DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico №2

Cara a cara

25 de Mayo de 2017 — Métodos Numéricos

Integrante	LU	Correo electrónico
Alvarez, Ezequiel Ángel	421/13	ezequiel.a.alvarez@gmail.com
Cámera, Joel Esteban	257/14	joel.e.camera@gmail.com
Sicardi, Sebastián Matías	042/13	smcsicardi@gmail.com

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://exactas.uba.ar

Índice

1. Introducción

Reconocer rostros es una tarea sencilla para los humanos. Desde la más temprana edad ya se pueden reconocer y son nuestro primario foco de atención en combinación con la identidad y la emoción. Podemos reconocer rostros con años de diferencia entre haberlos visto sin importar los diferentes estímulos visuales, las expresiones, el paso de la edad y las diferencias que puede tener como un cambio en el peinado, lentes o barba. Por lo tanto surge la siguiente pregunta ¿qué tan difícil puede ser para una computadora este tipo de reconocimiento?

Desafortunadamente, generar un modelo computacional de reconocimiento de rostros es difícil debido a que los mismos son complejos, multidimensionales y presentan estímulos visuales significativos.

En el presente trabajo nos propondremos a analizar cómo lograr un sistema de reconocimiento de caras usando reducción de dimensionalidad a partir de un algoritmo de aprendizaje automático. O, expresado en forma coloquial, debemos generar un *clasificador* tal que con una base de datos etiquetada de imágenes de rostros de personas tomadas de forma particular, y dado una nueva imagen de un rostro poder decidir si pertenece a una persona de la base.

2. Desarrollo

2.1. Aproximación Por Autocaras

En este trabajo buscamos extraer la información relevante de la imagen de un rostro, codificarla lo mas eficientemente posible y comparar esta codificación con una base de datos que esta codificada de forma similar. Un enfoque simple para extraer información contenida en una imagen de un rostro es capturar la variación en una colección de imágenes de rostros, independientemente de las características, y usar esta información para codificar y comparar imágenes de rostros individuales.

En términos matemáticos, buscamos encontrar los componentes principales de la distribución de rostros, o los autovectores de la matriz de covarianza del conjunto de imágenes de rostros, tratando cada imagen como un vector en espacio de dimensión alta. Los autovectores están ordenados, y cada uno cuenta con una diferente cantidad de variación entre las imágenes de los rostros.

Estos autovectores se pueden pensar como un conjunto de características que unidas caracterizan la variación entre las imágenes de los rostros. Cada posición en la imagen contribuye más o menos a cada autovector, por lo tanto se pueden mostrar a los autovectores como una imágenes "fantasmagoricas" de caras que denominamos *autocara*. Un ejemplo de las mismas puede observarse en la figura 1. Cada autocara difiere entre el conjunto de imágenes de entrenamiento.

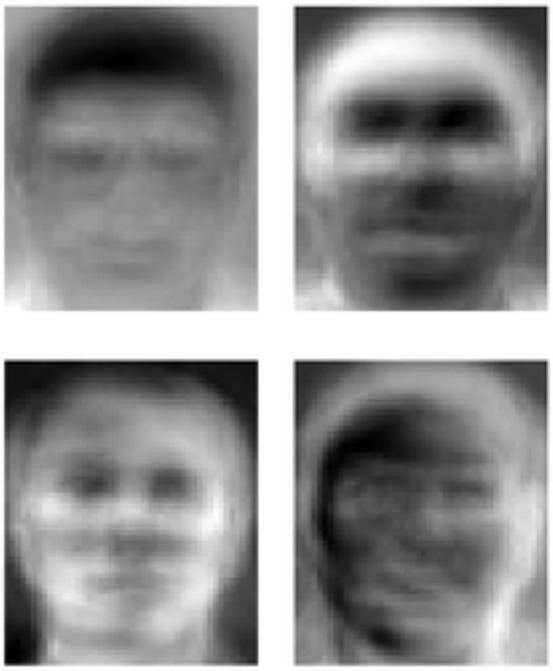


Figura 1: Algunas autocaras de AT&T Laboratories Cambridge

Cada cara puede ser representada exactamente en términos de una combinación lineal de autocaras. También, cada cara se puede aproximar usando solamente las "mejores" autocaras (aquellas que tienen los autovalores de valor más altos y, por lo tanto, cuentan con la mayor varianza dentro del conjunto de las imágenes de los rostros). [1]

2.2. Análisis por componentes principales

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 del mismo tamaño. Si consideramos a cada imagen como un vector

[1] Estos párrafos fueron extraídos del paper **Eigenfaces for Recognition** de Matthew Turk and Alex Pentland. Página 73.

en $\mathbb{R}^m$, tenemos $N \times M$ vectores.

El problema entonces radica en que las imágenes se encuentran en un espacio de dimensión alta, una imagen de 100×100 se convierte en un vector en $\mathbb{R}^{10,000}$. Trabajar en una dimensión tan alta es costoso y se ve afectado por la *La maldición de la dimensionalidad*, lo que significa que la densidad que puedan tener los datos en este espacio seguramente sea muy baja. Teniendo en cuenta esto, lo que buscamos son los componentes que cuentan con la mayor parte de la información para trabajar con una cantidad de variables más acotada. Para esto utilizaremos el método de *Análisis de Componentes Principales* (PCA, por sus sigla en inglés).

Éste método consiste en lo siguiente: para $i = 1, ..., n, x_i \in \mathbb{R}^m$ corresponde a la i-ésima imagen de nuestra base de datos almacenada por filas en un vector y sea $\mu = (x_1 + ... + x_n)/n$ el promedio de las imágenes. Definimos la matriz $X \in \mathbb{R}^{n \times m}$ como la matriz que contiene en la i-ésima fila al vector $(x_i - \mu)^t/\sqrt{n-1}$, y a la matriz

$$M = X^t X$$

como la matriz de covarianza de la muestra X. Siendo v_j el autovector de M asociado al j-ésimo autovalor al ser ordenados por su valor absoluto, definimos para $i = 1, ..., n$ la *transformación característica* de la cara x_i como el vector $\mathbf{tc}(x_i) = (v_1^t x_i, v_2^t x_i, ..., v_\alpha^t x_i) \in \mathbb{R}^\alpha$, 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 intención es que $\mathbf{tc}(x_i)$ resuma la información más relevante de la imagen, descartando los detalles o las zonas que no aportan rasgos distintivos.

Esto funciona ya que la matriz $M = X^t X$ es simétrica y por teorema sabemos que si la matriz es simétrica los autovalores pertenecen a $\mathbb{R}$, los autovectores asociados a $\mathbb{R}^n$ y son base ortonormal. Entonces no va a haber problemas en encontrar los componentes principales para generar la transformación característica.

Dada una nueva imagen y de una cara, que no se encuentra en el conjunto inicial de imágenes de entrenamiento, el problema de reconocimiento consiste en determinar a qué persona de la base de datos corresponde. Para esto, se calcula $\mathbf{tc}(y)$ y se compara con $\mathbf{tc}(x_i)$, para $i = 1, ..., n$, utilizando un criterio de selección adecuado.

2.2.1. Optimización sobre la matriz M

Si observamos un poco a la matriz X podemos ver que sus dimensiones son de cantidad de imágenes (C) x dimensión de las imágenes (D). Con esto podemos notar que la matriz $M = X^t X$ pertenece a $\mathbb{R}^{D \times D}$ ya que $X^t \in \mathbb{R}^{D \times C}$ y $X \in \mathbb{R}^{C \times D}$. Si sabemos que la cantidad de imágenes es mucho menor que la dimensión de cada una de estas ($C < D$) entonces van a haber $C - 1$ autovectores significativos (Los restantes autovectores van a estar asociados al autovalor cero).

Por lo tanto, si consideramos la matriz $\hat{M} = XX^t$ va a pertenecer a $\mathbb{R}^{C \times C}$, o sea, va a ser mucho más chica y más rápido calcularla.

Veamos un ejemplo, supongamos que queremos generar una base de datos con 100 imágenes de 92x112 píxeles cada una. Entonces, como la matriz X tiene de dimensión la cantidad de imágenes por la dimensión de cada una, $X \in \mathbb{R}^{100 \times D}$ donde $D = 92 \times 112 = 10304$, o sea, $X \in \mathbb{R}^{100 \times 10304}$. Si calculamos $M = X^t X$ la dimensión de la matriz es de 10304x10304, en cambio si calculamos $\hat{M} = XX^t$ la dimensión de la matriz es de 100x100. Con este simple ejemplo vemos que los cálculos se reducen enormemente.

Pero, ¿qué relación tiene esta matriz con los autovalores y autovectores de M? Si consideramos a λ_i como el autovalor i-ésimo de $\hat{M}$ y a u_i como el autovector asociado, tengo que

$$\hat{M} u_i = \lambda_i u_i$$
$$X X^t u_i = \lambda_i u_i$$

y si multiplico X^t por izquierda

$$X^t X X^t u_i = \lambda_i X^t u_i$$
$$X^t X (X^t u_i) = \lambda_i (X^t u_i)$$
$$M(X^t u_i) = \lambda_i (X^t u_i)$$

vemos que $X^t u_i$ es un autovector de la matriz $M = X^t X$ con el mismo λ_i autovalor. Por lo tanto, es mucho mas eficaz calcular los autovectores de $\hat{M}$ y luego hacer $X^t u_i$ para obtener los v_i autovectores de M.

Con este análisis, los cálculos se reducen enormemente, del orden del numero de píxeles de las imágenes (D) al orden del numero de imágenes del conjunto de entrenamiento (C). En la práctica, el conjunto de imágenes de entrenamiento suele ser mucho mas chico que el de las dimensiones de cada una de estas ($C < D$) y los cálculos se vuelven más manejables.

2.2.2. Método de la potencia y método de deflación

Para ir obteniendo los autovalores y autovectores de la matriz $M = X^t X$ utilizamos los **métodos de la potencia** y **deflación**.

Método de la potencia: Sea $A \in \mathbb{R}^{n \times n}$, sean $\lambda_1, ..., \lambda_n$ sus autovalores y sean $v_1, ..., v_n$ los autovectores asociados a sus respectivos autovalores tal que forman una base y los autovalores cumplen que $|\lambda_1| > |\lambda_2| \geq ... \geq |\lambda_n|$, entonces el método de la potencia devuelve el autovalor dominante (el de valor absoluto más grande) y el autovector asociado al mismo. El método se realiza de la siguiente forma:

MetodoPotencia(A,x_o,niter)

$v \leftarrow x_o$
for $i = 1, ..., niter$ **do**
  $v \leftarrow \frac{Bv}{||Bv||}$
end for
$\lambda \leftarrow \frac{v^t B v}{v^t v}$
Devolver λ, v

Deflación: Sea $A \in \mathbb{R}^{n \times n}$ una matriz con autovalores $|\lambda_1| > |\lambda_2| > ... > |\lambda_n|$ y una base ortonormal de autovectores. Entonces la matriz $B - \lambda_1 v_1 v_1^t$ tiene autovalores $0, \lambda_2, ..., \lambda_n$ con autovectores asociados $v_1, ..., v_n$.

La prueba de esto es simple:

- $(B - \lambda_1 v_1 v_1^t) v_1 = B v_1 - \lambda_1 v_1 (v_1^t v_1) = \lambda_1 v_1 = 0 v_1$
- $(B - \lambda_1 v_1 v_1^t) v_i = B v_i - \lambda_1 v_1 (v_1^t v_i) = \lambda_1 v_i$

En nuestro caso ambos métodos funcionan ya que tanto la matriz $M = X^t X$ como la optimizada $\hat{M} = X X^t$ son matrices simétricas y por teorema sabemos que si la matriz es simétrica los autovalores pertenecen a $\mathbb{R}$, los autovectores asociados a $\mathbb{R}^n$ y son base ortonormal. Por lo tanto, para ir obteniendo los autovalores y autovectores asociados principales para generar la transformación característica, aplicamos sucesivamente estos dos métodos.

Queda por ver cuál sería el *niter* correcto para que el método de la potencia termine de manera satisfactoria, esto es un detalle de implementación que se explicará más adelante.

2.3. Clasificación

Ahora queremos saber, dado una nueva imagen de una cara a que clase (o persona) pertenece de la misma. Para esto utilizamos el algoritmo de los k vecinos más cercanos (k-nearest neighbours[2] en ingles).

[2] `https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm`

Éste clasifica puntos en un espacio encontrando la **clase más común** entre los **k puntos más cercanos**. En otras palabras, cada punto de los k mas cercanos vota por su clase, y la que tenga mas votos es la ganadora, en caso de empate se elige una clase al azar entre las más votadas. En lenguaje coloquial esto se resume a *"dime quienes son tus vecinos y te diré quien eres"*.

La clasificación se realiza sobre el nuevo espacio de dimensiones que devuelve la transformación característica. A todas las imágenes de la base de datos y a las nuevas imágenes se les aplica esta **tc**.

Para aplicar esta clasificación, necesitamos definir nuestra distancia entre vectores. En nuestro caso utilizamos la distancia euclideana:

$$\sqrt{\sum_{i=1}^{n}(y - x_i)^2}$$

siendo y la imagen de una nueva cara y x_i una de las imágenes de la base.

Este algoritmo posee algunos puntos fuertes:

- Es intuitivo.
- No asume ninguna distribución probabilística de los datos de entrada. Esto es útil para entradas donde su distribución es desconocida.
- Puede responder rápidamente a cambios en la entrada. El algoritmo utiliza aprendizaje vago o *lazy learning*[3] en ingles, y no generaliza los datos. Esto permite que se puedan introducir cambios al modelo más rápido.

Pero no es perfecto y también tiene algunos puntos débiles:

- Es sensible a la ubicación de los datos. Como obtiene su información de la base de datos, si alguno posee alguna anomalía (debido a un error numérico de calculo por ejemplo) afecta significativamente la salida.
- El tiempo de computo. El *lazy learning* requiere que la mayor parte del computo se realice durante la etapa de testeo y no durante la etapa de entrenamiento. Esto puede causar un problema cuando el conjunto de datos es muy grande.
- Dimensionalidad. En el caso de que haya muchas dimensiones, las entradas pueden estar "cerca" de muchos puntos. Esto reduce la efectividad, dado que el algoritmo depende de la correlación entre lo cercano y lo similar. Por esto realizamos PCA.

2.4. Validación

Con todo lo anterior podríamos, dada una base de datos etiquetada de imágenes de rostros y una imagen, saber a que persona de la base corresponde esta última. Pero, ¿cómo sabemos que el clasificador funciona bien? Evaluar el modelo en los datos de entrenamiento puede darnos una impresión errónea. Por ello utilizamos *k-Fold Cross Validation*[4].

En *k-Fold Cross Validation*, la muestra original se fracciona aleatoriamente en k submuestras del mismo tamaño. De las k submuestras, una se utiliza para validar los datos en la etapa de testeo y el resto de ellas para entrenamiento. El proceso de *cross validation* se repite k veces tal que cada submuestra se utilice para validar exactamente una vez como se puede observar en la figura 2.

[3] El aprendizaje vago o *lazy learning* es un método de aprendizaje en el que la generalización más allá de los datos de entrenamiento es demorada hasta que se hace una pregunta al sistema. `https://en.wikipedia.org/wiki/Lazy_learning`

[4] `https://en.wikipedia.org/wiki/Cross-validation_(statistics)#k-fold_cross-validation`

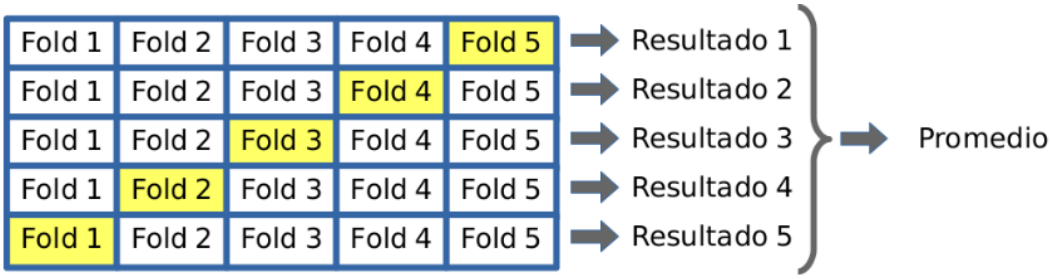


Figura 2: Ejemplo de k-Fold Cross Validation

Una vez que se obtienen los k resultados se realiza un promedio de ellos para obtener una estimación. La ventaja de este método sobre muchas repeticiones de es que todas las observaciones son utilizadas tanto para entrenamiento como para validación, y cada observación es utilizada para validación exactamente una vez.

2.5. Matriz de Confusión

A la hora de discernir si nuestro clasificador funciona bien debemos tener una métrica que sea comparable con otros algoritmos de la misma índole. Inicialmente es muy fácil pensar en la cantidad de imágenes clasificadas correctamente (*accuracy*) y compararlo, pero, ¿es esta métrica la mejor? Por ejemplo, si tenemos un análisis médico que detecta una enfermedad, se puede decir que acierta (dice que la persona tiene la enfermedad cuando la tiene y cuando está sano dice que lo está) en un 80 % de los casos, pero ¿qué pasa en ese 20 % restante? Podría resultar que ese 20 % consista de todas personas con la enfermedad pero que el análisis no las detecta, lo cual sería evidentemente muy grave. En comparación si ese 20 % consistiese de personas sin la enfermedad pero que da un "falso positivo", sería algo menos peligroso que en el otro caso.

Entonces tenemos en *accuracy* una métrica un tanto engañosa cuando lidiamos con este tipo de problemas, en el cual tenemos cuatro resultados posibles, *positivo verdadero* (detecta la enfermedad cuando la tiene), *negativo verdadero* (no la detecta si no la tiene), *falso positivo* (la detecta si no la tiene) y *falso negativo* (no la detecta y *si* la tiene), el caso grave.

Es por esto que utilizaremos la **matriz de confusión** la cual nos da una herramienta para poder comparar los cuatro valores y poder discernir entre el mejor clasificador de una forma más precisa. Por ejemplo, una experimentación del análisis explicado anteriormente podría dar como resultado el gráfico de la figura 3.

n=165	**Predicted: NO**	**Predicted: YES**
Actual: NO	50	10
Actual: YES	5	100

Figura 3: Ejemplo de Matriz de Confusión

De esta manera con este método obtenemos más información, lo cual resulta en un análisis más interesante. Este sistema lo podemos generalizar para aplicarlo a nuestro clasificador y ver como se comportan las distintas clases, como en el ejemplo de la figura 4

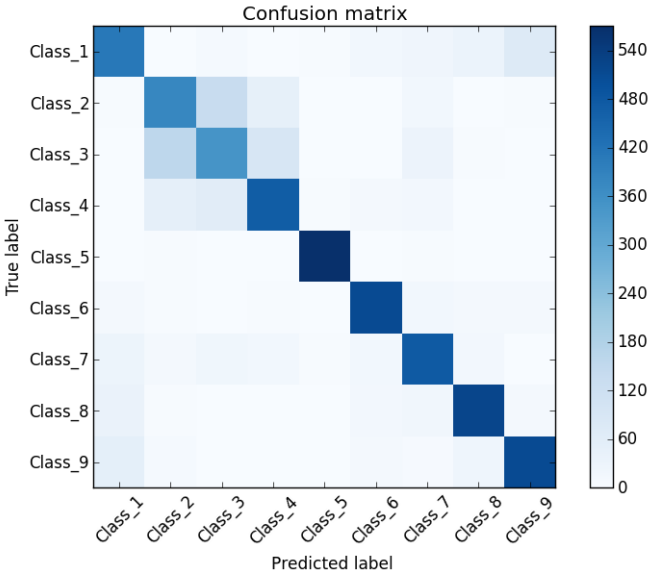


Figura 4: Ejemplo de Matriz de Confusión con varias clases

Vemos que de esta forma se obtiene mucha más información, como por ejemplo que las clases 2 y 3 tienden a ser confundidas por el clasificador, algo que no podríamos haber conocido de otra manera.

3. Implementación

A la hora de implementar el código decidimos no valernos de librerías externas para que nos manejen las matrices o las distintas estructuras de datos. Más bien, intentamos implementar todo nosotros desde cero valiéndonos solamente de lo que nos provee la librería estándar de C++, ya que no veíamos la necesidad de *pelearnos* con librerías cuando solamente estamos tratando con matrices que fácilmente podían ser implementadas como *vector de vectores*[5].

Con el propósito de mantener los datos ordenados y tener un mejor control de la información que toma el programa decidimos armar *structs*[6] con la información correspondiente que se necesitase. Por ejemplo, creamos la estructura *Matriz* el cual contiene información relevante como la cantidad de filas y columnas, aparte de los datos propiamente dichos.

Además creamos la estructura *Input* la cual contiene a todos los datos de entrada para una ejecución. Dicho de otra manera, dada una serie de imágenes de entrada en *Input* tenemos toda la información de éstas, como la cantidad de personas, el alto y ancho de las imágenes, el *path* de las mismas.

Una implementación importante que realizamos fue la detener una estructura que vincule a un autovector con su autovalor correspondiente, por ello hicimos la estructura $EigenVV$ que contiene ambos datos.

Una de las cosas más desafiantes fue realizar el procesamiento del formato PGM para obtener la información que necesitábamos. Lo que hicimos fue realizarlo según las especificaciones del mismo [7].

Ante la duda de que uno de los algoritmos sea erróneo por una *cancelación catastrófica* o por muchas fallas consecutivas de discretización de números racionales, decidimos implementar el algoritmo de suma de Kahan [8] con el objetivo de asegurarnos una sumatoria correcta.

Por último, se implementó una versión más rápida del algoritmo. Como ya se ha explicado en la sección anterior, la obtención de la matriz $\hat{M} = XX^t$ con los autovectores en vez de la que se explicitaba en el enunciado $M = X^tX$ con los autovectores, se obtienen los mismos resultados en un tiempo significativamente menor. Si se desea correr el algoritmo sobre la matriz $\hat{M}$ se corre *tp2* con el parámetro `--fast`, si no se le pasa ningún parámetro entonces corre el algoritmo sobre la matriz M.

3.1. Método de la Potencia

En la implementación del método de la potencia nos enfrentamos ante una importante decisión de implementación ¿Cuál es nuestro *criterio de parada*? Es decir, sabiendo que tenemos una función que tiende al autovalor dominante, ¿cómo hacemos para saber cuándo detener el algoritmo? Ante esta duda surgió una solución un tanto evidente: si estoy a una distancia menor que un ε lo detengo.

$$\|A\hat{x} - \lambda\hat{x}\| < \varepsilon$$

De esta forma nos aseguramos de obtener un vector lo más cercano posible a la solución real, con una tolerancia de un valor *epsilon*. Entonces lo que hicimos fue realizar 50 iteraciones del método de la potencia y luego chequear esta condición, en caso de no cumplirse se sigue iterando 50 veces más. Como resulta un tanto costoso calcular este valor (lo que llamaremos *residuo*) decidimos iterar una cierta cantidad de veces para ahorrarnos este costo. Con respecto a la tolerancia para la diferencia debíamos elegir un valor que nos pueda asegurar la mejor precisión posible sin tener que hacer una cantidad grosera de iteraciones. Por este motivo, se llegó a un compromiso entre precisión y eficiencia con una tolerancia de 0.01, lo cual nos dio resultados positivos en un tiempo acotado.

[5] `http://www.cplusplus.com/reference/vector/vector/`
[6] `http://www.cplusplus.com/doc/tutorial/structures/`
[7] `http://netpbm.sourceforge.net/doc/pgm.html`
[8] `https://en.wikipedia.org/wiki/Kahan_summation_algorithm`

3.2. K-Nearest Neighbours

Con el objetivo de mantener la implementación sencilla y directa, nuestro código asume en general que las personas se indexan desde 1.

La función de kNN toma al vector $X2$ de *Puntos* de entrenamiento, cada uno con su índice de persona/clase i y la transformación característica de sus coordenadas $tc(x_i)$. Toma también un *Punto* y que es la imagen a testear con su clase correcta y sus coordenadas transformadas, una cantidad k de vecinos a considerar y cuántas clases hay en total.

Luego procede a generar un vector de *distancias* entre cada *Punto* de $X2$ e y usando la distancia euclideana y guardando además a que clase pertenece.

Este vector es luego ordenado de forma descendiente respecto a la distancia y se recorren sus primeras k coordenadas, asignándole un voto a cada clase entre estas por cada vez que aparece.

Como se dijo previamente, no usamos ningún tipo de peso para considerar los votos.

4. Experimentación

4.1. Performance

Como un primer análisis decidimos medir la performance de la etapa de entrenamiento desde la generación de la matriz M y $\hat{M}$ hasta la obtención de los autovectores.

En este trabajo disponemos de la base de datos etiquetada **ORL faces**[9] correspondiente a 41 personas, con 10 imágenes por cada una de ellas. Esta base de datos se encuentra disponible en dos resoluciones distintas: 92×112 y 23×28 píxeles por cada imagen. La segunda correspondiente a un submuestreo de la base original.

Para comenzar, realizamos una medición de tiempos con las imágenes de 23×28. Dejamos fijas la cantidad de componentes en 16 y la cantidad de imágenes por persona en 10; y variamos la cantidad de personas entre 1 y 41. Dado la dimensionalidad de la matriz $\hat{M}$ contra la matriz M, la performance temporal sobre la primera debe ser mucho menor que la segunda ya que la matriz $M \in \mathbb{R}^{644 \times 644}$ en todos los casos, mientras que la matriz $\hat{M}$ se mueve entre $\mathbb{R}^{10 \times 10}$ con una persona, y $\mathbb{R}^{410 \times 410}$ con 41 personas, por lo tanto siempre es menor. Para este test, calculamos los tiempos de 100 corridas por cada persona agregada y luego tomamos el promedio de estos. Los resultados de éste experimento fueron los siguientes.

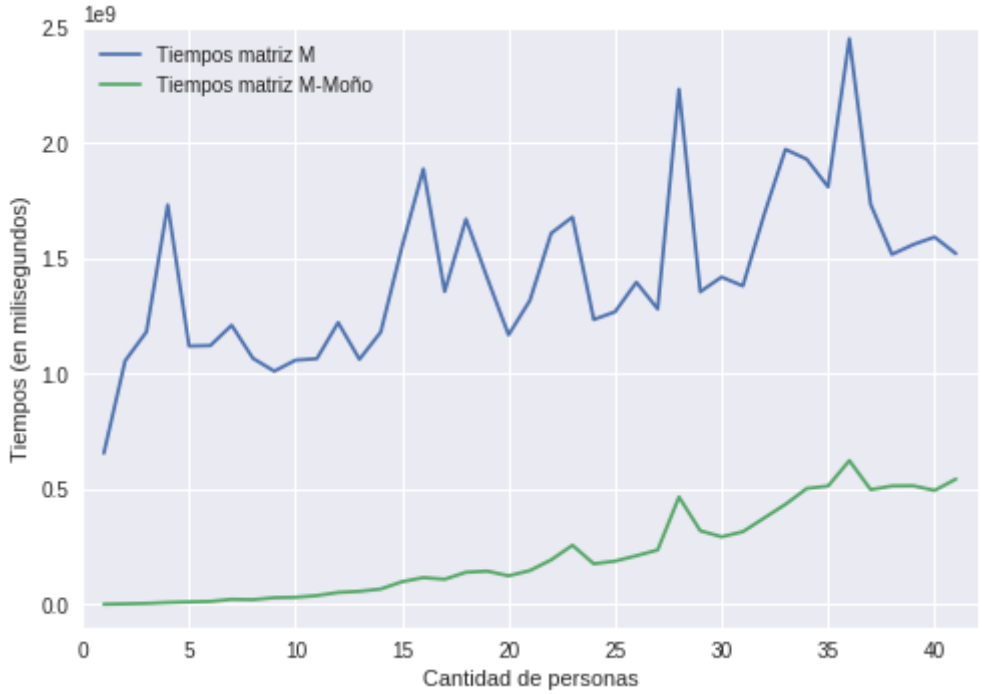


Figura 5: Comparación de performance temporal entre la matriz M y $\hat{M}$

Como se puede observar en la figura 5, los tiempos promedio de cómputo sobre la matriz $\hat{M}$ son mucho menores que los de la matriz M como se preveía. Algo que nos llama la atención es que los tiempos promedio sobre la matriz $\hat{M}$ generan una curva más suave que los de la matriz M pero no pudimos encontrar que puede producir este comportamiento.

Intentamos realizar el mismo test para las imágenes de 92×112 píxeles pero aún bajando la cantidad de repeticiones y la cantidad de imágenes por persona se hacia demasiado lento (tomando unas cuantas horas solo para medir con hasta 4 personas y 5 imágenes); por ello cambiamos la óptica del experimento y decidimos medir los tiempos de cómputo y el consumo de memoria de nuestra implementación en un sólo test con 41×10 imágenes (toda la base) en tamaño completo y en versión reducida, para poder observar el

[9] `http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html`

máximo tiempo de ejecución y consumo de memoria. Al igual que la medición anterior, tanto en performance temporal y espacial de la matriz $\hat{M}$ debería ser mucho menor que la de M. Los resultados temporales fueron los siguientes:

variante	tamaño	tiempo promedio
normal	112×92	628.286
`--fast`	112×92	3.651
normal	28×23	1.413
`--fast`	28×23	565

Velocidades promedio para 15 corridas en milisegundos.

Como pensabamos, la performance temporal de $\hat{M}$ sobre M es mucho mejor.

Luego, evaluamos el consumo de memoria para las mismas configuraciones. Para esto usamos `valgrind` con `massif` y consideramos el reporte de `useful-heap`.

variante	tamaño	consumo máximo
normal	112×92	865.620
`--fast`	112×92	36.577
normal	28×23	5.558
`--fast`	28×23	3.610

Cantidad máxima de memoria usada en kilo bytes

Esto corresponde con nuestros cálculos preliminares, que siendo $M \in \mathbb{R}^{10304 \times 10304}$ una matriz del tipo `double` tendría que consumir por si sola al rededor de 829,472 kilo bytes.

En nuestra opinión es evidente que la variante de la matriz M no escala a imágenes de tamaño mayor a 100×100 píxeles, tanto por su lentitud como su consumo de memoria.

4.2. Análisis

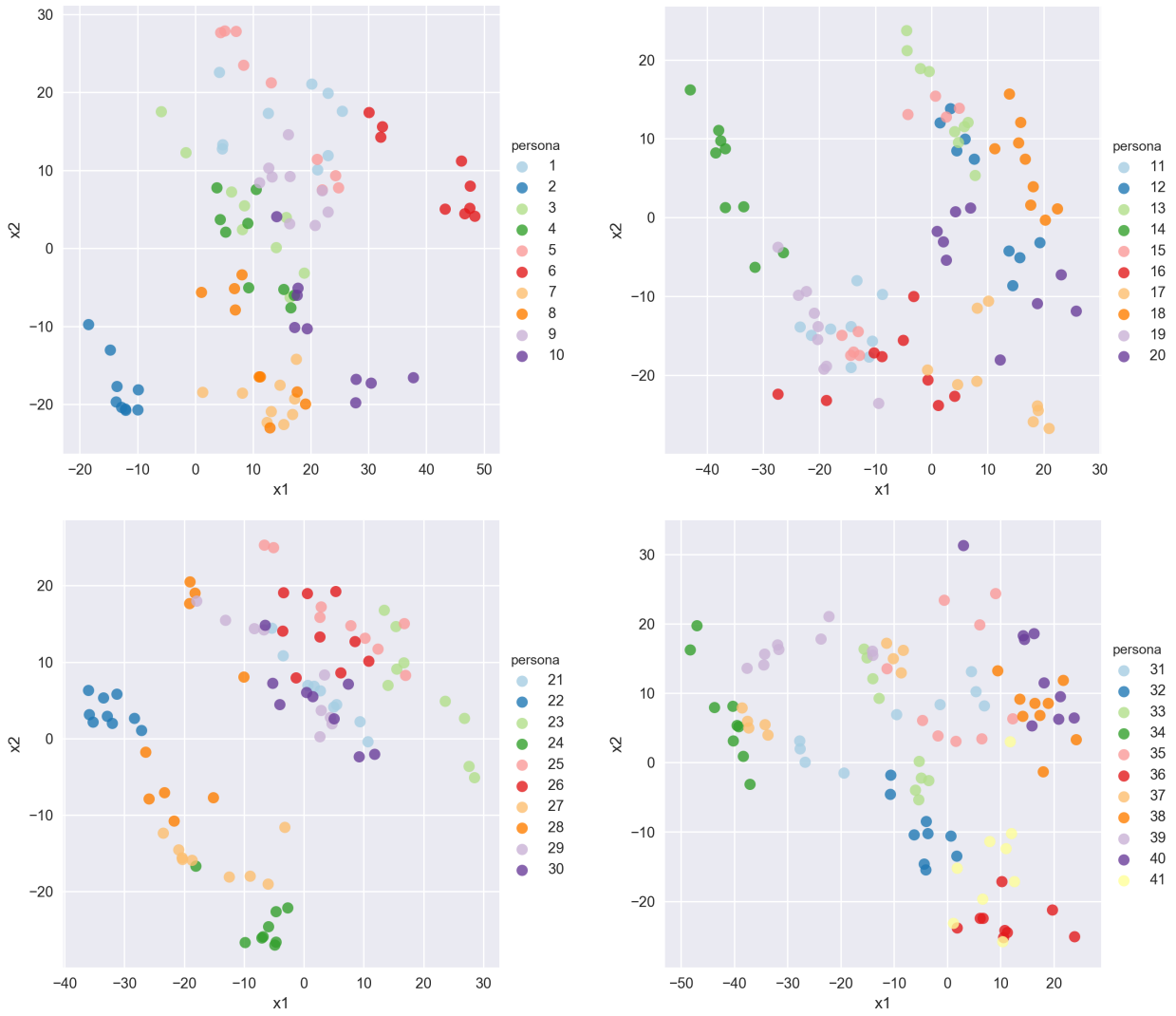
En esta sección trataremos de determinar los parámetros que hacen funcionar mejor al modelo. Estos son:

- M la cantidad de imágenes por persona
- Γ la cantidad de componentes principales
- k la cantidad de vecinos considerados por kNN

Lo primero que notamos es que no hay una diferencia visual perceptible entre visualizar los datos en su versión de 112×92 o en 28×23, por lo que usaremos la versión reducida para agilizar los experimentos en la mayoría de los casos.

Por lo tanto, el análisis siguiente se hizo sobre 41 personas con 9 imágenes cada una, dejando la restante para un análisis posterior sin *bias*.

Empecemos viendo nuestros datos graficando las primeras dos componentes principales de cada persona.

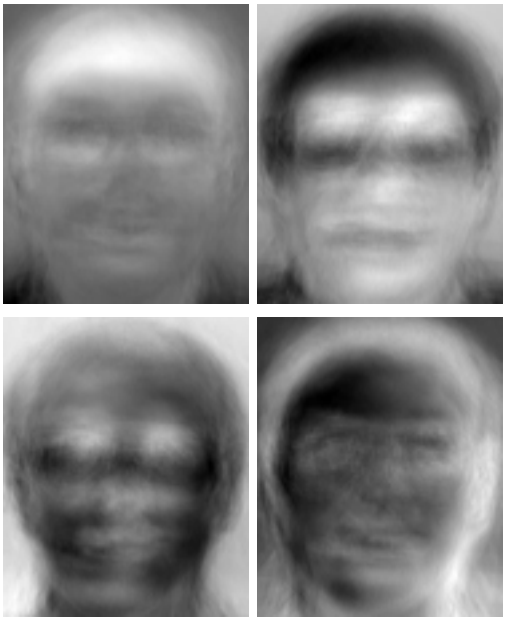


Primeras dos componentes principales.

Para mayor comodidad a la hora de graficar las 41 imagenes se dividieron en 4 gráficos con 10 imágenes cada una, se puede ver que, en lineas generales, las imagenes de una misma persona tienden a estar cerca unas de otras, de todas formas es fácil ver por qué necesitamos más componentes principales: Muchas de las caras conviven el mismo espacio en $\mathbb{R}^2$, con lo cual, si bien perdemos la habilidad de graficar los resultados, obtenemos más información teniendo más componentes.

Además, cada componente tiene asignado un autovector, y este vive en el *cara-space* por lo que podemos transformarlos en imágenes representables [10].

[10] Para esto usamos las imágenes con su resolución completa.

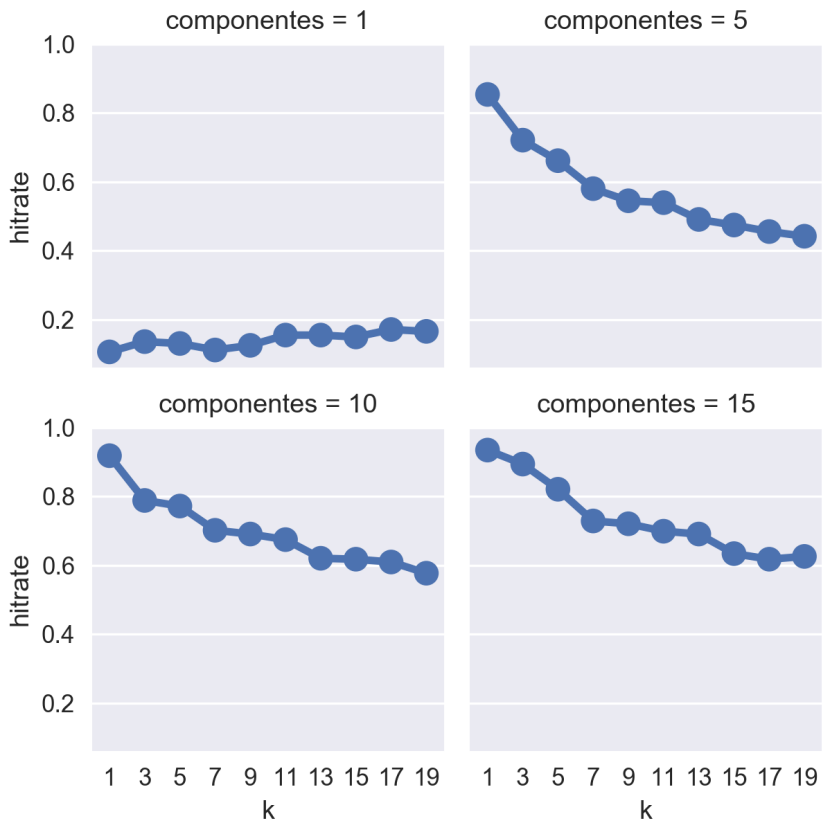


Primeros 4 autovectores (autocaras).

4.3. Cantidad de componentes y kNN

Para calcular el hitrate y el F1-score hay que tener cuidado de no caer en *overfitting*[11], por lo que procedemos a hacer *cross-validation* partiendo las imágenes de cada persona en 3 *folds* de 3 imágenes cada uno. En la siguiente figura vemos como el parámetro k afecta el *hitrate* o tasa de éxito para $\Gamma \in \{1, 5, 10, 15\}$.

[11] El sobreajuste (también es frecuente emplear el término en inglés *overfitting*) es el efecto de sobreentrenar un algoritmo de aprendizaje con unos ciertos datos para los que se conoce el resultado deseado. `https://en.wikipedia.org/wiki/Overfitting`

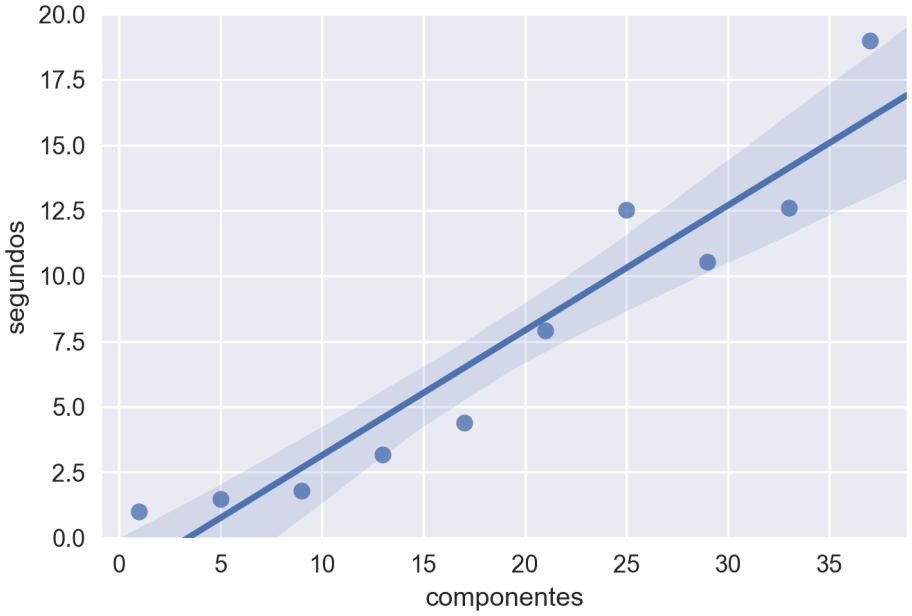


Hitrates para distintos Γ y k.

A simple vista vemos que, el *hitrate*

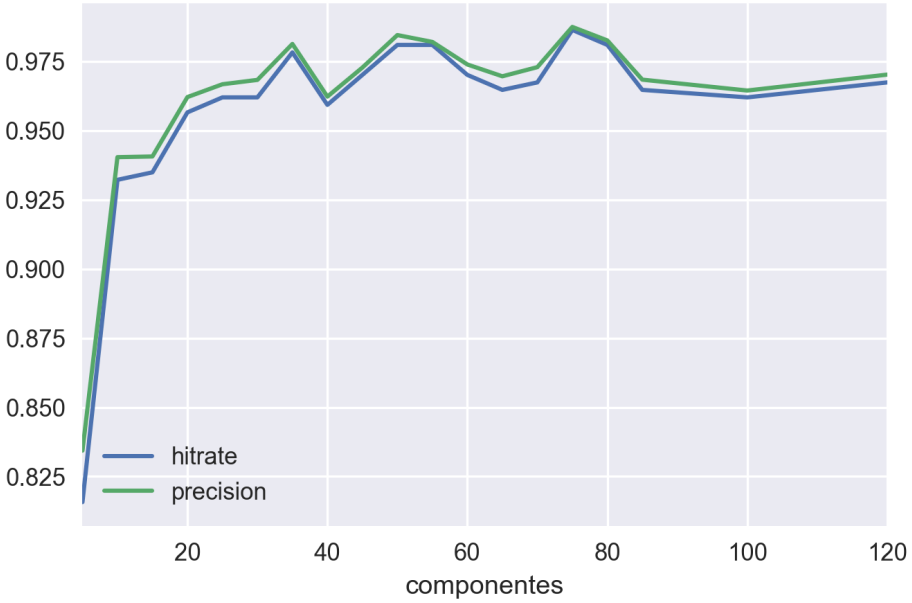
- disminuye en general al subir k
- aumenta al subir Γ

Pero subir Γ aumenta los tiempos de cómputo, como podemos apreciar en esta figura:



Relación entre Γ y los tiempos de cómputo.

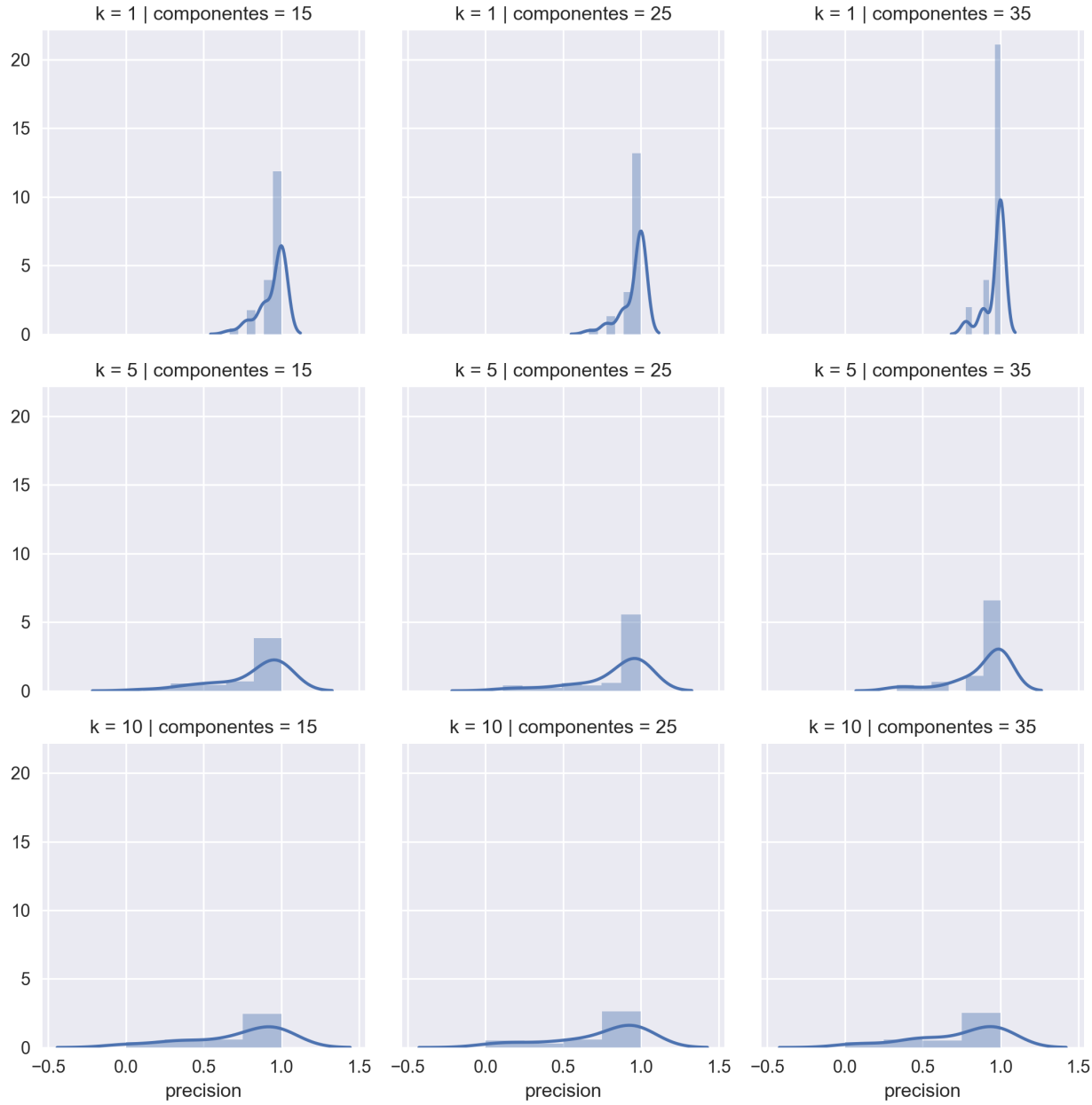
Hasta ahora vimos que nuestro modelo funciona mejor con $k = 1$ y que Γ tiene que ser mayor a 1. Veamos que pasa con el *hitrate* y la *precisión* al variar Γ. Estos los calculamos por persona y luego les calculamos el promedio.



Efectividad de Γ.

Por ahora vemos que el modelo tiene su pico de efectividad al rededor $\Gamma = 35$.

Pero puede ser que mientras Γ aumenta, k de alguna manera afecte los falsos positivos. Para esto volvemos a graficar Γ contra k, pero esta vez con el histograma de la precisión para cada persona.



Histograma de la precisión por clase respecto de Γ y k.

De nuevo observamos que $k = 1$ y $\Gamma = 35$ nos dan los mejores resultados.

4.4. Imágenes por persona

Hasta ahora usamos 3 *folds* de tamaño 3, por lo que usábamos 6 imágenes por persona. Es interesante ver que pasa con otras cantidades de imágenes, por lo que comparamos el *hitrate* por persona para 2, 4, 6 y 8 imágenes para cada una y obtuvimos la siguiente visualización.

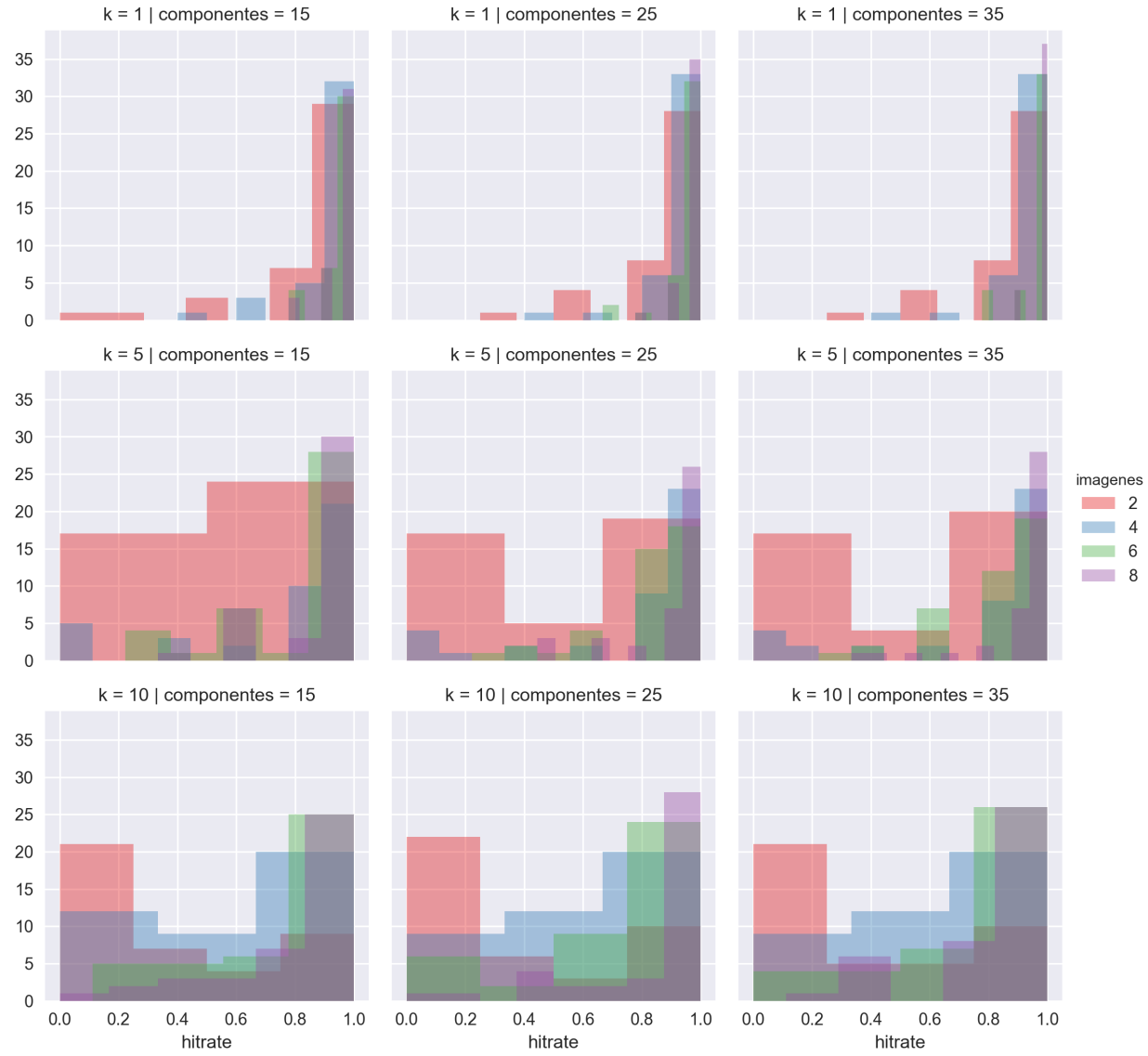
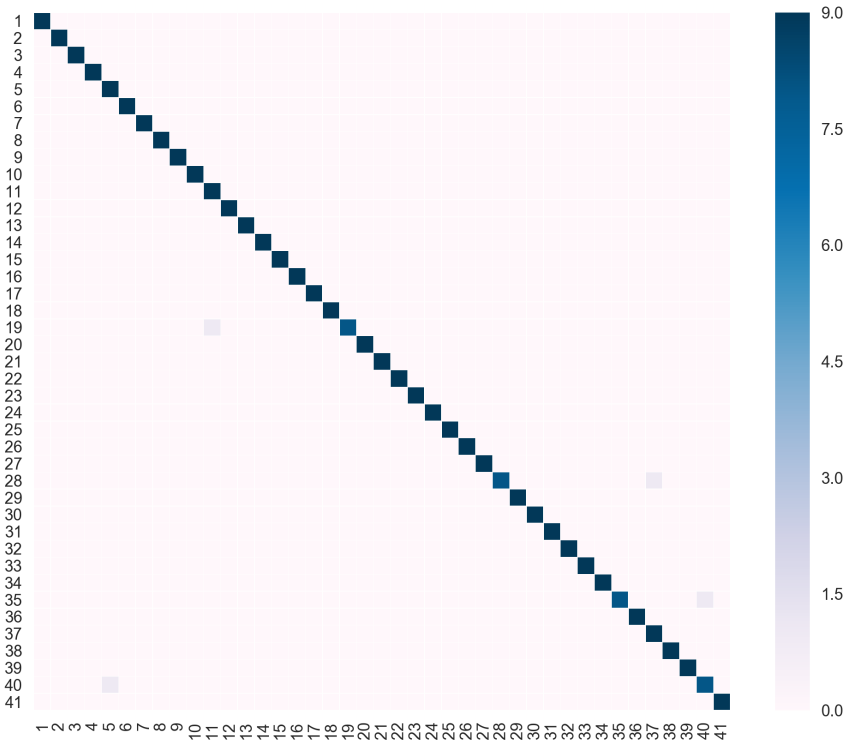


Figura 6: Histograma del *hitrate* respecto de Γ y k para distinta cantidad de imágenes por persona.

Claramente la cantidad de imágenes considerada afecta positivamente el *hitrate* del modelo, y podemos confirmar que:

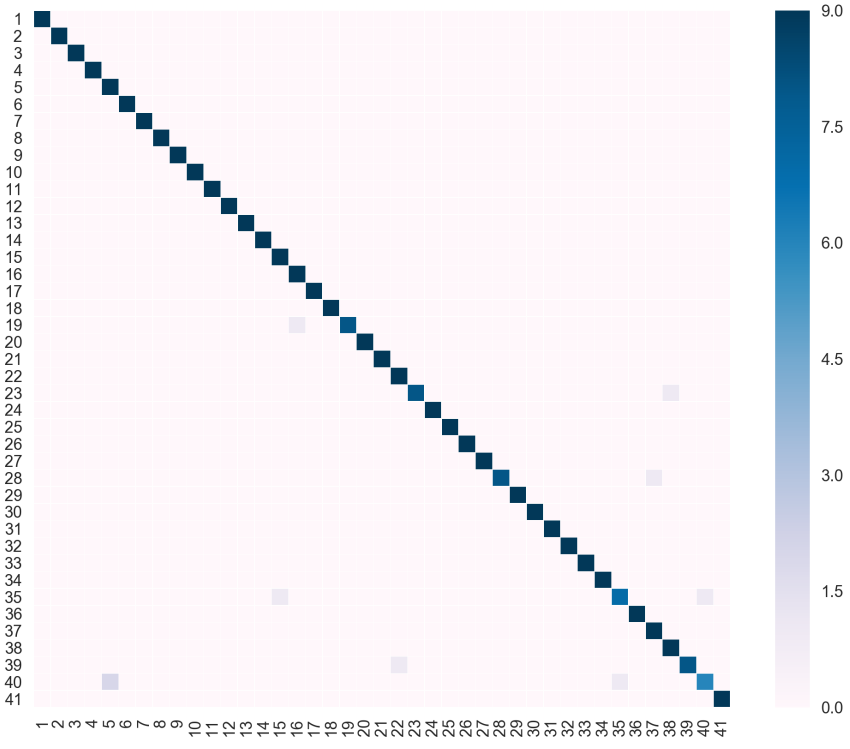
- Abajo a la izquierda en rojo tenemos la peor configuración del grupo, con 2 imágenes por persona $k = 10$ y $\Gamma = 15$. El *hitrate* apenas supera el 20 %.
- En la franja central vemos una configuración aceptable si usamos 8 imágenes, pero con menos bajaríamos del 90 % casi seguro.
- La esquina superior derecha presenta la mejor configuración posible $k = 1$ y $\Gamma = 35$. Usar 8 imágenes significa un *hitrate* que no baja de 90 %.

Para concluir presentamos la matriz de confusión con estos parámetros para 8 imágenes por persona y 9 *folds*, y para 6 imágenes y 3 *folds*.



Matriz de confusión para 9 *folds*.

Es muy fácil ver que los resultados son muy favorables, salvo escasas excepciones el clasificador acierta en su predicción, resultando en una diagonal muy marcada.



Matriz de confusión para 3 *folds*.

En cambio, vemos aquí como perdemos precisión cuando se reduce a un 3 *folds*, si bien la diagonal sigue

siendo muy marcada vemos que hay otros casos donde no clasifica correctamente.

4.5. Validación final

Con nuestros parámetros óptimos fijados, vamos a ver que pasa con las 41 imágenes que dejamos fuera de la experimentación, la 10^{ma} de cada persona. Con $k = 1$ y $\Gamma = 35$ nos queda:

Hitrate	92 %
Precison	92 %
F1-Score	92 %

El clasificador funcionó sin problemas con la excepción de estos casos:

persona actual		persona clasificada
1	$\rightarrow$	25
5	$\rightarrow$	40
10	$\rightarrow$	38

5. Esas raras caras nuevas: ¿Puedo detectar si una imagen es una cara?

En las secciones anteriores hemos visto como es posible detectar la pertenencia de una imagen de la base a una clase específica (es decir, una persona en particular) de una manera acertada, entonces *¿es posible detectar si una imagen tiene una cara o no?* dicho de otra manera, puedo detectar si una imagen cualquiera pertenece a mi espacio de imágenes (mi base de datos de prueba) o más aún, si una imagen es una cara que no estaba en la base, ¿puedo decir con certeza que es una imagen de una cara?

Una vez que tenemos una imagen y la procesamos a través de PCA y obtenemos su transformación característica nos da un vector de puntos en $\mathbb{R}^n$ sabemos que kNN funciona bajo el concepto de que los vectores que pertenecen a las imágenes de una misma persona son *cercanas* unas a otras. El concepto de cercanía se difumina cuando aumentamos la cantidad de puntos en nuestro vector, y es por esto que sucede la llamada "Maldición de la dimensionalidad".

Teniendo en cuenta esto debemos pensar, ¿hay alguna forma de discernir si un elemento pertenece al **Cara-Space** (es decir, el espacio de las imagenes que son caras)?

Inicialmente se planteó lo siguiente: dado una nube de puntos en $\mathbb{R}^n$ que son las caras de la base, calculo el *centroide* (es decir, el centro de masa, o promedio de los vectores) y formo una *n-esfera* (esfera en n dimensiones) que delimite mi *Cara-Space*, siendo el radio de la esfera la distancia máxima de los puntos al centroide.

Esto inicialmente parece bastante sensato, la idea de fondo es que por más que las caras sean de personas distintas son imágenes realmente muy parecidas por lo que deberían estar más cerca entre sí que, digamos, una imagen de una pokebola (una de las imágenes que se probaron), la cual no es una cara.

Ahora, consideremos el ejemplo de la figura 7, un ejemplo sintético en el cual tenemos tres clusters en dos dimensiones. Inicialmente el punto X vemos que no se encuentra en ningún cluster, sin embargo nuestro centroide, que es el promedio de todos los puntos, se ubicaría muy cercano a este punto. Esto haría que se reconozca a X como dentro del *Cara-Space* cuando se puede que estaba muy lejos de todos los clusters.

Figura 7: Ejemplo de *clusters* para 2 dimensiones

Es importante notar que si ya tenemos un error un tanto simple en dos dimensiones, al aumentar nuestro n estos errores se harían cada vez más y más comunes, pero se hacen *un poco* difíciles de graficar.

De este modo, podemos pensar que si bien la idea no está completamente bien, estamos bien encaminados. Calculando el centroide *para cada persona* podemos tener un *Cara-Space* un tanto más refinado que no incluye groseramente a todo el espacio dentro de la n-esfera.

Para probar esto lo que hicimos fue generar todos los centroides, obtener las mayores distancias y ver con las imágenes de prueba de la figura 8 cuáles eran los resultados.

Para poder diferenciar de una mejor manera los clusters de datos decidimos aumentar nuestra cantidad de componentes principales, con el trade-off de perder la habilidad de graficarlo, ya que ganaríamos poco, como se vio en secciones anteriores, por el hecho de que los puntos de las distintas personas están muy juntos en el espacio de dos componentes.

De esta forma decidimos realizar las pruebas con 15 componentes principales, para asegurarnos una buena precisión sin tanto tiempo de cómputo.

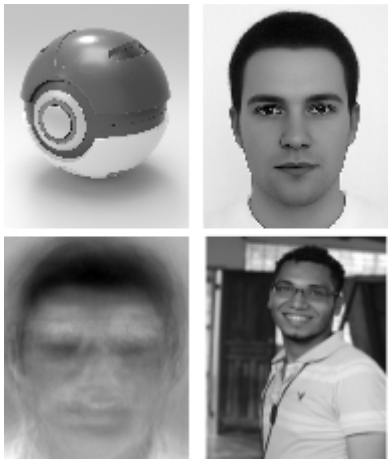


Figura 8: Pokebola, Jack, Autocara y John. Imagenes tomadas como ejemplo para la experimentación

Inicialmente obtuvimos un éxito, con estos criterios tenemos que la pokebola no pertenece al *Cara-Space* pero las caras de la base sí (aunque esto resulta un tanto obvio). De todas formas no es del todo correcta nuestra idea, ya que Jack (nuestra imagen muy similar a las de prueba pero que no está en la base) no la incluye en el *Cara-Space*. Queremos ver si hay alguna forma de incluir a una y desechar a la otra.

Experimentalmente llegamos a la conclusión de que multiplicando al radio de los centroides por un escalar $\alpha = 2$ obteníamos buenos resultados. Esto se puede explicar ya que, al ser una base finita que no contiene a todas las caras del *Cara-Space* estamos tomando algo mucho menor a nuestro espacio. Agrandando el espacio pero manteniendo la idea podemos acaparar más caras dentro del espacio sin tomar cosas de más.

Los resultados fueron sorprendentes, con $k = 2$ no solamente la Pokebola queda afuera del *Cara-Space* sino que Jack y la Autocara (una autocara generada por la implementación) quedan adentro del mismo. Aún más interesante, y esto fue un resultado inesperado, la imagen John (una imagen que no es de la misma forma que las de la base pero que contiene una cara) ¡Queda adentro del *Cara-Space*! Esto resulta muy interesante ya que nos podría ayudar al reconocimiento de rostros en imágenes.

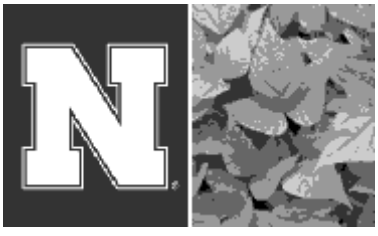


Figura 9: Logo de la Universidad de Nebraska-Lincoln y Matorral de hojas: ejemplo de imágenes sin caras que las encuentra en el *Cara-Space*

Sin embargo, no todo es color de rosas en nuestra implementación, en la figura 9 vemos dos ejemplos de

imágenes encontradas en el *Cara-Space* de manera errónea, o por lo menos nosotros no consideramos que eso sean caras. Si bien obtuvimos unos resultados muy razonables, la implementación es un tanto *naïf*, con lo cual hay mucho espacio a la mejora de este algoritmo, como bien se puede ver con las imágenes de prueba.

Figura 10: Grumpy Cat, ofendido porque no entró en el *Cara-Space*

Esto puede deberse al problema de la *maldición de la dimensionalidad*, ya que estamos tratando con un espacio de $\mathbb{R}^n$ con $n = 15$, esto se decidió ya que era un compromiso entre cantidad de componentes óptima y no perder la cercanía de los datos, ya que aumentando el n se obtendrían espacios demasiado grandes donde todos los puntos están muy alejados unos de otros. De todas formas, reducir las dimensiones no fue suficiente para encontrar un criterio suficiente acertado, como decía un cierto ilusionista "*puede fallar*".

6. Conclusión

A lo largo de este trabajo logramos resolver la dificultad de computar el enorme volumen de datos. Conseguimos poder reducir la dimensionalidad de nuestra base de datos y ésta no afecto al objetivo final que era hallar una buena clasificación de los rostros de las personas. Pudimos observar como una simple optimización en la matriz de covarianza hacia que los tiempos de computo se redujeran ampliamente. Buscamos una combinación de vecinos cercanos y componentes principales tal que el clasificador tenga la menor cantidad de errores posibles, en nuestro caso lo pudimos observar cuando la cantidad de vecinos considerados por kNN es de 1 y la cantidad de componentes principales es de 35 demostrándonos que el método es eficiente y certero. Luego, diseñamos, implementamos y experimentamos sobre una aproximación para detectar si una imagen contiene una cara o no, observando que es un problema muy difícil.

En definitiva, ajustamos el algoritmo experimentalmente para obtener los mejores resultados, de esta manera vemos que es necesario un análisis preliminar a la hora de realizar algoritmos de esta índole, ya que sin estas pruebas el algoritmo es como *un auto con las tuercas flojas*.

Con esto, podemos concluir que el método de PCA es rápido, simple y funciona bastante bien dentro de un entorno controlado como lo era la base de datos provista por la cátedra.