



DEPARTAMENTO  
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico Número 2

CSI:DC

Métodos Numéricos

Integrante	LU	Correo electrónico
Hofmann, Federico	745/14	federico2102@gmail.com
Terén Bernal, Guido Sebastian	749/14	guidoterén@gmail.com
Pyrih, Franco	520/12	fpyrih@dc.uba.ar



**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

<b>I</b>	<b>Introducción teórica</b>	<b>4</b>
<b>II</b>	<b>Desarrollo</b>	<b>5</b>
1.	Estructura de datos	5
2.	Métodos numéricos	5
2.1.	k vecinos más cercanos (kNN)	5
2.2.	Método de la potencia (power iteration)	5
2.2.1.	Deflación	6
2.3.	Análisis de componentes principales (PCA)	6
2.4.	Análisis discriminante de cuadrados mínimos parciales (PLS-DA)	7
2.5.	Validación cruzada k-plegada (k-fold x-validation)	7
3.	Implementación	8
4.	Experimentos, resultados y discusión	8
4.1.	Metodología general	8
4.1.1.	Verificación de los resultados	8
4.1.2.	Métricas de evaluación de clasificadores	9
4.1.3.	Mediciones de tiempo	9
4.1.4.	Archivo de salida de reporte	9
4.1.5.	Recopilación de datos y generación de gráficos	9
4.1.6.	Automatización de las pruebas	9
4.2.	Experimento 1: PCA + kNN vs PLS-DA + kNN variando $k$ , $\alpha$ y $\gamma$	9
4.2.1.	Hipótesis	10
4.2.2.	Resultados y discusión	10
4.3.	Experimento 2: PCA + kNN y PLS-DA + kNN variando la cantidad de imágenes de entrenamiento	16
4.3.1.	Hipótesis	16
4.3.2.	Resultados y discusión	16
4.4.	Experimento 3: variando $K$	18
4.4.1.	Hipótesis	18
4.4.2.	Resultados y discusión	18
4.5.	Observaciones	30
4.5.1.	Tiempo de cómputo de kNN cambiando tipo de datos	30
4.5.2.	Tiempo de cómputo doble ciclo anidado	30
5.	Dificultades encontradas y soluciones implementadas	30
5.1.	Violación de segmento con <code>argv[0]</code> grande	30
5.2.	Automatización de las pruebas	31
5.2.1.	Espacio en disco insuficiente	31
5.2.2.	Usuarios apagan computadoras durante pruebas	31
5.2.3.	Computadora rechaza usuario y contraseña	31
<b>III</b>	<b>Conclusiones</b>	<b>32</b>
6.	Tiempos de computo y efectividad de las métricas	32
7.	Experimentos pendientes	32
<b>IV</b>	<b>Apéndices</b>	<b>33</b>

<b>8. Apéndice A: código fuente relevante</b>	<b>33</b>
8.1. k vecinos más cercanos (kNN) . . . . .	33
8.2. Método de la potencia (power iteration) . . . . .	34
8.2.1. Deflación . . . . .	35
8.3. Análisis de componentes principales (PCA) . . . . .	36
8.4. Análisis discriminante de cuadrados mínimos parciales (PLS-DA) . . . . .	36
8.5. Validación cruzada k-plegada (k-fold x-validation) . . . . .	37
<b>9. Apéndice B</b>	<b>38</b>
9.1. Probabilidad de obtener 10 dígitos de cada clase tomando 100 al azar del conjunto de entrenamiento . . . . .	38
<b>V Referencias</b>	<b>39</b>
9.2. The C++ Programming Language . . . . .	39
9.3. CPlusPlus.com - ctime reference . . . . .	39
9.4. Documentación de LibreOffice Calc . . . . .	39
9.5. Mathworks.com cvpartition help . . . . .	39
9.6. gentoo.org wiki GCC Optimization -O . . . . .	39
9.7. llvm.org clang docs cmd option -ffast-math . . . . .	39
9.8. Numerical Analysis . . . . .	39
9.9. Apuntes de Métodos Numéricos en CubaWiki.com.ar . . . . .	39
9.10. Apunte de SSH de Marco Vanotti . . . . .	39
9.11. Ayuda de Gitlab en la instancia del DC para configurar acceso con par de cripto-llaves	39

## Parte I

# Introducción teórica

En este trabajo, tenemos una serie de imágenes de dígitos manuscritos (del 0 al 9) de los cuales, de una parte sabemos qué dígito representan y de otra, no. Las imágenes tienen 28 píxeles de alto por 28 de ancho, están en escala de grises y tienen fondo negro.

La idea es entrenar a la computadora usando los dígitos cuya clase conocemos (conjunto de entrenamiento), para luego poder reconocer los dígitos cuya clase no conocemos (conjunto de prueba). Para esto, vamos a considerar a cada imagen como un vector en el hiperespacio  $R^{784}$  ( $784 = 28 \times 28$ , cantidad de píxeles de las imágenes en cuestión), en el que cada elemento del mismo está entre 0 (negro) y 255 (blanco).

Luego, cuando recibamos una imagen de un dígito manuscrito y queramos saber de qué dígito se trata, vamos a calcular la distancia (vectorial, hiperespacial) entre éste y cada uno de los dígitos cuya clase conocemos, elegir a los  $k$  que estén más cerca del recibido y clasificarlo de la misma manera que esté clasificada la clase de dígito más numerosa dentro de esos  $k$ . Este método de clasificación es el de los *k-vecinos más cercanos* (o *kNN*, por las siglas en inglés de *k-nearest neighbors*).

Para salvarnos de pagar la totalidad del costo de comparar a cada dígito manuscrito recibido, con todos nuestros dígitos de entrenamiento, también vamos a implementar dos algoritmos de *reducción*: análisis de componentes principales (o PCA, por las siglas en inglés de *principal component analysis*) y análisis discriminante de cuadrados mínimos parciales (o PLS-DA, por las siglas en inglés de *partial least squares discriminant analysis*).

Estos identifican la información más relevante de nuestros datos usando propiedades de autovalores y autovectores, y nos van a permitir descartar información más redundante, para poder procesar menos datos (ganando en velocidad de cómputo) a un menor costo en efectividad de identificación. Para poder aplicarlos, vamos a organizar a nuestras imágenes de entrenamiento en una matriz que tendrá una fila por imagen y una columna por píxel (784 columnas).

PCA consiste en obtener el cambio de base de nuestra matriz de imágenes de entrenamiento que maximiza la varianza entre las 784 variables y hace cero a la covarianza entre las mismas.

De esta manera, podemos quedarnos con las dimensiones de mayor varianza de nuestro conjunto de imágenes de entrenamiento y descartar las demás (que son más redundantes) para acelerar el proceso de clasificación. PLS-DA es similar, pero considera también a qué clase pertenece cada imagen de entrenamiento y encuentra las relaciones fundamentales entre las mismas y su clasificación.

La idea es nuevamente encontrar una transformación de los datos, pero en este caso vamos a querer maximizar la covarianza entre las imágenes de entrenamiento y su clasificación (qué dígito representan). Nuevamente, nos quedaremos con la porción de los datos que capture la mayor varianza entre variables, para así reducir nuestro conjunto de datos a los más importantes y poder, en consecuencia, computar más rápido.

## Parte II

# Desarrollo

### 1. Estructura de datos

A la hora de organizar los datos que utilizamos a lo largo del trabajo, decidimos armar una matriz en la cual cada fila es una imagen diferente de la base de datos y la primera columna contiene el dígito (del 0 al 9) que representa cada imagen. La base de datos utilizada es la del sitio kaggle.com, la cual consta de 42000 imágenes de 28x28 píxeles cada una, por esto es que los vectores son de 754 coordenadas cada uno.

También creamos una segunda matriz con la misma estructura que la anterior para las imágenes a ser testeadas. En el caso de conocer de antemano qué dígito representa cada imagen, almacenamos al mismo en la primera columna de la matriz del mismo modo que lo hicimos con la matriz de imágenes de entrenamiento. Esto nos fue realmente útil al momento de corroborar si nuestro programa acertó a que dígito correspondía la imagen ya testada.

### 2. Métodos numéricos

En esta sección explicaremos los distintos métodos que utilizamos para encontrar a que dígito corresponde cada imagen nueva a ser testada. Incluimos aquí métodos de comparación de imágenes (kNN), métodos de pre-procesamiento (PCA y PLS-DA) y otros dos necesarios para el cálculo de autovalores y autovectores (método de la potencia y deflación).

#### 2.1. k vecinos más cercanos (kNN)

Este método consiste en comparar la imagen a testear con cada una de las imágenes de la base de entrenamiento. Lo que se busca mediante esta comparación es hallar las k imágenes más parecidas a la que se desea reconocer, y a partir de esas k imágenes obtenidas seleccionar un dígito para devolver como respuesta.

Para elegir las k imágenes mas parecidas nuestro procedimiento consto en hallar la distancia vectorial (recordemos que a cada imagen la tenemos representada como un vector) entre la imagen a testear y todas las imágenes de la base de entrenamiento. Esta distancia vectorial se calculó aplicando norma dos (suma de los cuadrados de todas las coordenadas del vector y raíz cuadrada del numero resultante) al resultado de la resta coordenada a coordenada de cada vector de entrenamiento con el de test. Estas distancias se fueron agregando una por una en un vector de tuplas de dos coordenadas cada una, donde la primera contenía la etiqueta (numero del 0 al 9) de la imagen en la base de entrenamiento con la cual se calculo la distancia, y la segunda coordenada era dicha distancia. Una vez obtenido este vector de distancias, con sus respectivas etiquetas, se procedió a ordenarlo de menor a mayor según las distancias obteniendo así las k menores distancias en las K primeras coordenadas del vector.

Ya con las k imágenes mas "parecidas", lo siguiente fue seleccionar aquella cuya etiqueta se repetía mas veces. Para ello se observo la primera coordenada de cada una de las k tuplas y se elegía como respuesta aquella que aparecía mas veces. Por ejemplo, si K fuese igual a 5 y de esas cinco imágenes tres fuesen ceros y las dos restantes ochos, la respuesta seria tres. Por otro lado, en el caso de que halla dos o mas imágenes con distintas etiquetas e igual cantidad de repeticiones, la respuesta seria elegida al azar entre cualquiera de ellas.

#### 2.2. Método de la potencia (power iteration)

Este es un método iterativo que se utiliza para obtener el mayor autovalor en modulo de una matriz. Para asegurar su convergencia en una cantidad finita de pasos es condición necesaria que la matriz a ser analizada sea cuadrada y posea un autovalor estrictamente mayor (en modulo) al resto.

En nuestra implementación, se calcula a partir de un vector inicial aleatorio y una cantidad de iteraciones prefijada. Su algoritmo es el siguiente:

```
MétodoDeLaPotencia(matriz_A, vector_inicial, cant_iteraciones)
  avector_ppal <- vector_inicial
  para i = 1 hasta cant_iteraciones:
    avector_ppal <- (matriz_A * avector_ppal)/norma2(matriz_A * avector_ppal)
  fin_para
  avalor_ppal <- (avector_ppal^t * matriz_A * avector_ppal)/(avector_ppal^t * avector_ppal)
  devolver avalor_ppal, avector_ppal
fin
```

Otras implementaciones utilizan una aproximación del autovector dominante como vector inicial. Otra variación posible es no usar una cantidad de iteraciones prefijada, sino que la misma esté determinada por la diferencia entre el último candidato a autovector principal calculado y el que fue generado antes que este (que el algoritmo finalice si la norma de dicha diferencia es menor a un cierto parámetro).

### 2.2.1. Deflación

Como se mencionó anteriormente, el método de la potencia solo nos brinda el mayor autovalor de una matriz “A” y su autovector asociado. Si queremos obtener más autovalores y autovectores debemos implementar el método de deflación.

Este método consiste en cambiar la matriz A “quitándole” el autovector previamente calculado para obtener así una nueva matriz A’ que posee el resto de los autovalores. De este modo se podrá aplicar nuevamente el método de la potencia y luego deflación y así  $\alpha$  veces, donde  $\alpha$  es la cantidad de autovalores y autovectores que se desea calcular.

Para realizar deflación se debe restar a la matriz A la multiplicación del autovalor previamente obtenido por su respectivo autovector por dicho autovector traspuesto. Las condiciones necesarias para realizar este método son las mismas que las del método de la potencia.

## 2.3. Análisis de componentes principales (PCA)

PCA no se trata de un método de clasificación en sí, sino que se utiliza para realizarle un pre-procesamiento a todas las imágenes de la base de entrenamiento buscando de este modo reducir las dimensiones de las mismas obteniendo mejoras en tiempos de cómputo.

Lo que se busca mediante este método es reducir las dimensiones de las imágenes quedándonos solamente con la información mas importante y relevante de cada una, quitándoles de este modo datos considerados como “ruido” ya que no aportan nueva información. Lo que buscamos entonces es minimizar la covarianza y maximizar la varianza entre todas las imágenes de la base de entrenamiento. Para lograr esto debemos realizarle un cambio de base adecuado a todas las imágenes.

Sea  $x^{(1)}, \dots, x^{(n)}$  una secuencia de  $n = 42.000$  imágenes de entrenamiento, con  $x^{(i)} \in \mathbb{R}^{784}, \forall i$ .

Y sea  $\mu$  su media, tal que  $\mu = \frac{1}{n}(x^{(1)t} + \dots + x^{(n)t})$ .

Definimos a la matriz X como la que contiene a todas las imágenes centradas respecto de la media:

$$X = \begin{pmatrix} x^{(1)t} - \mu \\ \vdots \\ x^{(n)t} - \mu \end{pmatrix}.$$

Lo siguiente consiste en multiplicar a la matriz X traspuesta por X y dividir a cada coordenada resultante por  $n - 1$ . La matriz obtenida será la matriz de covarianzas. Si recordamos cómo están definidas la varianza y la covarianza, notaremos que, al multiplicar a X por su traspuesta, obtendremos una matriz que tendrá todas las varianzas en su diagonal y las covarianzas en cualquier otro lugar.

Una vez obtenida la matriz de covarianzas "M", procedemos a calcular autovalores y autovectores de la misma mediante el método de la potencia y deflación mencionados en la sección anterior. Sabemos que M es simétrica semi-definida positiva y asumimos que todos sus autovalores son diferentes. Pero no necesitaremos calcularlos todos, sino que solo calcularemos los primeros  $\alpha$  autovalores y sus respectivos autovectores, donde  $\alpha$  es un parámetro de la implementación y definirá con cuántas dimensiones nos quedaremos.

Finalmente, se utiliza la matriz de autovectores traspuesta para realizar el cambio de base tanto a las imágenes de entrenamiento como a las imágenes a ser testeadas. Este último procedimiento se denomina transformación característica y consiste en centrar a cada imagen respecto de sus medias, dividir las por la raíz cuadrada de  $n - 1$  y multiplicar a la matriz de autovectores traspuestos por cada una de ellas.

Ya teniendo todos los datos pre-procesados se procede a realizarles algún otro método para decidir a qué categoría pertenece la imagen a testear. En nuestro caso, este último paso lo hacemos mediante kNN.

## 2.4. Análisis discriminante de cuadrados mínimos parciales (PLS-DA)

A diferencia de PCA, con PLS-DA se busca utilizar la información de las clases o etiquetas (dígitos del 0 al 9 que representa cada imagen). Para ello se intenta maximizar la covarianza entre cada imagen y su respectiva clase.

Para realizar lo planteado procedemos a generar dos matrices. Una de ellas es la matriz X de PCA, la cual es la matriz de las imágenes de entrenamiento centrada respecto de la media y dividida por la raíz cuadrada de  $n - 1$ , con el mismo n definido en PCA (cantidad de imágenes de entrenamiento). La otra matriz será la matriz Y, que contiene información sobre las etiquetas de cada imagen de la siguiente manera. Sus dimensiones serán n filas por 10 columnas y en cada fila tendrá un 1 en la columna cuyo número coincida con el número de la etiqueta de la imagen en dicha fila y un 0 en el resto de las posiciones. Por ejemplo, si la imagen en la fila 3 es un 5, entonces la matriz Y tendrá un 1 en la fila 3, columna 6.

Luego, el vector que maximice la covarianza entre las imágenes de la matriz X y la matriz Y será el autovector asociado al mayor autovalor de la matriz  $X^t Y Y^t X$ . Para calcularlo usamos el método de la potencia con  $\alpha = 1$ . Recordemos que el parámetro  $\alpha$  indicaba cuantos autovalores se desea calcular.

Finalmente se calcula el vector  $T_i$  como la matriz X multiplicada por el autovalor hallado y este vector se utiliza para actualizar las matrices X e Y de la siguiente manera. X será igual a  $X - T_i T_i^t X$  e Y será igual a  $Y - T_i T_i^t Y$ . A partir de estas dos nuevas matrices se puede volver a calcular  $X^t Y Y^t X$  y así calcular un nuevo autovalor y autovector. Este procedimiento se repetirá  $\gamma$  veces, donde  $\gamma$  es un parámetro de la implementación.

Al igual que en PCA, todo lo explicado previamente es un método de pre-procesamiento de los datos. Una vez realizados todos estos pasos se procede a realizar una transformación característica tanto a las imágenes de entrenamiento como a las de test, del mismo modo que se explico en PCA pero esta vez con los nuevos autovectores y por ultimo se aplica kNN a las imágenes en la nueva base.

## 2.5. Validación cruzada k-plegada (k-fold x-validation)

Para tomar mediciones sobre la calidad de nuestro clasificador realizamos diferentes tests sobre las imágenes de entrenamiento. Es decir, tomamos una cantidad de las imágenes de entrenamiento como test y las demás las utilizamos para entrenar. Para que este método sea lo mas efectivo posible, utilizamos la técnica de validación cruzada k-plegada. Para realizar esta técnica dividimos las imágenes de entrenamiento en partes iguales (los K pliegues) y vamos a realizar K iteraciones obteniendo las mediciones. En cada iteración vamos a tomar un pliegue para test y los restantes para entrenamiento. Como en cada iteración el pliegue tomado para test va variando, esto nos da una medición mas real de la calidad de nuestro clasificador.

### 3. Implementación

Para agregar todas las imágenes de la base de datos a nuestro código lo que hicimos fue cargarlas en una matriz creada mediante vectores de la librería *vector*. A esta matriz la llamamos *ImagenesEntrenamiento*. El llenado de esta matriz lo hicimos mediante una función a la que llamamos *imagenes.a.vectores* a la cual le pasábamos un parámetro extra desde el main cuya función era detectar si se debía particionar la base de entrenamiento para utilizar algunas imágenes para entrenamiento y otras para test o si no se debía particionar y utilizar todas para entrenar. En el caso de que el parámetro indicara que la base se debía particionar, la función automáticamente incluiría algunas imágenes en la matriz *ImagenesEntrenamiento* y otras en *ImagenesTest* de acuerdo a lo indicado en el archivo de particionamiento.

Una vez creada esta función ya teníamos listas las matrices sobre las que trabajaríamos. Lo siguiente fue crear la función kNN. Lo primero que hace kNN es calcular las distancias entre las imágenes y almacenarlas en el vector “distancias”. Para ello cada imagen de la matriz *ImagenesTest* entra en un ciclo en el cual se resta coordenada a coordenada con una imagen de *ImagenesEntrenamiento*, esas coordenadas ya restadas se elevan al cuadrado y luego se suman. Finalmente se les calcula la raíz cuadrada y es allí cuando se almacena en el vector recién mencionado, junto con la etiqueta de la imagen con la cual se le calculo la distancia (el vector *distancias* es un vector de tuplas `vector<pair(int, double)>`). Esto se realiza con todas las imágenes de entrenamiento, con lo cual el vector *distancias* tendrá tantas coordenadas como filas de *ImagenesEntrenamiento*.

Con el vector de distancias completado se procede a ordenarlo de menor a mayor según las distancias mediante la función *ordenarPrimerasKDistancias* obteniendo de este modo los “k vecinos mas cercanos.”<sup>a</sup> la imagen testeada.

Finalmente se llama a la función *vecinoGanador*, la cual busca entre las k etiquetas, la que mas se repite y esa sera la respuesta. Este procedimiento se realiza para cada imagen de *ImagenesTest*.

El algoritmo de kNN recibe también los parámetros  $\alpha$  y  $\gamma$  de PCA y PLS-DA respectivamente y a partir de ellos, según el método utilizado, decide cuantas coordenadas de cada vector utilizar. Esto lo implementamos ya que cuando se le realiza el cambio de base a la matriz *ImagenesEntrenamiento* y a la matriz *ImagenesTest*, lo que hacemos es sobrescribir dichas matrices con los nuevos valores, pero sin cambiar sus dimensiones. Con lo cual los primeros  $\alpha$  o  $\gamma$  valores de estas matrices serán los nuevos y el resto serán los mismos de antes. Por esto mismo es que debemos “decirle” a kNN cuando parar.

Otras funciones útiles que implementamos fueron “centrar” la cual tomaba como parámetros una matriz de imágenes y un vector de medias y lo que hace es centrar a todas las imágenes respecto de sus medias y dividir las por la raíz cuadrada de  $n-1$ , donde  $n$  es la cantidad de filas de la matriz de imágenes pasada como parámetro. Esta función se utilizó varias veces en los métodos PCA y PLS-DA. También creamos la función *trasponer*, la cual recibe como parámetro una matriz cualquiera y devuelve su traspuesta.

## 4. Experimentos, resultados y discusión

### 4.1. Metodología general

#### 4.1.1. Verificación de los resultados

Para poder verificar si nuestro programa acertó en la clasificación del dígito manuscrito recibido, en vez de usar 42000 imágenes de entrenamiento y 10000 de prueba, vamos a partir nuestro conjunto de imágenes de entrenamiento (cuya clasificación conocemos para toda imagen) en entrenamiento y prueba.

Como las imágenes podrían no estar distribuidas uniformemente, si bien a primera vista parecen ordenadas al azar, vamos a aplicar la técnica de validación cruzada K-plegada.

La idea es que partimos nuestro conjunto de imágenes de entrenamiento en K pliegues de igual tamaño y luego un pliegue a la vez va a ir asumiendo el rol de conjunto de imágenes de prueba,



mientras que todos los demás se utilizarán como entrenamiento.

Esta técnica es estadísticamente más robusta y evita problemas tales como el *overfitting*.

#### 4.1.2. Métricas de evaluación de clasificadores

Para evaluar la precisión de nuestros resultados utilizaremos tres métricas distintas.

La primera es Hit Rate, la cual simplemente evalúa la cantidad de aciertos sobre la cantidad de imágenes testeadas. El resultado de dicha metrica sera un porcentaje.

La segunda metrica que utilizaremos sera Recall. Lo que se busca acá es identificar la cantidad de imágenes que fueron correctamente clasificadas. En esta medición se incluirá falsos negativos, es decir la cantidad de imágenes que se clasificaron como no pertenecientes a cierta clase, cuando en realidad si pertenecían.

Por ultimo utilizaremos Precision, la cual es una metrica que informa la cantidad de imágenes correctamente clasificadas, y aquellas que se identificaron como pertenecientes a cierta clase y en realidad no lo eran.

#### 4.1.3. Mediciones de tiempo

Para realizar las mediciones de tiempo utilizamos la libreria ctime. Para que nuestras mediciones sean lo mas precisas posibles iniciabamos el tiempo justo antes de realizar el metodo y lo parabamos apenas terminado el mismo para no tomar en cuenta tiempos que no sean exclusivamente del metodo.

#### 4.1.4. Archivo de salida de reporte

Luego de cada ejecución, el programa escribe un archivo agregando la terminación “.report” al archivo de salida especificado por parámetro.

En este se encuentran los resultados de las mediciones de tiempo y el porcentaje de aciertos.

#### 4.1.5. Recopilación de datos y generación de gráficos

En este trabajo práctico realizamos experimentos para los cuales tomamos gran cantidad de mediciones. Para recopilar los datos, los volcamos en planillas de cálculo. Para armar los gráficos usamos herramientas incluídas en el mismo programa en el que armamos las planillas.

#### 4.1.6. Automatización de las pruebas

Para poder realizar pruebas más rápido, usamos el servicio de acceso remoto a las computadoras de los laboratorios de computación y scripts de bash.

Lo primero que tuvimos que hacer fue configurar el acceso por par de llaves criptográficas tanto en el servidor de Gitlab del departamento (que usamos para el desarrollo), como en el de acceso remoto a los laboratorios.

Luego escribimos los comandos a ejecutar en archivos ejecutables *.sh*. La idea general fue conectarse via *ssh*, clonar el repositorio, compilar el programa, ejecutar las pruebas y subir los resultados.

El nivel de automatización alcanzado no bastó para que se elijan las computadoras a usar y los casos de prueba a correr de forma autónoma, por lo que esto lo hicimos a mano. Tenemos pensado automatizar también ese paso en la próxima oportunidad.

### 4.2. Experimento 1: PCA + kNN vs PLS-DA + kNN variando $k$ , $\alpha$ y $\gamma$

En este experimento vamos a analizar la calidad de los resultados obtenidos combinando PCA y PLS-DA con kNN.

La idea es probar una variedad de combinaciones de los distintos parámetros que toman nuestros métodos, para luego evaluar la efectividad de nuestros resultados y los tiempos de ejecución obtenidos.

Nuestras variables van a ser: la cantidad  $k$  de vecinos más cercanos a considerar en kNN; y la cantidad  $\alpha$  o  $\gamma$  de dimensiones a considerar en PCA o PLS-DA, respectivamente.

#### 4.2.1. Hipótesis

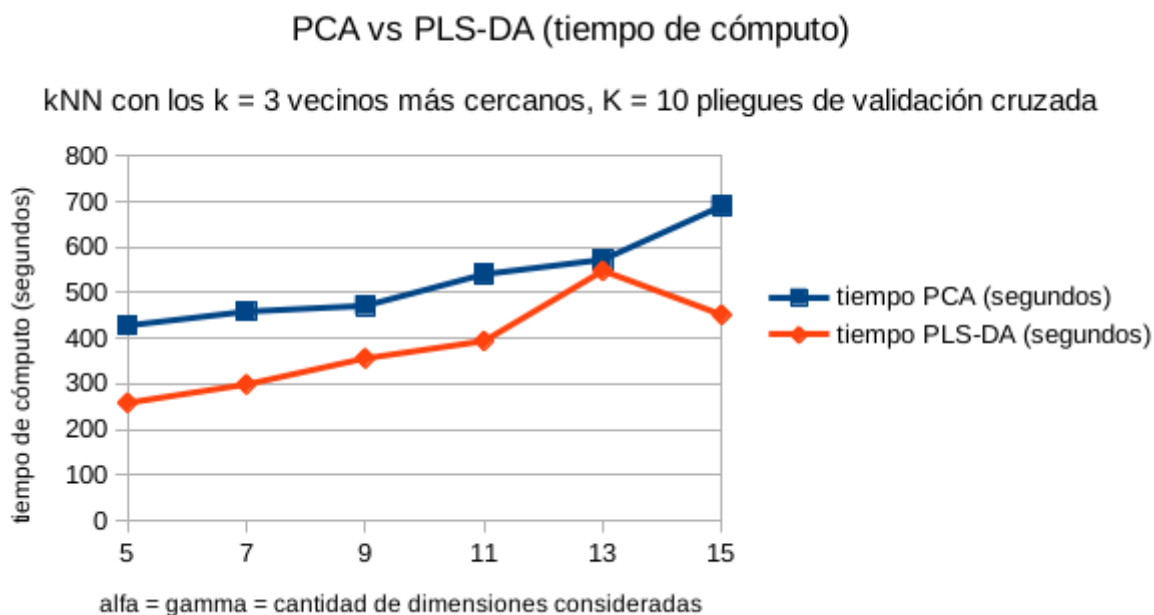
Esperamos que considerar una mayor cantidad de dimensiones al momento de identificar un dígito manuscrito, mejore nuestro porcentaje de aciertos, y a partir de cierto parámetro deje de hacerlo (cuando se empiecen a considerar dimensiones con muy poca varianza).

Por otro lado, considerar una mayor cantidad de dimensiones creemos que nos va a costar más tiempo de cómputo.

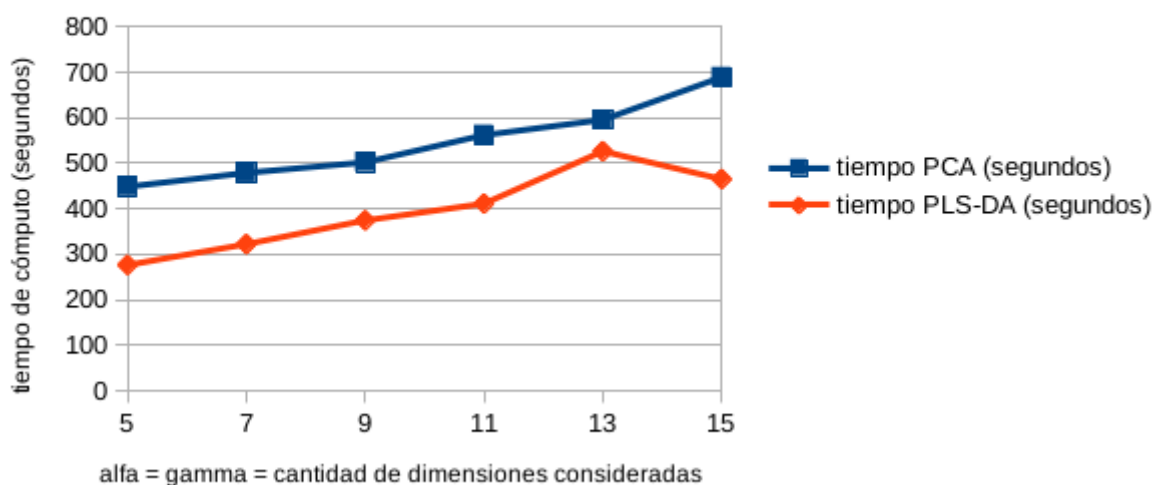
Pensamos que aumentar la cantidad  $k$  de vecinos de kNN, hará que aumente la cantidad de aciertos. Pero si la llegáramos a aumentar demasiado, podría llegar a perjudicar la clasificación (a diferencia de aumentar la cantidad de dimensiones, que suponemos que sólo pagaríamos con mayor tiempo de cómputo).

#### 4.2.2. Resultados y discusión

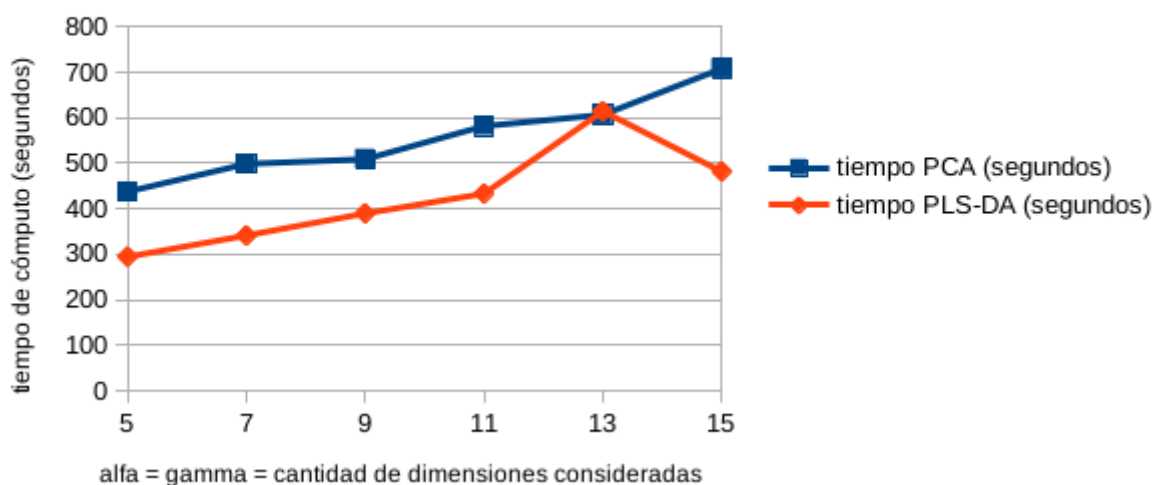
En los siguientes tres gráficos se muestra la variación en el tiempo de cómputo para los métodos PCA y PLS-DA comparándolos entre si, y variando la cantidad de vecinos mas cercanos a la hora de evaluarlos con Knn.



## PCA vs PLS-DA (tiempo de cómputo)

kNN con los  $k = 5$  vecinos más cercanos,  $K = 10$  pliegues de validación cruzada

## PCA vs PLS-DA (tiempo de cómputo)

kNN con los  $k = 7$  vecinos más cercanos,  $K = 10$  pliegues de validación cruzada

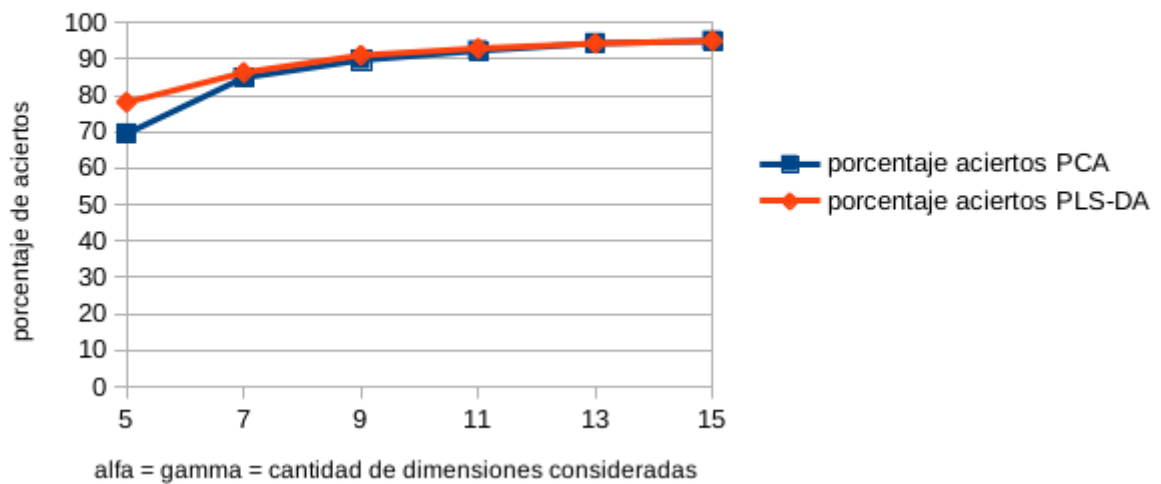
Como pudimos observar, para los tres experimentos el tiempo de computo de PLS-DA es siempre menor al de PCA. En el caso del primero y el ultimo, parece que ambos métodos parecen coincidir en tiempo cuando se considera trece dimensiones. A partir de este punto ambas métricas difieren, aumentando el tiempo de computo en PCA y disminuyendo para PLS-DA. Desconocemos a que se debe este hecho. Quedara como experimento a futuro considerar mas dimensiones para ver que sucede posteriormente e intentar concluir algo.

Por otro lado, para los tres  $k$  distintos los tiempos de computo parecen ser muy similares, es decir que no se observa un cambio que consideremos importante cambiando la cantidad de vecinos mas cercanos entre 3, 5 y 7. Esto puede deberse a que las operaciones que involucran a los  $k$  vecinos dentro del lenguaje utilizado para programarlos no varíen mucho por agregar solamente 2 y 4 números mas respectivamente.

En los gráficos que siguen analizaremos la eficiencia de ambos métodos a la hora de clasificar las imágenes. Para esto comparamos a PCA y PLS-DA nuevamente variando la cantidad de vecinos mas cercanos y utilizaremos las métricas de porcentaje de aciertos, precision y recall.

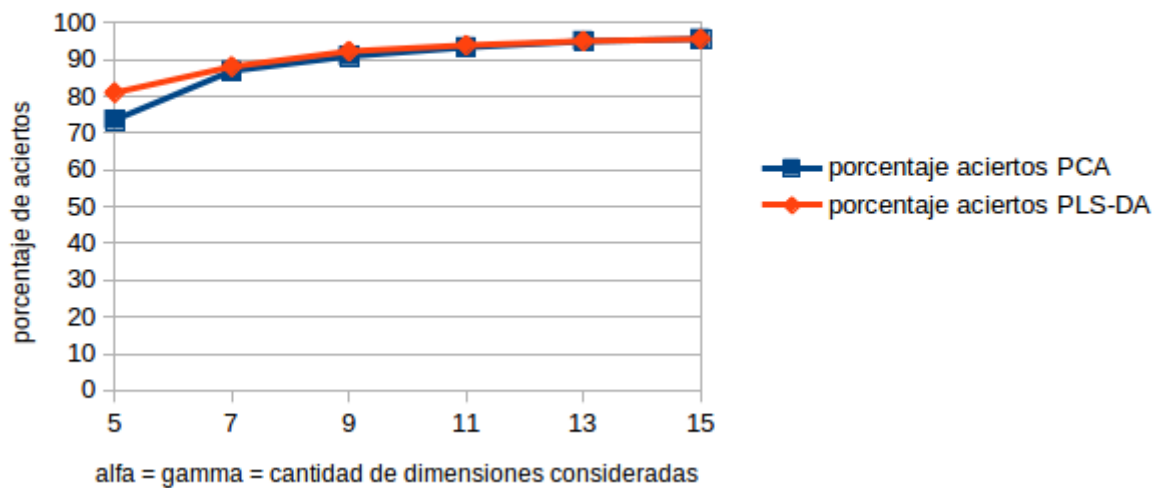
### PCA vs PLS-DA (porcentaje de aciertos)

kNN con los  $k = 3$  vecinos más cercanos,  $K = 10$  pliegues de validación cruzada

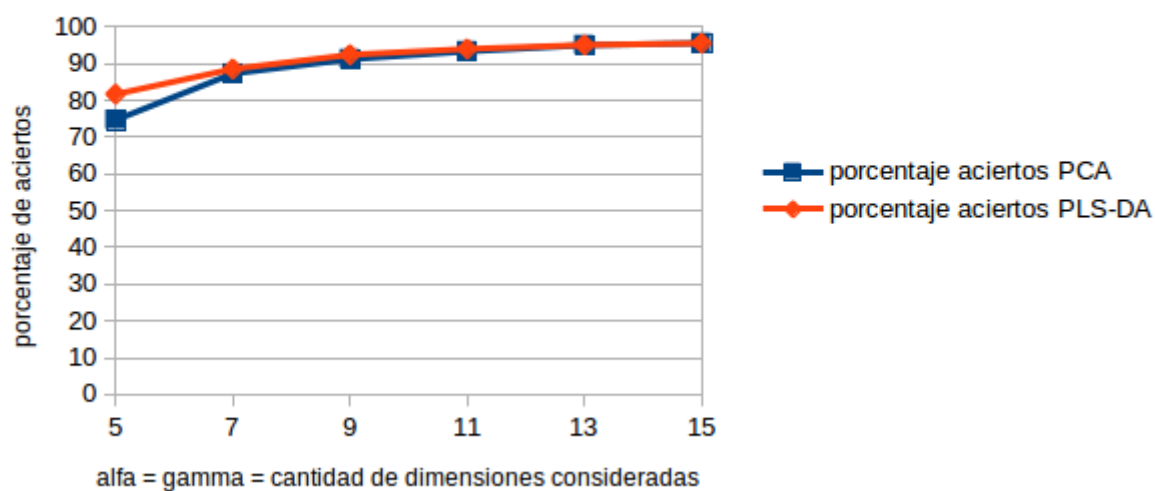


### PCA vs PLS-DA (porcentaje de aciertos)

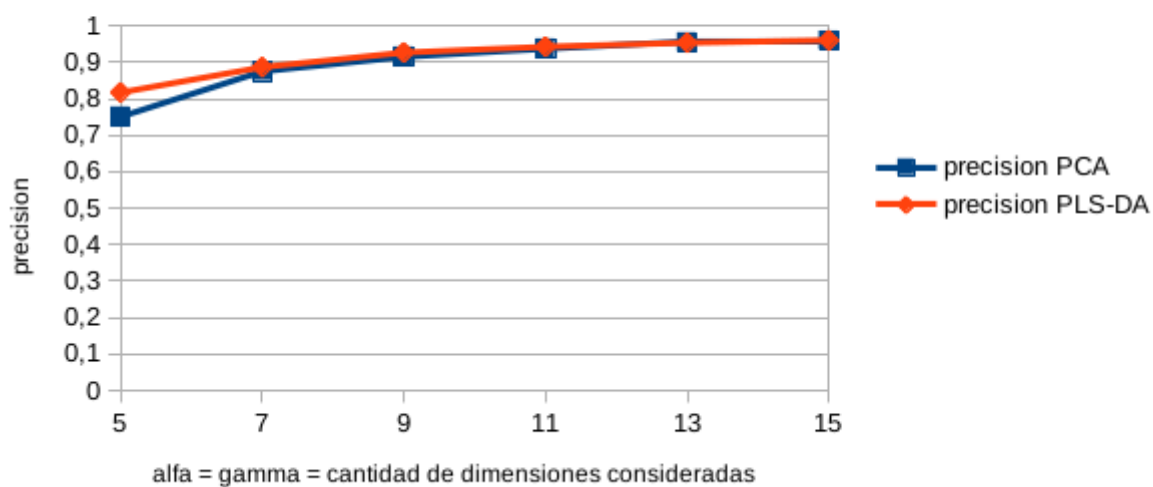
kNN con los  $k = 5$  vecinos más cercanos,  $K = 10$  pliegues de validación cruzada



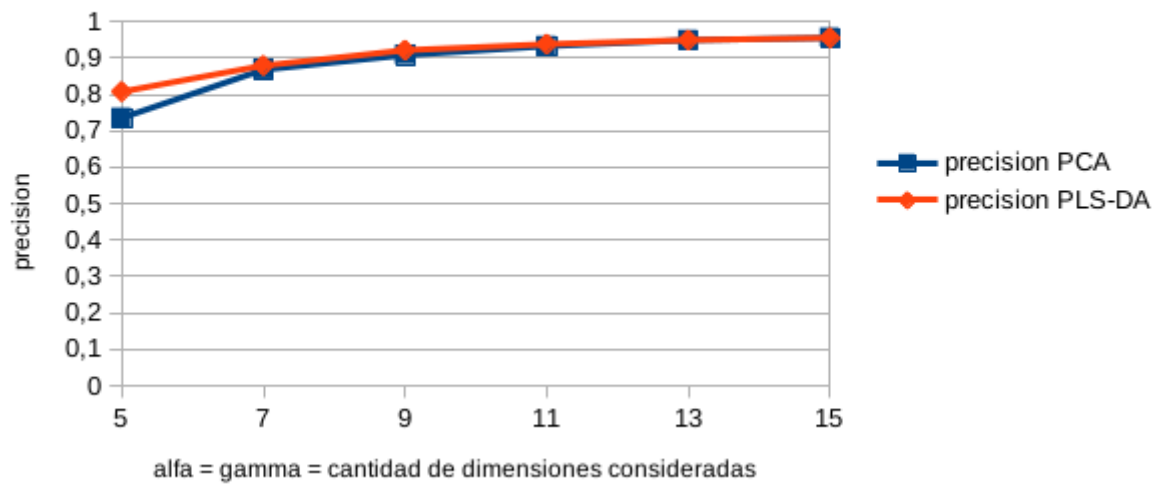
## PCA vs PLS-DA (porcentaje de aciertos)

kNN con los  $k = 7$  vecinos más cercanos,  $K = 10$  pliegues de validación cruzada

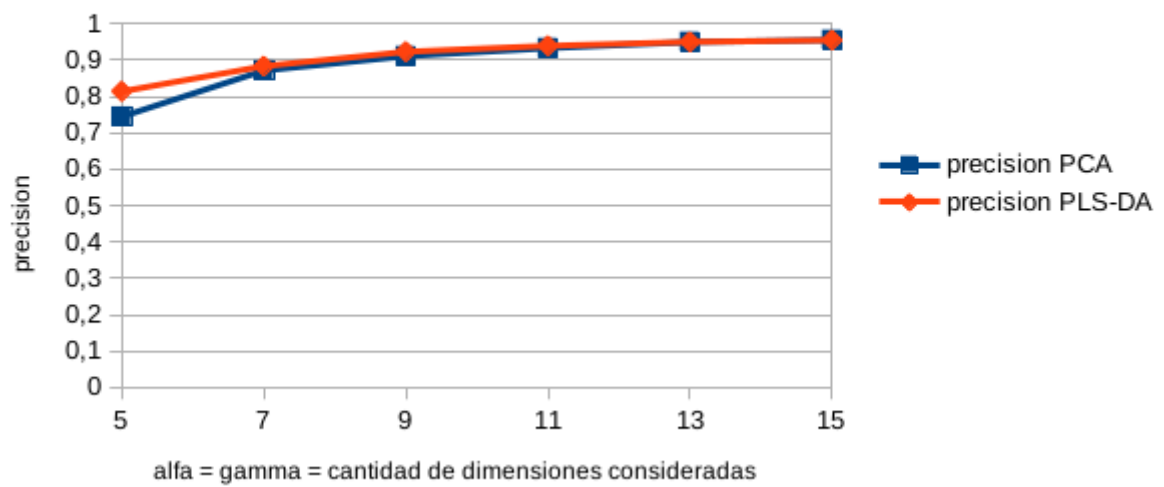
## PCA vs PLS-DA (precision)

kNN con los  $k = 3$  vecinos más cercanos,  $K = 10$  pliegues de validación cruzada

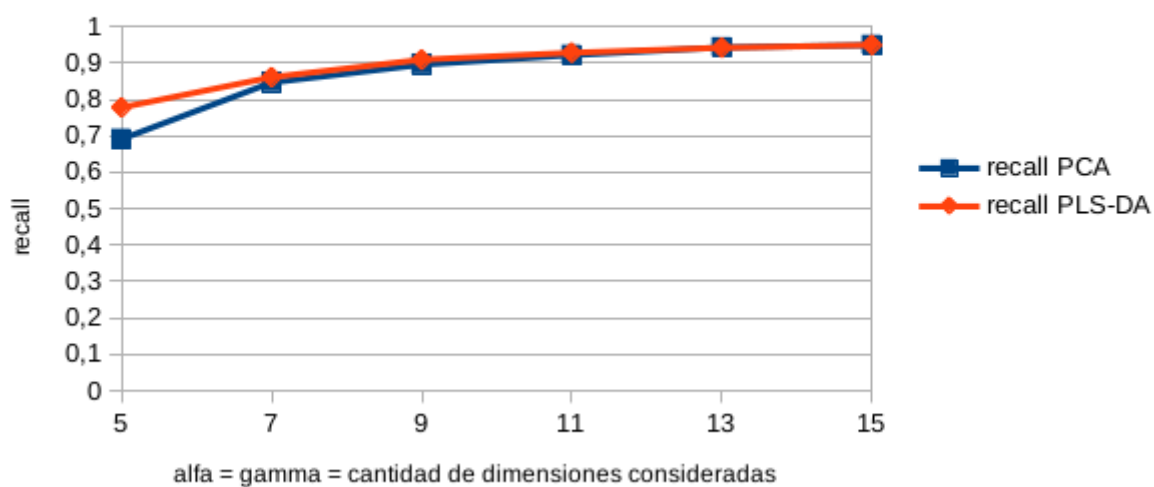
## PCA vs PLS-DA (precision)

kNN con los  $k = 5$  vecinos más cercanos,  $K = 10$  pliegues de validación cruzada

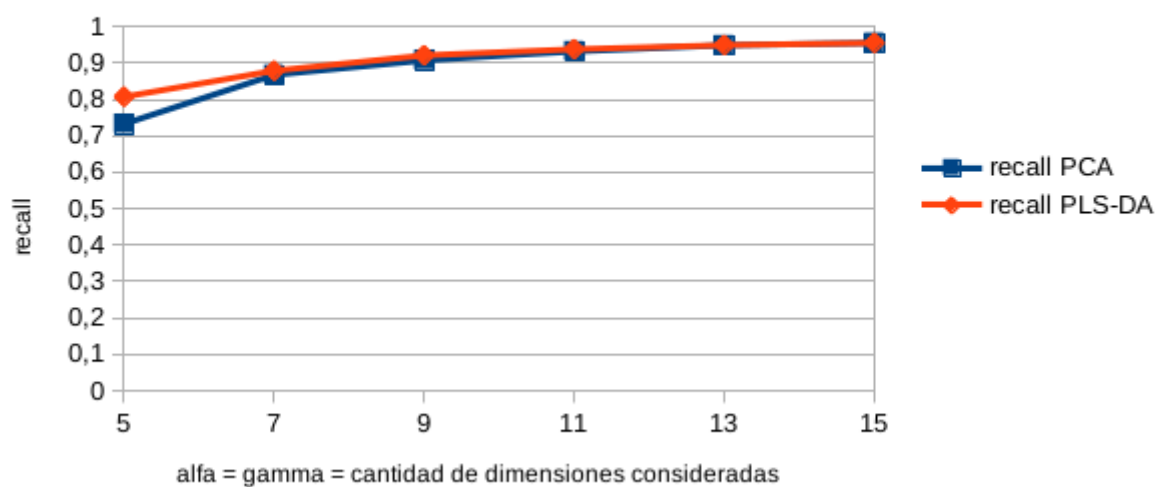
## PCA vs PLS-DA (precision)

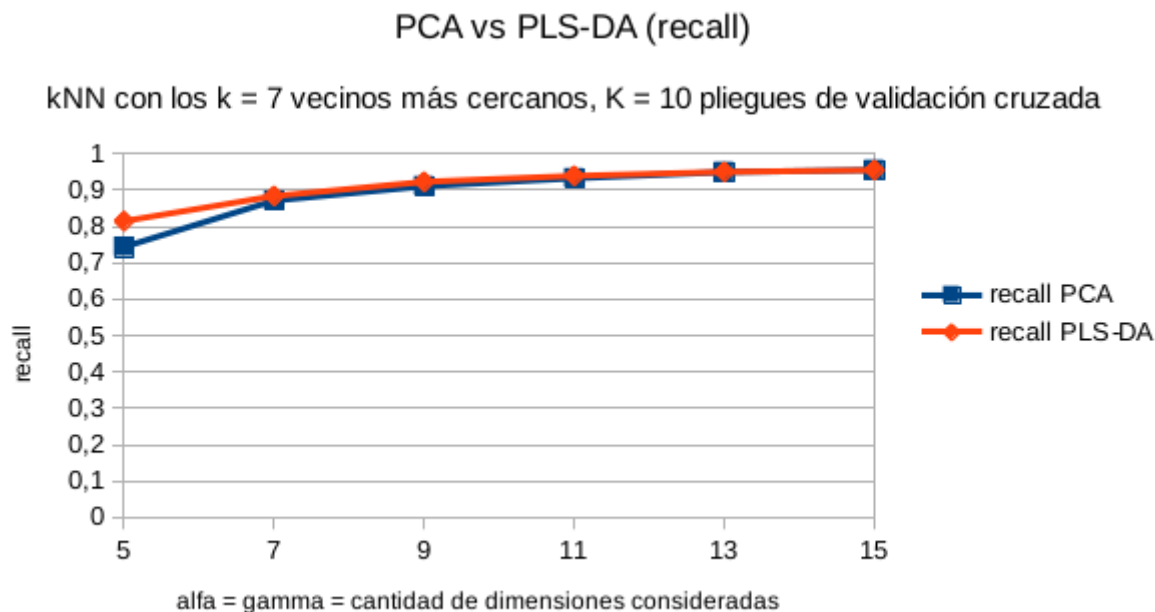
kNN con los  $k = 7$  vecinos más cercanos,  $K = 10$  pliegues de validación cruzada

## PCA vs PLS-DA (recall)

kNN con los  $k = 3$  vecinos más cercanos,  $K = 10$  pliegues de validación cruzada

## PCA vs PLS-DA (recall)

kNN con los  $k = 5$  vecinos más cercanos,  $K = 10$  pliegues de validación cruzada



Como se pudo ver, ambos métodos de clasificación se comportan de manera muy similar en todos los casos, en el sentido de que PLS-DA obtiene mejores resultados que PCA cuando se consideran 5 dimensiones, y a partir de 7 dimensiones en adelante estas dos parecen obtener resultados muy parecidos para todos los  $k$  y para todas las métricas de evaluación utilizadas.

Por otro lado, a medida que suben las dimensiones, se puede ver como mejora tanto porcentaje de aciertos, como precisión y recall para todos los  $k$  considerados hasta obtener resultados por encima del 90 %, lo cual consideramos que es muy bueno. De todos modos, esto ultimo no parece depender en absoluto de la variación de los  $k$  vecinos mas cercanos utilizados para la realización de Knn.

Una posible explicación para la no variación de los aciertos al cambiar el parámetro  $k$  puede ser que la clasificación de las imágenes en sus respectivas clases sea tan eficiente que ya a partir de  $k = 3$  sea suficiente para determinar a que clase pertenece.

#### 4.3. Experimento 2: PCA + kNN y PLS-DA + kNN variando la cantidad de imágenes de entrenamiento

En este experimento observar como van cambiando los resultados de PCA y de PLS-DA cuando varia la cantidad de temporadas de entrenamiento que utilizamos para calcular a cada uno de ellos. También compararemos a ambos métodos.

##### 4.3.1. Hipótesis

Suponemos que mientras más imágenes usemos para entrenar, mejores van a ser nuestros resultados en cuanto a porcentaje de aciertos.

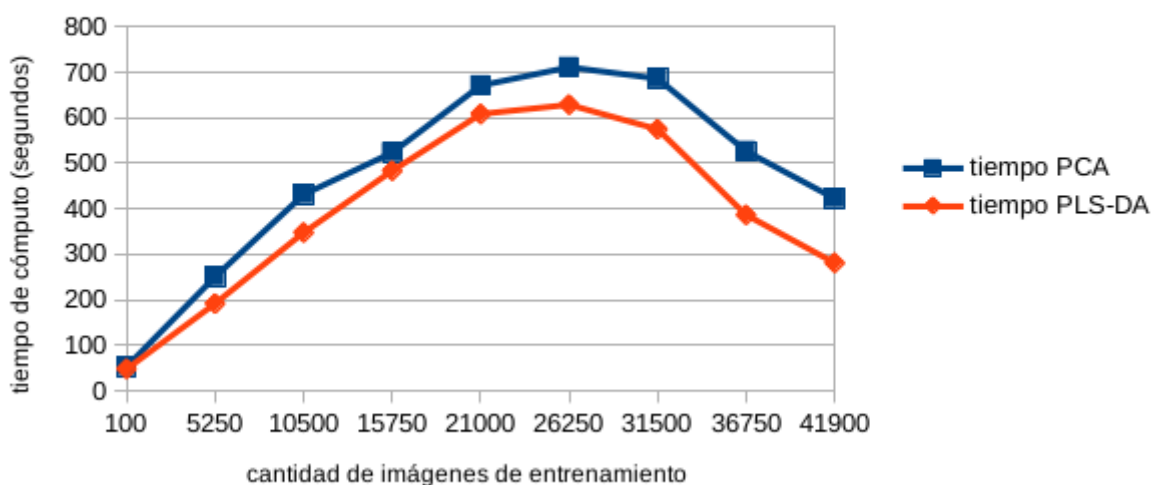
##### 4.3.2. Resultados y discusión

En los siguientes dos gráficos se muestra la variación de tiempo de computo y la variación en porcentaje de aciertos respectivamente comparando los dos métodos entre si.



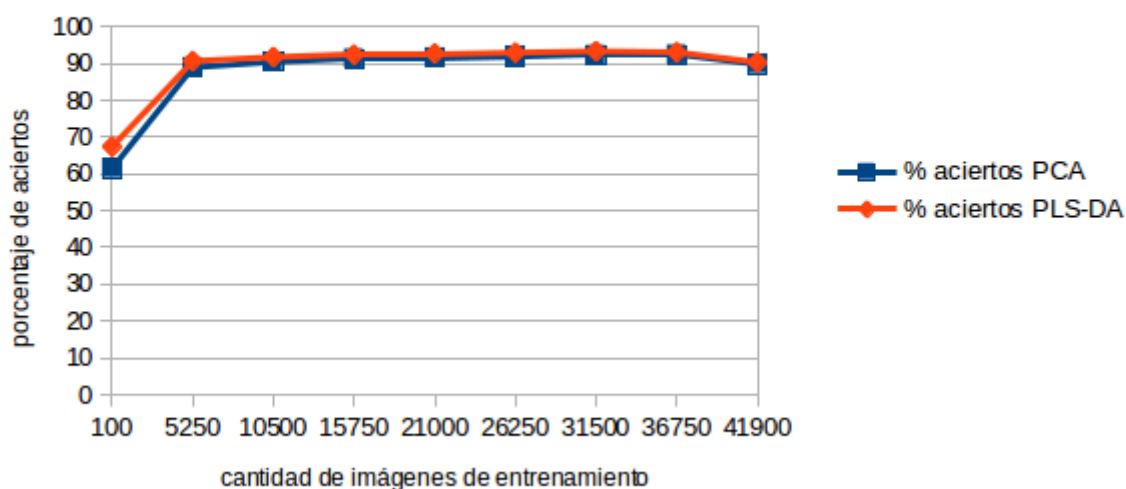
## PCA vs PLS-DA (tiempo de cómputo)

kNN con los  $k = 5$  vecinos más cercanos,  $K = 10$  iteraciones,  $\alpha = \gamma = 10$  dimensiones



## PCA y PLS-DA (porcentaje de aciertos)

kNN con los  $k = 5$  vecinos más cercanos,  $K = 10$  iteraciones,  $\alpha = \gamma = 10$  dimensiones



En el primer gráfico se ve como el tiempo de computo aumenta de manera similar en ambos métodos a medida que aumenta la cantidad de imágenes de entrenamiento y a partir de cierta cantidad, este comienza a disminuir. Suponemos que esto se debe al método utilizado para testear, el cual toma cierta cantidad de imágenes de entrenamiento y testea en todas las demás. De este modo, a medida que aumenta la cantidad de imágenes de entrenamiento, disminuye la cantidad de imágenes a ser testeadas, por ello, a partir de cierto punto, se encuentra muchas imágenes para entrenar y muy pocas para testear, con lo cual el tiempo de test disminuye. Si tenemos en cuenta que tanto las imágenes de entrenamiento como las de test se encuentran en dos matrices diferentes, esto puede ser pensado como que a medida que aumenta el tamaño de una matriz, disminuye el tamaño de la otra.

Pasando al segundo gráfico, es muy claro como el porcentaje de aciertos de ambos métodos aumenta a medida que se considera mas imágenes de entrenamiento, y parecen estabilizarse ambos en el 90 % de aciertos a partir de 5250 imágenes de entrenamiento. A partir de ese punto no parece influir en gran medida la cantidad de imágenes de entrenamiento utilizadas y hacia el final el porcentaje de aciertos disminuye levemente. Esto puede deberse a que incluyendo menos de 5250 imágenes no se

llega a considerar suficientes imágenes de cada clase para realizar la clasificación. Luego, a partir de ese número, la cantidad de imágenes ya es suficiente y por ello el porcentaje de aciertos se mantiene estable. Finalmente, no sabemos a qué se debe la leve disminución en porcentaje de aciertos hacia el final. Una idea que se nos ocurrió es que esto puede deberse a que utilizando tantas imágenes de entrenamiento, se acumulan datos sobre las mismas que, a pesar de los métodos de pre procesamiento, generan error. De todos modos haría falta realizar más experimentos para poder sostener dicha hipótesis.

#### 4.4. Experimento 3: variando $K$

La idea de este experimento, es repetir las experiencias de los experimentos 1 y 2, pero variando la cantidad  $K$  de pliegues a considerar para la validación cruzada.

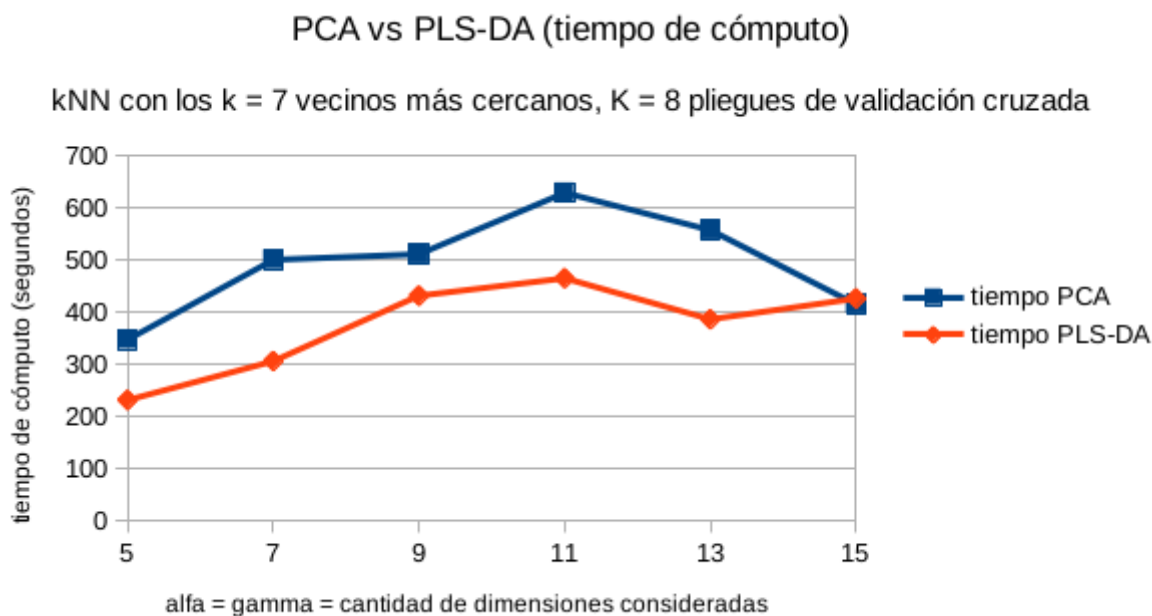
##### 4.4.1. Hipótesis

Para una mayor cantidad de pliegues, esperamos obtener mayores tiempos de cómputo en ambos experimentos.

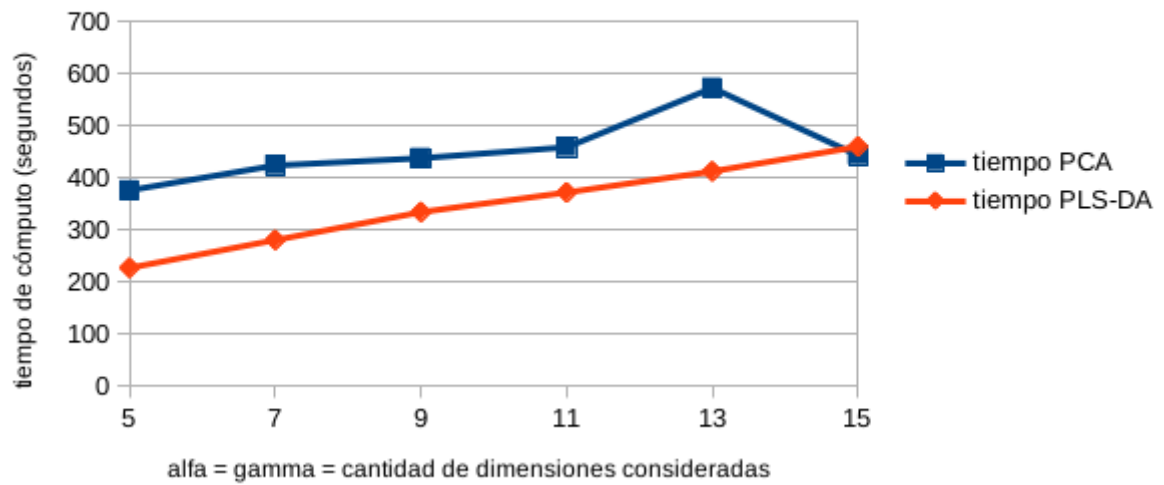
También pensamos que con mayor cantidad de pliegues nuestras métricas serán más precisas.

##### 4.4.2. Resultados y discusión

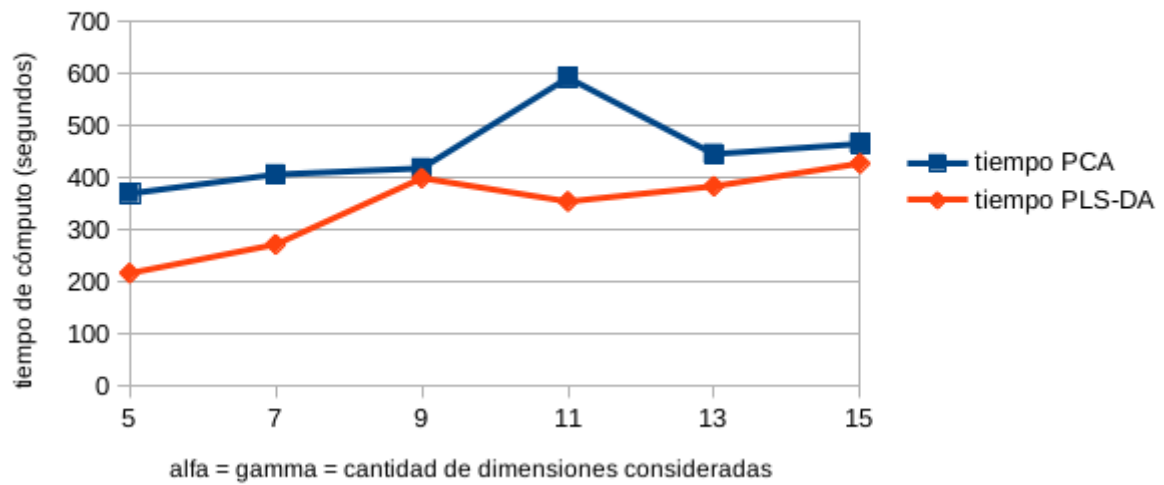
En los próximos 9 gráficos se muestra el resultado del experimento 1 para  $K = 8, 6$  y  $4$  pliegues de validación cruzada.



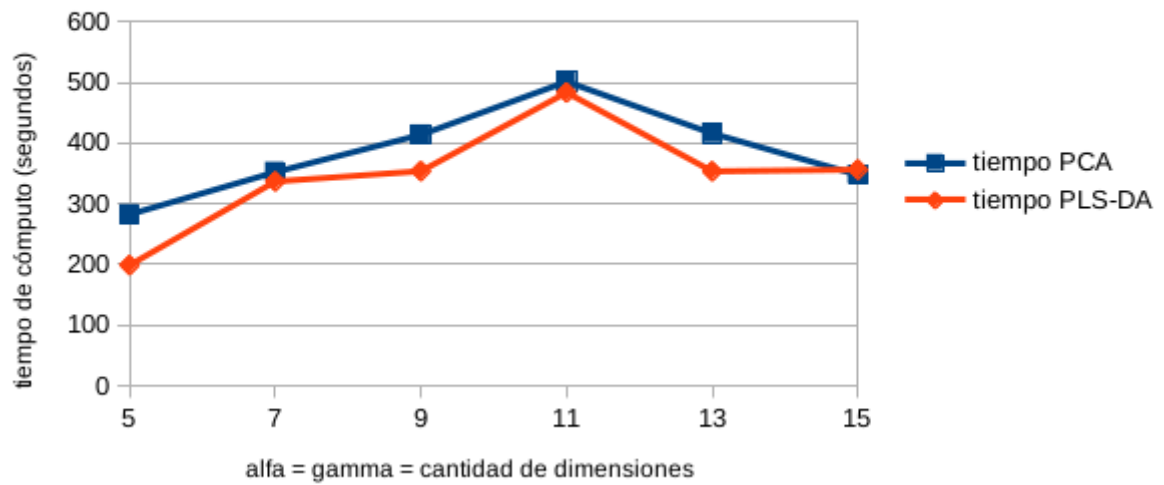
## PCA vs PLS-DA (tiempo de cómputo)

kNN con los  $k = 5$  vecinos más cercanos,  $K = 8$  pliegues de validación cruzada

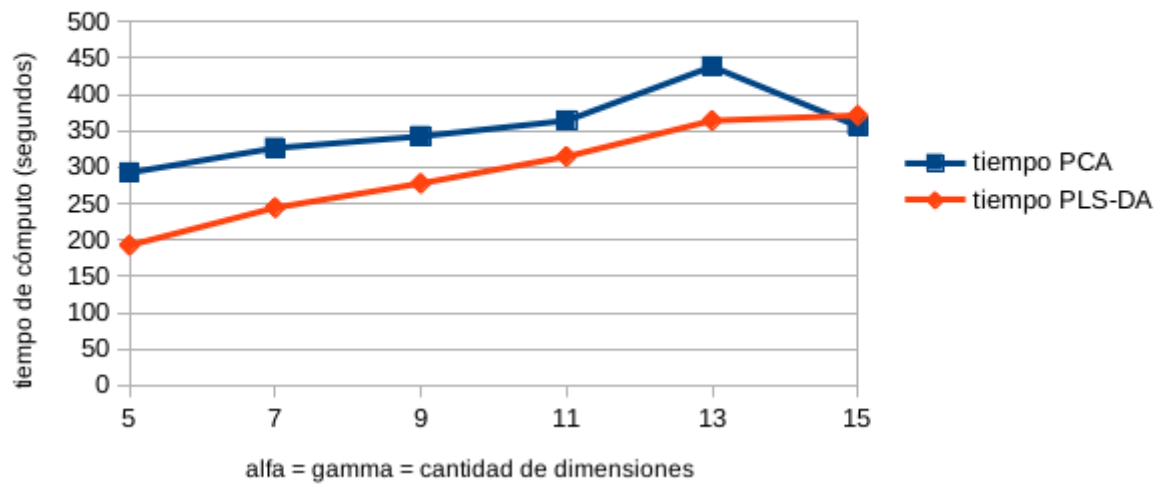
## PCA vs PLS-DA (tiempo de cómputo)

kNN con los  $k = 3$  vecinos más cercanos,  $K = 8$  pliegues de validación cruzada

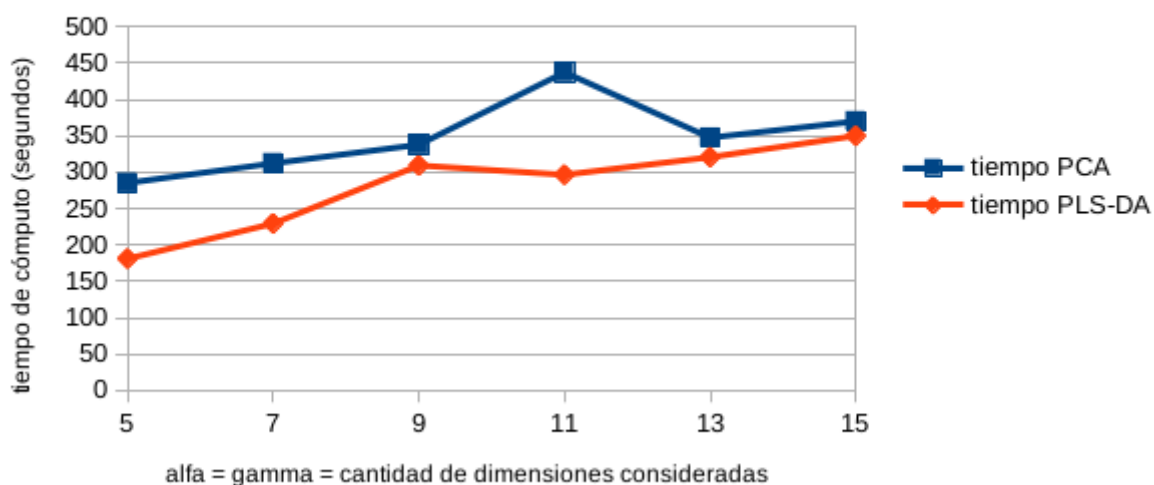
## PCA vs PLS-DA (tiempo de cómputo)

kNN con los  $k = 7$  vecinos más cercanos,  $K = 6$  pliegues de validación cruzada

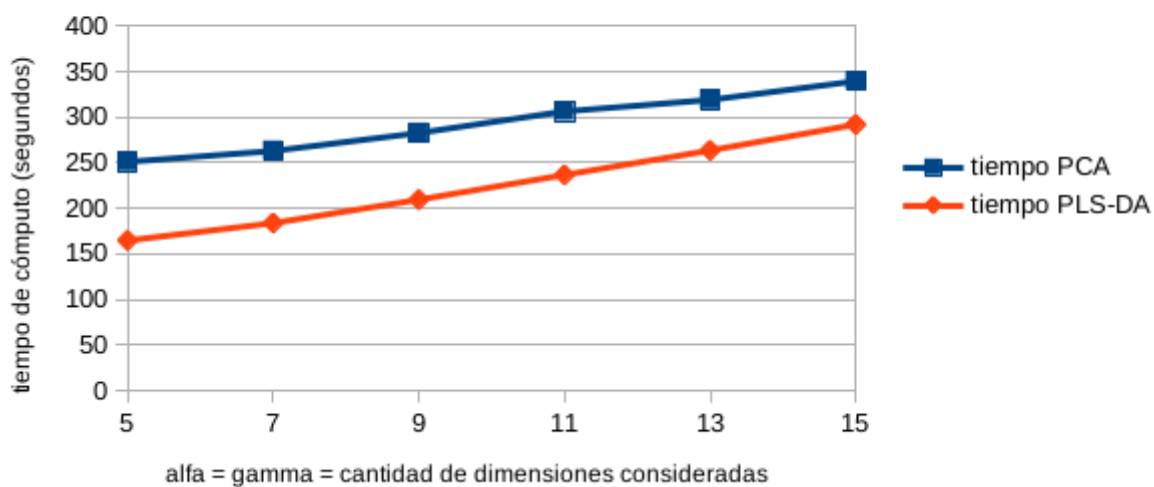
## PCA vs PLS-DA (tiempo de cómputo)

kNN con los  $k = 5$  vecinos más cercanos,  $K = 6$  pliegues de validación cruzada

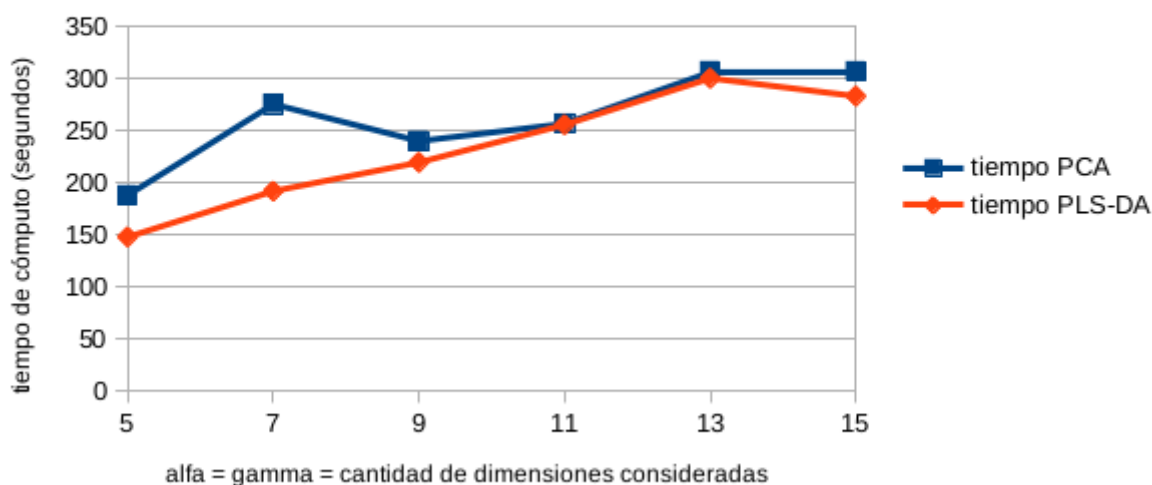
## PCA vs PLS-DA (tiempo de cómputo)

kNN con los  $k = 3$  vecinos más cercanos,  $K = 6$  pliegues de validación cruzada

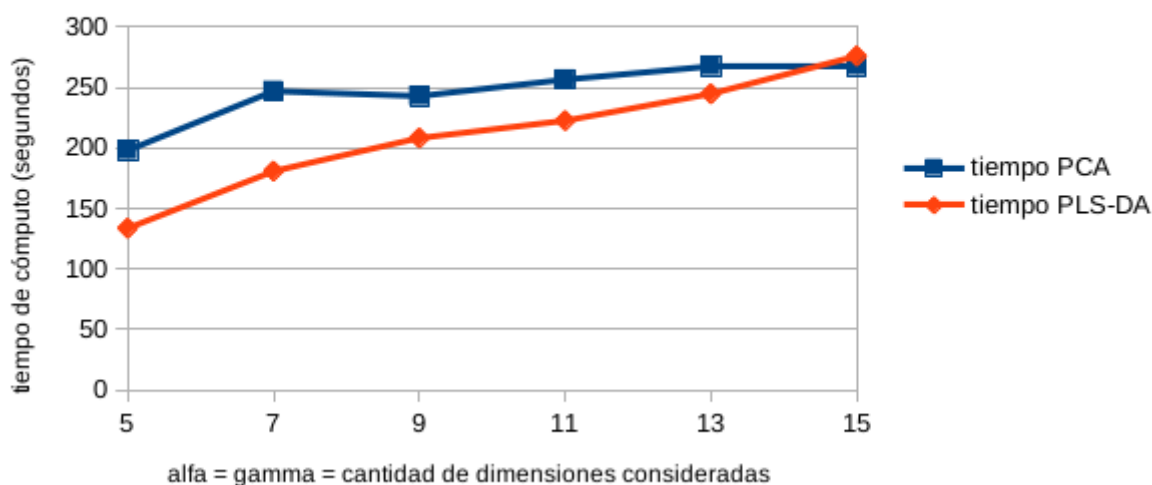
## PCA vs PLS-DA (tiempo de cómputo)

kNN con los  $k = 7$  vecinos más cercanos,  $K = 4$  pliegues de validación cruzada

## PCA vs PLS-DA (tiempo de cómputo)

kNN con los  $k = 5$  vecinos más cercanos,  $K = 4$  pliegues de validación cruzada

## PCA vs PLS-DA (tiempo de cómputo)

kNN con los  $k = 3$  vecinos más cercanos,  $K = 4$  pliegues de validación cruzada

A partir de los gráficos podemos ver como en general PLS-DA se mantiene por debajo de PCA en tiempo de cómputo. En algunos casos ambos métodos parecen coincidir para ciertas dimensiones y en muchos de ellos pareciera que a partir de la dimensión 15 el tiempo de PCA llega a ser levemente menor al de PLS-DA. Al igual que en el experimento 1, queda como experimento a futuro considerar muchas mas dimensiones a la hora de realizar estas mediciones para así poder llegar a conclusiones concretas en cuanto a lo que sucede con los tiempos en las dimensiones en cuestión y por que los mismos disminuyen luego de cierto punto en algunos casos.

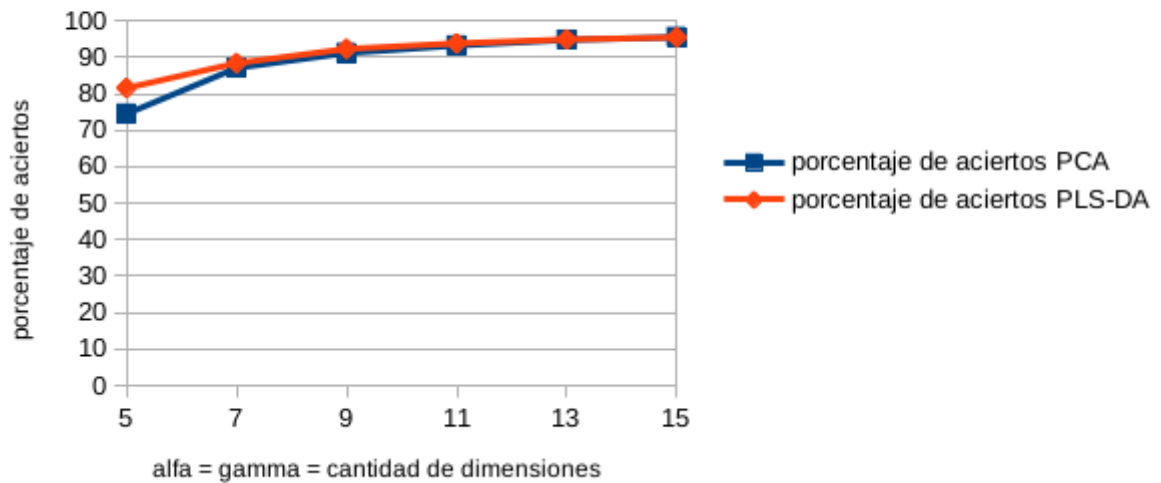
Por otro lado podemos observar como, a diferencia del experimento 1, los tiempos varían de manera un poco mas significativa para cada cantidad de pliegues distinta. Si bien para la primera cantidad de dimensiones ambos métodos parten con tiempos no muy diferentes a medida que varían los  $K$ , se puede ver que a medida que las dimensiones aumentan la diferencia de tiempos entre cada pliegue es cada vez mayor. Se podría decir que la diferencia de tiempos entre los métodos analizados con distintas cantidades de pliegues aumenta proporcionalmente a las dimensiones consideradas. Esto tiene

sentido ya que cuanto más pliegues haya, más veces se deberán procesar nuevamente todos los datos, y cuantas más dimensiones haya, mayores serán los datos a ser procesados.

Estos otros 9 gráficos que siguen muestran el porcentaje de aciertos experimento 1 para 4, 6 y 8 pliegues de validación cruzada.

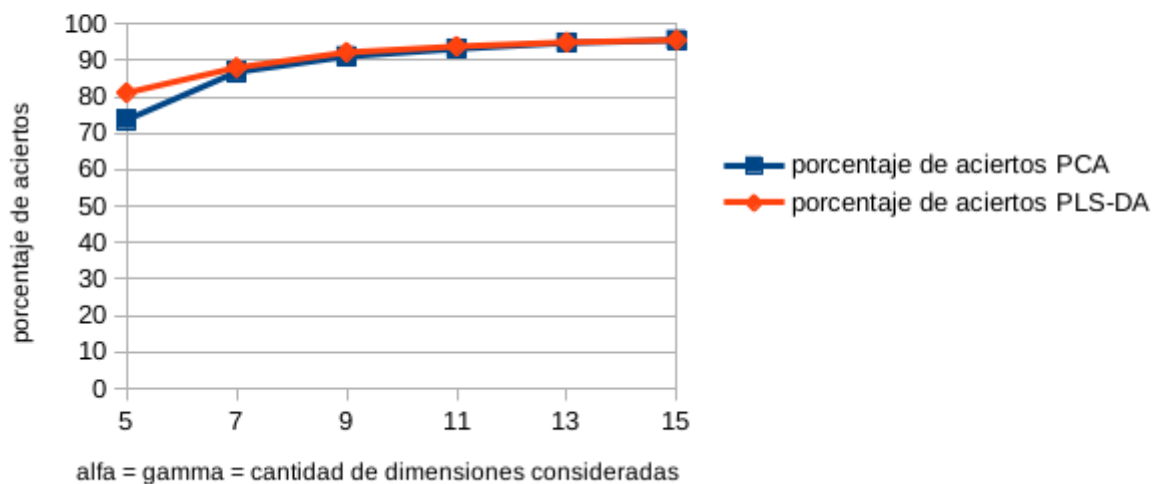
### PCA vs PLS-DA (porcentaje de aciertos)

kNN con los  $k = 7$  vecinos más cercanos,  $K = 8$  pliegues de validación cruzada

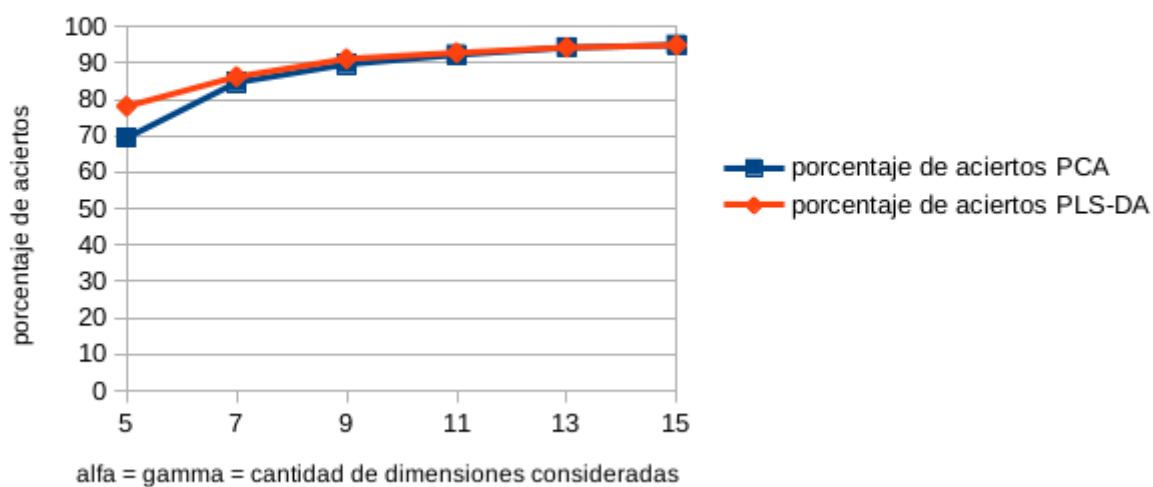


### PCA vs PLS-DA (porcentaje de aciertos)

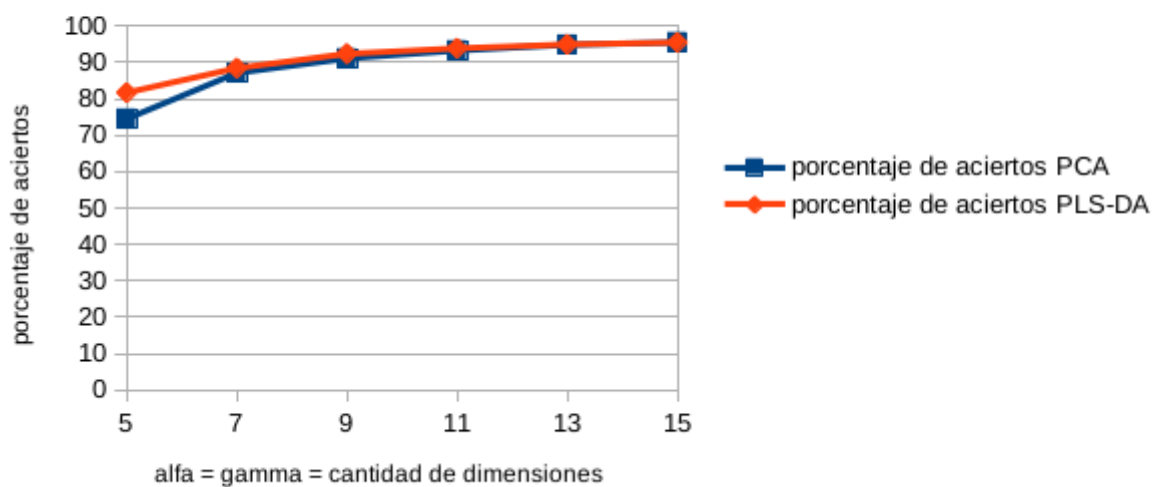
kNN con los  $k = 5$  vecinos más cercanos,  $K = 8$  pliegues de validación cruzada



## PCA vs PLS-DA (porcentaje de aciertos)

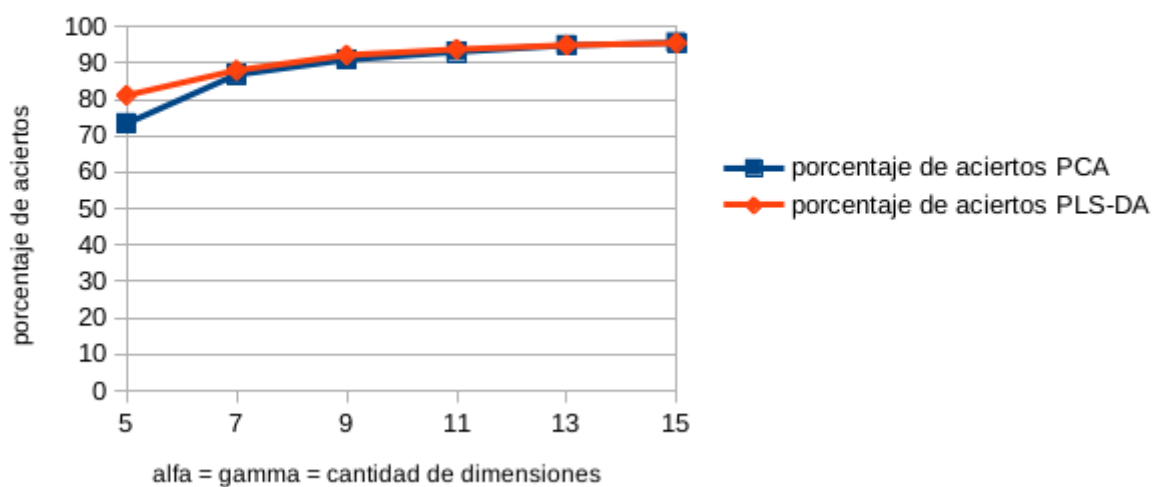
kNN con los  $k = 3$  vecinos más cercanos,  $K = 8$  pliegues de validación cruzada

## PCA vs PLS-DA (porcentaje de aciertos)

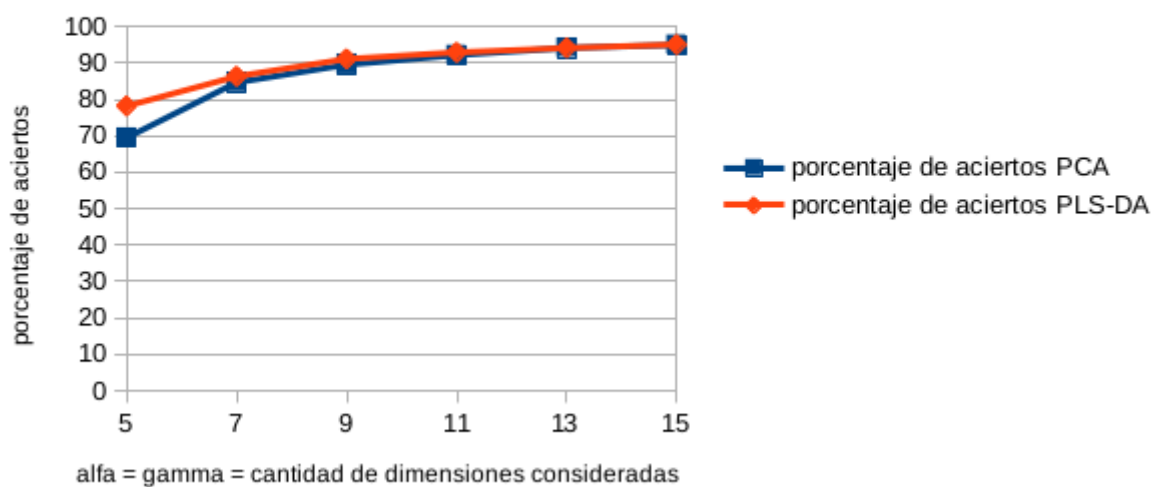
kNN con los  $k = 7$  vecinos más cercanos,  $K = 6$  pliegues de validación cruzada



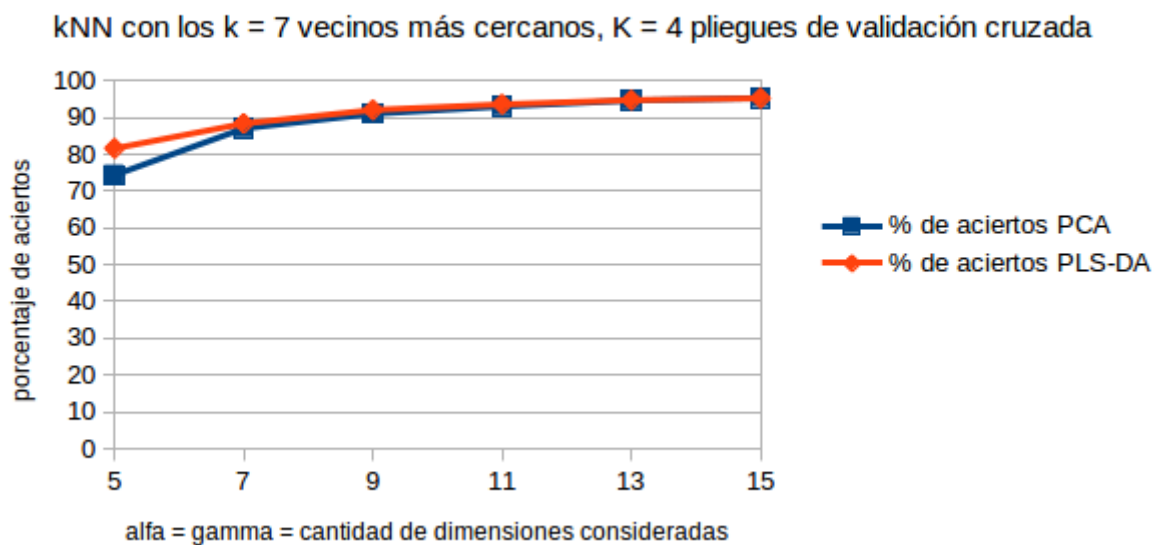
## PCA vs PLS-DA (porcentaje de aciertos)

kNN con los  $k = 5$  vecinos más cercanos,  $K = 6$  pliegues de validación cruzada

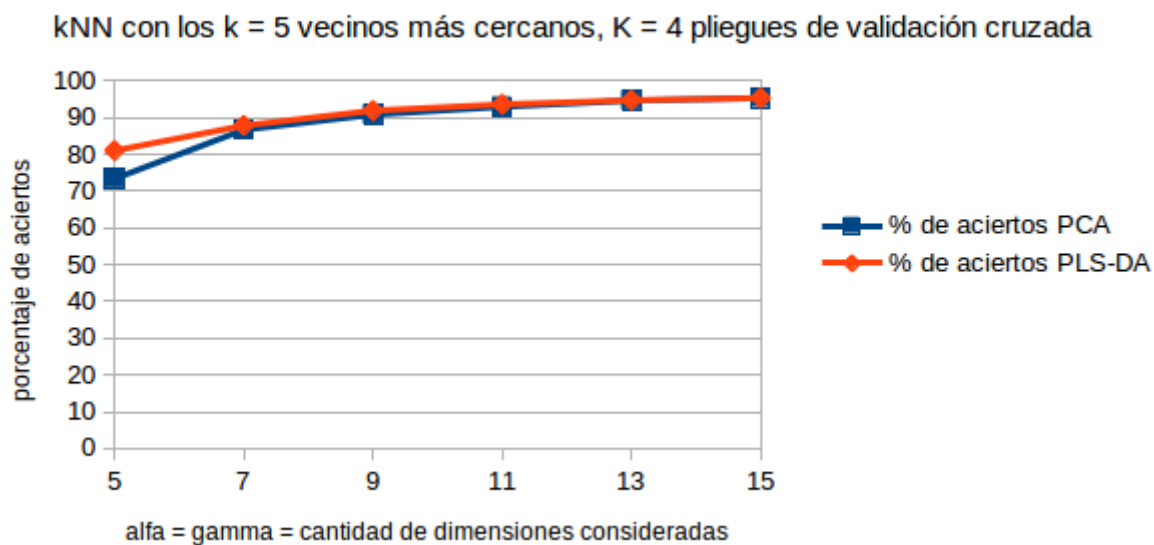
## PCA vs PLS-DA (porcentaje de aciertos)

kNN con los  $k = 3$  vecinos más cercanos,  $K = 6$  pliegues de validación cruzada

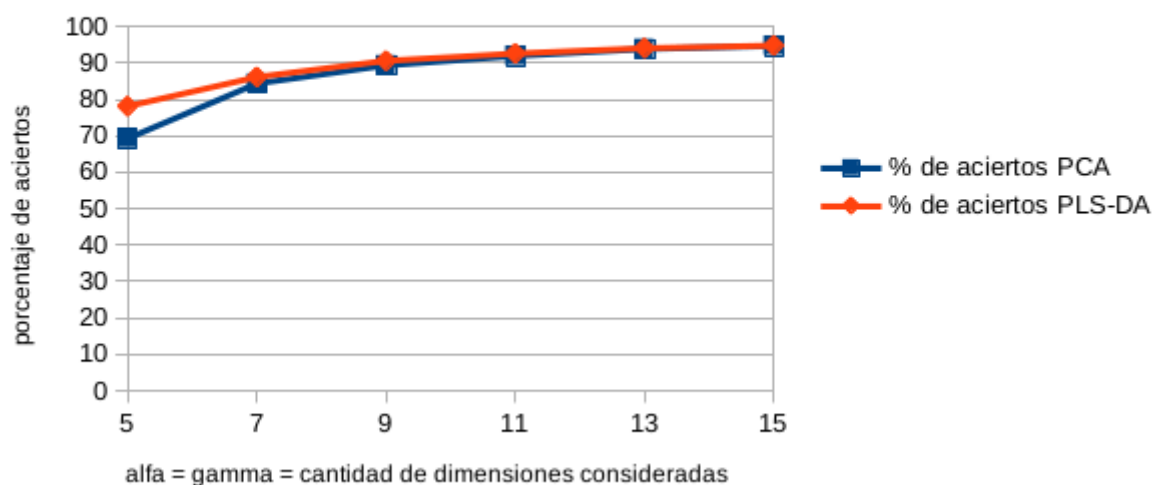
## PCA vs PLS-DA (porcentaje de aciertos)



## PCA vs PLS-DA (porcentaje de aciertos)



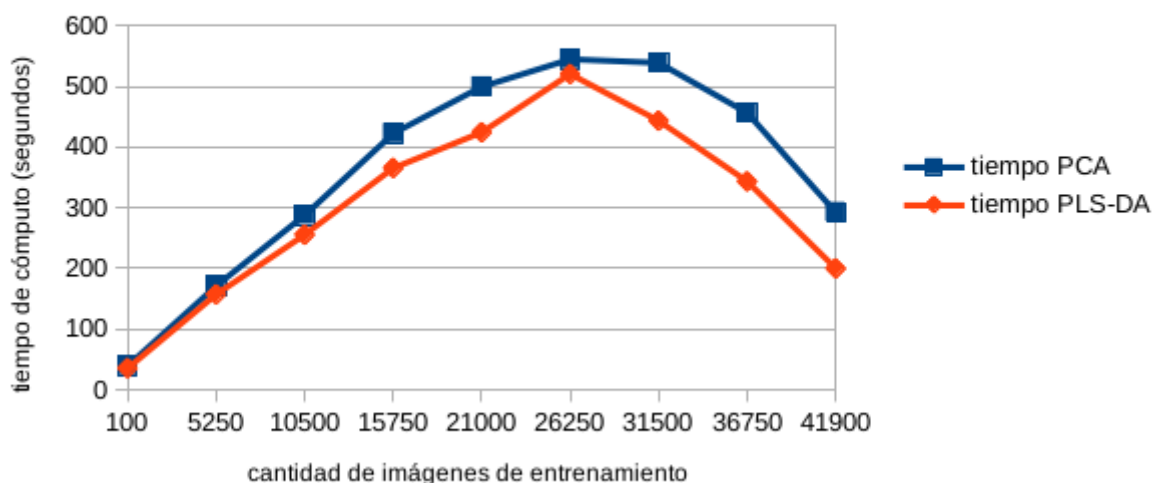
## PCA vs PLS-DA (porcentaje de aciertos)

kNN con los  $k = 3$  vecinos más cercanos,  $K = 4$  pliegues de validación cruzada

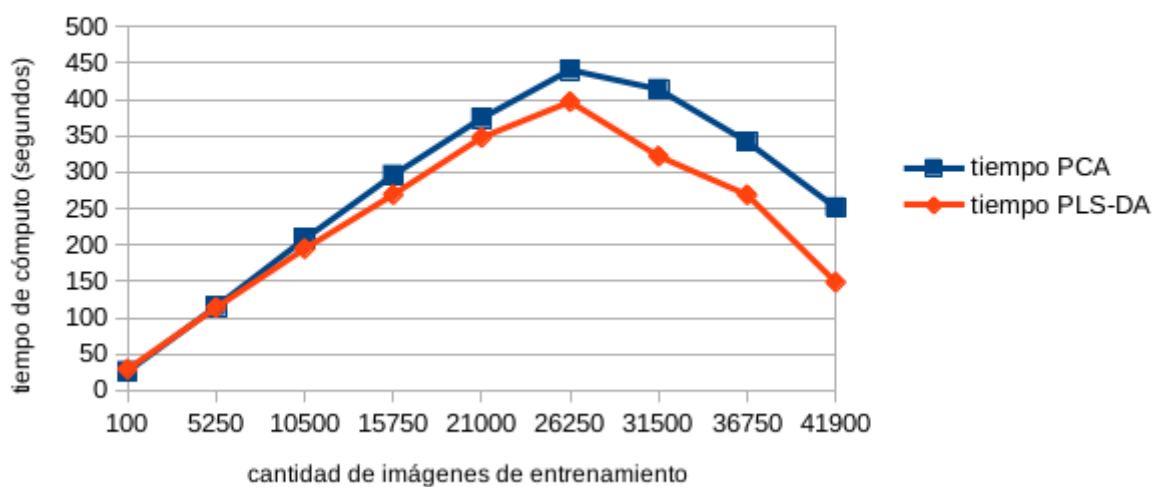
Como se puede ver, los porcentajes de aciertos se mantienen muy similares y hasta quizás iguales a los del experimento 1. En todos los casos PLS-DA comienza teniendo mejor porcentaje de aciertos considerando 5 dimensiones y luego ambos métodos obtienen los mismos resultados. Posiblemente esto se deba a que ya se obtenían muy buenos resultados con el resto de los parámetros, entonces la cantidad de pliegues no llega a ser tan influyente como para mejorar resultados tan eficientes.

En los últimos 6 gráficos veremos como varían los tiempos y porcentajes de aciertos del experimento 2 para distinta cantidad de pliegues.

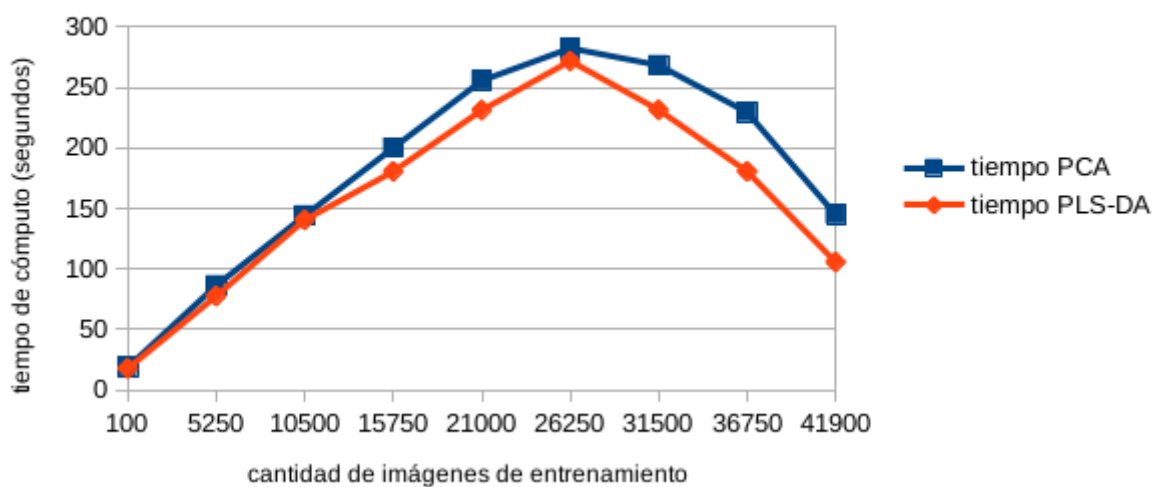
## PCA y PLS-DA (tiempo de cómputo)

kNN con los  $k = 5$  vecinos,  $K = 8$  iteraciones,  $\alpha = \gamma = 10$  dimensiones

## PCA y PLS-DA (tiempo de cómputo)

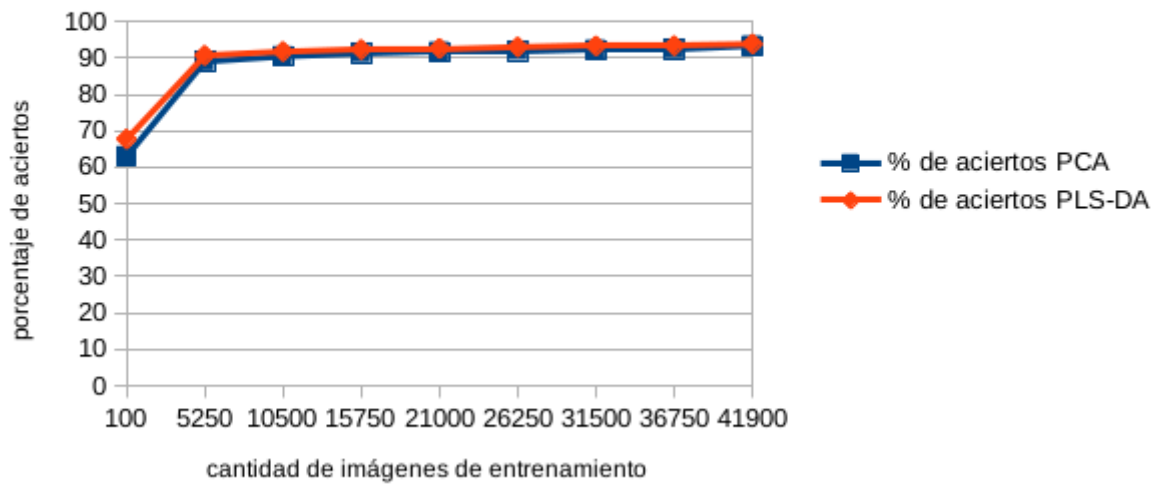
kNN con los  $k = 5$  vecinos más cercanos,  $K = 6$  iteraciones,  $\alpha = \gamma = 10$  dimensiones

## PCA y PLS-DA (tiempo de cómputo)

kNN con los  $k = 5$  vecinos más cercanos,  $K = 4$  iteraciones,  $\alpha = \gamma = 10$  dimensiones

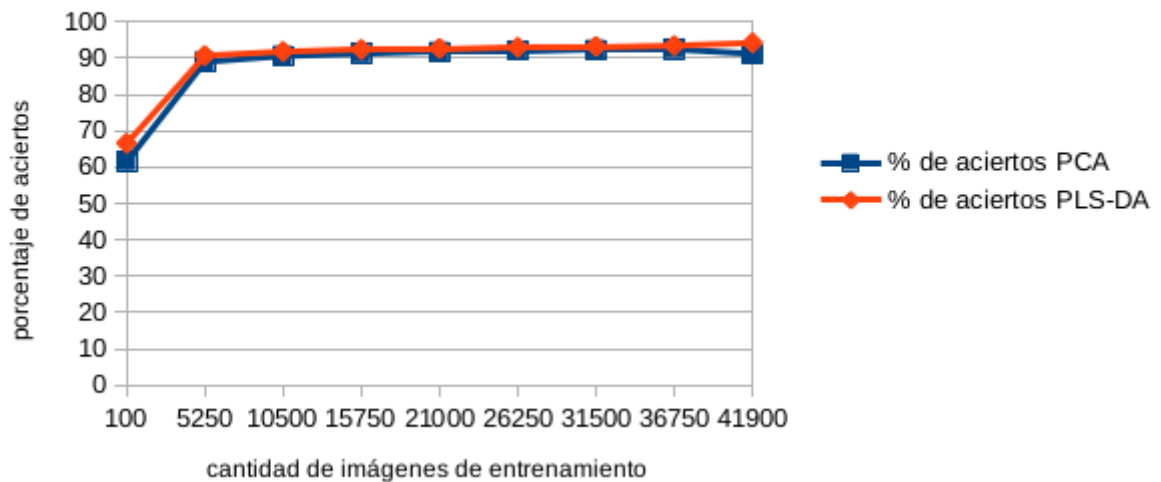
## PCA y PLS-DA (porcentaje de aciertos)

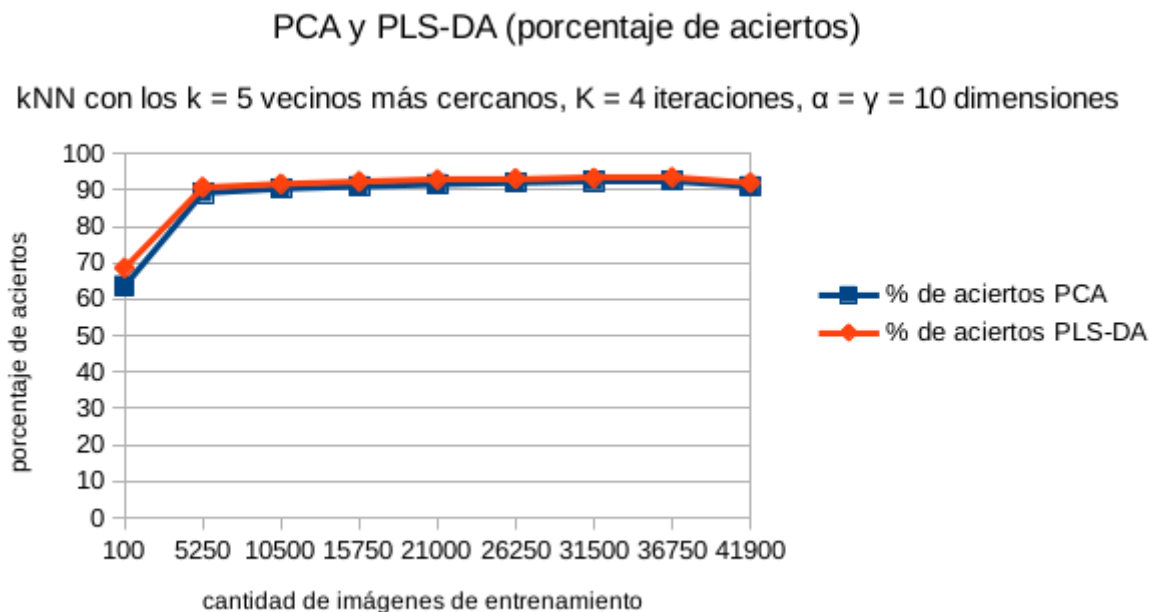
kNN con los  $k = 5$  vecinos más cercanos,  $K = 8$  iteraciones,  $\alpha = \gamma = 10$  dimensiones



## PCA vs PLS-DA (porcentaje de aciertos)

kNN con los  $k = 5$  vecinos más cercanos,  $K = 6$  iteraciones,  $\alpha = \gamma = 10$  dimensiones





En cuanto a los primeros tres gráficos, los de tiempo de cómputo, parece que nuestra hipótesis se confirma, es decir, el tiempo aumenta a medida que se aumenta la cantidad de pliegues. Creemos que esto se debe a que, al considerar más pliegues, los datos se deben procesar nuevamente por completo y esta es la parte que más tiempo toma.

Para los de porcentaje de aciertos, no se observan cambios, a excepción de donde se considera la máxima cantidad de imágenes de entrenamiento. A medida que aumentan los pliegues, se elimina la pendiente negativa. Esto último parece estar relacionado con lo planteado en nuestra hipótesis ya que mejora la clasificación en el único tramo donde esta empeoraba. Suponemos que esto se produce ya que, al haber más pliegues, hay más posibilidades de que se produzca una mejora de los datos y se evite entrenar siempre sobre cosas similares.

## 4.5. Observaciones

### 4.5.1. Tiempo de cómputo de kNN cambiando tipo de datos

kNN corre más rápido si definimos la matriz de imágenes de entrenamiento como vector de vectores de números enteros. Pero necesitamos que sea vector de vectores de números de punto flotante con precisión doble para poder aplicar los métodos de reducción.

### 4.5.2. Tiempo de cómputo doble ciclo anidado

Cuando tenemos dos ciclos anidados, si el ciclo de adentro hace el trabajo más pesado, termina mucho más rápido.

En nuestro caso, al multiplicar matrices por vectores, el proceso se realiza mucho más rápido cuando la matriz tiene menos filas. Conviene trasponer antes de realizar dicha multiplicación.

## 5. Dificultades encontradas y soluciones implementadas

### 5.1. Violación de segmento con `argv[0]` grande

`argv[0]` en C y en C++ es el vector de caracteres que guarda el nombre con el que se llamó al programa. Si se llama con la ruta completa (por ejemplo, `/home/user/documents/metnum/2c16/tp2/src/tp`)

puede provocar una violación de segmento difícil de rastrear.

Nos pasó esto con un script en un momento. La solución fue primero hacer `cd` al directorio donde está el ejecutable y luego llamar a `./tp`.

## **5.2. Automatización de las pruebas**

### **5.2.1. Espacio en disco insuficiente**

Cuando utilizamos las computadoras de los laboratorios de la facultad, no teníamos espacio suficiente entonces, al copiar y pegar, algunos tests.in quedaban rotos y luego cuando los usábamos para testear arrojaban resultados erróneos.

Quizá tenga que ver con un mensaje de error que vemos al conectarnos al servidor de acceso remoto de los laboratorios, que dice que no se pudo obtener la cuota de disco del usuario.

Nos vimos forzados a usar el disco local de cada computadora, en el que no corren los permisos de cada usuario. Esto conlleva riesgos de confidencialidad e integridad de los datos del trabajo práctico. Para disminuir ligeramente este riesgo, clonamos el repositorio dentro de una carpeta con punto (.) al principio, de forma tal que la mayoría de las veces no pueda ser vista (si el usuario no elige explícitamente ver las carpetas ocultas o con "ls -A"). Aunque este sistema de seguridad es bastante laxo.

Si el problema persiste más adelante, quizá implementemos carpetas cifradas o consultemos con los conservadores.

### **5.2.2. Usuarios apagan computadoras durante pruebas**

Encontramos que conviene correr las pruebas remotamente en los laboratorios durante el fin de semana o durante la noche, cuando no hay gente que las pueda apagar.

### **5.2.3. Computadora rechaza usuario y contraseña**

En una oportunidad nos sucedió de correr una serie de pruebas usando un script y que una de las máquinas nos pidiera contraseña (a pesar de que habíamos configurado para que así no fuera). Revisando, nos dimos cuenta que en esa computadora faltaban nuestros directorios de usuario. Y de hecho, había solamente 10 directorios de usuario en total. Por lo tanto, faltaban nuestros archivos de configuración de SSH.

Sorteamos el problema salteando esa máquina en los scripts.

## Parte III

# Conclusiones

## 6. Tiempos de computo y efectividad de las métricas

Basándonos en los 3 experimentos realizados, podemos concluir que, en todos los casos, ambos métodos utilizados tienen comportamientos muy similares. En cuanto a porcentaje de aciertos, con la cantidad de dimensiones suficientes, tanto PCA como PLS-DA, son igualmente efectivos a la hora de clasificar imágenes. Por otro lado, PLS-DA se mantiene siempre por debajo de PCA en tiempos de cómputo. Siendo esta última la diferencia más significativa entre ambos métodos, elegiríamos PLS-DA en el caso de tener que decidir entre una u otra.

Teniendo en cuenta el tercer experimento realizado, concluimos que: si aumentar la cantidad  $K$  de pliegues para validación cruzada aumenta el tiempo de computo a cambio de una leve mejora en porcentaje de aciertos, en general optaríamos por pagar dicho precio y tener resultados más precisos. De todos modos, esta decisión no será definitiva, ya que dependiendo del caso, puede requerirse priorizar el tiempo de cómputo.

Por otro lado, analizando el experimento 2, concluimos que se debe encontrar un buen balance entre cantidad de imágenes de test y cantidad de imágenes de entrenamiento. Al ir aumentando la cantidad de imágenes de entrenamiento, la efectividad de ambos métodos aumenta, aunque de manera muy sutil, pero esto puede llegar a ser muy útil según lo que se requiera. Parece una buena opción elegir muchas imágenes de entrenamiento, combinadas con muchos casos de prueba. De este modo, esta combinación reduciría el tiempo de cómputo (recordemos que en experimento 2 se muestra como a partir de cierta cantidad de imágenes de entrenamiento el tiempo de computo comienza a disminuir) y, al mismo tiempo mantendría buena efectividad en la clasificación.

## 7. Experimentos pendientes

Según pudimos observar durante la realización del trabajo, para las imágenes dadas se puede trabajar directamente con el tipo de dato *int* si aplicamos kNN sin pre-procesamiento. Al usar este tipo de dato el tiempo de cómputo se reduce de manera muy significativa y mantiene resultados muy buenos. Nos quedó pendiente considerar este caso y tomar mediciones precisas para mostrarlo, ya que a pesar de que lo pudimos probar mediante tests básicos, no pudimos realizar mediciones con la precisión que lo hicimos en otros experimentos.

Otro experimento que nos hubiese gustado realizar fue el de testear nuestro código con imágenes hechas por nosotros a mano y posteriormente editadas para que estén centradas, en escala de grises y con el tamaño deseado. De este modo hubiésemos podido aplicar todo lo realizado a algo hecho con nuestras propias manos y ver si nuestro programa sabe detectar correctamente lo que escribimos.

Además nos quedaron pendientes los experimentos mencionados en las discusiones, donde notamos que el tiempo de computo en algunos experimentos comenzaba a disminuir luego de haber aumentado constantemente. Para esto nos hubiese sido útil realizar mediciones considerando muchas más dimensiones.

También nos quedó pendiente fue programar un test que nos permita ver a qué dígitos corresponden los desaciertos de nuestro programa. De este modo podríamos haber sacado conclusiones sobre si el programa estaba muy entrenado para ciertos dígitos y no para otros, o si la partición de nuestra base de datos de entrenamiento excluía algún dígito en particular.

Por último, nos hubiese gustado participar de la competencia de Kaggle para así poder ver como funciona nuestro programa frente a imágenes totalmente ajenas a las de nuestra base de entrenamiento.



## Parte IV

# Apéndices

## 8. Apéndice A: código fuente relevante

### 8.1. k vecinos más cercanos (kNN)

```
vector<int> Knn(matriz& ImagenesEntrenamiento, matriz& ImagenesTest, int k, int alfaOgamma, int metodo)
{
    COUT << "REALIZANDO Knn " << endl;
    COUT << endl;

    vector<pair<int,double> > distancias;
    vector<pair<int,double> > k_vecinos;
    vector<int> respuestas;
    int f = 0;
    int i, j, distanciaCoordendas;
    double distanciaImagen;

    alfaOgamma = alfaOgamma + 1;

    respuestas.resize(ImagenesTest.size());
    distancias.resize(ImagenesEntrenamiento.size());
    //cout << "tamaño imagenes entrenamiento: " << ImagenesEntrenamiento.size() << endl;

    if(metodo == 0)
    {
        alfaOgamma = ImagenesEntrenamiento[0].size();
    }

    while(f < ImagenesTest.size())
    {
        i = 0;

        while(i < ImagenesEntrenamiento.size())
        {
            j = 1;
            distanciaImagen = 0;
            distanciaCoordendas = 0;

            while(j < alfaOgamma)
            {
                //cout << "i vale: " << i << " , j vale: " << j << " , f vale: " << f << endl;
                distanciaCoordendas = distanciaCoordendas + ((ImagenesEntrenamiento[i][j] - ImagenesTest[f][j])*(ImagenesEntrenamiento[i][j] - ImagenesTest[f][j]));
                j++;
            }

            distanciaImagen = sqrt(distanciaCoordendas);
            distancias[i] = (make_pair(ImagenesEntrenamiento[i][0],distanciaImagen));
            i++;
        }

        //cout << "dimension de vector distancias: " << distancias.size() << endl;

        k_vecinos = ordenarPrimeraskDistancias(distancias, k);
        //mostrarVectorOrdenado(k_vecinos);
        respuestas[f] = vecinoGanador(k_vecinos, f);
        //cout << "respuesta[" << f << "] = " << respuestas[f] << endl;

        //cout << "respuesta: " << respuestas[f] << endl;

        f++;
    }

    //mostrarVector(respuestas);

    return respuestas;
}
```

## 8.2. Método de la potencia (power iteration)

```
vector<double> metodoDeLaPotencia(matriz& matrizCovarianzas, int alfa, matriz& autovectoresTraspuestos) // devuelve u
{
    COUT << "CALCULANDO AUTOVALORES Y AUTOVECTORES" << endl;
    COUT << endl;

    vector<double> autovalores, x, K_MasUno; // X en la iteracion k + 1. Una vez que se tiene X = autovector, al mult
    srand(time(NULL)); // para que los numeros random no sean siempre los mismos. Con esto van a depender de la hora
    int k;
    double y, z;
    int i = 0;
    int infinito = 100; // ACA HAY QUE TESTEAR CON QUE NUMERO ES SUFICIENTE // capaz le pondría "pseudoinfinito" en v

    autovalores.resize(alfa);
    autovectoresTraspuestos.resize(alfa);
    x.resize(matrizCovarianzas.size());

    while(i < alfa)
    {
        for(int u = 0; u < x.size(); u++) //GENERAR UN VECTOR X DE TAMAÑO MATRIZCOVARIANZAS.SIZE() Y VALORES RANDOM
        {
            x[u] = 0 + rand();
        }

        x = normalizoX(x);

        autovectoresTraspuestos[i].resize(x.size());
        autovalores[i] = 0;
        k = 0;

        while(k < infinito)
        {
            x = matrizPorVector(matrizCovarianzas, x);
            x = normalizoX(x);
            k++;
        }

        K_MasUno = matrizPorVector(matrizCovarianzas,x);

        autovalores[i] = norma(K_MasUno);
        autovectoresTraspuestos[i] = x;
        if(alfa > 1)matrizCovarianzas = deflacion(matrizCovarianzas, autovalores[i], autovectoresTraspuestos[i]);
        i++;
    }

    return autovalores;
}
```

### 8.2.1. Deflación

```
matriz deflacion(matriz& matrizCovarianzas, double& autovalor, vector<double>& autovector)
{
    int i = 0;
    int j;

    while(i < matrizCovarianzas.size())
    {
        j = 0;

        while(j < matrizCovarianzas.size())
        {
            matrizCovarianzas[i][j] = matrizCovarianzas[i][j] - autovalor * autovector[i] * autovector[j];
            j++;
        }

        i++;
    }

    return matrizCovarianzas;
}

vector<double> normalizoX(vector<double>& x)
{
    int i = 0;
    double norm;

    norm = norma(x);

    while(i < x.size())
    {
        x[i] = x[i]/norm;
        i++;
    }

    return x;
}
```

### 8.3. Análisis de componentes principales (PCA)

```

if(metodo == 1) {
    met = "PCA + Knn";
    start = std::clock();
    //} PARA HACER LOS TESTS PONGO ESTA LLAVE ABAJO ASI SI SE HACE EL METODO DOS NO HACE PCA Y TARDA MENOS
    //HAY QUE VOLVERLA A PONER PARA QUE IMPRIMA LOS VECTORES

    matrizCovarianzas = matrizCovarianza(ImagenesEntrenamiento, media);
    autovalores = metodoDeLaPotencia(matrizCovarianzas, alfa, autovectoresTraspuestos);
    //mostrarVector(autovalores);

    COUT << "REALIZANDO TRANSFORMACION CARACTERISTICA" << endl << endl;

    centrar(ImageesEntrenamiento, media, ImageesEntrenamiento.size());

    int j = 0;
    while(j < ImageesEntrenamiento.size()) {
        ImageesEntrenamiento[j] = transformacionCaracteristica(ImageesEntrenamiento[j], autovectoresTraspuestos);
        j++;
    }

    centrar(ImageesTest, media, ImageesEntrenamiento.size());

    j = 0;
    while(j < ImageesTest.size()) {
        ImageesTest[j] = transformacionCaracteristica(ImageesTest[j], autovectoresTraspuestos);
        j++;
    }

    resultados = Knn(ImageesEntrenamiento, ImageesTest, k, alfa, metodo);

```

### 8.4. Análisis discriminante de cuadrados mínimos parciales (PLS-DA)

```

vector<double> PLS_DA(matriz& ImageesEntrenamiento, vector<double>& media, int Gamma, matriz& autovectoresTraspuestos)
{
    matriz matrizY, matrizX, matriz_Mi, aux2;
    vector<double> Ti, aux, autovalores;
    autovalores.resize(Gamma);
    autovectoresTraspuestos.resize(Gamma);

    media = calculoVectorMedias(ImageesEntrenamiento);

    matrizY = generarMatrizPreY(ImageesEntrenamiento);
    matrizY = armarMatrizY(matrizY, media, ImageesEntrenamiento.size());
    matrizX = armarMatrizX(ImageesEntrenamiento, media, ImageesEntrenamiento.size());

    int i = 0;

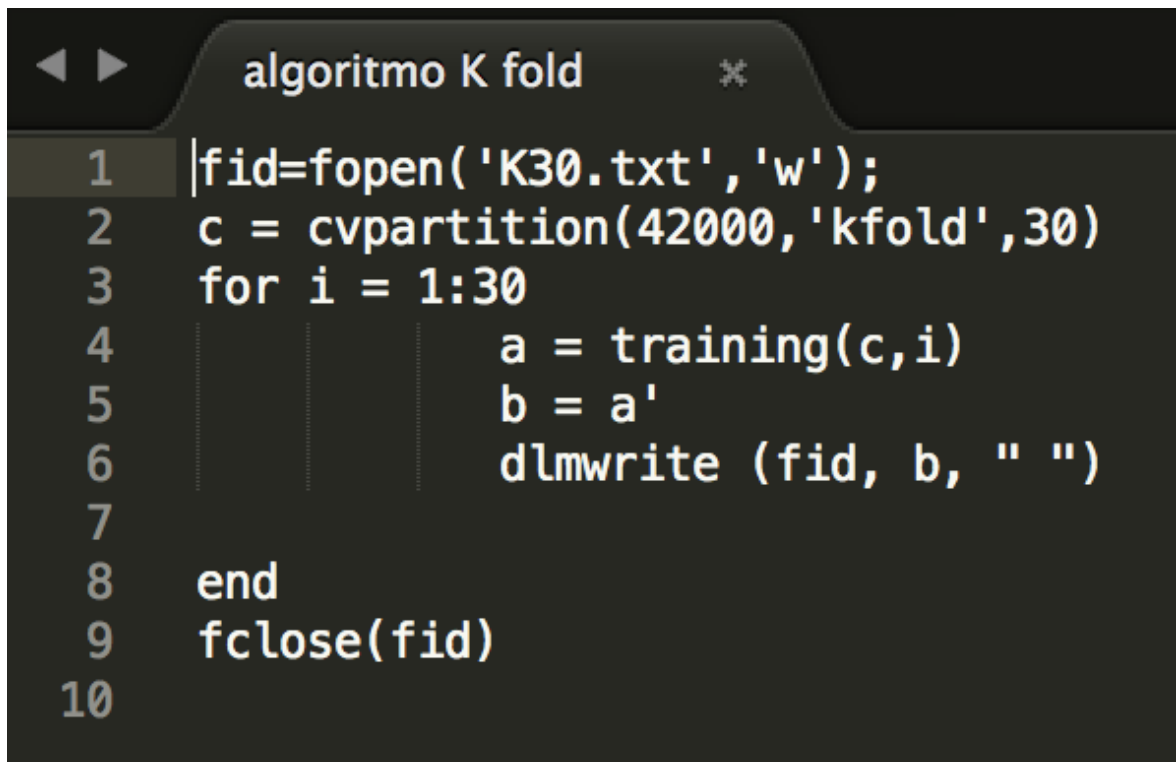
    while(i < Gamma)
    {
        COUT << "GENERANDO MATRIZ Mi: " << i + 1 << " DE " << Gamma << endl << endl;

        generarMatriz_Mi(matrizX, matrizY, matriz_Mi);
        aux = metodoDeLaPotencia(matriz_Mi, 1, aux2);
        Ti = matrizPorVector(matrizX, aux2[0]);
        normalizoX(Ti);
        matrizX = actualizoMatriz(matrizX, Ti);
        matrizY = actualizoMatriz(matrizY, Ti);
        autovalores[i] = aux[0];
        autovectoresTraspuestos[i] = aux2[0];
        i++;
    }

    return autovalores;
}

```

## 8.5. Validación cruzada k-plegada (k-fold x-validation)



```
1 fid=fopen('K30.txt','w');
2 c = cvpartition(42000,'kfold',30)
3 for i = 1:30
4     a = training(c,i)
5     b = a'
6     dlmwrite(fid, b, " ")
7
8 end
9 fclose(fid)
10
```

## 9. Apéndice B

### 9.1. Probabilidad de obtener 10 dígitos de cada clase tomando 100 al azar del conjunto de entrenamiento

Para calcular esta probabilidad, hay que contar cuántas imágenes de cada clase hay en las 42.000 que tenemos en nuestro conjunto de entrenamiento.

Si hay aproximadamente la misma cantidad de imágenes por clase, las probabilidades deberían ser lo suficientemente altas como para poder suponerlo cierto para un caso general.

## Parte V

# Referencias

### 9.2. The C++ Programming Language

[Cpp] <http://www.stroustrup.com/4th.html>

### 9.3. CPlusPlus.com - ctime reference

[CppType] <http://www.cplusplus.com/reference/ctime/clock/>

### 9.4. Documentación de LibreOffice Calc

[LOCalc] [https://help.libreoffice.org/Chart/Charts\\_in/es](https://help.libreoffice.org/Chart/Charts_in/es)

### 9.5. Mathworks.com cvpartition help

[Matlab] <https://www.mathworks.com/help/stats/cvpartition.html>

### 9.6. gentoo.org wiki GCC Optimization -O

[GCC-O] [https://wiki.gentoo.org/wiki/GCC\\_optimization#-O](https://wiki.gentoo.org/wiki/GCC_optimization#-O)

### 9.7. llvm.org clang docs cmd option -ffast-math

[GCC-ffast] <http://clang.llvm.org/docs/UsersManual.html#cmdoption-ffast-math>

### 9.8. Numerical Analysis

[Burden] [https://www.cengagebrain.com.mx/shop/isbn/0538733519?parent\\_category\\_rn=&top\\_category=&urlLangId=-1&errorViewName=ProductDisplayErrorView&categoryId=&urlRequestType=Base&partNumber=0538733519&cid=GB1](https://www.cengagebrain.com.mx/shop/isbn/0538733519?parent_category_rn=&top_category=&urlLangId=-1&errorViewName=ProductDisplayErrorView&categoryId=&urlRequestType=Base&partNumber=0538733519&cid=GB1)

### 9.9. Apuntes de Métodos Numéricos en CubaWiki.com.ar

[Apunte] [http://www.cubawiki.com.ar/images/4/45/Metnum\\_apunte.pdf](http://www.cubawiki.com.ar/images/4/45/Metnum_apunte.pdf)

### 9.10. Apunte de SSH de Marco Vanotti

[SSH] <https://gist.github.com/mvanotti/11058799>

### 9.11. Ayuda de Gitlab en la instancia del DC para configurar acceso con par de crypto-llaves

[GitSSH] <https://git.exactas.uba.ar/help/ssh/README>