRÁCTICA 4

centroide la posición del promedio de los objetos pertenecientes a dicho grupo. [2]

PCA.

Es un método matemático que se utiliza para reducir el número de variables de forma que pasemos a tener el mínimo número de nuevas variables y que representen a todas las antiguas variables de la forma más representativa posible. Es decir, si se reduce el número de variables a dos o tres nuevas, se pueden representar los datos originales en el plano o en un gráfico de 3 dimensiones y, así, se visualiza de forma gráfica un resumen de nuestros datos. El simple hecho de poder tener los datos de manera visible simplifica mucho el entender qué puede estar pasando y ayuda a tomar decisiones. [3]

II. OBJETIVOS

El alumno:

- Desarrollar una aplicación del análisis de componentes principales (PCA) y k-medias para el análisis de "familiar faces".

III. RESULTADOS

1. Eigenfaces

Las imágenes disponibles en el dataset *MIT Faces Recognition Project* se caracterizan por tener un formato *RAW* o crudo, lo cual quiere decir que no tienen un preprocessamiento y nos ofrece la ventaja de que podemos ajustarlas a nuestras necesidades. En este caso, decidimos redimensionar (*reshape*) todas las imágenes a 128x128 pixeles usando la función `cargaImagenes(path)` la cual simplemente recibe como parámetro el *path* y en caso de que no sea posible reformarse alguna de las imágenes, simplemente es omitida y no se considera dentro del análisis posterior.

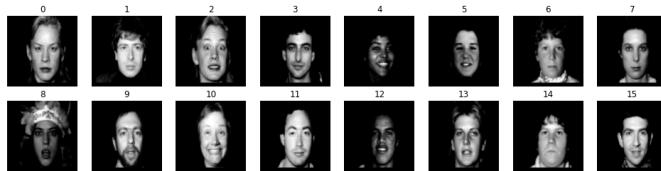


Fig. 1. Despliegue de las primeras 16 imágenes cargadas.

Preprocesamiento de las imágenes

Ahora bien, una vez cargadas las imágenes, procedimos a la preparación de la información para el Análisis de Componentes Principales (*PCA*, por sus siglas en inglés).

Primero, calculamos una “imagen promedio” de todo el dataset y ammacenamos las imágenes en un arreglo.

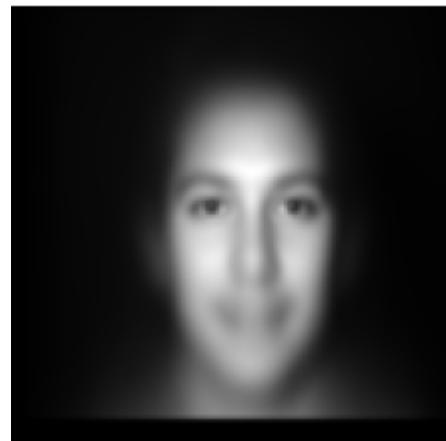


Fig. 2. Imagen promedio del dataset.

Posteriormente se calcularon los puntos centrados o *centered points* de cada imagen, los cuales no son nada más que la resta entre una imagen original y la imagen promedio.



Fig. 3. Punto centrado de la imagen 0.

Implementación de PCA

Se usa el método Singular Value Decomposition (SVD) porque se encarga de reducir la dimensionalidad de la información y devuelve las características principales de forma “comprimida”.

Implementarlo es, en realidad, bastante sencillo y se reduce a la siguiente ecuación.

$$\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^T$$

Donde:

\mathbf{X} , son los puntos centrados.

\mathbf{U} , almacena la información del espacio columna de \mathbf{X} y está ordenada jerárquicamente.

Σ , es una matriz diagonal que nos indica qué tan importantes son las columnas de \mathbf{U} y \mathbf{V} .

\mathbf{V}^T , contiene la información del espacio renglón de \mathbf{X} y está ordenada jerárquicamente.

RECONOCIMIENTO DE PATRONES. PRÁCTICA 4

```
U, Sigma, V_T = np.linalg.svd(X_centered,
full_matrices=False)
```

```
X: (3500, 16384)
U: (3500, 3500)
Sigma: (3500,)
V^T: (3500, 16384)
```

Fig. 4. Dimensiones de las matrices.

En la siguiente gráfica se muestran los primeros 500 valores singulares de los 3500 que se obtuvieron, estos contienen la importancia de las columnas de las matrices U y V.

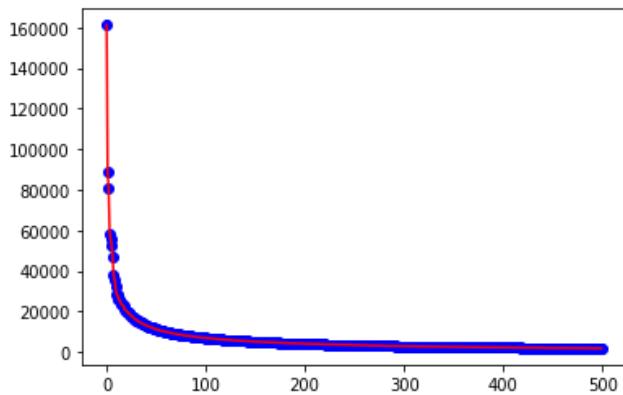


Fig. 5. Gráfica de los valores singulares.

La matriz de interés es V^T ya que contiene nuestros componentes principales. Para desplegarlos como imagen, simplemente tomamos las columnas de dicha matriz y le aplicamos un *reshape*. De esta manera, obtuvimos los siguientes resultados.

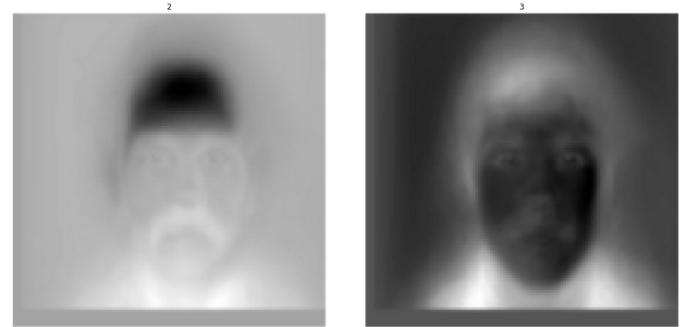
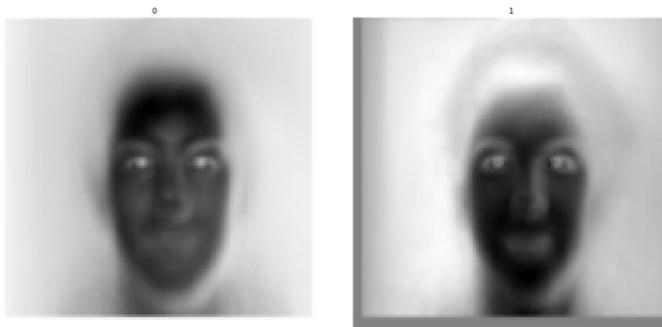


Fig. 6. Primeros cuatro componentes principales.

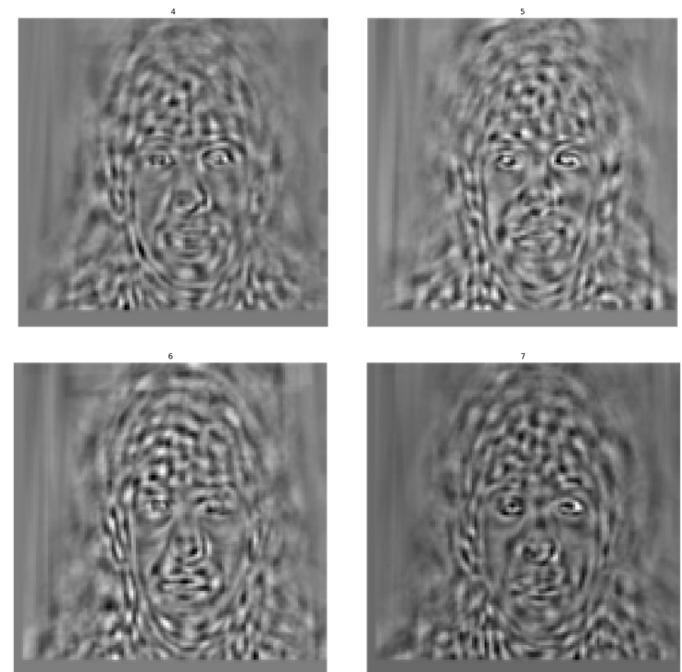


Fig. 7. Últimos cuatro componentes principales.

De estas imágenes podemos observar como los primeros cuatro componentes principales contienen una gran cantidad de información relativa a la estructura y componentes de una cara. Por otro lado, los últimos cuatro contienen información que aparentemente resulta ser menos relevante.

Posteriormente graficamos para visualizar los puntos de datos faciales proyectados en las dos primeras componentes principales de nuestra variable obtenida anteriormente.

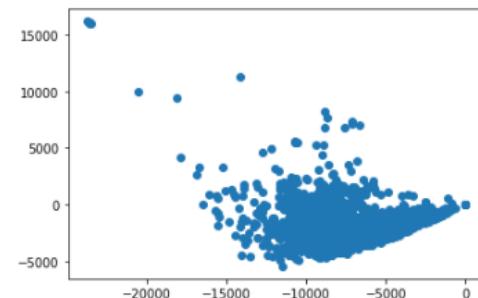


Fig. 8. Graficación de los componentes principales

RECONOCIMIENTO DE PATRONES. PRÁCTICA 4

Aplicamos K-Means sobre los datos proyectados para formar clusters, en esta práctica utilizaremos tres clusters, primero importamos la librería de “K Means” para posteriormente utilizar la función, a la cual se le pasaron el número de clusters, el número de EigenFaces, que en este caso fueron 500 y el número de componentes, que previamente ya habíamos definido como 5, esta función nos regresaba los nuevos grupos obtenidos.

```
m = 500
k = 3
kmeans = KMeans(n_clusters=k).fit(Y[:,m,
:numero_componentes])
CentroidesP = kmeans.cluster_centers_
```

Para graficar los centros de los clusters, primero definimos los tres colores, por el número de clusters que habíamos elegido, para posteriormente seleccionar los centros de la siguiente manera.

```
colores=['red', 'blue', 'cyan']
ax.scatter(CentroidesP[:, 0], CentroidesP[:, 1],
CentroidesP[:, 2], marker='*', c=colores, s=1000)
```

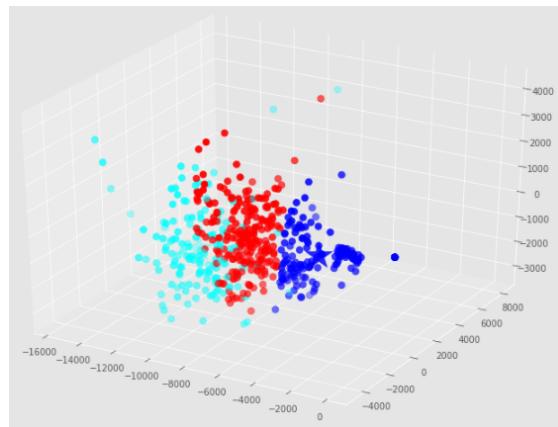


Fig. 9. Gráfica de los elementos en 3D y los centros de los clusters

2. Face Reconstruction and A Simple Face Detector.

Para la parte de reconstrucción facial, primero imprimimos una EigenFace, para poder observar el tipo de datos que nos arrojaba la parte 1, para poder imprimir la imagen, utilizaremos la biblioteca “imshow”, a la cual se le pasaba el número de EigenFace, el tamaño y el cmap, en este caso para imprimirla en escala de grises.

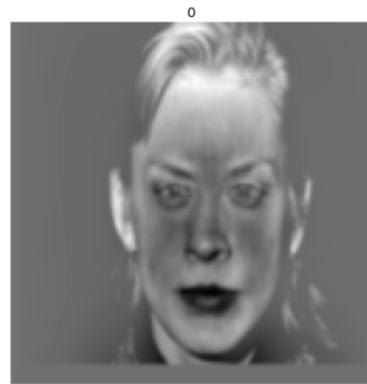


Fig. 10. Impresión de la 0 EigenFace

Para poder realizar la reconstrucción facial, realizamos una función llamada “mulMat”, esta función sólo recibe dos parámetros, el primer parámetro es el número de EigenFaces con las que se trabajara, el segundo parámetro es la imagen con la que se trabajara, en la función se realiza lo siguiente, lo primero es obtener en una matriz el número de EigenFaces con las que se trabajará, primero procederemos a multiplicar la matriz con el número de EigenFaces elegidas con la cara que se le va a realizar la reconstrucción facial, a esta matriz resultante, la multiplicaremos con la matriz que tiene contiene el número de EigenFaces a trabajar.

```
def mulMat(componentes, imagen):
    pc = V_T[:componentes]
    resultado = pc.T @ (pc @ X_centered[imagen])
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
```

Después de pasar por la multiplicación se imprimen las dos imágenes obtenidas, cabe recalcar que la imagen obtenida a partir de las EigenFaces, se le debe de hacer un reshape, para poder visualizarla.

En el caso de esta práctica se utilizaron 10 diferentes EigenFaces, esto con el objetivo de ver como iba cambiando la imagen resultante, después de utilizar un número grande EigenFaces, al realizar esto descrito anteriormente, nos pudimos dar cuenta que entre era mayor el número de EigenFaces, la imagen resultante se visualizaba mejor.

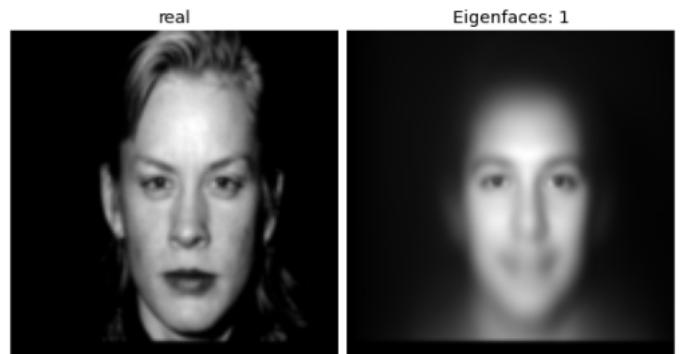


Fig. 11. Reconstrucción facial utilizando una EigenFace

RECONOCIMIENTO DE PATRONES. PRÁCTICA 4

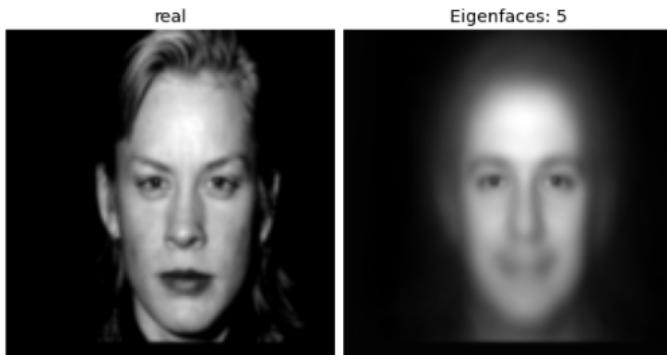


Fig. 12. Reconstrucción facial utilizando cinco EigenFaces



Fig. 13. Reconstrucción facial utilizando 1000 EigenFaces



Fig. 14. Reconstrucción facial utilizando 2000 EigenFaces

Face Morphing

Para poder crear la animación a partir de Morphing, primero seleccionamos las dos caras con las que trabajaríamos para poder transformar la primer cara en la segunda cara, posteriormente creamos una función llamada `generar_rostros_animacion` esta función recibe 5 parámetros, el primer parámetro es la matriz con todas las EigenFaces creadas en la primer parte, el segundo parámetro es el número menor de EigenFace a utilizar, el tercer parámetro es el número máximo EigenFaces a utilizar, el cuarto parámetro es la imagen, en este caso el rostro que utilizaremos y el quinto parámetro es un dato de tipo booleano, en el que le daremos el orden en que se generará la animación de rostro, es decir de mayor a menor o de menor a mayor, para que de esta manera se pueda crear la ilusión óptica.

```
def generar_rostros_animacion(V_T, menor, mayor,
                               imagen, inverse=False):
```

Después de obtener el orden en que se crearía la lista, se procederá a acceder a un for, en donde los resultados obtenidos se guardarán en una lista, es decir primero se iniciará con el mínimo de EigenFaces y cada interacción se guardará en la lista hasta llegar al número máximo de EigenFaces.

```
rostrosL = list()
for i in range(A,B,paso):
    pc = V_T[:i]
    resultado = pc.T@(pc@imagen)
    frame = np.reshape(resultado+mean_face, (128, 128))
    rostrosL.append((frame,i))
```

Al terminar de iterar esta función regresa la lista con los distintos rostros creados a partir de las EigenFaces; Por lo que se crearon dos listas, la primer lista para la primer cara que va de mayor a menor y una segunda lista para la segunda cara, la cual va de mayor a menor y se concatena con la primera lista ya obtenida.

```
rostros = generar_rostros_animacion(V_T, 10, 200,
                                     X_centered[0], True)
rostros = rostros + generar_rostros_animacion(V_T,
                                               10, 200, X_centered[1], False)
```

Después de obtener la lista, ocuparemos una función para poder crear la animación, para poder crear la animación, utilizamos la siguiente librería y función, para poder renderizar el video y de esa manera poder mostrarlo.

```
from IPython.display import HTML
HTML(ani.to_html5_video())
```



Fig. 15. Muestra el video con las dos caras

3. Generación de rostros

Además de la obtención de eigenfaces en el punto 1 y reconstrucción facial en el punto 2, realizamos de manera extra la generación de rostros. Para esto, utilizamos los resultados obtenidos en los puntos anteriores, es decir, las matrices resultantes después de aplicar PCA a nuestro conjunto de imágenes al igual que los datos centrados.

Lo primero que realizamos fue multiplicar la matriz

RECONOCIMIENTO DE PATRONES. PRÁCTICA 4

de componentes vectoriales V^T por la matriz transpuesta que contiene las 3000 primeras imágenes de rostros centrados (después de haber restado la “imagen promedio”). Después de multiplicar obtenemos una matriz A en la que se tienen las EigenFaces de las diferentes imágenes del dataset. Una vez con esta matriz, obtenemos los valores máximo y mínimo.

```
A = V_T@X_centered[:3000].T
mx = np.max(A, axis=0)
mn = np.min(A, axis=0)
```

Los valores máximo y mínimo obtenidos fueron utilizados como límites para la generación de valores aleatorios que generarían eigenfaces aleatorias, mismas que serían agregadas en una lista y estarían actuando como “ruido”.

```
lista = []
for i in range(3000):

    lista.append(np.random.normal(mn[i], mx[i], [1, 1])[0])
ruido = np.array(lista)
```

Posteriormente, creamos una función llamada *generar*, la cual se encargaría de generar el rostro tomando esta muestra aleatoria y el número de componentes principales a utilizar para la generación de la imagen. Para obtener la imagen generada aleatoriamente, multiplicamos la matriz de vectores singulares por la matriz de ruido para obtener su representación a través de las EigenFaces obtenidas. Una vez con la imagen generada, únicamente se realizó un reshape para poder desplegarla con matplotlib.

```
def generar(componentes, imagen):
    pc = V_T[:componentes]
    resultado = pc.T@imagen[:componentes]

    plt.imshow(np.reshape(resultado+mean_face.reshape(128*128, 1), (128, 128)), cmap="gray")
    plt.xticks([])
    plt.yticks([])
    plt.title("Generada "+str(componentes))
    plt.show()
```

Para la generación de estas imágenes aleatorias variamos el número de componentes principales a utilizar, desde 1 hasta 2000 para poder ver las variaciones de las imágenes generadas al modificar este parámetro.

```
for i in [1, 5, 10, 20, 50, 100, 200, 500, 1000, 2000]:
    imagen = ruido[:i]
    generar(i, imagen)
```

A continuación, se presentan los resultados obtenidos.

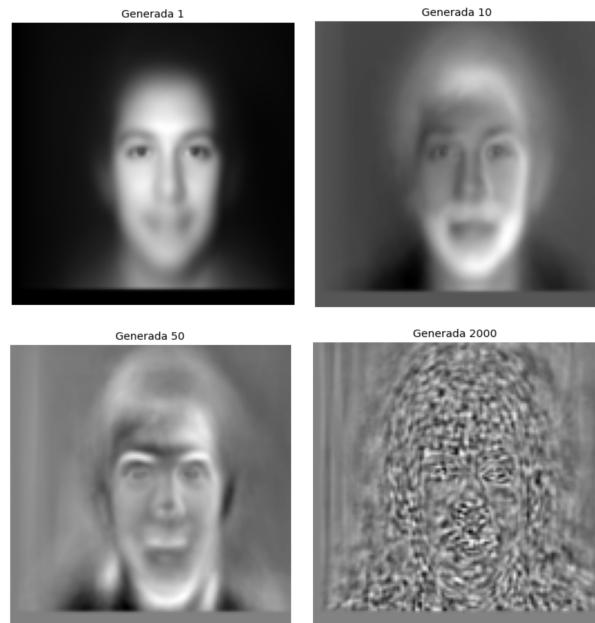


Figura 16. Rostro generado aleatoriamente

Como podemos observar en las imágenes, la primera imagen generada con una sola componente principal, se obtiene algo muy cercano a la imagen promedio. Por otra parte, cuando se toman 10 componentes principales se obtiene la mejor aproximación de un rostro aleatorio, siendo claro y muy diferente al rostro promedio. Sin embargo, cuando se utilizan 50 componentes el rostro generado comienza a distorsionarse, y finalmente, utilizando 2000 componentes ya se encuentra completamente distorsionado.

Algo interesante que pudimos observar es que para la generación de rostros sucedió todo lo contrario que con face morphing, para face morphing, entre más componentes principales se tomaban en cuenta, mejor era el morphing, mientras que para la generación de rostros, entre mayor fuera el número de componentes, mayor era también la distorsión del rostro llegando a una forma que incluso ya no podía ser considerado como un rostro.

A pesar de que no se obtuvieron resultados perfectos, consideramos que se obtuvieron resultados muy buenos que tal vez podrían ser mejorados si variamos los diferentes parámetros o incluso si el ruido que se genera es un poco más acotado y definido ya que en nuestro caso, solo generamos valores aleatorios entre un rango, es decir, no teníamos reglas para una asignación correspondiente a las imágenes originales.

IV. CONCLUSIONES

Al término de esta práctica, nos quedó más claro que era una eigenface y cómo pueden ser utilizadas para face morphing e incluso para la generación de rostros. Aunque al inicio de la realización de la práctica comenzamos a realizar el código, en base al documento de la práctica, tuvimos un error ya que estábamos trabajando con los datos incorrectos,

RECONOCIMIENTO DE PATRONES. PRÁCTICA 4

después de corregir el error y cargar los datos correctos, comenzamos con la creación del código para poder obtener nuestra matriz con todas las EigenFaces y poder aplicar face morphing y generar rostros aleatorios.

Algo que se nos hizo complicado al inicio fue saber qué datos eran los datos correctos ya que en la página del MIT había diversos archivos con diferentes formatos y no sabíamos cuáles eran los datos que debíamos usar. Otra dificultad al momento de realizar la práctica es que al principio solo estábamos copiando y pegando el código de la página de referencia sin entender por completo lo que se estaba haciendo, sin embargo, llegamos a un punto en el que no sabíamos cómo hacer el siguiente paso ya que estaba explicado en palabras pero no en código. Esto nos hizo regresar unos pasos atrás y comenzar a realizar el código por nuestra propia cuenta; de esta manera fue más sencillo seguir los pasos y a la vez entender todo lo que estábamos haciendo, a pesar de esto, tuvimos algunas dudas de conceptos teóricos que resolvimos consultando papers e información sobre PCA.

Con esta práctica comprendimos el uso de las EigenFaces y sus usos, que van desde face morphing o convertir un rostro a otro de manera progresiva o la generación de rostros aleatorios. Únicamente con las eigenfaces podemos obtener mucha información de un rostro, esto es muy relevante ya que de esta manera podemos conservar sólo las eigenfaces, reduciendo así el tamaño de las imágenes y el tiempo necesario para computarlas sin perder información relevante.

Durante la realización de esta práctica pudimos observar que el número de componentes utilizados podían hacer una gran diferencia. Por una parte, para face morphing, entre más componentes principales se tomaban en cuenta, mejor era la imagen obtenida o más parecida a la imagen destino, mientras que para la generación de rostros, entre mayor era el número de componentes, mayor era también la distorsión del rostro llegando a una forma que incluso ya no podía ser considerado como un rostro.

Al graficar los puntos de datos faciales proyectados en las dos primeras componentes principales nos dimos cuenta que se concentraban en la esquina inferior derecha, esto hizo que se nos ocurriera generar valores aleatorios en este rango y después generar un rostro aplicando una combinación lineal con las eigenfaces y después mostrarlo. No obtuvimos resultados perfectos pero tuvimos una muy buena aproximación que tal vez podría mejorar al hacer un poco menos aleatorio la generación de este vector de ruido y convertirse en una imagen más consistente y definida sin importar el número de componentes utilizados.

Otro punto importante de esta práctica es que aprendimos a realizar animaciones utilizando el módulo HTML de la librería IPython.display, esta librería nos permitió poder observar la transición del face morphing paso a paso sin

tener que ir desplazando la pantalla para ver las gráficas generadas con matplotlib. En la página de referencia se mencionaba la librería Bokeh para realizar gráficos interactivos, intentamos utilizarla pero no obtuvimos los resultados mostrados en la página ya que se debían hacer configuraciones específicas que no logramos encontrar.

REFERENCIAS

- [1] Eigenface | IDIS. (s. f.). proyectoidis.org/eigenface/ (accessed Nov. 25, 2021)
- [2] kmeans. (s. f.). [unioviendo.es/compnum/laboratorios_py/kmeans/kmeans.html](https://www.unioviendo.es/compnum/laboratorios_py/kmeans/kmeans.html) (accessed Nov. 25, 2021).
- [3] Carral, A. (2021, 20 mayo). [unioviendo. LIS Data Solutions.](https://www.lisdatasolutions.com/blog/algoritmo-pca-de-lo-complejo-a-lo-sencillo/)
<https://www.lisdatasolutions.com/blog/algoritmo-pca-de-lo-complejo-a-lo-sencillo/> (accessed Nov. 25, 2021)

Práctica 4. EigenFaces

<https://sandipanweb.wordpress.com/2018/01/06/eigenfaces-and-a-simple-face-detector-with-pca-svd-in-python/>

▼ 1. EigenFaces

Antes de aplicar PCA, debemos aplicar un preprocessamiento a las imágenes con las que vamos a trabajar.

```
import sys
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

from google.colab import drive
drive.mount('/content/drive')
```

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/conte

Cargamos las imágenes obtenidas de: [MIT Faces Recognition Project database](#).

```
# Motsrar imagen original
def display_face(filename):
    path = '/content/drive/MyDrive/Colab Notebooks/EigenFaces/rawdata/' + filename
    raw = np.fromfile(path, dtype='uint8', sep="")
    img = np.reshape(raw, (128, 128))
    plt.imshow(img, cmap="gray")
```

```
filename="1227"
display_face(filename)
```

▼ Carga de datos

Caras Originales

```
def cargaImagenes(path):
    original_images = []
```

```
i = 0
print(len(os.listdir(path)))
for filename in os.listdir(path):
    try:
        #print(filename)
        # Usamos uint8 por que así están codificadas las imágenes
        raw = np.fromfile(path+filename, dtype='uint8', sep="")
        img = np.reshape(raw, (128, 128))
    except FileNotFoundError:
        continue
#Para omitir imágenes que no pueden ser redimensionadas a 128x128
except ValueError as e:
    print(e)
    continue
original_images.append(img)
i += 1
if i >= 3500:
    break
return original_images
```

```
#path = '/content/drive/MyDrive/Colab Notebooks/EigenFaces/rawdata/'
#imágenes = cargaImagenes(path)
```

```
path = '/content/drive/MyDrive/Aux/EigenFaces/rawdata/'
imágenes = cargaImagenes(path)
```

```
3993
cannot reshape array of size 262144 into shape (128,128)
cannot reshape array of size 262144 into shape (128,128)
```

Imprimimos una muestra de las imágenes cargadas.

```
fig, axes = plt.subplots(2,8, figsize=(17,4))
for i in range(16):
    ax = axes.flat[i]
    ax.imshow(imágenes[i], cmap="gray")
    ax.set_title(i)
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.axis('off')
fig.subplots_adjust(hspace=0.2, wspace=0);
```



▼ Entrenamiento

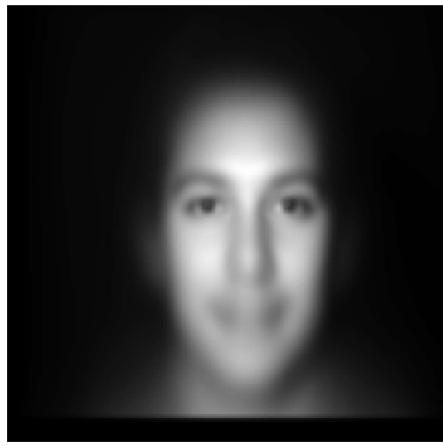
▼ Preprocesamiento de las imágenes

Calculamos una "imagen promedio" de todo el conjunto de imágenes.

```
imagenes_array = np.array(imagenes)
X = imagenes_array.reshape(imagenes_array.shape[0],128*128)
mean_face = np.mean(X, axis=0)
mean_face.shape

(16384,)
```

```
img = np.reshape(mean_face, (128, 128))
plt.imshow(img, cmap="gray")
plt.axis('off')
plt.show()
```



Obtenemos puntos centrados para aplicar PCA. Para esto, restamos la "imagen promedio" de cada imagen.

```
X_centered = X - mean_face.reshape((1,-1))
X_centered.shape
```

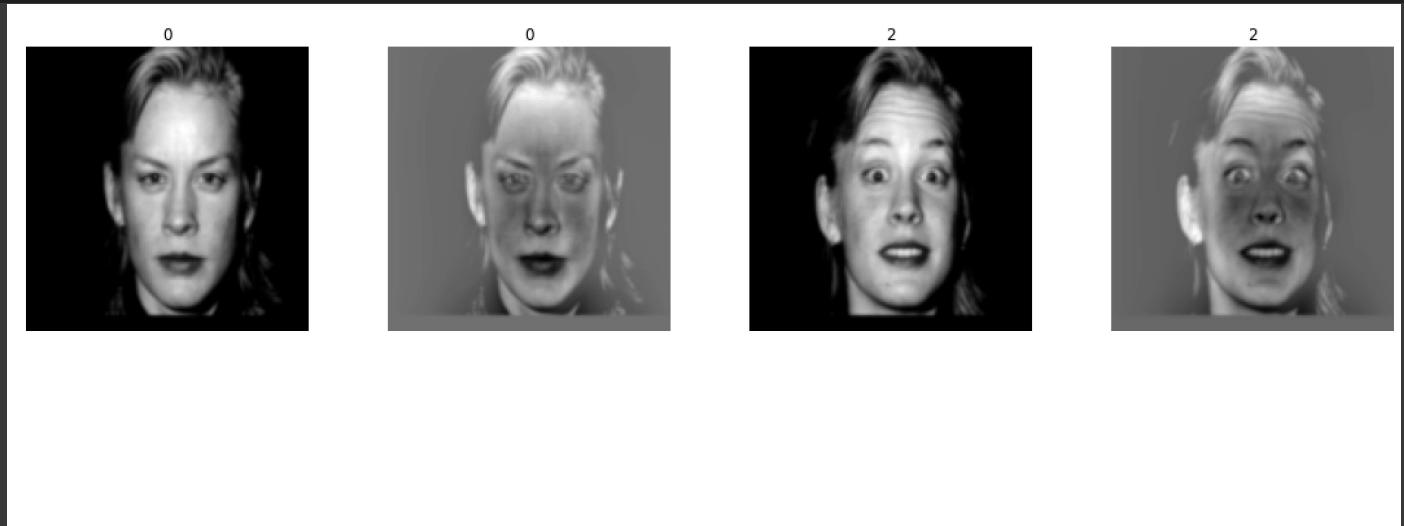
```
(3500, 16384)
```

Mostramos la comparación entre la imagen original y la imagen centrada.

```
fig, axes = plt.subplots(1,4, figsize=(20,4))
for i in range(0,3,2):
    ax = axes.flat[i]
    ax.imshow(imagenes[i], cmap="gray")
    ax.set_title(i)
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.axis('off')

    ax = axes.flat[i+1]
    ax.imshow(X_centered[i].reshape((128,128)), cmap="gray")
    ax.set_title(i)
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.axis('off')

fig.subplots_adjust(hspace=0.2, wspace=0)
plt.show()
```



▼ PCA

$$X = U\Sigma V^T$$

Calculamos la SVD de X y guardamos el resultado en las matrices U, Sigma y VT.

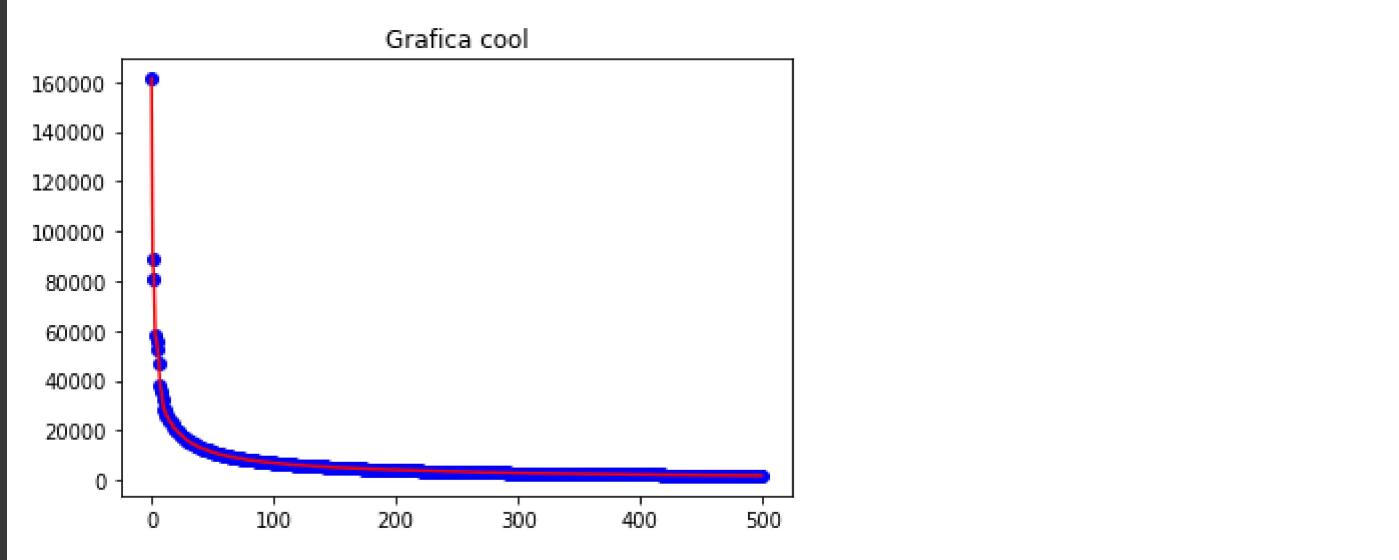
```
U, Sigma, V_T = np.linalg.svd(X_centered, full_matrices=False)
```

```
# Imprimimos las dimensiones de las matrices
print("X:", X_centered.shape)
print("U:", U.shape)
print("Sigma:", Sigma.shape)
print("V^T:", V_T.shape)
```

```
X: (3500, 16384)
U: (3500, 3500)
Sigma: (3500,)
V^T: (3500, 16384)
```

Mostramos y graficamos los valores singulares

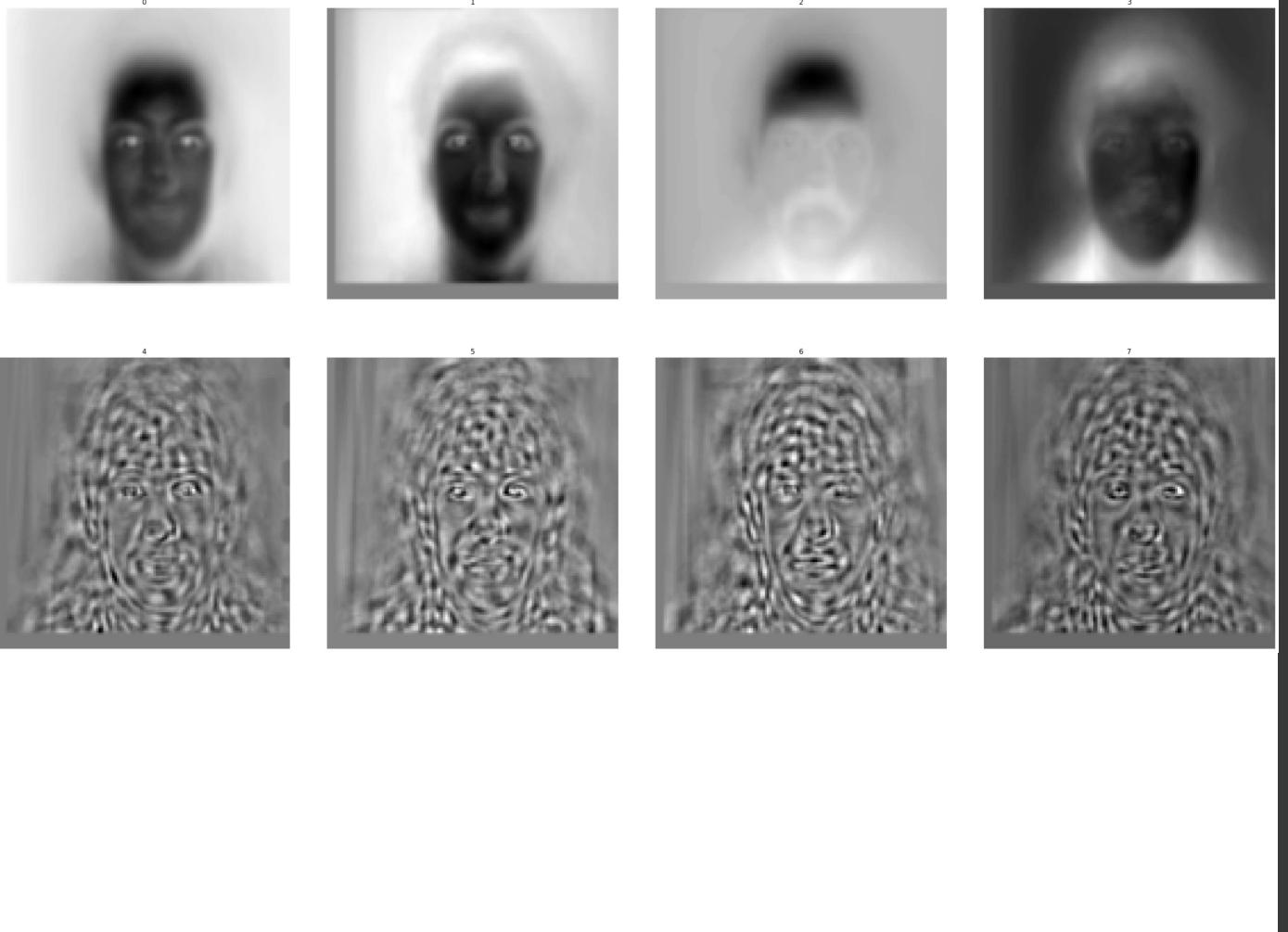
```
plt.plot(Sigma[:500], color = 'r')
plt.scatter(range(500),Sigma[:500],color = 'b')
plt.title("Grafica cool")
plt.show()
```



Graficamos los componentes principales.

```
fig, axes = plt.subplots(2,4, figsize=(40,20))
valores = [0,1,2,3,500,501,502,503]
for i in range(8):
    ax = axes.flat[i]
    ax.imshow(np.reshape(V_T[valores[i]], (128, 128)), cmap="gray")
    ax.set_title(i)
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.axis('off')

fig.subplots_adjust(hspace=0.2, wspace=0)
plt.show()
```



Ahora computamos una nueva matriz Y la cual será la matriz original proyectada en las primeros n componentes principales.

```
num_componentes = 5 # Numero de componentes principales
Y = np.matmul(X, V_T[:num_componentes,:].T)
```

```
Y.shape
```

```
(3500, 5)
```

Graficamos, utilizando la librería [Bokeh](#) de Python para visualizar los puntos de datos faciales proyectados en las dos primeras componentes principales.

```
# Output the visualization directly in the notebook
#output_file('first_glyphs.html', title='First Glyphs')
...
# Create a figure with no toolbar and axis ranges of [0,3]
fig = figure(title='Y',
              plot_height=300, plot_width=300,
              toolbar_location=None)

# Draw the coordinates as circles
fig.circle(x=Y[:,0], y=Y[:,1],
```

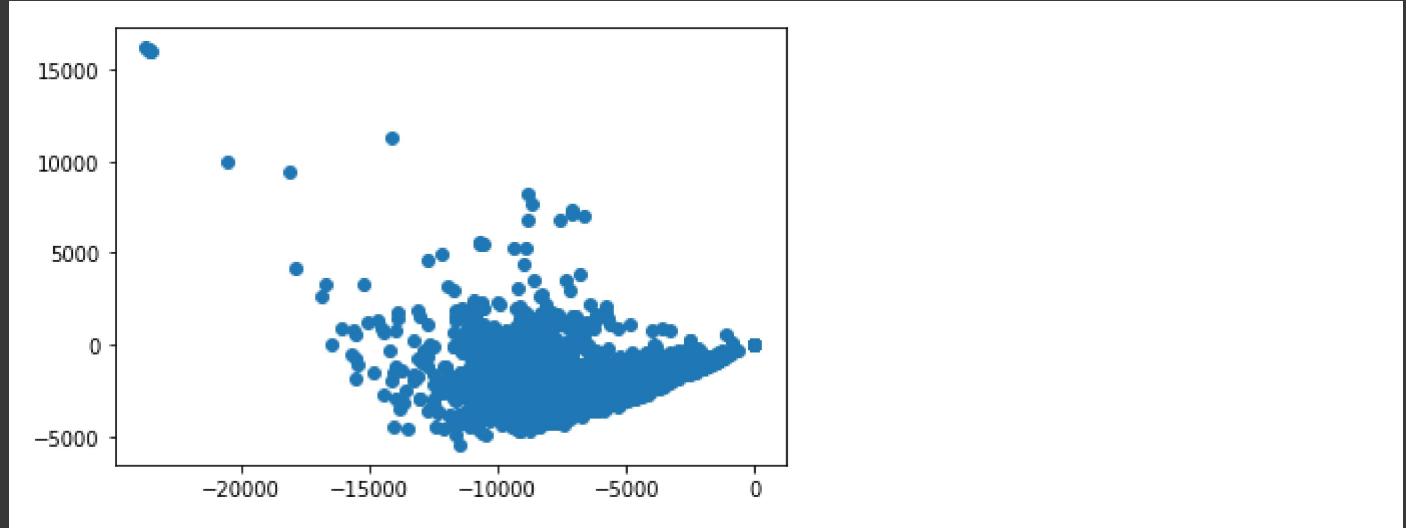
```

        color='blue', size=10, alpha=0.5)

# Show plot
show(fig)
'''

plt.scatter(x=Y[:,0], y=Y[:,1])
plt.show("Graficación de los rostros en los primeros 2 componentes")
plt.show()

```



Aplicamos K-Means sobre los datos proyectados para formar clusters.

```

from sklearn.cluster import KMeans

m = 500
k = 3
kmeans = KMeans(n_clusters=k).fit(Y[:m, :num_componentes])
CentroidesP = kmeans.cluster_centers_

```

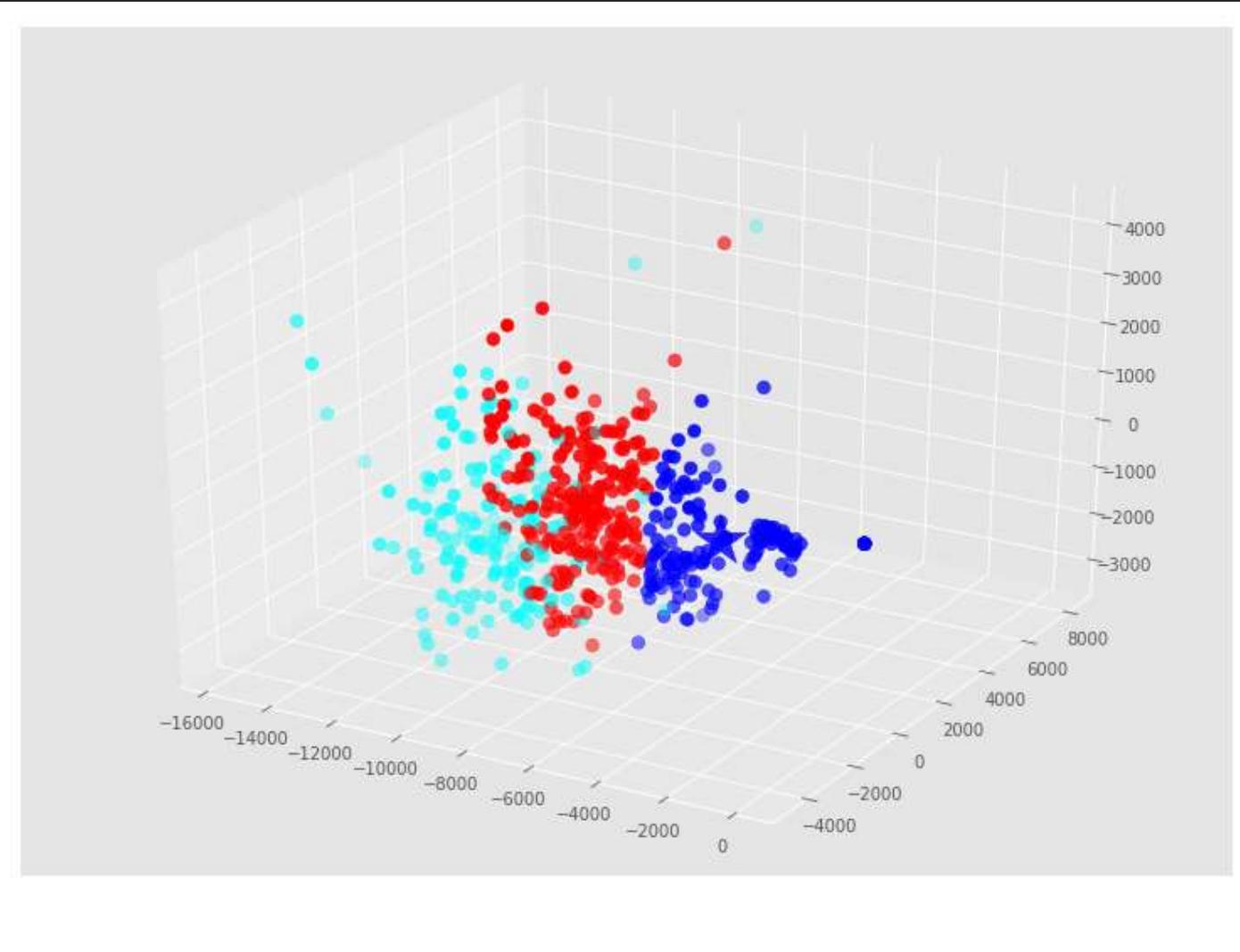
Graficamos los Clusters.

```

# Gráfica de los elementos en 3D y los centros de los clusters
from mpl_toolkits.mplot3d import Axes3D
plt.rcParams['figure.figsize'] = (10, 7)
plt.style.use('ggplot')
colores=['red', 'blue', 'cyan']
asignar=[]
for row in kmeans.labels_:
    asignar.append(colores[row])

fig = plt.figure()
ax = Axes3D(fig)
ax.scatter (Y[:m,0],Y[:m,1],Y[:m,2], marker='o', c=asignar, s=60)
ax.scatter(CentroidesP[:, 0], CentroidesP[:, 1], CentroidesP[:, 2], marker='*', c=colores,
plt.show()

```



▼ 2. Reconstrucción facial

Revisar:

<https://colab.research.google.com/drive/1T3cSvQZjKhh8s3Dxrb9gPCWhwhcz4Bmo#scrollTo=ktT5zKW5WZJK>

```
plt.imshow(np.reshape(X_centered[0], (128, 128)), cmap="gray")
plt.title(0)
plt.xticks([])
plt.yticks([])
plt.axis('off')
plt.show()
```



```
def mulMat(componentes,imagen):
    pc = V_T[:componentes] # (componentes, 12380)
    # imagen (12380, 1)
    # representación (componentes, 1)
    resultado = pc.T@(pc@X_centered[imagen])
    fig, axes = plt.subplots(1,2, figsize=(10,5))

    ax = axes.flat[0]
    ax.imshow(np.reshape(X_centered[imagen]+mean_face, (128, 128)), cmap="gray")
    ax.set_title("real")
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.axis('off')

    ax = axes.flat[1]
    ax.imshow(np.reshape(resultado+mean_face, (128, 128)), cmap="gray")
    ax.set_title("Eigenfaces: "+str(componentes))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.axis('off')

    fig.subplots_adjust(hspace=0.2, wspace=0)
    plt.show()

for i in [1,5,10,20,50,100,200,500,1000,2000]:
    mulMat(i,0)
```

real



Eigenfaces: 1



real



Eigenfaces: 5



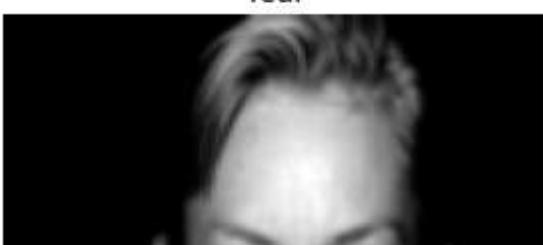
real



Eigenfaces: 10



real



Eigenfaces: 20





real



Eigenfaces: 50



real



Eigenfaces: 100

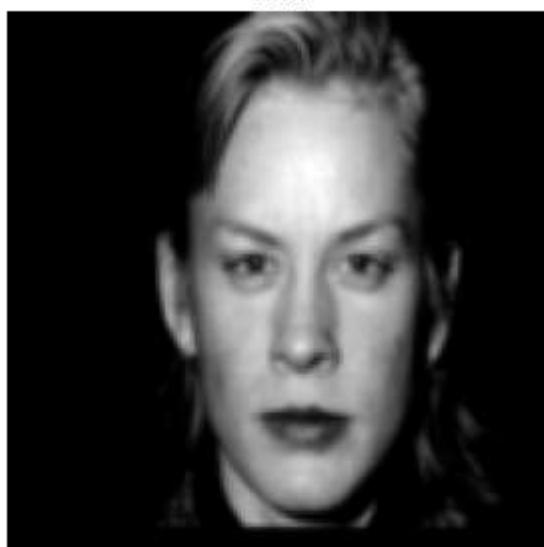
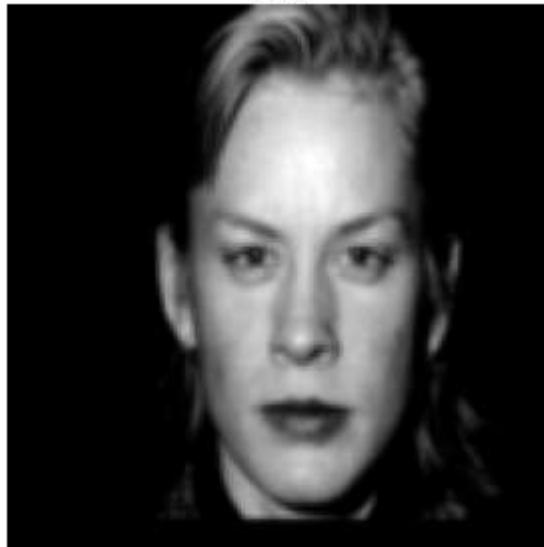


real



Eigenfaces: 200





▼ Generación de rostros

```
print(V_T.shape)
print(V_T.T.shape)
print(X_centered[0].shape)
print(X_centered.shape)
```

```
(3500, 16384)
(16384, 3500)
(16384,)
(3500, 16384)
```

```
A = V_T@X_centered[:3000].T  
print(np.max(A))  
print(np.min(A))
```

```
18152.604778284534  
-17059.660751092866
```

```
mx = np.max(A, axis=0)  
mn = np.min(A, axis=0)
```

```
lista = []  
for i in range(3000):  
    lista.append(np.random.normal(mn[i],mx[i],[1,1])[0])  
ruido = np.array(lista)
```

```
ruido.shape
```

```
(3000, 1)
```

```
def generar(componentes,imagen):  
    pc = V_T[:componentes]  
    resultado = pc.T@imagen[:componentes]
```

```
plt.imshow(np.reshape(resultado+mean_face.reshape(128*128,1), (128, 128)), cmap="gray")  
plt.xticks([])  
plt.yticks([])  
plt.title("Generada "+str(componentes))  
plt.show()
```

```
for i in [1,5,10,20,50,100,200,500,1000,2000]:  
    imagen = ruido[:i]  
    generar(i,imagen)
```

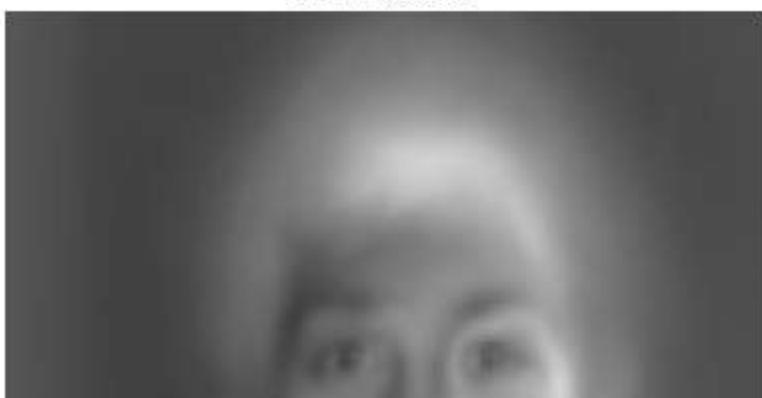
Generada 1



Generada 5



Generada 10





Generada 20



Generada 50



Generada 100



Generada 200



Generada 500





Generada 1000

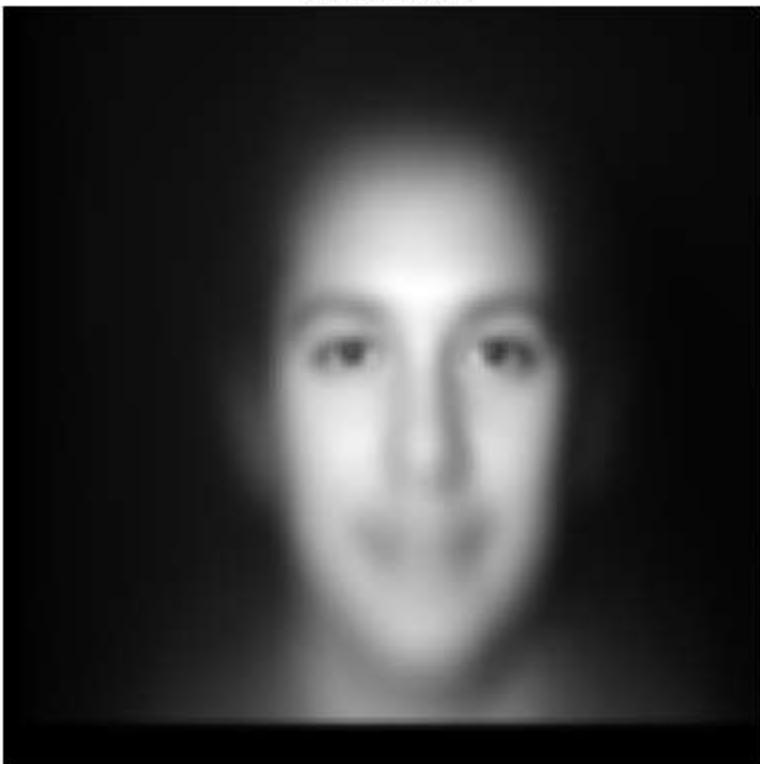


Generada 2000

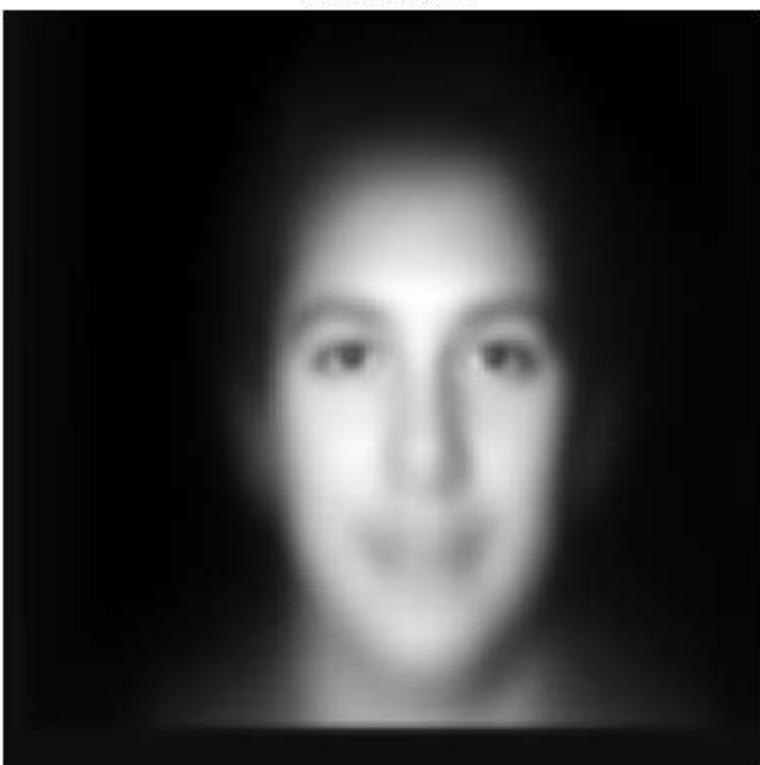


```
for i in range(1,20+1):
    imagen = ruido[:i]
    generar(i,imagen)
```

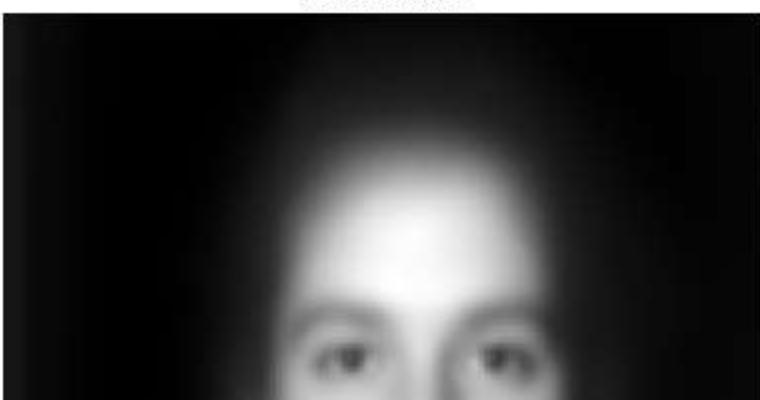
Generada 1



Generada 2

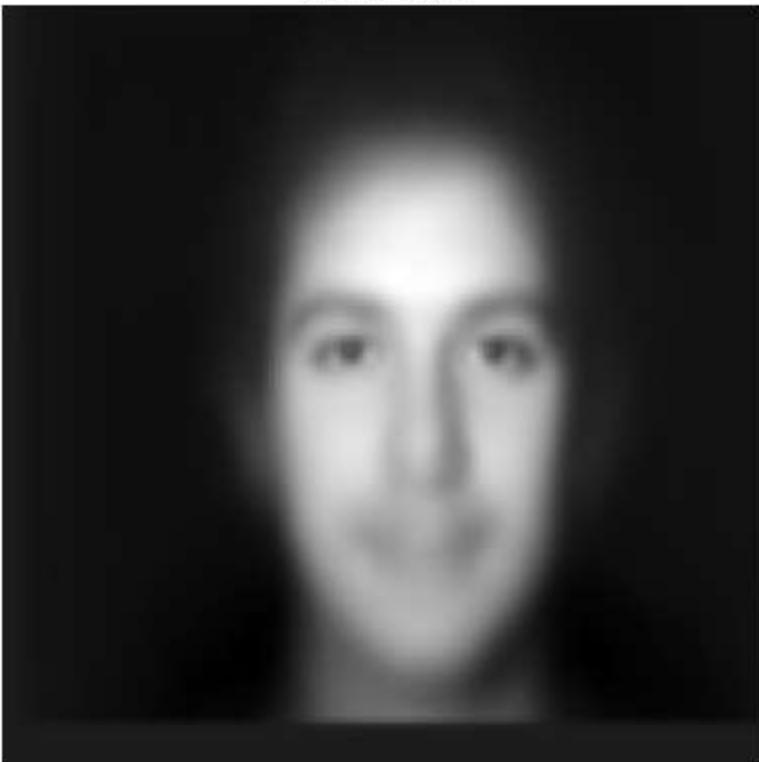


Generada 3





Generada 4



Generada 5



Generada 6



Generada 7



Generada 8





Generada 9



Generada 10



Generada 11





Generada 12



Generada 13





Generada 14



Generada 15



Generada 16





Generada 17



Generada 18



Generada 19



Generada 20



▼ Face Morphing

Referencia

```
def generar_rostros_animacion(V_T, menor, mayor, imagen, inverse=False):
    if inverse:
        A = mayor
        B = menor-1
        paso = -1
    else:
        A = menor+1
        B = mayor+1
        paso = 1

    rostrosL = list()
    for i in range(A,B,paso):
        pc = V_T[:i]
        resultado = pc.T@(pc@imagen)
```

```
frame = np.reshape(resultado+mean_face, (128, 128))
rostrosL.append((frame,i))
```

```
return rostrosL
```

```
rostros = generar_rostros_animacion(V_T, 10, 200, X_centered[0], True)
rostros = rostros + generar_rostros_animacion(V_T, 10, 200, X_centered[1], False)
```

```
fig, ax = plt.subplots()
i=0
def init():
    im = ax.imshow(rostros[0][0], cmap="gray")
    ax.set_title("Eigenfaces: "+str(rostros[0][1]))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.axis('off')
    return im,
def update(frame):
    global i
    i += 1
    im = ax.imshow(rostros[i][0], cmap="gray")
    ax.set_title("Eigenfaces: "+str(rostros[i][1]))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.axis('off')
    return im,
ani = FuncAnimation(fig, update, frames=range(len(rostros)-2),
                    init_func=init, blit=True)
```

Eigenfaces: 200

```
from IPython.display import HTML
HTML(ani.to_html5_video())
```

0:00 / 1:15



