DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

16 de julio de 2018 — Métodos numéricos

Integrante	LU	Correo electrónico
Luis Arroyo	913/13	luis.arroyo.90@gmail.com
Carlos Humpiri	942/14	jhumpiri1599@gmail.com
Matías Millassón	131/13	matiasmillasson@gmail.com
Eric Peker	548/13	epeker.135@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

En este trabajo práctico vamos a ayudar a las autoridades nacionales con la realización de un software de reconocimiento de personas en base a una foto. Para lograr este cometido nos basamos en técnicas de inteligencia Artificial, como por ejemplo kNN y PCA. Utilizamos PCA para reconocer los rasgos más característicos de las personas y kNN para decidir a qué persona corresponde cierta foto. Para mejorar el reconocimiento también se verificaron todas las respuestas con K-fold cross validation. Por último realizamos varios experimentos para medir, entre otras cosas, la efectividad de los métodos estudiados.
En principio nuestro trabajo se enfoca en imágenes muy pequeñas (menos de cuatro millones de píxeles). Pero creemos que con leves modificaciones la metodología desarrollada podría funcionar apropiadamente para imágenes más grandes.

Palabras claves: reconocimiento de caras, kNN, PCA, K-fold cross validation

1.1. Resolución del problema

El problema a resolver es que tenemos una base de datos de caras conocidas que luego nos servirá de entrenamiento para poder reconocer otras instancias de esas caras que no están presente en la base de datos que usamos de entrenamiento.
Ahora presentaremos técnicas de como entrenar nuestra base datos de entrenamiento y como clasificar nuestra imagen por medio de un método llamado kNN (k nearest neighbours). Como analizar todas las imágenes puede llegar a ser muy pesado se puede realizar previamente un análisis de componentes principales de la base de entrenamiento por medio del método PCA (Principal component analysis) para reducir la dimensionalidad del problema.

1.1.1. Método de kNN

El método de KNN se trata del algoritmo de k vecinos mas cercanos. Éste algoritmo considera a cada objeto de la base de entrenamiento como un punto euclídeo m-dimensional, para el cual se conoce a qué clase corresponde (en nuestro caso serian los Ids de las imágenes), para luego dada una nueva imagen, asociarle la clase correspondiente a los puntos más cercanos de la base de datos.

1.1.1.1 Procedimiento de k vecinos más cercanos

a) Se define una base de datos de entrenamiento como el conjunto D $= \{x_i : i = 1, ..., n\}$.

b) Luego, se define m como el numero total de píxeles de la imagen i-ésima almacenada por filas y representada como un vector $x_i \in \mathbb{R}^m$.

c) De esta forma dada una nueva imagen $x \in \mathbb{R}^m$, tal que $x \notin D$, para clasificar simplemente se busca el subconjunto de los k vectores $\{x_i\} \subseteq D$ más cercanos a x, y se le asigna la clase que posea el mayor número de repeticiones dentro de ese subconjunto es decir, la moda.

Veamos un ejemplo en la **figura 1**:

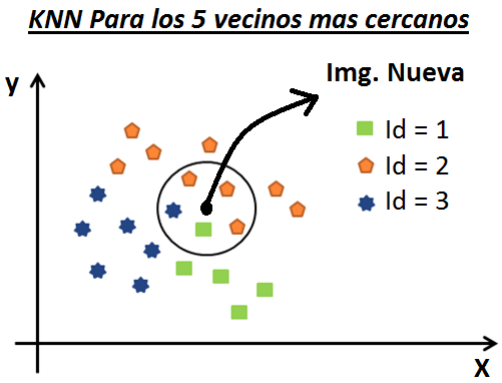


Figura 1: Forma Gráfica del algoritmo de KNN, para los 5 elementos mas cercanos. En este caso se clasificaría como id = 2

En este caso, de los cinco vecinos más cercanos hay tres que corresponden a la clase 2, uno a la clase 1 y el restante a la clase 3. Por lo tanto la imagen nueva será clasificada como una imagen perteneciente a la clase 2.

El algoritmo del vecino más cercano es muy sensible a la dimensión de los objetos y a la variación de la intensidad de las imágenes. Es por eso, que las imágenes dentro de la base de datos D se suelen procesar para lidiar con estos problemas. Para poder eliminar estas variaciones de dimensionalidad se propone el método de PCA que explicaremos a continuación.

1.1.2. Método de PCA

El método de PCA construye una trasformación lineal que escoge un nuevo sistema de coordenadas para el conjunto original de la base de datos, en el cual la varianza de mayor tamaño del conjunto de datos es capturada en el primer eje (llamado el Primer componente Principal), la segunda varianza mas grande es el segundo eje, y así sucesivamente. Para construir esta transformación lineal debe construirse la matriz de covarianza. Debido que esta matriz de covarianza es simétrica sabemos que la podemos diagonalizar ortogonalmente, entonces existe una base ortonormal de autovectores. Por ende, podemos construir una matriz V que tiene como columnas los autovectores de la matriz de covarianza, es por medio de esta que se hace la transformación de cambio de base. Pero el objetivo de hacer PCA es reducir la dimensionalidad del problema, por lo tanto hay una variable que dice cuántos autovectores van a ser considerados, llamémosla α. Como estos autovectores están asociados a los autovalores de la matriz de covarianza, solo se elijen los α autovalores mas grandes en módulo con sus correspondiente autovectores.

1.1.2.1 Procedimiento PCA o análisis de componentes principales

Sea $\mu = (x_1 + ... + x_n)/n$ el promedio de las imágenes $D = \{x_i : i = 1, ..., n\}$ tal que $x_i \in \mathbb{R}^m$. Definimos $X \in \mathbb{R}^{nxm}$ como la matriz que contiene en la i-ésima fila al vector $(x_i - \mu)^t/\sqrt{n-1}$. La matriz de covarianza de la muestra X se define como $M = XX^t$.
Siendo v_j el autovalor de M asociado al j-ésimo autovalor, al ser ordenados por su valor absoluto, definimos para $i = 1, ..., n$ la transformación característica de x_i como el vector $tc(x_i) = (v_1x_i, v_2x_i, ..., v_\alpha x_i) \in \mathbb{R}^\alpha$, donde $\alpha \in \{1, ..., m\}$ es un parámetro de la implementación. Este proceso corresponde a extraer las α primeras componentes principales de cada imagen. La idea es que $tc(x_i)$ resuma la información mas relevante de la imagen descartando las dimensiones menos significativas. Vale aclarar que $tc(x_i)$ en forma matricial seria $tc(x_i) = V^t x_i$, donde la matriz V tiene los v_1,v_2,...., v_α como columnas, entonces V$\in \mathbb{R}^{mx\alpha}$.

Para que se entienda mejor se va a hacer un gráfico en $\mathbb{R}^2$ de como se puede observar el cambio de base por medio de los autovectores, y poder analizar de que la mayor información esta en las componentes principales, que van a ser los autovectores asociados a los autovalores mas grandes en modulo.

Tenemos una base de datos que tiene solo dos dimensiones, vamos a poner que cada punto que dibujamos en $\mathbb{R}^2$ es una imagen de nuestra base de datos, después vamos a calcular la matriz covarianza de nuestra base de datos y calcular sus autovectores que son V_i y V_j, graficaremos sus rectas para ver el cambio de base y visualizar que en los autovalores mas grandes en modulo sus autovectores correspondientes tienen la mayor información de la base de datos.

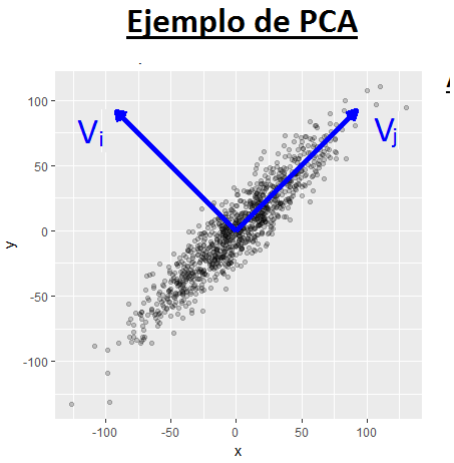


Figura 2: Componente Principales en $\mathbb{R}^2$ con el método de PCA

Como era de esperarse en la **figura 2** vemos que los autovectores asociados a los autovalores en modulo mas grandes son lo que tienen la mayor información de nuestra base datos.

2. Desarrollo

2.1. Solución para imágenes pequeñas y grandes

Dada una matriz A, los autovalores de AA^t y los de A^tA van a coincidir en los autovalores que son distinto de cero ya que claramente AA^t tiene menos autovalores que A^tA por que m >> n. Con respecto a los autovectores se van a relacionar por medio de la descomposición en valores singulares de la matriz A. Por esta razón, la idea general que vamos aplicar en nuestro trabajo es la siguiente:

a) **Imágenes grandes:** Tenemos nuestra matriz X $\in \mathbb{R}^{nxm}$ donde m >> n, donde n son las cantidad de imágenes que vamos a entrenar y m seria la cantidad de píxeles que hay en cada imagen. Si queremos calcular los autovalores y autovectores de $X^tX \in \mathbb{R}^{mxm}$ esta matriz es de dimensiones muy grande por lo tanto no entra en memoria, por lo cual no podríamos calcularle los autovalores y autovectores.
Por este motivo lo que hacemos es calcular los autovalores y autovectores de la matriz XX^t $\in \mathbb{R}^{nxn}$, donde esta matriz si entra en memoria (ya que n << m), con lo cual podemos calcular U y D $\in \mathbb{R}^{nxn}$, donde U es ortogonal y D diagonal (cabe aclarar que en D están los autovalores y en U están los autovectores asociados). Una vez que tenemos D, podemos calcular los valores singulares que son los $\sqrt{d_{11}} = |\sigma_1|$,... ,$\sqrt{d_{nn}} = |\sigma_n|$ y después por medio de la siguiente formula $v_i = \frac{X^t u_i}{\sigma_i}$ $i = 1, ..., n$ calculamos los autovectores de X^tX y por lo tanto tendríamos la matriz V $\in \mathbb{R}^{mxn}$ (cabe aclarar que si quisiéramos a V $\in \mathbb{R}^{mxm}$ lo que hacemos es agregar columnas como para completar una base ortonormal). Una vez que obtenemos V podemos usar la transformación tc(x) = V^tx, donde x $\in \mathbb{R}^m$ y $V^t \in \mathbb{R}^{mxn}$, para poder aplicar el método de PCA.

b) **Imágenes chicas:** Tenemos nuestra matriz X $\in \mathbb{R}^{nxm}$ donde m > n, donde n son las cantidad de imágenes que vamos a entrenar y m seria la cantidad de píxeles que hay en cada imagen. Si queremos calcular los autovalores y autovectores de $X^tX \in \mathbb{R}^{mxm}$ esta matriz si entra en memoria, por lo vamos a poder calcular las matriz V ortogonal y D diagonal $\in$ $\mathbb{R}^{mxm}$ (cabe aclarar que en D están los autovalores y en V están los autovectores asociados). Una vez que obtenemos V podemos usar la transformación tc(x) = V^tx, donde x $\in \mathbb{R}^m$ y $V^t \in \mathbb{R}^{mxm}$, para poder aplicar el método de PCA.

A continuación mostraremos la relación de autovalores y autovectores que nos permitieron hacer esa distinción.

2.2. Relación de los autovalores y autovectores de A^tA con AA^t

Sea A $\in \mathbb{R}^{mxn}$, queremos ver como se relaciona los autovalores y los autovectores de las siguientes matrices que son A^tA y AA^t.
Definiciones

a) **Semidefinida Positiva:** Sea B $\in \mathbb{R}^{nxn}$ es semidefinida positiva si $\forall$x $\neq 0$, donde x $\in \mathbb{R}^n$, $x^tBx \geq 0$.

b) **Norma:** $\forall$x $\in \mathbb{R}^n$, $||x|| \geq 0$.

c) **Ortogonalmente diagonalizable:** Sea B simétrica, donde B $\in \mathbb{R}^{nxn}$, es Ortogonalmente diagonalización, es decir, que existe una matriz Q $\in \mathbb{R}^{nxn}$ ortogonal y una matriz diagonal D $\in \mathbb{R}^{nxn}$ talque $A = QDQ^t$.

d) **Descomposición SVD:** Una DVS A $\in \mathbb{R}^{mxn}$ es una factorización del tipo $A = U\Sigma V^t$ con U $\in \mathbb{R}^{mxm}$ y V $\in \mathbb{R}^{nxn}$ ortogonales y $\Sigma \in \mathbb{R}^{mxn}$ una matriz formada con los Valores Singulares de A en su diagonal principal ordenados de mayor a menor.

Demostración Primero vamos a demostrar que A^tA y AA^t son simétricas y semidefinidas positivas respectivamente:
Paso a ver que son simétricas:
Quiero ver que $(A^tA)^t = A^tA$

$$(A^tA)^t = A^t(A^t)^t = A^tA \tag{1}$$

□
Quiero ver que $(AA^t)^t = AA^t$

$$(AA^t)^t = (A^t)^tA^t = AA^t \tag{2}$$

□
Ahora paso a demostrar que la matriz A es semidefinida positiva:
Quiero ver que si $\forall x \neq 0$, $x^tA^tAx \geq 0$.

$$x^tA^tAx = (Ax)^tAx = ||Ax|| \geq 0. \tag{3}$$

□
Quiero ver que si $\forall x \neq 0$, $x^tAA^tx \geq 0$.

$$x^tAA^tx = (A^tx)^tA^tx = ||A^tx|| \geq 0. \tag{4}$$

□

Paso a demostrar como se relaciona los autovalores y los autovectores de las siguientes matrices que son A^tA y AA^t.

Sea la descomposición SVD de $A = U\Sigma V^t$, sabemos que para calcular la matriz V ortogonal y los valores singulares de Σ, primero calculamos los autovalores y autovectores de A^tA.
Como A^tA es simétrica podemos diagonalizarla ortogonalmente y además como A^tA es semidefinida positiva sus autovalores son mayores o iguales a cero.
Sea λ_1,...,λ_n los autovalores (ordenados de mayor a menor) de A^tA, entonces los valores singulares de sigma son los siguientes $\sqrt{\lambda_1} = |\sigma_1|$,... ,$\sqrt{\lambda_n} = |\sigma_n|$ (estos σ_1,...,σ_n son los elementos de la diagonal de la matriz Σ).
Luego calculamos las columnas de la matriz U por medio de la siguiente formula:

$$AV = U\Sigma \tag{5}$$

Sea r el rango de la matriz A entonces tenemos σ_1,...,σ_r que son distintos de cero y σ_{r+1},...,σ_n son ceros, entonces vamos a calcular las priemras r columnas de la matriz U:

$$u_i = \frac{Av_i}{\sigma_i} \quad i = 1, ..., r \tag{6}$$

Y para calcular las demás columnas de la matriz U lo que hacemos es completar una base en el espacio de $\mathbb{R}^m$ (cabe aclarar que estas columnas están asociados a los valores singulares que valen cero).

Ahora que ya sabemos como es la descomposición SVD de A vemos que pinta tiene las descomposiciones de A^tA y AA^t, siendo que $A = U\Sigma V^t$.

$$A^tA = (U\Sigma V^t)^t(U\Sigma V^t) = V\Sigma^tU^tU\Sigma V^t \tag{7}$$

como U es ortogonal entonces $U^tU = I$

$$A^tA = V\Sigma^tI\Sigma V^t = V\Sigma^t\Sigma V^t \tag{8}$$

Además $\Sigma \in \mathbb{R}^{mxn}$ y $\Sigma^t \in \mathbb{R}^{nxm}$ entonces $\Sigma^t\Sigma \in \mathbb{R}^{nxn}$, en donde σ_1^2,...,σ_n^2 que son los elementos de la diagonal de $\Sigma^t\Sigma$ son los autovalores A^tA y en V están los autovectores correspondientes, entonces tenemos:

$$A^tA = V\Sigma^t\Sigma V^t = VDV^t \tag{9}$$

Ahora analicemos a AA^t

$$AA^t = (U\Sigma V^t)(U\Sigma V^t)^t = U\Sigma V^t V\Sigma^t U^t \tag{10}$$

como V es ortogonal entonces $V^tV = I$

$$A^tA = U\Sigma^t U^t = U\Sigma\Sigma^t U^t \tag{11}$$

Además $\Sigma \in \mathbb{R}^{mxn}$ y $\Sigma^t \in \mathbb{R}^{nxm}$ entonces $\Sigma\Sigma^t \in \mathbb{R}^{mxm}$, en donde σ_1^2,...,σ_m^2 que son los elementos de la diagonal de $\Sigma\Sigma^t$, Vamos a probar que los sigma al cuadrado son lo autovalores de AA^t y las columnas de la matriz U son los autovectores asociados a ese orden.
Sea u_i $\forall i = 1, ..., m$, una columna de la matriz U quiero ver que: $AA^tu_i = \sigma_i^2 u_i$

$$AA^tu_i = U\Sigma V^t(U\Sigma V^t)^t u_i = U\Sigma V^t V\Sigma^t U^t u_i \tag{12}$$

Como V es ortogonal entonces $V^tV = I$ y $U^tu_i = e_i^m$

$$U\Sigma V^t V\Sigma^t U^t u_i = U\Sigma\Sigma^t e_i^m \tag{13}$$

$\Sigma^t \in \mathbb{R}^{nxm}$ entonces $\Sigma^t e_i^m = \sigma_i e_i^n$ y $\Sigma \in \mathbb{R}^{mxn}$ entonces $\Sigma e_i^n = \sigma_i e_i^m$

$$U\Sigma\Sigma^t e_i^m = U\Sigma\sigma_i e_i^n = \sigma_i U\Sigma e_i^n = \sigma_i U\sigma_i e_i^m = \sigma_i^2 U e_i^m = \sigma_i^2 u_i \tag{14}$$

□
Con esto queda demostrado que σ_i^2 es autovalor a AA^t con autovector asociado u_i $\forall i = 1, ..., m$.

2.3. Análisis de K-fold cross-validation

Tenemos una base de datos que esta conformada por 41 personas diferentes, donde tenemos 10 fotos distintas de cada persona, acá donde vamos hacer una análisis de como particionar el espacio de nuestra base de datos.
Como tenemos 10 imágenes por cada persona, si queremos que cada partición este bien distribuida, los k-folds que vamos a poder aplicar son: 1-fold, 2-fold, 5-fold, 10-fold.
No tomamos las otras particiones como: 3-fold, 4-fold, 6-flod, etc. Ya que en esas particiones nuestra base de datos no estaría bien distribuida, con lo cual no podríamos aprovechar al máximo nuestra base de datos.

Si aplicáramos 1-fold estaríamos haciendo overfitting ya que muestro espacio de entrenamiento y nuestro espacio de testing coincidiría y además casi seguro nuestras métricas de evaluación estarían dando un alto porcentaje a la hora que nuestro algoritmo de reconocimiento de caras quiera predecir como se muestra en la **figura 3**.

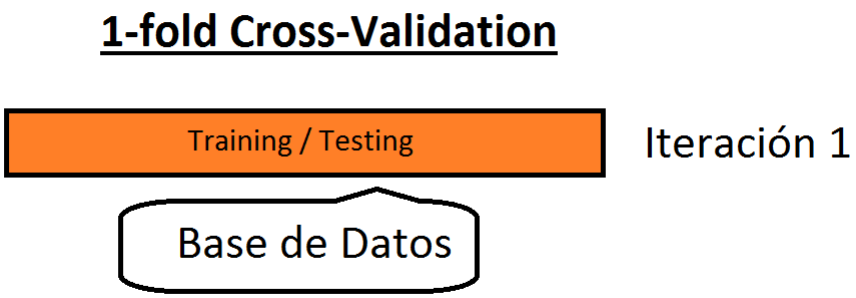


Figura 3: 1-flod Cross-Validation

Si aplicáramos 2-fold, dividiríamos nuestra base de datos en dos conjuntos, donde cada conjunto va tener 205 fotos, para formar estos dos conjuntos de 205 fotos cada uno, se formara agarrando las 5 primeras fotos de cada persona e irán en el primer conjunto y las 5 fotos restantes irán en el segundo conjunto.
Pero pueda pasar que con esta división de 2-fold estemos haciendo overfitting, ya que puede darse de forma casual que esta división se a pegue a la óptima y por eso nuestras métricas sean demasiado buenas, con lo cual nos estaríamos aprovechando de la división de la base da datos. Un ejemplo de la división de muestra en la **figura 4**.

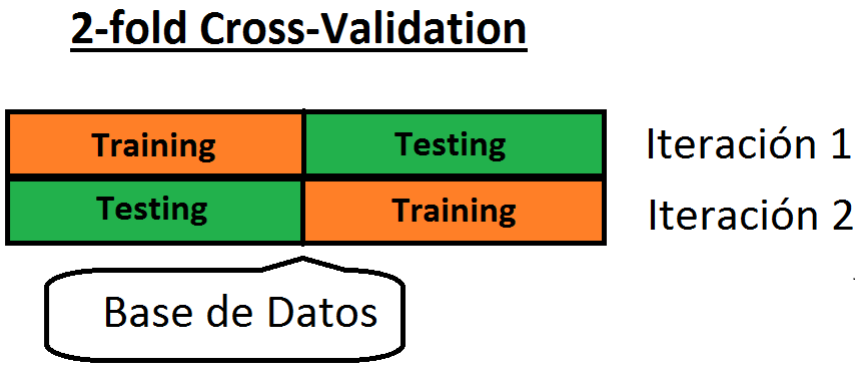


Figura 4: 2-flod Cross-Validation

Si aplicaríamos 5-fold, dividiríamos nuestra base de datos en 5 conjuntos, donde cada conjunto va tener 2 imágenes de cada persona, pero no debe haber una misma foto en distinto conjunto (lo que estamos pidiendo es que los conjuntos sean disjuntos), esto quiere decir cada conjunto va tener 82 imágenes en total.
En esta división es donde mejor podemos aprovechar nuestra base de datos, ya que tendríamos en cada iteración un 80 % de la base de datos de training y el 20 % restante seria testing, con lo cual vamos a poder reflejar en nuestras métricas que tal predice nuestro algoritmo de reconocimiento de caras. La forma de aplicarlo se ve en la **figura 5**..

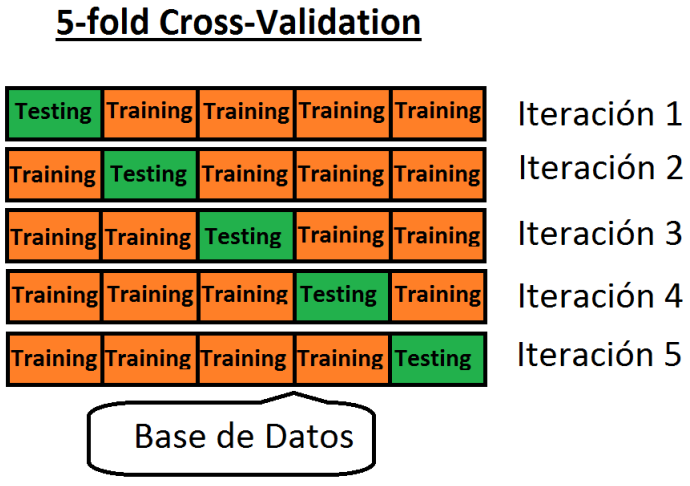


Figura 5: 5-flod Cross-Validation

Y por ultimo nos queda analizar 10-fold, este también lo podríamos aplicar, es muy parecido a 5-fold, salvo que ahora podemos particionar la base de datos en 10 conjuntos, donde cada conjunto va tener 41 fotos, pero 1 foto de cada persona y además los conjuntos deben ser disjuntos.
Lo malo de esta partición, es que tenemos muy poco testing, esto quiere decir que de la persona i-ésima (donde i=1,...41), tenemos 10 imágenes de las cuales vamos a entrenar con 9 de ellas y dejaremos una sola para hacer el testing. Pero puede pasar que también hagamos overfitting ya que tenemos mucha información a la hora de entrenar, por ende nuestras métricas nos pueden dar un buen porcentaje a la hora de aplicar nuestro algoritmo de reconocimiento de caras.

En resumen de ahora en mas en toda la experimentación que vamos hacer en adelante, utilizaremos 5-fold de la forma anterior explicada (dividir el espacio en 5 conjuntos), para poder analizar la variable k que mide la función de KNN (cantidad de vecinos mas cercanos) y la variable α que mide la cantidad de componentes principales a utilizar a la hora de aplicar PCA.

2.4. Implementación de la solución

2.4.1. Representación de las matrices

Dado que las matrices que utilizamos en este trabajo práctico para representar a las imágenes no tienen ninguna particularidad que nos permita elegir una estructura que ahorre espacio, las representamos utilizando un **Vector de Vectores de *float***. Considerando una matriz de tamaño n*n tendremos un vector de tamaño n, y cada elemento de este albergará otro vector de n elementos, representando cada fila. Entonces, si queremos acceder al elemento (i, j), accedemos primero al primer vector en la posición i lo cual nos devuelve la fila entera i representada como un vector. Luego, accedemos a la posición j de este último, representando la columna j;

Veamos algunos ejemplos de nuestra representación:

Matriz	Representación
$W = \begin{bmatrix} 0{,}0 & 0{,}0 \\ 1{,}0 & 0{,}0 \end{bmatrix}$	{ [0] = [0,0 0,0], [1] = [1,0 0,0]}
$Y = \begin{bmatrix} 0{,}0 & 0{,}0 & 1{,}0 \\ 1{,}0 & 0{,}0 & 0{,}0 \\ 0{,}0 & 1{,}0 & 0{,}0 \end{bmatrix}$	{ [0] = [0,0 0,0 1,0], [1] = [1,0 0,0 0,0], [2] = [0,0 1,0 0,0]}

2.4.2. Funciones implementadas

2.4.2.1 knn

Esta función ejecuta el algoritmo de knn para una imagen no etiquetada, según una base de datos de imágenes previamente clasificadas. En primer lugar, obtiene un vector con los k elementos de la base de datos más cercanos a la imagen. Luego obtiene la imagen que más aparece entre los k (es decir, la que tiene mayor frecuencia), la toma como predicción, y devuelve como respuesta al id de la nueva imagen etiquetada con esta respuesta. En caso de que haya más de un id con la frecuencia más alta la política para desempatar es calcular la suma de las distancias desde la imagen a clasificar hasta cada una de las imágenes de las clases más frecuentes y decidir por la clase que esté a menor distancia. Se detalla el pseudocódigo en **Algorithm 1**:

Algorithm 1: knn ($vector\langle imagen\rangle$ baseDeDatos, imagen nueva, int k) $\rightarrow \langle id,\ distancia\rangle$

```
1  vector⟨⟨idImagen, distancia⟩⟩ másCercanos;
2  for ( i ∈ [0, ..., Cantidad de imágenes en baseDeDatos) ) {
3      if tamaño(masCercanos) < k then
           /* Si todavía no llene el vector con k elementos, directamente agrego
              la imagen a los masCercanos. */
4          imagen ← baseDeDatos[i]
5          másCercanos.agregarOrdenado(⟨imagen.id, imagen.distancia⟩))
6      else
7          if distancia(baseDeDatos[i],nueva) < másCercanos.último().distancia then
               /* Si existe uno en la base de datos con distancia menor al mayor
                  entre los elegidos, saco a este y agrego al menor. */
8              imagen ← baseDeDatos[i]
9              másCercanos.eliminarÚltimo()
10             másCercanos.agregarOrdenado(⟨imagen.id, imagen.distancia⟩)
11         end
12     end
13 }
14
   /* Calculo la frecuencia de cada imagen entre las más cercanas. */
15 map⟨idImagen, ⟨frecuencia, distancia⟩⟩ frecuencias
16 for ( i ∈ [0, ..., Cantidad de imágenes en másCercanos) ) {
17     cercano ← masCercanos[i]
18     if ¬ frecuencias.estáDefinido(cercano) then
19         frecuencias[cercano] ← ⟨1, masCercanos[i].distancia⟩
20     else
21         frecuencias[cercano].frecuencia++
22         frecuencias[cercano].distancia += masCercanos[i].distancia
23     end
24 }
25
   /* Tomo el id de la imagen con frecuencia máxima, es decir, la clave con
      máximo valor en el mapa de frecuencias, si hay dos ids con la misma
      frecuencia tomo el que tiene las imágenes más cerca. */
26 predicción ← frecuencias.claveDeMáximoValor();
27 return ⟨nueva.id, predicción⟩
```

Data: Calcula knn de una imágen nueva, sobre una base de imágenes etiquetadas.

2.4.2.2 Creación de la matriz $\mathbf{M}_x$

Este algoritmo crea una matriz M_x a partir de la matriz A que contiene las imágenes pasada por parámetro. Se le resta a cada fila de la matriz el contenido del vector μ, luego se realiza multiplica la matriz por $\frac{1}{\sqrt{Filas(A)-1}}$. Eso nos permite obtener una matriz que contiene en la i-ésima fila al vector $(x_i - \mu)/\sqrt{n-1}$.
Luego transpone a X y devuelve M_x como el producto entre X y X^t. Se detalla el pseudocodigo en **Algorithm 2**:

Algorithm 2: creacion_matrix_M_x(Matriz A, vector μ) $\rightarrow$ Matriz M_x

```
/* el tamaño vector μ debe ser igual #(columnas A) */
/* X y A tienen las mismas dimensiones */
/* los vectores son matrices de dimensiones μ ∈ ℝ^{mx1}  */
/* B_μ es una matriz de mxm que todas sus columnas son iguales a μ */
```

1 Matriz X $\leftarrow A - B_\mu$
2 X $\leftarrow \frac{1}{\sqrt{\#(Filas(A))-1}} * X$
3 Matriz M_x $\leftarrow$X^T*X

Data: Creación de la matriz M_x

2.4.2.3 Funciones para calcular la matrices V y D

Aquí vamos a mostrar como implementamos las funciones necesarias para calcular las matrices V y D donde D es la matriz diagonal que tiene los autovalores de M_x y V es la matriz donde están los autovectores asociados.

Para calcular el autovalor dominante programamos el método de la potencia. Este método consiste en multiplicar la matriz pasada por parámetro por el vector pasado como parámetro y luego verificar que se cumpla la definición de autovalor ($Av = \lambda v$). Si no vale la igualdad se sigue iterando hasta la máxima cantidad de repeticiones. El algoritmo finaliza si se encuentra el autovalor o si se iteró lo suficiente. Se detalla el pseudocodigo en **Algorithm 3**:

Algorithm 3: método_potencia(Matriz A, vector x, int repeticiones) $\rightarrow$ (int λ,vector v)

```
/* los vectores son matrices de dimensiones v ∈ ℝ^{mx1}  */
```

1 vector v $\leftarrow x$
2 int i $\leftarrow 0$
3 **do**
4 | Matriz y $\leftarrow A * v$
5 | int k $\leftarrow ||y||_2$
6 | v $\leftarrow \frac{1}{k} * y$
7 | k $\leftarrow ||v||_2^2$
8 | $\lambda \leftarrow v^T * A * v$
9 | $\lambda \leftarrow \lambda$ / k
10 | i $\leftarrow$ i +1
11 **while** *($i <$ repeticiones && $Av \neq \lambda v$)*;

Data: Calcula el autovalor dominante A con su autovector asociado

En este pseudocódigo vamos a calcular los autovalores y autovectores correspondiente de la matriz X, donde la matriz D que es diagonal, contiene los autovalores y la matriz U la cual tendrá como columnas los autovectores correspondiente. Para realizar esta operación se usara el método de la potencia y además el método de deflación, el cual consiste en una vez calculado un autovalor α y su autovector v correspondiente hacemos por medio de la siguiente formula $X = X$ - α vv^t, que el autovalor α que es dominante en la matriz X, pase a no ser lo y se convierta para la matriz resultante autovalor 0 y el autovector v siga siendo asociado a él. Se detalla el pseudocodigo en **Algorithm 4**:

Algorithm 4: generación_U_D(Matriz A, Matriz U, Matriz D, int alpha)

```
   /* A, U, D tienen las mismas dimensiones. */
1  autovector ← matriz(A.filas(), 1);
2  for ( 1 ∈ [1, ..., alpha] ) {
       /* Se crea un vector x_0 con valores random diferentes de 0 y se lo
          normaliza. */
3      x_0 ← matrizRandom(A.cantFilas(), 1)
4      x_0.normalizar()
       /* Se utiliza el método de la potencia con 500 repeticiones para obtener
          el autovector y autovalor asociado de A. */
5      autovalor, autovector ← metodo_potencia(A, x_0, 500)
       /* Se pone al autovector obtenido como la columna i de U, y al autovalor
          como el elemento (i, i) de D. */
6      U_i ← autovector
7      D_ii ← autovalor
       /* Deflación */
8      autovector.normalizar()
9      B ← autovector * autovector^T
10     B ← B * autovalor
11 }
```

Data: Generación de matrices U y D.

En este pseudocódigo vamos aplicar el método de PCA el cual hemos explicado en la parte de desarrollo. Se detalla de forma abstracta el pseudocodigo que se implemento viendo **Algorithm 5**:

Algorithm 5: Metodo_PCA(vector⟨*imagen*⟩ Entrenamiento, vector⟨*imagen*⟩ Img_clasificar, int alfa, int k) → ⟨*id*, *distancia*⟩ solucion

```
1  Matriz A ← PasarImagenAmatriz(Entrenamiento)
2  Matriz μ ← vector_con_los_promedios_de(A)
3  Matriz M_x ← creacion_matriz_M_x(A, μ)
   /* Las matrices V y D tienen las mismas dimensiones que M_x.  */
4  Matriz V
5  Matriz D
6  V, D ← generacion_U_D(M_x,alfa)
   /* Aplicamos la transformación lineal a la base de datos de entrenamiento */
7  for ( i ∈ [1, ..., Tam(Entrenamiento))] ) {
       /* Entrenamiento ∈ IR^nx1 y V^T ∈ IR^αxn */
8      Entrenamiento[i] ← ((Img_clasificar[i]−μ[i]) / sqrt(Entrenamiento.size()−1))
9      Matriz y ← V^T * Entrenamiento
10     Entrenamiento ← y
11 }
   /* Aplicamos la transformación lineal tc a la base de datos a clasificar */
12 for ( i ∈ [1, ..., Tam(Img_clasificar))] ) {
       /* Img_clasificar ∈ IR^nx1 y V^T ∈ IR^αxn */
13     Img_clasificar[i] ← ((Img_clasificar[i]−μ[i]) / sqrt(Img_clasificar.size()−1))
14     Matriz z ← V^T * Img_clasificar
15     Img_clasificar ← z
16 }
17 solucion ← knn(Entrenamiento,Img_clasificar,k)
```

Data: Aplicacion del Algoritmo de PCA

2.4.2.4 Funciones para cuando tenemos imágenes grandes

Para resolver esta parte lo que vamos hacer, es ya no calcular la matriz $M_x = X^T X$ ya que es muy grande y no entra en memoria, pero una alternativa para solucionar el problema es calcular la siguiente matriz $M_x^{'} = XX^T$, ya que esta sí entra en memoria. Para volver a tener la matriz V que tiene los autovectores de la matriz $M_x = X^T X$ haremos un función que permita calcularlo usando los autovalores y autovectores de las matriz $M_x^{'} = XX^T$. Se detalla los pseudocodigos en **Algorithm 6** y **Algorithm 7**:

Algorithm 6: conversionUaV(Matriz X, Matriz U, Matriz D, Matriz V)

```
1  for ( i ∈ [1, ..., #(Columnas(V)] ) {
       /* e_i es el i-ésimo vector canónico IR^n  */
2      Matriz e_i
3      Matriz u_i ← U * e_i
4      float d_ii ← sqrt(abs((D)_ii))
5      V[i] ← ((X*u_i)/d_ii)^t
6  }
```

Data: Crea la matriz V que tiene los autovectores de la matriz M_x

Algorithm 7: Metodo_PCA_Imagenes_Full(vector$\langle imagen \rangle$ Entrenamiento, vector$\langle imagen \rangle$ Img_clasificar) $\rightarrow \langle id,\ distancia \rangle$ solucion

```
   /* X y A tienen las mismas dimensiones */
   /* los vectores son matrices de dimensiones μ ∈ IR^mx1  */
   /* B_μ es una matriz de mxm que todas sus columnas son iguales a μ */
 1 Matriz A ← PasarImagenAmatriz(Entrenamiento)
 2 Matriz μ ← vector_con_los_promedios_de(A)
 3 Matriz X ← A − B_μ
 4 X ← 1/sqrt(#(Filas(A))−1) * X
 5 Matriz M'_x ← X * X^T
   /* Las matrices U y D tienen las mismas dimensiones que M'_x.  */
 6 Matriz U
 7 Matriz D
 8 U, D ← generacion_U_D(M'_x,alfa)
   /* La matriz V ∈ IR^mxα. */
 9 Matriz V
   /* Aquí armamos la matriz V que tiene los autovectores de la matriz M_x. */
10 V ← conversionUaV(X^t,U,D)
   /* Aplicamos la transformación lineal a la base de datos de entrenamiento */
11 for ( i ∈ [1, ..., Tam(Entrenamiento))] ) {
       /* Entrenamiento ∈ IR^nx1 y V^T ∈ IR^αxn */
12     Entrenamiento[i] ← (Img_clasificar[i]−μ[i] / sqrt(Entrenamiento.size()−1))
13     Matriz y ← V^T * Entrenamiento
14     Entrenamiento ← y
15 }
   /* Aplicamos la transformación lineal tc a la base de datos a clasificar */
16 for ( i ∈ [1, ..., Tam(Img_clasificar))] ) {
       /* Img_clasificar ∈ IR^nx1 y V^T ∈ IR^αxn */
17     Img_clasificar[i] ← (Img_clasificar[i]−μ[i] / sqrt(Img_clasificar.size()−1))
18     Matriz z ← V^T * Img_clasificar
19     Img_clasificar ← z
20 }
21 solucion ← knn(Entrenamiento,Img_clasificar,k)
```

Data: Aplicacion del Algortimo de PCA para imagenes grandes

2.5. Experimentación

2.6. Estudio de los parámetros de entrada

En esta sección vamos a mostrar los experimentos realizados para ver como afectan cualitativamente y cuantitativamente los distintos parámetros de nuestra implementación, es decir la cantidad de vecinos y de autovectores a considerar, si se realiza PCA o no, etc.

2.6.1. Experimentos relacionados a la calidad de la clasificación

Para los siguientes experimentos realizamos 5-fold para entrenar por lo explicado en la sección 2.3.
En los gráficos se muestra la métrica *accuracy*, la cual se define de la siguiente manera

$$accuracy = \frac{cantidad\ de\ aciertos}{cantidad\ de\ intentos}$$

2.6.1.1 Estudio de los parámetros k y α

Ahora veamos como afecta la cantidad de vecinos (k) y la cantidad de autovalores (α) tenidos en cuenta. Se hizo un 5-fold y se usó PCA.

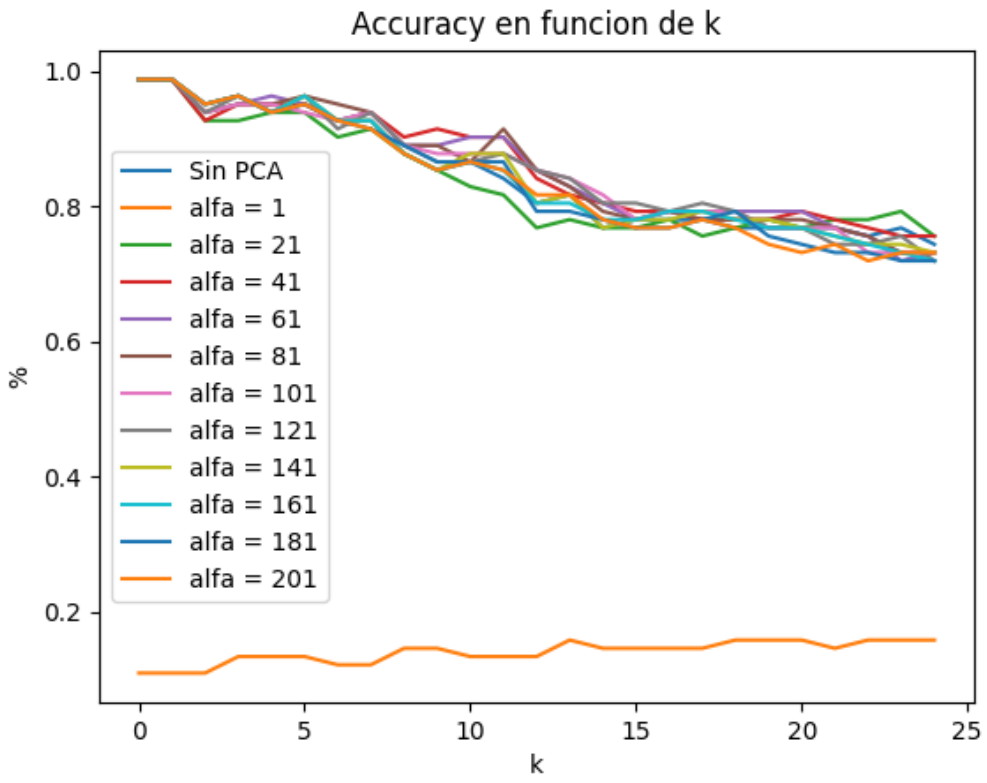


Figura 6

La leyenda aclara la cantidad de autovalores a considerar. Desde la parte cualitativa se puede observar en la figura 6 que cuando aumenta la cantidad de vecinos a considerar el reconocimiento es levemente peor. Principalmente se observa que si se cuentan hasta casi diez vecinos el desempeño

del clasificador es muy bueno, mientras que a partir de diez el accuracy es de aproximadamente el 80 %. Este resultado es razonable ya que hay diez imágenes en cada clase. Si se elige usar un solo autovector clasifica muy mal porque no tiene información suficiente, pero si se utilizan 21 autovectores ya es suficiente para que tenga un gran accuracy. No hay diferencias significativas entre usar PCA y no hacerlo.

2.6.2. librerias de python para la experimentación

Para el calculo del acurracy, precisión y recall utilizando la librería de python sklearn usando las funciones
para el calculo del acurracy : accuracy_score(true, pred)
para la precision :precision_score(true,pred, average=macro)
para el recall : recall_score(true, pred, average=macro)
donde true es una lista indexada donde cada posición corresponde a las imágenes verdaderas y pred corresponde a las imágenes que fueron predictas. Luego average='macro' lo que hace es decirle a la función que el cálculo es para un vector de valores multiclase por lo que tiene que tomar una clase y evaluar el precision y recall en función de todo el espacio muestral o sea si esta en la clase es un positivo pero si no esta en la clase es negativo. Y luego de haber calculado el precision y recall para las 41 clases hace un promedio de todas las clases y devuelve el valor final de preccision, acurracy y recall.

2.6.3. Experimentos relacionados al tiempo de ejecución

2.6.3.1 Estudio del tiempo de entrenamiento con PCA

Si no se realiza PCA nuestro clasificador prácticamente no tiene entrenamiento, en cambio cuando se hace PCA lo tiene y el costo temporal no es para nada despreciable ya que se ejecutan multiplicaciones de matrices, se calculan autovalores y autovectores, etc. Por ende decidimos experimentar el tiempo de ejecución del entrenamiento (es decir, PCA pero sin knn) variando la cantidad de autovalores que se calculan.

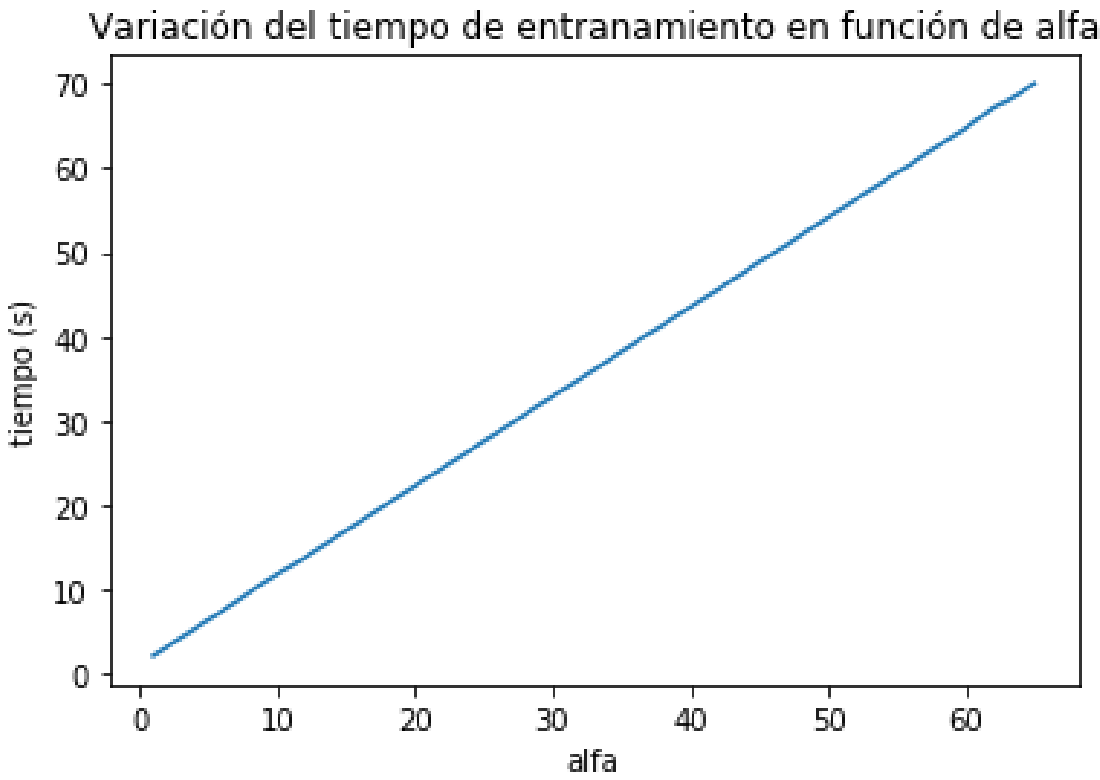


Figura 7: Gráfico de cantidad de tiempo en función de alfa

En la figura 7 se puede ver que el tiempo de ejecución del entrenamiento aumenta linealmente con respecto a la cantidad de autovalores a calcular. Por lo que no es tan malo colocar un alfa mediano (es decir, entre un 25 % y un 50 % de la cantidad de filas de la imagen) en caso de que no clasifique con buen accuracy si se utilizan pocas componentes principales.

2.6.3.2 Estudio del tiempo de clasificación con y sin PCA

Dados los resultados de la sección 2.6.1.1 se fijó la cantidad de vecinos en 5 y la cantidad de componentes a considerar en 21.

Tipo de clasificación	Tiempo promedio (s)
Con PCA	0,08436112
Sin PCA	0,49745378

Creemos que la gran diferencia entre ambos tiempos de clasificación se debe a que cuando uno realiza PCA no está analizando la imagen por completo sino que solo las (en este caso, 5) componentes principales. Este resultado muestra que es una gran ventaja aplicar PCA, ya que si bien el tiempo que se necesita para el entrenamiento es mucho mayor lo compensa a mediano plazo porque la clasificación se hace en un tiempo alrededor de seis veces menor.

2.6.4. Conclusión

La conclusión a la que se puede llegar es que conviene utilizar PCA porque si bien su tiempo de entrenamiento es mayor su tiempo de clasificación es mucho menor que sin PCA, además solo considerando alrededor de 20 autovalores el accuracy es similar a la versión sin PCA por lo que tampoco se estaría perdiendo calidad. Con respecto a la cantidad de vecinos a considerar creemos que entre 3 y 8 el clasificador posee muy buena calidad.

2.7. Acurracy precision and recall

Debido al que el experimento siguiente vamos a realizar un k-fold que podría tener un posible overfitting ya que al elegir los elementos de manera aleatoria es posible que se evalue mas de una vez una imagen, lo que vamos a hacer es lo que fue mencionado anteriormente al principio del desarrollo y tomar 5 folds luego vamos a calcular, acurracy precisión y recall utilizando la librería de python.

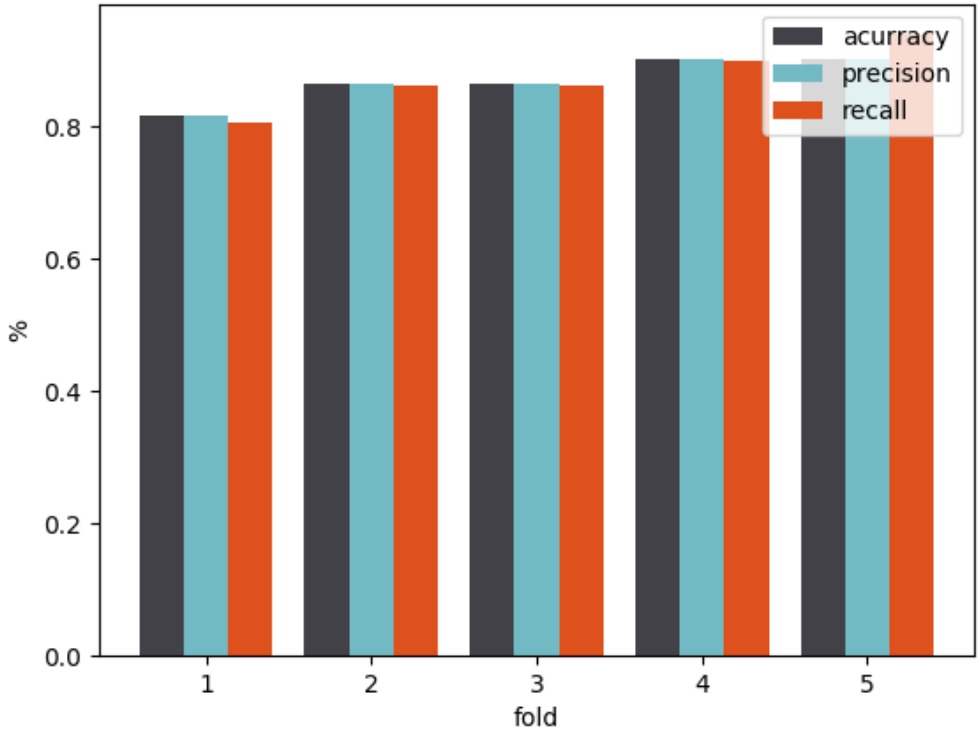


Figura 8: Acurracy, Precision, Recall

Estos son los valores para los 3 distintos parametros en cada uno de los folds, luego tomando un promedio de los 5 folds nos queda:
Acurracy: 0.870
Recall: 0.873
Precision: 0.870

2.8. Variando la cantidad de imágenes de aprendizaje

En este experimento vamos a ir regulando la cantidad de imágenes que nuestro programa va a utilizar para aprender, Como nuestra base de datos esta determinada por 10 imágenes de 41 personas distintas, hemos decidido que para tener una muestra homogénea y bien distribuida vamos a variar en la cantidad de imágenes por persona desde 1 imagen hasta 9, no vamos a aprender con 10 imágenes porque, por un lado nos quedamos sin imágenes para el test y si quisiéramos usar las mismas imágenes que usamos para entrenar estaríamos comparando con imágenes que ya están en nuestra base por lo que acertaría en un 100 %. Una observación mas es que, no vamos a aplicar

el método de k-fold directamente, sino que vamos a aplicar su variable randomnizada, ya que si quisiéramos particionar el grupo en conjuntos disjuntos de exactamente la cantidad de tamaños que necesitamos para ir variando el k, sucedería que no podemos hacerlo y tendríamos que o bien descartar imágenes y tener menos casos de test. Por lo que hemos decidido, que para cada grupo de imágenes a entrenar se tomará una x cantidad de imágenes correspondientes a una persona de manera aleatoria en 10 iteraciones.

obs. Cada vez que mencionemos 1..9 imágenes de entrenamiento, nos referimos a que vamos a utilizar de 1 a 9 imágenes por cada clase.

Por otro lado, Vamos a fijar los parámetros de α y κ donde α es la cantidad de componentes principales y κ es la cantidad de vecinos cercanos a considerar $\alpha = 40$ debido a que es una cantidad de autovalores bastante importante por lo que nos permitiría una buena aproximación a las imágenes reales $\kappa = 9$ ya que, cuando aumentamos de 9 vecinos, nuestro programa empezaba a fallar cada vez mas y a medida que se incrementaba, el rendimiento iba disminuyendo.

Lo que vamos a medir en este experimento va a ser, para cada conjunto de imágenes para entrenar, calculamos su Acurracy, Precisión y Recall, como estamos en un conjunto multiclase, vamos a determinar el Acurracy Recall y Precisión de cada clase con todo el conjunto y luego tomar el promedio, Este calculo se realizará con la librería de python scikit-learn que nos permite realizar estos cálculos de manera sencilla.

1. Resultado del experimento:

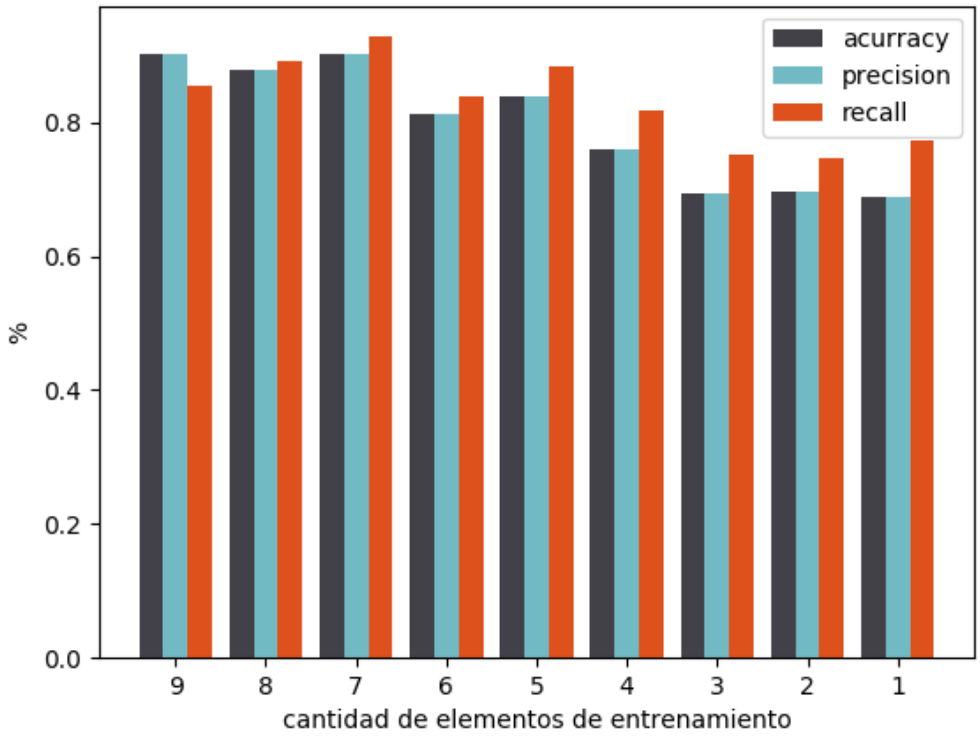


Figura 9: Acurracy, Precision, Recall

Como era relativamente de esperarse, podemos observar que, a medida que decrece la cantidad de imágenes de entrenamiento, nuestras mediciones empiezan a decrecer gradualmente.

Aunque se nota un cambio bastante abrupto cuando la cantidad de imágenes de entrenamiento es menor igual a 3, donde los valores decaen por debajo del 40 %, el peor de los casos lo tenemos cuando usamos una sola imagen para entrenar el cual nos da valores por debajo del 10 %

Ahora, como resultado de este experimento nos dimos cuenta que, estamos cometiendo un error al mantener fija la cantidad de vecinos, porque. Para cuando tenemos 9 imágenes de una persona en una base de datos, y estas personas son 41 tenemos casi 400 posibles vecinos para algún punto cualquiera, pero a medida que reducimos la cantidad de imágenes para cada una de nuestras clases empieza a perder sentido el tomar tantos vecinos, por ejemplo en el caso de que usamos solo 1 imagen de cada persona para entrenar,tenemos 41 clases, con solo una imagen, Entonces decidimos tomar los 10 mas cercanos, como resultado vamos a tener que los 10 mas cercanos pertenecen a 10 clases distintas, por lo cual el algoritmo se confunde. Para solventar esto, vamos a a tomar la cantidad de vecinos equivalente a la cantidad máxima de imágenes para entrenar, en otras palabras si hay 3 imágenes de entrenamiento, solo vamos a tomar a los 3 vecinos mas cercanos.

2. Resultado del experimento 2

gradual.png

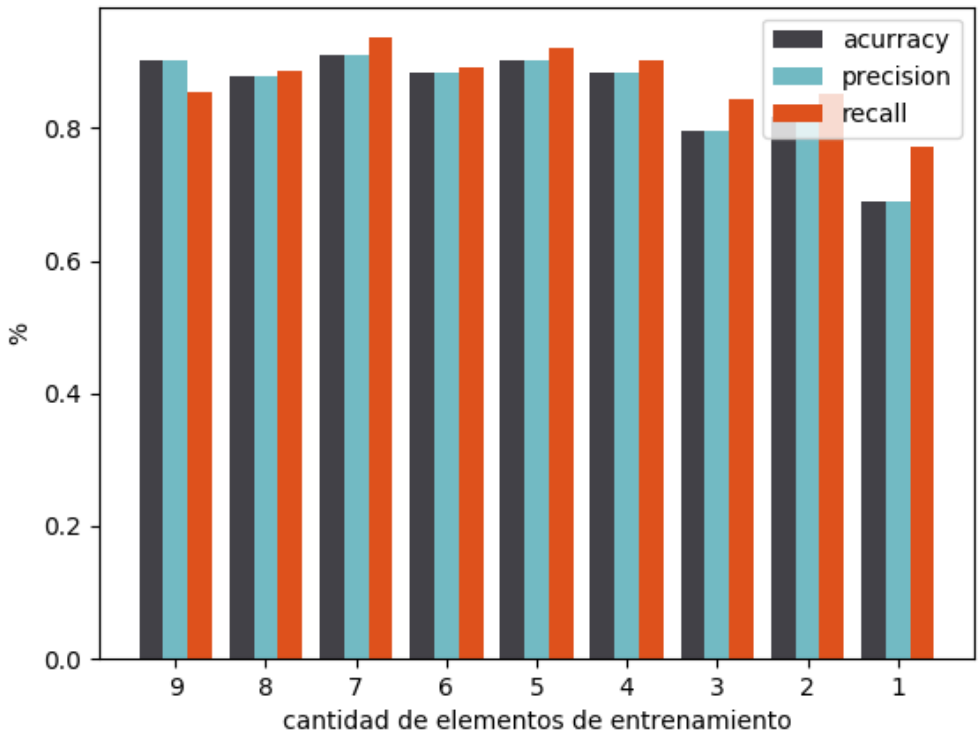


Figura 10: Acurracy, Precision, Recall

Efectivamente estábamos en lo correcto, el algoritmo de KNN fallaba cuando teníamos una cantidad mas reducida de imágenes en nuestra base de dato, y al tomar un κ demasiado grande, estábamos tomando mas de una clase entera en nuestro análisis por lo que arrojaba errores a la hora de calcular los k vecinos mas cercanos.

Ahora, reduciendo la cantidad de vecinos a la cantidad máxima de imágenes por persona que hay en la base de entrenamiento Sigue sucediendo que, el rendimiento en general del programa va disminuyendo a medida que reducimos la cantidad de imágenes por persona, pero lo de una manera leve , casi que tiene una distribución uniforme.

3. Experimento temporal:

   Como se pide en el enunciado medimos el tiempo de nuestro experimento el cual no nos aporta demasiada información ya que, estamos trabajando con matrices todo el tiempo por lo cual tenemos una complejidad determinada por el tamaño de estas matrices.

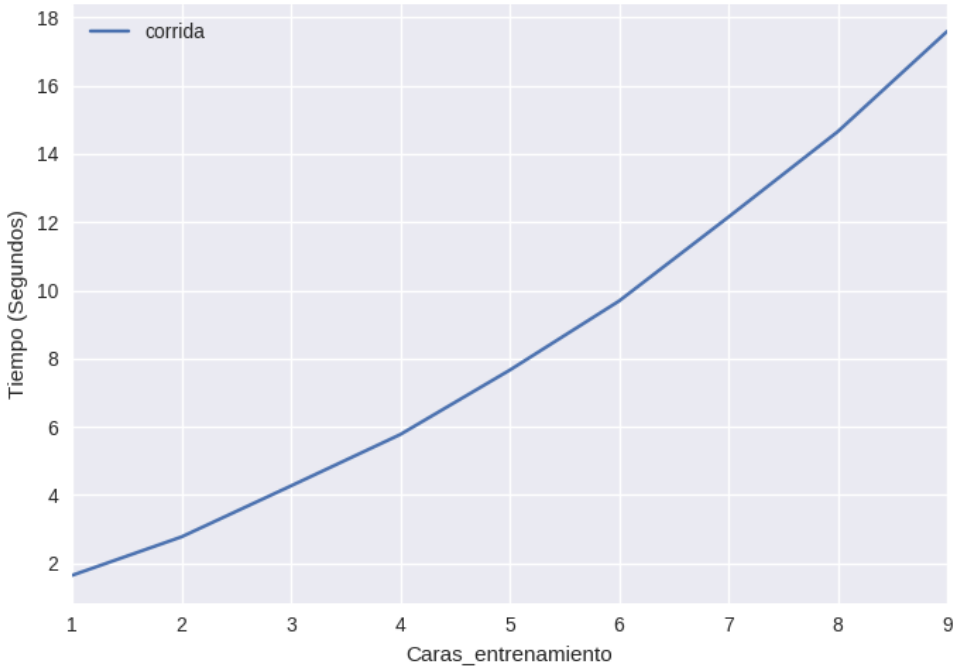
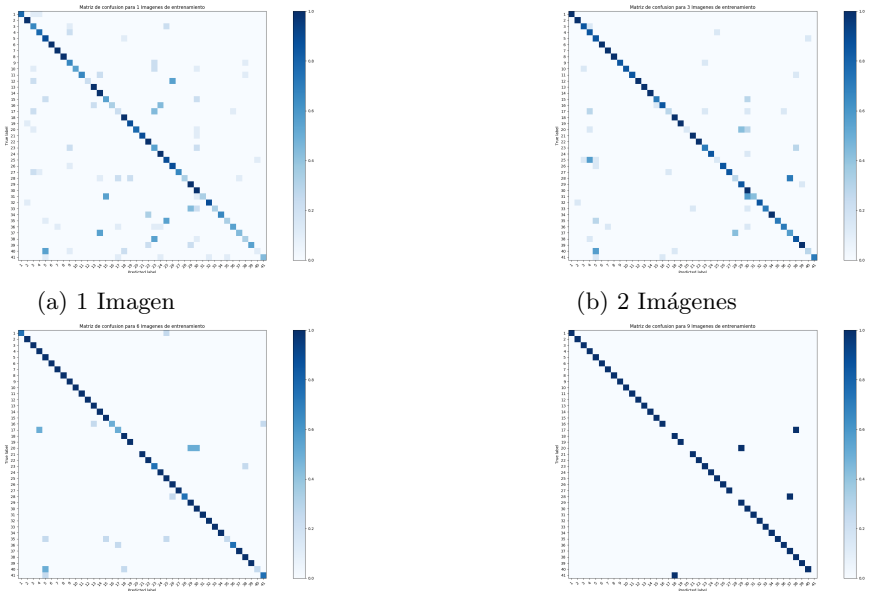


Figura 11: Tiempos del algoritmo

Podemos observar que tiene un carácter polinómico, pero no nos vamos a detener en evaluar algo que ya sabemos y podemos determinar con exactitud.

Hasta ahora nosotros venimos calculando métricas que solo nos dan un promedio del funcionamiento del programa en general en base a todas las clases, pero nos interesaría también ver que es lo que sucede exactamente con cada clase, para eso vamos a utilizar las matrices de confusión para observar nuestras clases.

4. Matriz de confusión experimento 2



(a) 1 Imagen

(b) 2 Imágenes

(c) 6 Imágenes

(d) 9 Imágenes

Figura 12: Matriz de confusión por cantidad de imágenes de entrenamiento

Se puede ver que dentro de todo las matrices que tienen pocas imágenes a la hora de ser entrenadas tiene un mayor nivel de dispersión y de confusión, cosa que a medida que aumentamos la cantidad de imágenes de entrenamiento va reduciendo gradualmente. También lo que podemos destacar es que cuando tenemos un solo elemento de testeo hay un 100 % de precisión y 100 % de error cuando el algoritmo falla y esto sucede porque como tenemos una sola imagen de testeo es a todo o nada Por lo que vamos a realizar una tercera muestra de los datos Para medir el acurracy, recall y precisión nosotros tomamos un promedio de estos valores para cada iteración, pero para poder promediar una matriz de confusión no podemos tomar el promedio, porque, las matrices de confusión toman como datos de entrada las clases verdaderas y las clases predichas, por lo que, para cada cantidad de imágenes de entrenamiento, lo que vamos a hacer es que durante las 10 iteraciones, vamos a sumar el total de todas las predicciones realizadas, por ejemplo, si para las 5 imágenes de la clase 1 predijo en la primera iteración que 4 son de la clase 1 y una de la clase 3 y en la segunda iteración, las 5 son de la clase 1.

Entonces en la matriz de confusión vamos a tener 9 aciertos a la clase 1 y un acierto a la clase 3 para los elementos de la clase 1.

De esta forma vamos a poder tener una muestra mas visual de donde es que siempre el algoritmo esta confundiendo, y que clases son las mas propensas a eso.

5. Matriz de confusión "promediada"

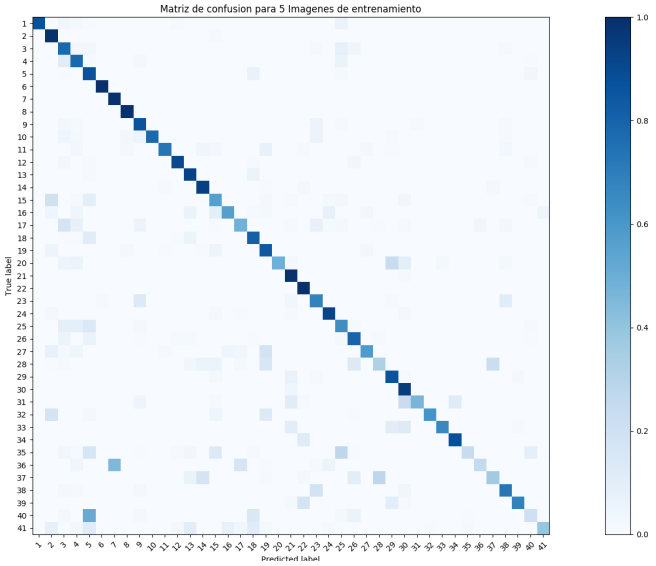
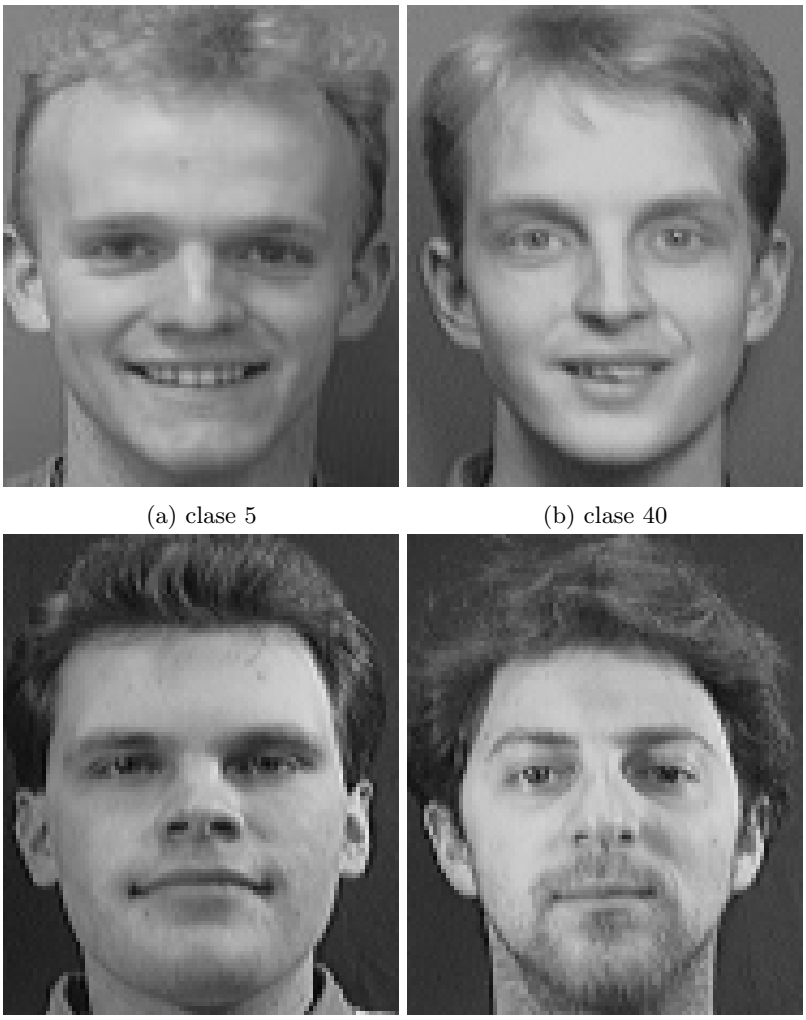


Figura 13: Matriz de confusión normalizada para 10 iteraciones

No vamos a mostrar las matrices en todas sus variantes, porque después de haberlas graficados, pudimos ver que, todas, para cada uno de los k tienen exactamente la misma forma que lo tiene esta, pero con un ligero cambio de color entre la cantidad de imágenes, mientras mas grande la base de entrenamiento, menor es el error. Pero de todas formas, siempre se mantienen con exactamente esta misma distribución. Hemos decidido normalizar esta matriz para que se pueda apreciar el porcentaje de aciertos que hay en total. las clases que menos acierta son las clases: 28, 36 37 38 40 y 41, que sus correspondientes aciertos están por del bajo del 20 % en particular la clase 40 se confunde en un mas del 60 % con la clase 5 y la clase 36 en un mismo porcentaje aproximado con la clase 7

Ahora nos resulta interesante ver como son estas personas para que falle de este modo el algoritmo así que vamos a buscarlas y comparar visualmente si en realidad se parecen en algo o no.



(a) clase 5 (b) clase 40

(c) clase 36 (d) clase 7

Figura 14: Individuos sospechosos

Observando a los poseedores de las imágenes de cada clase podemos ver que, el algoritmo esta fallando pero no por cualquier razón, sino que por ejemplo en la clase 5 y 40, ambos son rubios menemistas de ojos claros, con rasgos muy similares y además en todas sus fotos salen sonriendo por lo que tiene mucho sentido que el algoritmo crea que ambas personas son la misma.

Pero no tanto así como sucedió con la clase 36 y la clase 7, que quizás, haciendo un poco de fuera podamos encontrarle algunos que otros rasgos parecidos, como por ejemplo el pelo y los ojos, pero no se parecen demasiado. Aunque de todas formas, que de 41 personas, haya confundido con tanta frecuencia solo a dos personas que no tienen rangos parecidos no es tan grave.

2.8.1. Conclusión

Como conclusión podemos decir que no es necesario que la cantidad de imágenes que necesitamos en nuestra base de datos sea demasiado grande, porque ya hemos visto que tomando solo una imagen de entrenamiento hemos tenido un gran porcentaje de acierto. Pero para que esto se de, se tiene que tomar en cuenta que no podemos tomar una cantidad de vecinos mas grande que la cantidad de imágenes que tenemos por persona en nuestra base de datos porque esto nos traería grandes errores. Como se ven en el primer experimento, que teniendo solo 1 imagen por persona, tomábamos los 10 vecinos y cuando realizábamos la moda, la moda se hacia con 10 clases distintas

en las cuales todas estas clases decían que la imagen pertenecía a ella por lo que generaba una deformidad severa en los resultados del experimento.

3. Conclusiones

En este trabajo analizamos e implementamos un sistema de reconocimiento de caras a partir de imágenes. En primer lugar utilizamos una idea sencilla, KNN, que funcionó de una manera mucho mejor de lo esperada. Lidiando con problemas en los tiempo de ejecución y con la gran cantidad de dimensiones, desarrollamos el método PCA para reducir la cantidad de variables de las muestras.

Analizamos, con la técnica de K-fold Cross-Validation, los resultados obtenidos y los niveles de Precisión, Recall y Acurracy que se lograron al ir variando los diferentes parámetros de nuestros métodos. Con estos datos, estimamos cuáles eran aquellos parámetros que conseguían los mejores resultados posibles.

También para no caer en errores de promedios a la hora de evaluar las métricas de Precisión, Recall y Acurracy de nuestro algoritmo al predecir, usamos una análisis mas detallado usando la matriz de confusión, con el cual pudimos encontrar en donde falla nuestro algoritmo de reconocimiento de caras a la hora de predecir y ver específicamente en que imágenes resulta errónea la predicción.

Medimos la eficiencia de ambos métodos (KNN sin PCA y con PCA), tanto en efectividad como en costo temporal, y encontramos que al aplicar KNN con PCA es el mejor.

4. Anexo

4.1. Enunciado

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Reconocimiento de caras

Introducción

Como plan estratégico para el desarrollo del país, las autoridades nacionales están a días de inaugurar el prometedor *soja valley* que será el pilar económico de los años venideros. Dado que el rimbombante emprendimiento albergará a las industrias más importantes para el país, se necesita un sistema de reconocimiento de trabajadores que ingresen a las instalaciones dado que se estará manejando información confidencial a diario.

Si bien es muy factible que existan mejores formas y más económicas para reconocer diariamente a un trabajador, las autoridades están muy interesadas en desarrollar un sistema biométrico basado en reconocimiento facial.

En este contexto, las autoridades han contactado al Departamento de Computación para el desarrollo del sistema de reconocimiento facial que se utilizará en el flamante *soja valley*. Como punto de partida para la realización de un prototipo, nos han provisto de una base de datos preliminar para poder realizar una prueba de concepto.

El foco de este trabajo está puesto en la experimentación científica de un método de reconocimiento caras simplificado usando clasificación por vecinos más cercanos y reducción de la dimensionalidad.

Metodología

Se quiere desarrollar un algoritmo de clasificación supervisada el cual se deberá *entrenar* con una base de caras **conocidas** que luego nos servirá para reconocer otras instancias de esas caras no presentes en la base de datos de entrenamiento.

Como instancias de entrenamiento, se tiene un conjunto de N personas, cada una de ellas con M imágenes distintas de sus rostros en escala de grises y del mismo tamaño. Cada una de estas imágenes sabemos a qué persona corresponde.

Reconocimiento de caras

El objetivo del reconocimiento de caras consiste en utilizar la información de la base de datos para, dada una nueva imagen de una cara sin etiquetar, determinar a quién corresponde. En este trabajo práctico nos vamos a basar en el trabajo de Turk and Pentland [3] en el cual se plantea un esquema muy simple de reconocimiento de caras.

Una primera aproximación es utilizar el conocido algoritmo de *k vecinos más cercanos* o kNN [2], por su sigla en inglés. En su versión más simple, este algoritmo considera a cada objeto de la base de entrenamiento como un punto en el espacio euclídeo m-dimensional, para el cual se conoce a qué clase corresponde (en nuestro caso, qué persona es) para luego, dado un nuevo objeto, asociarle la clase del o los puntos más cercanos de la base de datos.

Procedimiento de k vecinos más cercanos

- Se define una base de datos de entrenamiento como el conjunto $\mathcal{D} = \{x_i \,:\, i = 1, \ldots, n\}$.
- Luego, se define m como el número total de píxeles de la imagen i-ésima almacenada por filas y representada como un vector $x_i \in \mathbb{R}^m$.
- De esta forma, dada una nueva imagen $x \in \mathbb{R}^m$, talque $x \notin \mathcal{D}$, para clasificarla simplemente se busca el subconjunto de los k vectores $\{x_i\} \subseteq \mathcal{D}$ más cercanos a x, y se le asigna la clase que posea el mayor número de repeticiones dentro de ese subconjunto, es decir, la moda.

El algoritmo del vecino más cercano es muy sensible a la dimensión de los objetos y a la variación de la intensidad de las imágenes. Es por eso, que las imágenes dentro de la base de datos $\mathcal{D}$ se suelen *preprocesar* para lidiar con estos problemas.

Teniendo en cuenta esto, una alternativa interesante de preprocesamiento es buscar reducir la cantidad de dimensiones de las muestras para trabajar con una cantidad de variables más acotada y, simultáneamente, buscando que las nuevas variables tengan información representativa para clasificar los objetos de la base de entrada.

En esta dirección, consideraremos el método de reducción de dimensionalidad *Análisis de componentes principales* o PCA (por su sigla en inglés) dejando de la lado los procesamientos de imágenes que se puedan realizar previamente o alternativamente a aplicar PCA.

Análisis de componentes principales

El método de análisis de componentes principales o PCA consiste en lo siguiente.

Sea $\mu = (x_1 + \ldots + x_n)/n$ el promedio de las imágenes $\mathcal{D} = \{x_i : i = 1, \ldots, n\}$ tal que $x_i \in \mathbb{R}^m$. Definimos $X \in \mathbb{R}^{n \times m}$ como la matriz que contiene en la i-ésima fila al vector $(x_i - \mu)^t/\sqrt{n-1}$. La matriz de covarianza de la muestra X se define como $M = X^t X$.

Siendo v_j el autovector de M asociado al j-ésimo autovalor, al ser ordenados por su valor absoluto, definimos para $i = 1, \ldots, n$ la *transformación característica* de x_i como el vector $\mathbf{tc}(x_i) = (v_1\, x_i, v_2\, x_i, \ldots, v_\alpha\, x_i)^t \in \mathbb{R}^\alpha$, donde $\alpha \in \{1, \ldots, m\}$ es un parámetro de la implementación. Este proceso corresponde a extraer las α primeras *componentes principales* de cada imagen. La idea es que $\mathbf{tc}(x_i)$ resuma la información más relevante de la imagen, descartando los dimensiones menos significativas.

Imágenes de gran tamaño

La factibilidad de aplicar PCA es particularmente sensible al tamaño de las imágenes de la base de datos. Por ejemplo, al considerar imágenes en escala de grises de 100×100 píxeles implica trabajar con matrices de tamaño 10000×10000. Una alternativa es reducir el tamaño de las imágenes, por ejemplo, mediante un submuestreo (eliminación intercalada de filas y columnas de la imagen).

Sin embargo, es posible superar esta dificultad en los casos donde el número de muestras es menor que el número de variables, ya que es posible encontrar una relación entre los

autovalores y autovectores de la matriz de covarianza $M = X^t * X$ y la matriz $\hat{M} = X * X^t$.

Clasificación con kNN y PCA

El método PCA previamente presentado sirve para realizar una transformación de los datos de entrada a otra base y así trabajar en otro espacio con mejores propiedades que el original. Por lo tanto, el proceso completo de clasificación se puede resumir como:

Dada una nueva imagen x de una cara, se calcula $\mathbf{tc}(x)$, y se compara con $\mathbf{tc}(x_i)$, $\forall x_i \in \mathcal{D}$, para luego clasificar mediante kNN.

Validación cruzada

Finalmente, nos concentramos en la evaluación de los métodos y en la correcta elección de sus parámetros.

Dado que necesitamos conocer previamente a qué persona corresponde una imagen para poder estimar la correctitud de la clasificación, una alternativa es particionar la base de entrenamiento en dos, utilizando una parte de ella en forma completa para el entrenamiento y la restante como test, pudiendo así corroborar la clasificación realizada, al contar con el etiquetado del entrenamiento. Sin embargo, realizar toda la experimentación sobre una única partición de la base podría resultar en una incorrecta estimación de parámetros, dando lugar al conocido problema de *overfitting*.

Por lo tanto, se estudiará la técnica de *cross validation*[2], en particular el *K-fold cross validation*[1], para realizar una estimación de los parámetros de los métodos que resulte estadísticamente más robusta.

Enunciado

Se pide implementar un programa en `C` o `C++` que lea desde archivos las imágenes de entrenamiento correspondientes a distintas caras y que, utilizando los métodos descriptos en la sección anterior, dada una nueva imagen de una cara determine a qué persona pertenece.

Para ello, el programa **deberá** implementar el algoritmo de kNN así como también la reducción de dimensión utilizando PCA.

Con el objetivo de obtener la transformaciones características de cada método, **se deberá** implementar el método de la potencia con deflación para la estimación de autovalores/autovectores de la matriz de covarianza en el caso de PCA.

Se recomienda realizar tests para verificar la implementación del método en casos donde los autovalores y autovectores sean conocidos se antemano. También, puede resultar de utilidad comparar con los datos provistos por la cátedra, utilizando Matlab/Octave o alguna librería de cálculo numérico.

[1] No confundir el K de K-fold con el k de kNN, ambos son parámetros de los métodos respectivos que no están relacionados necesariamente

Se pide realizar un estudio experimental de los métodos propuestos sobre una base de entrenamiento utilizando la técnica *K-fold cross validation* mencionada anteriormente, con el objetivo de analizar el poder de clasificación y encontrar los mejores parámetros de los métodos.

Se pide desarrollar una herramienta alternativa que permita trabajar bajo ciertas condiciones con imágenes de tamaño mediano/grande y probar lo siguiente:

- Dada una matriz de covarianza $M = X^t * X$, encontrar una relación entre sus autovalores y sus autovectores con los de la matriz $\hat{M} = X * X^t$.

Se **deberá** trabajar al menos con la siguiente base de datos caras:
`http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html`.

Experimentación

Para guiar la experimentación, se detallan los siguientes lineamientos y preguntas:

- Analizar la calidad de los resultados obtenidos al combinar kNN con y sin PCA, para un rango amplio de combinaciones de valores de k y α. Llamamos k a la cantidad de vecinos a considerar en el algoritmo kNN y α a la cantidad de componentes principales a tomar.
- Analizar la calidad de los resultados obtenidos al combinar kNN con PCA, para un rango amplio de cantidades de imágenes de entrenamiento. Utilizar desde muy pocas imágenes de entrenamiento hasta todas las disponibles para identificar en que situación se comporta mejor cada uno de los métodos.
- ¿Cómo se relaciona k con el tamaño del conjunto de entrenamiento? Pensar el valor máximo y mínimo que puede tomar k y qué sentido tendrían los valores.

También, **se debe** considerar en los análisis anteriores el tiempo de ejecución.

La calidad de los resultados obtenidos será analizada mediante diferentes métricas:

1. Accuracy
2. Precision/recall
3. F1-Score

En particular, la métrica más importante que **debe** reportarse en los experimentos es la tasa de efectividad lograda o *accuracy*, es decir, la cantidad caras correctamente clasificadas respecto a la cantidad total de casos analizados. También, se **debe** utilizar al menos otra de las métricas mencionadas, aunque no necesariamente para todos los experimentos realizados.

- Realizar los experimentos de los ítems anteriores para valores distintos de K del método K-fold[2], donde K a la cantidad de particiones consideradas para el cross-validation.

[2]Para esta tarea en particular, se recomienda leer la rutina `cvpartition` provista por Octave/MATLAB.

    - Justificar el por qué de la elección de los mismos. ¿Cuál sería su valor máximo?
    - ¿En qué situaciones es más conveniente utilizar K-fold con respecto a no utilizarlo?
    - ¿Cómo afecta el tamaño del conjunto de entrenamiento?
- En base a los resultados obtenidos para ambos métodos, seleccionar aquella combinación de parámetros que se considere la mejor alternativa, con su correspondiente justificación, compararlas entre sí y sugerir un método para su utilización en la práctica.

En todos los casos es **obligatorio** fundamentar los experimentos planteados, proveer los archivos e información necesaria para replicarlos, presentar los resultados de forma conveniente y clara, y analizar los mismos con el nivel de detalle apropiado. En caso de ser necesario, es posible también generar instancias artificiales con el fin de ejemplificar y mostrar un comportamiento determinado.

Puntos opcionales (no obligatorios)

- Mostrar que si tenemos la descomposición $M = U\Sigma V^t$, V es la misma matriz que obtenemos al diagonalizar la matriz de covarianzas.
- Realizar experimentos utilizando otras imágenes de caras tomadas por el grupo. Tener en cuenta lo mencionado sobre el tamaño de las matrices a procesar con PCA. Reportar resultados y dificultades encontradas.
- Proponer y/o implementar alguna mejora al algoritmo de kNN. Por ejemplo, no considerar votación y utilizar la cercanía a la media de de cada clase como criterio de clasificación.
- Implementar y experimentar un métodos de detección de caras para encontrar si una imagen contiene o no una cara. Proponer un valor de confianza para la respuesta de la detección. Es decir, se quiere intentar responder la pregunta de si el sistema es capaz no solo de reconocer caras con los que fue entrenado, sino si es posible discernir entre una imagen de una cara que no se encontraba en la base de entrenamiento y una imagen con un objeto o contenido que no sea una cara.
- Aplicar técnicas de procesamiento de imágenes a las imágenes de caras previo a la clasificación[1]. Analizar como impacta en la clasificación la alteración de la intensidad de los píxeles, el ruido introducido en las imágenes y la variación de la ubicación y posición de las personas.

Formato de entrada/salida

El ejecutable producido por el código fuente entregado deberá contar con las funcionalidades pedidas en este apartado. El mismo deberá tomar al menos tres parámetros por línea de comando con la siguiente convención:

```
$ ./tp2 -m <method> -i <train_set> -q <test_set> -o <classif>
```

donde:

- `<method>` el método a ejecutar con posibilidad de extensión (0: kNN, 1: PCA + kNN, ... etc)
- `<train_set>` será el nombre del archivo de entrada con los datos de entrenamiento.
- `<test_set>` será el nombre del archivo con los datos de test a clasificar
- `<classif>` el nombre del archivo de salida con la clasificación de los datos de test de `<test_set>`

Todos los archivos de entrada/salida deberán estar en `.csv` y siguiendo el formato siguiente:

```
<archivo1>, id1,
 ...

<archivoN>, idN,
```

donde `idX` es un entero positivo entre 1 y la cantidad de personas en el dataset.

Un ejemplo de invocación con los datos provistos por la cátedra sería el siguiente:

```
$ ./tp2 -m 1 -i train.csv -q test.csv -o result.csv
```

Además, el programa deberá imprimir por consola un archivo, cuyo formato queda a criterio del grupo, indicando la tasa de reconocimiento obtenida para cada conjunto de test y los parámetros utilizados para los métodos.

Nota: cada grupo tendrá la libertad de extender las funcionalidades provistas por su ejecutable. En particular, puede ser de utilidad alguna variante de toma de parámetros que permita entrenar con un porcentaje de la base de datos de entrenamiento y testear con el resto (ver archivos provistos por la cátedra). Además, puede ser conveniente separar la fase de entrenamiento de la de testeo/consulta para agilizar los cálculos.

Fecha de entrega

- Formato Electrónico: Viernes 25 de Mayo de 2018 hasta las 23:59 hs, enviando el trabajo (informe + código) a la dirección `metnum.lab@gmail.com`. El subject del email debe comenzar con el texto `[TP2]` seguido de la lista de apellidos de los integrantes del grupo separados por punto y coma `;`.
  Se ruega no sobrepasar el máximo permitido de archivos adjuntos de 20MB. Tener en cuenta al realizar la entrega de no ajuntar bases de datos disponibles en la web, resultados duplicados o archivos de backup.
- Formato físico: Lunes 28 de Mayo de 2018 a las 18 hs. en la clase de laboratorio.
- Pautas de laboratorio: `http://www-2.dc.uba.ar/materias/metnum/homepage.html`

Importante: El horario es estricto. Los correos recibidos después de la hora indicada serán considerados re-entrega.

Referencias

[1] Tinku Acharya and Ajoy K Ray. *Image processing: principles and applications*. John Wiley & Sons, 2005.

[2] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012.

[3] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of cognitive neuroscience*, 3(1):71–86, 1991.

4.2. Código

```
int main(int argc, char **argv) {
  srand(time(NULL));
    bool metodoConPCA = false;
    bool metodoAlternativo = false;
    char *entrenamiento = NULL;
    char *test = NULL;
   char *salida = NULL;
    leerArgumentos(argc,argv,metodoConPCA,metodoAlternativo,&entrenamiento,&test,&salida);
    vector<imagen> imagenesParaEntrenar = leerArchivo(entrenamiento);
    vector<imagen> imagenesAClasificar = leerArchivo(test);
    if(metodoConPCA){
      unsigned int alfa = 41;

      // x nxm
      matrix x = matrix(imagenesParaEntrenar);
      // mu mx1
      vector<float> mu = x.vector_promedio();
      x.resta_matrix_vector(mu);
      x.division_escalar(sqrt(x.dame_filas()-1));
      // xt mxn
      matrix xt = x.trasponer();

      // Mtodo  alternativo: v mxAlpha
      //Sino: v mxm
      matrix v = matrix(x.dame_columnas(), metodoAlternativo ? alfa : x.dame_columnas());

      if(metodoAlternativo) {
          // mx nxn ya que se multiplica x*xt
          matrix mx = matrix(x.dame_filas(),x.dame_filas());
          mx.multiplicacion(x,xt);
          // u nxn
          matrix u = matrix(mx.dame_filas(),mx.dame_filas());
          // d nxn
          matrix d = matrix(mx.dame_filas(),mx.dame_filas());
          mx.generacion_U_D(u,d,alfa);
          // v m x alpha
          xt.conversionUaV(u,d,v);
          // vt alpha x m
          v = v.trasponer();
      } else {
          // mx mxm ya que se multiplica xt*x
          matrix mx = matrix(x.dame_columnas(),x.dame_columnas());
          mx.multiplicacion(xt,x);
          // v mxm, d mxm
          matrix d = matrix(x.dame_columnas(),x.dame_columnas());
          mx.generacion_U_D(v,d,alfa);
          // vt m x m
          v = v.trasponer();
      }

      // aplico el cambio de base a las imagenes
      for(int i = 0; i < imagenesParaEntrenar.size(); i++){
        imagenesParaEntrenar[i].calcularXRaya(mu,imagenesParaEntrenar.size());
         // aplico tc
        // tc alpha x 1
         matrix tc = aplicarTc(imagenesParaEntrenar[i], v);
```

```
        imagenesParaEntrenar[i].setData(tc.dameMatriz());
      }
      for (int i = 0; i < imagenesAClasificar.size(); i++){
        // calculo x(raya)*
        imagenesAClasificar[i].calcularXRaya(mu,imagenesParaEntrenar.size());
        // aplico tc
        matrix tc = aplicarTc(imagenesAClasificar[i], v);
        imagenesAClasificar[i].setData(tc.dameMatriz());
      }
    }
    vector<tuple<string,int>> solucion = knn(imagenesParaEntrenar,imagenesAClasificar,9);
    escribirArchivo(salida,solucion);
}

class matrix {
public:
    // Fila& operator[] (unsigned int x) {
    //   return filas[x];
    // }
    ~matrix();
    matrix(unsigned int n,unsigned int m);
    matrix(vector<imagen> imagenes);
    void agregar_elemento(uint fila, uint columna, float elemento);
    void mostrar();
    matrix trasponer();
    void resta_matrix_vector(vector<float> &v);
    void multiplicacion_escalar(float escalar);
    void division_escalar(float escalar);
    void multiplicacion(matrix &A,matrix &B);
    unsigned int dame_filas();
    unsigned int dame_columnas();
    float dame_elem_matrix(unsigned int fila, unsigned int columna);
    vector<float> vector_promedio();
    void normalizar();
    void normalizar_2();
    float metodo_potencia(matrix &x, int repeticiones, matrix &autovector);
    void generacion_U_D(matrix& autovectores,matrix& autovalores, int alfa);
    bool verificacion(matrix autovector, float lambda);
    void restar(matrix&A);
    void absoluto();
    void conversionUaV(matrix& U,matrix &D,matrix &V);
    bool comparar(matrix &b);
    void rellenar_columna_con_vector(uint columna, matrix& V);
    void deflacion(matrix &autovector, float autovalor);
    vector<vector<float>> dameMatriz();
private:
    vector<vector <float> > matriz;
    unsigned int filas;
    unsigned int columnas;
};

//funciones auxiliares



float norma_Inf(matrix &v) {
  float max = 0;
  for (unsigned int i = 0; i < v.dame_filas(); i++) {
```

```
    if (abs(v.dame_elem_matrix(i, 0)) > max) {
      max = abs(v.dame_elem_matrix(i, 0));
    }
  }
  return max;
}

float norma_euclidea_cuadrada(matrix &A, matrix &B) {
  matrix R(1, 1);
  R.multiplicacion(A, B);
  return R.dame_elem_matrix(0, 0);
}

float norma_2(matrix& A) {
  assert(A.dame_columnas() == 1);
  float sumatoria = 0;
  for (size_t i = 0; i < A.dame_filas(); i++) {
    sumatoria = sumatoria + pow(A.dame_elem_matrix(i,0),2);
  }
  return sqrt(sumatoria);
}

matrix crear_canonico(uint filas,uint i){
 matrix a(filas,1);
 a.agregar_elemento(i,0,1);
 return a;
}

float matrix::metodo_potencia(matrix &x, int repeticiones, matrix &autovector) {
  matrix v = x;
  unsigned int i = 0;
  float autovalor = 0;
  do{

    autovector.multiplicacion((*this), v);
    autovector.normalizar_2();
    v = autovector;

   matrix transpuesta_v = v.trasponer();
   float norma_cuadrada = norma_euclidea_cuadrada(transpuesta_v, v);

   matrix C((*this).dame_filas(), 1);
   C.multiplicacion((*this), v);
   matrix D(1, 1);
   D.multiplicacion(transpuesta_v, C);

   autovector = v;

   D.division_escalar(norma_cuadrada);
   autovalor = D.dame_elem_matrix(0, 0);
    ++i;
  }while (i < repeticiones && !verificacion(autovector,autovalor));

  return autovalor;
}

float dame_random() {
  float random;
```

```
  do {
    random = distribution(generator);
  } while(abs(random) <= EPSILON);
}

// constructor de una matrix de tamao n llena de ceros

matrix::~matrix() {}

matrix::matrix(unsigned int filas, unsigned int columnas) {

  matriz.resize(filas);
  for (unsigned int i = 0; i < filas; i++) {
    matriz[i].resize(columnas);
  }
  for (size_t i = 0; i < filas; i++) {
    for (size_t j = 0; j < columnas; j++) {
      matriz[i][j] = 0;
    }
  }
  this->filas = filas;
  this->columnas = columnas;
}

matrix::matrix(vector<imagen> imgs){
  columnas = imgs[0].tamanio(); // asumo que todas las imagenes tienen el mismo tamao
  filas = imgs.size();
  matriz.resize(imgs.size());
  for (size_t i = 0; i < imgs.size(); i++) {
    vector<float> actual = imgs[i].data();
    for (size_t j = 0; j < imgs[0].tamanio(); j++) {
      matriz[i].push_back((float)actual[j]);
    }
  }
}

unsigned int matrix::dame_filas() {
  return this->filas;
}

unsigned int matrix::dame_columnas() {
  return this->columnas;
}

matrix matrix::trasponer() {
  matrix traspuesta = matrix(this->columnas,this->filas);
  for (size_t i = 0; i < filas; i++) {
    for (size_t j = 0; j < columnas; j++) {
      traspuesta.agregar_elemento(j, i, dame_elem_matrix(i, j));
    }
  }
  return traspuesta;
}

vector<float> matrix::vector_promedio() {
  vector<float> promedio;
  promedio.resize(dame_columnas());
  for (size_t i = 0; i < columnas; i++) {
```

```
    float sumatoria = 0;
    for (size_t j = 0; j < filas; j++) {
      sumatoria = sumatoria + dame_elem_matrix(j, i);
    }
    promedio[i] = sumatoria / dame_filas();
  }
  return promedio;
}

void matrix::resta_matrix_vector(vector<float> &v) {
  for (size_t j = 0; j < dame_columnas(); j++) {
    for (size_t i = 0; i < dame_filas(); i++) {
      float elemento = dame_elem_matrix(i, j);
      agregar_elemento(i, j, elemento - v[j]);
    }
  }
}

void matrix::division_escalar(float escalar) {
  //cout << escalar << "\n";
  //assert(abs(escalar) > EPSILON);
  for (size_t i = 0; i < dame_filas(); i++) {
    for (size_t j = 0; j < dame_columnas(); j++) {
      float division = dame_elem_matrix(i, j) / escalar;
      agregar_elemento(i, j, division);
    }
  }
}


void matrix::multiplicacion_escalar(float escalar) {
  //cout << escalar << "\n";
  //assert(abs(escalar) > EPSILON);
  for (size_t i = 0; i < dame_filas(); i++) {
    for (size_t j = 0; j < dame_columnas(); j++) {
      float division = dame_elem_matrix(i, j) * escalar;
      agregar_elemento(i, j, division);
    }
  }
}

float producto_interno(matrix &A, matrix &B, unsigned int fila, unsigned int columna) {
  float resultado = 0;
  for (size_t i = 0; i < A.dame_columnas(); i++) {
    resultado = resultado + A.dame_elem_matrix(fila, i) * B.dame_elem_matrix(i, columna);
  }
  return resultado;
}

void matrix::multiplicacion(matrix &A, matrix &B) {
  assert(A.dame_columnas() == B.dame_filas());
  for (size_t i = 0; i < this->dame_filas(); i++) {
    for (size_t j = 0; j < this->dame_columnas(); j++) {
      this->agregar_elemento(i, j, producto_interno(A, B, i, j));
    }
  }
}
```

```
void matrix::mostrar() {
  std::cout << '\n';
  for (unsigned int i = 0; i < filas; i++) {
    for (unsigned int j = 0; j < columnas; j++) {
      std::cout << dame_elem_matrix(i, j) << ' ';
    }
    std::cout << '\n';
  }
  std::cout << '\n';
}

float matrix::dame_elem_matrix(unsigned int fila, unsigned int columna) {
  return matriz[fila][columna];
}


void matrix::agregar_elemento(uint fila, uint columna, float elemento) {
  if (abs(elemento) < EPSILON) {
    matriz[fila][columna] = 0;
  } else {
    matriz[fila][columna] = elemento;
  }
}

void matrix::normalizar() {
  assert(columnas == 1);
  float norma = norma_Inf(*this);
  division_escalar(norma);
}
void matrix::normalizar_2() {
  assert(columnas == 1);
  float norma = norma_2(*this);
  division_escalar(norma);
}

bool matrix::verificacion(matrix autovector, float autovalor){
  matrix a(autovector.dame_filas(),1);
  a.multiplicacion((*this),autovector),
  autovector.multiplicacion_escalar(autovalor);
  return autovector.comparar(a);
}

bool matrix::comparar(matrix& b){
  matrix a(*this);
  a.restar(b);
  a.absoluto();
  for (size_t i = 0; i < a.dame_filas(); i++) {
    for (size_t j = 0; j < a.dame_columnas(); j++) {
      float elemento = a.dame_elem_matrix(i,j);
      if (elemento >= EPSILON) {
        return false;
      }
    }
  }
  return true;
}

void matrix::restar(matrix&A){
```

```
  assert(dame_filas() == A.dame_filas());
  assert(dame_columnas() == A.dame_columnas());
  for (size_t i = 0; i < dame_filas(); i++) {
    for (size_t j = 0; j < dame_columnas(); j++) {
      float elemento = A.dame_elem_matrix(i, j);
      elemento = dame_elem_matrix(i, j) - elemento;
      agregar_elemento(i, j, elemento);
    }
  }
}

void matrix::absoluto(){
  for (size_t i = 0; i < dame_filas(); i++) {
    for (size_t j = 0; j < dame_columnas(); j++) {
      float elemento = abs(dame_elem_matrix(i, j));
      agregar_elemento(i, j, elemento);
    }
  }
}

void matrix::deflacion(matrix &autovector, float autovalor){
  autovector.normalizar_2();
  matrix autovector_traspuesto = autovector.trasponer();
  matrix B(dame_filas(),dame_columnas());
  B.multiplicacion(autovector,autovector_traspuesto);
  B.multiplicacion_escalar(autovalor);
  restar(B);
}


void matrix::generacion_U_D(matrix& U,matrix& D, int alfa){
  assert(dame_filas() == dame_columnas());
  assert(dame_filas() == U.dame_columnas());
  assert(U.dame_filas() == U.dame_columnas());
  assert(D.dame_filas() == D.dame_columnas());
  assert(D.dame_filas() == U.dame_columnas());

  matrix autovector(dame_filas(),1);
  matrix x_0(dame_filas(),1);
  for (size_t i = 0; i < alfa; i++) {
    float autovalor = 0;
    //genera vector random
    // TODO: hacer un vector inicial con la media de la matriz
    for (size_t j = 0; j < x_0.dame_filas(); j++) {
       x_0.agregar_elemento(j,0,rand()%100+1);
    }
    x_0.normalizar_2();

    autovalor = this->metodo_potencia(x_0,500,autovector);

    //si no son parecidos, cambiamos la semilla del vector
    //hacer deflacion

    U.rellenar_columna_con_vector(i, autovector);
    D.agregar_elemento(i, i, autovalor);
    this->deflacion(autovector,autovalor);
  }
}
```

```
void matrix::rellenar_columna_con_vector(uint columna, matrix& V){
  //asume que X (this) viene ya traspuesto
  V.normalizar_2();
  assert(V.dame_columnas() == 1);
  assert(V.dame_filas() == dame_filas());
  for (size_t i = 0; i < dame_filas(); i++) {
    agregar_elemento(i, columna, V.dame_elem_matrix(i, 0));
  }
}
//devuelve una matrix de nxm
void matrix::conversionUaV(matrix& U,matrix &D,matrix &V) {
  for (size_t i = 0; i < V.dame_columnas(); i++) {
    matrix e_i = crear_canonico(U.dame_filas(),i);
    // e_i.mostrar();
    matrix u_i(U.dame_filas(),1);
    u_i.multiplicacion(U,e_i);
    // u_i.mostrar();
    // D.mostrar();
    float d_i_i = sqrt(abs(D.dame_elem_matrix(i,i)));
    // std::cout << "d " << d_i_i<<'\n';
    // mostrar();
    matrix v_i(dame_filas(),1);
    v_i.multiplicacion((*this),u_i);
    // v_i.mostrar();
    v_i.division_escalar(d_i_i);
    //v_i.normalizar_2();
    V.rellenar_columna_con_vector(i,v_i);
  }
}

matrix aplicarTc(imagen a, matrix &v){
  vector<imagen> aux;
  aux.push_back(a);
  matrix x = matrix(aux);
  x = x.trasponer();
  matrix resultado = matrix(v.dame_filas(),x.dame_columnas());
  resultado.multiplicacion(v,x);
  return resultado;
}

vector<vector<float>> matrix::dameMatriz(){
  return matriz;
}

class imagen{
  public:
    imagen(string nombre, int identificador){
      id = identificador;
      ruta = nombre;
      uchar *data = NULL;
      width = 0;
      height = 0;
      PPM_LOADER_PIXEL_TYPE pt = PPM_LOADER_PIXEL_TYPE_INVALID;
      LoadPPMFile(&data, &width, &height, &pt, ruta.c_str());
      for (int i = 0; i < width*height; ++i){
        datos.push_back((float)data[i]);
      }
```

```
    // delete [] data;
  }

  ~imagen(){
  }

  int getId(){
    return id;
  }

  string getPath(){
    return ruta;
  }

  float distancia(imagen img){
    float respuesta = 0;
    for (size_t i = 0; i < this->tamanio(); i++) {
      respuesta += (img.datos[i]-this->datos[i])*(img.datos[i]-this->datos[i]);
    }
    return sqrt(respuesta);
  }

  int tamanio(){
    return height*width;
  }

  vector<float> data(){
    return datos;
  }

  void setData(vector<vector<float>> M_datos){
    for (int i = 0; i < M_datos.size(); ++i){
      for (int j = 0; j < M_datos[i].size(); ++j){
        datos[i+j] = M_datos[i][j];
      }
    }
  }

  void calcularXRaya(vector<float> mu,int cantidadDeImagenes){
    for(int i = 0; i < this->tamanio(); ++i){
      datos[i] = (datos[i]-mu[i])/sqrt(cantidadDeImagenes-1);
    }
  }

private:
  string ruta;
  vector<float> datos;
  int height;
  int width;
  int id;
  PPM_LOADER_PIXEL_TYPE pt;

protected:

};
```