Trabajo Práctico Especial

Reconocimiento Facial con PCA y KPCA

Métodos Numéricos Avanzados

Grupo Probeta - 2020Q2

Integrantes

- *58638 Brula, Matías Alejandro*
- *58322 Maspero, Gerónimo*
- *59266 Neuss, Julián Jorge*
- *59356 Tallar, Julián*
- *58663 Vindis, Davor*
- *59347 Vuoso, Julián Matías*

Índice

1. Introducción

Los avances tecnológicos junto con el aumento de la capacidad de procesamiento y almacenamiento de información surgidos en las últimas décadas han llevado a múltiples organizaciones y sistemas orientados a la persona a elegir al reconocimiento facial como recurso primordial en la seguridad.

Precisamente este proyecto consiste en el diseño y desarrollo de un sistema de reconocimiento facial. Esto implica que dada una imagen con un rostro humano, el sistema deberá determinar a qué persona pertenece de las cargadas en la base de datos del sistema. Puede que no corresponda a ninguno. A su vez, se buscará indicar el nivel de error con el que se obtiene el match.

Para el pre-procesamiento de las caras -las cargadas dentro de la base de datos- utilizaremos dos algoritmos, posiblemente antiguos. Estos algoritmos son los llamados *Principal Components Analysis* (PCA) y *Kernel Principal Components Analysis* (KPCA). Estos junto con un algoritmo para el cálculo de autovectores y autovalores programado exclusivamente por los integrantes del grupo, buscarán hacer el pre-procesamiento completo de las caras disponibles en la base de datos. Ambos algoritmos resuelven el problema de manera distinta pero a su vez similar. Estará en nosotros determinar empíricamente cuál de estos dos y en qué contextos, resuelve de mejor manera (mayor precisión, velocidad, etc.) el problema planteado.

Por el otro lado, para la clasificación y distinción de las imágenes, es decir la decisión de si una cara pertenece o no a las del sistema, se utilizará una librería de redes neuronales que agiliza y facilita el proceso.

2. Análisis Matemático

A continuación nos adentraremos en el análisis exclusivamente matemático involucrado en todo el procesamiento de nuestra tecnología de reconocimiento. Diferenciamos el tratamiento PCA del KPCA, dos posibles resoluciones del procesamiento de la información. Ambos involucran un mismo algoritmo especial de obtención de autovalores y autovectores detallado en este apartado.

2.1. Objetivo

La finalidad del siguiente análisis es sentar las bases para poder realizar el código de cada uno de los algoritmos, viendo qué elementos son necesarios en cada una de las etapas para saber qué hace falta persistir o qué se puede compartir entre cada algoritmo. Finalmente, explicaremos cómo se realiza el cálculo de autovalores y autovectores necesario en cada algoritmo.

Tanto para PCA como para KPCA, se parte de la misma base. Se tienen M imágenes I_1, I_2, … , I_M de caras en blanco y negro del mismo tamaño, donde cada imagen es representada con una matriz de N x N. Estas imágenes servirán como train set para el reconocimiento. Se aplanan dichas matrices, obteniendo un único vector r_i por imagen de N^2.

Se calcula el vector de cara promedio o la media de los M vectores

$$Y = \frac{1}{M} * \sum_{i=1}^{M} r_i \qquad (1)$$

Y luego se la utiliza para normalizar los datos y que la sumatoria de los mismos dé 0. Por tanto, obtenemos los X_i representando cada imagen

$$X_i = r_i - Y \qquad (2)$$

Y armamos una matriz de N^2 x M con los datos normalizados

$$A=[X_1\ X_2\ \dots\ X_M] \qquad (3)$$

Esta matriz se utilizará de base para los cálculos de cada uno de los dos algoritmos que se describen en detalle a continuación. En ambos casos, el objetivo es obtener K vectores llamados “eigenfaces”, que juntos forman una base de un “face space”.

Si uno proyecta una imagen sobre el face space, obtiene un vector de pesos que representará a la imagen de entrada. Este vector tendrá los coeficientes necesarios en una combinación lineal con los eigenfaces como base y lo llamaremos vector “ohm”. Describe la contribución de cada eigenface para representar la imagen de entrada.

Por último, al conjunto de proyecciones de las imágenes de entrenamiento, formado por M vectores columna ohm, lo llamaremos “ohm space”.

2.2. Principal Component Analysis (PCA) [14] y [15]

Para representar el espacio representado por las imágenes, debemos computar la matriz de covarianza entre las mismas. Para ello, podemos aprovechar a la matriz A mencionada anteriormente.

$$Cov = \frac{1}{M} * \sum_{i=1}^{M} X_i X_i^T = A A^T \qquad (4)$$

Podríamos calcular los autovectores u_i de AA^T para describir a la matriz de covarianza. Sin embargo, dicha matriz es de N^2 x N^2 , por lo que es excesivamente grande y no es práctico calcular sus autovectores. Lo conveniente es calcular en su lugar los

autovectores de la matriz A^TA, de M x M , y luego calcular los autovectores de la matriz de covarianza en base a ellos, multiplicando a ambos lados por la matriz A.

En concreto, calculamos los autovalores y autovectores de A^TA

$$A^T A\, v_i = \lambda_i\, v_i \qquad (5)$$

Y calculamos los M mejores autovectores de la matriz de covarianza AA^T , es decir, los asociados a los autovalores más grandes, que se corresponden con los M autovalores de la matriz A^TA. Los autovalores serán los mismos que los de la matriz A^TA.

$$u_i = A\, v_i \qquad (6)$$

Tomamos valores de u_i con $||u_i|| = 1$

De esos M autovectores, nos quedamos únicamente con los k autovectores correspondientes a los mayores k autovalores. Podemos ver que hay una gran diferencia entre los autovalores, siendo innecesario tener la totalidad de los mismos. Con solo el grupo de mayor valor, tenemos casi la totalidad de la información del subespacio y evitamos tener que realizar operaciones con matrices excesivamente grandes.

Así, obtenemos k eigenfaces que representan al eigenspace $V=[u_1\ u_2\ \dots\ u_K]$, siendo cada u_i un vector columna de N^2 posiciones ($V \in N^2$ x k).

A partir del eigenspace, debemos proyectar los M vectores normalizados del train set en este subespacio obtenido. Así, por cada imagen obtenemos su vector ohm, conformando así un ohmspace del train set. Para obtener el ohm de cada imagen, basta con hacer el producto entre el eigenspace (transpuesto) y cada X_i ($\Omega_i \in$ k x 1)

$$\Omega_i = V^T X_i \qquad (7)$$

Así, obtenemos el ohmspace $\Omega=[\Omega_1\ \Omega_2\ \dots\ \Omega_M]$, siendo cada Ω_i un vector columna de k posiciones ($\Omega \in$ k x M).

En caso de querer reconocer una imagen, usamos el mismo procedimiento: aplanamos la imagen, la normalizamos con la media del train set, obteniendo un X_i , y calculamos su vector ohm con (7).

2.3. Kernel Principal Component Analysis (KPCA) [7] y [17]

KPCA es una extensión de PCA basada en métodos de kernel. Se computa PCA en otro espacio vectorial F, de dimensión superior al espacio de entrada. Esto se logra aplicando una función sobre los vectores r_i.

$$: R^{N^2} \rightarrow F$$

Luego, la matriz de covarianza se puede calcular como

$$Cov = \frac{1}{M} * \sum_{i=1}^{M} (X_i)\,(X_i)^T \qquad (8)$$

Sin embargo, en la práctica lo que se elige no es una función φ, sino una función de kernel de las funciones conocidas. Algunos de ellos son el polinómico, el gaussiano, el sigmoid. En nuestro caso, elegimos el kernel polinómico, definido como

$$k(X, Y) = (X^T * Y + 1)^P \qquad (9)$$

Aplicamos la función sobre los datos normalizados, armando una matriz K' de M x M

$$K' = \{k(X_i, X_j)\} = \{(X_i^T * X_j + 1)^p\}\ para\ i,j = 1, 2, ..., M \qquad (10)$$

Para obtener una matriz con normalizados (centrados), sin conocer la función φ, aplicamos la siguiente operación sobre K', obteniendo K

$$K = \{k_{mn} = k'_{mn} - \frac{1}{M} * \sum_{i=1}^{M} k'_{im} - \frac{1}{M} * \sum_{i=1}^{M} k'_{in} + \frac{1}{M^2} * \sum_{i=1}^{M} \sum_{j=1}^{M} k'_{ij}\} \quad para\ m, n = 1, 2, ..., M \qquad (11)$$

Como desconocemos la función φ, el cálculo del face space deberá hacerse en base a la matriz K. Es por ello que utilizaremos la misma para armar las eigenfaces y no la matriz de covarianza, la cual desconocemos.

Calculamos los M autovalores y autovectores de la matriz K

$$Kv_i = \lambda_i v_i \qquad (12)$$

Normalizamos los autovectores en función de cada autovalor obtenido

$$v_i = v_i / (\|v_i\|_2 * \sqrt{M * \lambda_i}) \qquad (13)$$

Los autovalores correspondientes a la matriz de covarianza se pueden calcular en base a los obtenidos en la matriz K, aunque no serán utilizados más adelante.

$$a_i = \lambda_i / M$$

De esos M autovectores, nos quedamos únicamente con los k autovectores correspondientes a los mayores k autovalores. Nuevamente, podemos ver que hay una gran diferencia entre los autovalores, siendo innecesario tener la totalidad de los mismos. Con solo el grupo de mayor valor, tenemos casi la totalidad de la información del subespacio y evitamos tener que realizar operaciones con matrices excesivamente grandes.

Así, obtenemos k eigenfaces que representan al eigenspace V=[u_1 u_2 ... u_K], siendo cada u_i un vector columna de M posiciones (V ϵ M x k).

A partir del eigenspace, debemos proyectar los M vectores normalizados del train set en este subespacio obtenido. Así, por cada imagen obtenemos su vector ohm, conformando así un ohmspace del train set. Para obtener el ohm de cada imagen, basta con hacer el producto entre el eigenspace (transpuesto) y la función kernel aplicada al X_i con la matriz inicial A. (Ω_i ϵ K x 1)

$$\Omega_i = V^T k(X_i, A) \qquad (14)$$

En el caso del ohmspace, como ya aplicamos la función kernel sobre cada par de X_i, podemos directamente usar la matriz K calculada previamente para obtenerlo completo, tomando la transpuesta para que cada columna se relacione con una fila de K.

$$\Omega = V^T K^T \qquad (15)$$

Así, obtenemos el ohmspace Ω=[Ω_1 Ω_2 ... Ω_M], siendo cada Ω_i un vector columna de K posiciones (Ω ϵ K x M).

En caso de querer reconocer una imagen, usamos el mismo procedimiento: aplanamos la imagen, la normalizamos con la media del train set, obteniendo un X_i, y calculamos su vector ohm con (14), ya que, en ese caso, la matriz K no se calculó con el X de entrada.

2.4. Cálculo de Autovalores y Autovectores

Una pieza fundamental en el proceso de PCA y KPCA es la obtención de autovalores y autovectores. Este cálculo se consignó llevar a cabo por algún medio matemático ajeno al automatizado por el método de *eig*. Hay diversos caminos y algoritmos que tomar para llegar a estos resultados y el escogido fue el explicado a continuación.

Para la obtención de autovalores:

1. Se calculó la **descomposición QR** mediante el método de reducción de **Householder** de la matriz A. Este algoritmo va iterando hasta triangularizar una matriz que al ser

similar a AA^T tiene los mismos autovalores. Al tratarse de matrices muy grandes, limitamos las iteraciones a un máximo de 300 (escogido empíricamente).

Luego para el cálculo de autovectores:

2. Para cada valor de S (Los generalizamos como S_i)
   - **2.i.** X = A - S_i * Id ⇒ Siendo Id la matriz identidad de dimensión = A
   - **2.ii.** Aplicamos **RREF** (Reduced Row Echelon Form: Eliminación Gauss-Jordan) a X
   - **2.iii.** Normalizamos lo obtenido dividiendo por su norma 2
3. Luego resta trasponer lo obtenido para que quede en la forma deseada ⇒ $U = U^T$

2.5. Reconocimiento y Clasificación

Para el reconocimiento de caras a partir del eigenspace y el ohmspace obtenidos con el training set, lo primero que debemos hacer es normalizar la imagen de entrada. Para ello, aplicamos lo mencionado en el punto 2.1, obteniendo el X correspondiente a la imagen. Luego, calculamos el ohm de dicho X según el algoritmo utilizado (PCA o KPCA), como mencionamos en las secciones anteriores.

Una vez obtenido el ohm de la imagen de entrada, lo que debemos hacer es comparar de alguna manera este ohm con cada una de las columnas del ohm space para ver a qué cara se acerca más la imagen.

Una manera de hacer esto es buscar una distancia dentro del face space, comparando las distancias cuadradas entre pesos y dividiendo por cada autovalor. Aplicamos la siguiente fórmula, siendo w_i cada elemento de un ohm particular

$$e_r = min\,(i) \,||\, \Omega - \Omega^i ||, \qquad ||\, \Omega - \Omega^i || = \sum_{i=1}^{K} \frac{1}{\lambda i} * (w_i - w_i^k)^{\,2}$$

Si el resultado obtenido de error es menor a un threshold definido, la imagen de entrada es reconocida como la imagen de testing I_i .

Sin embargo, los resultados obtenidos con este método no fueron buenos, en especial aplicando KPCA. Por esta razón, decidimos utilizar una red neuronal de una librería. Sobre esto hablaremos en las próximas secciones, ya que no realizamos el análisis matemático utilizado para construirla.

3. Sistema de Reconocimiento

En este apartado se hará hincapié en el trabajo orientado al desarrollo práctico del análisis previamente detallado. El lenguaje utilizado es *Python*, el mismo fue elegido debido a su simplicidad de uso, velocidad, y esencialmente el soporte y librerías *Open Source*. Al momento de desarrollarlo, se utilizó la versión 3.6.

3.1. Utilización e Interfaces

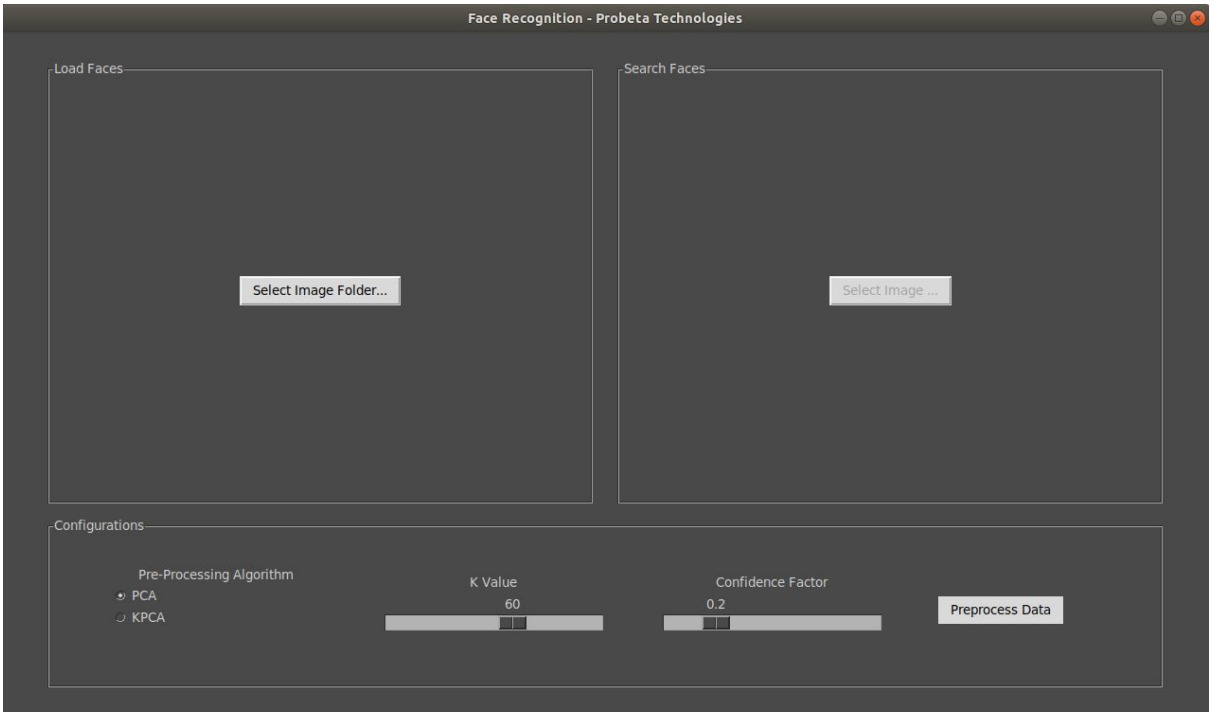
Para poder ejecutar la aplicación, es necesario tener una versión operativa de Python3 (al menos en la versión 3.6) junto con su instalador pip3 asociado a dicha versión de Python. Además, se deben instalar los siguientes módulos e interfaces

- Tkinter → En Ubuntu: sudo apt-get install python3-pil python3-pil.imagetk
  Puede descargarse desde https://www.activestate.com/products/tcl/downloads/
- Pillow → pip3 install Pillow
- NumPy → pip3 install numpy
- OpenCv → pip3 install opencv-python
- TensorFlow → pip3 install tensorflow

Una vez finalizadas las instalaciones, se ejecuta la aplicación corriendo la siguiente línea en la consola, ubicado dentro del directorio con el proyecto

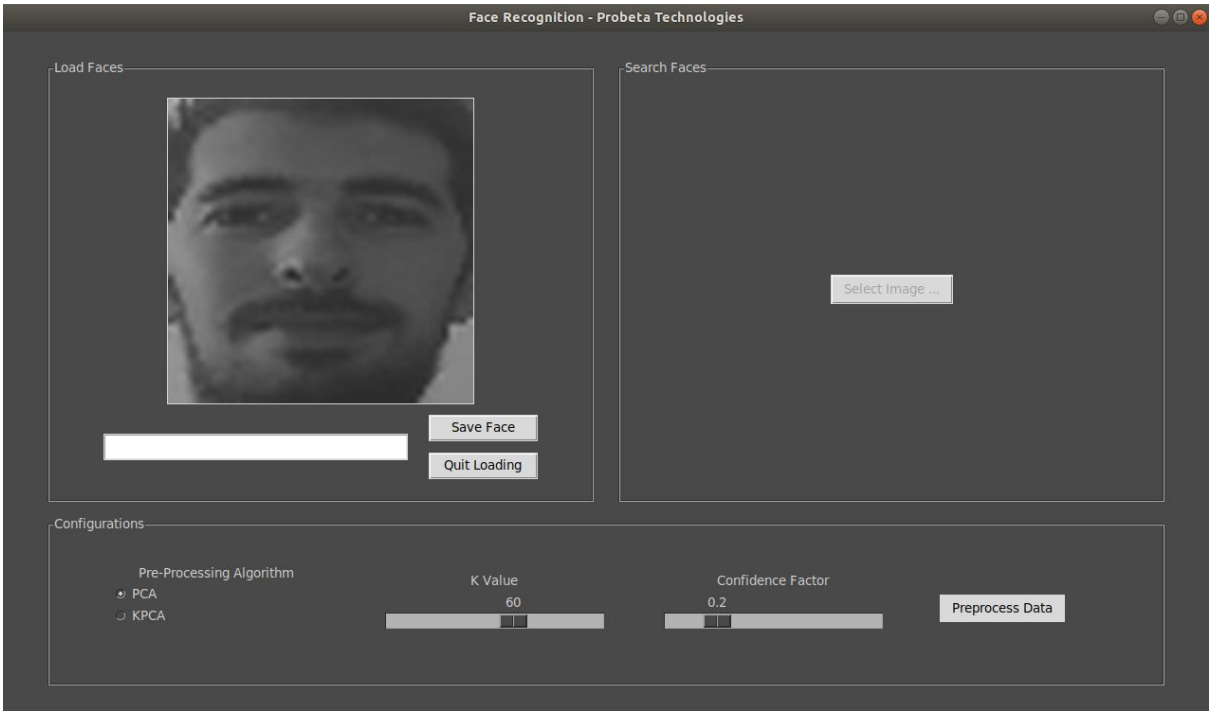
$ python3 application.py

Al ejecutarlo, aparece la siguiente pantalla. Veamos cada uno de los casos de uso.



Cargar imágenes

Para cargar nuevas imágenes, se selecciona “Select Image Folder...” (del lado izquierdo). Aquí, podremos elegir un directorio donde se encuentren las imágenes que deseamos cargar. Una vez elegido uno, debemos elegir qué caras deseamos agregar al sistema. Se ejecuta un “reconocedor de caras” sobre cada imagen (más detalles en la próxima sección) y, por cada cara encontrada, se pregunta quién es la persona encontrada.



Uno puede guardar la cara asociada al nombre ingresado (“Save Face”), saltear una cara (dejando vacío el nombre y tocando Save) o bien terminar la carga (“Quit Loading”) en caso de haber cargado las caras deseadas.

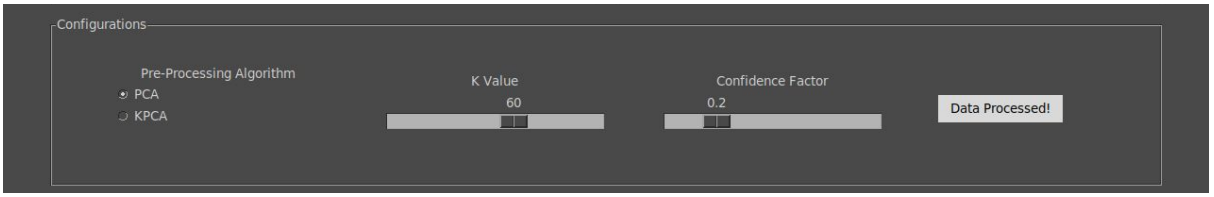
Configuración y pre-procesamiento

En la parte inferior, encontramos un bloque de configuración del sistema. Aquí podremos configurar los distintos parámetros utilizados.

- ➔ El algoritmo utilizado para el pre-procesamiento, ya sea PCA o KPCA
- ➔ El valor de k utilizado en los algoritmos para determinar cuántas eigenfaces se desean utilizar en los cálculos.
- ➔ El valor de confianza utilizado para distinguir caras dentro de una foto, tanto en la carga como en el reconocimiento. Si el valor es muy chico, el algoritmo puede detectar una cara donde no la hay, pero si el valor es muy grande, puede que no detecte alguna cara existente. El valor default de 0,2 es útil para reconocer todas las caras de una imagen, pero puede dar falsos positivos.

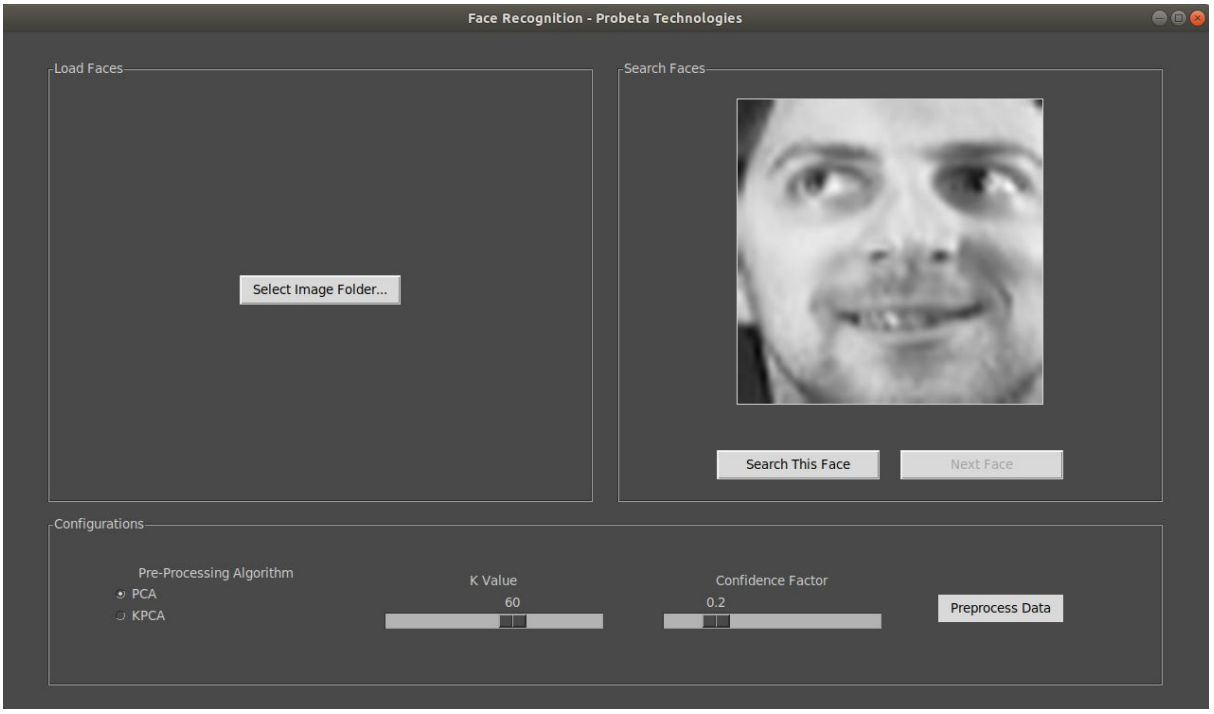
Una vez elegidas las configuraciones anteriores, debemos pre procesar la data ingresada (“Preprocess Data”), tanto las imágenes cargadas como la configuración elegida. Cada vez que se realizan modificaciones sobre las configuraciones o bien se cargan nuevas caras, se debe pre procesar nuevamente para que vean reflejados los cambios en el reconocimiento.

Finalizado el pre procesamiento, vemos el mensaje “Data Processed!” en el botón, informándonos que ha terminado y que podemos proseguir con el reconocimiento.

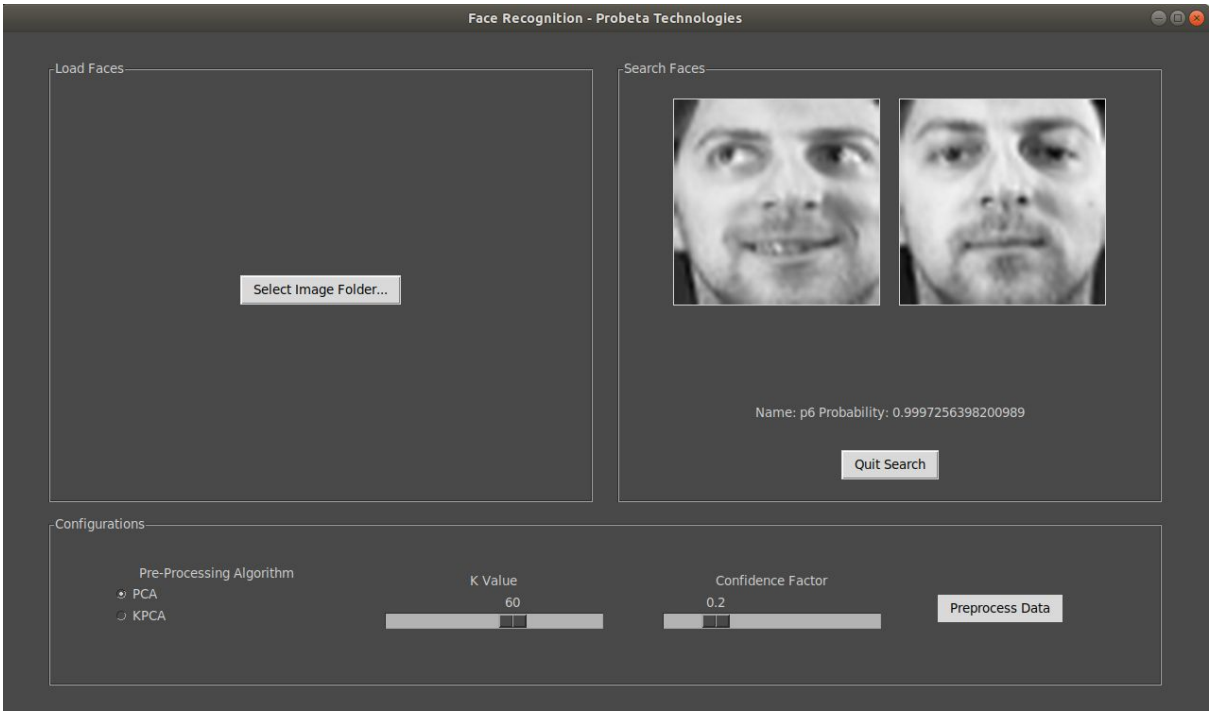


Búsqueda y reconocimiento de imágenes

Para reconocer una cara, se selecciona “Select Image...” (del lado derecho). Aquí, podremos elegir una imagen que se desea reconocer. En caso de que se encuentre más de una cara dentro de la misma imagen, podremos pasar a la siguiente cara (“Next Face”). De lo contrario, podemos buscar la cara seleccionada con “Search This Face”.

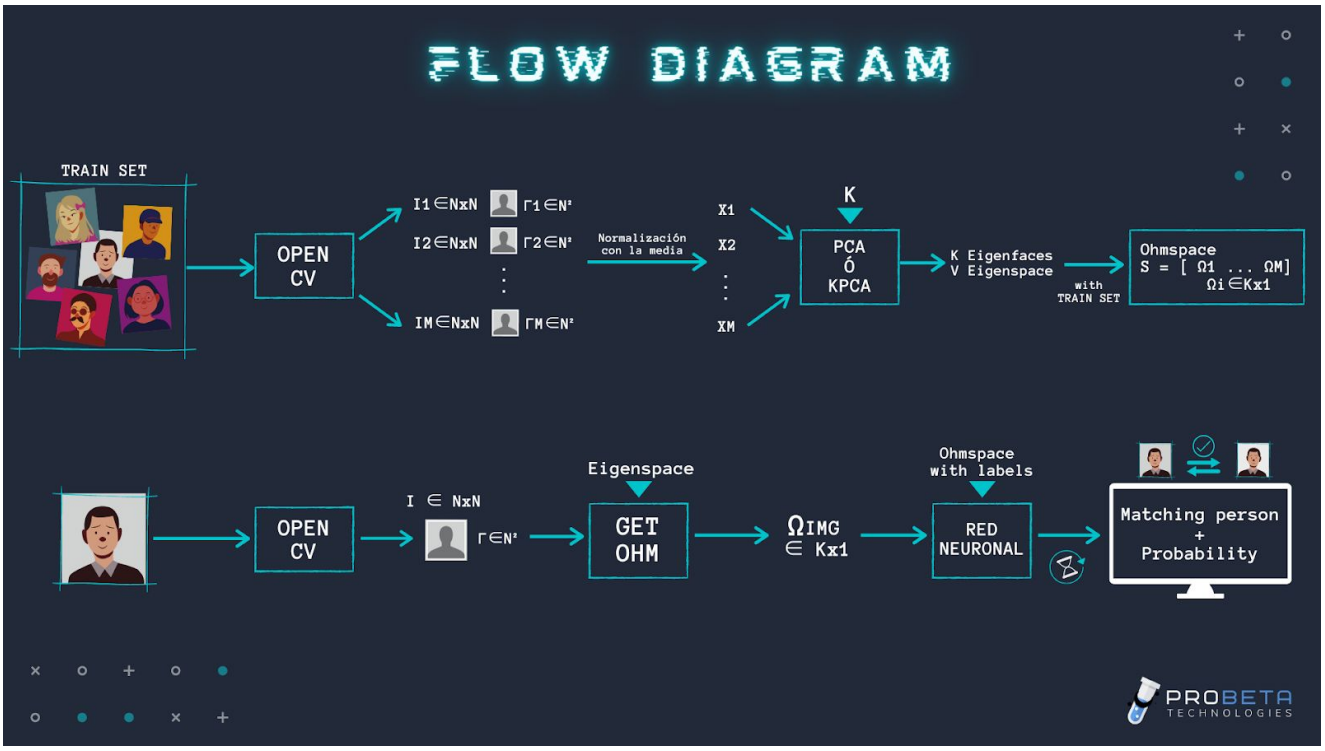


Una vez finalizada la ejecución del algoritmo, el programa mostrará la imagen ingresada junto con una de las imágenes de la persona reconocida. Además, mostrará el nombre de la persona reconocida, junto con la probabilidad de que la predicción sea correcta.



3.2. Implementación Lógica

A continuación, mostramos un diagrama de flujo de la aplicación



La parte superior corresponde a la carga de las imágenes de entrenamiento y el procesamiento de las mismas para armar el eigenspace y el ohmspace. La parte inferior corresponde al reconocimiento de una imagen.

Flujo de entrenamiento

Se cargan M imágenes, que formarán parte del train set. Éstas pasan por un módulo de OpenCV, que toma únicamente las caras de dichas imágenes y unifica su tamaño a NxN (50x50 en nuestro caso). Luego se aplanan dichas matrices, obteniendo vectores de N^2. Tomamos la media de las M imágenes y se la restamos a cada vector, obteniendo X_1 , X_2, ... , X_M.

Los vectores X_i son la entrada del algoritmo de pre procesamiento, ya sea PCA o KPCA. Cada uno opera como se indicó en la sección 2, obteniendo en la salida K eigenfaces (autovectores) que forman un eigenspace que llamamos V. Aplicándolo sobre el train set normalizado, es decir, sobre X_1 , X_2, ... , X_M , obtenemos un ohmspace con las proyecciones de cada X_i sobre el espacio formado por las eigenfaces.

Flujo de reconocimiento

Se carga una imagen, que pasa por un módulo de OpenCV para obtener una cara de la imagen y ajustarla a NxN. Se aplana dicha matriz y se le resta la media, obteniendo así un X correspondiente.

A partir del eigenspace obtenido en el entrenamiento y el vector X propio de la imagen, obtenemos su proyección ohm de la imagen. Y se manda como entrada para la red neuronal, la cual recibe también el ohmspace con los labels correspondientes para entrenar. Como salida de la red, obtenemos un label de la persona que ha matcheado junto con la probabilidad de que la predicción sea correcta.

3.3. Detalles de implementación

A continuación, explicamos brevemente algunos módulos y algunas decisiones tomadas.

Persistencia

Para persistir datos, utilizamos archivos dentro de diferentes directorios dentro de nuestra aplicación. En primer lugar, se guardan los vectores extraídos de cada cara en una carpeta particular para la persona. Además, persistimos la matriz A, los autovalores y autovectores, un índice para saber a quién corresponde cada vector, la media de la matriz y el ohmspace. Para ello, hacemos uso de las funciones de NumPy de guardar sus vectores y matrices en archivos.

OpenCV

Utilizando la librería OpenCV y sus funciones de deep neural network, buscamos dentro de cada imagen dónde hay caras en función del nivel de confianza configurado. Sobre cada cara encontrada, trazamos un rectángulo, lo extraemos de la foto original, la convertimos a escala de grises y la ajustamos a un tamaño de 50x50, ya que el algoritmo espera que todas las fotos sean del mismo tamaño. En este caso, el N elegido es 50.

Interfaz gráfica

Utilizamos la interfaz Tkinter junto con Pillow para armar la interfaz gráfica.

NumPy

Utilizamos la librería NumPy para realizar los cálculos con matrices y vectores.

Red Neuronal

Como mencionamos anteriormente, utilizamos una red neuronal para clasificar las imágenes y reconocer de quién es la cara buscada. Para ello, utilizamos TensorFlow, en particular las funciones de la librería Keras. Procedemos a describir la composición de la red.

Utilizamos un modelo secuencial con 2 capas ocultas, sumadas a la capa de salida. La entrada de esta red serán K neuronas, siendo K la cantidad de eigenfaces en el eigenspace. Estos serán los coeficientes para combinar linealmente las caras. Luego, colocamos dos capas relu (rectified linear unit): la primera con K * ⅔ + #labels y la segunda con la mitad de neuronas de la primera.

El primer valor surge de una “rule of thumb” de redes neuronales, entendiendo que dicho valor es una buena base para una capa oculta. El segundo tiene el objetivo de que el cambio en la cantidad de neuronas no sea tan abrupto al pasar a la salida, teniendo una capa intermedia con un valor intermedio de neuronas.

Por último, utilizamos una capa softmax con una cantidad de neuronas igual a la #labels (es decir, a la cantidad de personas diferentes cargadas). Este tipo de capa se utiliza para normalizar la salida de una red neuronal a una distribución probabilística sobre las clases de salida, en este caso el nombre de la persona.

Luego de crear la red, la compilamos y la ajustamos sobre el eigenspace utilizando un valor de 1500 iteraciones (fijado heurísticamente). Todas las entradas (el eigenspace y un ohm particular) son normalizadas antes de ingresar a la red.

Cuando se desea realizar una predicción a partir de un ohm, se crea la red con el eigenspace, los labels y la cantidad de personas, y luego se realiza una predicción sobre el ohm en cuestión. De dicha predicción, nos quedamos únicamente con aquella que tiene la mayor probabilidad y devolvemos el label y la probabilidad de que la predicción sea correcta.

3.4. Obtención de Autovalores y Autovectores

Si bien la obtención de los autovalores y autovectores fundamental para el procesamiento de las imágenes fue analíticamente explicado en el apartado anterior, veamos brevemente cómo fue llevada y bajada esa misma lógica al código en Python.
En primer lugar, los autovalores. Dada una matriz A:

1. Q, R ← Descomposición de A por el método de Householder tomado del paper [11]
  1.i. S ← Autovalores ordenados descendentemente por valor absoluto
2. Para cada valor de S (Los generalizamos como S_i)
  2.i. X ← A - S_i * Id → Siendo Id la matriz identidad de dimensión = A
  2.ii. *Ared* ← **RREF** de X [Algoritmo explicado al finalizar estos pasos. (*)]
  2.iii. Obtenemos los primeros N-1 elementos de la última columna de Ared, cambiandoles el signo, y los insertamos en un nuevo vector **R.**
  R ← valores
  2.iv. **R** ← 1 Insertamos un 1 en el último valor del vector **R**
  2.v. **R** ← R / ||R|| ⇒ Normalizamos todo R dividiendo por su norma 2
  2.vi. Insertamos **R** en matriz de salida **U**
3. Una vez completado el ciclo de **2.** para cada S_i, tendremos en U una matriz con todos los autovectores, uno por cada autovalor. Luego resta trasponer U para que quede en la forma deseada ⇒ U ← U^T

(*) **RREF:** El objetivo de este método es obtener la matriz en forma escalonada reducida (método Gauss-Jordan) de la matriz ampliada.

El algoritmo implementado en el código opera de la siguiente forma:

1. Se sitúa en la columna distinta de cero extrema izquierda.
2. Si la primera fila tiene un cero en esta columna, intercambia con otra que no lo tenga.
3. Obtiene ceros debajo de este elemento delantero, sumando múltiplos adecuados de la fila superior a las filas debajo de ella.
4. Cubre la fila superior y repite el proceso anterior con la submatriz restante. Repite a su vez con el resto de las filas (aquí la matriz se encuentra en forma escalonada).
5. Comenzando con la última fila distinta de cero, avanza hacia arriba: para cada fila obtiene un 1 delantero e introduce ceros arriba de éste sumando múltiplos correspondientes a las filas correspondientes.

4. Pruebas

Esta sección no se debe dejar de lado, ya que la aplicación real de los algoritmos no suele ser perfecta. En esta, tomamos un dataset extraído del sitio

https://github.com/lloydmeta/Olivetti-PNG

Para realizar pruebas para cada uno de los algoritmos, buscando hacer comparaciones heurísticas entre ambos.

El dataset está compuesto por 400 fotos de caras estandarizadas para pruebas de reconocimiento facial. Esto significa que las caras son del mismo tamaño, están centradas y en blanco y negro y tienen distintos perfiles y expresiones faciales por cada persona. Son 40 personas, con 10 fotos por persona.

Las siguientes son las primeras 100 fotos del dataset

Para realizar las pruebas, tomamos las 10 fotos de 26 personas sobre las 40 disponibles y dedicamos 8 fotos de cada persona para entrenamiento y 2 fotos para reconocimiento. En total, había cargadas 222 fotos, 208 del dataset estándar más otras 14 fotos diversas de los integrantes del grupo.

En primer lugar, determinamos el valor de K, de iteraciones en el cálculo de autovalores y de epochs (para la red neuronal) de forma heurística. En el caso del valor de K y para PCA, nos basamos en el porcentaje de información promedio contenido en los primeros n autovectores para tomar la decisión. Dados los autovalores y autovectores normalizados y ordenados de mayor a menor por su autovalor, realizamos el siguiente cálculo:

```
cov = np.dot(a, np.transpose(a))        # Calculamos la matriz de Cov
aux = np.dot(u, (np.diag(s) ** 0.5))
for i in range(0, len(u[0])):           # len(u[0]) = #Autovectores
   b = np.dot(aux[:,0:i], np.transpose(aux[:,0:i])) # Tomo i autovec
   sum = 0
   for j in range(0, len(u[0])):        # len(u[0]) = #Autovectores
      sum += b[j,j]/cov[j,j]   # Acumulamos la razon entre diagonales
       print("Primeros ", i + 1, " autovectores, % de info promedio:",
sum / len(u[0]))
```

Para la cantidad de imágenes de nuestro dataset, observamos que, tomando **K=60**, obtenemos buenos resultados y un porcentaje de información promedio del 97,45%.

Para KPCA, no podemos realizar el mismo cálculo ya que, como explicamos anteriormente, no conocemos la función φ como para poder calcular la matriz de covarianza. Por lo tanto, y dados los resultados obtenidos, tomamos el mismo valor de K.

Para determinar la cantidad de iteraciones, tanto para el cálculo manual de autovalores como para el entrenamiento de la red neuronal, la prueba fue completamente empírica. Se realizaron pruebas para distintos valores, tomando aquellos con los que obtuvimos mejores resultados. Así, tomamos un límite de **300 iteraciones** para el cálculo de autovalores y **1500 iteraciones** (epochs) para la red neuronal.

Además, en KPCA debíamos determinar el grado de la función polinómica utilizada como kernel. Se realizaron pruebas con varios valores (mencionamos 1 y 2 más adelante), llegando a la conclusión de que los mejores resultados se obtienen con grado **p=1**.

Por último, para mantener consistencia en los resultados, las mismas se realizarán un valor discreto de veces e intentando mantener las condiciones externas constantes (la misma computadora, sin otros programas corriendo al mismo tiempo, etc.).

4.1. PCA

Dadas las condiciones mencionadas anteriormente y habiendo cargado las 222 fotos mencionadas, corremos el programa para reconocer las 52 fotos de prueba restantes de las estandarizadas (2 de cada una de las 26 personas). Así, obtuvimos un promedio de 44 predicciones correctas y 8 incorrectas. Esto representa un **84,61% de efectividad**.

También analizamos el promedio de las probabilidades obtenidas, tanto en caso de acierto como en caso de error. Tomando únicamente los **casos de acierto** obtenemos un **95,36%** de probabilidad promedio, mientras que si tomamos sólo los **casos de error** obtenemos un **63,09%**.

Analizando un poco estos resultados, creemos que la efectividad general es muy buena, a pesar de que se esperan que ésta disminuya en caso de utilizar imágenes no estandarizadas o distintas cantidades de imágenes por persona. La probabilidad promedio en caso de acierto es muy alta, lo cual es positivo, y la de error, aún no siendo tan baja (es superior al 50%), es bastante menor a la anterior. Por lo tanto, no sería difícil notar en muchas de estas predicciones que son incorrectas, aunque algunas de ellas sí tienen probabilidades altas.

4.2. KPCA

Dadas las condiciones mencionadas anteriormente y habiendo cargado las 222 fotos mencionadas, corremos el programa para reconocer las 52 fotos de prueba restantes de las estandarizadas (2 de cada una de las 26 personas). En primer lugar, realizamos las pruebas utilizando una función polinómica de **grado 1**. Así, obtuvimos un promedio de 46 predicciones correctas y 6 incorrectas. Esto representa un **88,46% de efectividad**.

También analizamos el promedio de las probabilidades obtenidas, tanto en caso de acierto como en caso de error. Tomando únicamente los **casos de acierto** obtenemos un **93,77%** de probabilidad promedio, mientras que si tomamos sólo los **casos de error** obtenemos un **77,95%**.

Nuevamente, la efectividad general es muy buena (y superior a PCA), a pesar de que se esperan que ésta disminuya en caso de utilizar imágenes no estandarizadas o distintas cantidades de imágenes por persona. La probabilidad promedio en caso de acierto es alta,

pero la de error no es tan baja, aún siendo distante a la de acierto. Por lo tanto, en este caso sería más difícil distinguir las predicciones correctas e incorrectas.

Tomando un kernel polinómico de **grado 2**, obtuvimos una efectividad de tan solo el 42,31%. Y tomando grados mayores, la efectividad bajaba aún más. Es por ello que decidimos utilizar el de grado 1.

4.3. Comparación y Conclusiones

A partir de los resultados anteriores y las distintas pruebas realizadas, podemos extraer varias conclusiones.

En primer lugar, podemos ver que la efectividad obtenida con KPCA es levemente superior a la obtenida con PCA, aunque con valores de probabilidad de acierto peores. Como era esperado, KPCA dio mejores resultados que PCA, pero no hubo tanta diferencia. Creemos que parte de esto se debe a haber utilizado la misma red neuronal con las mismas configuraciones en ambos casos, cuando podríamos ajustar algún valor exclusivamente para KPCA.

Por otro lado, un problema recurrente en la elección de parámetros fue el error de precisión en los cálculos. En el caso de los autovalores, notamos que aquellos de mayor valor eran varios órdenes superiores a los que se encontraban más adelante en el vector ordenado. El hecho de estar realizando varias cuentas iterativas con valores tan pequeños provoca un error mayor en las cuentas.

En el caso del parámetro K, notamos que, al subir su valor, aumentaba la precisión del sistema (aunque cada vez menos, ya que el porcentaje de información ya era elevado). Sin embargo, también comenzábamos a tomar autovalores muy pequeños en comparación con los primeros, por lo que no era conveniente aumentarlo más (lo que también traería matrices de mayor dimensión).

Con respecto a la cantidad de iteraciones para el cálculo de autovalores, tomamos dicho valor ya que, de ser más bajo, la diferencia con respecto al cálculo automático era más significativa. Un valor alto de iteraciones también ralentiza el pre-procesamiento (aunque también dependerá de la cantidad de fotos cargadas).

En cuanto a la cantidad de epochs, hay que ser cuidadosos con la elección de dicho valor ya que, de ser muy alto, puede producirse “overfitting”. Esto es, ajustarse tanto al dataset de entrenamiento que cualquier foto fuera del mismo es predecida erróneamente. Además, cuanto mayor es la cantidad, más lento es el reconocimiento. El valor elegido nos daba una mejora en la precisión, pero manteniendo buenos resultados fuera del training set.

Por último, al elegir el grado de la función kernel polinómica, notamos que subir el valor de p aumentaba enormemente la efectividad del sistema. Creemos que esto está relacionado con lo mencionado anteriormente de los valores pequeños. Al elevarlos a una potencia mayor, el error en los mismos crece con el mismo grado del polinomio. Hicimos algunas pruebas con grados mayores (como 4), pero la probabilidad de acierto en las predicciones era menor al 10%, por lo que las predicciones tenían efectividad casi nula. Por ello, terminamos fijando el grado en 1.

5. Conclusiones y posibles mejoras

En primer lugar, resaltar la importancia del acceso a la información de los papers otorgados por la cátedra, específicamente a la librería de redes neuronales. Todo el contenido y código otorgado compuso la base de la totalidad del desarrollo de este proyecto reduciendo enormemente el margen de error. Creemos que en ausencia de la librería de redes dispuesta, el sistema hubiera llegado a un éxito muy limitado.

Por otro lado, el diseño de un algoritmo matemático de obtención de autovalores y autovectores trajo consigo algunas complicaciones. En un principio nos encontramos decididos a afrontar el algoritmo por un camino que parecía perfecto para el objetivo. Luego de pruebas en matrices simples, erróneamente asumimos su correcto funcionamiento y procedimos a concretar las demás etapas del proyecto. Luego, a la hora de las pruebas del sistema completo con todos sus módulos, nos encontramos con ciertos errores y una muy alta ineficiencia y distorsión de resultados. Es así que finalmente decidimos cambiar completamente el algoritmo hasta llegar al propuesto en esta entrega. Éste terminó por convencernos con su rendimiento y funcionamiento.

Con el sistema funcionando correctamente, se nos ocurrieron algunas potenciales mejoras del sistema desarrollado.

- Es el caso de un pre-procesamiento de las imágenes tanto de carga como de búsqueda en pos de evitar que los diferentes contextos terminen por afectar en el óptimo funcionamiento del mismo.
- Asimismo, si se implementaran optimizaciones en los cálculos que involucran a las matrices en el procesamiento, el algoritmo sin dudas reduciría su tiempo de trabajo.
- Por otra parte, la implementación de una base de datos aceleraría la búsqueda y cálculo para un dataset de mayor tamaño.
- Por último, mencionar que la interacción con el usuario no fue la prioridad en este proyecto. Es así que, a pesar de haberse conformado un sistema con una -a nuestro criterio- correcta usabilidad, existen cientos de variantes y posibles mejoras en la interfaz gráfica planteada.

6. Fuentes Utilizadas

Links de consulta

[1] https://stackoverflow.com/
[2] https://www.geeksforgeeks.org/
[3] https://www.math.purdue.edu/~shao92/documents/Algorithm%20REF.pdf
[4] https://medium.com/@louisdevitry
[5] https://stattrek.com/matrix-algebra/echelon-transform.aspx
[6] https://www.youtube.com/watch?v=9oSkUej63yk&ab_channel=CodeEmporium
[7] http://fourier.eng.hmc.edu/e161/lectures/kernelPCA/node4.html
[8] https://namanuiuc.github.io/assets/projects/KPCA/slides.pdf
[9] https://www.youtube.com/watch?v=wQ8BIBpya2k&ab_channel=sentdex
[10] https://www.tensorflow.org/guide/keras/sequential_model
[11] Householder QR
https://www.cs.cornell.edu/~bindel/class/cs6210-f09/lec18.pdf
[12] https://www.pyimagesearch.com/2018/02/26/face-detection-with-opencv-and-deep-learning/

Papers y documentos académicos

[13] https://www.face-rec.org/algorithms/PCA/jcn.pdf
[14] M. A. Turk and A. P. Pentland. Face recognition using eigenfaces. In Proceedings. 1991 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pages 586–591, Jun 1991.
https://www.cs.utexas.edu/~grauman/courses/spring2007/395T/papers/turk_pentland_1991.pdf
[15] M. Turk and A. Pentland. Eigenfaces for Recognition, Journal of Cognitive Neuroscience, March 1991
http://www.vision.jhu.edu/teaching/vision08/Handouts/case_study_pca1.pdf
[16] https://eprints.lancs.ac.uk/id/eprint/69825/1/face.pdf
[17] Hala M. Ebied. Feature Extraction Using PCA and Kernel-PCA for Face Recognition, IEEE 8th International Conference on Informatics and Systems, 2012
https://ieeexplore.ieee.org/abstract/document/6236591